



Python的语法扩展系统

Moshmosh

Thautwarm&李欣宜

The awareness of low-level implementation details brings the appreciation of an abstraction and the intuitive explanation for it.

— Oleg Kiselyov

目录

CONTENTS

- >> 提供语法和语义的语言不仅仅是工具， 还是思维方式
- >> 表达能力的极限， 由内破除， 还是从外破除？
- >> Moshmosh： 我的Python不可能这么甜美清新
- >> 下班时在干什么？ 有没有空？ 可以来contribute吗？



1 提供语法和语义的语言 不仅仅是工具，还是思维方式



“语言只是工具”是现代社会最为荒谬的说法之一。即便存在海量的常见任务，他们在部分语言里很容易做到，而在其他语言里则如履薄冰；但这并不是本质问题。

来自语言设计的盲点，设计时藏匿的偏见，未证明无误的硬性约束，将会固化语言使用者的思想，阻碍他们见到背后的风景。



细说不是胡说，语言不是工具

Again, “语言只是工具”是现代社会最为荒谬的说法之一。
不仅仅是对自然语言，程序语言也是一样的哟。

百度一下“语言和思维”，我们能找到马克思爷爷的至理名言。



细说不是胡说，语言不是工具

马克思认为，语言是思维本身的要素，思想的生命表现的要素；
语言是思想的直接现实。

Marx > 语言是人们在社会劳动过程中，适应交流意识、传递信息的需要而产生的。
语言一经产生，又成为思维存在和发展的必要因素。

Marx> 思维和语言是相互依存、相互促进的。语言是现实的思维，是思维的物质外壳；语言的外壳又总是包含着思维的内容。思维的发展推动语言的发展，语言的发展又促进思维的发展。一般来说，语言的发展水平标志着思维的发展水平。但是，思维和语言又不是等同的，它们有各自的相对独立性和特殊规律。

语言思维是人类特有的意识形式，但它并不排斥人类直观思维、动作思维和其他特殊类型思维。然而，思维决不能以赤裸裸的形式存在，它从一开始就受着物质的纠缠，任何类型的思维都有其物质外壳。

细说不是胡说，语言不是工具

*思维的发展推动语言的发展，
语言的发展又促进思维的发展。
一般来说，
语言的发展水平标志着思维的发展水平。*

— Marx

语言决定思维模型

说到质数，
人们想到什么？



GNU-APL

∇PRIMS X

[1] (2=+/0=(1X)◦.|1X)/1X

[2] ∇



C++

```
dyn_arr_ty<int> prims(int n)
{
    dyn_arr_ty<int> prims = {};
    for (int i = 2; i < n; i++)
    {
        int bound = sqrt(i);
        for (int j = 2; j <= bound; j++)
        {
            if (i % j == 0)
                break;
            else if (j + 1 > bound) {
                prims.push_back(i);
            }
        }
    }
    return prims;
}
```



Haskell

```
prims = filterPrims [2..]
  where filterPrims (x:xs) =
        x : filterPrims ([e | e <- xs, e `mod` x /= 0])
```

语言决定思维模型

在实际业务中处理数据。



Haskell

case vehicle of

```
Car {passengers, brand=Level i}    ->  
    (0.9 + fromIntegral i / 2.0) * passengers  
Car {passengers}                    -> 0.9 * passengers  
Taxi {fares, brand = Level i}      -> (1.0 + i) * fares  
Bus {passengers} | passengers > 20 -> 2.0  
Bus {passengers}                    -> 1.0
```

虽然模式匹配似乎还不甚流行，但它仅是编程语言走向未来必然经过的一个极其不起眼的、实现简单的基础设施。

语言决定思维模型

Python

```
if isinstance(vehicle, Car):
    passengers = vehicle.passengers
    tmp = vehicle.brand
    if isinstance(tmp, Level):
        ret = (0.9 + tmp.i/2) * passengers
    else:
        ret = 0.9 * passengers
elif isinstance(vehicle, Taxi) and isinstance(vehicle.brand, Level):
    fares = vehicle.fares
    tmp = vehicle.brand
    ret = (1 + i) * fares

elif isinstance(vehicle, Bus):
    passengers = vehicle.passengers
    if passengers > 20:
        ret = 2
    else:
        ret = 1

else:
    raise SomeException
```

在实际业务中处理数据。

语言决定思维模型



Python?

```
with match(vehicle):
    if Car(~passengers, brand=Level(i)) or Car(~passengers) and do(i=0):
        ret = (0.9 + i/2) * passengers
    if Taxi(~fares, brand=Level(i)):
        ret = (1 + i) * fares
    if Bus(~passengers) and when(passengers > 20):
        ret = 2
    if Bus(~passengers):
        ret = 1
```

语言决定思维模型

一篇生动的博客

Think about it: Is someone whose first programming language was APL going to think about programming ever after in the same way as someone whose first programming language was COBOL or Assembly? “I find that my early exposure to APL warped my brain by forcing me to ask ‘what’s this mean?’ and ‘is this a reusable operation?’ and ‘what’s a pithy summary for all this algorithmic fluff?’” notes one present-day Python programmer.

— [How Your First Programming Language Warps Your Brain](#)

一位老大爷的名言

A language that doesn’t affect the way you think about programming, is not worth knowing.

— Alan Perlis([ALGOL 60](#))

语言决定思维模型

细说，不是胡说；语言，不止工具。

你所常用的语言决定了你思考的流向，和解决问题的方式，不管是对细节的实现还是对整体框架的设计。

上述提到的一些的简单案例，意在表达，不同语言的使用者，使用不同的心智模型去解决问题。

而对于这个现象的发生，我归因于语言本身。编程语言会对你的思维方式进行诱导，试图将你同化。

回到现实，即便是上述提到的这些简单案例，在包括Python的很多语言内，居然都并没有很好的解决方案。

人们不得不成规模地重复工作，或是任由冗余在codebase里猖獗；抛弃更深远的抽象和语义，最终代码的编写成为了让人烦恼的苦力。

Python是有极限的！我不写Python了！

我们不继续谈语言和思维的问题了，也不谈一些高级的特性是多么make sense却没有支持。

就说Python。Python是有极限的，只从语义语法上讲。和性能、GIL相关的问题我们放在一边。

1. 没有多行lambda
2. 作用域管理规则/name shadow(let-binding)
3. 表达式和语句区分，表达式内部不能包含语句
4. 没有语法宏，代码操作不够自动
5. 没有variant类型(只能靠一大堆抽象类和继承去workaround)
6. 对数据类型的方法不能扩展，或者进行扩展是初级的，没有基于类型的多态

那么告辞？

摆脱编程语言的
给你的限制？

搭嘎！ 阔托瓦鲁！

Python有一堆好东西：

1. Python有良好的启动速度(看向Julia)
2. Python的package系统高度可自定义(importlib, import mechanism)
3. PyPI: 分布广泛的极速镜像，自由方便的注册系统，用法千奇百怪
4. 规范的解释器和虚拟机实现
5. 字节码层面支持运行时报错定位
6. 可用package领域覆盖面大
7. 标准库功能强大，有大量封装程度很高的API
8. 简单、一致、直观的语法设计

作为一门拥有上述特性的脚本语言，即便存在着诸多不足，但当我想做点什么有趣的东西，写作乐软件，写文字冒险游戏，搞AI刷手游，甚至连接硬件和外部世界交互，也理所当然地使用Python。Python作为我第一门深入钻研的通用编程语言，放弃是不能放弃的。

所以要想办法从Python现有的语法限制中脱身。

摆脱编程语言的
给你的限制

解放我的Python世界!

```
● ● ●

# moshmosh?
# +quick-lambda

# +pipeline
xs = map(_ % 2, range(10)) | list
# -pipeline

assert list(map(lambda _: _ % 2, range(10))) == xs

# +pattern-matching
with match(11, 4):
    if a and (_, _) and when(a < 42):
        print("case `(_, _) and when(a < 42)` matched for {}".format(a))
    if _:
        print("case not matched")
```

我预期的语法(及语义)扩展系统:

1. 首行用`moshmosh?`标志模块
2. `+extension名 (extension参数)`开启扩展
3. `-extension名 (extension参数)`关闭扩展
4. 可以自定义扩展并注册
5. 在这套系统下, 有很多简单的自定义扩展可供练手

6 任何在不使用该系统时拥有的功能 (PYC二进制文件发布, C扩展等等), 在使用该系统后得以保持, 拥有工业级的可靠性和稳定性



2 表达能力的极限， 由内破除， 还是从外破除？



想要在Python的基础上拥有更多的语法、语义，
有很多办法。

我们要选择其中一个实现简单、易维护、
拥有鲁棒性的方式。

什么是“从外部破除”

利用external程序对Python进行扩展，主要是替换或包装Python的executable文件。

这并不是非常常见，一些例子如下

- [PyJulia](#)的python-ji可执行文件，使用了PyCall代理python main模块的execution
- 从修改过的CPython源代码里编译的Python，使用完全相同的Python字节码虚拟机，但在语法语义上可以存在差异。例如我的早期作品[thautwarm/flowpython](#)

此外，代码生成也是外部方法之一，例如[Cython](#)和[Nuitka](#)。

外部工具的问题在于，通常会引入复杂笨重的依赖，这对于实际的产品项目是不友好的。

什么是“由内部破除”

就纯Python而言，利用internal组件对Python进行扩展，是“由内部破除”。
例如，利用

1. `inspect.getsource`(e.g., [alexmojaki/sorcery](#))
2. 从字节码恢复信息(现有工具: [rocky/python-uncompyle6](#))
3. `codecs`标准库(e.g., [asottile/future-fstrings](#))
4. `importlib/import mechanism`(e.g., [lihaoyi/macropy](#))

邪恶的inspect.getsource

```
from inspect import getsource

@getsource
def f(x):
    for i in x:
        pass

expected = \
"""
@getsource
def f(x):
    for i in x:
        pass

""".strip()

assert f == expected
```

这种方法最为简单,可以直接地用在运行时函数上并拿到其源代码。但它也是**唯一**一个不可信的做法。

它依赖于源代码的存在(阻止二进制发布),并存在文件IO操作(潜在的breakage)。

残念的字节码信息恢复



```
from uncomyle6 import code_deparse
from io import StringIO

def f(x):
    if 0:
        print(x)
    for i in x:
        pass

s = StringIO()

code_deparse(f.__code__, out=s)
func_body_codestr = s.getvalue()
actual = \
"""
for i in x:
    pass
""".strip()
assert func_body_codestr == actual
```

if 0
没了

空行
没了

Python会调用一个简单的字节码优化器消除一些代码，

例如***if 0***的情形。于是无法从字节码恢复真实的源代码。

并且，由于uncomyle6的实现问题，源代码的行号也没有从字节码中恢复。

机智的codecs利用



```
# -*- coding: future_fstrings -*-  
thing = 'world'  
print(f'hello {thing}')
```

[asottile/future-fstrings](#) 通过codecs标准库注册一个新的encoding，从而能够接触到一次源代码重写的过程。这个重写本身是为了解决Python的encoding问题；future-fstrings用它重写了tokens，将3.6以后才有的f-string语法转为相应的str.format语法。

但实际使用这个机制，利用future-fstrings的已有代码，会有一次多余的源码重写。同时，该机制不能直接用在REPL或者main模块里。

Python import mechanism




```
# main.py
import macropy.activate
import actual_module

# actual_module.py
from macropy.quick_lambda import macros, f, _
from functools import reduce

print(list(map(f[_ + 1], [1, 2, 3]))) # [2, 3, 4]
print(reduce(f[_ * _], [1, 2, 3])) # 6
```

[lihaoyi/macropy](#)利用了Python的import hook机制，代理了从源代码转为字节码的过程。

和codecs的利用一样，该机制不能用在REPL或者main模块里。

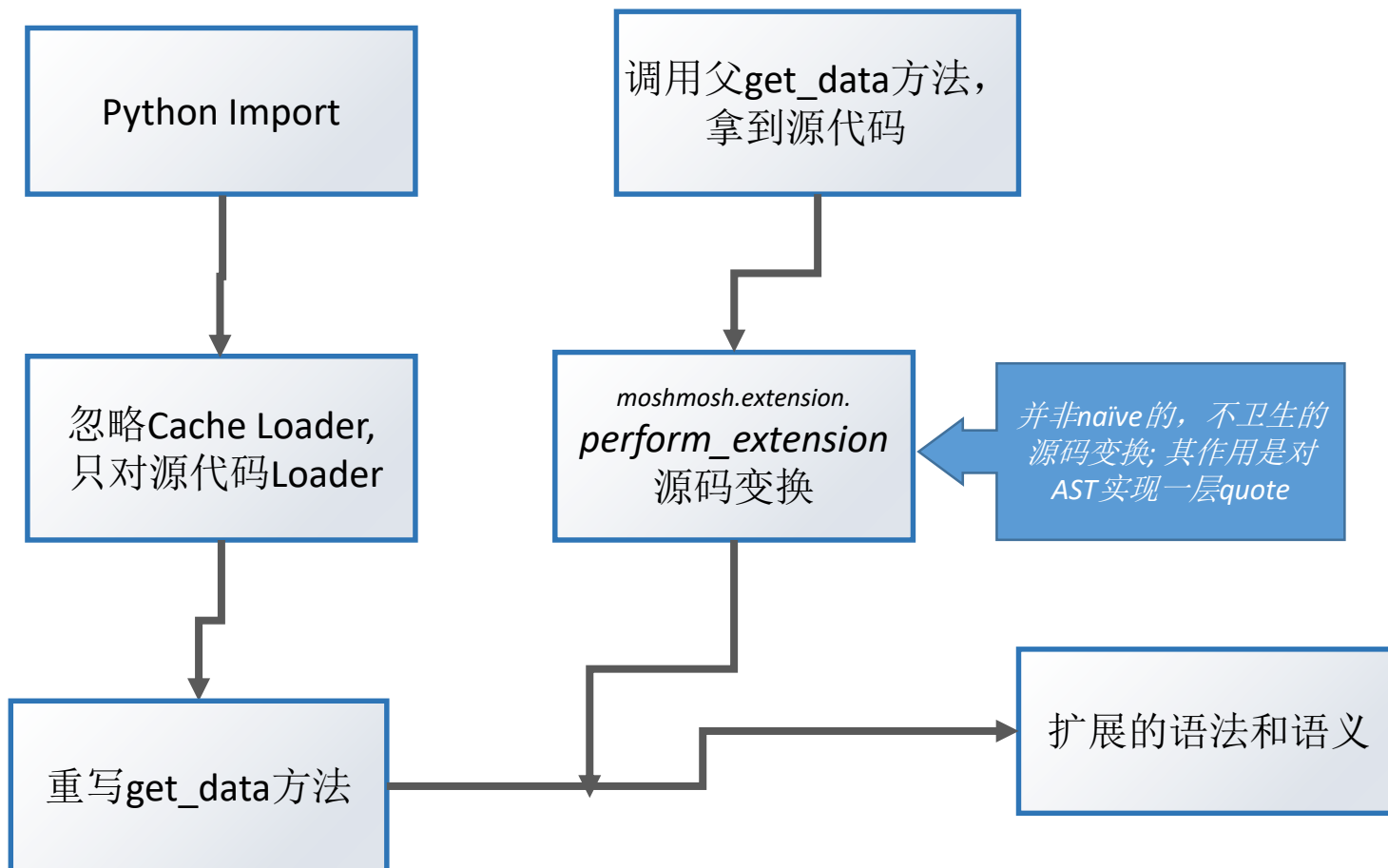


出于方便使用者的考量，我们目前希望我们的语法语义扩展系统在安装、使用和自定义扩展上快速、简单且无冗余，于是采用了基于“由内破除Python表达能力限制”的方法，具体来讲，就是利用Python import mechanism。

```
pip install -U moshmosh-base --no-compile
```

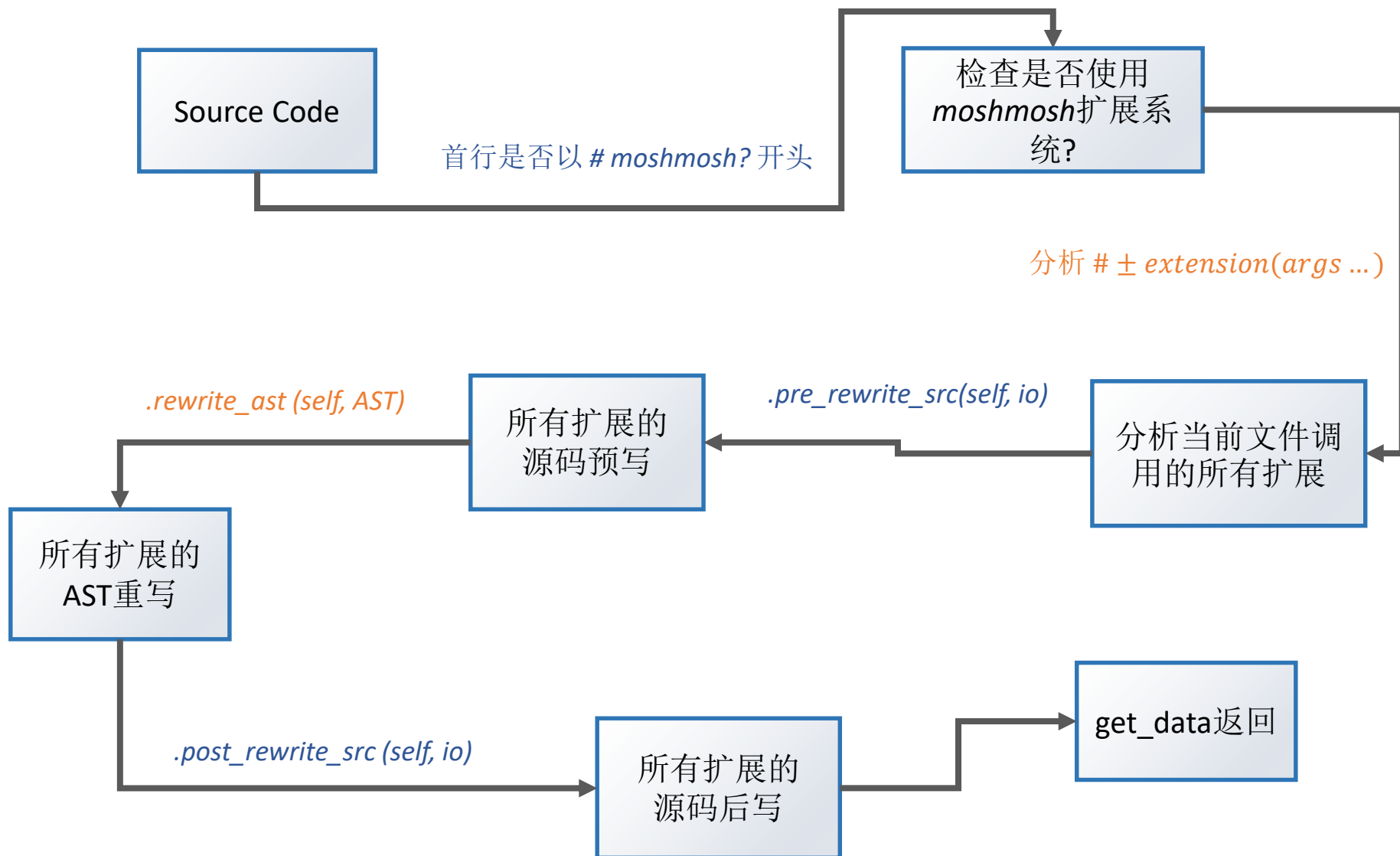
如何工作?

[moshmosh/extension_register.py](#) (只有45行)



perform extension

[moshmosh/extension.py](#)





3 Moshmosh: 我的Python 不可能这么甜美清新



世界上最快的Python Pattern Matching,
Elixir的Pin Operators,
来源于Scala的Quick Lambdas,
Quotation — — Template Python,
...

在IPython Console/Jupyter Notebook里重新
激发敲码的热情。

Template-Python

[moshmosh/extensions/template_python.py](https://github.com/moshmosh/extensions/template_python.py)

```

# moshmosh?
# +template-python

@quote
def f(x):
    x + 1
    x = y + 1

from moshmosh.ast_compat import ast
from astpretty import pprint

stmts = f(ast.Name("a"))
pprint(ast.fix_missing_locations(stmts[0]))
pprint(ast.fix_missing_locations(stmts[1]))
```

```

Expr(
  lineno=7,
  col_offset=4,
  value=BinOp(
    lineno=7,
    col_offset=4,
    left=Name(lineno=7, col_offset=4, id='a', ctx=Load()),
    op=Add(),
    right=Num(lineno=7, col_offset=8, n=1),
  ),
)

Assign(
  lineno=8,
  col_offset=4,
  targets=[Name(lineno=8, col_offset=4, id='a', ctx=Store())],
  value=BinOp(
    lineno=8,
    col_offset=8,
    left=Name(lineno=8, col_offset=8, id='y', ctx=Load()),
    op=Add(),
    right=Num(lineno=8, col_offset=12, n=1),
  ),
)
```

Pattern-Matching

[moshmosh/extensions/pattern_matching](https://moshmosh.com/extensions/pattern_matching)

```
# moshmosh?
# +pattern-matching

with match(1, 2):
    if (a, pin(3)):
        print(a)
    if (_, pin(2)) and (pin(1), _):
        print(10)
    if _:
        print(5)

with match([1, 2, 3, 4]):
    if [1, 2, a, b]:
        print(a + b)

class GreaterThan:
    def __init__(self, v):
        self.v = v

    def __match__(self, cnt: int, to_match):
        if isinstance(to_match, int) and cnt is 0 and to_match > self.v:
            return () # matched

with match(114, 514):
    if (GreaterThan(42)() and a, b):
        print(b, a)
```

pin(val): 用作用域内的值val进行比较的模式

and: 满足多个解构规则的组合模式

or: 满足其中一个解构规则的组合模式

A(a, ...): 调用A.__match__进行模式匹配

isinstance(type): 检查类型的pattern

(a, *b, c): 匹配tuple

[a, *b, c]: 匹配列表

Pattern-Matching

[benchmark.py](#)

Moshmosh实现

```
# In [1]: %timeit test_pampy(data)
# 56.6 µs ± 824 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

# In [2]: %timeit test_mm(data)
# 3.93 µs ± 86.5 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

基于template-python扩展实现。

性能比Pampy高数量级倍。

简单直接的自定义pattern，真实的tree pattern matching。

Match的每个分支是语句而不是表达力有限的表达式。

...

Quick-Lambda & Pipeline

[moshmosh/extensions/quick_lambdas.py](https://moshmosh.com/extensions/quick_lambdas.py)

[moshmosh/extensions/pipelines.py](https://moshmosh.com/extensions/pipelines.py)



```
f(_ + 1)          -> f(lambda _: _ + 1)
f(_, _ + 2)       -> f(lambda _: _, lambda _: _ + 2)
reduce(_0 + _1, seq) -> reduce(lambda _0, _1: _0 + _1, seq)

f(_0_)            -> lambda _0_: f(_0_)
f(_2_, _0_)       -> lambda _0_, _1_, _2_: f(_2_, _0_)
map(_[0] + _[1], _0_) -> lambda _0_: map(lambda _: _[0] + _[1], _0_)
```



```
data TaggedFixT f t = InT {tag :: t, outT :: f (TaggedFixT f t)}  
deriving instance Functor f => Functor (TaggedFixT f)  
deriving instance Foldable f => Foldable (TaggedFixT f)  
deriving instance Traversable f => Traversable (TaggedFixT f)
```

有必要提一下。仅仅是出现了map, reduce, filter, flatMap这些字眼，就理解为所谓的“函数式编程”，这一想法是完全错误的。

```
data ReimuBase a = RAssign MName a  
| RDecide a  
| RCall a [a]  
...  
deriving instance Foldable (ReimuBase a)  
deriving instance Ord a => Ord (ReimuBase a)  
deriving instance Generic (ReimuBase a)  
deriving instance Functor ReimuBase  
deriving instance Foldable ReimuBase  
deriving instance Traversable ReimuBase  
deriving instance Eq b => Eq (TaggedFixT ReimuBase b)  
deriving instance Ord b => Ord (TaggedFixT ReimuBase b)
```

随着闭包和高阶函数逐渐推广，证明了函数式编程的实用性。但一些基础的东西被众多新旧语言吸收，不是说他们就靠近了函数式编程。

引用透明，（基于函数类型的）多态，基于类型递归、递归类型、递归函数的问题求解模型，都是函数式编程极为重要的组成部分。

没有这些，不要开口就“函数式，函数式”，否则，不仅误导新人，还会沦为老手们的笑柄；而Python离idiomatic的函数式编程还有很长的路要走，并且也不是一定要走这条路。

而moshmosh在做的事，只是扩展Python，以迎合程序语言的发展趋势和日益无法回避的实际需求，而不是在写“函数式Python”！

如何实现一个扩展?



```
def times(a, b):  
    return (a, b)
```

```
# +scoped-operator(*, times)  
assert (1 * 5) == (1, 5)
```

```
# -scoped-operator(+, times)  
assert (1 * 5) == 5
```

```
def list_diff(a, b):  
    return [i for i in a if i not in b]
```

```
# +scoped-operator(+, list_diff)
```

```
assert [1, 2] - [1] == [2]  
assert [1, 2] - [3] == [1, 2]  
assert [1, 2] - [1, 2, 3] == []
```

我们以moshmosh-base中默认提供的最简单的扩展,

Scoped-Operator为例, 讲解如何利用moshmosh实现语义扩展。

该扩展加上大量空行和一些import, 共计60行。

Scoped-Operator使得用户可以将二元运算的语义局部地修改成一个特定的二元函数的调用, 这提升了Python的DSL能力。

如何实现一个扩展?

[moshmash/extensions/scoped_operators.py](https://github.com/moshmash/moshmash/blob/master/extensions/scoped_operators.py)

```
from moshmash.extension import Extension
from moshmash.ast_compat import ast

class ScopedOperator(Extension):
    identifier = "scoped-operator"

    def __init__(self, op_name: str, func_name: str):
        op_name = opname_map.get(op_name, op_name)
        func_name = func_name
        self.visitor = ScopedOperatorVisitor(
            self.activation,
            op_name,
            func_name
        )

    def rewrite_ast(self, node):
        return self.visitor.visit(node)
```

```
# +scoped-operator(+, myfunc)
```

identifier

op_name

func_name

如何实现一个扩展?

[moshmosh/extensions/scoped_operators.py](https://github.com/moshmosh/extensions/scoped_operators.py)



```
# https://github.com/python/cpython/blob/master/Parser/Python.asdl#L102
opname_map = {
    "+": 'Add',
    "-": 'Sub',
    "*": 'Mult',
    "/": 'Div',
    "%": 'Mod',
    "**": 'Pow',
    "<<": 'LShift',
    ">>": 'RShift',
    "|": 'BitOr',
    "^": 'BitXor',
    "&": 'BitAnd',
    '//': 'FloorDiv'
}
```

如何实现一个扩展?

[moshmash/extensions/scoped_operators.py](https://github.com/moshmash/moshmash/blob/master/extensions/scoped_operators.py)

左边的代码，找到指定的二元运算符，将对应的BinOp重写，得到了二元函数的调用。

判断扩展是否在该行激活

递归处理嵌套表达式

扩展在当前行未激活，但在表达式内部可能激活，所以递归处理

```
class ScopedOperatorVisitor(ast.NodeTransformer):
    """
    `a op b -> func(a, b)`, recursively.
    The `op => func` pair is given by users.
    """
    def __init__(self, activation, op_name: str, func_name: str):
        self.pair = (op_name, func_name)
        self.activation = activation

    def visit_BinOp(self, n: ast.BinOp):
        if n.lineno in self.activation:
            name = n.op.__class__.__name__
            pair = self.pair
            if name == pair[0]:
                fn = ast.Name(pair[1], ast.Load())
                return ast.Call(
                    fn,
                    [self.visit(n.left), self.visit(n.right)],
                    [],
                    lineno=n.lineno,
                    col_offset=n.col_offset
                )
        return self.generic_visit(n)
```

*IPython/Jupyter Notebook*支持

```

In [1]: # +pattern-matching
        # +quick-lambda
        # +pipeline

In [2]: def test_fn(data):
        with match(data):
            if (e, isinstance(int) and count):
                res = [e] * count
            if (_, *t1) or [-, *t1]:
                res = t1
            if "42":
                res = 42
        return res

In [3]: test_fn((1, 2, 3))
Out[3]: (2, 3)

In [4]: [(1, 3, 5), [1, 3, 5]] | map(test_fn(_), _0_) | list
Out[4]: [(3, 5), [3, 5]]

In [5]: test_fn("42")
Out[5]: 42

In [6]: test_fn(1)
-----
NotExhaustive:

In [7]: from functools import reduce
Out[7]: reduce(_0 + _1, [1, 2, 3]) | print([_0_] * 3)
[6, 6, 6]
```

IPython/Jupyter Notebook 支持



[moshmosh-ipython.sh](#)(only works for Linux users)

```
ipython profile create &&
```

```
pip install -U moshmosh-base -i https://pypi.org/simple --no-compile &&
```

```
wget https://raw.githubusercontent.com/thautwarm/moshmosh/master/moshmosh_ipy.py &&
```

```
mv moshmosh_ipy.py /home/$USER/.ipython/profile_default/startup/moshmosh_ipy.py
```



4 下班时在干什么? 有没有空? 可以来contribute吗?

添加一个新的扩展? 理由? 实际的use case?
为扩展添加静态检查支持(mypy plugins)?
文档?



THANK YOU



thautwarm:

- github.com/thautwarm
- twshere@outlook.com

李欣宜:

- github.com/li-xin-yi
- lixinyi@guandata.com