函数式编程

函数式编程（functional programming）
是一种编程范式（Programming paradigm）

编程范式中**范是**模范的意思，**范式**即模式、方法，编程范式就是如何编写程序的方法论

《Python函数式编程》
(Functional Programming in Python - David Mertz)

1. 函数是一等的（对象）。所有和数据相关的事情都可以通过函数自身来解决。什么意思呢？函数就像变量一样来使用,函数可以像变量一样被创建，修改，并当成变量一样传递，返回或是在函数中嵌套函数。

2. 递归作为主要控制结构。

3. 关注列表处理（列表处理 就是LISt Processing， 这是lisp名字的由来），列表经常被用于子列表的递归来替代循环。

4. 纯函数式编程语言会避免命令式语言中先赋值给一个变量，而另一个相同变量来跟踪程序状态这样的副作用。

5. 函数式编程不鼓励或完全不允许声明，替而代之的是表达式求值（换句话说，函数+参数）。通常情况下，一个程序就是一个表达式。
6. 函数式编程担忧的是计算什么，而不是如何计算。所以你的代码变成了在描述你要干什么，而不是怎么去干

7. 许多函数式编程利用高阶函数

非函数式编程👇

```
In : x = 1

In : def increment():
...:     global x
...:     x += 1
...:

In : increment()

In : x
Out: 2
```

函数式编程👇

```
In : def increment(x):
...:     return x + 1
...:

In : x = 1

In : increment(x)
Out: 2
```

```
In : (1 + 2) * 3 - 4
Out: 5
```

```
In : from operator import mul, add, sub

In : sub(mul(add(1, 2), 3), 4)
Out: 5
```

```
In : reduce(mul, [1, 2, 3, 4, 5])
Out: 120
```

PyFunctional 👇

```
In : from functional import seq

In :

In : (seq(1, 2, 3, 4)
...:     .map(lambda x: x * 2)
...:     .filter(lambda x: x > 4)
...:     .reduce(lambda x, y: x + y)
...: )
Out: 14
```

peewee 👇

```
tweets = (Tweet
          .select()
          .where(
              (Tweet.created_date >= date.today()) &
              (Tweet.is_published == True))
          .count())
```

函数式编程优点

1. 接近自然语言，易于理解

2. 方便debug和单元测试。由于函数式编程不依赖、也不会改变外界的状态，只要给定输入参数，返回的结果必定相同，可以把函数当做单元处理，debug和写单元测试比较容易

3. 易于「并发编程」。由于函数式编程不修改变量，也就没有资源竞争的问题，所以不需要加锁保护可变状态，也就没有死锁问题，所以可以很放心地把工作分摊到多个线程，进行并发编程

函数式编程缺点

1. 函数式编程相对于面向对象和过程式的编程性能要差，比如不断递归有堆栈的消耗，但是Python语言没有做尾递归优化。

2. 函数式编程不适合做IO操作，也不适合写GUI。Python内置很多数据类型，有时候就是需要对这些数据类型做更新，如果非要用函数式会增加时间消耗，产生额外的中间数据。当然也有一些场景非常适合，比如下面讲的4个语言特性部分。不要为了用函数式编程而用，而是在适合的地方使用适合的技术，不要无视现实环境。

函数式语言主要特性

1. 闭包(Closure)
2. 高阶函数(Higher-order function)
3. 柯里化(Currying)
4. 偏应用函数(Partially Applied Functions)

偏应用函数

```
In : def log(level, message):
...:     print(f'[{level}]: {message}')
...:

In : def warn_log(message):
...:     return log('WARN', message)
...:

In : warn_log('This is a warn')
[WARN]: This is a warn
```

```
In : from functools import partial

In : warn_log = partial(log, 'WARN')

In : warn_log('This is a warn')
[WARN]: This is a warn
```

partialmethod

```
from functools import partialmethod


class Cell(object):
    def __init__(self):
        self._alive = False
    @property
    def alive(self):
        return self._alive
    def set_state(self, state):
        self._alive = bool(state)
    set_alive = partialmethod(set_state, True)
    set_dead = partialmethod(set_state, False)

c = Cell()
c.alive          # False
c.set_alive()
c.alive          # True
```

柯里化

```
In : def add(x, y, z):
....:     return x + y + z
....:

In : def addX(z):
....:     def a(y):
....:         def _(x):
....:             return x + y + z
....:         return _
....:     return a
....:

In : add(1, 2, 3)
Out: 6

In : addX(1)(2)(3)
Out: 6
```

```
In : from toolz import curry

In  : @curry
....: def add(x, y, z):
....:     return x + y + z
....:

In : add(1)
Out: <function add at 0x10ab1e620>

In : add(1)(2)
Out: <function add at 0x10ab1e620>

In : add(1)(2)(3)
Out: 6

# curry做的事情相当于这个效果👇：多次partial
In: partial(partial(add, 1), 2)(3)
```

```
In : add(1, 2)
Out: <function add at 0x10ab1e620>

In : add(1, 2)(3)
Out: 6

In : add(1, 2, 3)
Out: 6

In : add(1)(2, 3)
Out: 6
```

延伸阅读

1. https://www.ibm.com/developerworks/linux/library/l-prog/index.html
2. https://docs.python.org/3/howto/functional.html
3. https://docs.python.org/3/library/functional.html
4. http://www.ruanyifeng.com/blog/2012/04/functional_programming.html
5. http://dongweiming.github.io/Expert-Python/#59
6. https://github.com/suor/funcy/
7. https://github.com/kachayev/fn.py
8. https://github.com/EntilZha/PyFunctional
9. https://github.com/coleifer/peewee