# nsd1907_py02_day02

## 记账练习

```
(0) 收入
(1) 支出
(2) 查询
(3) 退出
请选择(0/1/2/3): 0
金额: 10000
备注: salary
(0) 收入
(1) 支出
(2) 查询
(3) 退出
请选择(0/1/2/3): 1
金额: 500
备注: buy shoes
(0) 收入
(1) 支出
(2) 查询
(3) 退出
请选择(0/1/2/3): 2
日期          收入       支出       余额       备注
2019-12-10   0          0         10000     init data
2019-12-10   10000      0         20000     salary
2019-12-10   0          500       19500     buy shoes
(0) 收入
(1) 支出
(2) 查询
(3) 退出
请选择(0/1/2/3): 3
Bye-bye


# 存储到文件中的数据，它的形式是：
# 1.把全部的记录存到一个大列表中；2.每一笔记录存在小列表中
[
    ['2019-12-10', 0, 0, 10000, 'init data'],
    ['2019-12-10', 10000, 0, 20000, 'salary'],
]
# 在大列表中取出最新余额的方法
>>> data
[['2019-12-10', 0, 0, 10000, 'init data']]
>>> data[-1]
['2019-12-10', 0, 0, 10000, 'init data']
>>> data[-1][-2]
10000
```

# 函数

## 参数

- 以key=val形式存在的参数，称作关键字参数
- 以args形式存在的参数，称作位置参数

```
>>> def func1(name, age):
...   print('%s is %s years old' % (name, age))
...
>>> func1()   # 错误，参数个数不够
>>> func1('tom', 20, 200)   # 错误，参数个数太多
>>> func1('tom', 20)   # OK
tom is 20 years old
>>> func1(20, 'tom')   # 语法正确，语义不对
20 is tom years old
>>> func1(age=20, name='tom')  # OK
>>> func1(age=20, 'tom')  # 语法错误，关键字参数必须在位置参数后
>>> func1(20, name='tom')  # 错误，因为name得到了多个值
>>> func1('tom', age=20)   # OK
```

## 参数组

- 定义函数时，参数名前加上*表示把参数放到元组中

```
# func1函数使用元组args保存参数
>>> def func1(*args):
...   print(args)
...
>>> func1()
()
>>> func1('hao')
('hao',)
>>> func1('hao', 100, 200, 'tom', 'jerry')
('hao', 100, 200, 'tom', 'jerry')
```

- 定义函数时，参数名前加上**表示把参数放到字典中

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

- 调用函数时，给参数加上*表示把参数拆成一个个的个体

```
>>> def myadd(x, y):
...   return x + y
...
>>> nums = [234, 7432]
>>> myadd(nums[0], nums[1])
7666
>>> myadd(*nums)
7666
```

- 调用函数时，给参数加上**表示把字典参数拆成一个个的个体

```
>>> def myadd(x, y):
...   return x + y
...
>>> adict = {'x': 100, 'y': 25}
>>> myadd(**adict)   # 等价于下面的写法
125
>>> myadd(x=100, y=25)
125
```

算术程序：

```
1 + 1 = 2
Very Good!!!
Continue(y/n)? y
87 + 69 = 100
Wrong Answer.
87 + 69 = 200
Wrong Answer.
87 + 69 = 300
Wrong Answer.
87 + 69 = 156
Continue(y/n)? n
Bye-bye
```

# 匿名函数

- lambda关键字后面的名称是函数参数
- 冒号后面表达式的结果是匿名函数的返回值

```
>>> def add(x, y):
...   return x + y
...
>>> add(2, 3)
5
>>> add2 = lambda x, y: x + y
>>> add2(5, 6)
11
```

## filter函数

- 它的第一个参数是函数，该函数接受一个参数，返回True或False
- 它的第二个参数是序列对象
- 序列对象的每个值作为参数传给第一个函数，返回真保留，否则丢弃

```
def func1(x):
    return True if x % 2 == 0 else False

if __name__ == '__main__':
    nums = [randint(1, 100) for i in range(10)]
    print(nums)
    result = filter(func1, nums)
    result2 = filter(lambda x: True if x % 2 == 0 else False, nums)
    print(list(result))
    print(list(result2))
```

## map函数

- 它的第一个参数是函数，该函数用于加工数据
- 它的第二个参数是序列对象
- 序列对象中的每一个数据都会传给函数进行加工，保留加工结果

```
from random import randint

def func2(x):
    return x * 2 + 1

if __name__ == '__main__':
    nums = [randint(1, 100) for i in range(10)]
    print(nums)
    result = map(func2, nums)
    result2 = map(lambda x: x * 2 + 1, nums)
    print(list(result))
    print(list(result2))
```

# 变量作用域

```
# 在函数外面定义的变量，是全局变量。全局变量从定义开始，到程序结束，一直可见可用
>>> a = 100
>>> def func1():
...     print(a)
...
>>> func1()
100
# 函数内定义的变量是局部变量。局部变量只能在函数内使用
>>> def func2():
...     b = 'hello world'
...     print(b)
...
```

```
>>> func2()
hello world
>>> print(b)   # Error
# 全局和局部存在同名变量。局部变量将会遮盖住全局变量
>>> def func3():
...   a = 200
...   print(a)
...
>>> func3()
200
>>> print(a)
100
# 如果希望在局部将全局变量改变，需要使用global关键字
>>> def func4():
...   global a
...   a = 200
...   print(a)
...
>>> func4()
200
>>> print(a)
200
```

## 偏函数

- 改造现有函数，生成新函数
- 将现有函数的某些参数固定下来，生成新函数

```
>>> def add(a, b, c, d, e):
...   return a + b + c + d + e
...
>>> add(10, 20, 30, 40, 1)
101
>>> add(10, 20, 30, 40, 8)
108
>>> add(10, 20, 30, 40, 18)
118
# 改靠现有的add函数，把前4项的值固定来，生成新函数，新函数只需要一个参数，即原函数的第5个参数
>>> from functools import partial
>>> myadd = partial(add, 10, 20, 30, 40)
>>> myadd(1)
101
>>> myadd(8)
108


# int函数默认认为传入的字符串是10进制数形式
>>> int('10')
10
>>> int('10', base=2)  # base=2，说明字符串10是2进制数
2
```

```
>>> int('11', base=2)
3
>>> int('10111100', base=2)
188
# 改造int函数，生成新函数intX，用于指字符串是X进制
>>> int2 = partial(int, base=2)
>>> int2('10111100')
188
>>> int8 = partial(int, base=8)
>>> int8('11')
9
```

## 递归函数

- 函数包含对自身的调用

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

## 快速排序

```
>>> nums = [11, 50, 51, 54, 57, 88, 68, 2, 81, 89]
# 假设第一个数是中间值，比中间值大的放到一个列表，小的放到另一个列表
>>> middle = nums[0]   # middle是数字
>>> smaller = [2]
>>> larger = [50, 51, 54, 57, 88, 68, 81, 89]
# 把3个部分拼接
>>> smaller + [middle] + larger
[2, 11, 50, 51, 54, 57, 88, 68, 81, 89]
# 继续用相同的方法把smaller列表和larger列表进行排序
# 当列表只有一项或是空列表，就不用再排序了
```

## 生成器

- 生成器表达式，与列表解析语法一样，只是用()替代[]

```
# 列表解析
>>> ['192.168.1.%s' % i for i in range(1, 255)]
>>> ('192.168.1.%s' % i for i in range(1, 255))
<generator object <genexpr> at 0x7fa4d93ae518>
>>> for ip in ('192.168.1.%s' % i for i in range(1, 255)):
...   print(ip)
```

- 函数形式的生成器。普通的函数用return返回一个值；生成器函数用yield返回很多中间值

```
>>> def mygen():
...   yield 10
...   yield 200
...   a = 100 + 200
...   yield a
...   yield 'hello world'
...
>>> mg = mygen()
>>> mg
<generator object mygen at 0x7fa4d1a13eb8>
>>> for i in mg:
...   print(i)
```