fx Documentation

Release 0.2

Philip Xu

November 20, 2012

CONTENTS

Contents:

CHAPTER

ONE

FX - A FUNCTIONAL PROGRAMMING APPROACH

```
 ___
|  _|_ _
|  _|_'_|
|_| |_,_|igher-order function coding.
```

1.1 Introduction

TL;DR - YAGNI.

Inspired by Haskell's rich set of operators, this is an approach to functional programming with operators in Python.

> "It's fun... It's insane... It's insanely fun."
>
> —John Doe

1.1.1 Features

- Currying functions with `<<`, `&`
- Piping output of functions with `|`
- Composing functions with `**`
- Flipping order of arguments of function with `~`
- and more

1.1.2 Examples

```
>>> from fx import f
>>> double_all = f(map) << 2 .__mul__ | list
>>> double_all([1, 2, 3])
[2, 4, 6]
>>> double_all |= f(map) << str | ' '.join
>>> double_all([1, 2, 3])
'2 4 6'
>>> sum_upto = 1 .__add__ | f(range) << 1 | sum
>>> sum_upto(100)
```

```
5050
>>> parse_hex_str = ~f(int) << 16
>>> parse_hex_str('ff')
255
>>> parse_hex_str('c0ffee')
12648430
>>> # project euler problem 1
>>> euler_p1 = f(range) << 1 | f(filter) << (lambda n: n % 3 == 0 or n % 5 == 0) | sum
>>> euler_p1(10)
23
>>> euler_p1(1000)
233168
>>> # project euler problem 20
>>> fact = f(lambda n: 1 if n == 1 else n * fact(n - 1))
>>> euler_p20 = str ** fact | sum ** f(map) << int
>>> euler_p20(10)
27
>>> euler_p20(100)
648
```

1.2 Requirements

- CPython >= 2.6

1.3 Installation

Install from PyPI:

```
pip install fx
```

Install from source, download source package, decompress, then `cd` into source directory, run:

```
make install
```

1.4 License

BSD New, see LICENSE for details.

1.5 Links

Documentation: http://fx.readthedocs.org/

Issue Tracker: https://bitbucket.org/pyx/fx/issues/

Source Package @ PyPI: http://pypi.python.org/pypi/fx/

Mercurial Repoistory @ bitbucket: https://bitbucket.org/pyx/fx/

Git Repoistory @ Github: https://github.com/pyx/fx/

CHAPTER

TWO

TUTORIAL

"Divide et impera."

2.1 Function Creation

`Function` is a function/callable wrapper.

```
>>> from fx import Function
>>> length = Function(len)
>>> length([1, 2, 3])
3
```

Alias `f` can be used instead, for convenience and succinctness.

```
>>> from fx import f
>>> length = f(len)
>>> length(range(5))
5
```

When used on a non-callable object, the newly created `Function` instance will return the same object when called as a function.

```
>>> the_answer = f(42)
>>> the_answer()
42
```

2.2 Function Invocation

Calling method `invoke()` on a `Function` instance will invoke the wrapped function with supplied arguments.

```
>>> minus = f(lambda a, b: a - b)
>>> minus.invoke(5, 2)
3
```

`call()` is an alias to `invoke()`.

```
>>> minus.call(5, 2)
3
```

`Function` overloads `__call__()`, which means, instance of `Function` can be invoked like a normal function.

```
>>> minus(3, 2)
1
```

Keyword arguments are supported as well.

```
>>> minus.invoke(b=2, a=3)
1
>>> minus.call(b=2, a=3)
1
>>> minus(b=2, a=3)
1
```

`Function` overloads `__pos__()`, which implements unary operator +, when used, calls `invoke()` with no arguments.

```
>>> lst = f(list)
>>> lst()
[]
>>> +lst
[]
>>> lst() == +lst
True
```

`value` is a read-only property, when accessed, calls `invoke()` with no arguments.

```
>>> lst(range(3))
[0, 1, 2]
>>> lst.value
[]
```

2.3 Function Application

Partial function application can be done with method `apply()`.

```
>>> five_minus = minus.apply(5)
>>> five_minus(2)
3
>>> five_minus_four = five_minus.apply(4)
>>> five_minus_four.value
1
```

`apply()` accepts arbitrary arguments that wrapped function accepts.

```
>>> five_minus_four = minus.apply(5, 4)
>>> five_minus_four.value
1
>>> five_minus_four = minus.apply(b=4, a=5)
>>> five_minus_four.value
1
```

Operator `<<` is overloaded as function application operator, so the above code can be rewritten with << like this.

```
>>> five_minus = minus << 5
>>> five_minus(2)
3
>>> five_minus_four = five_minus << 4
>>> five_minus_four.value
1
```

```
>>> five_minus_four = minus << 5 << 4
>>> five_minus_four.value
1
```

<<= works as well.

```
>>> m = minus
>>> m <<= 5
>>> m <<= 4
>>> m()
1
>>> m.value
1
```

Operator `&` is overloaded as function application operator, too.

```
>>> five_minus = minus & 5
>>> five_minus(2)
3
>>> five_minus_four = five_minus & 4
>>> five_minus_four.value
1
>>> five_minus_four = minus & 5 & 4
>>> five_minus_four.value
1
>>> m = minus
>>> m &= 5
>>> m &= 4
>>> m()
1
>>> m.value
1
```

Why do we need two different operators doing seemingly the same thing? It is because they have different precedence, and that helps.

Consider this scenario, we want to do something to each element in a sequence, one way to do it is using `map` maps a function over this sequence.

```
>>> seq = [1, 3, 5, 7, 9]
>>> list(map(str, seq))
['1', '3', '5', '7', '9']
```

With partial function application, even functions require more than one arguments can be used to map over a single sequence, for example, we can double every element in this way.

```
>>> mul = f(lambda a, b: a * b)
>>> double = mul << 2
>>> list(map(double, seq))
[2, 6, 10, 14, 18]
```

Instead of hard-coding `seq` here, we can use partial function application technique again, creating a function that can be re-used over and over again.

```
>>> double_all = f(map) << double
>>> list(double_all(seq))
[2, 6, 10, 14, 18]
>>> list(double_all(range(5)))
[0, 2, 4, 6, 8]
```

```
>>> list(double_all('Hello'))
['HH', 'ee', 'll', 'll', 'oo']
```

If we don't need all these intermediate functions, `double_all` can be coded in one line.

```
>>> double_all = f(map) << (f(lambda a, b: a * b) << 2)
>>> list(double_all(seq))
[2, 6, 10, 14, 18]
```

This is where operator `&` comes in handy, by using both function application operators, we can eliminate some parentheses.

```
>>> double_all = f(map) & f(lambda a, b: a * b) << 2
>>> list(double_all(seq))
[2, 6, 10, 14, 18]
```

2.4 Function Composition

In the above example, we have to wrap the result of `map` with a list constructor `list` just to make sure the result will be the same in Python 2.x and Python 3.x, because this is one place where Python 2.x and Python 3.x differ.

```
>>> double_all = f(map) & f(lambda a, b: a * b) << 2
>>> list(double_all(seq))
[2, 6, 10, 14, 18]
```

But typing all these `list(` and `)` is no fun, there is a way to avoid this, we can compose `double_all` with `list`.

```
>>> new_double_all = f(list).compose(double_all)
>>> new_double_all(seq)
[2, 6, 10, 14, 18]
```

`**` is the function composition operator, keep in mind that this operator is **right-associative**, just like the function composition operator `(.)` in Haskell.

```
>>> new_double_all = f(list) ** double_all
>>> new_double_all(seq)
[2, 6, 10, 14, 18]
```

Because both `__pow__()` and `__rpow__()` are implemented, of the two operants of operator `**`, one instance of `Function` will suffice to make it work.

Since `double_all` is already an instance of `Function`, there is no need to wrap `list` in a `Function`.

```
>>> new_double_all = list ** double_all
>>> new_double_all(seq)
[2, 6, 10, 14, 18]
```

With function composition operator `**`, it is possible to refine `double_all` into a one-liner.

```
>>> double_all = list ** f(map) & f(lambda a, b: a * b) << 2
>>> double_all(seq)
[2, 6, 10, 14, 18]
```

Here is a more complicated example.

```
>>> from itertools import count, takewhile as tw
>>> takewhile = f(tw)
>>> select = f(filter)
```

```
>>> odd = lambda n: n % 2
>>> lt_20 = lambda n: n < 20
>>> reverse = lambda s: s[::-1]
>>> + reverse ** list ** (select << odd) ** (takewhile << lt_20) ** count
[19, 17, 15, 13, 11, 9, 7, 5, 3, 1]
```

It's easier to read function composition expressions from right to left:

> From all whole numbers (`count`), we keep taking numbers as long as it is less than 20 (`takewhile << lt_20`), pick all odd numbers from the resulting sequence (`select << odd`), make it into a list (`list`), reverse it (`reverse`), and get the result (`+`).

Warning: Coding in this style is fun, but tend to get hairy soon. *Don't Try This at Home.*

2.5 Function Pipeline

When called with a callable, method `pipe()` will return an instance of `Function`, which when invoked, will pipe the output of current function into that callable. It works very similarly to pipelines in Unix-like systems, thus the name.

To put it simply, piping the output of functions does the same thing as function composition, just in reversed direction, that is, it is evaluated from left to right.

Remember the examples from last section?

```
>>> double_all = f(map) & f(lambda a, b: a * b) << 2
>>> list(double_all(seq))
[2, 6, 10, 14, 18]
>>> new_double_all = f(list).compose(double_all)
>>> new_double_all(seq)
[2, 6, 10, 14, 18]
```

Rewriting `new_double_all` with `pipe()`.

```
>>> new_double_all = double_all.pipe(list)
>>> new_double_all(seq)
[2, 6, 10, 14, 18]
```

Like in a Unix shell, we can use `|` as pipe operator.

```
>>> new_double_all = double_all | list
>>> new_double_all(seq)
[2, 6, 10, 14, 18]
```

Both `__or__()` and `__ror__()` are implemented, so only one of the two operants needs to be an instance of `Function` to make it work.

Let's take a look at the last example of last section, again.

```
>>> from itertools import count, takewhile as tw
>>> takewhile = f(tw)
>>> select = f(filter)
>>> odd = lambda n: n % 2
>>> lt_20 = lambda n: n < 20
>>> reverse = lambda s: s[::-1]
>>> + reverse ** list ** (select << odd) ** (takewhile << lt_20) ** count
[19, 17, 15, 13, 11, 9, 7, 5, 3, 1]
```

The following proves that function pipeline is equivalent to function composition in reversed direction.

```
>>> s = count | (takewhile << lt_20) | (select << odd) | list | reverse
>>> +s
[19, 17, 15, 13, 11, 9, 7, 5, 3, 1]
```

Since operator `<<` has higher precedence than `|`, parentheses can often be omitted.

```
>>> s = count | takewhile << lt_20 | select << odd | list | reverse
>>> +s
[19, 17, 15, 13, 11, 9, 7, 5, 3, 1]
```

2.6 Reversed Function Application

It is sometimes convenient to reverse the expected order of arguments, method `reverse_apply()` helps in this situation.

It returns an instance of `Function`, which takes arguments like the original one, but in reversed order.

```
>>> minus = f(lambda a, b: a - b)
>>> minus(2, 1)
1
>>> subtract = minus.reverse_apply()
>>> subtract(2, 1)
-1
```

You can get the 'flipped' function via read-only property `flip`, too. It's named after Haskell's `flip` function.

```
>>> subtract = minus.flip
>>> subtract(2, 1)
-1
>>> minus.flip(2, 1)
-1
```

Flipping a 'flipped' function again will cancel each other out.

```
>>> minus(2, 1)
1
>>> minus.flip(2, 1)
-1
>>> minus.flip.flip(2, 1)
1
>>> minus.flip.flip.flip(2, 1)
-1
>>> minus.flip.flip.flip.flip(2, 1)
1
```

The flip operator `~` does the same thing.

```
>>> minus(2, 1)
1
>>> (~minus)(2, 1)
-1
>>> (~~minus)(2, 1)
1
>>> (~~~minus)(2, 1)
-1
>>> (~~~~minus)(2, 1)
1
```

2.7 Implicit Function Invocation

Operators `!=` and `==` are overloaded, so that when using these two operators to compare anything to an instance of `Function`, it is equivalent to compare against that instance's `value`.

For example:

```
>>> s = f(range) | list
>>> (s << 3).value
[0, 1, 2]
>>> s << 3 == [0, 1, 2]
True
>>> s << 3 != [0, 1, 2]
False
>>> f(range) << 3 | list == [0, 1, 2]
True
>>> [0, 1, 2] != f(range) << 3 | list
False
>>> f(range) << 3 | list == f(range) << 3 | list
True
>>> f(range) << 3 | list != s << 3
False
```

We can test if a value is in a `Function`'s output, in the form of `value in function`.

For `Function` that its `value` supports membership test operator `in`, (either by supporting the iterator protocol or implementing it's `__contains__` method), membership testing will be delegated to its `value`.

```
>>> one_to_ten = list ** f(range) << 1 << 11
>>> 1 in one_to_ten
True
>>> 10 in one_to_ten
True
>>> 11 in one_to_ten
False
>>> one_to_ten.value
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

For `Function` that its `value` does not support membership test operator `in`, equality is checked instead.

```
>>> the_answer = len ** f(range) << 42
>>> 42 in the_answer
True
>>> 41 in the_answer
False
>>> the_answer()
42
```

Instances of `Function` support iterator protocol.

For `Function` which its `value` is an iterable object, iteration is delegated to that object.

```
>>> one_to_three = f(range) << 1 << 4
>>> for i in one_to_three:
...     print(i)
1
2
3
>>> [i * 2 for i in one_to_three]
[2, 4, 6]
```

For `Function` which its `value` is not an iterable, a 1-tuple with `value` as the only element will be used to iterated over.

```
>>> the_answer = f(42)
>>> the_answer()
42
>>> [i for i in the_answer]
[42]
```

Warning: If you don't know how these features work, as described in this section, they might lead to surprising results and possibly cause more problems than they solve. *You've been warned.*

2.8 Overloaded Operators

In summary, `Function` overloads the following operators.

Operator	Description
`value in f`	Check if value in f's output
`!=`, `==`	Evaluates then compare
`|`	Pipe operator
`&`	Low cohesive application operator
`<<`	High cohesive application operator
`+f`	Low cohesive invoke operator
`~x`	Flip operator
`**`	Function composition operator
`f(arguments...)`	High cohesive invoke operator

2.9 Utility Functions

Package `fx` also provides a couple utility functions, `compose()` and `flip()`.

They work like `Function`'s methods with the same name, except that two functions instead of one are required because there is no `self`.

```
>>> from fx import compose
>>> g = compose(lambda n: -n, abs)
>>> g(-1)
-1

>>> from fx import flip
>>> greater_then = lambda a, b: a > b
>>> greater_then(1, 2)
False
>>> less_then = flip(greater_then)
>>> less_then(1, 2)
True
```

CHAPTER

THREE

API REFERENCE

3.1 Function Wrapper

class `fx.`**`Function`**(*function*)

A function wrapper class.

Implements operators for function composition, arguments flipping, partial application, and more.

```
>>> fmap = Function(map)
>>> double_all = fmap << 2 .__mul__ | list
>>> double_all([1, 2, 3])
[2, 4, 6]

>>> mul = Function(lambda a, b: a * b)
>>> double_all_str = fmap << str ** (mul << 2) | ' '.join
>>> double_all_str([1, 2, 3])
'2 4 6'
```

classmethod `clone`(*function*)

Creates a Function object of the same type as `cls`.

Note: All methods and operators that return an instance of `Function`, return a copy created by `clone()`. That is, `Function` object will not be changed in place by it's methods and operators.

`__init__`(*function*)

Creates a function wrapper object.

```
>>> f = Function(42)
>>> f() == 42
True
>>> f.value == 42
True
>>> f == 42
True
>>> g = Function(lambda a: a + 1)
>>> g(1)
2
>>> g(f())
43
>>> succ = Function(g)
>>> succ(0), succ(1), succ(2)
(1, 2, 3)
>>> times_2 = Function(2 .__mul__)
```

```
>>> [times_2(n) for n in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

invoke(**args*, ***kwargs*)
Invokes the wrapped function with `args` and `kwargs`.

Function invocation:

```
>>> f = Function(int)
>>> f.invoke('2')
2
```

`call` is an alias:

```
>>> f.call('2')
2
```

`value` is a read-only property, when accessed, calls `invoke`:

```
>>> f.value
0
>>> two = f.apply('2')
>>> two.value
2
```

High cohesive invoke operator `()` (a.k.a call operator):

```
>>> f = Function(int)
>>> f('2')
2
>>> f = Function(max)
>>> f(1, 1, 2, 3, 5, 8)
8
```

Low cohesive invoke operator + (positive, unary plus):

```
>>> f = Function(int)
>>> two = f.apply('2')
>>> +two
2
>>> +f.apply('2')
2
```

call(**args*, ***kwargs*)
an alias to `invoke()`.

value
read-only property, with `invoke()` as getter.

__call__(**args*, ***kwargs*)
an alias to `invoke()`, implements high cohesive invoke operator `()`.

__pos__()
an alias to `invoke()`, implements low cohesive invoke operator (unary) +.

compose(*function*)
Creates a Function as composition of `function` with `self`.

Function composition:

```
>>> f = Function(lambda a: -a).compose(abs)
>>> f(-1)
-1
```

Function composition operator `**`:

```
>>> f = Function(lambda a: -a) ** abs
>>> f(-1)
-1
```

`**` works on either side, no need to wrap both sides:

```
>>> f = Function(list) ** map << 1 .__add__
>>> f(range(10))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> f = list ** Function(map) << 1 .__add__
>>> f(range(10))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

__pow__ (*function*)
an alias to `compose()`, implements function composition operator `**`.

__ror__ (*function*)
an alias to `compose()`, implements pipe operator `|` (reflected operands version).

pipe (*function*)
Creates a Function that pipes output into `function` if invoked.

Piping output:

```
>>> f = Function(range).pipe(sum).pipe(int.__neg__)
>>> f(1, 101)
-5050
```

Pipe operator `|`:

```
>>> f = Function(range) | sum | int.__neg__
>>> f(1, 101)
-5050

>>> f = Function(map) << 1 .__add__ | list
>>> f(range(10))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> sum_upto = 1 .__add__ | Function(range) << 1 | sum
>>> sum_upto(100)
5050
```

__or__ (*function*)
an alias to `pipe()`, implements pipe operator `|`.

__rpow__ (*function*)
an alias to `pipe()`, implements function composition operator `**` (reflected operands version).

apply (**args*, ***kwargs*)
Creates a Function with partial function application.

Currying:

```
>>> add = Function(lambda a, b: a + b)
>>> succ = add.apply(1)
```

```
>>> succ(0)
1
```

High cohesive application operator <<:

```
>>> add_1 = add << 1
>>> add_1(2)
3
```

Low cohesive application operator `&`:

```
>>> times = Function(lambda a, b: a * b)
>>> f = Function(map) & times << 2 & range(8) | list
>>> f.value
[0, 2, 4, 6, 8, 10, 12, 14]
```

Partial application:

```
>>> f = Function(max) << 1 << 3 << 5 << 7 << 9 << 2 << 4 << 6 << 8
>>> f.value
9
>>> f(20)
20
>>> f(-1)
9
```

Partial application with keyword argument:

```
>>> int_from_hex = Function(int).apply(base=16)
>>> int_from_hex('0xff')
255
```

__lshift__(*argument*)
an alias to `apply()`, implements high cohesive application operator <<.

__and__(*argument*)
an alias to `apply()`, implements low cohesive application operator `&`.

reverse_apply()
Creates a Function that reversely apply positional arguments.

Reversed positional arguments application:

```
>>> minus = Function(lambda a, b: a - b)
>>> minus(8, 5)
3
>>> subtract = minus.reverse_apply()
>>> subtract(8, 5)
-3
```

`flip` is a read-only property for easier referencing:

```
>>> minus.flip(8, 5)
-3
>>> minus.flip.flip(8, 5)
3
```

Flip operator ~

```
>>> (~minus)(8, 5)
-3
>>> subtract = ~minus
```

```
>>> subtract(8, 5)
-3
```

flip
read-only property, evaluates to the 'flipped' version of current function.

__invert__()
an alias to `reverse_apply()`, implements flip operator ~.

__eq__(*other*)
`self == other`

Compares `self.value` with `other` for equality.

```
>>> f = Function(sum) << [5, 4, 3, 2]
>>> f == 14
True
```

implements operator ==, which means evaluate then check for equality.

__ne__(*other*)
`self != other`

Compares `self.value` with `other` for inequality.

```
>>> f = Function(sum) << [5, 4, 3, 2]
>>> f != 14
False
```

implements operator !=, which means evaluate then check for inequality.

__contains__(*value*)
`value in self`

Returns True if `value` is in `self.value`. If `self.value` is not iterable, equality is checked instead.

```
>>> f = Function(range) << 100
>>> 1 in f
True
>>> -1 in f
False
>>> f = Function(42)
>>> 42 in f
True
>>> 43 in f
False
```

__iter__()
Returns an iterator object.

If `self.value` is iterable, returns `iter(self.value)`, otherwise a 1-tuple with function's output will be created, and iterator of this tuple will be returned, so that calling `iter()` on a Function object will not fail.

```
>>> f = Function(range) << 10
>>> [n for n in f]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> f = Function(42)
>>> [n for n in f]
[42]
```

3.2 Utility Functions

`fx.compose`(*f*, *g*)

Function composition.

```
compose(f, g) -> f .  g
```

```
>>> add_2 = lambda a: a + 2
>>> mul_5 = lambda a: a * 5
>>> mul_5_add_2 = compose(add_2, mul_5)
>>> mul_5_add_2(1)
7
>>> add_2_mul_5 = compose(mul_5, add_2)
>>> add_2_mul_5(1)
15
```

`fx.flip`(*f*)

Creates a function that takes arguments in reverse order.

```
flip(f) -> g
```

```
>>> minus = lambda a, b: a - b
>>> minus(5, 3)
2
>>> subtract = flip(minus)
>>> subtract(5, 3)
-2
>>> list(zip(range(5), range(5, 10), range(10, 15)))
[(0, 5, 10), (1, 6, 11), (2, 7, 12), (3, 8, 13), (4, 9, 14)]
>>> fzip = flip(zip)
>>> list(fzip(range(5), range(5, 10), range(10, 15)))
[(10, 5, 0), (11, 6, 1), (12, 7, 2), (13, 8, 3), (14, 9, 4)]
```

3.3 Alias

class `fx.f`

an alias to `Function` for less typing.

Instead of using `Function()` every time, you can:

```
>>> from fx import f
>>> all_odd = all ** f(map) << (lambda n: n % 2)
>>> all_odd([2, 3, 4])
False
>>> all_odd([7, 5, 9])
True
```

CHAPTER

FOUR

CHANGELOG

- 0.1

  Initial release.

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX