化名与匿名

化名

在 Python 中，我们可以给一个函数取个**化名**（alias）。

以下的代码，我们先是定义个了一个名为 _is_leap 的函数，而后为它另取了一个化名：

In [2]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

def _is_leap(year):
    return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)

year_leap_bool = _is_leap
year_leap_bool                #<function __main__._is_leap(year)>
year_leap_bool(800)           # _is_leap(800) -> True

id(year_leap_bool)            # id() 这个函数可以查询某对象的内存地址
id(_is_leap)                  # year_leap_bool 和 _is_leap 其实保存在同一个地址中，也就是说

type(year_leap_bool)
type(_is_leap)                # 它们都是 function
```

Out[2]:

```
<function __main__._is_leap(year)>
```

Out[2]:

```
True
```

Out[2]:

```
4547071648
```

Out[2]:

```
4547071648
```

Out[2]:

```
function
```

Out[2]:

```
function
```

我们可以看到的是， id(year_leap_bool) 和 id(_is_leap) 的内存地址是一样的 —— 它们是同一个对象，它们都是函数。所以，当你写 year_leap_bool = _is_leap 的时候，相当于给 _is_leap() 这个函数取了个化名。

在什么样的情况下，要给一个函数取一个化名呢？

在任何一个工程里，为函数或者变量取名都是很简单却不容易的事情 —— 因为可能会重名（虽然已经尽量用变量的作用域隔离了），可能会因取名含混而令后来者费解……

所以，仅仅为了少敲几下键盘而给一个函数取个更短的化名，实际上并不是好主意，更不是好习惯。尤其现在的编辑器都支持自动补全和多光标编辑的功能，变量名再长都不构成负担。

更多的时候，为函数取一个化名，应该是为了提高代码可读性 —— 对自己或其他人都很重要。

lambda

写一个很短的函数可以用 lambda 关键字。

下面是用 def 关键字写函数：

In [14]:

```
def add(x, y):
    return x + y
add(3, 5)
```

Out[14]:

```
8
```

下面是用 lambda 关键字写函数：

In [17]:

```
add = lambda x, y: x + y
add(3, 5)
```

Out[17]:

```
8
```

lambda 的语法结构如下：

> lambda_expr ::= "lambda" [parameter_list] ":" expression

以上使用的是 BNF 标注。当然，BNF 是你目前并不熟悉的，所以，有疑惑别当回事儿。

反正你已经见到示例了：

```
lambda x, y: x + y
```

先写上 lambda 这个关键字，其后分为两个部分，： 之前是参数，之后是表达式；这个表达式的值，就是这个函数的返回值。

> **注意**： lambda 语句中，： 之后有且只能有一个表达式。

而这个函数呢，没有名字，所以被称为"匿名函数"。

add = lambda x, y: x + y

就相当于是给一个没有名字的函数取了个名字。

lambda 的使用场景

那 lambda 这种匿名函数的用处在哪里呢？

作为某函数的返回值

第一个常见的用处是*作为另外一个函数的返回值*。

让我们看看 The Python Tutorial (https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions) 中的一个例子。

In [22]:

```
def make_incrementor(n):
    return lambda x: x + n

f = make_incrementor(42)
f(0)

f(1)
```

Out[22]:

```
42
```

Out[22]:

```
43
```

这个例子乍看起来很令人迷惑。我们先看看 f = make_incrementor(42) 之后，f 究竟是什么东西：

In [23]:

```
def make_incrementor(n):
    return lambda x: x + n

f = make_incrementor(42)
f

id(make_incrementor)
id(f)
```

Out[23]:

```
<function __main__.make_incrementor.<locals>.<lambda>(x)>
```

Out[23]:

```
4428443296
```

Out[23]:

```
4428726888
```

首先，要注意， f 并不是 make_incrementor() 这个函数的化名，如果是给这个函数取个化名，写法应该是：

```
f = make_incrementor
```

那 f 是什么呢？它是 <function __main__.make_incrementor.<locals>.<lambda>(x)> ：

> - f = make_incrementor(42) 是将 make_incrementor(42) 的返回值保存到 f 这个变量之中；
> - 而 make_incrementor() 这个函数接收到 42 这个参数之后，返回了一个函数： lambda x: x + 42 ；
> - 于是， f 中保存的函数是 lambda x: x + 42 ；
> - 所以， f(0) 是向这个匿名函数传递了 0 ，而后，它返回的是 0 + 42 。

作为某函数的参数

可以拿一些可以接收函数为参数的内建函数做例子。比如， map() (https://docs.python.org/3/library/functions.html#map)。

> map (*function*, *iterable*, ...)
>
> Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see itertools.starmap() (https://docs.python.org/3/library/itertools.html#itertools.starmap).

map() 这个函数的第一个参数，就是用来接收函数的。随后的参数，是 iterable —— 就是可被迭代的对象，比如，各种容器，例如：列表、元组、字典什么的。

In [35]:

```
def double_it(n):
    return n * 2

a_list = [1, 2, 3, 4, 5, 6]

b_list = list(map(double_it, a_list))
b_list

c_list = list(map(lambda x: x * 2, a_list))
c_list
```

Out[35]:

```
[2, 4, 6, 8, 10, 12]
```

Out[35]:

```
[2, 4, 6, 8, 10, 12]
```

显然用 lambda 更为简洁。另外，类似完成 double_it(n) 这种简单功能的函数，常常有"用过即弃"的必要。

In [60]:

```
phonebook = [
    {
        'name': 'john',
        'phone': 9876
    },
    {
        'name': 'mike',
        'phone': 5603
    },
    {
        'name': 'stan',
        'phone': 6898
    },
    {
        'name': 'eric',
        'phone': 7898
    }
]

phonebook
list(map(lambda x: x['name'], phonebook))
list(map(lambda x: x['phone'], phonebook))
```

Out[60]:

```
[{'name': 'john', 'phone': 9876},
 {'name': 'mike', 'phone': 5603},
 {'name': 'stan', 'phone': 6898},
 {'name': 'eric', 'phone': 7898}]
```

Out[60]:

```
['john', 'mike', 'stan', 'eric']
```

Out[60]:

```
[9876, 5603, 6898, 7898]
```

可以给 map() 传递若干个可被迭代对象：

In [63]:

```
a_list = [1, 3, 5]
b_list = [2, 4, 6]

list(map(lambda x, y: x * y, a_list, b_list))
```

Out[63]:

```
[2, 12, 30]
```

以上的例子都弄明白了，再去看 The Python Tutorial (https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions) 中的例子，就不会有任何疑惑了：

In [1]:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda p: p[1])
pairs
```

Out[1]:

```
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```