27 高阶函数与函数式编程-慕课专栏

从前面章节，我们知道 *Python* 函数是以对象的形式实现的，属于 **一等对象** (*first-class object*)。根据编程语言理论，一等对象必须满足以下条件：

- 可在运行时创建；
- 能赋值给变量或者某种数据结构；
- 能作为参数传递给函数；
- 能作为函数执行结果返回；

Python 函数同时满足这几个条件，因而也被称为 **一等函数** 。 **高阶函数** 则是指那些以函数为参数，或者将函数作为结果返回的函数。对高阶函数稍加利用，便能玩出很多花样来。本节从一些典型的案例入手，讲解 *Python* 函数高级用法。合理应用函数式编程技巧，不仅能让代码更加简洁优雅，还能提高开发效率和程序质量。

函数式编程技巧最适合用在数据处理场景，接下来以成绩单计算为例，展开讲解。原始数据如下：

```
scores = [
   {
      'name': '小雪',
      'chinese': 90,
      'math': 75,
      'english': 85,
   },
   {
      'name': '小明',
      'chinese': 70,
      'math': 95,
      'english': 80,
   },
   {
      'name': '小丽',
      'chinese': 85,
      'math': 85,
      'english': 90,
   },
   {
      'name': '小宇',
      'chinese': 85,
      'math': 95,
      'english': 90,
   },
   {
      'name': '小刚',
      'chinese': 65,
      'math': 70,
      'english': 55,
   },
   {
      'name': '小新',
      'chinese': 85,
      'math': 85,
      'english': 80,
   },
]
```

sorted

排序是我们再熟悉不过的场景，如果待排序元素可以直接比较，调用 *sorted* 函数即可：

```
>>> numbers = [2, 8, 6, 9, 7, 0, 1, 7, 0, 3]
>>> sorted(numbers)
[0, 0, 1, 2, 3, 6, 7, 7, 8, 9]
```

对比较复杂的数据进行排序，则需要一些额外的工作。假如语文老师想对语文成绩进行排序，改如何进行呢？

sorted 支持指定一个自定义排序函数 *key* ，该函数以列表元素为参数，返回一个值决定该元素的次序。由于我们需要根据语文成绩对元素进行排序，因此需要实现一个函数将语文成绩提取出来作为比较基准：

```
def by_chinese(item):
    return item['chinese']
```

现在只需要将 by_chinese 函数作为 key 参数传给 sorted 即可实现语文成绩排序：

```
>>> for item in sorted(scores, key=by_chinese):
...     print(item['name'], item['chinese'])
...
小刚 65
小明 70
小丽 85
小宇 85
小新 85
小雪 90
```

自定义排序函数还可以控制升降序，如果需要按分数从高到底依次排序，可以返回成绩的负数作为排序基准：

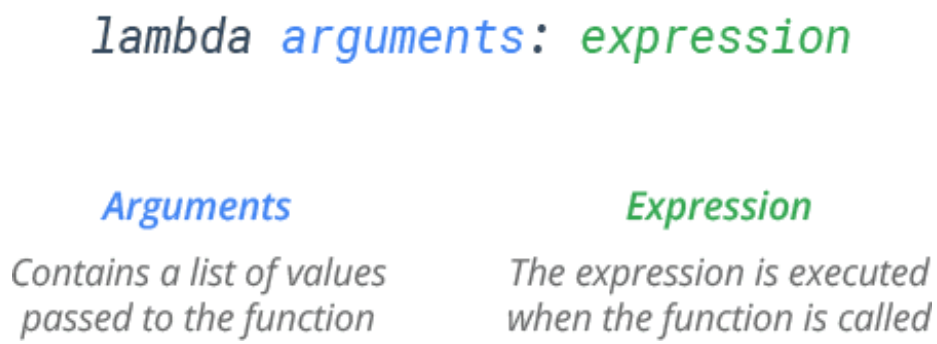
```
def by_chinese_desc(item):
    return -item['chinese']

>>> for item in sorted(scores, key=by_chinese_desc):
...     print(item['name'], item['chinese'])
...
小雪 90
小丽 85
小宇 85
小新 85
小明 70
小刚 65
```

当然了，通过 *sorted* 函数 *reverse* 参数控制升降序，是一个更好的编程习惯，逻辑更清晰：

```
>>> for item in sorted(scores, key=by_chinese, reverse=True):
...     print(item['name'], item['chinese'])
...
小雪 90
小丽 85
小宇 85
小新 85
小明 70
小刚 65
```

lambda

像 *by_chinese* 这样直接返回结果的极简函数，其实没有必要大动干戈，用 **匿名函数** 定义即可。*Python* 中的 *lambda* 关键字用于定义匿名函数，匿名函数只需给出参数列表以及一个表达式作为函数返回值：



这样一来，*by_chinese* 这个自定义排序函数，可以这样来定义：

```
by_chinese = lambda item: item['chinese']
```

相应地，我们实现语文成绩排序的代码编程这样子：

```
>>> for item in sorted(scores, key=lambda item: item['chinese']):
...     print(item['name'], item['chinese'])
...
小刚 65
小明 70
小丽 85
小宇 85
小新 85
小雪 90
```

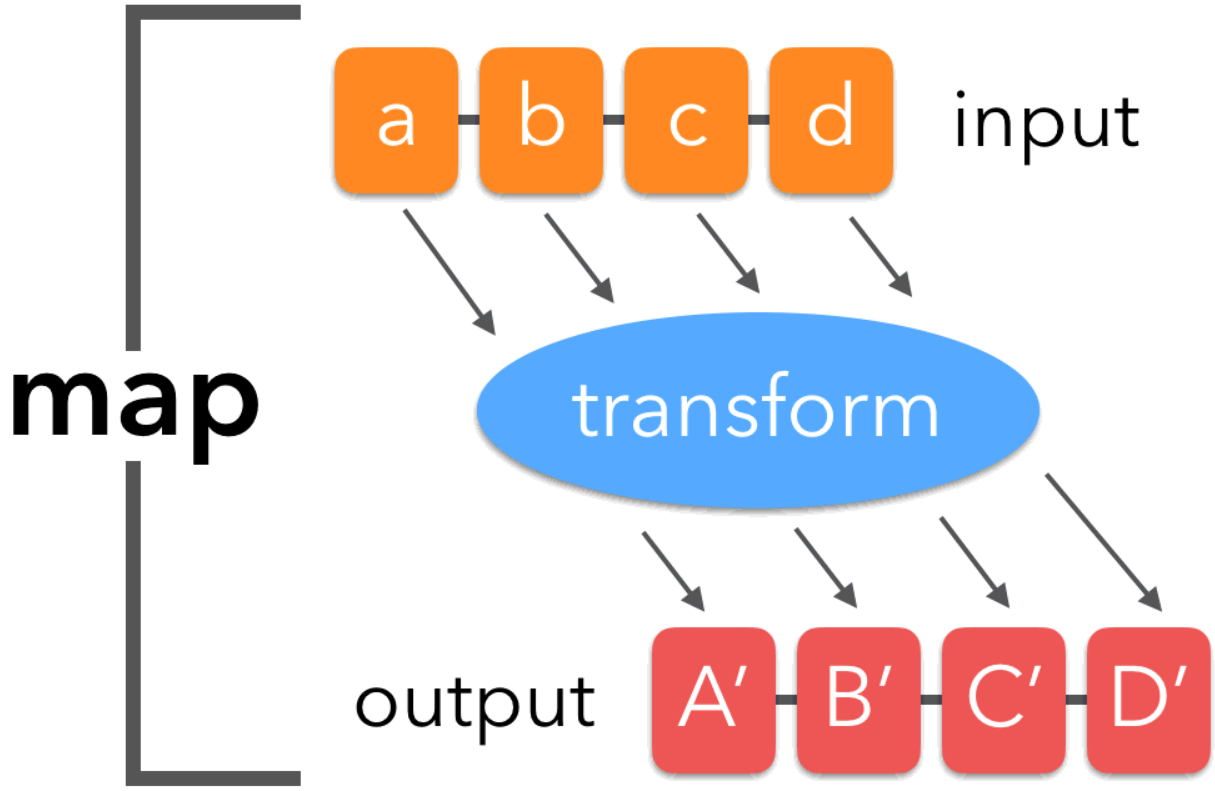
数学老师来了，也只需要改动一点点，就能实现数学成绩排序了：

```
>>> for item in sorted(scores, key=lambda item: item['math']):
...     print(item['name'], item['math'])
...
小刚 70
小雪 75
小丽 85
小新 85
小明 95
小宇 95
```

函数式编程语言一般都会提供 *map* 、 *filter* 以及 *reduce* 这 3 个高阶函数，再复杂的数据统计处理任务都可以转换成这些算子的组合。因此，不少大数据平台，例如 *Hadoop* 等，都以 map 、 reduce 为基础算子。*Python* 内部也自带了这几个高阶函数，我们分别来看：

map

map 函数接受 **转换函数** 以及一个 **可迭代对象** 作为参数，返回另一个生成器，其元素是输入元素的转换结果：



班主任需要知道每位童鞋的平均分，*map* 函数能否帮到他？我们可以先写一个简单的函数来计算平均成绩：

```
def calculate_total(item):
    return {
        'name': item['name'],
        'total': item['chinese'] + item['math'] + item['english'],
    }
```

这样一来，我们用 *map* 函数将成绩列表中的每个元素转换成平均成绩不就可以了吗？

```
>>> for item in map(calculate_total, scores):
...     print(item['name'], item['total'])
...
小雪 250
小明 245
小丽 260
小宇 270
小刚 190
小新 250
```

不仅如此，我们还可以进一步组合，对平均成绩进行排序：

```
>>> for item in sorted(map(calculate_total, scores), key=lambda item: item['total'], reverse=True):
...     print(item['name'], item['total'])
...
小宇 270
小丽 260
小雪 250
小新 250
小明 245
小刚 190
```

这就是高阶函数与算子组合的巨大威力，寥寥几行代码便完成了复杂的数据处理。尽管如此，这种风格也不能过分滥用。刻意应用高阶函数和函数式编程，代码可能更不可读，适得其反。

此外，*Python* 程序一般不直接使用 *map* 函数，而是通过更有 *Python* 格调的推导式：

```
totals = [calculate_total(item) for item in scores]
```

filter

filter 函数根据指定 **判定函数** ，对可迭代对象中的元素进行过滤，过滤结果同样以可迭代对象的形式返回。判定函数以某个元素为参数，返回 *true* 或 *false* 标识该元素是否应该出现在结果中。

学校举办英语竞赛，英语老师想挑一些成绩比较好的童鞋前去参加。她设定了一个标准，上次考试成绩在 *90* 分或者以上。为了从成绩单中筛选出符合条件的童鞋，我们需要先定义一个判定函数：

```
lambda item: item['english'] >= 90
```

这个 *lambda* 函数某位同学成绩为输出，输出英语成绩是否大于等于 *90* 分。有了判定函数，调用 filter 即可将符合条件的同学给过滤出来：

```
>>> for item in filter(lambda item: item['english'] >= 90, scores):
...     print(item['name'], item['english'])
...
小丽 90
小宇 90
```

同样，*Python* 程序一般不直接使用 *filter* 函数，而是通过更有 *Python* 格调的推导式：

```
candidates = [item for item in scores if item['english'] >= 90]
```

reduce

reduce 函数对可迭代对象的全部元素进行归并，归并方法由归并函数确定。归并函数以两个元素为输入，将两个元素进行合并，最后将结果作为输出返回。求和是一种典型的合并算子，归并函数如下：

```
add = lambda a, b: a + b
```

```
>>> add(1, 2)
3
>>> add(3, 4)
7
```

将数学运算做成算子后，可以任何组合做一些有趣的事情。例如，与 *reduce* 函数组合，对整数序列求和：

```
>>> numbers = [2, 8, 6, 9, 7, 0, 1, 7, 0, 3]
>>> import functools
>>> functools.reduce(add, numbers)
43
```

你也许会说，对序列求和，*sum* 函数就可以胜任了，何必搞得这么复杂？

```
>>> sum(numbers)
43
```

是的，由于求和是一个很常见的操作，*sum* 函数被单独拿出来实现。*reduce* 作为通用的归并操作，则更灵活，威力也更强大。只需编写归并函数，便可实现任何你想要的归并操作。例如，对数字序列求乘积：

```
>>> numbers = [6, 1, 2, 4, 3, 1, 3, 7, 5, 2]
>>> functools.reduce(lambda a, b: a*b, numbers)
30240
```

operator

Python 内置了常用数学运算算子，诸如加、减、乘、除等等，无须重复编写 lambda 函数来实现，以加法为例：

```
>>> import operator
>>> operator.add(1, 2)
3
```

这样一来，对整数序列求乘积可以进一步简化成这样：

```
>>> numbers = [6, 1, 2, 4, 3, 1, 3, 7, 5, 2]
>>> functools.reduce(operator.mul, numbers)
30240
```

operator 模块中的算子就不一一赘述了，请自行前去探索一番。

functools

functools 是一个用于操作高阶函数以及可调用对象的模块，里面提供了几个有意思的工具函数，非常有用。我们前面刚接触过的 *reduce* 函数，就是来自 *functools* 模块。除此之外，还有以下这些：

- *cached_property* ，带缓存功能的属性装饰器，将留到类机制部分讲解；

- *lru_cache* ，为函数提供缓存功能的装饰器；
- *partial* ，生成 **偏函数** ；
- *wraps* ，包装函数，使它看起来更像另一个函数，一般在装饰器实现时用到；
- *etc*

接下来，我们以 *partial* 为例展开讲解，看看什么是偏函数以及如何在实际中应用偏函数。

send_email 是一个调用 *smtplib* 发送邮件的函数，实现了邮件封装发送逻辑。*SMTP* 协议连接可以是普通可能是 *TCP* ，也可能是 *SSL* 连接。因此，*smtplib* 提供了两个不同的连接类，*SMTP* 以及 *SMTP_SSL* 。为了保持灵活性，*send_email* 函数也将连接对象参数化：

```
def send_email(host, port, user, password, fr, to, subject, body,
       smtp_cls=smtplib.SMTP_SSL):


   conn = smtp_cls(host, port)
```

如果采用普通 *TCP* 连接，可以这样调用 *send_email* ：

```
send_email(
   host=host,
   port=port,

   smtp_cls=smtplib.SMTP,
)
```

如果采用普通 *SSL* 连接，由于 *smtp_cls* 参数默认就是 *smtplib.SMTP_SSL* ，因此可以这样调用 *send_email* ：

```
send_email(
   host=host,
   port=port,

)
```

在某些场景，我们需要给 *smtplib.SMTP_SSL* 指定证书，例如：

```
smtplib.SMTP_SSL(
   host=host,
   port=port,

   keyfile='xxxx',
   certfile='xxxx',
)
```

这是否意味着我们需要改造 *send_email* 函数呢？是否有办法将 *smtplib.SMTP_SSL* 与两个与证书相关的参数绑定后再传给 *send_email* 函数呢？当然了，这对 *functools.partial* 来说，完全不在话下：

```
send_email(
    host=host,
    port=port,

    smtp_cls=functools.partial(smtplib.SMTP_SSL, keyfile='xxxx', certfile='xxxx'),
)
```

functools.partial 函数返回一个可调用对象，被调用时相当于调用 *smtplib.SMTP_SSL* ，而且自动附上那两个用于指定证书的参数。这样一来，当语句 `smtp_cls(host, port)` 执行时，最终等价于：

```
smtplib.SMTP_SSL(host, port, keyfile='xxxx', certfile='xxxx')
```

这就是偏函数典型的应用场景，你可能对 *functools.partial* 函数很好奇，后续我们找机会研究一番。