

进阶语法

BBCLOUD 林凡

https://github.com/lvancer/course_python

lin029011@163.com

大纲

不定参数

迭代器

匿名方法

函数式编程

装饰器

练习



不定参数

我们在前面学习的框架中，经常用到不定参数。

不定参数中，传入的参数个数是任意的，可以为0个，也可以是1个，2个，无数个。

两种形式：`*args`、`**kw`。

```
def func(*args):           # 不定参数
    print(len(args))       # 不定参数个数
    print(args)            # 参数是一个元组
    print(args[0])         # 参数获取

func('arg1', 'arg2', 4)    # 可以写入任意多个参数
```

输出：

```
3
('arg1', 'arg2', 4)
arg1
```

单个星号 `*args` 表示连续的一段参数，可以写入任意多个参数。

实际传入的变量是一个元组。使用基本的元组操作就可以获得参数信息。

lin029011@163.com

不定参数

```
def func2(**kw):           # 字典型不定参数
    print(kw)              # 参数是一个字典
    print(kw['a'])         # 获取参数

func2(a=1, b=2, c=3)      # 可以写入任意多个赋值的参数
```

输出：

```
{'a': 1, 'b': 2, 'c': 3}
1
```

两个星号 ****kw** 可以写入任意多个带名称的参数。

实际的参数会以字典式传入。

组合使用：两种不定参数可以单独使用，也可以组合使用。

```
def func3(name, *args)
def func3(name, **kw)
def func3(name, *args, **kw)
```

需要注意的是必须严格按上面的顺序。

不定参数

拆包：`*` 和 `**` 其实是一种拆包语法。

```
t = [1, 2, 3, 4, 5, 6]
func(*t)          # 列表拆包为参数
```

```
d = {'a': 1, 'b': 2}
func2(**d)        # 字典拆包为参数
```

拆包后，其参数结构就与不定参数一样了
实战中，我们以数据库连接为例，
可以将连接信息单独保存，方便维护。

```
import pymysql
connect_info = {
    'host': '127.0.0.1',
    'port': 3306,
    'user': 'root',
    'password': '123456',
    'db': 'test'
}
conn = pymysql.connect(**connect_info)
```


迭代器

迭代器：记录遍历位置。

```
l = [1, 2, 3, 4, 5]
it = iter(l)           # 创建迭代器
i = next(it)           # 获取下一个
```

iter方法从列表中创建一个迭代器。

next方法时迭代器的唯一方法，不断获取下一个值。

迭代器的遍历：

```
# 迭代器类型用于for循环
for i in it:
    print(i)
```

```
i = next(it)           # 获取下一个
while True:             # next的循环输出
    print(i)
    try:
        i = next(it)
    except StopIteration:
        break
```

迭代到没有值时，抛出**StopIteration**异常。

迭代器是能否进行**循环语句操作**的标志。列表等数据类型都有迭代器。

lin029011@163.com

迭代器

创建迭代器类： `__iter__`、`__next__`。

在任意类中复写这两个方法就可实现迭代器。

创建了一个可以生成N次方列表的迭代器。

```
iter = PowerIter(2)
print(next(iter))
print(next(iter))

for i in iter:
    print(i)
```

实例化对象后，通过next可以调用到`__next__`。

也可以直接放入循环语句中使用。

这种方式避免了多大的内存占用。

```
class PowerIter:
```

```
    def __init__(self, power, max=1000):
        self.data = 0
        self.power = power
        self.max = max
```

```
    def __iter__(self):
        return self
```

```
    def __next__(self):
        self.data += 1
        ret = self.data ** self.power
        if ret > self.max:
            raise StopIteration
        return ret
```

迭代器

生成器：`yield`。生成的就是迭代器。

通过方法实现前一个迭代器一样的功能。

```
x = power_iter(2)
print(next(x))
print(next(x))
```

`yield`就是一种`return`，但没有运行完。

通过`next`调用，直到真正的`return`就会停止生成器。方法结束。

迭代器可以转化为列表。这时迭代器会不断执行`next`直到结束，将结果放入列表中。

```
print(list(x))
```

 输出：

```
[1, 4, 9, 16, 25,
```

直接通过列表的特殊语法也可以生成一个列表。

```
[i ** 2 for i in range(100)]
```

```
def power_iter(power, max=1000):
    data = 0
    while True:
        data += 1
        ret = data ** power
        if ret > max:
            return
        yield ret          # 生成器
```


匿名方法

匿名方法：`lambda`开头定义的方法，没有方法名。

```
f1 = lambda : 0           # 无参数
f2 = lambda x: x if x > 0 else 0  # 单参数
f3 = lambda x, y: x + y      # 多参数

f3(1, 2)    # 调用
```

`lambda`后面是参数，可以跟上多个参数。

冒号后面就是方法实现，不需要进行`return`，直接返回。

调用上，匿名方法可以用变量来存储，然后用变量来调用。

这种将方法作为变量的编程方式就是函数式编程。

函数式编程

函数式编程：将方法本身作为变量进行使用。

lambda是函数式编程的基本单元之一，其实所有的方法都可以用变量赋值。

```
f = lambda x: x*x
def func(x):
    return x*x
print(f)
print(func)
x = func      # 赋值
```

输出：

```
<function <lambda> at 0x000001F2F27DB268>
<function func at 0x000001F2F27DB6A8>
```

打印出的内容表明是方法。方法可以用来赋值，调用时与lambda调用一样。

方法做为参数：如GUI中的回调方法command。

```
def funcx(f):
    return f(1)      # 调用传入的方法
print(funcx(f))     # 将方法作为参数传入
```

函数式编程

过滤：filter。

```
x = filter(lambda d: d % 2, range(10))  
print(list(x))
```

参数1：用于过滤的方法，返回True时，会放入返回值。

参数2：要过滤的列表。

返回一个迭代器，转化为列表。

```
[1, 3, 5, 7, 9]
```

函数式编程

MapReduce : **map** 、 **reduce** 。参数1是**方法**，参数2是**迭代对象**。

```
l = [3, 4, 1, 2, -1, -2, -3]

y = map(lambda x: abs(x), l)    # Map操作
print(list(y))

from functools import reduce
z = reduce(lambda a, b: a + b, l)  # Reduce操作
print(z)
```

```
[3, 4, 1, 2, 1, 2, 3]
4
```

map将迭代对象每个都执行一次方法【单参数】，返回到迭代器。

reduce则将迭代对象从第一个到最后一个是进行累计，累计方法为两参数，得到一个结果。

如例中做相加，则会将元素1与元素2相加得到结果，再与元素3相加，以此类推。

lin029011@163.com

函数式编程

方法作为返回值：

在方法内定义一个方法，作为返回值。

需要先获得方法，然后再调用。我们在GUI选择框中command使用的也是这个语法。

```
checkboxbutton = tk.Checkbutton(root, text="选择框", variable=check,
                             command=lambda : checkboxbutton_select(check))
```

一种用法是进行延时计算。【懒加载】

下面要介绍的装饰器也是其用途之一。

```
def f():
    def g():          # 方法内定义一个方法
        print('g')
    return g          # 返回一个方法
```

```
f()          # 获得方法
f()()        # 调用返回的方法
```

```
def lazy_sum(*args):
    def _sum():
        return sum(args)
    return _sum
```

```
ret = lazy_sum(1, 2, 3) # 没有马上计算
ret()                  # 调用时计算结果
```

lin029011@163.com

装饰器

装饰器：对方法进行装饰，其本质就是方法的嵌套。

先定义一个参数为方法，返回值也为方法的方法 `deco`。

`deco` 中定义了一个 `wrapper` 方法，对传递来的方法就行装饰。

重新定义的方法除了调用原方法外，还加入了新东西。

最终执行的就是嵌套了 `deco` 的新方法。

这种编程方式叫做面向切面编程【AOP】。

Python 中用 `@` 语法来做这个操作。

在方法上加上 `@deco`，

这个方法就自动变成了 `deco(func)`。

`deco` 就是一个搞简易装饰器。

```
def deco(f):  
    def wrapper():  
        print('deco')  
        return f()  
    return wrapper
```

```
def func():  
    print('func')
```

```
deco(func)()
```

```
@deco  
def func():  
    print('func')
```

`func()`

`deco`
`func`

lin029011@163.com

装饰器

定义装饰器：

`deco_name` 替换为装饰器名称。

在 **前置操作** 中，写入要在方法前执行代码。

在 **后置操作** 中，写入在方法执行后的操作。

```
from functools import wraps

def deco_name(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        # ...前置操作...
        ret = f(*args, **kwargs)
        # ...后置操作...
        return ret
    return wrapper
```

装饰器

带参数的装饰器：

让装饰器带参数的方法是

在原装饰器上再嵌套一层，传入参数。

使用时带上参数就可以了。

```
@deco_name_args(1, 2)
def ff():
    pass
```

```
def deco_name_args(a, b):
    def decorator(f):
        @wraps(f)
        def wrapper(*args, **kwargs):
            # ...前置操作...
            ret = f(*args, **kwargs)
            # ...后置操作...
            return ret
        return wrapper
    return decorator
```

装饰器

实战场景：切面编程在Web开发中比较常见，使用场景也比较多。

Log日志

```
import logging, time
def log(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        logging.info('{} call {}'.format(time.time(), f.__name__))
        return func(*args, **kwargs)
    return wrapper
```

在方法前加上这个装饰器，就能在每次执行时记录一条执行日志。

装饰器

登录验证

```
def auth(f):  
    @wraps(f)  
    def wrapper(*args, **kwargs):  
        auth = kwargs['auth']           # 获取验证信息  
        if not auth:                     # 验证失败  
            return                       # 失败处理  
        return f(*args, **kwargs)  
    return wrapper
```

在每次执行该方法前，进行登录验证，如果失败则进行处理。

练习

