pro$lambda$

Release 0.3.6

Nov 10, 2020

Contents:

pro_lambda make it possible to modify your functions with standart mathematical and logical operators:

```
from pro_lambda import pro_lambda

some = pro_lambda(lambda : 1)
other = some + 1
# then we call result as if it was (lambda: 1)() + 1
assert other() == 2

some = pro_lambda(lambda x, y: x+y)
other = some + 1
# here we pass some arguments
assert other(1, 2) == 4

# we can also use another function on the right side
other = some + (lambda z, y: z - y)
assert other(1, y = 2, z = 3) == 4
```

It also supports async functions:

```
import asyncio
from pro_lambda import pro_lambda


async def main():

    async def _some(x):
        await asyncio.sleep(0.3)
        return x

    _save = _some
    some = pro_lambda(_some)
    other = some + (lambda: 1)
    assert some.is_async
    assert await other(1) == 2

    some = pro_lambda(lambda : 1)
    other = some + _some

    assert other.is_async
    assert await other(x=1) == 2

    some = pro_lambda(_some)
    other = some + _some
    assert other.is_async
    assert await other(x=1) == 2

    other = some == 1

    assert other.is_logical
    assert await other(1)
    assert not await other(2)

asyncio.run(main())
```

CHAPTER 1

API

1.1 pro_lambda

class pro_lambda.**pro_lambda**(*foo: Callable*)

Function modifier. Modified functions can work with mathematical operators, mix with other functions. Async supported

```
>>> some = pro_lambda(lambda : 1)
```

Now *some* can be used with math operators:

```
>>> other = some + 1
>>> other() # 1 + 1
2
```

And with other functions: >>> other = other - (lambda: 10) >>> other() # 1 + 1 - 10 8

Or with other LambdaMaths (LambdaMath is callable):

```
>>> other = other + pro_lambda(lambda : 8)
>>> other()
0
```

Parametrized functions are also supported: >>> some = pro_lambda(lambda x, y: x + y) >>> other = some - 5 >>> other(1, 1) # 1 + 1 - 5 3

Right-side function can also be parametrised, it's arguments will become keyword-only arguments:

```
>>> some = pro_lambda(lambda x, y: x + y)
>>> other = some + (lambda z, y: z - y)
>>> other(1, 2, z=3) # (1 + 2) - (3 - 2)
2
```

If any of two functions is async or awaitable, result is also async:

```
>>> async def foo():
...     await asyncio.sleep(1)
...     return 1
>>> some = pro_lambda(foo)
>>> other = some + 1
>>> await other()
2
```

is_async
If self.foo is async foo or it returns awaitable returns True

is_logical
Returns True if self.foo is a product of logical operator (==, >, >=, <, <=, &, |)

1.2 tools

class pro_lambda.tools.**ClsInitMeta**
Special metaclass for making class initialisers with @cls_init decorator

pro_lambda.tools.**cls_init**(*foo*)
Decorator, mark foo as class initialisier

pro_lambda.tools.**deco_log_exception**(*msg*, *logger: logging.Logger = None*, *except_=None*, *raise_=True*)
Decorator, decorated foo runs with exception logger

Parameters

- **logger** – logger to use, by default root is used
- **except** – exceptions that should be ignored (not logged)
- **raise** – if False, will not raise exception, only logs it

pro_lambda.tools.**log_exception**(*msg: str = 'error in context'*, *logger: logging.Logger = None*, *except_=None*, *raise_=True*)
Contextmanager, while in this context, any exception will be logged with logger

Parameters

- **msg** – msg for exception logger
- **logger** – logger to use with exceptions
- **except** – iterable of exceptions to ignore in logging
- **raise** – if False, will not raise exception, but will log it

Returns

pro_lambda.tools.**skip_not_needed_kwargs**(*foo*)
Decorator, decorated foo will silently skip not needed kwargs

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index