TFP

TFP is a python library that gives you functional programming.

Features

- **Lightweight**: < 10KB.
- **Pure**: Independent of other libraries (except for the package of the system).
- **Efficient**: Not at the expense of speed.
- **Wonderful**: A wonderful way for Functional Programming.

TFP is changed from kachayev / fn.py. We take the essence and discard the dross of it and furthermore extract and concentrate the essence.

At present, two main features are inherited: function composition and the anonymous function.

Function composition

> *In mathematics, function composition is the pointwise application of one function to the result of another to produce a third function. For instance, the functions* $f : X \to Y$ *and* $g : Y \to Z$ *can be composed to yield a function which maps* x *in* X *to* $g(f(x))$ *in* Z*. Intuitively, if* z *is a function of* y*, and y is a function of* x*, then* z *is a function of* x*. The resulting composite function is denoted* $g \circ f : X \to Z$*, defined by* $(g \circ f)(x) = g(f(x))$ *for all* x *in* X*.*
>
> ——*From wikipedia: Function composition*

For example——Find the norm-2 of a vector

> ***norm-2 of a vector****: a vector is a one-dimensional array (list), calculate the* ***square*** *of each element of it, then* ***add*** *them all together, finally calculate the* ***root*** *of it.*

You might do it like this:

```
1  from math import sqrt
2  vec = [3, 4]
3  norm_2 = sqrt(sum(map(lambda x: x*x, vec)))
```

This does not look too bad. However, this hell-style layering call looks less friendly when faced with tedious processes.

For another example

A string containing many English words separated by spaces. Now, you first **get a list** of words in this string, and then **filter out** words whose length is longer than 3, then turn all these words into **lowercase**, and then **filter out** the words that start with 'a' or 's', and **sort** them alphabetically, and finally **prints** a list of words on the screen.

You might do the task like this:

```
words = "Arya Sansa Brandon Snow Hodor Lady Ghost Cersei Imp Jaime Renly Joffery"
print(sorted(filter(lambda s: s.startswith(('a', 's')), map(str.lower, filter(lambda x: len(x) > 3, words.split(" "))))))
```

WTF! Yes, it works, and you can see the result `['arya', 'sansa', 'snow']` (Yes, I like these three roles) on your screen. You think you can make the process more elegant in this way (just adjust the format of the code) :

```
words = "Arya Sansa Brandon Snow Hodor Lady Ghost Cersei Imp Jaime Renly Joffery"
print(
    sorted(
        filter(
            lambda s: s.startswith(('a', 's')),
            map(
                str.lower,
                filter(
                    lambda x: len(x) > 3, words.split(" ")
                )
            )
        )
    )
)
```

You may have tried your best to achieve maximum legibility, but even so, threr are a lot of nested functions that it is a bit obscure.

We can do better with TFP:

Create a pipe to perform this task divided into six steps according to the description. step 1~6:

1. get a list of words in a string
2. filter out words whose length is longer than 3
3. turn all these words into lowercase
4. filter out the words that start with 'a' or 's'
5. sort them alphabetically
6. prints them on the screen

```
# Create a pipe to perform this task
pipeline = FP >> (lambda li: li.split(" ")) \
     >> partial(filter, FP >> len >> (_ > 3))\
     >> partial(map, str.lower)\
     >> partial(filter, lambda s: s.startswith(('a', 's')))\
     >> sorted\
     >> print

# Then throw the list into the created pipe
pipeline <= words

# You can use the same pipe to handle another string
another_words = "Balon Samwell Theon Yara Arynn Jon Lysa Robin Mord Frey Walder Pyp "
another_words | pipeline
```

Output:

```
['arya', 'sansa', 'snow']
['arynn', 'samwell']
```

NOTE: I used an underline-expressionin `_ > 3` (I tentatively name it as an underscore-expression) the 3rd lines of the above code. This is a special anonymous function to be discussed below. In fact, `_ > 3` roughly equivalent to:

```
lambda x: x > 3
```

Let's dissect the above code

Function pipelines

The function pipeline operator `>>` is used to compose functions to form a pipeline where each function passes its results to be consumed by the next one. And `FP` can create a function pipeline, then you can throw functions into the created pipeline one by one using operator `>>` . Hence,

```
>>> pipeline = FP >> func1 >> func2 >> func3
```

is a function pipeline that calls `func1()`, than pass the result to `func2()`, which goes to `func3()` . The code above is equivalent to the nested function definition:

```
>>> pipeline = lambda x: f3(f2(f1(x)))
```

In fact, you can pass the first function `func1` to `FP` as an argument:

```
>>> pipeline = FP(func1) >> func2 >> func3
```

For exampl, find the root of a number (may not necessarily be positive, when it is negative, find the root of the opposite number of the number):

```
>>> sqrt = FP(math.sqrt)
>>> safe_sqrt = abs >> sqrt
>>> safe_sqrt(-4)
2.0
```

In the code above, the argument is passed first to the `abs()` function and then is redirected it to the `sqrt()`. The arguments flow in the same direction that the flow operators (`>>` and `<<`) points to, so you can also do it like this:

```
>>> sqrt = FP(math.sqrt)
>>> safe_sqrt = sqrt << abs
>>> safe_sqrt(-4)
2.0
```

Now, let's go back to the previous example——Find the norm-2 of a vector:

```
vec = [3, 4]
norm_2 = FP >> (lambda li: map(_**2, li))\
           >> sum\
           >> math.sqrt\
           >>print
norm_2 <= vec
```

The result `5.0` will appear on your screen.

Throw the arguments into the pipeline

Once a pipeline is created, you can feed arguments to it in three ways:

1. function call: `pipeline(data)`
2. pipe operator: `data | pipeline` or `pipeline | data`
3. arrow operator: `pipeline <= data`

After putting the data into the front pipeline, the new data generated by the pipeline can continue to be put into the rear pipeline. In other words, pipe operator can be chained with function pipelines.

```
-16 | FP(abs) | FP(math.sqrt) | FP(math.sqrt)
-16 | FP(abs) >> FP(math.sqrt) | FP(math.sqrt)
-16 | FP(abs) >> FP(math.sqrt) >> FP(math.sqrt)
-16 | abs >> FP(math.sqrt) >> FP(math.sqrt)
-16 | abs >> FP(math.sqrt) >> math.sqrt
```

The results of the above five lines of code are `2.0`.

Anonymous function

Usage example

Function $f(x) = x + 10$.

In the traditional way, you may write:

```
def f(x):
  return x + 10

# or
f = lambda x: x + 10
```

Now, you can do it in a more elegant way:

```
f = _ + 10
```

Function $f(x, y) = x^2 + y^2$ is `f = _**2 + _**2`.

Underscore expression

As you see above, underscore expression gives you a wonderful way to create anonymous functions. This is a great initiative. Thanks kachayev.

In the underscore expression (you can also call it placeholder expression), **the number of underscores is in accordance with the number of independent variables. The order of the arguments is in accordance with the order of the underscores.**

When you create an underscore expression, you can call it right now, for instance `(_**2 + _**2)(3, 4)` and you will get `5.0`. Obviously, **underscore expression is a callable object just like a function**. You can print it to see what it is:

```
>>> _**2 + _**2
<class 'anoyfunc.Underscore':a callable object.Roughly equivalent to
function: lambda x1, x2: x1 ** 2 + x2 ** 2>
```

Now, you may be concerned about the efficiency of the underline expression. Firstly let's compare the efficiency of kachayev's underline expression with the way the system provides, these two methods run the same function $f(x, y, z) = x - 1 + \frac{5y}{4z}$:

```
f = lambda x, y, z: x - 1 + y * 5 / (4 * z)
g = _ - 1 + _ * 5 / (4 * _)

running_times = 10_000
t0 = time()
for i in range(1, n):
    a = f(i, i, i)
t1 = time()
dt1 = t1 - t0

t2 = time()
for i in range(1, n):
    a = g(i, i, i)
t3 = time()
dt2 = t3 - t2
```

The results are shown in the following table:

Running times	**5_000**	**50_000**	**50_000**	**500_000**	**5_000_000**
System's time-consuming	0.0045	0.0605	0.4181	2.8523	25.4367
kachayev's time-consuming	0.1665	1.2492	9.8923	80.9178	770.9814
time-consuming ratio	37.0015	20.6445	23.6627	28.3687	30.3098

NOTE: the time duration is measured in seconds.

Very scary!!! Their time-consuming ratio is nearly thirty times (the average ratio is 28). It is clear that kachayev's simplicity is at the expense of efficiency. I have to remold it, or I'll never use it.

If you analyze his source code, you will find that a number of recursion calls occur every time you call an underscore function, especially when that function is very complex. However, it also reflects the agility of the author's thinking.

So, to get closer to the efficiency of the system itself, I changed the internal implementation. When you create an underscore expression, I generate a corresponding function inside. Every time you call the underscore expression, you are actually executing the function inside. There is no recursion during the calling procedure. We still use the same example to test it:

The results are shown in the following table:

Running times	**5_000**	**50_000**	**50_000**	**500_000**	**5_000_000**
System's time-consuming	0.0045	0.0605	0.4181	2.8523	25.4367
TFP's time-consuming	0.0105	0.1110	0.6466	4.3295	44.0001
time-consuming ratio	2.3334	1.8347	1.5467	1.5179	1.7298

Take a look at the ratio between TFP and kachayev:

Running times	**5_000**	**50_000**	**50_000**	**500_000**	**5_000_000**
time-consuming ratio	15.8571	11.2523	15.2993	18.6897	17.5223

NOTE: the time duration is measured in seconds.

Obviously, the efficiency of TFP is quite close to the efficiency of the system. And kachayev's method is very time-consuming, which has to be the reason we carefully consider using it. The more complicated the function, the more noticeable the difference.

UUnderscore expression

> *Double underscore expression.*

There is a flaw in underscore expression, for instance, polynomial $f(x) = -x^2 + x - 1$ cannot be expressed in underscore expression. This is why we have the uunderscore expression (`__`).

[NOTE]: `-_**2 + _ -1` *is* $f(x, y, z) = -x^2 + x - 1$.

In the uunderscore expression, all the double underscore represent the same independent variable. So polynomial $f(x) = -x^2 + x - 1$ is equivalent to `-__**2 + __ - 1`.

Similarly, uunderscore expression is a callable object just like a function. You can print it to see what it is:

```
>>> -__**2 + __ - 1
<class 'anoyfunc.UUnderscore':a callable object.Roughly equivalent to
function: lambda x: (-x ** 2) + x - 1>
```

the uunderscore expression can contain the underscore expression in which case the whole expression will be an uunderscoreexpression. But it is very important that the first argument must be the __ and the order of the rest arguments is in accordance with the order of the _ when you pass the arguments. For instance:

```
>>> __ **2 + _ - 2*__ + _/3
<class 'anoyfunc.UUnderscore':a callable object.Roughly equivalent to
function: lambda x, x1, x2: x ** 2 + x1 - 2 * x + x2 / 3>
```

Review

The underscore expression is independent of function composition, you can use them alone or together. Before using them, import them first:

```
from compfunc import FP
from anoyfunc import underscore as _
from functools import partial, reduce

pipeline = FP >> partial(map, _*2) \
           >> partial(filter, _ > 10) \
           >> partial(reduce, _ + _)\
           >> print
range(0, 10) | pipeline  # 60
```

If you find any mistake in the article, please specify where the mistake is and leave your amendment opinions. Constructive comments or criticisms are welcome!

Enjoy TFP!