

# functools模块

## reduce

- `functools.reduce(function, iterable[, initial])`
- 就是减少的意思
- 初始值没有提供就在可迭代对象中取一个

```
1 from functools import reduce
2
3 s = sum(range(10))
4 print(s)
5
6 s = reduce(lambda x: x, range(10))
7 print(s) # TypeError: <lambda>() takes 1 positional argument but 2 were given
```

从上面的异常推断lambda应该2个参数

```
1 s = reduce(lambda x,y: print(x,y), range(10))
2 print(s)
3
4 返回结果如下
5 0 1
6 None 2
7 None 3
8 None 4
9 None 5
10 None 6
11 None 7
12 None 8
13 None 9
14 None
```

上一次lambda函数返回值会成为下一次的x

```
1 from functools import reduce
2
3 s = sum(range(10))
4 print(s)
5
6 s = reduce(lambda x,y: x + y, range(10), 100)
7 print(s)
```

sum只能求和，reduce能做更加复杂的迭代计算。

思考：5的阶乘怎么做？

# partial

## 偏函数

- 把函数部分参数固定下来，相当于为部分的参数添加了固定的默认值，形成一个新的函数，并返回这个新函数
- 这个新函数是对原函数的封装

```
1 from functools import partial
2
3 def add(x, y):
4     return x + y
5
6 newadd = partial(add, y=5)
7
8 print(newadd(4))
9 print(newadd(4, y=15))
10 print(newadd(x=4))
11 print(newadd(4, 6)) # 可以吗
12 print(newadd(y=6, x=4))
13
14 import inspect
15 print(inspect.signature(newadd))
```

```
1 from functools import partial
2
3 def add(x, y, *args):
4     return x + y + sum(args)
5
6 newadd = partial(add, 1, 2, 3, 4, 5)
7
8 print(newadd())
9 print(newadd(1))
10 print(newadd(1, 2))
11 print(newadd(x=1)) #
12 print(newadd(x=1, y=2)) #
13
14 import inspect
15 print(inspect.signature(newadd))
```

## partial本质

```
1 def partial(func, *args, **keywords):
2     def newfunc(*fargs, **fkeywords): # 包装函数
3         newkeywords = keywords.copy()
4         newkeywords.update(fkeywords)
5         return func(*(args + fargs), **newkeywords)
6     newfunc.func = func # 保留原函数
7     newfunc.args = args # 保留原函数的位置参数
8     newfunc.keywords = keywords # 保留原函数的关键字参数参数
9     return newfunc
10
11 def add(x, y):
12     return x + y
13
14 foo = partial(add, 4)
```

尝试分析functools.wraps的实现。类别下面的实现

```
1 from functools import partial, wraps
2 import inspect
3
4 def add(a, b, c, d):
5     return a + b + c + d
6
7 newadd = partial(add, b=2, c=3, d=4)
8 print(inspect.signature(newadd))
```

## lru\_cache

@functools.lru\_cache(maxsize=128, typed=False)

- lru即Least-recently-used，最近最少使用。cache缓存
- 如果maxsize设置为None，则禁用LRU功能，并且缓存可以无限制增长。当maxsize是二的幂时，LRU功能执行得最好
- 如果typed设置为True，则不同类型的函数参数将单独缓存。例如，f(3)和f(3.0)将被视为具有不同结果的不同调用
- Python 3.8 简化了调用，可以使用

```
1 @functools.lru_cache
2 def add():
3     pass
4
5 # 等价于
6 @functools.lru_cache(128)
7 def add():
8     pass
```

```
1 from functools import lru_cache
2 import time
3
4 @lru_cache()
5 def add(x, y=5):
6     print('-' * 30)
7     time.sleep(3)
8     return x + y
9
10 print(1, add(4, 5))
11 print(2, add(4, 5))
12 print(3, add(x=4, y=5))
13 print(4, add(y=5, x=4))
14 print(5, add(4.0, 5))
15 print(6, add(4))
```

到底什么调用才能用缓存呢？

## lru\_cache本质

- 内部使用了一个字典
- key是由\_make\_key函数构造出来

```
1 from functools import _make_key
2
3 print(_make_key((4, 5), {}, False))
4 print(_make_key((4, 5), {}, True))
5 print(_make_key((4,), {'y':5}, False))
6 print(_make_key((), {'x':4, 'y':5}, False))
7 print(_make_key((), {'y':5, 'x':4}, False))
```

## 应用

```
1 # 斐波那契数列lru_cache版
2 from functools import lru_cache
3
4 @lru_cache()
5 def fib(n):
6     return 1 if n < 3 else fib(n-1) + fib(n-2)
7
8 print(fib(101))
```

## 总结

### lru\_cache装饰器应用

- 使用前提
  1. 同样的函数参数一定得到同样的结果，至少是一段时间内，同样输入得到同样结果
  2. 计算代价高，函数执行时间很长
  3. 需要多次执行，每一次计算代价高
- 本质是建立函数调用的参数到返回值的映射
- 缺点
  - 不支持缓存过期，key无法过期、失效
  - 不支持清除操作
  - 不支持分布式，是一个单机的缓存
- lru\_cache适用场景，单机上需要空间换时间的地方，可以用缓存来将计算变成快速的查询

学习lru\_cache可以让我们了解缓存背后的原理。