pysweet-func

Nat Sothanaphan

Oct 19, 2022

OVERVIEW

Composable Python via syntactic sugar.

CHAPTER

ONE

PYSWEET-FUNC

1.1 Why pysweet?

Consider the following variants of the same logic:

```
acc = []

for x in range(10):
    y = x + 1

    if y % 2 == 0:
        acc.extend([y, y * 2])
```

```
acc = [
    z for y in (x + 1 for x in range(10))
    for z in [y, y * 2] if y % 2 == 0
]
```

```
from itertools import chain

acc = list(chain.from_iterable(map(
    lambda x: [x, x * 2],
    filter(
        lambda x: x % 2 == 0,
        map(lambda x: x + 1, range(10)),
    ),
)))
```

- The imperative style can grow complex as requirements evolve;
- The comprehension style can get complicated when nested;
- The functional style is not very readable in Python.

In JavaScript, the same logic can be written:

```
acc = [...Array(10).keys()]
    .map(x => x + 1)
```

(continues on next page)

(continued from previous page)

```
    .filter(x => x % 2 === 0)
    .flatMap(x => [x, x * 2])
```

Can we write analogous code in Python?

Now you can with `pysweet`!

```
from pysweet import Iterable_

acc = (
    Iterable_(range(10))
    .map(lambda x: x + 1)
    .filter(lambda x: x % 2 == 0)
    .flat_map(lambda x: [x, x * 2])
    .to_list()
)
```

`pysweet` also offers many other similar features.

`pysweet` is:

- lightweight, with around 100 lines of code;
- mostly syntactic sugar, so it is performant, easy to debug, and easy to learn;
- successfully used in production.

Sweeten Python with `pysweet`!

1.2 Sample features

- Iterable with method chaining, in the spirit of JavaScript and Scala:

```
from pysweet import Iterable_

(
    Iterable_([1, 2])
    .map(lambda x: x + 1)
    .to_list()
)
# [2, 3]
```

- Multi-expression lambda, common in modern languages:

```
from pysweet import block_

val = lambda: block_(
    x := 1,
    x + 1,
)
# val() == 2
```

- Statement as expression, in the spirit of Scala and Haskell (`if_` is also the ternary operator):

```
from pysweet import if_, try_, raise_

if_(
    True,
    lambda: 1,
    lambda: 2,
)
# 1

try_(
    lambda: raise_(Exception('test')),
    catch=lambda e: str(e),
)
# 'test'
```

1.3 Next steps

- Installation
- Documentation

CHAPTER

TWO

PYSWEET

2.1 pysweet.expression

pysweet.expression.**async_block_**(**expressions: Union[Callable[[Any], Any], _Await[Any, Any]]*) → Coroutine[Any, Any, Any]

Asynchronous code block evaluating expressions in order. Use `await_` to await a specific expression.

```
>>> from asyncio import sleep, run
...
>>> async def add_one(x):
...     await sleep(0.1)
...     return x + 1
...
>>> main = lambda x: async_block_(
...     await_(lambda _: add_one(x)),
...     lambda y: y * 2,
...     await_(add_one),
...     print,
... )
...
>>> run(main(1))
5
```

Parameters
***expressions** – Expressions.

Returns
Coroutine.

pysweet.expression.**async_try_**(*do: Callable[[], Coroutine[Any, Any, _S]]*, *catch: Callable[[Exception], Coroutine[Any, Any, _T]]*) → Coroutine[Any, Any, Union[_S, _T]]

Asynchronous `try` expression.

```
>>> from asyncio import run
>>> async def do():
...     return 1
>>> async def catch(e):
...     return 2
>>> run(async_try_(do, catch))
1
```

Parameters

- **do** – Asynchronous callback.
- **catch** – Asynchronous callback if `do` raises an exception.

Returns
Coroutine returning result of `do` or `catch`.

`pysweet.expression.`**`async_with_`**(*context: AsyncContextManager[_S]*, *do: Callable[[_S], Coroutine[Any, Any, _T]]*) → Coroutine[Any, Any, _T]

`async with` expression.

```
>>> from asyncio import Lock, run
>>> lock = Lock()
>>> async def main():
...     return 1
>>> run(async_with_(lock, lambda _: main()))
1
```

Parameters

- **context** – Asynchronous context manager.
- **do** – Asynchronous callback.

Returns
Coroutine running `do` in the context of `context`.

`pysweet.expression.`**`await_`**(*func: Callable[[_S], Coroutine[Any, Any, _T]]*) → _Await[_S, _T]

`await` expression. Only valid inside an `async_block_`.

Parameters
func – Asynchronous transform.

Returns
Internal `_Await` object.

`pysweet.expression.`**`block_`**(**expressions: Any*) → Any

Code block evaluating to the last expression.

```
>>> val = lambda: block_(
...     x := 1,
...     x + 1,
... )
>>> val()
2
```

Parameters
***expressions** – Expressions.

Returns
Last element of `expressions`.

`pysweet.expression.`**`if_`**(*condition: Any*, *then_do: Callable[[], _S]*, *else_do: Callable[[], _T]*) → Union[_S, _T]

`if` expression.

```
>>> if_(True, lambda: 1, lambda: 2)
1
```

Parameters

- **condition** – Condition.
- **then_do** – Callback if `condition` is truthy.
- **else_do** – Callback if `condition` is falsy.

Returns

Result of `then_do` or `else_do`.

`pysweet.expression.`**`raise_`**(*exception: Exception*) → NoReturn

`raise` expression.

```
>>> val = lambda: raise_(Exception('test'))
>>> val()
Traceback (most recent call last):
    ...
Exception: test
```

Parameters

exception – Exception.

Returns

No return.

`pysweet.expression.`**`try_`**(*do: Callable[[], _S]*, *catch: Callable[[Exception], _T]*) → Union[_S, _T]

`try` expression.

```
>>> val = lambda: try_(
...     lambda: 1,
...     catch=lambda e: 2,
... )
>>> val()
1
```

Parameters

- **do** – Callback.
- **catch** – Callback if `do` raises an exception.

Returns

Result of `do` or `catch`.

`pysweet.expression.`**`with_`**(*context: ContextManager[_S]*, *do: Callable[[_S], _T]*) → _T

`with` expression.

```
>>> from threading import Lock
>>> lock = Lock()
>>> with_(lock, lambda _: 1)
1
```

Parameters

- **context** – Context manager.
- **do** – Callback.

Returns

Result of `do` in the context of `context`.

2.2 pysweet.func

`pysweet.func.`**compose_**(**funcs: Callable[[Any], Any]*) → Callable[[Any], Any]

Compose single-argument functions, evaluting from left to right.

```
>>> compose_(lambda x: x + 1, lambda x: x * 2)(1)
4
```

Parameters

funcs – Functions.

Returns

Composed function.

`pysweet.func.`**pack_**(*func: Callable[[...], _T]*) → Callable[[tuple], _T]

Pack function arguments into a single `tuple`.

```
>>> list(map(pack_(lambda x, y: x + y), [(1, 2), (3, 4)]))
[3, 7]
```

Parameters

func – Function.

Returns

Function mapping `(x, y, ...)` to `func(x, y, ...)`.

2.3 pysweet.iterable

class `pysweet.iterable.`**Iterable_**(*it: Iterable[_A]*)

Bases: `Iterable[_A]`

`Iterable` with method chaining.

Parameters

it – Wrapped `Iterable`.

consume() → None

Iterate over self.

```
>>> Iterable_(range(3)).map(print).consume()
0
1
2
```

Returns
None.

extend(*it: Iterable[_B]*) → *Iterable_*[Union[_A, _B]]

Chain self with another `Iterable`, immutably.

```
>>> Iterable_(range(5)).extend([5, 6]).to_list()
[0, 1, 2, 3, 4, 5, 6]
```

Parameters
it – `Iterable`.

Returns
Extended `Iterable_`.

filter(*f: Callable[[_A], Any]*) → *Iterable_*[_A]

Filter `f` over self immutably.

```
>>> Iterable_(range(5)).filter(lambda x: x % 2 == 0).to_list()
[0, 2, 4]
```

Parameters
f – Function.

Returns
Filtered `Iterable_`.

flat_map(*f: Callable[[_A], Iterable[_B]]*) → *Iterable_*[_B]

Map `f` over self and chain results, immutably.

```
>>> Iterable_(range(5)).flat_map(lambda x: [x, x + 1]).to_list()
[0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
```

Parameters
f – Function.

Returns
Flat-mapped `Iterable_`.

map(*f: Callable[[_A], _B]*) → *Iterable_*[_B]

Map `f` over self immutably.

```
>>> Iterable_(range(5)).map(lambda x: x * 2).to_list()
[0, 2, 4, 6, 8]
```

Parameters
f – Function.

Returns
Mapped `Iterable_`.

pipe(*f: Callable[[Iterable[_A]], Iterable[_B]]*) → *Iterable_*[_B]

Transform self with `f` immutably.

```
>>> Iterable_([0, 1, 2]).pipe(lambda x: x + x).val
[0, 1, 2, 0, 1, 2]
```

Parameters
f – Function.

Returns
Transformed `Iterable_`.

to_dict() → dict

Unwrap underlying `Iterable` as a `dict`.

```
>>> Iterable_([('a', 1), ('b', 2)]).to_dict()
{'a': 1, 'b': 2}
```

Returns
Wrapped `Iterable` converted to `dict`.

to_list() → List[_A]

Unwrap underlying `Iterable` as a `list`.

```
>>> Iterable_(range(5)).to_list()
[0, 1, 2, 3, 4]
```

Returns
Wrapped `Iterable` converted to `list`.

property val: Iterable[_A]

Get underlying `Iterable`.

```
>>> Iterable_([0, 1, 2]).val
[0, 1, 2]
```

Returns
Wrapped `Iterable`.

zip() → *Iterable_*[tuple]

Zip self immutably.

```
>>> Iterable_(dict(a=1, b=2).items()).zip().to_list()
[('a', 'b'), (1, 2)]
```

Returns
Zipped `Iterable_`.

2.4 pysweet.types

2.5 pysweet.value

class pysweet.value.**Value_**(*val: _A*)

Bases: Generic[_A]

Pipeable value.

Parameters
val – Wrapped value.

pipe(*f: Callable[[_A], _B]*) → *Value_*[_B]

Transform self with f immutably.

```
>>> Value_(2).pipe(lambda x: x + 1).val
3
```

Parameters
f – Function.

Returns
Transformed Value_.

property val: _A

Get underlying value.

```
>>> Value_(2).val
2
```

Returns
Wrapped value.

PYTHON MODULE INDEX

p

INDEX