nsd1912-py02-day02

关键字参数

- 调用函数时，参数直接是一个名字的形式，如arg，称作位置参数；
- 调用函数时，参数以key=val的形式提供，如key=val，称作关键字参数

```
>>> def func1(name, age):
...   print('%s is %s years old' % (name, age))
...
>>> func1('tom', 20)    # OK
tom is 20 years old
>>> func1(20, 'tom')    # 语法正确，语义不正确
20 is tom years old
>>> func1(age=20, name='tom')  # OK
tom is 20 years old
>>> func1(age=20, 'tom')  # 语法错误，关键字参数必须在后
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
>>> func1(20, name='tom')  # 错误，变量name得到多个值
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: func1() got multiple values for argument 'name'
>>> func1('tom', age=20)  # OK
tom is 20 years old
>>> func1()  # Error，参数不足
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: func1() missing 2 required positional arguments: 'name' and 'age'
>>> func1('tom', 20, 30)  # Error，参数太多
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: func1() takes 2 positional arguments but 3 were given
```

参数组

- 参数组允许调用函数时，传参的个数不固定
- 在参数前加*表示使用元组接收参数
- 在参数前加**表示使用字典接收参数

```
>>> def func1(*canshu):
...   print(canshu)
...
>>> func1()
()
>>> func1('hao', 123, 'tom', 'jerry')
('hao', 123, 'tom', 'jerry')

>>> def func2(**canshu):
...   print(canshu)
...
>>> func2()
{}
>>> func2(name='tom', age=20)
{'name': 'tom', 'age': 20}
```

- 调用函数时，在参数前加*表示将序列对象拆开，在参数前加**表示将字典转为key=val的形式

```
>>> def add(x, y):
...   return x + y
...
>>> nums = [10, 20]
>>> add(*nums)  # 拆分nums成为10和20两个参数
30
>>> d1 = {'x': 5, 'y': 8}
>>> add(**d1)  # 将d1拆为x=5, y=8
13
```

匿名函数

- 当函数非常简单，只有一行代码构成函数的代码块，可以使用匿名函数
- 匿名函数使用lambda关键字进行声明
- lambda关键字后面的名字是参数
- 匿名函数表达式的结果自动成为返回值，不需要return语句

```
>>> def add(x, y):
...   return x + y
...
>>> add(5, 10)
15

>>> myadd = lambda x, y: x + y
>>> myadd(5, 10)
15
```

filter函数

- 它接接受两个参数
- 第一个参数是函数，接受一个参数，必须返回True或False

- 第二个参数是序列对象
- filter将序列对象中的每一个数据都当成函数的参数，返回True的留下，返回False的过滤掉

```
from random import randint

def func1(x):
    # 结果要不是1，要么是0。1为真，0为假
    return x % 2

if __name__ == '__main__':
    nums = [randint(1, 100) for i in range(10)]
    print(nums)
    result1 = filter(func1, nums)
    print(list(result1))
    result2 = filter(lambda x: x % 2, nums)
    print(list(result2))
```

map函数

- 它接接受两个参数
- 第一个参数是函数，接受一个参数，返回对数据的处理结果
- 第二个参数是序列对象
- map函数在工作时，将序列对象的每一个数据交给函数处理，得到处理后的结果

```
from random import randint

def func1(x):
    return x + 1

if __name__ == '__main__':
    nums = [randint(1, 100) for i in range(10)]
    print(nums)
    result1 = map(func1, nums)
    print(list(result1))
    result2 = map(lambda x: x + 1, nums)
    print(list(result2))
```

变量的作用域

```
# 在函数外面定义的变量是全局变量，它在定义开始到程序结束一直可见可用
>>> x = 10
>>> def func1():
...   print(x)
...
>>> func1()
10

# 在函数内定义的变量是局部变量，它只能在函数内使用
>>> def func2():
...   a = 100
...   print(a)
...
>>> func2()
100
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined

# 函数内部如果存在与全局相同的变量名，函数执行时，局部变量将遮盖住全局变量
>>> def func3():
...   x = 'hello world'
...   print(x)
...
>>> func3()
hello world
>>> x
10

# 如果希望在函数内改变全局变量，需要在函数内使用 global关键字声明
>>> def func4():
...   global x
...   x = 'hello tedu'
...   print(x)
...
>>> func4()
hello tedu
>>> x
'hello tedu'
```

偏函数

- 改造现有函数，将函数的一些参数固定下来，生成新函数

```
>>> def add(a, b, c, d, e):
...   return a + b + c + d + e
...
>>> add(10, 20, 30, 40, 5)
105
>>> add(10, 20, 30, 40, 1)
101
>>> add(10, 20, 30, 40, 3)
103
>>> from functools import partial
# 用partial改造add函数，为add函数固定4个参数，生成新函数赋值给 myadd
>>> myadd = partial(add, 10, 20, 30, 40)
>>> myadd(1)
101
>>> myadd(5)
105
```

- 使用偏函数改造int函数

```
>>> int('10101100', base=2)
172
>>> int('11101100', base=2)
236
>>> int2 = partial(int, base=2)
>>> int2('10101100')
172
>>> int8 = partial(int, base=8)
>>> int16 = partial(int, base=16)
>>> int8('23')
19
>>> int16('a')
10
```

递归函数（选修）

- 在函数内部又包含对自身的调用

```
# 阶乘
5!=5x4x3x2x1
5!=5x4!
5!=5x4x3!
5!=5x4x3x2!
5!=5x4x3x2x1!
1!=1

def func1(x):
    if x == 1:
        return 1

    return x * func1(x- 1)
#          5 * func1(4)
#          5 * 4 * func1(3)
#          5 * 4 * 3 * func1(2)
#          5 * 4 * 3 * 2 * func1(1)
#          5 * 4 * 3 * 2 * 1

if __name__ == '__main__':
    print(func1(5))
```

生成器

- 可以采用生成器表达式，与列表解析语法一致

```
>>> ips = ('192.168.1.%s' % i for i in range(1, 255))
>>> ips
<generator object <genexpr> at 0x7f4f7f7df258>
>>> for ip in ips:
...   print(ip)
```

- 可以采用生成器函数。也是函数，普通函数通过return返回一个最终结果，生成器函数通过yield返回多个中间结果

```
>>> def mygen():
...   yield 10
...   a = 10 + 5
...   yield a
...   yield 'hello world'
...
>>> mg = mygen()
>>> mg
<generator object mygen at 0x7f4f7f7df308>
>>> for data in mg:
...   print(data)
...
10
15
hello world
```

模块

- 模块导入时，python将会到sys.path定义的路径下去搜索。如果搜索不到则报错

```
>>> import sys
>>> sys.path
```

- 自定义的模块可以拷贝到/usr/local/lib/python3.6/site-packages中
- 也可以定义PYTHONPATH环境变量来指定自己的模块目录

```
[root@localhost day02]# mkdir /opt/mylibs
[root@localhost day02]# cp qsort.py /opt/mylibs/
[root@localhost day02]# export PYTHONPATH=/opt/mylibs
>>> import sys
>>> sys.path
['', '/opt/mylibs', ...]
```

tarfile模块

- 用于压缩、解压缩

```
>>> import tarfile
# 压缩
>>> tar = tarfile.open('/tmp/mytest.tar.gz', 'w:gz')
>>> tar.add('/etc/hosts')
>>> tar.add('/etc/security')
>>> tar.close()
[root@localhost day02]# ls /tmp/mytest.tar.gz
/tmp/mytest.tar.gz
[root@localhost day02]# tar tvzf /tmp/mytest.tar.gz

# 解压缩，指定解压位置为 /var/tmp，如果不指定则解压到当前位置
>>> tar = tarfile.open('/tmp/mytest.tar.gz')
>>> tar.extractall(path='/var/tmp')
>>> tar.close()
[root@localhost day02]# ls /var/tmp/etc/
hosts  security
```

hashlib模块

- 用于计算哈希值

```
[root@localhost day02]# echo -n 123456 > /tmp/123.txt
[root@localhost day02]# md5sum /tmp/123.txt
e10adc3949ba59abbe56e057f20f883e  /tmp/123.txt
>>> import hashlib
>>> d1 = '123456'
>>> b1 = d1.encode()
>>> b1
b'123456'
>>> m = hashlib.md5(b1)
>>> m.hexdigest()
'e10adc3949ba59abbe56e057f20f883e'
# 也可以分批量计算 hash值
>>> m1 = hashlib.md5()
>>> m1.update(b'12')
>>> m1.update(b'34')
>>> m1.update(b'56')
>>> m1.hexdigest()
'e10adc3949ba59abbe56e057f20f883e'
```