7.1 可接受任意数量参数的函数¶

问题¶

你想构造一个可接受任意数量参数的函数。

解决方案¶

为了能让一个函数接受任意数量的位置参数，可以使用一个*参数。例如：

```
def avg(first, *rest):
    return (first + sum(rest)) / (1 + len(rest))

# Sample use
avg(1, 2) # 1.5
avg(1, 2, 3, 4) # 2.5
```

在这个例子中，rest是由所有其他位置参数组成的元组。然后我们在代码中把它当成了一个序列来进行后续的计算。

为了接受任意数量的关键字参数，使用一个以**开头的参数。比如：

```
import html

def make_element(name, value, **attrs):
    keyvals = [' %s="%s"' % item for item in attrs.items()]
    attr_str = ''.join(keyvals)
    element = '<{name}{attrs}>{value}</{name}>'.format(
                name=name,
                attrs=attr_str,
                value=html.escape(value))
    return element

# Example
# Creates '<item size="large" quantity="6">Albatross</item>'
make_element('item', 'Albatross', size='large', quantity=6)

# Creates '<p>&lt;spam&gt;</p>'
make_element('p', '<spam>')
```

在这里，attrs是一个包含所有被传入进来的关键字参数的字典。

如果你还希望某个函数能同时接受任意数量的位置参数和关键字参数，可以同时使用*和**。比如：

```
def anyargs(*args, **kwargs):
    print(args) # A tuple
    print(kwargs) # A dict
```

使用这个函数时，所有位置参数会被放到args元组中，所有关键字参数会被放到字典kwargs中。

讨论¶

一个*参数只能出现在函数定义中最后一个位置参数后面，而 **参数只能出现在最后一个参数。 有一点要注意的是，在*参数后面仍然可以定义其他参数。

```
def a(x, *args, y):
    pass

def b(x, *args, y, **kwargs):
    pass
```

这种参数就是我们所说的强制关键字参数，在后面7.2小节还会详细讲解到。

7.10 带额外状态信息的回调函数¶

问题¶

你的代码中需要依赖到回调函数的使用(比如事件处理器、等待后台任务完成后的回调等)， 并且你还需要让回调函数拥有额外的状态值，以便在它的内部使用到。

解决方案¶

这一小节主要讨论的是那些出现在很多函数库和框架中的回调函数的使用——特别是跟异步处理有关的。 为了演示与测试，我们先定义如下一个需要调用回调函数的函数：

```
def apply_async(func, args, *, callback):
    # Compute the result
    result = func(*args)

    # Invoke the callback with the result
    callback(result)
```

实际上，这段代码可以做任何更高级的处理，包括线程、进程和定时器，但是这些都不是我们要关心的。 我们仅仅只需要关注回调函数的调用。下面是一个演示怎样使用上述代码的例子：

```
>>> def print_result(result):
...     print('Got:', result)
...
>>> def add(x, y):
...     return x + y
...
>>> apply_async(add, (2, 3), callback=print_result)
Got: 5
>>> apply_async(add, ('hello', 'world'), callback=print_result)
Got: helloworld
>>>
```

注意到 `print_result()` 函数仅仅只接受一个参数 `result` 。不能再传入其他信息。 而当你想让回调函数访问其他变量或者特定环境的变量值的时候就会遇到麻烦。

为了让回调函数访问外部信息，一种方法是使用一个绑定方法来代替一个简单函数。 比如，下面这个类会保存一个内部序列号，每次接收到一个 `result` 的时候序列号加1：

```
class ResultHandler:

    def __init__(self):
        self.sequence = 0

    def handler(self, result):
        self.sequence += 1
        print('[{}] Got: {}'.format(self.sequence, result))
```

使用这个类的时候，你先创建一个类的实例，然后用它的 `handler()` 绑定方法来做为回调函数：

```
>>> r = ResultHandler()
>>> apply_async(add, (2, 3), callback=r.handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=r.handler)
[2] Got: helloworld
>>>
```

第二种方式，作为类的替代，可以使用一个闭包捕获状态值，例如：

```
def make_handler():
    sequence = 0
    def handler(result):
        nonlocal sequence
```

```
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
    return handler
```

下面是使用闭包方式的一个例子：

```
>>> handler = make_handler()
>>> apply_async(add, (2, 3), callback=handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler)
[2] Got: helloworld
>>>
```

还有另外一个更高级的方法，可以使用协程来完成同样的事情：

```
def make_handler():
    sequence = 0
    while True:
        result = yield
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
```

对于协程，你需要使用它的 `send()` 方法作为回调函数，如下所示：

```
>>> handler = make_handler()
>>> next(handler) # Advance to the yield
>>> apply_async(add, (2, 3), callback=handler.send)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler.send)
[2] Got: helloworld
>>>
```

讨论¶

基于回调函数的软件通常都有可能变得非常复杂。一部分原因是回调函数通常会跟请求执行代码断开。 因此，请求执行和处理结果之间的执行环境实际上已经丢失了。如果你想让回调函数连续执行多步操作， 那你就必须去解决如何保存和恢复相关的状态信息了。

至少有两种主要方式来捕获和保存状态信息，你可以在一个对象实例(通过一个绑定方法)或者在一个闭包中保存它。 两种方式相比，闭包或许是更加轻量级和自然一点，因为它们可以很简单的通过函数来构造。 它们还能自动捕获所有被使用到的变量。因此，你无需去担心如何去存储额外的状态信息(代码中自动判定)。

如果使用闭包，你需要注意对那些可修改变量的操作。在上面的方案中， `nonlocal` 声明语句用来指示接下来的变量会在回调函数中被修改。如果没有这个声明，代码会报错。

而使用一个协程来作为一个回调函数就更有趣了，它跟闭包方法密切相关。 某种意义上来讲，它显得更加简洁，因为总共就一个函数而已。 并且，你可以很自由的修改变量而无需去使用 `nonlocal` 声明。 这种方式唯一缺点就是相对于其他Python技术而言或许比较难以理解。 另外还有一些比较难懂的部分，比如使用之前需要调用 `next()` ，实际使用时这个步骤很容易被忘记。 尽管如此，协程还有其他用处，比如作为一个内联回调函数的定义(下一节会讲到)。

如果你仅仅只需要给回调函数传递额外的值的话，还有一种使用 `partial()` 的方式也很有用。 在没有使用 `partial()` 的时候，你可能经常看到下面这种使用lambda表达式的复杂代码：

```
>>> apply_async(add, (2, 3), callback=lambda r: handler(r, seq))
[1] Got: 5
>>>
```

可以参考7.8小节的几个示例，教你如何使用 `partial()` 来更改参数签名来简化上述代码。

7.11 内联回调函数¶

问题¶

当你编写使用回调函数的代码的时候，担心很多小函数的扩张可能会弄乱程序控制流。 你希望找到某个方法来让代码看上去更像是一个普通的执行序列。

解决方案¶

通过使用生成器和协程可以使得回调函数内联在某个函数中。 为了演示说明，假设你有如下所示的一个执行某种计算任务然后调用一个回调函数的函数(参考7.10小节)：

```
def apply_async(func, args, *, callback):
    # Compute the result
    result = func(*args)

    # Invoke the callback with the result
    callback(result)
```

接下来让我们看一下下面的代码，它包含了一个 `Async` 类和一个 `inlined_async` 装饰器：

```
from queue import Queue
from functools import wraps

class Async:
    def __init__(self, func, args):
        self.func = func
        self.args = args

def inlined_async(func):
    @wraps(func)
    def wrapper(*args):
        f = func(*args)
        result_queue = Queue()
        result_queue.put(None)
        while True:
            result = result_queue.get()
            try:
                a = f.send(result)
                apply_async(a.func, a.args, callback=result_queue.put)
            except StopIteration:
                break
    return wrapper
```

这两个代码片段允许你使用 `yield` 语句内联回调步骤。比如：

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

@inlined_async
def test():
    r = yield Async(add, (2, 3))
    print(r)
    r = yield Async(add, ('hello', 'world'))
    print(r)
    for n in range(10):
        r = yield Async(add, (n, n))
        print(r)
    print('Goodbye')
```

如果你调用 `test()` ，你会得到类似如下的输出：

```
5
helloworld
0
```

```
2
4
6
8
10
12
14
16
18
Goodbye
```

你会发现，除了那个特别的装饰器和 `yield` 语句外，其他地方并没有出现任何的回调函数(其实是在后台定义的)。

讨论¶

本小节会实实在在的测试你关于回调函数、生成器和控制流的知识。

首先，在需要使用到回调的代码中，关键点在于当前计算工作会挂起并在将来的某个时候重启(比如异步执行)。 当计算重启时，回调函数被调用来继续处理结果。`apply_async()` 函数演示了执行回调的实际逻辑， 尽管实际情况中它可能会更加复杂(包括线程、进程、事件处理器等等)。

计算的暂停与重启思路跟生成器函数的执行模型不谋而合。 具体来讲，`yield` 操作会使一个生成器函数产生一个值并暂停。 接下来调用生成器的 `__next__()` 或 `send()` 方法又会让它从暂停处继续执行。

根据这个思路，这一小节的核心就在 `inline_async()` 装饰器函数中了。 关键点就是，装饰器会逐步遍历生成器函数的所有 `yield` 语句，每一次一个。 为了这样做，刚开始的时候创建了一个 `result` 队列并向里面放入一个 `None` 值。 然后开始一个循环操作，从队列中取出结果值并发送给生成器，它会持续到下一个 `yield` 语句， 在这里一个 `Async` 的实例被接受到。然后循环开始检查函数和参数，并开始进行异步计算 `apply_async()` 。 然而，这个计算有个最诡异部分是它并没有使用一个普通的回调函数，而是用队列的 `put()` 方法来回调。

这时候，是时候详细解释下到底发生了什么了。主循环立即返回顶部并在队列上执行 `get()` 操作。 如果数据存在，它一定是 `put()` 回调存放的结果。如果没有数据，那么先暂停操作并等待结果的到来。 这个具体怎样实现是由 `apply_async()` 函数来决定的。 如果你不相信会有这么神奇的事情，你可以使用 `multiprocessing` 库来试一下， 在单独的进程中执行异步计算操作，如下所示：

```
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()
    apply_async = pool.apply_async

    # Run the test function
    test()
```

实际上你会发现这个真的就是这样的，但是要解释清楚具体的控制流得需要点时间了。

将复杂的控制流隐藏到生成器函数背后的例子在标准库和第三方包中都能看到。 比如，在 `contextlib` 中的 `@contextmanager` 装饰器使用了一个令人费解的技巧， 通过一个 `yield` 语句将进入和离开上下文管理器粘合在一起。另外非常流行的 `Twisted` 包中也包含了非常类似的内联回调。

7.12 访问闭包中定义的变量¶

问题¶

你想要扩展函数中的某个闭包，允许它能访问和修改函数的内部变量。

解决方案¶

通常来讲，闭包的内部变量对于外界来讲是完全隐藏的。 但是，你可以通过编写访问函数并将其作为函数属性绑定到闭包上来实现这个目的。例如：

```
def sample():
    n = 0
    # Closure function
    def func():
        print('n=', n)

    # Accessor methods for n
    def get_n():
        return n

    def set_n(value):
        nonlocal n
        n = value

    # Attach as function attributes
    func.get_n = get_n
    func.set_n = set_n
    return func
```

下面是使用的例子:

```
>>> f = sample()
>>> f()
n= 0
>>> f.set_n(10)
>>> f()
n= 10
>>> f.get_n()
10
>>>
```

讨论¶

为了说明清楚它如何工作的，有两点需要解释一下。首先，`nonlocal` 声明可以让我们编写函数来修改内部变量的值。其次，函数属性允许我们用一种很简单的方式将访问方法绑定到闭包函数上，这个跟实例方法很像(尽管并没有定义任何类)。

还可以进一步的扩展，让闭包模拟类的实例。你要做的仅仅是复制上面的内部函数到一个字典实例中并返回它即可。例如：

```
import sys
class ClosureInstance:
    def __init__(self, locals=None):
        if locals is None:
            locals = sys._getframe(1).f_locals

        # Update instance dictionary with callables
        self.__dict__.update((key,value) for key, value in locals.items()
                             if callable(value) )
    # Redirect special methods
    def __len__(self):
        return self.__dict__['__len__']()
```

```
# Example use
def Stack():
    items = []
    def push(item):
        items.append(item)

    def pop():
        return items.pop()

    def __len__():
        return len(items)

    return ClosureInstance()
```

下面是一个交互式会话来演示它是如何工作的：

```
>>> s = Stack()
>>> s
<__main__.ClosureInstance object at 0x10069ed10>
>>> s.push(10)
>>> s.push(20)
>>> s.push('Hello')
>>> len(s)
3
>>> s.pop()
'Hello'
>>> s.pop()
20
>>> s.pop()
10
>>>
```

有趣的是，这个代码运行起来会比一个普通的类定义要快很多。你可能会像下面这样测试它跟一个类的性能对比：

```
class Stack2:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def __len__(self):
        return len(self.items)
```

如果这样做，你会得到类似如下的结果：

```
>>> from timeit import timeit
>>> # Test involving closures
>>> s = Stack()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
0.9874754269840196
>>> # Test involving a class
>>> s = Stack2()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
1.0707052160287276
>>>
```

结果显示，闭包的方案运行起来要快大概8%，大部分原因是因为对实例变量的简化访问， 闭包更快是因为不会涉及到额外的self变量。

Raymond Hettinger对于这个问题设计出了更加难以理解的改进方案。不过，你得考虑下是否真的需要在你代码中这样做， 而且它只是真实类的一个奇怪的替换而已，例如，类的主要特性如继承、属性、描述器或类方法都是不能用的。并且你要做一些其他的工作才能让一些特殊方法生效(比如上面 `ClosureInstance` 中重写过的 `__len__()` 实现。)

最后，你可能还会让其他阅读你代码的人感到疑惑，为什么它看起来不像一个普通的类定义呢？ (当然，他们也想知

道为什么它运行起来会更快)。尽管如此，这对于怎样访问闭包的内部变量也不失为一个有趣的例子。

总体上讲，在配置的时候给闭包添加方法会有更多的实用功能， 比如你需要重置内部状态、刷新缓冲区、清除缓存或其他的反馈机制的时候。

7.2 只接受关键字参数的函数¶

问题¶

你希望函数的某些参数强制使用关键字参数传递

解决方案¶

将强制关键字参数放到某个*参数或者单个*后面就能达到这种效果。比如：

```
def recv(maxsize, *, block):
    'Receives a message'
    pass

recv(1024, True) # TypeError
recv(1024, block=True) # Ok
```

利用这种技术，我们还能在接受任意多个位置参数的函数中指定关键字参数。比如：

```
def minimum(*values, clip=None):
    m = min(values)
    if clip is not None:
        m = clip if clip > m else m
    return m

minimum(1, 5, 2, -5, 10) # Returns -5
minimum(1, 5, 2, -5, 10, clip=0) # Returns 0
```

讨论¶

很多情况下，使用强制关键字参数会比使用位置参数表意更加清晰，程序也更加具有可读性。 例如，考虑下如下一个函数调用：

```
msg = recv(1024, False)
```

如果调用者对recv函数并不是很熟悉，那他肯定不明白那个False参数到底来干嘛用的。 但是，如果代码变成下面这样子的话就清楚多了：

```
msg = recv(1024, block=False)
```

另外，使用强制关键字参数也会比使用**kwargs参数更好，因为在使用函数help的时候输出也会更容易理解：

```
>>> help(recv)
Help on function recv in module __main__:
recv(maxsize, *, block)
    Receives a message
```

强制关键字参数在一些更高级场合同样也很有用。 例如，它们可以被用来在使用*args和**kwargs参数作为输入的函数中插入参数，9.11小节有一个这样的例子。

7.3 给函数参数增加元信息¶

问题¶

你写好了一个函数，然后想为这个函数的参数增加一些额外的信息，这样的话其他使用者就能清楚的知道这个函数应该怎么使用。

解决方案¶

使用函数参数注解是一个很好的办法，它能提示程序员应该怎样正确使用这个函数。 例如，下面有一个被注解了的函数：

```
def add(x:int, y:int) -> int:
    return x + y
```

python解释器不会对这些注解添加任何的语义。它们不会被类型检查，运行时跟没有加注解之前的效果也没有任何差距。 然而，对于那些阅读源码的人来讲就很有帮助啦。第三方工具和框架可能会对这些注解添加语义。同时它们也会出现在文档中。

```
>>> help(add)
Help on function add in module __main__:
add(x: int, y: int) -> int
>>>
```

尽管你可以使用任意类型的对象给函数添加注解(例如数字，字符串，对象实例等等)，不过通常来讲使用类或者字符串会比较好点。

讨论¶

函数注解只存储在函数的 `__annotations__` 属性中。例如：

```
>>> add.__annotations__
{'y': <class 'int'>, 'return': <class 'int'>, 'x': <class 'int'>}
```

尽管注解的使用方法可能有很多种，但是它们的主要用途还是文档。 因为python并没有类型声明，通常来讲仅仅通过阅读源码很难知道应该传递什么样的参数给这个函数。 这时候使用注解就能给程序员更多的提示，让他们可以正确的使用函数。

参考9.20小节的一个更加高级的例子，演示了如何利用注解来实现多分派(比如重载函数)。

7.4 返回多个值的函数¶

问题¶

你希望构造一个可以返回多个值的函数

解决方案¶

为了能返回多个值，函数直接return一个元组就行了。例如：

```
>>> def myfun():
... return 1, 2, 3
...
>>> a, b, c = myfun()
>>> a
1
>>> b
2
>>> c
3
```

讨论¶

尽管myfun()看上去返回了多个值，实际上是先创建了一个元组然后返回的。 这个语法看上去比较奇怪，实际上我们使用的是逗号来生成一个元组，而不是用括号。比如下面的：

```
>>> a = (1, 2) # With parentheses
>>> a
(1, 2)
>>> b = 1, 2 # Without parentheses
>>> b
(1, 2)
>>>
```

当我们调用返回一个元组的函数的时候，通常我们会将结果赋值给多个变量，就像上面的那样。 其实这就是1.1小节中我们所说的元组解包。返回结果也可以赋值给单个变量， 这时候这个变量值就是函数返回的那个元组本身了：

```
>>> x = myfun()
>>> x
(1, 2, 3)
>>>
```

7.5 定义有默认参数的函数¶

问题¶

你想定义一个函数或者方法，它的一个或多个参数是可选的并且有一个默认值。

解决方案¶

定义一个有可选参数的函数是非常简单的，直接在函数定义中给参数指定一个默认值，并放到参数列表最后就行了。例如：

```
def spam(a, b=42):
    print(a, b)

spam(1) # Ok. a=1, b=42
spam(1, 2) # Ok. a=1, b=2
```

如果默认参数是一个可修改的容器比如一个列表、集合或者字典，可以使用None作为默认值，就像下面这样：

```
# Using a list as a default value
def spam(a, b=None):
    if b is None:
        b = []
    ...
```

如果你并不想提供一个默认值，而是想仅仅测试下某个默认参数是不是有传递进来，可以像下面这样写：

```
_no_value = object()

def spam(a, b=_no_value):
    if b is _no_value:
        print('No b value supplied')
    ...
```

我们测试下这个函数：

```
>>> spam(1)
No b value supplied
>>> spam(1, 2) # b = 2
>>> spam(1, None) # b = None
>>>
```

仔细观察可以发现到传递一个None值和不传值两种情况是有差别的。

讨论¶

定义带默认值参数的函数是很简单的，但绝不仅仅只是这个，还有一些东西在这里也深入讨论下。

首先，默认参数的值仅仅在函数定义的时候赋值一次。试着运行下面这个例子：

```
>>> x = 42
>>> def spam(a, b=x):
...     print(a, b)
...
>>> spam(1)
1 42
>>> x = 23 # Has no effect
>>> spam(1)
1 42
>>>
```

注意到当我们改变x的值的时候对默认参数值并没有影响，这是因为在函数定义的时候就已经确定了它的默认值了。

其次，默认参数的值应该是不可变的对象，比如None、True、False、数字或字符串。 特别的，千万不要像下面这样写代码：

```
def spam(a, b=[]): # NO!
    ...
```

如果你这么做了，当默认值在其他地方被修改后你将会遇到各种麻烦。这些修改会影响到下次调用这个函数时的默认值。比如：

```
>>> def spam(a, b=[]):
...     print(b)
...     return b
...
>>> x = spam(1)
>>> x
[]
>>> x.append(99)
>>> x.append('Yow!')
>>> x
[99, 'Yow!']
>>> spam(1) # Modified list gets returned!
[99, 'Yow!']
>>>
```

这种结果应该不是你想要的。为了避免这种情况的发生，最好是将默认值设为None， 然后在函数里面检查它，前面的例子就是这样做的。

在测试None值时使用 `is` 操作符是很重要的，也是这种方案的关键点。 有时候大家会犯下下面这样的错误：

```
def spam(a, b=None):
    if not b: # NO! Use 'b is None' instead
        b = []
    ...
```

这么写的问题在于尽管None值确实是被当成False， 但是还有其他的对象(比如长度为0的字符串、列表、元组、字典等)都会被当做False。 因此，上面的代码会误将一些其他输入也当成是没有输入。比如：

```
>>> spam(1) # OK
>>> x = []
>>> spam(1, x) # Silent error. x value overwritten by default
>>> spam(1, 0) # Silent error. 0 ignored
>>> spam(1, '') # Silent error. '' ignored
>>>
```

最后一个问题比较微妙，那就是一个函数需要测试某个可选参数是否被使用者传递进来。 这时候需要小心的是你不能用某个默认值比如None、 0或者False值来测试用户提供的值(因为这些值都是合法的值，是可能被用户传递进来的)。 因此，你需要其他的解决方案了。

为了解决这个问题，你可以创建一个独一无二的私有对象实例，就像上面的_no_value变量那样。 在函数里面，你可以通过检查被传递参数值跟这个实例是否一样来判断。 这里的思路是用户不可能去传递这个_no_value实例作为输入。 因此，这里通过检查这个值就能确定某个参数是否被传递进来了。

这里对 `object()` 的使用看上去有点不太常见。`object` 是python中所有类的基类。 你可以创建 `object` 类的实例，但是这些实例没什么实际用处，因为它并没有任何有用的方法， 也没有任何实例数据(因为它没有任何的实例字典，你甚至都不能设置任何属性值)。 你唯一能做的就是测试同一性。这个刚好符合我的要求，因为我在函数中就只是需要一个同一性的测试而已。

7.6 定义匿名或内联函数¶

问题¶

你想为 `sort()` 操作创建一个很短的回调函数，但又不想用 `def` 去写一个单行函数， 而是希望通过某个快捷方式以内联方式来创建这个函数。

解决方案¶

当一些函数很简单，仅仅只是计算一个表达式的值的时候，就可以使用lambda表达式来代替了。比如：

```
>>> add = lambda x, y: x + y
>>> add(2,3)
5
>>> add('hello', 'world')
'helloworld'
>>>
```

这里使用的lambda表达式跟下面的效果是一样的：

```
>>> def add(x, y):
...     return x + y
...
>>> add(2,3)
5
>>>
```

lambda表达式典型的使用场景是排序或数据reduce等：

```
>>> names = ['David Beazley', 'Brian Jones',
...          'Raymond Hettinger', 'Ned Batchelder']
>>> sorted(names, key=lambda name: name.split()[-1].lower())
['Ned Batchelder', 'David Beazley', 'Raymond Hettinger', 'Brian Jones']
>>>
```

讨论¶

尽管lambda表达式允许你定义简单函数，但是它的使用是有限制的。 你只能指定单个表达式，它的值就是最后的返回值。也就是说不能包含其他的语言特性了， 包括多个语句、条件表达式、迭代以及异常处理等等。

你可以不使用lambda表达式就能编写大部分python代码。 但是，当有人编写大量计算表达式值的短小函数或者需要用户提供回调函数的程序的时候， 你就会看到lambda表达式的身影了。

7.7 匿名函数捕获变量值¶

问题¶

你用lambda定义了一个匿名函数，并想在定义时捕获到某些变量的值。

解决方案¶

先看下下面代码的效果：

```
>>> x = 10
>>> a = lambda y: x + y
>>> x = 20
>>> b = lambda y: x + y
>>>
```

现在我问你，a(10)和b(10)返回的结果是什么？如果你认为结果是20和30，那么你就错了：

```
>>> a(10)
30
>>> b(10)
30
>>>
```

这其中的奥妙在于lambda表达式中的x是一个自由变量， 在运行时绑定值，而不是定义时就绑定，这跟函数的默认值参数定义是不同的。 因此，在调用这个lambda表达式的时候，x的值是执行时的值。例如：

```
>>> x = 15
>>> a(10)
25
>>> x = 3
>>> a(10)
13
>>>
```

如果你想让某个匿名函数在定义时就捕获到值，可以将那个参数值定义成默认参数即可，就像下面这样：

```
>>> x = 10
>>> a = lambda y, x=x: x + y
>>> x = 20
>>> b = lambda y, x=x: x + y
>>> a(10)
20
>>> b(10)
30
>>>
```

讨论¶

在这里列出来的问题是新手很容易犯的错误，有些新手可能会不恰当的使用lambda表达式。 比如，通过在一个循环或列表推导中创建一个lambda表达式列表，并期望函数能在定义时就记住每次的迭代值。例如：

```
>>> funcs = [lambda x: x+n for n in range(5)]
>>> for f in funcs:
... print(f(0))
...
4
4
4
4
4
>>>
```

但是实际效果是运行是n的值为迭代的最后一个值。现在我们用另一种方式修改一下：

```
>>> funcs = [lambda x, n=n: x+n for n in range(5)]
>>> for f in funcs:
... print(f(0))
...
0
1
2
3
4
>>>
```

通过使用函数默认值参数形式，lambda函数在定义时就能绑定到值。

7.8 减少可调用对象的参数个数¶

问题¶

你有一个被其他python代码使用的callable对象，可能是一个回调函数或者是一个处理器， 但是它的参数太多了，导致调用时出错。

解决方案¶

如果需要减少某个函数的参数个数，你可以使用 `functools.partial()` 。 `partial()` 函数允许你给一个或多个参数设置固定的值，减少接下来被调用时的参数个数。 为了演示清楚，假设你有下面这样的函数：

```
def spam(a, b, c, d):
    print(a, b, c, d)
```

现在我们使用 `partial()` 函数来固定某些参数值：

```
>>> from functools import partial
>>> s1 = partial(spam, 1) # a = 1
>>> s1(2, 3, 4)
1 2 3 4
>>> s1(4, 5, 6)
1 4 5 6
>>> s2 = partial(spam, d=42) # d = 42
>>> s2(1, 2, 3)
1 2 3 42
>>> s2(4, 5, 5)
4 5 5 42
>>> s3 = partial(spam, 1, 2, d=42) # a = 1, b = 2, d = 42
>>> s3(3)
1 2 3 42
>>> s3(4)
1 2 4 42
>>> s3(5)
1 2 5 42
>>>
```

可以看出 `partial()` 固定某些参数并返回一个新的callable对象。这个新的callable接受未赋值的参数， 然后跟之前已经赋值过的参数合并起来，最后将所有参数传递给原始函数。

讨论¶

本节要解决的问题是让原本不兼容的代码可以一起工作。下面我会列举一系列的例子。

第一个例子是，假设你有一个点的列表来表示(x,y)坐标元组。 你可以使用下面的函数来计算两点之间的距离：

```
points = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

import math
def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return math.hypot(x2 - x1, y2 - y1)
```

现在假设你想以某个点为基点，根据点和基点之间的距离来排序所有的这些点。 列表的 `sort()` 方法接受一个关键字参数来自定义排序逻辑， 但是它只能接受一个单个参数的函数(distance()很明显是不符合条件的)。 现在我们可以通过使用 `partial()` 来解决这个问题：

```
>>> pt = (4, 3)
>>> points.sort(key=partial(distance,pt))
>>> points
[(3, 4), (1, 2), (5, 6), (7, 8)]
>>>
```

更进一步，`partial()` 通常被用来微调其他库函数所使用的回调函数的参数。 例如，下面是一段代码，使用 `multiprocessing` 来异步计算一个结果值， 然后这个值被传递给一个接受一个result值和一个可选logging参数的回调函数：

```
def output_result(result, log=None):
    if log is not None:
        log.debug('Got: %r', result)

# A sample function
def add(x, y):
    return x + y

if __name__ == '__main__':
    import logging
    from multiprocessing import Pool
    from functools import partial

    logging.basicConfig(level=logging.DEBUG)
    log = logging.getLogger('test')

    p = Pool()
    p.apply_async(add, (3, 4), callback=partial(output_result, log=log))
    p.close()
    p.join()
```

当给 `apply_async()` 提供回调函数时，通过使用 `partial()` 传递额外的 `logging` 参数。 而 `multiprocessing` 对这些一无所知——它仅仅只是使用单个值来调用回调函数。

作为一个类似的例子，考虑下编写网络服务器的问题，`socketserver` 模块让它变得很容易。 下面是个简单的echo服务器：

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        for line in self.rfile:
            self.wfile.write(b'GOT:' + line)

serv = TCPServer(('', 15000), EchoHandler)
serv.serve_forever()
```

不过，假设你想给EchoHandler增加一个可以接受其他配置选项的 `__init__` 方法。比如：

```
class EchoHandler(StreamRequestHandler):
    # ack is added keyword-only argument. *args, **kwargs are
    # any normal parameters supplied (which are passed on)
    def __init__(self, *args, ack, **kwargs):
        self.ack = ack
        super().__init__(*args, **kwargs)

    def handle(self):
        for line in self.rfile:
            self.wfile.write(self.ack + line)
```

这么修改后，我们就不需要显式地在TCPServer类中添加前缀了。 但是你再次运行程序后会报类似下面的错误：

```
Exception happened during processing of request from ('127.0.0.1', 59834)
Traceback (most recent call last):
...
TypeError: __init__() missing 1 required keyword-only argument: 'ack'
```

初看起来好像很难修正这个错误，除了修改 `socketserver` 模块源代码或者使用某些奇怪的方法之外。 但是，如果使用 `partial()` 就能很轻松的解决——给它传递 `ack` 参数的值来初始化即可，如下：

```
from functools import partial
serv = TCPServer(('', 15000), partial(EchoHandler, ack=b'RECEIVED:'))
serv.serve_forever()
```

在这个例子中， `__init__()` 方法中的ack参数声明方式看上去很有趣，其实就是声明ack为一个强制关键字参数。 关于强制关键字参数问题我们在7.2小节我们已经讨论过了，读者可以再去回顾一下。

很多时候 `partial()` 能实现的效果，lambda表达式也能实现。比如，之前的几个例子可以使用下面这样的表达式：

```
points.sort(key=lambda p: distance(pt, p))
p.apply_async(add, (3, 4), callback=lambda result: output_result(result,log))
serv = TCPServer(('', 15000),
        lambda *args, **kwargs: EchoHandler(*args, ack=b'RECEIVED:', **kwargs))
```

这样写也能实现同样的效果，不过相比而已会显得比较臃肿，对于阅读代码的人来讲也更加难懂。 这时候使用 `partial()` 可以更加直观的表达你的意图(给某些参数预先赋值)。

7.9 将单方法的类转换为函数¶

问题¶

你有一个除 `__init__()` 方法外只定义了一个方法的类。为了简化代码，你想将它转换成一个函数。

解决方案¶

大多数情况下，可以使用闭包来将单个方法的类转换成函数。 举个例子，下面示例中的类允许使用者根据某个模板方案来获取到URL链接地址。

```
from urllib.request import urlopen

class UrlTemplate:
    def __init__(self, template):
        self.template = template

    def open(self, **kwargs):
        return urlopen(self.template.format_map(kwargs))

# Example use. Download stock data from yahoo
yahoo = UrlTemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')
for line in yahoo.open(names='IBM,AAPL,FB', fields='sl1c1v'):
    print(line.decode('utf-8'))
```

这个类可以被一个更简单的函数来代替：

```
def urltemplate(template):
    def opener(**kwargs):
        return urlopen(template.format_map(kwargs))
    return opener

# Example use
yahoo = urltemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')
for line in yahoo(names='IBM,AAPL,FB', fields='sl1c1v'):
    print(line.decode('utf-8'))
```

讨论¶

大部分情况下，你拥有一个单方法类的原因是需要存储某些额外的状态来给方法使用。 比如，定义UrlTemplate类的唯一目的就是先在某个地方存储模板值，以便将来可以在open()方法中使用。

使用一个内部函数或者闭包的方案通常会更优雅一些。简单来讲，一个闭包就是一个函数， 只不过在函数内部带上了一个额外的变量环境。闭包关键特点就是它会记住自己被定义时的环境。 因此，在我们的解决方案中，`opener()` 函数记住了 `template` 参数的值，并在接下来的调用中使用它。

任何时候只要你碰到需要给某个函数增加额外的状态信息的问题，都可以考虑使用闭包。 相比将你的函数转换成一个类而言，闭包通常是一种更加简洁和优雅的方案。

第七章：函数¶

使用 `def` 语句定义函数是所有程序的基础。 本章的目标是讲解一些更加高级和不常见的函数定义与使用模式。 涉及到的内容包括默认参数、任意数量参数、强制关键字参数、注解和闭包。 另外，一些高级的控制流和利用回调函数传递数据的技术在这里也会讲解到。

Contents: