funcy documentation

Release 1.9

Alexander Schepanovski

Sep 13, 2017

Contents

Funcy is designed to be a layer of functional tools over python.

Special topics:

CHAPTER 1

Overview

Start with:

```
pip install funcy
```

Import stuff from funcy to make things happen:

```
from funcy import whatever, you, need
```

Merge collections of same type (works for dicts, sets, lists, tuples, iterators and even strings):

```
merge(coll1, coll2, coll3, ...)
join(colls)
merge_with(sum, dict1, dict2, ...)
```

Walk through collection, creating its transform (like map but preserves type):

```
walk(str.upper, {'a', 'b'})             # {'A', 'B'}
walk(reversed, {'a': 1, 'b': 2})        # {1: 'a', 2: 'b'}
walk_keys(double, {'a': 1, 'b': 2})     # {'aa': 1, 'bb': 2}
walk_values(inc, {'a': 1, 'b': 2})      # {'a': 2, 'b': 3}
```

Select a part of collection:

```
select(even, {1,2,3,10,20})                  # {2,10,20}
select(r'^a', ('a','b','ab','ba'))           # ('a','ab')
select_keys(callable, {str: '', None: None}) # {str: ''}
compact({2, None, 1, 0})                     # {1,2}
```

Manipulate sequences:

```
take(4, iterate(double, 1)) # [1, 2, 4, 8]
first(drop(3, count(10)))   # 13

remove(even, [1, 2, 3])     # [1, 3]
concat([1, 2], [5, 6])      # [1, 2, 5, 6]
```

```
cat(map(range, range(4)))    # [0, 0, 1, 0, 1, 2]
mapcat(range, range(4))      # same
flatten(nested_structure)    # flat_list
distinct('abacbdd')          # list('abcd')

split(odd, range(5))         # ([1, 3], [0, 2, 4])
split_at(2, range(5))        # ([0, 1], [2, 3, 4])
group_by(mod3, range(5))     # {0: [0, 3], 1: [1, 4], 2: [2]}

partition(2, range(5))       # [[0, 1], [2, 3]]
chunks(2, range(5))          # [[0, 1], [2, 3], [4]]
pairwise(range(5))           # iter: [0, 1], [1, 2], ...
```

And functions:

```
partial(add, 1)              # inc
curry(add)(1)(2)             # 3
compose(inc, double)(10)     # 21
complement(even)             # odd
all_fn(isa(int), even)       # is_even_int

one_third = rpartial(operator.div, 3.0)
has_suffix = rcurry(str.endswith)
```

Create decorators easily:

```
@decorator
def log(call):
    print call._func.__name__, call._args
    return call()
```

Abstract control flow:

```
walk_values(silent(int), {'a': '1', 'b': 'no'})
# => {'a': 1, 'b': None}

@once
def initialize():
    "..."

with suppress(OSError):
    os.remove('some.file')

@ignore(ErrorRateExceeded)
@limit_error_rate(fails=5, timeout=60)
@retry(tries=2, errors=(HttpError, ServiceDown))
def some_unreliable_action(...):
    "..."

class MyUser(AbstractBaseUser):
    @cached_property
    def public_phones(self):
        return self.phones.filter(public=True)
```

Ease debugging:

```
squares = {tap(x, 'x'): tap(x * x, 'x^2') for x in [3, 4]}
# x: 3
```

```
# x^2: 9
# ...

@print_exits
def some_func(...):
    "..."

@log_calls(log.info, errors=False)
@log_errors(log.exception)
def some_suspicious_function(...):
    "..."

with print_durations('Creating models'):
    Model.objects.create(...)
    # ...
# 10.2 ms in Creating models
```

CHAPTER 2

Cheatsheet

Hover over function to get its description.

Sequences

Create	*count() cycle() repeat() repeatedly() iterate() re_all() re_iter()*
Access	*first() second() last() nth() some() take()*
Slice	*take() drop() rest() butlast() takewhile() dropwhile() split_at() split_by()*
Transform	*map() mapcat() keep() pluck() pluck_attr() invoke()*
Filter	*filter() remove() keep() distinct() where() without()*
Join	*cat() concat() flatten() mapcat() interleave() interpose()*
Partition	*chunks() partition() partition_by() split_at() split_by()*
Group	*split() count_by() count_reps() group_by() group_by_keys() group_values()*
Aggregate	*ilen() reductions() sums() all() any() none() one() count_by() count_reps()*
Iterate	*pairwise() with_prev() with_next() izip_values() izip_dicts() tree_leaves() tree_nodes()*

Collections

Join	*merge() merge_with() join() join_with()*
Transform	*walk() walk_keys() walk_values()*
Filter	*select() select_keys() select_values() compact()*
Dicts *	*flip() zipdict() pluck() where() itervalues() iteritems() izip_values() izip_dicts() project() omit()*
Misc	*empty() get_in() set_in() update_in()*

Functions

Create	*identity() constantly() func_partial() partial() rpartial() iffy() caller() re_finder() re_tester()*
Transform	*complement() iffy() autocurry() curry() rcurry()*
Combine	*compose() rcompose() juxt() all_fn() any_fn() none_fn() one_fn() some_fn()*

Other topics

Content tests	*all() any() none() one() is_distinct()*
Type tests	*isa() is_iter() is_list() is_tuple() is_set() is_mapping() is_seq() is_seqcoll() is_seqcont() iterable()*
Decorators	*decorator wraps unwrap autocurry()*
Control flow	*once() once_per() once_per_args() collecting() joining() post_processing()*
Error handling	*retry() silent() ignore() suppress() limit_error_rate() fallback() raiser()*
Debugging	*tap() log_calls() log_enters() log_exits() log_errors() log_durations() log_iter_durations()*
Caching	*memoize() cache() cached_property() make_lookuper() silent_lookuper()*
Regexes	*re_find() re_test() re_all() re_iter() re_finder() re_tester()*
Strings	*cut_prefix() cut_suffix() str_join()*
Objects	*cached_property() monkey() invoke() pluck_attr() namespace*
Primitives	*isnone() notnone() inc() dec() even() odd()*

CHAPTER 3

Extended function semantics

Many of funcy functions expecting predicate or mapping function as an argument can take something uncallable instead of it with semantics described in this table:

f passed	Function	Predicate
`None`	*identity*	bool
string	*re_finder(f)*	*re_tester(f)*
int or slice	`itemgetter(f)`	`itemgetter(f)`
mapping	`lambda x: f[x]`	`lambda x: f[x]`
set	`lambda x: x in f`	`lambda x: x in f`

Supporting functions

Here is a full list of functions supporting extended function semantics:

Group	Functions
Sequence transformation	*map()*, *imap()*, *keep()*, *ikeep()*, *mapcat()*, *imapcat()*
Sequence filtering	*filter()*, *ifilter()*, *remove()*, *iremove()*, *distinct()*, *idistinct()*
Sequence splitting	*dropwhile()*, *takewhile()*, *split()*, *split_by()*
Sequence chunking	*group_by()*, *count_by()*, *partition_by()*, *ipartition_by()*
Collection transformation	*walk()*, *walk_keys()*, *walk_values()*
Collection filtering	*select()*, *select_keys()*, *select_values()*
Content tests	*all()*, *any()*, *none()*, *one()*, *some()*, *is_distinct()*
Function logic	*all_fn()*, *any_fn()*, *none_fn()*, *one_fn()*, *some_fn()*
Function tools	*compose()*, *rcompose()*, *complement()*, *juxt()*, *ijuxt()*

CHAPTER 4

Python 3 support

Funcy works with python 3 as of version 0.9. However, it has slightly different interface. It follows python 3 convention of "iterator by default" for utilities like *map()*, *filter()* and such. When funcy has two versions of utility (list and iterator) they are named like *keep()* and *ikeep()* in python 2 and `lkeep()` and *keep()* in python 3. You can look up a full table of differently named functions below.

Writing cross-python code

You can do that two ways: writing python 2 code that works in python 3 or vice versa. You can import python 2 or 3 style functions from `funcy.py2` or `funcy.py3`:

```
from funcy.py2 import whatever, you, need

# write python 2 style code here
```

```
from funcy.py3 import whatever, you, need

# write python 3 style code here
```

You can even import *map()*, *imap()*, *filter()*, *ifilter()*, `zip()` and `izip()`.

Full table of python dependent function names

Python 2 / list	Python 2 / iterator	Python 3 / list	Python 3 / iterator
map()	*imap()*	lmap()	*map()*
filter()	*ifilter()*	lfilter()	*filter()*
zip()	izip()	lzip()	zip()
remove()	*iremove()*	lremove()	*remove()*
keep()	*ikeep()*	lkeep()	*keep()*
without()	*iwithout()*	lwithout()	*without()*
concat()	*iconcat()*	lconcat()	*concat()*
cat()	*icat()*	lcat()	*cat()*
flatten()	*iflatten()*	lflatten()	*flatten()*
mapcat()	*imapcat()*	lmapcat()	*mapcat()*
distinct()	*idistinct()*	ldistinct()	*distinct()*
split()	*isplit()*	lsplit()	*split()*
split_at()	*isplit_at()*	lsplit_at()	*split_at()*
split_by()	*isplit_by()*	lsplit_by()	*split_by()*
partition()	*ipartition()*	lpartition()	*partition()*
chunks()	*ichunks()*	lchunks()	*chunks()*
partition_by()	*ipartition_by()*	lpartition_by()	*partition_by()*
reductions()	*ireductions()*	lreductions()	*reductions()*
sums()	*isums()*	lsums()	*sums()*
juxt()	*ijuxt()*	ljuxt()	*juxt()*
where()	*iwhere()*	lwhere()	*where()*
pluck()	*ipluck()*	lpluck()	*pluck()*
pluck_attr()	*ipluck_attr()*	lpuck_attr()	*pluck_attr()*
invoke()	*iinvoke()*	linvoke()	*invoke()*
-	*izip_values()*	-	zip_values()
-	*izip_dicts()*	-	zip_dicts()

Contents:

CHAPTER 5

Sequences

This functions are aimed at manipulating finite and infinite sequences of values. Some functions have two flavors: one returning list and other returning possibly infinite iterator, the latter ones follow convention of prepending `i` before list-returning function name.

When working with sequences, see also `itertools` standard module. Funcy reexports and aliases some functions from it.

Generate

repeat(*item*[, *n*])

Makes an iterator yielding `item` for `n` times or indefinitely if `n` is omitted. *`repeat()`* simply repeat given value, when you need to reevaluate something repeatedly use *`repeatedly()`* instead.

When you just need a length `n` list or tuple of `item` you can use:

```
[item] * n
# or
(item,) * n
```

count(*start=0*, *step=1*)

Makes infinite iterator of values: `start, start + step, start + 2*step, ...`.

Could be used to generate sequence:

```
imap(lambda x: x ** 2, count(1))
# -> 1, 4, 9, 16, ...
```

Or annotate sequence using `zip()` or `izip()`:

```
zip(count(), 'abcd')
# -> [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]

# print code with BASIC-style numbered lines
```

```
for line in izip(count(10, 10), code.splitlines()):
    print '%d %s' % line
```

See also `enumerate()` and original `itertools.count()` documentation.

cycle (*seq*)
Cycles passed `seq` indefinitely returning its elements one by one.

Useful when you need to cyclically decorate some sequence:

```
for n, parity in izip(count(), cycle(['even', 'odd'])):
    print '%d is %s' % (n, parity)
```

repeatedly (*f*[, *n*])
Takes a function of no args, presumably with side effects, and returns an infinite (or length `n` if supplied) iterator of calls to it.

For example, this call can be used to generate 10 random numbers:

```
repeatedly(random.random, 10)
```

Or one can create a length `n` list of freshly-created objects of same type:

```
repeatedly(list, n)
```

iterate (*f*, *x*)
Returns an infinite iterator of `x, f(x), f(f(x)), ...` etc.

Most common use is to generate some recursive sequence:

```
iterate(inc, 5)
# -> 5, 6, 7, 8, 9, ...

iterate(lambda x: x * 2, 1)
# -> 1, 2, 4, 8, 16, ...

step = lambda ((a, b)): (b, a + b)
imap(first, iterate(step, (0, 1)))
# -> 0, 1, 1, 2, 3, 5, 8, ... (Fibonacci sequence)
```

Manipulate

This section provides some robust tools for sequence slicing. Consider Slicings or `itertools.islice()` for more generic cases.

take (*n*, *seq*)
Returns a list of the first `n` items in the sequence, or all items if there are fewer than `n`.

```
take(3, [2, 3, 4, 5]) # [2, 3, 4]
take(3, count(5))     # [5, 6, 7]
take(3, 'ab')         # ['a', 'b']
```

drop (*n*, *seq*)
Skips first `n` items in the sequence, returning iterator yielding rest of its items.

```
drop(3, [2, 3, 4, 5]) # iter([5])
drop(3, count(5))     # count(8)
drop(3, 'ab')         # empty iterator
```

first(*seq*)
Returns the first item in the sequence. Returns `None` if the sequence is empty. Typical usage is choosing first of some generated variants:

```
# Get a text message of first failed validation rule
fail = first(rule.text for rule in rules if not rule.test(instance))

# Use simple pattern matching to construct form field widget
TYPE_TO_WIDGET = (
    [lambda f: f.choices,           lambda f: Select(choices=f.choices)],
    [lambda f: f.type == 'int',     lambda f: TextInput(coerce=int)],
    [lambda f: f.type == 'string',  lambda f: TextInput()],
    [lambda f: f.type == 'text',    lambda f: Textarea()],
    [lambda f: f.type == 'boolean', lambda f: Checkbox(f.label)],
)
return first(do(field) for cond, do in TYPE_TO_WIDGET if cond(field))
```

Other common use case is passing to *map()* or `imap()`. See last example in *iterate()* for such example.

second(*seq*)
Returns the second item in given sequence. Returns `None` if there are less than two items in it.

Could come in handy with sequences of pairs, e.g. `dict.items()`. Following code extract values of a dict sorted by keys:

```
map(second, sorted(some_dict.items()))
```

And this line constructs an ordered by value dict from a plain one:

```
OrderedDict(sorted(plain_dict.items(), key=second))
```

nth(*n*, *seq*)
Returns nth item in sequence or `None` if no one exists. Items are counted from 0, so it's like indexed access but works for iterators. E.g. here is how one can get 6th line of *some_file*:

```
nth(5, repeatedly(open('some_file').readline))
```

last(*seq*)
Returns the last item in the sequence. Returns `None` if the sequence is empty. Tries to be efficient when sequence supports indexed or reversed access and fallbacks to iterating over it if not.

rest(*seq*)
Skips first item in the sequence, returning iterator starting just after it. A shortcut for *drop(1, seq)*.

butlast(*seq*)
Returns an iterator of all elements of the sequence but last.

ilen(*seq*)
Calculates length of iterator. Will consume it or hang up if it's infinite.

Especially useful in conjunction with filtering or slicing functions, for example, this way one can find common start length of two strings:

```
ilen(takewhile(lambda (x, y): x == y, zip(s1, s2)))
```

Unite

concat(**seqs*)
iconcat(**seqs*)
Concats several sequences into one. *iconcat()* returns an iterator yielding concatenation.

iconcat() is an alias for itertools.chain().

cat(*seqs*)
icat(*seqs*)
Concatenates passed sequences. Useful when dealing with sequence of sequences, see *concat()* or *iconcat()* to join just a few sequences.

Flattening of various nested sequences is most common use:

```
# Flatten two level deep list
cat(list_of_lists)

# Get a flat html of errors of a form
errors = icat(inline.errors() for inline in form)
error_text = '<br>'.join(errors)

# Brace expansion on product of sums
# (a + b)(t + pq)x == atx + apqx + btx + bpqx
terms = [['a', 'b'], ['t', 'pq'], ['x']]
map(cat, product(*terms))
# [list('atx'), list('apqx'), list('btx'), list('bpqx')]
```

icat() is an alias for itertools.chain.from_iterable().

flatten(*seq*, *follow=is_seqcont*)
iflatten(*seq*, *follow=is_seqcont*)
Flattens arbitrary nested sequence of values and other sequences. follow argument determines whether to unpack each item. By default it dives into lists, tuples and iterators, see *is_seqcont()* for further explanation.

See also *cat()* or *icat()* if you need to flatten strictly two-level sequence of sequences.

tree_leaves(*root*, *follow=is_seqcont*, *children=iter*)
itree_leaves(*root*, *follow=is_seqcont*, *children=iter*)
A way to list or iterate over all the tree leaves. E.g. this is how you can list all descendants of a class:

```
tree_leaves(Base, children=type.__subclasses__, follow=type.__subclasses__)
```

tree_nodes(*root*, *follow=is_seqcont*, *children=iter*)
itree_nodes(*root*, *follow=is_seqcont*, *children=iter*)
A way to list or iterate over all the tree nodes. E.g. this is how you can list all classes in hierarchy:

```
tree_nodes(Base, children=type.__subclasses__, follow=type.__subclasses__)
```

interleave(**seqs*)
Returns an iterator yielding first item in each sequence, then second and so on until some sequence ends. Numbers of items taken from all sequences are always equal.

interpose(*sep*, *seq*)
Returns an iterator yielding elements of seq separated by sep.

Helpful when str.join() is not good enough. This code is a part of translator working with operation node:

```
def visit_BoolOp(self, node):
    # ... do generic visit
    node.code = mapcat(translate, interpose(node.op, node.values))
```

Transform and filter

Most of functions in this section support *Extended function semantics*. Among other things it allows to rewrite examples using *re_tester()* and *re_finder()* tighter.

map (*f*, *seq*)
imap (*f*, *seq*)
Extended versions of map() and imap().

filter (*pred*, *seq*)
ifilter (*pred*, *seq*)
Extended versions of filter() and ifilter().

remove (*pred*, *seq*)
iremove (*pred*, *seq*)
Return a list or an iterator of items of seq that result in false when passed to pred. The results of this functions complement results of standard *filter()* and ifilter().

A handy use is passing *re_tester()* result as pred. For example, this code removes any whitespace-only lines from list:

```
remove(re_tester('^\s+$'), lines)
```

Note, you can rewrite it shorter using *Extended function semantics*:

```
remove('^\s+$', lines)
```

keep (*[f]*, *seq*)
ikeep (*[f]*, *seq*)
Maps seq with given function and then filters out falsy elements. Simply filters seq when f is absent. In fact these functions are just handy shortcuts:

```
keep(f, seq)  == filter(bool, map(f, seq))
keep(seq)     == filter(bool, seq)

ikeep(f, seq) == ifilter(bool, imap(f, seq))
ikeep(seq)    == ifilter(bool, seq)
```

Natural use case for *keep()* is data extraction or recognition that could eventually fail:

```
# Extract numbers from words
keep(re_finder(r'\d+'), words)

# Recognize as many colors by name as possible
keep(COLOR_BY_NAME.get, color_names)
```

An iterator version can be useful when you don't need or not sure you need the whole sequence. For example, you can use *first()* - *ikeep()* combo to find out first match:

```
first(ikeep(COLOR_BY_NAME.get, color_name_candidates))
```

Alternatively, you can do the same with *some()* and `imap()`.

One argument variant is a simple tool to keep your data free of falsy junk. This one returns non-empty description lines:

```
keep(description.splitlines())
```

Other common case is using generator expression instead of mapping function. Consider these two lines:

```
keep(f.name for f in fields)      # sugar generator expression
keep(attrgetter('name'), fields) # pure functions
```

mapcat (*f*, **seqs*)
imapcat (*f*, **seqs*)
Maps given sequence(s) and then concatenates results, essentially a shortcut for `cat(map(f, *seqs))`. Come in handy when extracting multiple values from every sequence item or transforming nested sequences:

```
# Get all the lines of all the texts in single flat list
mapcat(str.splitlines, bunch_of_texts)

# Extract all numbers from strings
mapcat(partial(re_all, r'\d+'), bunch_of_strings)
```

without (*seq*, **items*)
iwithout (*seq*, **items*)
Returns sequence with `items` removed, preserves order. Designed to work with a few `items`, this allows removing unhashable objects:

```
no_empty_lists = without(lists, [])
```

In case of large amount of unwanted elements one can use *remove()*:

```
remove(set(unwanted_elements), seq)
```

Or simple set difference if order of sequence is irrelevant.

Split and chunk

split (*pred*, *seq*)
isplit (*pred*, *seq*)
Splits sequence items which pass predicate from the ones that don't, essentially returning a tuple `filter(pred, seq), remove(pred, seq)`.

For example, this way one can separate private attributes of an instance from public ones:

```
private, public = split(re_tester('^_'), dir(instance))
```

Split absolute and relative urls using extended predicate semantics:

```
absolute, relative = split(r'^http://', urls)
```

split_at (*n*, *seq*)
isplit_at (*n*, *seq*)
Splits sequence at given position, returning a tuple of its start and tail.

split_by (*pred*, *seq*)

isplit_by(*pred*, *seq*)
Splits start of sequence, consisting of items passing predicate, from the rest of it. Works similar to `takewhile(pred, seq)`, `dropwhile(pred, seq)`, but returns lists and works with iterator `seq` correctly:

```
split_by(bool, iter([-2, -1, 0, 1, 2]))
# [-2, -1], [0, 1, 2]
```

takewhile(*pred*[, *seq*])
Yeilds elements of `seq` as long as they pass `pred`. Stops on first one which makes predicate falsy:

```
# Extract first paragraph of text
takewhile(re_tester(r'\S'), text.splitlines())

# Build path from node to tree root
takewhile(bool, iterate(attrgetter('parent'), node))
```

dropwhile(*pred*[, *seq*])
This is a mirror of *takewhile()*. Skips elements of given sequence while `pred` is true and yields the rest of it:

```
# Skip leading whitespace-only lines
dropwhile(re_tester('^\s*$'), text_lines)
```

group_by(*f*, *seq*)
Groups elements of `seq` keyed by the result of `f`. The value at each key will be a list of the corresponding elements, in the order they appear in `seq`. Returns `defaultdict(list)`.

```
stats = group_by(len, ['a', 'ab', 'b'])
stats[1] # -> ['a', 'b']
stats[2] # -> ['ab']
stats[3] # -> [], since stats is defaultdict
```

One can use *split()* when grouping by boolean predicate. See also `itertools.groupby()`.

group_by_keys(*get_keys*, *seq*)
Groups elements of `seq` having multiple keys each into `defaultdict(list)`. Can be used to reverse grouping:

```
posts_by_tag = group_by_keys(attrgetter(tags), posts)
sentences_with_word = group_by_keys(str.split, sentences)
```

group_values(*seq*)
Groups values of `(key, value)` pairs. May think of it like `dict()` but collecting collisions:

```
group_values(keep(r'^--(\w+)=(.+)', sys.argv))
```

partition(*n*[, *step*], *seq*)
ipartition(*n*[, *step*], *seq*)
Returns a list of lists of `n` items each, at offsets `step` apart. If `step` is not supplied, defaults to `n`, i.e. the partitions do not overlap. Returns only full length-`n` partitions, in case there are not enough elements for last partition they are ignored.

Most common use is deflattening data:

```
# Make a dict from flat list of pairs
dict(ipartition(2, flat_list_of_pairs))
```

```
# Structure user credentials
{id: (name, password) for id, name, password in ipartition(3, users)}
```

A three argument variant of *partition()* can be used to process sequence items in context of their neighbors:

```
# Smooth data by averaging out with a sliding window
[sum(window) / n for window in ipartition(n, 1, data_points)]
```

Also look at *pairwise()* for similar use. Other use of *partition()* is processing sequence of data elements or jobs in chunks, but take a look at *chunks()* for that.

chunks(*n*[, *step*], *seq*)
ichunks(*n*[, *step*], *seq*)

Returns a list of lists like *partition()*, but may include partitions with fewer than `n` items at the end:

```
chunks(2, 'abcde')
# -> ['ab', 'cd', 'e'])

chunks(2, 4, 'abcde')
# -> ['ab', 'e'])
```

Handy for batch processing.

partition_by(*f*, *seq*)
ipartition_by(*f*, *seq*)

Partition `seq` into list of lists or iterator of iterators splitting at `f(item)` change.

Data handling

distinct(*seq*, *key=identity*)
idistinct(*seq*, *key=identity*)

Returns the given sequence with duplicates removed. Preserves order. If `key` is supplied then distinguishes values by comparing their keys.

Note: Elements of a sequence or their keys should be hashable.

---

with_prev(*seq*, *fill=None*)

Returns an iterator of a pair of each item with one preceding it. Yields *fill* or *None* as preceding element for first item.

Great for getting rid of clunky `prev` housekeeping in for loops. This way one can indent first line of each paragraph while printing text:

```
for line, prev in with_prev(text.splitlines()):
    if not prev:
        print '    ',
    print line
```

Use *pairwise()* to iterate only on full pairs.

with_next(*seq*, *fill=None*)

Returns an iterator of a pair of each item with one next to it. Yields *fill* or *None* as next element for last item. See also *with_prev()* and *pairwise()*.

pairwise(*seq*)
Yields pairs of items in `seq` like `(item0, item1), (item1, item2), ...`. A great way to process sequence items in a context of each neighbor:

```
# Check if seq is non-descending
all(left <= right for left, right in pairwise(seq))
```

count_by(*f*, *seq*)
Counts numbers of occurrences of values of `f` on elements of `seq`. Returns `defaultdict(int)` of counts.

Calculating a histogram is one common use:

```
# Get a length histogram of given words
count_by(len, words)
```

count_reps(*seq*)
Counts number of repetitions of each value in `seq`. Returns `defaultdict(int)` of counts. This is faster and shorter alternative to `count_by(identity, ...)`

reductions(*f*, *seq*[, *acc*])
ireductions(*f*, *seq*[, *acc*])
Returns a sequence of the intermediate values of the reduction of `seq` by `f`. In other words it yields a sequence like:

```
reduce(f, seq[:1], [acc]), reduce(f, seq[:2], [acc]), ...
```

You can use *`sums()`* or *`isums()`* for a common use of getting list of partial sums.

sums(*seq*[, *acc*])
isums(*seq*[, *acc*])
Same as *`reductions()`* or *`ireductions()`* with reduce function fixed to addition.

Find out which straw will break camels back:

```
first(i for i, total in enumerate(isums(straw_weights))
        if total > camel_toughness)
```

CHAPTER 6

Collections

Unite

merge (**colls*)
Merges several collections of same type into one: dicts, sets, lists, tuples, iterators or strings. For dicts values of later dicts override values of former ones with same keys.

Can be used in variety of ways, but merging dicts is probably most common:

```
def utility(**options):
    defaults = {...}
    options = merge(defaults, options)
    ...
```

If you merge sequences and don't need to preserve collection type, then use *concat()* or *iconcat()* instead.

join (*colls*)
Joins collections of same type into one. Same as *merge()*, but accepts iterable of collections.

Use *cat()* and *icat()* for non-type preserving sequence join.

Transform and select

All functions in this section support *Extended function semantics*.

walk (*f*, *coll*)
Returns a collection of same type as `coll` consisting of its elements mapped with the given function:

```
walk(inc, {1, 2, 3}) # -> {2, 3, 4}
walk(inc, (1, 2, 3)) # -> (2, 3, 4)
```

When walking dict, `(key, value)` pairs are mapped, i.e. this lines *flip()* dict:

```
swap = lambda (k, v): (v, k)
walk(swap, {1: 10, 2: 20})
```

walk() works with strings too:

```
walk(lambda x: x * 2, 'ABC')    # -> 'AABBCC'
walk(compose(str, ord), 'ABC') # -> '656667'
```

One should probably use *map()* or imap() when doesn't need to preserve collection type.

walk_keys (*f*, *coll*)

Walks keys of coll, mapping them with the given function. Works with mappings and collections of pairs:

```
walk_keys(str.upper, {'a': 1, 'b': 2}) # {'A': 1, 'B': 2}
walk_keys(int, json.loads(some_dict))  # restore key type lost in translation
```

Important to note that it preserves collection type whenever this is simple dict, defaultdict, OrderedDict or any other mapping class or a collection of pairs.

walk_values (*f*, *coll*)

Walks values of coll, mapping them with the given function. Works with mappings and collections of pairs.

Common use is to process values somehow:

```
clean_values = walk_values(int, form_values)
sorted_groups = walk_values(sorted, groups)
```

Hint: you can use *partial(sorted, key=...)* instead of sorted() to sort in non-default way.

Note that walk_values() has special handling for defaultdicts. It constructs new one with values mapped the same as for ordinary dict, but a default factory of new defaultdict would be a composition of f and old default factory:

```
d = defaultdict(lambda: 'default', a='hi', b='bye')
walk_values(str.upper, d)
# -> defaultdict(lambda: 'DEFAULT', a='HI', b='BYE')
```

select (*pred*, *coll*)

Filters elements of coll by pred constructing a collection of same type. When filtering a dict pred receives (key, value) pairs. See *select_keys()* and *select_values()* to filter it by keys or values respectively:

```
select(even, {1, 2, 3, 10, 20})
# -> {2, 10, 20}

select(lambda (k, v): k == v, {1: 1, 2: 3})
# -> {1: 1}
```

select_keys (*pred*, *coll*)

Select part of a dict or a collection of pairs with keys passing the given predicate.

This way a public part of instance attributes dictionary could be selected:

```
is_public = complement(re_tester('^_'))
public = select_keys(is_public, instance.__dict__)
```

select_values (*pred*, *coll*)

Select part of a dict or a collection of pairs with values passing the given predicate.

Strip falsy values from dict:

```
select_values(bool, some_dict)
```

compact (*coll*)
Removes falsy values from given collection. When compacting a dict all keys with falsy values are trashed.

Extract integer data from request:

```
compact(walk_values(silent(int), request_dict))
```

Dict utils

merge_with (*f*, **dicts*)
join_with (*f*, *dicts*)
Merge several dicts combining values for same key with given function:

```
merge_with(list, {1: 1}, {1: 10, 2: 2})
# -> {1: [1, 10], 2: [2]}

merge_with(sum, {1: 1}, {1: 10, 2: 2})
# -> {1: 11, 2: 2}

join_with(first, ({n % 3: n} for n in range(100, 110)))
# -> {0: 102, 1: 100, 2: 101}
```

zipdict (*keys*, *vals*)
Returns a dict with the `keys` mapped to the corresponding `vals`. Stops pairing on shorter sequence end:

```
zipdict('abcd', range(4))
# -> {'a': 0, 'b': 1, 'c': 2, 'd': 3}

zipdict('abc', count())
# -> {'a': 0, 'b': 1, 'c': 2}
```

flip (*mapping*)
Flip passed dict swapping its keys and values. Also works for sequences of pairs. Preserves collection type:

```
flip(OrderedDict(['aA', 'bB']))
# -> OrderedDict([('A', 'a'), ('B', 'b')])
```

project (*mapping*, *keys*)
Returns a dict containing only those entries in `mapping` whose key is in `keys`.

Most useful to shrink some common data or options to predefined subset. One particular case is constructing a dict of used variables:

```
merge(project(__builtins__, names), project(globals(), names))
```

omit (*mapping*, *keys*)
Returns a copy of `mapping` with `keys` omitted. Preserves collection type:

```
omit({'a': 1, 'b': 2, 'c': 3}, 'ac')
# -> {'b': 2}
```

izip_values(**dicts*)
Yields tuples of corresponding values of given dicts. Skips any keys not present in all of the dicts. Comes in handy when comparing two or more dicts:

```
max_change = max(abs(x - y) for x, y in izip_values(items, old_items))
```

izip_dicts(**dicts*)
Yields tuples like `(key, value1, value2, ...)` for each common key of all given dicts. A neat way to process several dicts at once:

```
changed_items = [id for id, (new, old) in izip_dicts(items, old_items)
                 if abs(new - old) >= PRECISION]

lines = {id: cnt * price for id, (cnt, price) in izip_dicts(amounts, prices)}
```

See also *izip_values()*.

get_in(*coll*, *path*, *default=None*)
Returns a value corresponding to `path` in nested collection:

```
get_in({"a": {"b": 42}}, ["a", "b"])     # -> 42
get_in({"a": {"b": 42}}, ["c"], "foo")   # -> "foo"
```

set_in(*coll*, *path*, *value*)
Creates a nested collection with the `value` set at specified `path`. Original collection is not changed:

```
set_in({"a": {"b": 42}}, ["a", "b"], 10)
# -> {"a": {"b": 10}}

set_in({"a": {"b": 42}}, ["a", "c"], 10)
# -> {"a": {"b": 42, "c": 10}}
```

update_in(*coll*, *path*, *update*, *default=None*)
Creates a nested collection with a value at specified `path` updated:

```
update_in({"a": {}}, ["a", "cnt"], inc, default=0)
# -> {"a": {"cnt": 1}}
```

Data manipulation

where(*mappings*, ***cond*)
iwhere(*mappings*, ***cond*)
Looks through each value in given sequence of dicts, returning a list or an iterator of all the dicts that contain all key-value pairs in `cond`:

```
where(plays, author="Shakespeare", year=1611)
# => [{"title": "Cymbeline", "author": "Shakespeare", "year": 1611},
#     {"title": "The Tempest", "author": "Shakespeare", "year": 1611}]
```

Iterator version could be used for efficiency or when you don't need the whole list. E.g. you are looking for the first match:

```
first(iwhere(plays, author="Shakespeare"))
# => {"title": "The Two Gentlemen of Verona", ...}
```

pluck(*key*, *mappings*)
ipluck(*key*, *mappings*)
Returns a list or an iterator of values for `key` in each mapping in the given sequence. Essentially a shortcut for:

```
map(operator.itemgetter(key), mappings)
```

pluck_attr(*attr*, *objects*)
ipluck_attr(*attr*, *objects*)
Returns a list or an iterator of values for `attr` in each object in the given sequence. Essentially a shortcut for:

```
map(operator.attrgetter(attr), objects)
```

Useful when dealing with collections of ORM objects:

```
users = User.query.all()
ids = pluck_attr('id', users)
```

invoke(*objects*, *name*, **args*, ***kwargs*)
iinvoke(*objects*, *name*, **args*, ***kwargs*)
Calls named method with given arguments for each object in `objects` and returns a list or an iterator of results.

Content tests

is_distinct(*coll*, *key=identity*)
Checks if all elements in the collection are different:

```
assert is_distinct(field_names), "All fields should be named differently"
```

Uses `key` to differentiate values. This way one can check if all first letters of `words` are different:

```
is_distinct(words, key=0)
```

all(*[pred]*, *seq*)
Checks if `pred` holds every element in a `seq`. If `pred` is omitted checks if all elements of `seq` are truthy (which is the same as in built-in `all()`):

```
they_are_ints = all(is_instance(n, int) for n in seq)
they_are_even = all(even, seq)
```

Note that, first example could be rewritten using *`isa()`* like this:

```
they_are_ints = all(isa(int), seq)
```

any(*[pred]*, *seq*)
Returns `True` if `pred` holds for any item in given sequence. If `pred` is omitted checks if any element of `seq` is truthy.

Check if there is a needle in haystack, using *extended predicate semantics*:

```
any(r'needle', haystack_strings)
```

none(*[pred]*, *seq*)
Checks if none of items in given sequence pass `pred` or is truthy if `pred` is omitted.

Just a stylish way to write `not any(...)`:

```
assert none(' ' in name for name in names), "Spaces in names not allowed"
```

one (*[pred], seq*)
Returns true if exactly one of items in `seq` passes `pred`. Cheks for truthiness if `pred` is omitted.

some (*[pred], seq*)
Finds first item in `seq` passing `pred` or first that is true if `pred` is omitted.

Low-level helpers

empty (*coll*)
Returns an empty collection of the same type as `coll`.

iteritems (*coll*)
Returns an iterator of items of a `coll`. This means `key, value` pairs for any dictionaries:

```
list(iteritems({1, 2, 42}))
# -> [1, 42, 2]

list(iteritems({'a': 1}))
# -> [('a', 1)]
```

itervalues (*coll*)
Returns an iterator of values of a `coll`. This means values for any dictionaries and just elements for other collections:

```
list(itervalues({1, 2, 42}))
# -> [1, 42, 2]

list(itervalues({'a': 1}))
# -> [1]
```

CHAPTER 7

Functions

identity (*x*)
Returns its argument.

constantly (*x*)
Returns function accepting any args, but always returning `x`.

caller (**args*, ***kwargs*)
Returns function calling its argument with passed arguments.

partial (*func*, **args*, ***kwargs*)
Returns partial application of `func`. A re-export of `functools.partial()`. Can be used in a variety of ways. DSLs is one of them:

```
field = dict
json_field = partial(field, json=True)
```

rpartial (*func*, **args*)
Partially applies last arguments in `func`:

```
from operator import div
one_third = rpartial(div, 3.0)
```

Arguments are passed to `func` in the same order as they came to *`rpartial()`*:

```
separate_a_word = rpartial(str.split, ' ', 1)
```

func_partial (*func*, **args*, ***kwargs*)
Like *`partial()`* but returns a real function. Which is useful when, for example, you want to create a method of it:

```
setattr(self, 'get_%s_display' % field.name, func_partial(_get_FIELD_display,␣
↪field))
```

Note: use *`partial()`* if you are ok to get callable object instead of function as it's faster.

curry (*func*[, *n*])
Curries function. For example, given function of two arguments `f(a, b)` returns function:

```
lambda a: lambda b: f(a, b)
```

Handy to make a partial factory:

```
make_tester = curry(re_test)
is_word = make_tester(r'^\w+$')
is_int = make_tester(r'^[1-9]\d*$')
```

But see *re_tester()* if you really need this.

rcurry (*func*[, *n*])
Curries function from last argument to first:

```
has_suffix = rcurry(str.endswith)
filter(has_suffix("ce"), ["nice", "cold", "ice"])
# -> ["nice", "ice"]
```

Can fix number of arguments when it's ambiguous:

```
to_power = rcurry(pow, 2) # curry 2 first args in reverse order
to_square = to_power(2)
to_cube = to_power(3)
```

autocurry (*func*)
Constructs a version of `func` returning its partial applications until sufficient arguments are passed:

```
def remainder(what, by):
    return what % by
rem = autocurry(remainder)

assert rem(10, 3) == rem(10)(3) == rem()(10, 3) == 1
assert map(rem(by=3), range(5)) == [0, 1, 2, 0, 1]
```

Can clean your code a bit when *partial()* makes it too cluttered.

compose (**fs*)
Returns composition of functions:

```
extract_int = compose(int, r'\d+')
```

Supports *Extended function semantics*.

rcompose (**fs*)
Returns composition of functions, with functions called from left to right. Designed to facilitate transducer-like pipelines:

```
# Note the use of iterator function variants everywhere
process = rcompose(
    partial(iremove, is_useless),
    partial(imap, process_row),
    partial(ichunks, 100)
)

for chunk in process(data):
    write_chunk_to_db(chunk)
```

Supports *Extended function semantics*.

juxt (**fs*)
ijuxt (**fs*)
Takes several functions and returns a new function that is the juxtaposition of those. The resulting function takes a variable number of arguments, and returns a list or iterator containing the result of applying each function to the arguments.

iffy (*[pred], action[, default=identity]*)
Returns function, which conditionally, depending on `pred`, applies `action` or `default`. If `default` is not callable then it is returned as is from resulting function. E.g. this will call all callable values leaving rest of them as is:

```
map(iffy(callable, caller()), values)
```

Common use it to deal with messy data:

```
dirty_data = ['hello', None, 'bye']
map(iffy(len), dirty_data)              # => [5, None, 3]
map(iffy(isa(str), len, 0), dirty_data) # => [5, 0, 3], also safer
```

Function logic

This family of functions supports creating predicates from other predicates and regular expressions.

complement (*pred*)
Constructs a negation of `pred`, i.e. a function returning a boolean opposite of original function:

```
is_private = re_tester(r'^_')
is_public = complement(is_private)

# or just
is_public = complement(r'^_')
```

all_fn (**fs*)
any_fn (**fs*)
none_fn (**fs*)
one_fn (**fs*)
Construct a predicate returning `True` when all, any, none or exactly one of `fs` return `True`. Support short-circuit behavior.

```
is_even_int = all_fn(isa(int), even)
```

some_fn (**fs*)
Constructs function calling `fs` one by one and returning first true result.

Enables creating functions by short-circuiting several behaviours:

```
get_amount = some_fn(
    lambda s: 4 if 'set of' in s else None,
    r'(\d+) wheels?',
    compose({'one': 1, 'two': 2, 'pair': 2}, r'(\w+) wheels?')
)
```

If you wonder how on Earth one can *`compose()`* dict and string see *Extended function semantics*.

CHAPTER 8

Decorators

`@decorator`

Transforms a flat wrapper into a decorator with or without arguments. `@decorator` passes special `call` object as a first argument to a wrapper. A resulting decorator will preserve function module, name and docstring. It also adds `__wrapped__` attribute referring to wrapped function and `__original__` attribute referring to innermost wrapped one.

Here is a simple logging decorator:

```
@decorator
def log(call):
    print call._func.__name__, call._args, call._kwargs
    return call()
```

`call` object also supports by name arg introspection and passing additional arguments to decorated function:

```
@decorator
def with_phone(call):
    # call.request gets actual request value upon function call
    request = call.request
    # ...
    phone = Phone.objects.get(number=request.GET['phone'])
    # phone arg is added to *args passed to decorated function
    return call(phone)

@with_phone
def some_view(request, phone):
    # ... some code using phone
    return # ...
```

A better practice would be adding keyword argument not positional. This makes such decorators more composable:

```
@decorator
def with_phone(call):
    # ...
```

```
    return call(phone=phone)

@decorator
def with_user(call):
    # ...
    return call(user=user)

@with_phone
@with_user
def some_view(request, phone=None, user=None):
    # ...
    return # ...
```

If a function wrapped with `@decorator` has arguments other than `call`, then decorator with arguments is created:

```
@decorator
def joining(call, sep):
    return sep.join(call())
```

You can see more examples in `flow` and `debug` submodules source code.

@contextmanager (*func*)

A decorator helping to create context managers. Resulting functions also behave as decorators.

A simple example:

```
@contextmanager
def tag(name):
    print "<%s>" % name,
    yield
    print "</%s>" % name

with tag("h1"):
    print "foo",
# -> <h1> foo </h1>
```

Using as decorator:

```
@tag('strong')
def shout(text):
    print text.upper()

shout('hooray')
# -> <strong> HOORAY </strong>
```

@wraps (*wrapped*[, *assigned*][, *updated*])

An utility to pass function metadata from wrapped function to a wrapper. Copies all function attributes including `__name__`, `__module__` and `__doc__`.

In addition adds `__wrapped__` attribute referring to the wrapped function and `__original__` attribute referring to innermost wrapped one.

Mostly used to create decorators:

```
def some_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
```

```
        do_something(*args, **kwargs)
        return func(*args, **kwargs)
```

But see also *@decorator* for that. This is extended version of `functools.wraps()`.

unwrap(*func*)
Get the object wrapped by `func`.

Follows the chain of `__wrapped__` attributes returning the last object in the chain.

This is a backport from python 3.4.

class ContextDecorator
A base class or mixin that enables context managers to work as decorators.

CHAPTER 9

Flow

@silent

Ignore all real exceptions (descendants of `Exception`). Handy for cleaning data such as user input:

```
brand_id = silent(int)(request.GET['brand_id'])
ids = keep(silent(int), request.GET.getlist('id'))
```

And in data import/transform:

```
get_greeting = compose(silent(string.lower), re_finder(r'(\w+)!'))
map(get_greeting, ['a!', ' B!', 'c.'])
# -> ['a', 'b', None]
```

Note: Avoid silencing non-primitive functions, use *`@ignore()`* instead and even then be careful not to swallow exceptions unintentionally.

@ignore(*errors*, *default=None*)

Same as *`@silent`*, but able to specify `errors` to catch and `default` to return in case of error caught. `errors` can either be exception class or tuple of them.

suppress(**errors*)

A context manager which suppresses given exceptions under its scope:

```
with suppress(HttpError):
    # Assume this request can fail, and we are ok with it
    make_http_request()
```

@once

@once_per_args

@once_per(**argnames*)

Call function only once, once for every combination of values of its arguments or once for every combination of given arguments. Thread safe. Handy for various initialization purposes:

```
# Global initialization
@once
def initialize_cache():
    conn = some.Connection(...)
    # ... set up everything

# Per argument initialization
@once_per_args
def initialize_language(lang):
    conf = load_language_conf(lang)
    # ... set up language

# Setup each class once
class SomeManager(Manager):
    @once_per('cls')
    def _initialize_class(self, cls):
        pre_save.connect(self._pre_save, sender=cls)
        # ... set up signals, no dups
```

raiser(*exception_or_class=Exception*, **args*, ***kwargs*)
Constructs function that raises given exception with given arguments on any invocation.

@retry(*tries*, *errors=Exception*, *timeout=0*)
Every call of the decorated function is tried up to `tries` times. The first attempt counts as a try. Retries occur when any subclass of `errors` is raised (`errors` can be an exception class or a list/tuple of exception classes). There will be a delay in `timeout` seconds between tries.

A common use is to wrap some unreliable action:

```
@retry(3, errors=HttpError)
def download_image(url):
    # ... make http request
    return image
```

You can pass callable as `timeout` to achieve exponential delays or other complex behavior:

```
@retry(3, errors=HttpError, timeout=lambda a: 2 ** a)
def download_image(url):
    # ... make http request
    return image
```

fallback(**approaches*)
Tries several approaches until one works. Each approach is either callable or a tuple `(callable, errors)`, where errors is an exception class or a tuple of classes, which signal to fall back to next approach. If `errors` is not supplied then fall back is done for any `Exception`:

```
fallback(
    (partial(send_mail, ADMIN_EMAIL, message), SMTPException),
    partial(log.error, message),
    raiser(FeedbackError, "Unable to notify admin")
)
```

limit_error_rate(*fails*, *timeout*, *exception=ErrorRateExceeded*)
If function fails to complete `fails` times in a row, calls to it will be intercepted for `timeout` with `exception` raised instead. A clean way to short-circuit function taking too long to fail:

```
@limit_error_rate(fails=5, timeout=60, exception=RequestError('Temporary␣
↪unavailable'))
def do_request(query):
    # ... make a http request
    return data
```

Can be combined with *ignore()* to silently stop trying for a while:

```
@ignore(ErrorRateExceeded, default={'id': None, 'name': 'Unknown'})
@limit_error_rate(fails=5, timeout=60)
def get_user(id):
    # ... make a http request
    return data
```

@collecting

Transforms generator or other iterator returning function into list returning one.

Handy to prevent quirky iterator-returning properties:

```
@property
@collecting
def path_up(self):
    node = self
    while node:
        yield node
        node = node.parent
```

Also makes list constructing functions beautifully yielding.

@joining(*sep*)

Wraps common python idiom "collect then join" into a decorator. Transforms generator or alike into function, returning string of joined results. Automatically converts all elements to separator type for convenience.

Goes well with generators with some ad-hoc logic within:

```
@joining(', ')
def car_desc(self):
    yield self.year_made
    if self.engine_volume: yield '%s cc' % self.engine_volume
    if self.transmission:  yield self.get_transmission_display()
    if self.gear:          yield self.get_gear_display()
    # ...
```

Use `unicode` separator to get unicode result:

```
@joining(u', ')
def car_desc(self):
    yield self.year_made
    # ...
```

See also *str_join()*.

@post_processing(*func*)

Passes decorated function result through `func`. This is the generalization of *@collecting* and *@joining()*. Could save you writing a decorator or serve as extended comprehensions:

```
@post_processing(dict)
def make_cond(request):
    if request.GET['new']:
```

```
        yield 'year__gt', 2000
    for key, value in request.GET.items():
        if value == '':
            continue
        # ...
```

CHAPTER 10

String utils

re_find(*regex*, *s*, *flags=0*)

Finds `regex` in `s`, returning the match in the simplest possible form guessed by captures in given regular expression:

Captures	Return value
no captures	a matched string
single positional capture	a substring matched by capture
only positional captures	a tuple of substrings for captures
only named captures	a dict of substrings for captures
mixed pos/named captures	a match object

Returns `None` on mismatch.

```
# Find first number in a line
silent(int)(re_find(r'\d+', line))

# Find number of men in a line
re_find(r'(\d+) m[ae]n', line)

# Parse uri into nice dict
re_find(r'^/post/(?P<id>\d+)/(?P<action>\w+)$', uri)
```

re_test(*regex*, *s*, *flags=0*)

Tests whether `regex` can be found in `s`.

re_all(*regex*, *s*, *flags=0*)

re_iter(*regex*, *s*, *flags=0*)

Returns a list or iterator of all matches of `regex` in `s`. Matches are presented in most simple form possible, see table in *re_find()* docs.

```
# A fast and dirty way to parse ini section into dict
dict(re_iter('(\w+)=(\w+)', ini_text))
```

re_finder(*regex*, *flags=0*)

Returns a function that calls *re_find()* for it's sole argument. It's main purpose is quickly constructing

mapper functions for `map()` and friends.

See also *Extended function semantics*.

re_tester(*regex*, *flags=0*)
Returns a function that calls `re_test()` for it's sole argument. Aimed at quick construction of predicates for use in `filter()` and friends.

See also *Extended function semantics*.

str_join([*sep=""*], *seq*)
Joins sequence with `sep`. Same as `sep.join(seq)`, but forcefully converts all elements to separator type, `str` by default.

See also `joining()`.

cut_prefix(*s*, *prefix*)
Cuts prefix from given string if it's present.

cut_suffix(*s*, *suffix*)
Cuts suffix from given string if it's present.

CHAPTER 11

Calculation

@memoize(*key_func=None*)

Memoizes decorated function results, trading memory for performance. Can skip memoization for failed calculation attempts:

```
@memoize                            # Omitting parentheses is ok
def ip_to_city(ip):
    try:
        return request_city_from_slow_service(ip)
    except NotFound:
        return None                 # return None and memoize it
    except Timeout:
        raise memoize.skip(CITY)    # return CITY, but don't memoize it
```

Additionally `@memoize` exposes its memory for you to manipulate:

```
# Prefill memory
ip_to_city.memory.update({...})

# Forget everything
ip_to_city.memory.clear()
```

Custom *key_func* could be used to work with unhashable objects, insignificant arguments, etc:

```
@memoize(key_func=lambda obj, verbose=None: obj.key)
def do_things(obj, verbose=False):
    # ...
```

@make_lookuper

As *@memoize*, but with prefilled memory. Decorated function should return all available arg-value pairs, which should be a dict or a sequence of pairs. Resulting function will raise `LookupError` for any argument missing in it:

```
@make_lookuper
def city_location():
```

```
    return {row['city']: row['location'] for row in fetch_city_locations()}
```

If decorated function has arguments then separate lookuper with its own lookup table is created for each combination of arguments. This can be used to make lookup tables on demand:

```
@make_lookuper
def function_lookup(f):
    return {x: f(x) for x in range(100)}

fast_sin = function_lookup(math.sin)
fast_cos = function_lookup(math.cos)
```

Or load some resources, memoize them and use as a function:

```
@make_lookuper
def translate(lang):
    return make_list_of_pairs(load_translation_file(lang))

russian_phrases = map(translate('ru'), english_phrases)
```

@silent_lookuper

Same as *@make_lookuper*, but returns `None` on memory miss.

@cache(*timeout*, *key_func=None*)

Caches decorated function results for `timeout`. It can be either number of seconds or `datetime.timedelta`:

```
@cache(60 * 60)
def api_call(query):
    # ...
```

Cache can be invalidated before timeout with:

```
api_call.invalidate(query)  # Forget cache for query
api_call.invalidate_all()   # Forget everything
```

Custom `key_func` could be used same way as in *@memoize*:

```
# Do not use token in cache key
@cache(60 * 60, key_func=lambda query, token=None: query)
def api_call(query, token=None):
    # ...
```

CHAPTER 12

Type testing

isa(**types*)
Returns function checking if its argument is of any of given `types`.

Split labels from ids:

```
labels, ids = split(isa(str), values)
```

is_mapping(*value*)
is_set(*value*)
is_list(*value*)
is_tuple(*value*)
is_seq(*value*)
is_iter(*value*)
These functions check if value is `Mapping`, `Set`, `list`, `tuple`, `Sequence` or iterator respectively.

is_seqcoll(*value*)
Checks if `value` is a list or a tuple, which are both sequences and collections.

is_seqcont(*value*)
Checks if `value` is a list, a tuple or an iterator, which are sequential containers. It can be used to distinguish between value and multiple values in dual-interface functions:

```
def add_to_selection(view, region):
    if is_seqcont(region):
        # A sequence of regions
        view.sel().add_all(region)
    else:
        view.sel().add(region)
```

iterable(*value*)
Tests if `value` is iterable.

CHAPTER 13

Objects

@cached_property

Creates a property caching its result. One can rewrite cached value simply by assigning property. And clear cache by deleting it.

A great way to lazily attach some data to an object:

```
class MyUser(AbstractBaseUser):
    @cached_property
    def public_phones(self):
        return list(self.phones.filter(confirmed=True, public=True))
```

@monkey(*cls_or_module*, *name=None*)

Monkey-patches class or module by adding decorated function or property to it named `name` or the same as decorated function. Saves overwritten method to `original` attribute of decorated function for a kind of inheritance:

```
# A simple caching of all get requests,
# even for models for which you can't easily change Manager
@monkey(QuerySet)
def get(self, *args, **kwargs):
    if not args and list(kwargs) == ['pk']:
        cache_key = '%s:%d' % (self.model, kwargs['pk'])
        result = cache.get(cache_key)
        if result is None:
            result = get.original(self, *args, **kwargs)
            cache.set(cache_key, result)
        return result
    else:
        return get.original(self, *args, **kwargs)
```

class namespace

A base class that prevents its member functions turning into methods:

```
class Checks(namespace):
    is_str = lambda value: isinstance(value, str)
```

```
    max_len = lambda l: lambda value: len(value) <= l

field_checks = all_fn(Checks.is_str, Checks.max_len(30))
```

CHAPTER 14

Debugging

tap(*value*, *label=None*)

Prints value and then returns it. Useful to tap into some functional pipeline for debugging:

```
fields = (f for f in fields_for(category) if section in tap(tap(f).sections))
# ... do something with fields
```

If `label` is specified then it's printed before corresponding value:

```
squares = {tap(x, 'x'): tap(x * x, 'x^2') for x in [3, 4]}
# x: 3
# x^2: 9
# x: 4
# x^2: 16
# => {3: 9, 4: 16}
```

@log_calls(*print_func*, *errors=True*, *stack=True*)
@print_calls(*errors=True*, *stack=True*)

Will log or print all function calls, including arguments, results and raised exceptions. Can be used as decorator or tapped into call expression:

```
sorted_fields = sorted(fields, key=print_calls(lambda f: f.order))
```

If `errors` is set to `False` then exceptions are not logged. This could be used to separate channels for normal and error logging:

```
@log_calls(log.info, errors=False)
@log_errors(log.exception)
def some_suspicious_function(...):
    # ...
    return result
```

@log_enters(*print_func*)
@print_enters
@log_exits(*print_func*, *errors=True*, *stack=True*)

@print_exits(*errors=True*, *stack=True*)
Will log or print every time execution enters or exits the function. Should be used same way as *@log_calls()* and *@print_calls()* when you need to track only one event per function call.

@log_errors(*print_func*, *label=None*, *stack=True*)
@print_errors(*label=None*, *stack=True*)
Will log or print all function errors providing function arguments causing them. If `stack` is set to `False` then each error is reported with simple one line message.

Can be combined with *@silent* or *@ignore()* to trace occasionally misbehaving function:

```
@silent
@log_errors(logging.warning)
def guess_user_id(username):
    initial = first_guess(username)
    # ...
```

Can also be used as context decorator:

```
with print_errors('initialization', stack=False):
    load_this()
    load_that()
    # ...
# SomeException: a bad thing raised in initialization
```

@log_durations(*print_func*, *label=None*)
@print_durations(*label=None*)
Will time each function call and log or print its duration:

```
@log_durations(logging.info)
def do_hard_work(n):
    samples = range(n)
    # ...

# 121 ms in do_hard_work(10)
# 143 ms in do_hard_work(11)
# ...
```

A block of code could be timed with a help of context manager:

```
with print_durations('Creating models'):
    Model.objects.create(...)
    # ...

# 10.2 ms in Creating models
```

log_iter_durations(*seq*, *print_func*, *label=None*)
print_iter_durations(*seq*, *label=None*)
Wraps iterable `seq` into generator logging duration of processing of each item:

```
for item in print_iter_durations(seq, label='hard work'):
    do_smth(item)

# 121 ms in iteration 0 of hard work
# 143 ms in iteration 1 of hard work
# ...
```

CHAPTER 15

Primitives

isnone (*x*)
Checks if `x` is `None`. Handy with filtering functions:

```
remove(isnone, list_of_dirty_data)
```

Plays nice with *`silent()`*, which returns `None` on fail:

```
remove(isnone, imap(silent(int), strings_with_numbers))
```

Note that it's usually simpler to use *`keep()`* or *`compact()`* if you don't need to distinguish between `None` and other falsy values.

notnone (*x*)
Checks if `x` is not `None`. A shortcut for `complement(isnone)` meant to be used when `bool` is not specific enough. Compare:

```
select_values(notnone, data_dict) # removes None values
compact(data_dict)                # removes all falsy values
```

inc (*x*)
Increments its argument by 1.

dec (*x*)
Decrements its argument by 1.

even (*x*)
Checks if `x` is even.

odd (*x*)
Checks if `x` is odd.

Essays:

- Why Every Language Needs Its Underscore
- Functional Python Made Easy

- Abstracting Control Flow
- Painless Decorators

You can also look at the code or create an issue.

Python Module Index

f

Index

J

K

L

M

N

O

P

R

S

T

U

W

Z