Functionali

Release 0.1.0

Abhinav Omprakash

Dec 01, 2021

CONTENTS

Putting the fun in functional programming

Functional programming is a fundamentally different way of solving problems, and once It clicks, it's pure joy after that. A lot of ideas in this library have been taken from Clojure and Haskell, so the credit goes to those languages. If you find your favorite function missing, or find ways to improve this project, I'd love to hear it.

There are quite a few functions in the library, And they can seem quite overwhelming at first. These functions can be divided into four major categories-

1. Higher order functions. For example `foldr`, `curry`, `flip`,
2. Sequence traversing functions. For example `first`, `rest`, `last`.
3. Sequence transforming functions. For example `cons`, `concat`, `flatten`.
4. predicates. For example `is_even`, `is_prime`, `is_nested`.

CHAPTER

ONE

SEQUENCE TRAVERSING FUNCTIONS

`first`(iterable)	Returns the first item in an iterable or `None` if iterable is empty.
`rest`(iterable)	Returns an iterator of all but the first element in the iterable.
`ffirst`(iterable)	same as `first(first(iterable))` expects a nested iterable, returns None if iterable is empty
`second`(iterable)	Returns the second item in iterable, or `None` if length is less than 2
`third`(iterable)	Returns the third item in iterable, or `None` if length is less than 3
`fourth`(iterable)	Returns the fourth item in iterable, or `None` if length is less than 4
`fifth`(iterable)	Returns the fifth item in iterable, or `None` if length is less than 5
`last`(iterable)	returns the last element in the iterable.
`butlast`(iterable)	returns an iterable of all but the last element in the iterable
`take`(n, iterable)	Returns the first n number of elements in iterable.
`drop`(n, iterable)	Returns All the Elements after the first n number of elements in iterable.
`take_while`(predicate, iterable)	Constructs a iterable list by taking elements from `iterable` while `predicate` is true, Stop taking after the first element falsifies the predicate.
`drop_while`(predicate, iterable)	Drops elements from `iterable` while `predicate` is true, And returns a tuple of the remaining elements in `iterable`.
`split_with`(predicate, iterable)	Equivalent to `(take_while(predicate, iterable), drop_while(predicate, iterable))`
`iter_`(iterable)	Returns appropriate iterator for the given iterable.
`reversed_`(iterable)	Returns appropriate reversed iterator for the given iterable.
`count`(iterable)	counts the number of elements in the iterable, works with map objects, filter objets, and iterators.
`count_`(iterable)	returns a tuple of the number of elements in the iterable and the iterable itself.

functionali.**first**(*iterable: Iterable[Any]*) → Optional[Any]

Returns the first item in an iterable or `None` if iterable is empty. If iterable is a dict, returns a tuple of the First

key-value pair

```
>>> first([1,2,3,4,5])
1
>>> first({1:"a", 2:"b"})
(1, "a")
```

Added in version: 0.1.0

functionali.**rest**(*iterable: Iterable*) → Iterator

Returns an iterator of all but the first element in the iterable. If iterable is empty it returns an empty iterator.

```
>>> list(rest([1,2,3,4,5]))
[2, 3, 4, 5]
```

```
>>> tuple(rest({1:"a", 2:"b", 3:"c"}))
((2,"b"), (3, "c"))
```

```
>>> tuple(rest([]))
()
```

Added in version: 0.1.0

functionali.**ffirst**(*iterable: Iterable[Any]*) → Optional[Any]

same as `first(first(iterable))` expects a nested iterable, returns None if iterable is empty

```
>>> ffirst([[1,2], [3,4], [5,6]])
1
```

Added in version: 0.1.0

functionali.**second**(*iterable: Iterable[Any]*) → Optional[Any]

Returns the second item in iterable, or `None` if length is less than 2

```
>>> second([1,2,3,4,5])
2
```

Added in version: 0.1.0

functionali.**third**(*iterable: Iterable[Any]*) → Optional[Any]

Returns the third item in iterable, or `None` if length is less than 3

```
>>> third([1,2,3,4,5])
3
```

Added in version: 0.1.0

functionali.**fourth**(*iterable: Iterable[Any]*) → Optional[Any]

Returns the fourth item in iterable, or `None` if length is less than 4

```
>>> fourth([1,2,3,4,5])
4
```

Added in version: 0.1.0

functionali.**fifth**(*iterable: Iterable[Any]*) → Optional[Any]

Returns the fifth item in iterable, or `None` if length is less than 5

```
>>> fifth([1,2,3,4,5])
5
```

Added in version: 0.1.0

functionali.**last**(*iterable: Iterable[Any]*) → Optional[Any]
returns the last element in the iterable.

```
>>> last([1,2,3,4])
4
>>> last({1: 'a', 2: 'b', 3: 'c'})
(3, "c")
```

Added in version: 0.1.0

functionali.**butlast**(*iterable: Iterable[Any]*) → Optional[Tuple[Any]]
returns an iterable of all but the last element in the iterable

```
>>> butlast([1, 2, 3])
(1, 2)
```

Added in version: 0.1.0

functionali.**take**(*n: int*, *iterable: Iterable*) → Tuple
Returns the first n number of elements in iterable. Returns an empty tuple if iterable is empty

```
>>> take(3, [1,2,3,4,5])
(1, 2, 3)
>>> take(2, {1: "a", 2: "b", 3: "c"})
((1, "a"), (2, "b"))
```

Added in version: 0.1.0

functionali.**drop**(*n: int*, *iterable: Iterable*) → Tuple
Returns All the Elements after the first n number of elements in iterable. Returns an empty tuple if iterable is empty

```
>>> drop(3, [1,2,3,4,5])
(4,5)
>>> drop(2, {1: "a", 2: "b", 3: "c"})
((3, "c"),)
```

Added in version: 0.1.0

functionali.**take_while**(*predicate: Callable*, *iterable: Iterable*) → Tuple
Constructs a iterable list by taking elements from `iterable` while `predicate` is true, Stop taking after the first element falsifies the predicate.

```
>>> take_while(is_even, [2,4,6,7,8,9,10])
(2,4,6) # Notice that it does not include 8 and 10
```

```
>>> def is_even_dict(d):
        #checks if the key of dict d is even
        return d[0]%2==0
>>> take_while(is_even_dict, {2:"a", 4:"b",5:"c"})
    ((2, "a"), (4, "b"))
```

Added in version: 0.1.0

functionali.**drop_while**(*predicate: Callable*, *iterable: Iterable*) → Tuple
Drops elements from `iterable` while `predicate` is true, And returns a tuple of the remaining elements in `iterable`.

```
>>> drop_while(is_even, [2,4,6,7,8,9,10])
(7,8,9, 10)
```

```
>>> def is_even_dict(d):
        #checks if the key of dict d is even
        return d[0]%2==0
>>> drop_while(is_even_dict, {2:"a", 4:"b",5:"c"})
    ((5, "c"),)
```

Added in version: 0.1.0

functionali.**split_with**(*predicate: Callable*, *iterable: Iterable*) → Tuple[Tuple, Tuple]
Equivalent to `(take_while(predicate, iterable), drop_while(predicate, iterable))`

```
>>> split_with(is_even, [2, 4, 6, 7, 8, 9, 10])
((2, 4, 6), (7, 8, 9, 10))
```

Added in version: 0.1.0

functionali.**iter_**(*iterable: Iterable*) → Iterator
Returns appropriate iterator for the given iterable. This is mainly created because python's `iter` returns an iterable of keys instead of keys and values for `dict`.

```
>>> tuple(iter_({1: "a", 2: "b", 3: "c"}))
((1, "a"),(2, "b"), (3, "c"))
```

Added in version: 0.1.0

functionali.**reversed_**(*iterable: Iterable*) → Iterator
Returns appropriate reversed iterator for the given iterable. This is mainly created because python's `reversed` returns an iterable of keys instead of keys and values for `dict`.

```
>>> tuple(reversed_({1: "a", 2: "b", 3: "c"}))
((3, 'c'), (2, 'b'), (1, 'a'))
```

Added in version: 0.1.0

functionali.**count**(*iterable: Iterable*) → int
counts the number of elements in the iterable, works with map objects, filter objets, and iterators. `count` will consume iterators, use `count_` if you want access to the iterators. Added in version: 0.1.2

```
>>> count(iter([1,2,3]))
3
```

functionali.**count_**(*iterable: Iterable*) → Tuple[int, Iterable]
returns a tuple of the number of elements in the iterable and the iterable itself. This can be used if you wish to find the length of iterators and want to consume the iterators later on. Added in version: 0.1.2 >>> count(iter([1,2,3])) (3,[1,2,3])

CHAPTER

TWO

SEQUENCE TRANSFORMING FUNCTIONS

cons(arg, iterable)	Returns a deque with arg as the first element.
conj(iterable, *args)	Short for conjoin, adds element to the iterable, at the appropriate end.
concat(iterable, *args)	Add items to the end of the iterable.
argmap(functions, args)	Maps the same argument(s) to multiple functions.
argzip(sequence, *args)	Similar to zip, but instead of zipping iterables, It zips an argument(s) with all the values of the iterable.
unzip(sequence)	Opposite of zip.
interleave(*seqs)	Similar to clojure's interleave.
flatten(sequence)	Returns the contents of a nested sequence as a flat sequence.
insert(element, iterable, *[, key])	Inserts `element` right before the first element in the iterable that is greater than `element`
remove(predicate, iterable)	Opposite of filter; Constructs an iterable of elements that falsify the predicate.

functionali.**remove**(*predicate: Callable*, *iterable: Iterable*) → Tuple

Opposite of filter; Constructs an iterable of elements that falsify the predicate.

```
>>> remove(lambda x: x==1, [1,1,9,1,1]
[9]
>>> remove(lambda x: x%2==0, range(10))
[1,3,5,7,9] # filter would return [2,4,6,8]
```

Added in version: 0.1.0

functionali.**cons**(*arg: Any*, *iterable: Iterable*) → collections.deque

Returns a deque with arg as the first element.

Adds to the left of a deque.

```
>>> cons(5, [1,2,3,4])
deque([5, 1, 2, 3, 4])
```

```
>>> cons(3, deque([1,2]))
deque([3, 1, 2])
```

```
>>> cons((3, "c"), {1:"a", 2: "b"})
deque([(3, "c"), (1, "a"), (2, "b")])
```

Added in version: 0.1.0

functionali.**conj**(*iterable: Iterable*, **args: Any*) → Iterable

Short for conjoin, adds element to the iterable, at the appropriate end. Adds to the left of a deque.

```
>>> conj([1,2,3,4],5)
[1, 2, 3, 4, 5]
```

```
>>> conj(deque([1,2]), 3,4)
deque([4, 3, 1, 2])
```

```
>>> conj([1,2,3,4],5,6,7,8)
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> conj([1,2,3,4],[5,6,7])
[1, 2, 3, 4, [5, 6, 7]]
```

```
>>> conj((1,2,3,4),5,6,7)
(1, 2, 3, 4, 5, 6, 7)
```

```
>>> conj(range(10), 11)
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11)
```

```
>>> conj({1:"a", 2:"b"}, {3:"c"})
{1: 'a', 2: 'b', 3: 'c'}
```

Added in version: 0.1.0

functionali.**concat**(*iterable*, **args*)

Add items to the end of the iterable.

```
>>> concat([1,2,3,4],5)
[1, 2, 3, 4, 5]
>>> concat(deque([1,2]), 3,4)
deque([1, 2, 3, 4])
```

Added in version: 0.1.0

functionali.**argmap**(*functions: Iterable[Callable]*, *args: Iterable*) → Generator

Maps the same argument(s) to multiple functions.

```
>>> inc = lambda x:x+1
>>> dec = lambda x:x-1
>>> list(argmap([inc, dec],[1]))
    [2,0]
```

you can even map multiple arguments

```
>>> add = lambda a,b: a+b
>>> sub = lambda a,b:  a-b
>>> list(argmap([add, sub], [2, 1])) # two arguments
    [3, 1]
```

Added in version: 0.1.0

functionali.**argzip**(*sequence: Iterable[Callable]*, **args: Any*) → Generator

Similar to zip, but instead of zipping iterables, It zips an argument(s) with all the values of the iterable. for example.

```
>>> list(argzip([1,2,3,4], "number"))
[(1, 'number'), (2, 'number'), (3, 'number'), (4, 'number')]
>>> list(argzip([1,2,3,4], "number", "int"))
[(1, 'number', 'int'), (2, 'number', 'int'), (3, 'number', 'int'), (4, 'number',
↪'int')]
```

Added in version: 0.1.0

functionali.**unzip**(*sequence: Iterable*) → Tuple[Any]

Opposite of zip. Unzip is shallow.

```
>>> unzip([[1,'a'], [2,'b'], [3,'c']])
((1, 2, 3), ('a', 'b', 'c'))
>>> unzip([ [1,'a','A'], [2, 'b','B'], [3,'c','C'] ])
((1, 2, 3), ('a', 'b', 'c'), ('A', 'B', 'C'))
```

shallow nature of unzip.

```
>>> unzip([ [[1,'num'],['a','str']], [[2,'num'],['b','str']] ])
(([1, 'num'], [2, 'num']), (['a', 'str'], ['b', 'str']))
```

Added in version: 0.1.0

functionali.**interleave**(**seqs: Iterable*) → Tuple

Similar to clojure's interleave. returns a flat sequence with the contents of iterables interleaved.

```
>>> interleave([1,2,3],["a","b","c"])
(1, 'a', 2, 'b', 3, 'c')
>>> interleave([1,2,3],["int","int","int"], ["a","b","c"],["str","str","str" ])
(1, 'int', 'a', 'str', 2, 'int', 'b', 'str', 3, 'int', 'c', 'str')
```

Added in version: 0.1.0

functionali.**flatten**(*sequence: Iterable*) → Tuple

Returns the contents of a nested sequence as a flat sequence. Flatten is recursive.

```
>>> flatten([1,2,[3,[4],5],6,7])
(1, 2, 3, 4, 5, 6, 7)
```

Added in version: 0.1.0

functionali.**insert**(*element: Any*, *iterable: Iterable*, ***, *key: Callable = <function <lambda>>*) → Tuple

Inserts `element` right before the first element in the iterable that is greater than `element`

```
>>> insert(3, [1,2,4,2])
(1,2,3,4,2)
```

```
>>> insert((2, "b"), {1:"a", 3:"c"})
((1, "a"), (2, "b"), (3, "c"))
```

Using the key Parameter

```
>>> Person = namedtuple("Person", ("name", "age"))
>>> person1 = Person("John", 18)
>>> person2 = Person("Abe", 50)
>>> person3 = Person("Cassy", 25)
>>> insert(person3, (person1, person2), key=lambda p:p.age)
    (person1, person3, person2)
>>> insert(person3, (person1, person2), key=lambda p:p.name)
    (person3, person1, person2)
```

Added in version: 0.1.0

CHAPTER

THREE

PREDICATES

identity(x)	Returns its argument as it is.
equals(a[, b])	if only a is passed, a function is returned that returns True when the arg passed to it is equal to a; else returns True when a,``b`` and *args are equal.
is_(a[, b])	if only a is passed, a function is returned that returns True when the arg passed is the same object as a ; else returns True when a,``b`` and *args are.
less_than(a[, b])	if only a is passed, a function is returned that returns True when the arg passed to is less than a; else returns True when a is less than``b`` and *args.
greater_than(a[, b])	if only a is passed, a function is returned that returns True when the arg passed to is greater than a; else returns True when a is greater than``b`` and *args.
less_than_eq(a[, b])	if only a is passed, a function is returned that returns True when the arg less than or equal to a; else returns True when a is less than or equal to b and *args.
greater_than_eq(a[, b])	if only a is passed, a function is returned that returns True when the arg greater than or equal to a; else returns True when a is greater than or equal to b and *args.
complement(expr)	Takes in a predicate or a Boolean expression and returns a negated version of the predicate or expression.
is_even(num)	Returns true when num is even.
is_odd(num)	Returns true when num is odd
is_divisible(divident, divisor)	Returns true if dividend is divisible by divisor.
is_divisible_by(divisor)	Takes a divisor And returns a function (closure) That expects a dividend.
is_numeric(entity)	Return True if entity Is an int, float, or a complex.
is_atom(entity)	Everything that is NOT an iterable(except strings) are considered atoms.
contains(entity, collection)	Checks whether collection contains the given entity.
is_empty(collection)	Returns true if the collection is empty.
is_nested(collection)	returns true if a collection is nested.
all_predicates(*predicates)	Takes a set of predicates and returns a function that takes an entity and checks if it satisfies all the predicates.
some_predicates(*predicates)	Takes a set of predicates and returns a function that takes an entity and checks if it satisfies some of the predicates.

functionali.**identity**(*x*)

Returns its argument as it is.

functionali.**equals**(*a*, *b=None*, **args*)

if only `a` is passed, a function is returned that returns True when the `arg` passed to it is equal to `a`; else returns True when `a`,``b`` and `*args` are equal.

with one argument

```
>>> equals_one = equals(1)
>>> equals_one(1)
True
>>> equals_one(2)
False
```

with two or more arguments

```
>>> equals(1,1,1)
True
>>> equals(1,1,2)
False
```

Added in version: 0.1.0

functionali.**is_**(*a*, *b=None*, **args*)

if only `a` is passed, a function is returned that returns True when the `arg` passed is the same object as `a` ; else returns True when `a`,``b`` and `*args` are.

with one argument

```
>>> d1 = {1,2,3}
>>> d2 = {1,2,3}
>>> is_d1 = is_(d1)
>>> is_d1(d2)
>>> False
>>> d1 == d2
>>> True
```

with two or more arguments

```
>>> is_(d1,d1)
>>> True
>>> is_(d1,d1,d2)
>>> False
```

Added in version: 0.1.0

functionali.**less_than**(*a*, *b=None*, **args*)

if only `a` is passed, a function is returned that returns True when the `arg` passed to is less than `a`; else returns True when `a` is less than``b`` and `*args`.

with one argument

```
>>> less_than_one = less_than(1)
>>> less_than_one(2)
False
>>> less_than_one(0)
True
```

with two or more arguments

```
>>> less_than(1,2)
>>> True
>>> less_than(1,2,3)
True
>>> less_than(1,2,3,1)
False
```

Useful to use with filter

```
>>> list(filter(less_than(5),range(10)))
[0,1,2,3,4]
```

Added in version: 0.1.0

functionali.**greater_than**(*a*, *b=None*, **args*)

if only `a` is passed, a function is returned that returns True when the `arg` passed to is greater than `a`; else returns True when `a` is greater than``b`` and `*args`.

with one argument

```
>>> greater_than_one = greater_than(1)
>>> greater_than_one(2)
True
>>> greater_than_one(0)
False
```

with two or more arguments

```
>>> greater_than(2,1)
>>> True
>>> greater_than(3,2,1)
True
>>> greater_than(3,2,1,3)
False
```

Useful to use with filter

```
>>> list(filter(greater_than(5),range(10)))
[6,7,8,9]
```

Added in version: 0.1.0

functionali.**less_than_eq**(*a*, *b=None*, **args*)

if only `a` is passed, a function is returned that returns True when the `arg` less than or equal to `a`; else returns True when `a` is less than or equal to `b` and `*args`.

with one argument

```
>>> less_than_or_eq_to_one = less_than_eq(1)
>>> less_than_or_eq_to_one(2)
False
>>> less_than_or_eq_to_one(1)
True
```

with two or more arguments

```
>>> less_than_eq(1,2)
>>> True
>>> less_than_eq(1,2,3)
True
>>> less_than_eq(1,2,3,1)
True
```

Useful to use with filter

```
>>> list(filter(less_than_eq(5),range(10)))
[0,1,2,3,4,5]
```

Added in version: 0.1.0

functionali.**greater_than_eq**(*a*, *b=None*, **args*)

if only `a` is passed, a function is returned that returns True when the `arg` greater than or equal to `a`; else returns True when `a` is greater than or equal to `b` and `*args`.

with one argument

```
>>> greater_than_eq_one = greater_than_eq(1)
>>> greater_than_eq_one(2)
True
>>> greater_than_eq_one(1)
True
```

with two or more arguments

```
>>> greater_than_eq(2,1)
>>> True
>>> greater_than_eq(3,2,1)
True
>>> greater_than_eq(3,2,1,3)
True
```

Useful to use with filter

```
>>> list(filter(greater_than_eq(5),range(10)))
[5,6,7,8,9]
```

Added in version: 0.1.0

functionali.**complement**(*expr: Union[bool, Callable[[Any], bool]]*) → Union[bool, Callable[[Any], bool]]

Takes in a predicate or a Boolean expression and returns a negated version of the predicate or expression.

```
>>> complement(True)
>>> False
```

```
>>> def fn(el): # returns the Boolean of el
    return bool(el)
>>> negated_fn = complement(fn)
>>> fn(1)
>>> True
>>> negated_fn(1)
>>> False
```

Added in version: 0.1.0

functionali.**is_even**(*num: int*) → bool
Returns true when num is even.

Added in version: 0.1.0

functionali.**is_odd**(*num: int*) → bool
Returns true when num is odd

Added in version: 0.1.0

functionali.**is_divisible**(*divident: Union[int, float]*, *divisor: Union[int, float]*) → bool
Returns true if dividend is divisible by divisor.

Added in version: 0.1.0

functionali.**is_divisible_by**(*divisor: Union[int, float]*) → Callable[[Union[int, float]], bool]
Takes a `divisor` And returns a function (closure) That expects a dividend. returns true if it passes the divisibility test. for e.g.

```
>>> f = is_divisible_by(5)
>>> f(10)
True
>>> f(7)
False
```

This is particularly useful to use with a filter.

```
>>> list(filter(is_divisible_by(5), [1,2,3,4,5,6,7,8,9,10]))
[5, 10]
```

Suppose you want to filter out numbers that are divisible by 2 or 3

```
>>> list(filter(some_predicates([is_divisible_by(2), is_divisible_by(3)]), range(1,␣
↪10)))
[2, 3, 4, 6, 8, 9, 10]
```

Added in version: 0.1.0

functionali.**is_numeric**(*entity: Any*) → bool
Return True if `entity` Is an `int`, `float`, or a `complex`.

Added in version: 0.1.0

functionali.**is_atom**(*entity: Any*) → bool
Everything that is NOT an iterable(except strings) are considered atoms.

```
>>> is_atom("plain string")
    True
>>> is_atom(1)
    True
>>> is_atom([1, 2])
    False
```

Added in version: 0.1.0

functionali.**contains**(*entity: Any*, *collection: Iterable*) → bool
Checks whether collection contains the given entity. Note, won't automatically convert a tuple of keys and values to a `dict`.

Added in version: 0.1.0

functionali.**is_empty**(*collection: Iterable*) → bool
Returns true if the collection is empty.

Added in version: 0.1.0

functionali.**is_nested**(*collection: Iterable*) → bool
returns true if a collection is nested. Added in version: 0.1.0

functionali.**all_predicates**(**predicates: Callable[[Any], bool]*) → Callable[[Any], bool]
Takes a set of predicates and returns a function that takes an entity and checks if it satisfies all the predicates.

```
>>> even_and_prime = all_predicates(is_even, is_prime)
>>> even_and_prime(2)
True
>>> even_and_prime(4)
False
>>> even_and_prime(3)
False
```

Added in version: 0.1.0

functionali.**some_predicates**(**predicates: Callable[[Any], bool]*) → Callable[[Any], bool]
Takes a set of predicates and returns a function that takes an entity and checks if it satisfies some of the predicates.

```
>>> even_or_prime = some_predicates(is_even, is_prime)
>>> even_or_prime(2)
True
>>> even_and_prime(4)
True
>>> even_and_prime(3)
True
```

Added in version: 0.1.0

CHAPTER

FOUR

HIGHER ORDER FUNCTIONS

`reduce`(fn, iterable[, initial])	Similar to python's reduce, but can be prematurely terminated with `reduced`.
`reduced`(x)	Use with `functionali.reduce` to prematurely terminate `reduce` with the value of `x`.
`flip`(fn)	returns a function that takes in a flipped order of args.
`foldr`(fn, iterable[, initial])	Fold right.
`curry`(fn)	Returns a curried version of the function.
`threadf`(arg, forms)	Thread first, passes `arg` as the first argument to the first function in `forms` and passes the result as the first argument to the second form and so on.
`threadl`(arg, forms)	Thread last, passes `arg` as the last argument to the first function in `forms` and passes the result as the last argument to the second form and so on.
`trampoline`(fn, *args)	takes a function `fn` and calls if with `*args`.
`comp`(*fns)	returns a composed function that takes a variable number of args, and applies it to `fns` passed from right to left.

functionali.**reduce**(*fn*, *iterable*, *initial=None*)

Similar to python's reduce, but can be prematurely terminated with `reduced`. Works with dictionaries too.

Usage:

```
>>> # Reducing over dictionaries.
>>> def inc_value(result, kv_pair):
        k = kv_pair[0]
        v = kv_pair[1]
        return result[k]= v+1
>>> reduce(inc_value, {"a":1,"b":2}, {})
{'a': 2, 'b': 3}
```

```
>>> #premature termination with reduced
>>> def inc_while_odd(result, element):
        if element%2==0:
            return reduced(result)
        else:
            result.append(element+1)
            return result
>>> reduce(inc_while_odd, [1,3,5,6,7,8],[])
[2, 4, 6]
# increments uptil 5 (third element) and prematurely terminates.
```

functionali.**reduced**(*x*)

Use with functionali.reduce to prematurely terminate reduce with the value of x.

Usage:

```
>>> reduce(lambda acc, el: reduced("!"), [1,3,4])
"!"
# reduce is prematurely terminated and returns a value of "!"
```

functionali.**flip**(*fn: Callable*) → Callable

returns a function that takes in a flipped order of args. Usage:

```
>>> f = lambda a,b : a-b
>>> f(1,3)
-2
>>> f(3,1)
2
>>> flipped_f = flip(f)
>>> flipped_f(3,1)
-2
>>> flipped_f(1,3)
2
```

Added in version: 0.1.0

functionali.**foldr**(*fn: Callable*, *iterable: Iterable*, *initial: Optional[Any] = None*) → Any

Fold right. Stack safe implementation

Added in version: 0.1.0

functionali.**curry**(*fn: Callable*) → Callable

Returns a curried version of the function.

```
>>> def fn(arg1, arg2, arg3):  # test function
    return [arg1, arg2, arg3]
>>> curried_fn = curry(fn)
>>> curried_fn(1)(2)(3)
    [1, 2, 3]
```

Added in version: 0.1.0

functionali.**threadf**(*arg: Any*, *forms: Iterable[Union[Callable, Iterable]]*) → Any

Thread first, passes arg as the first argument to the first function in forms and passes the result as the first argument to the second form and so on.

see also threadl.

```
>>> from functionali import identity
>>> from operator import add, sub, mul
>>> threadf(5, [identity])
>>> 5
```

```
>>> threadf(5, [identity, [add, 2]])
>>> 7
```

```
>>> threadf(5, [[sub, 2]])
>>> 3 # threadf(5, [[sub, 2]]) -> sub(5, 2) -> 5-2 -> 3
```

```
>>> # combining multiple functions
>>> threadf(5, [identity, (add, 1), (sub, 1), (mul, 3)])
15
```

functionali.**threadl**(*arg: Any*, *forms: Iterable[Union[Callable, Iterable]]*) → Any

Thread last, passes `arg` as the last argument to the first function in `forms` and passes the result as the last argument to the second form and so on.

see also `threadf`.

```
>>> from functionali import identity
>>> from operator import add, sub, mul
>>> threadl(5, [identity])
>>> 5
```

```
>>> threadl(5, [identity, [add, 2]])
>>> 7
```

```
>>> threadl(5, [[sub, 2]])
>>> -3 # threadl(5, [[sub, 2]]) -> sub(2, 5) -> 2-5 -> -3
```

```
>>> # combining multiple functions
>>> threadl(5, [identity, (add, 1), (sub, 1), (mul, 3)])
-15
```

functionali.**trampoline**(*fn: Callable*, **args: Any*)

takes a function `fn` and calls if with `*args`. if `fn` returns a function, calls the function until a function is not returned i.e. the base case is reached. function `fn` must return a function in its recursive case. Useful for optimizing tail recursive functions or mutual recursions.

```
>>> def fact(x, curr=1, acc=1):
>>>     if curr == x:
>>>         return curr*acc
>>>     else:
>>>         return lambda: fact(x, curr+1, acc*curr)
>>> trampoline(fact, 3) == 6
>>> trampoline(fact, 100000000000) # does not raise RecursionError
```

functionali.**comp**(**fns: Callable*)

returns a composed function that takes a variable number of args, and applies it to `fns` passed from right to left.

Added in version: 0.1.2

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

U