sidekick Documentation

Release 0.8.1

Fábio Macêdo Mendes

Jun 23, 2020

Contents

Warning: Beta software You are using a software that has not reached a stable version yet. Please beware that interfaces might change, APIs might disappear and general breakage can occur before *1.0*.

If you plan to use this software for something important, please read the roadmap, and the issue tracker in Github. If you are unsure about the future of this project, please talk to the developers, or (better yet) get involved with the development of Sidekick!

CHAPTER 1

Overview

Sidekick is a functional library of functions and types designed to make functional programming more pleasant in Python. It emphasizes working with simple functions, immutable and data-centric types and iterators.

Sidekick uses (or maybe abuses) Python expressiveness to introduce some specific domain syntax and idioms that makes functional programming more natural, easy to use and still somewhat Pythonic. This approach distinguishes Sidekick from other projects like toolz, which explicitly aim to provide "low-tech" solutions based on common idioms or other libraries that simply port ideas and libraries from other programming languages to Python (even when that means alien Haskell-isms).

There are obvious limitations of what we can do syntax-wise as a library, and more so in what we can provide that will still play nicely with the rest of Python ecosystem. Sidekick is just one proposal in trying to find a nice balance between practicality and theoretical purity, between expressiveness and familiarity. There are many other excellent libraries in this landscape which can also be used alongside with Sidekick, and we certainly want to encourage you to explore other options in Python and in other programming languages as well.

1.1 So, what's up with functional programming?

Since functional programming (FP) means different things to different people, it is perhaps necessary to point out where we want to go with Sidekick. People often point at correctness, composability and ease of debugging and testing as major motivations. Those are certainly worthy goals when trying to design any system and functional programming offers some guidelines that help us to stay on track. Those ideas are fundamental to grasp real FP languages like Haskell, but are also useful in generic multi-paradigm languages such as Python.

CHAPTER 2

Contents

2.1 Introduction to functional programming

Functional programming, as the name implies, leans heavily towards using functions for solving problems. This contrasts with other approaches as, for instance, organizing code with classes or even relying too much on standard control flow structures such as `for`, `if`'s, `with`, etc.

Python is a multi-paradigm language that allows functional programming but was not conceived specifically for it. Functional programming is only practical if we have means of easily creating new functions and using them in cooperation. This means that functional code tends to be littered with many small functions that are passed to other (high order) functions to accomplish some elaborate task. Think of those functions as Lego bricks: they are simple, easy to grasp, tend to fit nicely with each other, but when used in conjunction can create elaborate structures.

Creating functions in Python usually means lots of `def` statements and `lambdas`[1]. This approach is perhaps a little bit clumsy for everyday FP use, hence Sidekick provides more efficient ways to construct new functions by implementing some new functionality on top of standard Python idioms. Those approaches can be split into 3 basic ideas:

1) *Specialization*: constrain a generic function by fixing some of its arguments.
2) *Creation*: provide easy ways to construct commonly used lambdas.
3) *Composition*: join two or more functions to generate a new function that inherit behaviors of both.

We will discuss those ideas in detail in the next sections.

2.1.1 Creating functions by specialization

Consider a simple function:

```
def succ(n: int) -> int:
    """
    Return the next integer after n.
```

(continues on next page)

[1] The name "lambda" has roots in lambda calculus, but do not expect beginners to make any sense of this. The lambda syntax was borrowed from LISP, in which the author of the language really wanted to make the association with lambda calculus explicit.

(continued from previous page)

```
    """

    return n + 1
```

It seems like it is only a function declaration, but in reality it does a few more things: it gives the function a name, defines its documentation string and sets an implicit scope for global variables. For top level functions of your public API, in which you probably want to declare all those properties, this is a very efficient syntax.

Put the same function in a different context and `def`'s start to feel verbose [#map]:

```
def increment_seq(numbers):
    def succ(n):
        return n + 1

    return map(succ, numbers)
```

Not only this code spends two whole lines to define such a trivial function, it forces us to explicitly come up with a name which does not clarify its intent anymore than seeing `n + 1` in the body of the function. Naming is hard, and we should avoid doing it if it has no purpose in making some piece of code more readable or comprehensible.

Lambdas are a much better fit for this case:

```
def increment_seq(numbers):
    return map(lambda x: x + 1, numbers)
```

But even that definition can be improved. `lambda x: x + 1` is just a special case of addition in which one of the arguments is one. Indeed, that notion that we can create new functions by "specializing" more generic functions is very a common functional programming idiom. Python does not have have a builtin system for doing that, but we can use a few functions from the standard lib for the same effect.

```
from functools import partial
from operator import add

def increment_seq(numbers):
    return map(partial(add, 1), numbers)
```

I wouldn't say it provides any tangible advantage over the previous case, but this code illustrates a powerful technique that can be really useful in other situations. We now want to take the good ideas from this example and make them more idiomatic and easy to use.

Currying

A simple function receives one single parameter and return a single value. In Python and most programming languages, the "function interface" can be considerably more complicated: the function might receive several parameters, keyword arguments, splicing (the `*args` and `**kwargs` expansions), and it might even produce side effects outside of what is visible from the inputs and outputs of the function.

If is often convenient for computer scientists to pretend in an idealized world in which all functions receive a single argument and return a single value. In fact, it is not even hard to transform some of those real world functions in this idealized version. Consider the function bellow:

```
def add(x, y):
    return x + y
```

This "complicated" two argument function can be easily simplified into a single argument function by passing x and y as values in a tuple, as such:

```
def add_tuple(args):
    return args[0] + args[1]
```

A second approach is to think that a multi-argument function is just a function that returns a second function that receives the remaining arguments. The function is evaluated only after the last argument is passed. This strange encoding is called *"currying"* after the computer scientist Haskell Curry, and is a very important idea in a foundational field of computer science called Lambda calculus.

The curried version of the "add" funtion is show bellow.

```
def add_curried(x):
    return lambda y: x + y
```

As crazy as `add_curried` may look, it is so powerful that some languages adopt it as their standard way of calling functions. This does not work very nicely Python, however, because the syntax becomes ugly and execution inefficient:

```
>>> add(1, 2) == add_tuple((1, 2)) == add_curried(1)(2) == 3
True
```

A nice middle ground between the standard multi-argument function and the fully curried version is the technique of "auto-currying": we execute the function normally if the callee passes all arguments, but curry it if some of them are missing. An auto-curried `add` function is implemented like this:

```
def add(x, y=None):
    # y was not given, so we curry!
    if y is None:
        return lambda y: x + y

    # y was given, hence we compute the sum
    else:
        return x + y
```

```
>>> add(1)(2) == add(1, 2)
True
```

One nice thing about auto-currying is that it doesn't (usually) break preexisting interfaces. This new add function continues to be useful in contexts that the standard implementation would be applied, but it now also accepts receiving an incomplete set of arguments transforming add in a convenient factory.

Even for only two arguments, implementing auto-currying this way already seems like a lot of trouble. Fortunately, the *`sidekick.functions.curry()`* decorator automates this whole process and we can implement auto-curried functions with very little extra work:

```
import sidekick.api as sk

@sk.curry(2)  # The 2 stands for the number of arguments
def add(x, y):
    return x + y
```

Ok, it is good that we can automatically curry functions. But why would anyone want to do that in any real world programming problem?

Remember when we said that the increment function (`lambda x:  x + 1`) was just a special case of addition when one of the arguments is was fixed to 1? This kind of "specialized" functions are trivial to create using curried functions: just apply the arguments you want to fix and the result will be a specialized version of the original function:

```
>>> incr = add(1)  # Fix first argument of add to 1
>>> incr(41)
42
```

2.1.2 Quick lambdas

Operators like `+`, `-`, `*`, `/`, etc are functions recognized as being so useful that they deserve an special syntax in the language. They are obvious candidates for creating a library of factory functions such as the example:

```
def incrementer(n):
    return lambda x: x + n

def multiplier(n):
    return lambda x: x * n

...
```

While there is no denying that those functions might be useful, such a library probably is not. It is hard to advocate for this approach when it is easier to define those simple one-liners on the fly than actually remembering their names.

Magic X, Y

Sidekick implements a clever approach that first appear in Python in popular functional programming libraries such as fn.py and placeholder. It exposes the "magic argument object" `X` that creates those simple one-liners using a very straightforward syntax: every operation we do with the magic object X, returns a function that would perform the same operation if X was the argument. For instance, to tell the X object to create a function that adds some number to its argument, just add this number to X:

```
from sidekick.api import X

incr = (X + 1)  # same as lambda x: x + 1
```

In a similar spirit, there is a second operator Y for creating functions of two arguments:

```
from sidekick import X, Y

div  = (X / Y)  # same as lambda x, y: x / y
rdiv = (Y / X)  # same as lambda x, y: y / x
```

Y is consistently treated as the second argument of the function, even if the expression does not involve X. Hence,

```
>>> incr = (Y + 1)  # return lambda x, y: y + 1
>>> incr("whatever", 41)
42
```

This magic object is great to create one-liners on the fly without having to remember names and function signatures. Functions created with the magic X and Y work very nicely when creating elaborate functional pipelines.

```
>>> nums = range(1000)
>>> squares = map((X * X), nums)
>>> odds = filter(X % 2, nums)
```

We will create more interesting examples later using other sidekick functions and operators.

The X and Y special objects cover most functionality in Python's `operator` module, but provides a more flexible and perhaps a more intuitive interface. But just as *operator* has its share of oddities and caveats (e.g., division is called truediv, it does not expose have reverse operations such as radd, rsub, etc). There are some limitations of what the magic X and Y can do.

Some operations **do not** work with those magic objects. Those are intrinsic limitations of Python syntax and runtime and will *never* be fixed in Sidekick.

- **Short circuit operators:** `(X or Y)` and `(X and Y)`. There is no perfect way to reproduce short circuit evaluation with functions, hence sidekick does not provide any real alternative.
- **Identity checks:** `(X is value)` or `(X is not value)`. Use *sidekick.pred.is_identical()* or its negative `~sk.is_identical(value)` instead.
- **Assignment operators:** `(X += value)`. Assignment operators are statements and cannot be assigned to values. This includes item deletion and item assignment for the same reason.
- **Containment check**: `(X in seq)` or `(seq in X)`. Use *sidekick.op.contains()* instead.
- **Method calling**: `X.attr` immediately returns a function that retrieves the `.attr` attribute of its argument. We cannot specify a method to obtain a behavior similar to operator.methodcaller. *sidekick.functions.method()* can produce method callers with an specialized syntax.

`sidekick.op` module

Most functions that would be created with the X and Y magical objects are present in Python's own operator module, which exposes Python operators and special methods as regular functions. Sidekick provides a copy of this module that exposes curried versions of those functions. It is convenient to create simple one-liners via partial application

```
>>> from sidekick import op
>>> succ = op.add(1)
>>> succ(41)
42
```

2.1.3 Composing functions

We mentioned before that good functional code behave like LEGO bricks: they can easily fit each other and we can organize them in countless and creative ways. The most common form of composition has the shape of a pipeline: we start with some piece of data and pass it through a series of functions to obtain the final result. Sidekick captures that idea in the *pipe()* function.

```
>>> sk.pipe(10, op.mul(4), op.add(2))
42
```

The data flow from the first argument from left to right: i.e., `10` is passed to `op.mul(4)`, resulting in `40`, which is then passed to `op.add(2)` to obtain `42`. Notice we are relying on the fact that functions in *sidekick.op* are all auto-curried.

It is often desirable to abstract the operations in a pipe without mentioning data. That is, we want to extract the transformations into a function and pass data later at call site. This is responsibility of the *pipeline()* function.

```
>>> func = sk.pipeline(op.mul(4), op.add(2))
>>> func(10)
42
```

Attentive readers might realize that `pipeline(*funcs)` is equivalent to `lambda x: pipe(x, *funcs)`.

Pipelining is a simple form of function composition. In mathematics, the standard notation for function composition passes the arguments in opposite direction (i.e., data flows from the right to the left)

```
>>> func = sk.compose(op.add(2), op.mul(4))
>>> func(10)
42
```

`sk.compose(*funcs)` is equivalent to `sk.pipeline(*reversed(funcs))`.

Composition syntax

Many functional languages have special operators dedicated to function composition. Python don't, but that does not prevent us from being creative. Most sidekick functions are not actually real functions, but rather instances of an special class *fn*. fn-functions extend regular functions in a number of interesting ways.

The first and perhaps more fundamental change is that fn-functions accept bitwise shift operators (>> and <<) to represent function composition. The argument flows through the composed function in the same direction that bit shift arrows points to:

```
>>> f = op.mul(4) >> op.add(2)  # First multiply by 4, then add 2
>>> f(10)
42
```

Obviously only sidekick-enabled functions accept this syntax. We can support arbitrary Python callables by prefixing the pipeline with the fn object, making it behave essentially as an identity function

```
>>> succ = lambda x: x + 1
>>> incr_pos = fn >> abs >> succ  # (or fn << incr << abs)
>>> incr_pos(-41)
42
```

The other bitwise operators `|  ^  &  ~` are re-purposed to compose logical operations on predicate functions. That is, `f | g` returns a function that produces the logical OR between f(x) and g(x), `f & g` produces the logical AND and so on. Evaluation is done in a "short circuit" fashion: `g` is only evaluated if f returns a "falsy" value like Python's `or` operator.

Logical composition of predicate functions is specially useful in methods such as filter, take_while, etc, that receive predicates.

```
>>> sk.filter(sk.is_divisible_by(3) | sk.is_divisible_by(2), range(10))
sk.iter([0, 2, 3, 4, 6, 8, ...])
```

The pipe operator `|` represents the standard or, while `^` is interpreted as exclusive or.

```
>>> sk.filter(sk.is_divisible_by(3) ^ sk.is_divisible_by(2), range(10))
sk.iter([2, 3, 4, 8, 9])
```

fn also extend regular functions with additional methods and properties, but we refer to the class documentation for more details.

Applicative syntax

Bitwise operators compose fn-functions. Sidekick also provides a syntax to apply arguments into functions. Ideally, the syntax should be equal to F#'s function application operation, i.e.,

```
x |> function == function <| x == function(x)
```

Unfortunately, this is not valid Python and it could only be implemented changing the core language, not as a mere library feature. The next best thing would be to re-purpose existing operators. Unfortunately, we do not have very good options here: bitwise operators are already taken (hence, we can't use the pipe `|` operator), comparison operators do not work well in chained operations[3] and arithmetic operators have a high precedence which makes the annoying to use.

Sidekick makes the following (admittedly less than ideal) choices:

```
f ** arg == f < arg == f(arg)
arg // f == arg > f == f(arg)
f @ arg  == sk.apply(f, arg)  # This is the applicative version, more on this later!
```

We do not really encourage the use of those operators, but they are here just to explore how the language would look like if it had dedicated operators like `g <| f <| x` and `x |> f |> g`. In practical terms, overloading `>`, `<`, `**` and `//` to represent function application just save us the annoyance of wrapping a long argument in parenthesis when we want to do function application. This is acceptable in interactive sessions, but it is highly non advisable in production code. That is why sidekick comes with a `evil` module to control when the extended interface should be enabled.

By default, fn-functions accept `>`, `<`, `**` and `//` to perform function application. We can execute *sidekick.evil.no_evil()* to disable this behavior. If you are feeling lucky, however, *sidekick.evil.forbidden_powers()* extend this behavior even to regular Python functions. It uses ugly hacks and obfuscated code, so we could not stress more to never use it in production code.

2.2 API Guide

This module describes the sidekick API and is a map to where to look for specific functions and types.

2.3 sidekick.evil

This module modifies the C Python interpreter and enable some Sidekick idioms to regular Python functions. It also have some internal facilities to patch builtin types that uses techniques similar to those found in the forbiddenfruit package. Use if you are curious, but do not use it for anything serious.

The public API has only two functions, *no_evil()* and *forbidden_powers()*. For other functionality, dig the code and know what you are doing.

2.3.1 Functions

forbidden_powers(**kwargs)	Apply all powers.
no_evil()	Remove overloading arithmetic operators from fn-functions.

[3] Python translates chains like `x > f > g` to `x > f and f > g`. This breaks the chain of function application and makes those operators unusable for this task.

2.3.2 API reference

sidekick.evil.**no_evil**()
Remove overloading arithmetic operators from fn-functions.

sidekick.evil.**forbidden_powers**(***kwargs*)
Apply all powers.

2.4 sidekick.functions

The functions in this module are responsible for creating, transforming, composing and introspecting other functions. Some of those functions might be familiar from the standard lib's functools module. In spite of those similarities, this module is not a drop-in replacement of the standard lib's `functools` module.

This module also exposes the *fn* type, that extends standard Python functions with new methods and operators. This extended function behavior is applied to most sidekick's functions and can be easily re-used to extend user code.

2.4.1 Basic types

fn(func)	Base class for function-like objects in Sidekick.

2.4.2 Function introspection and conversion

Stub	Represent a function declaration Stub.
arity	Return arity of a function.
signature	Return the signature of a function.
stub	Return a *Stub* object representing the function signature.
to_callable(func)	Convert argument to callable.
to_function(func[, name])	Return object as as Python function.
to_fn(func)	Convert callable to an *fn* object.

2.4.3 Partial application

curry(n[, func])	Return the curried version of a function of n arguments.
partial(*args, **kwargs)	Return a new function that partially apply the given arguments and keywords.
rpartial(func, *args, **kwargs)	Partially apply arguments from the right.
method	Return a function that calls a method of its argument with the given values.

2.4.4 Composition

compose	Create function that apply argument from right to left.
pipe	Pipe a value through a sequence of functions.
pipeline	Similar to compose, but order of application is reversed.

Continued on next page

Table 5 – continued from previous page

thread	Similar to pipe, but accept extra arguments to each function in the pipeline.
rthread	Like thread, but data is passed as last argument to functions, instead of first.
thread_if	Similar to thread, but each form must be a tuple with (test, fn, ... args) and only pass the argument to fn if the boolean test is True.
rthread_if	Similar to rthread, but each form must be a tuple with (test, fn, ... args) and only pass the argument to fn if the boolean test is True.
juxt	Juxtapose several functions.

2.4.5 Combinators

identity	The identity function.
ridentity	Return last positional argument.
always	Return a function that always return x when called with any number of arguments.
rec	Fix func first argument as itself.
trampoline	Decorator that implements tail call elimination via the trampoline technique.
value	Evaluate argument, if it is a function or return it otherwise.
call(*args, **kwargs)	Return a function caller.
do	Runs `func` on `x`, returns `x`.

2.4.6 Runtime control

once	Limit function to a single invocation.
thunk	A thunk that represents a lazy computation.
call_after	Creates a function that invokes func once it's called more than n times.
call_at_most	Creates a function that invokes func while it's called less than n times.
throttle	Limit the rate of execution of func to once at each `dt` seconds.
background	Return a function that executes in the background.
error	Raises the given exception.
raising	Creates function that raises the given exception.
retry	Retry to execute function at least n times before raising an error.
catch	Handle exception in function.

2.4.7 Transform arguments

flip	Flip the order of arguments in a binary operator.
select_args	Creates a function that calls func with the arguments reordered.

Continued on next page

Table 8 – continued from previous page

keep_args	Uses only the first n positional arguments to call func.
reverse_args	Creates a function that invokes func with the positional arguments order reversed.
skip_args	Skips the first n positional arguments before calling func.
splice_args	Return a function that receives a sequence as single argument and splice them into func.
variadic_args	Return a function that receives variadic arguments and pass them as a tuple to func.

2.4.8 API reference

class sidekick.functions.**fn**(*func*)

Base class for function-like objects in Sidekick.

curry(*arity*, *func=None*, ***kwargs*) → Union[sidekick.functions.fn.Curried, callable]

Return a curried function with given arity.

generator

Decorates generator function to return a sidekick iterator instead of a regular Python generator.

Examples

```
>>> @sk.generator
... def fibonacci():
...     x = y = 1
...     while True:
...         yield x
...         x, y = y, x + y
>>> fibonacci()
sk.iter([1, 1, 2, 3, 5, 8, ...])
```

partial(**args*, ***kwargs*)

Return a fn-function with all given positional and keyword arguments applied.

result(**args*, ***kwargs*)

Return a result instance after function call.

Exceptions are converted to Err() cases.

rpartial(**args*, ***kwargs*)

Like partial, but fill positional arguments from right to left.

single(**args*, ***kwargs*)

Similar to partial, but with a few constraints:

- Resulting function must be a function of a single positional argument.
- Placeholder expressions are evaluated passing this single argument to the resulting function.

Example

```
>>> add = fn(lambda x, y: x + y)
>>> g = add.single(_, 2 * _)
```

(continues on next page)

(continued from previous page)

```
>>> g(10)   # g(x) = x + 2 * x
30
```

Returns fn

thunk(**args*, ***kwargs*)
Pass all arguments to function, without executing.

Returns a thunk, i.e., a zero-argument function that evaluates only during the first execution and re-use the computed value in future evaluations.

See also:

thunk()

classmethod wraps(*func*, *fn_obj=None*)
Creates a fn function that wraps another function.

sidekick.functions.**quick_fn**(*func: callable*) → sidekick.functions.fn.fn
Faster fn constructor.

This is about twice as fast as the regular fn() constructor. It assumes that fn is

class sidekick.functions.**Stub**
Represent a function declaration Stub.

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

name
Alias for field number 0

render(*imports=False*) → str
Render complete stub file, optionally including imports.

required_imports() → set
Return set of imported symbols

signatures
Alias for field number 1

sidekick.functions.**arity**
Return arity of a function.

Examples

```
>>> from operator import add
>>> sk.arity(add)
2
```

sidekick.functions.**signature**
Return the signature of a function.

sidekick.functions.**stub**
Return a *Stub* object representing the function signature.

sidekick.functions.**to_function**(*func: Any*, *name=None*) → function
Return object as as Python function.

Non-functions are wrapped into a function definition.

sidekick.functions.**to_callable**(*func: Any*) → Callable
Convert argument to callable.

This differs from to_function in which it returns the most efficient version of object that has the same callable interface as the argument.

This *removes* sidekick's function wrappers such as fn and try to convert argument to a straightforward function value.

This defines the following semantics:

- Sidekick's fn: extract the inner function.
- None: return the identity function.
- Mappings: map.__getitem__
- Functions, methods and other callables: returned as-is.

sidekick.functions.**to_fn**(*func: Any*) → sidekick.functions.fn.fn
Convert callable to an *fn* object.

If func is already an *fn* instance, it is passed as is.

sidekick.functions.**curry**(*n*, *func=None*)
Return the curried version of a function of n arguments.

Curried functions return partial applications of the function if called with missing arguments:

```
>>> add = sk.curry(2, lambda x, y: x + y)
```

We can call a function two ways:

```
>>> add(1, 2) == add(1)(2)
True
```

This is useful for building simple functions from partial application

```
>>> succ = add(1)
>>> succ(2)
3
```

curry() is itself a curried function, hence it can be called as

```
>>> add = sk.curry(2)(lambda x, y: x + y)
```

or equivalently as a decorator

```
>>> @sk.curry(2)
... def add(x, y):
...     return x + y
```

Currying usually requires functions of fixed number of arguments (the number of arguments is called the *arity* of a function). We can control how many arguments participate in the auto-currying by passing the arity number as the first argument to the *curry()* function.

Variadic functions are accepted, and arity is understood as the minimum number of arguments necessary to invoke the function. The caller can, however, specify additional arguments.

But it accepts more than 2 arguments, if needed. (Notice that only the first two arguments auto-curry.)

```
>>> add = sk.curry(2, lambda *args: sum(args))
>>> add(1, 2, 3, 4)
10
```

Sometimes we don't want to specify the arity of a function or don't want to think too much about it. *curry()* accepts `'auto'` as an arity specifier that makes it try to infer the arity automatically. Under the hood, it just calls *arity()* to obtain the correct value.

Sidekick curries most functions where it makes sense. Variadic functions cannot be curried if the extra arguments can be passed by position. This decorator inspect the decorated function to determine if it can be curried or not.

sidekick.functions.**partial**(**args*, ***kwargs*) → sidekick.functions.fn.fn

Return a new function that partially apply the given arguments and keywords.

Additional positional and keyword arguments after partially applied to function

Parameters **func** – Function or func-like object.

Examples

```
>>> from operator import add
>>> incr =  sk.partial(add, 1)
>>> incr(41)
42
```

See also:

rpartial()

sidekick.functions.**rpartial**(*func: Optional[Callable]*, **args*, ***kwargs*) → sidekick.functions.fn.fn

Partially apply arguments from the right.

Examples

```
>>> from operator import truediv as div
>>> half = sk.rpartial(div, 2)
>>> half(42)
21.0
```

See also:

partial()

sidekick.functions.**method**

Return a function that calls a method of its argument with the given values.

A method caller object. It can be used as a function

```
>>> pop_first = sk.method("pop", 0)
>>> pop_first([1, 2, 3])
1
```

or as a function factory.

```
>>> pop_first = sk.method.pop(0)
>>> pop_first([1, 2, 3])
1
```

The second usage is syntactically cleaner and prevents the usage of invalid Python names. All method calls performed in the `sk.method` object returns the corresponding methodcaller function.

`sidekick.functions.`**`compose`**

Create function that apply argument from right to left.

compose(f, g, h, ...) ==> f << g << h << ...

Example

```
>>> f = sk.compose((X + 1), (X * 2))
>>> f(2)  # double than increment
5
```

See also:

pipe() pipeline()

`sidekick.functions.`**`pipe`**

Pipe a value through a sequence of functions.

I.e. `pipe(data, f, g, h)` is equivalent to `h(g(f(data)))` or to `data | f | g | h`, if `f, g, h` are fn objects.

Examples

```
>>> from math import sqrt
>>> sk.pipe(-4, abs, sqrt)
2.0
```

See also:

pipeline() compose() thread() rthread()

`sidekick.functions.`**`pipeline`**

Similar to compose, but order of application is reversed.

pipeline(f, g, h, ...) ==> f >> g >> h >> ...

Example

```
>>> f = sk.pipeline((X + 1), (X * 2))
>>> f(2)  # increment and double
6
```

See also:

pipe() compose()

`sidekick.functions.`**`thread`**

Similar to pipe, but accept extra arguments to each function in the pipeline.

Arguments are passed as tuples and the value is passed as the first argument.

Examples

```
>>> sk.thread(20, (op.div, 2), (op.mul, 4), (op.add, 2))
42.0
```

See also:

pipe() rthread()

sidekick.functions.**rthread**
Like thread, but data is passed as last argument to functions, instead of first.

Examples

```
>>> sk.rthread(2, (op.div, 20), (op.mul, 4), (op.add, 2))
42.0
```

See also:

pipe() thread()

sidekick.functions.**thread_if**
Similar to thread, but each form must be a tuple with (test, fn, ... args) and only pass the argument to fn if the boolean test is True.

If test is callable, the current value to the callable to decide if fn must be executed or not.

Like thread, Arguments are passed as tuples and the value is passed as the first argument.

Examples

```
>>> sk.thread_if(20, (True, op.div, 2), (False, op.mul, 4), (sk.is_even, op.add,␣
↪2))
12.0
```

See also:

thread() rthread_if()

sidekick.functions.**rthread_if**
Similar to rthread, but each form must be a tuple with (test, fn, ... args) and only pass the argument to fn if the boolean test is True.

If test is callable, the current value to the callable to decide if fn must be executed or not.

Like rthread, Arguments are passed as tuples and the value is passed as the last argument.

Examples

```
>>> sk.rthread_if(20, (True, op.div, 2), (False, op.mul, 4), (sk.is_even, op.add,␣
↪2))
0.1
```

See also:

thread() rthread_if()

sidekick.functions.**juxt**
Juxtapose several functions.

Creates a function that calls several functions with the same arguments and return a tuple with all results.

It return a tuple with the results of calling each function. If last=True or first=True, return the result of the last/first call instead of a tuple with all the elements.

Examples

We can create an argument logger using either first/last=True

```
>>> sqr_log = sk.juxt(print, (X * X), last=True)
>>> sqr_log(4)
4
16
```

Consume a sequence

```
>>> pairs = sk.juxt(next, next)
>>> nums = iter(range(10))
>>> pairs(nums), pairs(nums)
((0, 1), (2, 3))
```

sidekick.functions.**identity**
The identity function.

Return its first argument unchanged. Identity accepts one or more positional arguments and any number of keyword arguments.

Examples

```
>>> sk.identity(1, 2, 3, foo=4)
1
```

See also:

ridentity()

sidekick.functions.**ridentity**
Return last positional argument.

Similar to identity, but return the last positional argument and not the first. In the case the function receives a single argument, both identity functions coincide.

Examples

```
>>> sk.ridentity(1, 2, 3)
3
```

See also:

identity()

sidekick.functions.**always**
Return a function that always return x when called with any number of arguments.

Examples

```
>>> f = sk.always(42)
>>> f('answer', for_what='question of life, the universe ...')
42
```

sidekick.functions.**rec**
Fix func first argument as itself.

This is a version of the Y-combinator and is useful to implement recursion from scratch.

Examples

In this example, the factorial receive a second argument which is the function it must recurse to. rec pass the function to itself so now the factorial only needs the usual numeric argument.

```
>>> sk.map(
...     sk.rec(lambda f, n: 1 if n == 0 else n * f(f, n - 1)),
...     range(10),
... )
sk.iter([1, 1, 2, 6, 24, 120, ...])
```

sidekick.functions.**power**(*func: Optional[Callable], n: int*) → sidekick.functions.fn.fn
Return a function that applies f to is argument n times.

power(f, n)(x) ==> f(f(...f(x))) # apply f n times.

Examples

```
>>> g = sk.power((2 * X), 3)
>>> g(10)
80
```

sidekick.functions.**trampoline**
Decorator that implements tail call elimination via the trampoline technique.

Parameters **func** – A function that returns an args tuple to call it recursively or raise StopIteration when done.

Examples

```
>>> @sk.trampoline
... def fat(n, acc=1):
...     if n > 0:
...         return n - 1, acc * n
...     else:
...         raise StopIteration(acc)
>>> fat(5)
120
```

sidekick.functions.**call**(**args*, ***kwargs*) → sidekick.functions.fn.fn
Return a function caller.

Creates a function that receives another function and apply the given arguments.

Examples

```
>>> caller = sk.call(1, 2)
>>> caller(op.add), caller(op.mul)
(3, 2)
```

This function can be used as a decorator to declare self calling functions:

```
>>> @sk.call()
... def patch_module():
...     import builtins
...
...     builtins.evil = lambda: print('Evil patch')
...     return True
```

The variable `patch_module` will be assigned to the return value of the function and the function object itself will be garbage collected.

`sidekick.functions.`**`value`**

Evaluate argument, if it is a function or return it otherwise.

Parameters **`fn_or_value`** – Callable or some other value. If input is a callable, call it with the provided arguments and return. Otherwise, simply return.

Examples

```
>>> sk.value(42)
42
>>> sk.value(lambda: 42)
42
```

`sidekick.functions.`**`do`**

Runs `func` on `x`, returns `x`.

Because the results of `func` are not returned, only the side effects of `func` are relevant.

Logging functions can be made by composing `do` with a storage function like `list.append` or `file.write`

Examples

```
>>> log = []
>>> inc = sk.do(log.append) >> (X + 1)
>>> [inc(1), inc(11)]
[2, 12]
>>> log
[1, 11]
```

`sidekick.functions.`**`once`**

Limit function to a single invocation.

Repeated calls to the function return the value of the first invocation.

Examples

This is useful to wrap initialization routines or singleton factories. >>> @sk.once ... def configure(): ... print('setting up...') ... return {'status': 'ok'} >>> configure() setting up... {'status': 'ok'}

See also:

thunk() call_after() call_at_most()

sidekick.functions.**thunk**

A thunk that represents a lazy computation.

Python thunks are represented by zero-argument functions that compute the value of computation on demand and store it for subsequent invocations.

This function is designed to be used as a decorator.

Example

```
>>> @sk.thunk(host='localhost', port=5432)
... def db(host, port):
...     print(f'connecting to SQL server at {host}:{port}...')
...     return {'host': host, 'port': port}
>>> db()
connecting to SQL server at localhost:5432...
{'host': 'localhost', 'port': 5432}
>>> db()
{'host': 'localhost', 'port': 5432}
```

See also:

once()

sidekick.functions.**call_after**

Creates a function that invokes func once it's called more than n times.

Parameters

- **n** – Number of times before starting invoking n.
- **func** – Function to be invoked.
- **default** – Value returned before func() starts being called.

Example

```
>>> f = sk.call_after(2, (X * 2), default=0)
>>> [f(1), f(2), f(3), f(4), ...]
[0, 0, 6, 8, ...]
```

See also:

once() call_at_most()

sidekick.functions.**call_at_most**

Creates a function that invokes func while it's called less than n times. Subsequent calls to the created function return the result of the last func invocation.

Parameters

- **n** – The number of calls at which func is no longer invoked.
- **func** – Function to restrict.

Examples

```
>>> log = sk.call_at_most(2, print)
>>> log("error1"); log("error2"); log("error3"); log("error4")
error1
error2
```

See also:

once() call_after()

sidekick.functions.**throttle**

Limit the rate of execution of func to once at each `dt` seconds.

When rate-limited, returns the last result returned by func.

Example

```
>>> f = sk.throttle(1, (X * 2))
>>> [f(21), f(14), f(7), f(0)]
[42, 42, 42, 42]
```

sidekick.functions.**background**

Return a function that executes in the background.

The transformed function return a thunk that forces the evaluation of the function in a blocking manner. Function can also be used as a decorator.

Parameters

- **func** – Function or callable wrapped to support being called in the background.
- **timeout** – Timeout in seconds.
- **default** – Default value to return if if function timeout when evaluation is requested, otherwise, raises a TimeoutError.

Examples

```
>>> fib = lambda n: 1 if n <= 2 else fib(n - 1) + fib(n - 2)
>>> fib_bg = sk.background(fib, timeout=1.0)
>>> result = fib_bg(10)  # Do not block execution, return a thunk
>>> result()             # Call the result to get value (blocking operation)
55
```

sidekick.functions.**error**

Raises the given exception.

If argument is not an exception, raises ValueError(exc).

Examples

```
>>> sk.error('some error')
Traceback (most recent call last):
...
ValueError: some error
```

See also:

- *raising()*: create a function that raises an error instead of raising it immediately

sidekick.functions.**raising**

Creates function that raises the given exception.

If argument is not an exception, raises ValueError(exc). The returning function accepts any number of arguments.

Examples

```
>>> func = sk.raising('some error')
>>> func()
Traceback (most recent call last):
...
ValueError: some error
```

See also:

- *raising()*: create a function that raises an error instead of raising it immediately

sidekick.functions.**retry**

Retry to execute function at least n times before raising an error.

This is useful for functions that may fail due to interaction with external resources (e.g., fetch data from the network).

Parameters

- **n** – Maximum number of times to execute function
- **func** – Function that may raise errors.
- **error** – Exception or tuple with suppressed exceptions.
- **sleep** – Interval in which it sleeps between attempts.

Example

```
>>> queue = [111, 7, None, None]
>>> process = sk.retry(5, lambda n: queue.pop() * n)
>>> process(6)
42
```

sidekick.functions.**catch**

Handle exception in function. If the exception occurs, it executes the given handler.

Examples

```
>>> nan = sk.always(float('nan'))
>>> div = sk.catch(ZeroDivisionError, (X / Y), handler=nan)
>>> div(1, 0)
nan
```

The function can be used to re-write exceptions by passing the optional raises parameter.

```
>>> @sk.catch(KeyError, raises=ValueError("invalid name"))
... def get_value(name):
...     return data[name]
```

sidekick.functions.**flip**
Flip the order of arguments in a binary operator.

The resulting function is always curried.

Examples

```
>>> from operator import sub
>>> rsub = sk.flip(sub)
>>> rsub(2, 10)
8
```

sidekick.functions.**select_args**
Creates a function that calls func with the arguments reordered.

Examples

```
>>> double = sk.select_args([0, 0], (X + Y))
>>> double(21)
42
```

sidekick.functions.**keep_args**
Uses only the first n positional arguments to call func.

Examples

```
>>> incr = sk.keep_args(1, (X + 1))
>>> incr(41, 'whatever')
42
```

sidekick.functions.**reverse_args**
Creates a function that invokes func with the positional arguments order reversed.

Examples

```
>>> concat = sk.reverse_args(lambda x, y, z: x + y + z)
>>> concat("a", "b", "c")
'cba'
```

sidekick.functions.**skip_args**
Skips the first n positional arguments before calling func.

Examples

```
>>> incr = sk.skip_args(1, (X + 1))
>>> incr('whatever', 41)
42
```

sidekick.functions.**splice_args**

Return a function that receives a sequence as single argument and splice them into func.

Parameters

- **func** – Function that receives several positional arguments.
- **slice** – If given and is a slice, correspond to the slice in the input arguments that will be passed to func.

Example

```
>>> vsum = sk.splice_args(max)
>>> vsum([1, 2, 3, 4])
4
```

sidekick.functions.**variadic_args**

Return a function that receives variadic arguments and pass them as a tuple to func.

Parameters **func** – Function that receives a single tuple positional argument.

Example

```
>>> vsum = sk.variadic_args(sum)
>>> vsum(1, 2, 3, 4)
10
```

2.5 sidekick.op

This module exposes functionality analogous to the operator module in the standard lib. The main difference is that binary operators are curried and all functions are fn-functions. So we can use it as a drop-in replacement to the operator module:

```
>>> from sidekick import op
>>> op.add(1, 2)
3
```

But one that accepts additional idioms

```
>>> succ = op.add(1)
>>> succ(41)
42
```

2.5.1 API reference

fn-aware functions from the builtin operator module.

sidekick.op.**setitem**
Same as `a[b] = c`.

sidekick.op.**add**
Same as `a + b`.

sidekick.op.**and_**
Same as `a & b`.

sidekick.op.**concat**
Same as `a + b`, for a and b sequences.

sidekick.op.**contains**
Same as `b in a` (note reversed operands).

sidekick.op.**count_of**
Return the number of times b occurs in a.

sidekick.op.**delitem**
Same as `del a[b]`.

sidekick.op.**eq**
Same as `a == b`.

sidekick.op.**floordiv**
Same as `a // b`.

sidekick.op.**ge**
Same as `a >= b`.

sidekick.op.**getitem**
Same as `a[b]`.

sidekick.op.**gt**
Same as `a > b`.

sidekick.op.**iadd**
Same as `a += b`.

sidekick.op.**iand**
Same as `a &= b`.

sidekick.op.**iconcat**
Same as `a += b`, for a and b sequences.

sidekick.op.**ifloordiv**
Same as `a //= b`.

sidekick.op.**ilshift**
Same as `a <<= b`.

sidekick.op.**imod**
Same as `a %= b`.

sidekick.op.**imul**
Same as `a *= b`.

sidekick.op.**index_of**
Return the first index of b in a.

sidekick.op.**ior**
Same as `a |= b`.

sidekick.op.**ipow**
Same as `a **= b`.

sidekick.op.**irshift**
Same as `a >>= b`.

sidekick.op.**is_**
Same as `a is b`.

sidekick.op.**is_not**
Same as `a is not b`.

sidekick.op.**isub**
Same as `a -= b`.

sidekick.op.**itruediv**
Same as `a /= b`.

sidekick.op.**ixor**
Same as `a ^= b`.

sidekick.op.**le**
Same as `a <= b`.

sidekick.op.**lshift**
Same as `a << b`.

sidekick.op.**lt**
Same as `a < b`.

sidekick.op.**mod**
Same as `a % b`.

sidekick.op.**mul**
Same as `a * b`.

sidekick.op.**ne**
Same as `a != b`.

sidekick.op.**or_**
Same as `a | b`.

sidekick.op.**pow**
Same as `a ** b`.

sidekick.op.**rshift**
Same as `a >> b`.

sidekick.op.**sub**
Same as `a - b`.

sidekick.op.**div**
Same as `a / b`.

sidekick.op.**truediv**
Same as `a / b`.

sidekick.op.**xor**
Same as `a ^ b`.

sidekick.op.**abs**
Same as `abs(a)`.

sidekick.op.**index**
Same as `a.__index__()`.

sidekick.op.**inv**
Same as `~a`.

sidekick.op.**invert**
Same as `~a`.

sidekick.op.**neg**
Same as `-a`.

sidekick.op.**not_**
Same as `not a`.

sidekick.op.**pos**
Same as `+a`.

sidekick.op.**truth**
Return True if a is true, False otherwise.

sidekick.op.**matmul**
Same as `a @ b`.

sidekick.op.**imatmul**
Same as `a @= b`.

sidekick.op.**length_hint**
Return an estimate of the number of items in obj.

This is useful for presizing containers when building from an iterable.

If the object supports len(), the result will be exact. Otherwise, it may over- or under-estimate by an arbitrary amount. The result will be an integer >= 0.

sidekick.op.**attrgetter**
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

sidekick.op.**itemgetter**
itemgetter(item, ...) –> itemgetter object

Return a callable object that fetches the given item(s) from its operand. After f = itemgetter(2), the call f(r) returns r[2]. After g = itemgetter(2, 5, 3), the call g(r) returns (r[2], r[5], r[3])

sidekick.op.**methodcaller**
methodcaller(name, ...) –> methodcaller object

Return a callable object that calls the given method on its operand. After f = methodcaller('name'), the call f(r) returns r.name(). After g = methodcaller('name', 'date', foo=1), the call g(r) returns r.name('date', foo=1).

2.6 sidekick.pred

This module contains a collection of predicate functions that implement common tests. Since all functions are fn-enabled, it accepts bitwise logic to compound predicates into more interesting expressions, like in the example bellow.

```
>>> sk.filter(sk.is_odd & sk.is_positive, range(-5, 10))
sk.iter([1, 3, 5, 7, 9])
```

2.6.1 Composing predicates

cond	Conditional evaluation.
any_pred(*predicates)	Return a new predicate function that performs a logic OR to all arguments.
all_pred(*predicates)	Return a new predicate that performs an logic AND to all predicate functions.

2.6.2 Testing values

is_a	Check if x is an instance of cls.
is_equal	Check if `x == y`.
is_identical	Check if `x is y`.
is_false	Check if argument is Falseby identity.
is_true	Check if argument is Trueby identity.
is_none	Check if argument is Noneby identity.
is_truthy	Check if argument is truthy.
is_falsy	Check if argument is falsy.

2.6.3 Numeric tests

is_even	Check if argument is even.
is_odd	Check if argument is odd.
is_negative	Check if argument is negative.
is_positive	Check if argument is positive.
is_strictly_negative	Check if argument is strictly negative.
is_strictly_positive	Check if argument is strictly positive.
is_zero	Check if argument is zero.
is_nonzero	Check if argument is nonzero.
is_divisible_by	Check if x is divisible by n.

2.6.4 Sequences

is_distinct	Test if all elements in sequence are distinct.
is_iterable	Test if argument is iterable.
is_seq_equal	Return True if the two sequences are equal.

2.6.5 String tests

has_pattern(pattern[, st])	Check if string contains pattern.

2.6.6 API reference

sidekick.pred.**cond**
Conditional evaluation.

Return a function that tests the argument with the cond function, and then executes either the `then` or `else_`

branches.

Examples

```
>>> collatz = sk.cond(sk.is_even, _ // 2, (3 * _) + 1)
>>> [collatz(1), collatz(2), collatz(3), collatz(4)]
[4, 1, 10, 2]
```

sidekick.pred.**any_pred**(**predicates*)
Return a new predicate function that performs a logic OR to all arguments.

This behaves in a short-circuit manner and returns the first truthy result, if found, or the last falsy result, otherwise.

sidekick.pred.**all_pred**(**predicates*)
Return a new predicate that performs an logic AND to all predicate functions.

This behaves in a short-circuit manner and returns the first falsy result, if found, or the last truthy result, otherwise.

sidekick.pred.**is_a**
Check if x is an instance of cls.

Equivalent to isinstance, but auto-curried and with the order or arguments flipped.

Parameters

- **cls** – Type or tuple of types to test for.
- **x** – Instance.

Examples

```
>>> is_int = sk.is_a(int)
>>> is_int(42), is_int(42.0)
(True, False)
```

sidekick.pred.**is_equal**
Check if `x == y`.

sidekick.pred.**is_identical**
Check if `x is y`.

sidekick.pred.**is_false**
Check if argument is Falseby identity.

Examples

```
>>> sk.is_false("not_false")
False
>>> sk.is_false(False)
True
```

sidekick.pred.**is_true**
Check if argument is Trueby identity.

Examples

```
>>> sk.is_true("not_true")
False
>>> sk.is_true(True)
True
```

sidekick.pred.**is_none**
Check if argument is Noneby identity.

Examples

```
>>> sk.is_none("not_none")
False
>>> sk.is_none(None)
True
```

sidekick.pred.**is_truthy**
Check if argument is truthy.

This is the same as calling `bool(x)`

sidekick.pred.**is_falsy**
Check if argument is falsy.

This is the same as calling `not bool(x)`

sidekick.pred.**is_even**
Check if argument is even.

Examples

```
>>> sk.is_even(1)
False
>>> sk.is_even(42)
True
```

sidekick.pred.**is_odd**
Check if argument is odd.

Examples

```
>>> sk.is_odd(42)
False
>>> sk.is_odd(1)
True
```

sidekick.pred.**is_negative**
Check if argument is negative.

Examples

```
>>> sk.is_negative(42)
False
>>> sk.is_negative(-10)
True
```

sidekick.pred.**is_positive**
Check if argument is positive.

Examples

```
>>> sk.is_positive(-10)
False
>>> sk.is_positive(42)
True
```

sidekick.pred.**is_strictly_negative**
Check if argument is strictly negative.

Examples

```
>>> sk.is_strictly_negative(0)
False
>>> sk.is_strictly_negative(-10)
True
```

sidekick.pred.**is_strictly_positive**
Check if argument is strictly positive.

Examples

```
>>> sk.is_strictly_positive(0)
False
>>> sk.is_strictly_positive(42)
True
```

sidekick.pred.**is_zero**
Check if argument is zero.

Examples

```
>>> sk.is_zero(42)
False
>>> sk.is_zero(0)
True
```

sidekick.pred.**is_nonzero**
Check if argument is nonzero.

Examples

```
>>> sk.is_nonzero(0)
False
>>> sk.is_nonzero(42)
True
```

sidekick.pred.**is_divisible_by**
Check if x is divisible by n.

Examples

```
>>> sk.is_divisible_by(2, 42)  # 42 is divisible by 2
True
>>> even = sk.is_divisible_by(2)
>>> even(4), even(3)
(True, False)
```

sidekick.pred.**has_pattern**(*pattern*, *st=NOT_GIVEN*)
Check if string contains pattern.

This function performs a pattern scan. If you want to match the beginning of the string, make it start with a "^". Add a "$" if it needs to match the end.

Examples

```
>>> sk.has_pattern("\d{2}", "year, 1942")
True
>>> sk.has_pattern("^\d{2}$", "year, 1942")
False
```

This function is also very useful to filter or process string data.

```
>>> is_date = sk.has_pattern("^\d{4}-\d{2}-\d{2}$")
>>> is_date("1917-03-08")
True
>>> is_date("08/03/1917")
False
```

sidekick.pred.**is_distinct**
Test if all elements in sequence are distinct.

sidekick.pred.**is_iterable**
Test if argument is iterable.

sidekick.pred.**is_seq_equal**
Return True if the two sequences are equal.

2.7 sidekick.properties

Functions in this module are helpers intended to create convenient and declarative idioms when declaring classes. Perhaps it is not entirely correct calling them "functional", but since some patterns such as lazy properties are common in functional libraries, Sidekick has a module for doing that.

2.7.1 Properties and descriptors

lazy([func])	Mark attribute that is initialized at first access rather than during instance creation.
alias(attr, *, mutable, transform, prepare)	An alias to another attribute.
delegate_to(attr, mutable)	Delegate access to an inner variable.
property([fget, fset, fdel, doc])	A Sidekick-enabled property descriptor.

2.7.2 API reference

sidekick.properties.**lazy**(*func=None, *, shared: bool = False, name: str = None, attr_error: Union[Type[Exception], bool] = False*)

Mark attribute that is initialized at first access rather than during instance creation.

Usage is similar to `@property`, although lazy attributes do not override *setter* and *deleter* methods, allowing instances to write to the attribute.

Optional Args:

shared: A shared attribute behaves as a lazy class variable that is shared among all classes and instances. It differs from a simple class attribute in that it is initialized lazily from a function. This can help to break import cycles and delay expensive global initializations to when they are required.

name: By default, a lazy attribute can infer the name of the attribute it refers to. In some exceptional cases (when creating classes dynamically), the inference algorithm might fail and the name attribute must be set explicitly.

attr_error (Exception, bool): If False or an exception class, re-raise attribute errors as the given error. This prevent erroneous code that raises AttributeError being mistakenly interpreted as if the attribute does not exist.

Examples

```
import math

class Vec:
    @sk.lazy
    def magnitude(self):
        print('computing...')
        return math.sqrt(self.x**2 + self.y**2)

    def __init__(self, x, y):
        self.x, self.y = x, y
```

Now the `magnitude` attribute is initialized and cached upon first use:

```
>>> v = Vec(3, 4)
>>> v.magnitude
computing...
5.0
```

The attribute is writable and apart from the deferred initialization, it behaves just like any regular Python attribute.

```
>>> v.magnitude = 42
>>> v.magnitude
42
```

Lazy attributes can be useful either to simplify the implementation of `__init__` or as an optimization technique that delays potentially expensive computations that are not always necessary in the object's lifecycle. Lazy attributes can be used together with quick lambdas for very compact definitions:

```
import math
from sidekick import placeholder as _

class Square:
    area = sk.lazy(_.width * _.height)
    perimeter = sk.lazy(2 * (_.width + _.height))
```

sidekick.properties.**delegate_to**(*attr: str*, *mutable: bool = False*)

Delegate access to an inner variable.

A delegate is an alias for an attribute of the same name that lives in an inner object of an instance. This is useful when the inner object contains the implementation (remember the "composition over inheritance mantra"), but we want to expose specific interfaces of the inner object.

Parameters

- **attr** – Name of the inner variable that receives delegation. It can be a dotted name with one level of nesting. In that case, it associates the property with the sub-attribute of the delegate object.
- **mutable** – If True, makes the the delegation read-write. It writes new values to attributes of the delegated object.

Examples

```
class Queue:
    pop = sk.delegate_to('_data')
    push = sk.delegate_to('_data.append')

    def __init__(self, data=()):
        self._data = list(data)

    def __repr__(self):
        return f'Queue({self._data})'
```

Now `Queue.pop` simply redirects to the pop method of the `._data` attribute, and `Queue.push` searches for `._data.append`

```
>>> q = Queue([1, 2, 3])
>>> q.pop()
3
>>> q.push(4); q
Queue([1, 2, 4])
```

sidekick.properties.**alias**(*attr: str*, ***, *mutable: bool = False*, *transform: Optional[Callable] = None*, *prepare: Optional[Callable] = None*)

An alias to another attribute.

Aliasing is another simple form of self-delegation. Aliases are views over other attributes in the instance itself:

Parameters

- **attr** – Name of aliased attribute.
- **mutable** – If True, makes the alias mutable.
- **transform** – If given, transform output by this function before returning.
- **prepare** – If given, prepare input value before saving.

Examples

```
class Queue(list):
    push = sk.alias('pop')
```

This exposes two additional properties: "abs_value" and "origin". The first is just a read-only view on the "magnitude" property. The second exposes read and write access to the "start" attribute.

sidekick.properties.**property**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

A Sidekick-enabled property descriptor.

It is a drop-in replacement for Python's builtin properties. It behaves similarly to Python's builtin, but also accepts quick lambdas as input functions. This allows very terse declarations:

```
from sidekick.api import placeholder as _

class Vector:
    sqr_radius = sk.property(_.x**2 + _.y**2)
```

lazy() is very similar to property. The main difference between both is that properties are not cached and hence the function is re-executed at each attribute access. The desired behavior will depend a lot on what you want to do.

2.8 sidekick.proxy

Sidekick's *sidekick.functions.thunk()* is a nice way to represent a lazy computation through a function. It, however, breaks existing interfaces since we need to call the result of thunk to obtain its inner value. *sidekick.proxy* implements a few types that help exposing lazy objects as proxies, sharing the same interfaces as the wrapped objects. This is great for code that relies in duck-typing.

Lazy objects are useful both to declare objects that use yet uninitialized resources and as an optimization that we like to call *"opportunistic procrastination"*: we delay computation to the last possible time in the hope that we can get away without doing it at all. This is a powerful strategy since our programs tend to compute a lot of things in advance that are not always used during program execution.

2.8.1 Proxy types and functions

Proxy(obj)	Base class for proxy types.
deferred(func, *args, **kwargs)	Wraps uninitialized object into a proxy shell.
zombie(*args, **kwargs)	Provides zombie[class] syntax.
touch(obj)	Return a non-proxy, non-zombie version of object.
import_later(path[, package])	Lazily import module or object.

2.8.2 Deferred proxies vs zombies

Sidekick provides two similar kinds of deferred objects: *deferred* and *zombie()*. They are both initialized from a callable with arbitrary arguments and delay the execution of that callable until the result is needed. Consider the custom `User` class bellow.

```
>>> class User:
...     def __init__(self, **kwargs):
...         for k, v in kwargs.items():
...             setattr(self, k, v)
...     def __repr__(self):
...         data = ('%s: %r' % item for item in self.__dict__.items())
...         return 'User(%s)' % ', '.join(data)
>>> a = sk.deferred(User, name='Me', age=42)
>>> b = sk.zombie(User, name='Me', age=42)
```

The main difference between deferred and zombie, is that Zombie instances assume the type of the result after they awake, while deferred objects are proxies that simply mimic the interface of the result.

```
>>> a
Proxy(User(name: 'Me', age: 42))
>>> b
User(name: 'Me', age: 42)
```

We can see that deferred instances do not change class, while zombie instances do:

```
>>> type(a), type(b)                                    # doctest: +ELLIPSIS
(<class '...deferred'>, <class 'User'>)
```

This limitation makes zombie objects somewhat restricted. Delayed execution cannot return any type that has a different C layout as regular Python objects. This excludes all builtin types, C extension classes and even Python classes that define __slots__. On the plus side, zombie objects fully assume the properties of the delayed execution, including its type and can replace them in almost any context.

A slightly safer version of *zombie()* allows specifying the return type of the resulting object. This opens zombies up to a few additional types (e.g., types that use __slots__) and produces checks if conversion is viable or not.

We specify the return type as an index before declaring the constructor function:

```
>>> rec = sk.zombie[sk.record](sk.record, x=1, y=2)
>>> type(rec)                                           # doctest: +ELLIPSIS
<class '...SpecializedZombie'>
```

Touch it, and the zombie awakes

```
>>> sk.touch(rec)
record(x=1, y=2)
```

```
>>> type(rec)                                           # doctest: +ELLIPSIS
<class '...record'>
```

2.8.3 API reference

class sidekick.proxy.**Proxy**(*obj*)

Base class for proxy types.

class sidekick.proxy.**deferred**(*func*, **args*, ***kwargs*)
Wraps uninitialized object into a proxy shell.

Object is declared as a thunk and is initialized the first time some attribute or method is requested.

The proxy delegates all methods to the lazy object. Proxies work nicely with duck typing, but are a poor fit to code that relies in explicit instance checks since the deferred object is a *Proxy* instance.

Usage:

```
>>> from operator import add
>>> x = sk.deferred(add, 40, 2)  # add function not called yet
>>> print(x)                     # any interaction triggers object␣
↪construction!
42
```

class sidekick.proxy.**zombie**(**args*, ***kwargs*)
Provides zombie[class] syntax.

Implementation is in the metaclass.

sidekick.proxy.**touch**(*obj*)
Return a non-proxy, non-zombie version of object. Regular objects are returned as-is.

sidekick.proxy.**import_later**(*path*, *package=None*)
Lazily import module or object.

Lazy imports can dramatically decrease the initialization time of your python modules, specially when heavy weights such as numpy, and pandas are used. Beware that import errors that normally are triggered during import time now can be triggered at first use, which may introduce confusing and hard to spot bugs.

Parameters

- **path** – Python path to module or object. Modules are specified by their Python names (e.g., 'os.path') and objects are identified by their module path + ":" + object name (e.g., "os.path.splitext").
- **package** – Package name if path is a relative module path.

Examples

```
>>> np = sk.import_later('numpy')  # Numpy is not yet imported
>>> np.array([1, 2, 3])  # It imports as soon as required
array([1, 2, 3])
```

It also accepts relative imports if the package keyword is given. This is great to break circular imports.

```
>>> mod = sk.import_later('.sub_module', package=__package__)
```

2.9 sidekick.seq

2.9.1 Basic types

iter	Base sidekick iterator class.

Continued on next page

Table 16 – continued from previous page

generator	Decorates generator function to return a sidekick iterator instead of a regular Python generator.

2.9.2 Basic manipulation of sequences

cons	Construct operation.
uncons	De-construct sequence.
first	Return the first element of sequence.
second	Return the second element of sequence.
nth	Return the nth element in a sequence.
find	Return the (position, value) of first element in which predicate is true.
only	Return the single element of sequence or raise an error.
last	Return last item (or items) of sequence.
is_empty	Return True if sequence is empty.
length	Return length of sequence, consuming the iterator.
consume	Consume iterator for its side-effects and return last value or None.

2.9.3 Creating new sequences

cycle	Return elements from the iterable until it is exhausted.
iterate	Repeatedly apply a function func to input.
iterate_indexed	Similar to *iterate()*, but also pass the index of element to func.
repeat	for the specified number of times.
repeatedly	Make infinite calls to a function with the given arguments.
singleton	Return iterator with a single object.
unfold	Invert a fold.

2.9.4 Filtering and select sub-sequences

filter	Return an iterator yielding those items of iterable for which function(item) is true.
remove	Opposite of filter.
separate	Split sequence it two.
drop	Drop items from the start of iterable.
rdrop	Drop items from the end of iterable.
take	Return the first entries iterable.
rtake	Return the last entries iterable.
unique	Returns the given sequence with duplicates removed.
dedupe	Remove duplicates of successive elements.
converge	Test convergence with the predicate function by passing the last two items of sequence.

2.9.5 Grouping items

group_by	Group collection by the results of a key function.
chunks	Partition sequence into non-overlapping tuples of length n.
chunks_by	Partition sequence into chunks according to a function.
window	Return a sequence of overlapping sub-sequences of size n.
pairs	Returns an iterator of a pair adjacent items.
partition	Partition sequence in two.
distribute	Distribute items of seq into n different sequences.

2.9.6 Reducing sequences

fold	Perform a left reduction of sequence.
reduce	Like fold, but does not require initial value.
scan	Returns a sequence of the intermediate folds of seq by func.
acc	Like *scan()*, but uses first item of sequence as initial value.
fold_by	Reduce each sequence generated by a group by.
reduce_by	Similar to fold_by, but only works on non-empty sequences.
fold_together	Folds using multiple functions simultaneously.
reduce_together	Similar to fold_together, but only works on non-empty sequences.
scan_together	Folds using multiple functions simultaneously.
acc_together	Similar to fold_together, but only works on non-empty sequences.
product	Multiply all elements of sequence.
products	Return a sequence of partial products.
sum	Sum all arguments of sequence.
sums	Return a sequence of partial sums.
all_by	Return True if all elements of seq satisfy predicate.
any_by	Return True if any elements of seq satisfy predicate.
top_k	Find the k largest elements of a sequence.

2.9.7 API reference

class sidekick.seq.**iter**

Base sidekick iterator class.

This class extends classical Python iterators with a few extra operators. Sidekick iterators accepts slicing, indexing, concatenation (with the + sign) repetition (with the * sign) and pretty printing.

Operations that return new iterators (e.g., slicing, concatenation, etc) consume the data stream. Operations that simply peek at data execute the generator (and thus may produce side-effects), but cache values and do not consume data stream.

copy() → sidekick.seq.iter.iter

Return a copy of iterator. Consuming the copy do not consume the original iterator.

Internally, this method uses itertools.tee to perform the copy. If you known that the iterator will be consumed, it is faster and more memory efficient to convert it to a list and produce multiple iterators.

peek (*n: int*) → Tuple
Peek the first n elements without consuming the iterator.

tee (*n=1*) → Tuple[sidekick.seq.iter.iter]
Split iterator into n additional copies.

The copy method is simply an alias to iter.tee(1)[0]

sidekick.seq.**generator**
Decorates generator function to return a sidekick iterator instead of a regular Python generator.

Examples

```
>>> @sk.generator
... def fibonacci():
...     x = y = 1
...     while True:
...         yield x
...         x, y = y, x + y
>>> fibonacci()
sk.iter([1, 1, 2, 3, 5, 8, ...])
```

sidekick.seq.**cons**
Construct operation. Add x to beginning of sequence.

Examples

```
>>> sk.cons(0, range(1, 6))
sk.iter([0, 1, 2, 3, 4, 5])
```

See also:

uncons()

sidekick.seq.**uncons**
De-construct sequence. Return a pair of (`first`, `*rest`) of sequence.

If default is given and if seq is an empty sequence return (default, empty_sequence), otherwise raise a ValueError.

Examples

```
>>> head, tail = sk.uncons(range(6))
>>> head
0
>>> tail
sk.iter([1, 2, 3, 4, 5])
```

See also:

cons() first()

sidekick.seq.**first**
Return the first element of sequence.

Raise ValueError or return the given default if sequence is empty.

Examples

```
>>> sk.first("abcd")
'a'
```

See also:

second() last() nth()

sidekick.seq.**second**
Return the second element of sequence.

Raise ValueError or return the given default if sequence has last than 2 elements.

Examples

```
>>> sk.second("abcd")
'b'
```

See also:

first() last() nth()

Notes

There is no third, fourth, etc, because we can easily create those functions using nth(n). Sidekick implements first/second to help selecting items of a pair, which tends to appear frequently when working with dictionaries.

sidekick.seq.**nth**
Return the nth element in a sequence.

Return the default if sequence is not large enough or raise a ValueError if default is not given.

Warning: If seq is an iterator, consume the first n items.

Examples

```
>>> sk.nth(2, "abcd")
'c'
```

See also:

first() second() last()

sidekick.seq.**find**
Return the (position, value) of first element in which predicate is true.

Raise ValueError if sequence is exhausted without a match.

Examples

```
>>> sk.find((X == 11), [2, 3, 5, 7, 11, 13, 17])
(4, 11)
```

sidekick.seq.**only**

Return the single element of sequence or raise an error.

Parameters

- **seq** – Input sequence.
- **default** – Optional default value, returned if sequence is empty.

Examples

```
>>> sk.only([42])
42
>>> sk.only([], default=42)
42
>>> sk.only([42, 43])
Traceback (most recent call last):
...
ValueError: sequence is too long
```

sidekick.seq.**last**

Return last item (or items) of sequence.

Parameters

- **seq** – Input sequence
- **default** – If given, and sequence is empty, return it. An empty sequence with no default value raises a ValueError.
- **n** – If given, return a tuple with the last n elements instead. If default is given and sequence is not large enough, fill it with the value, otherwise raise a ValueError.

Examples

```
>>> sk.last("abcd")
'd'
```

Notes

If you don't want to raise errors if sequence is not large enough, use *rtake()*:

```
>>> tuple(sk.rtake(5, "abc"))  # No error!
('a', 'b', 'c')
>>> sk.last("abc", n=5, default="-")
('-', '-', 'a', 'b', 'c')
```

See also:

first() second() nth() rtake()

sidekick.seq.**is_empty**
Return True if sequence is empty.

> **Warning:** This function consume first element of iterator. Use this only to assert that some iterator was consumed without using it later or create a copy with itertools.tee that will preserve the consumed element.

Examples

```
>>> nums = iter(range(5))
>>> sum(nums)  # exhaust iterator
10
>>> sk.is_empty(nums)
True
```

sidekick.seq.**length**
Return length of sequence, consuming the iterator.

If limit is given, consume sequence up to the given limit. This is useful to test if sequence is longer than some given size but without consuming the whole iterator if so.

Examples

```
>>> sk.length(range(10))
10
```

sidekick.seq.**consume**
Consume iterator for its side-effects and return last value or None.

Parameters

- **seq** – Any iterable
- **default** – Fallback value returned for empty sequences.

Examples

```
>>> it = map(print, [1, 2, 3])
>>> sk.consume(it)
1
2
3
```

sidekick.seq.**cycle**
Return elements from the iterable until it is exhausted. Then repeat the sequence indefinitely.

cycle(seq) ==> seq[0], seq[1], . . . , seq[n - 1], seq[0], seq[1], . . .

Examples

```
>>> sk.cycle([1, 2, 3])
sk.iter([1, 2, 3, 1, 2, 3, ...])
```

sidekick.seq.**iterate**
Repeatedly apply a function func to input.

If more than one argument to func is passed, it iterate over the past n values. It requires at least one argument, if you need to iterate a zero argument function, call *repeatedly()*

Iteration stops if if func() raise StopIteration.

iterate(f, x) ==> x, f(x), f(f(x)), ...

Examples

Simple usage, with a single argument. Produces powers of two.

```
>>> sk.iterate((X * 2), 1)
sk.iter([1, 2, 4, 8, 16, 32, ...])
```

Now we call with two arguments to func to produce Fibonacci numbers

```
>>> sk.iterate((X + Y), 1, 1)
sk.iter([1, 1, 2, 3, 5, 8, ...])
```

See also:

repeatedly()

sidekick.seq.**iterate_indexed**
Similar to *iterate()*, but also pass the index of element to func.

iterate_indexed(f, x) ==> x, f(0, x), f(1, <previous>), ...

Parameters

- **func** – Iteration function (index, value) -> next_value.
- **x** – Initial value of iteration.
- **idx** – Sequence of indexes. If not given, uses start, start + 1, ...
- **start** – Starting value for sequence of indexes.

Examples

```
>>> sk.iterate_indexed(lambda i, x: i * x, 1, start=1)
sk.iter([1, 1, 2, 6, 24, 120, ...])
```

sidekick.seq.**repeat**
for the specified number of times. If not specified, returns the object endlessly.

Examples

```
>>> sk.repeat(42, times=5)
sk.iter([42, 42, 42, 42, 42])
```

sidekick.seq.**repeatedly**
Make infinite calls to a function with the given arguments.

Stop iteration if func() raises StopIteration.

Examples

```
>>> sk.repeatedly(list, (1, 2))
sk.iter([[1, 2], [1, 2], [1, 2], [1, 2], [1, 2], ...])
```

sidekick.seq.**singleton**
Return iterator with a single object.

Examples

```
>>> sk.singleton(42)
sk.iter([42])
```

sidekick.seq.**unfold**
Invert a fold.

Similar to iterate, but expects a function of seed -> (seed', x). The second value of the tuple is included in the resulting sequence while the first is used to seed func in the next iteration. Stops iteration if func returns None or raise StopIteration.

Examples

```
>>> sk.unfold(lambda x: (x + 1, x), 0)
sk.iter([0, 1, 2, 3, 4, 5, ...])
```

sidekick.seq.**filter**
Return an iterator yielding those items of iterable for which function(item) is true.

Behaves similarly to Python's builtin filter, but accepts anything convertible to callable using *sidekick.functions.to_callable()* as predicate and return sidekick iterators instead of regular ones.

filter(pred, seq) ==> seq[a], seq[b], seq[c], ...

in which a, b, c, ... are the indexes in which pred(seq[i]) == True.

Examples

```
>>> sk.filter((X % 2), range(10))
sk.iter([1, 3, 5, 7, 9])
```

See also:

remove() separate()

sidekick.seq.**remove**
Opposite of filter. Return those items of sequence for which pred(item) is False

Examples

```
>>> sk.remove((X < 5), range(10))
sk.iter([5, 6, 7, 8, 9])
```

See also:

filter(). separate()

sidekick.seq.**separate**

Split sequence it two. The first consists of items that pass the predicate and the second of those items that don't.

Similar to (filter(pred, seq), filter(!pred, seq)), but only evaluate the predicate once per item.

Parameters

- **pred** – Predicate function
- **seq** – Iterable of items that should be separated.
- **consume** – If given, fully consume the iterator and return two lists. This is faster than separating and then converting each element to a list.

Examples

```
>>> sk.separate((X % 2), [1, 2, 3, 4, 5])
(sk.iter([1, 3, 5]), sk.iter([2, 4]))
```

See also:

filter() remove()

sidekick.seq.**drop**

Drop items from the start of iterable.

If key is a number, drop at most this number of items for iterator. If it is a predicate, drop items while key(item) is true.

drop(key, seq) ==> seq[n], seq[n + 1], ...

n is either equal to key, if its a number or is the index for the first item in which key(item) is false.

Examples

```
>>> sk.drop((X < 5), range(10))
sk.iter([5, 6, 7, 8, 9])
```

```
>>> sk.drop(3, range(10))
sk.iter([3, 4, 5, 6, 7, 8, ...])
```

See also:

take() rdrop()

sidekick.seq.**rdrop**

Drop items from the end of iterable.

Examples

```
>>> sk.rdrop(2, [1, 2, 3, 4])
sk.iter([1, 2])
```

```
>>> sk.rdrop((X >= 2), [1, 2, 4, 1, 2, 4])
sk.iter([1, 2, 4, 1])
```

See also:

drop() rtake()

sidekick.seq.**take**
Return the first entries iterable.

If key is a number, return at most this number of items for iterator. If it is a predicate, return items while key(item) is true.

take(key, seq) ==> seq[0], seq[1], . . . , seq[n - 1]

n is either equal to key, if its a number or is the index for the first item in which key(item) is false.

This function is a complement of *drop()*. Given two identical iterators `seq1` and `seq2`, `take(key, seq1) + drop(key, seq2)` yields the original sequence of items.

Examples

```
>>> sk.take((X < 5), range(10))
sk.iter([0, 1, 2, 3, 4])
```

See also:

drop()

sidekick.seq.**rtake**
Return the last entries iterable.

Examples

```
>>> sk.rtake(2, [1, 2, 3, 4])
sk.iter([3, 4])
```

```
>>> sk.rtake((X >= 2), [1, 2, 4, 1, 2, 4])
sk.iter([2, 4])
```

See also:

take() rdrop()

sidekick.seq.**unique**
Returns the given sequence with duplicates removed.

Preserves order. If key is supplied map distinguishes values by comparing their keys.

Parameters

- **seq** – Iterable of objects.
- **key** – Optional key function. It will return only the first value that evaluate to a unique key by the key function.
- **exclude** – Optional sequence of keys to exclude from seq
- **slow** – If True, allows the slow path (i.e., store seen elements in a list, instead of a hash).

Examples

```
>>> sk.unique(range(10), key=(X % 5))
sk.iter([0, 1, 2, 3, 4])
```

Note: Elements of a sequence or their keys should be hashable, otherwise it must use a slow path.

See also:

dedupe()

sidekick.seq.**dedupe**
Remove duplicates of successive elements.

Parameters

- **seq** – Iterable of objects.
- **key** – Optional key function. It will yield successive values if their keys are different.

See also:

unique()

sidekick.seq.**converge**
Test convergence with the predicate function by passing the last two items of sequence. If pred(penultimate, last) returns True, stop iteration.

Examples

We start with a converging (possibly infinite) sequence and an explicit criteria.

```
>>> seq = sk.iterate((X / 2), 1.0)
>>> conv = lambda x, y: abs(x - y) < 0.01
```

Run it until convergence

```
>>> it = sk.converge(conv, seq); it
sk.iter([1.0, 0.5, 0.25, 0.125, 0.0625, 0.03125, ...])
>>> sum(it)
1.9921875
```

sidekick.seq.**fold**
Perform a left reduction of sequence.

Examples

```
>>> sk.fold(op.add, 0, [1, 2, 3, 4])
10
```

See also:

reduce() scan()

sidekick.seq.**reduce**
Like fold, but does not require initial value.

This function raises a ValueError on empty sequences.

Examples

```
>>> sk.reduce(op.add, [1, 2, 3, 4])
10
```

See also:

fold() acc()

sidekick.seq.**scan**
Returns a sequence of the intermediate folds of seq by func.

In other words it generates a sequence like:

func(init, seq[0]), func(result[0], seq[1]), ...

in which result[i] corresponds to items in the resulting sequence.

See also:

acc() fold()

sidekick.seq.**acc**
Like *scan()*, but uses first item of sequence as initial value.

See also:

scan() reduce()

sidekick.seq.**reduce_by**
Similar to fold_by, but only works on non-empty sequences.

Initial value is taken to be the first element in sequence.

Examples

```
>>> sk.reduce_by(X % 2, op.add, [1, 2, 3, 4, 5])
{1: 9, 0: 6}
```

See also:

fold() group_by() reduce_by()

sidekick.seq.**fold_by**
Reduce each sequence generated by a group by.

More efficient than performing separate operations since it does not store intermediate groups.

Examples

```
>>> sk.fold_by(X % 2, op.add, 0, [1, 2, 3, 4, 5])
{1: 9, 0: 6}
```

See also:

fold() group_by() reduce_by()

sidekick.seq.**fold_together**
Folds using multiple functions simultaneously.

Examples

```
>>> seq = [1, 2, 3, 4, 5]
>>> sk.fold_together(seq, sum=((X + Y), 0), prod=((X * Y), 1))
{'sum': 15, 'prod': 120}
```

sidekick.seq.**reduce_together**
Similar to fold_together, but only works on non-empty sequences.

Initial value is taken to be the first element in sequence.

Examples

```
>>> seq = [1, 2, 3, 4, 5]
>>> sk.reduce_together(seq, sum=(X + Y), prod=(X * Y), max=max, min=min)
{'sum': 15, 'prod': 120, 'max': 5, 'min': 1}
```

sidekick.seq.**scan_together**
Folds using multiple functions simultaneously.

Initial value is passed as a tuple for each (operator, value) keyword argument.

Examples

```
>>> seq = [1, 2, 3, 4, 5]
>>> for acc in sk.scan_together(seq, sum=(X + Y, 0), prod=(X * Y, 1)):
...     print(acc)
{'sum': 0, 'prod': 1}
{'sum': 1, 'prod': 1}
{'sum': 3, 'prod': 2}
{'sum': 6, 'prod': 6}
{'sum': 10, 'prod': 24}
{'sum': 15, 'prod': 120}
```

sidekick.seq.**acc_together**
Similar to fold_together, but only works on non-empty sequences.

Initial value is taken to be the first element in sequence.

Examples

```
>>> seq = [1, 2, 3, 4, 5]
>>> for acc in sk.acc_together(seq, sum=(X + Y), prod=(X * Y)):
...     print(acc)
{'sum': 1, 'prod': 1}
{'sum': 3, 'prod': 2}
{'sum': 6, 'prod': 6}
{'sum': 10, 'prod': 24}
{'sum': 15, 'prod': 120}
```

sidekick.seq.**product**
Multiply all elements of sequence.

Examples

```
>>> sk.product([1, 2, 3, 4, 5])
120
```

See also:

products() sum()

sidekick.seq.**products**
Return a sequence of partial products.

Examples

```
>>> sk.products([1, 2, 3, 4, 5])
sk.iter([1, 2, 6, 24, 120])
```

See also:

acc() sums()

sidekick.seq.**sum**
Sum all arguments of sequence.

It exists only in symmetry with *product()*, since Python already has a builtin sum function that behaves identically.

Examples

```
>>> sk.sum([1, 2, 3, 4, 5])
15
```

See also:

sums() product()

sidekick.seq.**sums**
Return a sequence of partial sums.

Same as `sk.fold((X + Y), seq, 0)`

Examples

```
>>> sk.sums([1, 2, 3, 4, 5])
sk.iter([1, 3, 6, 10, 15])
```

See also:

acc() products()

sidekick.seq.**all_by**
Return True if all elements of seq satisfy predicate.

`all_by(None, seq)` is the same as the builtin `all(seq)` function.

Examples

```
>>> sk.all_by((X % 2), [1, 3, 5])
True
```

See also:

any_by()

sidekick.seq.**any_by**
Return True if any elements of seq satisfy predicate.

`any_by(None, seq)` is the same as the builtin `any(seq)` function.

Examples

```
>>> sk.any_by(sk.is_divisible_by(2), [2, 3, 5, 7, 11, 13])
True
```

See also:

all_by()

sidekick.seq.**top_k**
Find the k largest elements of a sequence.

Examples

```
>>> sk.top_k(3, "hello world")
('w', 'r', 'o')
```

sidekick.seq.**group_by**
Group collection by the results of a key function.

Examples

```
>>> sk.group_by((X % 2), range(5))
{0: [0, 2, 4], 1: [1, 3]}
```

See also:

reduce_by() fold_by()

sidekick.seq.**chunks**
Partition sequence into non-overlapping tuples of length n.

Parameters

- **n** – Number of elements in each partition. If n is a sequence, it selects partitions by the sequence size.
- **seq** – Input sequence.
- **pad** – If given, pad a trailing incomplete partition with this value until it has n elements.
- **drop** – If True, drop the last partition if it has less than n elements.

Examples

Too see the difference between padding, dropping and the regular behavior.

```
>>> sk.chunks(2, range(5))
sk.iter([(0, 1), (2, 3), (4,)])
```

```
>>> sk.chunks(2, range(5), pad=None)
sk.iter([(0, 1), (2, 3), (4, None)])
```

```
>>> sk.chunks(2, range(5), drop=True)
sk.iter([(0, 1), (2, 3)])
```

Using sequences, we can create more complicated chunking patterns.

```
>>> sk.chunks(sk.cycle([2, 3]), range(10))
sk.iter([(0, 1), (2, 3, 4), (5, 6), (7, 8, 9)])
```

A trailing ellipsis consumes the rest of the iterator.

```
>>> sk.chunks([1, 2, 3, ...], range(10))
sk.iter([(0,), (1, 2), (3, 4, 5), (6, 7, 8, 9)])
```

See also:

chunks_by() window()

sidekick.seq.**chunks_by**

Partition sequence into chunks according to a function.

It creates a new partition every time the value of func(item) changes.

Parameters

- **func** – Function used to control partition creation
- **seq** – Input sequence.
- **how** – Control how func is used to create new chunks from iterator.
  - 'values' (default): create a new chunk when func(x) changes value
  - **'pairs': create new chunk when func(x, y) for two successive values** is True.
  - **'left': create new chunk when func(x) is True. x is put in the** chunk to the left.
  - 'right': create new chunk when func(x) is True. x is put in the chunk to the right.
  - **'drop': create new chunk when func(x) is True. x dropped from output** sequence. It behaves similarly to str.split.

Examples

Standard chunker

```
>>> sk.chunks_by((X // 3), range(10))
sk.iter([(0, 1, 2), (3, 4, 5), (6, 7, 8), (9,)])
```

Chunk by pairs

```
>>> sk.chunks_by((Y <= X), [1, 2, 3, 2, 4, 8, 0, 1], how='pairs')
sk.iter([(1, 2, 3), (2, 4, 8), (0, 1)])
```

Chunk by predicate. The different versions simply define in which chunk the split location will be allocated

```
>>> sk.chunks_by(sk.is_odd, [1, 2, 3, 2, 4, 8], how='left')
sk.iter([(1,), (2, 3), (2, 4, 8)])
>>> sk.chunks_by(sk.is_odd, [1, 2, 3, 2, 4, 8], how='right')
sk.iter([(1, 2), (3, 2, 4, 8)])
>>> sk.chunks_by(sk.is_odd, [1, 2, 3, 2, 4, 8], how='drop')
sk.iter([(), (2,), (2, 4, 8)])
```

See also:

chunks() partition()

sidekick.seq.**window**

Return a sequence of overlapping sub-sequences of size n.

`n == 2` is equivalent to a pairwise iteration.

Examples

Pairwise iteration:

```
>>> [''.join(p) for p in sk.window(2, "hello!")]
['he', 'el', 'll', 'lo', 'o!']
```

See also:

pairs()

sidekick.seq.**pairs**

Returns an iterator of a pair adjacent items.

This is similar to window(2), but requires a fill value specified either with `prev` or `next` to select if it will form pairs with the preceeding of the following value.

It must specify either prev or next, never both.

Parameters

- **seq** – Input sequence.
- **prev** – If given, fill this value in the first item and iterate over all items preceded with the previous value.
- **next** – If given, fill this value in the last item and iterate over all items followed with the next value.

Examples

```
>>> [''.join(p) for p in sk.pairs("hello!", prev="-")]
['-h', 'he', 'el', 'll', 'lo', 'o!']
>>> [''.join(p) for p in sk.pairs("hello!", next="!")]
['he', 'el', 'll', 'lo', 'o!', '!!']
```

See also:

window()

sidekick.seq.**partition**
Partition sequence in two.

Returns a sequence with elements before and after the key separator.

Parameters

- **key** – An integer index or predicate used to partition sequence.
- **seq** – Input sequence.

Examples

```
>>> a, b = sk.partition(2, [5, 4, 3, 2, 1])
>>> a
sk.iter([5, 4])
>>> b
sk.iter([3, 2, 1])
```

```
>>> a, b = sk.partition((X == 3), [1, 2, 3, 4, 5])
>>> a
sk.iter([1, 2])
>>> b
sk.iter([3, 4, 5])
```

sidekick.seq.**distribute**
Distribute items of seq into n different sequences.

Parameters

- **n** – Number of output sequences.
- **seq** – Input sequences.

Examples

```
>>> a, b = sk.distribute(2, [0, 1, 2, 3, 4, 5, 6])
>>> a
sk.iter([0, 2, 4, 6])
>>> b
sk.iter([1, 3, 5])
```

2.10 References

2.10.1 Python Standard library

- itertools: The Python standard library for iterator tools.
- functools: The Python standard library for function tools.
- operator: Python operator as functions.

2.10.2 Other Python projects

These projects also provide functional utilities within Python. Their functionality sometimes overlaps with Sidekick, and they often can be used together very naturally.

- Toolz: Excellent library centered around iterators.
- Funcy: Library somewhat inspired in Underscore.js.
- Fn.py: Interesting library with some Haskelisms.
- More itertools: Extend itertools with many more functions.

2.10.3 Other influential libraries and programming languages

- Clojure: A functional language whose standard library has several counterparts in `sidekick`.
- Elixir: A functional language inspired both in Ruby and in Erlang
- Underscore.js: A very popular FP library for JavaScript

2.10.4 Other resources

- Awesome functional programming: List of links and other resources related to FP in Python.
- Functional Programming HOWTO: The description of functional programming features from the official Python docs.

2.11 Frequently asked questions

2.11.1 Usage

Why a new functional programming library?

Python is a multi-paradigm programming language and has some level of support of functional programming. That said, most APIs tend to encourage object oriented interfaces since that is what the language encourages and supports better. Sidekick aims to make functional programming viable in Python.

2.11.2 Concepts

Immutable data types

Immutable instances cannot change its value or internal state during the program execution. Python has some notable immutable types: numbers, strings, tuples, and others. In contrast, mutable data types can be modified. Good examples are lists and dictionaries.

2.12 License

2.13 Changelog and Roadmap

2.13.1 Roadmap and backlog

Sidekick is still a Beta-quality library in the sense that the API isn't stable and there are a few bugs lurking (but then, as with all code). Some parts of the library are stable and are very unlikely to change. If you plan to use Sidekick, please contact the authors, inform about your usage and contribute to this roadmap.

1.0.x

- Stabilize API

0.9.x

- Small changes with deprecation warnings, when possible.
- Define a deprecation path and implement helpers.

0.8.x

- Create sidekick.collections
- Better integration of maybe and result types
- Move Union/Record to sidekick.adt with other classical functional types
- Document all modules
- Remove cruft and clean the experimental package
- Remove all flake8 errors
- Continuous integration with strict tests

2.13.2 Changelog

0.8.3

- Document `sidekick.evil`, `sidekick.op`, ``sidekick.pred`.
- Build documentation cleanly and add it to the CI

0.8.2

- Remove the legacy `sidekick.functools` module.

0.8.1

- Split `sidekick.lazytools` into `proxy` and `properties` submodules.
- Rename properties parameters to be more consistent.
- Rework zombie[cls] factory to allow instance checks.

0.8.0

This is a major refactor preparing to 1.0.0

- Documentation now uses sphinx.ext.doctest instead of manuel and passes all tests.
- Moved fn and most of functools to sidekick.functions.
- Created sk.iter().
- Start changelog and roadmap.
- Refactor documentation
- Create the `import sidekick.api as sk` idiom.
- Move functions from `sidekick.functools` to `sidekick.functions`
- Move functions from `sidekick.itertools` to `sidekick.seq`

CHAPTER 3

Indices and tables

- genindex
- modindex

Python Module Index

S

Index

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z