目 录

1 函数式编程

函数式编程是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量。对于函数式编程的函数，只要输入是确定的，输出就是确定的。

函数式编程的另一个特点是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数。

1.1 高阶函数

对于 python 而言，变量可以指向函数，而函数名也可以作为一个变量。如果一个函数可以接收另一个函数作为参数，那么这种函数就称之为高阶函数。

1.1.1 变量可以指向函数

在 python 中，函数本身可以赋值给变量，也就是说，变量可以指向函数。如下例所示：

```
# abs()是python内置的求绝对值的函数
f = abs
f(-10)
# 输出结果为10
# 变量f指向abs函数本身
```

1.1.2 函数名也是变量

在 python 中，函数名就是指向函数的变量，所以也可以把函数名当作变量。如下例所示：

```
# abs()是python内置的求绝对值的函数
abs = 10
# abs现在指向10，无法再通过abs(-10)调用求绝对值函数
```

1.1.3 传入函数

对于高阶函数，它可以接收另一个函数作为参数。如下例所示：

```
def add(x, y, f):
    return f(x) + f(y)

add(-5, 6, abs)
# 输出结果为11
```

1.1.4 map 函数

map() 函数接收两个参数，一个是函数，另一个是序列，map 将传入的函数依次作用到序列的每个元素，并把结果作为新的 list 返回。如下例所示：

```
def f(x):
    return x * x

map(f, [1, 2, 3])
# 返回[1, 4, 9]
```

1.1.5 reduce 函数

reduce() 函数接收两个参数，一个是函数，另一个是序列，reduce 将一个函数作用在一个序列上。这个函数必须接受两个参数，然后 reduce 将结果和序列的下一个元素传入函数，依次进行计算。reduce 函数的效果如下所示：

```
reduce(f, [x1, x2, x3]) = f(f(x1, x2), x3)
```

reduce 的使用如下例所示：

```
def add(x, y):
    return x + y

reduce(add, [1, 3, 5, 7, 9])
# 输出结果为25
```

1.1.6 filter 函数

filter() 函数接收两个参数，一个是函数，另一个是序列。filter() 把传入的函数依次作用到序列的每个元素，然后根据返回值是 True 还是 False 决定保留还是丢弃该元素。如下例所示：

```
def is_odd(n):
    return n % 2 == 1

filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15])
# 返回[1, 5, 9, 15]
```

1.1.7 sorted 函数

sorted() 函数接收两个参数，一个是序列，另一个是函数，它通过这个函数来实现自定义的排序。如下例所示：

```
def reversed_cmp(x, y):
```

```
        if x > y:
            return -1
        else:
            return 1
        return 0

    sorted([36, 5, 12, 9, 21], reversed_cmp)
    # 返回[36, 21, 12, 9, 5]
```

1.1.8 将函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。如下例所示：

```
    def lazy_sum(*args):
        def sum():
            ax = 0
            for n in args:
                ax = ax + n
            return ax
        return sum

    f = lazy_sum(1, 3, 5, 7, 9)
    # f现在指向sum()函数
    f()
    # 输出结果为25
```

需要注意的是，每次调用都会返回一个新的函数，这些函数的地址值不同，如下所示：

```
    f1 = lazy_sum(1, 3, 5, 7, 9)
    f2 = lazy_sum(1, 3, 5, 7, 9)
    f1 == f2
    # 输出结果为False
```

还有一点需要注意的是，返回函数中的变量如果后续发生了变化，返回函数的输出结果也会发生改变，如下例所示：

```
    def count():
        fs = []
        for i in range(1, 4):
            def f():
                return i * i
            fs.append(f)
        return fs

    f1, f2, f3 = count()
    # f1、f2和f3都引用了变量i，i最终值为3，所以f1()、f2()和f3()的输出结果都为9
```

如果想让该变量不发生变化，可以再嵌套一个函数，将这个变量作为函数的参数，如下所示：

```
def count():
    fs = []
    for i in range(1, 4):
        def f(j):
            def g():
                return j * j
            return g
        fs.append(f(i))
    return fs
```

1.2 关键字 lambda

关键字 lambda 可以用于创建匿名函数，例子如下所示：

```
lambda x: x * x
# 以上这个匿名函数等价于
def f(x):
    return x * x
```

匿名函数只能有一个表达式，返回值就是该表达式的结果。再举一个例子：

```
map(lambda x: x*x, [1, 2, 3, 4])
# 返回[1, 4, 9, 16]
```

需要知道的是，匿名函数也是一个函数对象，有自己的函数地址，所以可以让变量指向这个函数，如下所示：

```
f = lambda x: x*x
f(5)
# 输出结果为25
```

1.3 装饰器

装饰器可以在代码运行期间动态地增加功能，在本质上，装饰器就是一个返回函数的高阶函数。装饰器的使用如下所示：

```
def log(func):
    def wrapper(*args, **kw):
        print 'call %s():' % func.__name__
        return func(*args, **kw)
    return wrapper

# 以下log就是一个装饰器
@log
def now():
    print '2017-2-16'
```

把 @log 放到 now() 函数的定义处，相当于增加了语句：

```
now = log(now)
```

当调用 now 函数时，实际是执行以下语句：

```
now = log(now)
now()
```

如果装饰器本身需要传入参数，可以写一个返回装饰器的高阶函数，如下例所示：

```
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print '%s %s():' % (text, func.__name__)
            return func(*args, **kw)
        return wrapper
    return decorator

@log('execute')
def now():
    print '2017-2-16'
```

当调用 now 函数时，实际是执行以下语句：

```
now = log('execute')(now)
now()
```

需要注意的是，因为返回了 wrapper 函数，所以 now 现在指向的是 wrapper 的地址，now.__name__ 的值为“wrapper”。此时，之后如果有依赖函数签名的代码就会出错。

我们可以使用 python 内置的 functools.wraps 将 wrapper 函数的 __name__ 改为“func”，如下所示：

```
import functools

def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print 'call %s()', % func.__name__
        return func(*args, **kw)
    return wrapper
```

1.4 偏函数

偏函数是 functools 模块定义的一个函数 functool.partial，它将一个函数的某些参数给固定住，然后返回一个对应的新函数。

举个例子，int() 函数可以把字符串转换为整数。int() 函数还提供了额外的 base 参数，默认值为 10，如果传入 base=N 参数，就可以做 N 进制的转换。如下所示：

```
int('12345')
# 输出结果为12345
int('12345', base=8)
# 输出结果为5349
```

functools.partial 可以帮助我们建立一个偏函数，如下所示：

```
int2 = functools.partial(int, base=2)
int2('1000000')
# 输出结果为64
```

需要知道的是，int2 函数只是将 int 函数的 base 参数设定为默认值 2。在实际函数调用时还可以传入其他值，如下所示：

```
int2('1000000', base=10)
# 输出结果为1000000
```