舒文的python学习笔记

前言

呕心沥血3日所得,希望得到大家支持,该笔记已完全开源点击右上角github标志可以见到源文件,感谢你学习这么好,长得这么好看还来支持我.可以的话请舒文喝冰可乐咧!谢谢你

关舒文

第一章 程序设计方法

程序设计语言

程序设计语言包含 3 大类:

- 机器语言(一种二进制语言)
- 汇编语言
- 高级语言(包含C,C++,C#,Go等)

> 其中机器语言和汇编语言统称为低级语言

编译和解释

高级语言按照计算机执行方式不同可分为:

- 静态语言(C,C++,Java)
- 脚本语言(JavaScript,PHP,python)

静态语言采用**编译**运行(**编译器**执行编译),脚本语言采用**解释**(**解释器**直接解释)执行.

编译的好处

1. 对于相同源代码,目标代码执行速度更快;
2. 目标代码不需要解释器即可运行,在同类型操作系统上可灵活使用.

解释的好处

1. 只需要保留原代码,程序纠错和维护十分方便;
2. 只需要存在解释器,源代码可以在任何平台运行,可移植性较好.

第二章 python程序基本语法

程序的框架

python使用**缩进**来表明程序的格式框架

注释

使用 `#` 来表示单行注释, `'''` 表示多行注释:

```
print('hello world') #单行注释

'''
这是一个多行注释
'''

def hello():
    '''函数注释常用多行注释'''
```

命名与保留字

python语言允许使用 大写字母，小写字母，数字，下划线，汉字 等字符或字符组合给变量命名.*不允许将数字用作名称首字符.变量命名区分大小写*.特别的,变量命名不能与保留字相同(注意大小写):

and	as	assert	break	class	continue
def	del	elif	else	except	finally
for	from	False	global	if	import
in	is	lambda	nonlocal	not	None
or	pass	raise	return	try	True
while	with	yield			

字符串索引与切片

索引操作

用 双引号" 或 单引号' 表示字符串,例如字符串 "Hello" 和 'world' .对字符串

```
>>> Tempstr = "Hello world"
```

的第一个字符 'H' 有两种索引方式:

```
Temstr[0] = "H" #正向索引,第一个字符为0
Temstr[-11] = "H" #反向索引最后一个字符为-1
```

切片操作

我们可以通过切片操作来实现字符串的区间访问或列表的区间访问:

```
object[start_index : end_index : step] #object可为字符串或列表
```

> **start_index**: 表示起始索引(包含该索引本身);该参数省略时,表示从对象"端点"开始取值,至于是从“起点”还是从"终点"开始,则由step参数的正负决定,step为正从"起点"开始,为负从"终点"开始.
>
> **end_index**: 表示终止索引(不包含该索引本身);该参数省略时, 表示一直取到数据"端点", 至于是到"起点"还是到"终点", 同样由step参数的正负决定, step为正时直到"终点", 为负时直到"起点".
>
> **step**: 正负数均可,其绝对值大小决定了切取数据时的"步长",而正负号决定了"切取方向",正表示"从左往右"取值,负表示"从右往左"取值.当step省略时,默认为1,即从左往右以增量1取值.
> 切片操作例子:

```
>>> Tempstr = "hello"
>>> Templist = [1, 2, 3, 4, 5, 6]
# 正向切片
>>> Tempstr[1::3] = "eo"
>>> Templist[-5:5:2] = [2,4]
# 反向切片
>>> Tempstr[::-1] = "olleh"
# 矛盾切片
>>> a[-1:6]= []
# start_index=-1在end_index=6的右边，因此从右往左取值，但step=1则决定了从左往右取值，两者矛盾.
# 正负索引混合切片
>>> a[-1:6:-1] =  [9, 8, 7]
# start_index=-1在end_index=6的右边，因此从右往左取值，而step=-1同样决定了从右往左取值.
```

赋值语句

python语言中，`=` 用于赋值,即将等号右侧的计算结果赋值给左侧变量.例如:

```
i = 0 #将0赋给变量 i
```

同步赋值语句:

```
<变量 1>,...,<变量 N> = <表达式 1>,...,<表达式 N>
```

我们可以通过同步赋值语句来实现两个变量的交换:

```
x, y = y, x
```

第三章 基本数据类型

数字类型

整数

默认情况下使用的是十进制,四种进制的表示如下表:

进制类型	引导符号	描述	例子
十进制 Decimal	无	数字0-9组成	12345, 7829
二进制 Binary	0b 或 0B	数字0-1组成	0b101, 0B110
八进制 Octal	0o 或 0O	数字0-7组成	0o711, 0O127
十六进制 Hexadecimal	0x 或 0X	数字0-9,字母a-f(A-F)组成	0xABC

浮点数

浮点数表示带有小数的数值,python要求*浮点数必须带有小数部分*以便区分浮点数和整数,有两种表示方法:

- 十进制表示法: `1999.0` , `0.0` , `-77.` ;
- 科学计数法: `4.3e-3` , `9.6e5` .

浮点数的特点

对于科学计数法 $a\mathrm{E}\pm b=a\times 10^b$,在python中系数$a$最长可输出16个数字;十进制表示的浮点数最长可输出17个数字.

> 根据 `sys.float_info` 的结果计算机能够提供15个数字的准确性,最后一位存在误差.

复数

python语言中,复数的虚数部分通过后缀"J"或"j"表示,复数表示采用以下形式:

```
z = (a+bj) # a为实部,b为虚部
```

*需要特别注意,复数类型中的实数部分和虚数部分都是**浮点**类型.*
可以使用 `real` 操作获取复数实部, `imag` 操作获取虚部:

```
>>> (1.23e-4+5j).real
0.000123
>>> (1.23e-4+5j).imag
5.0
```

数字类型操作

内置数值运算操作符

操作符	描述	对应增强赋值操作符
x - y	x与y之差	-=
x * y	x与y之积	*=
x + y	x与y之和	+=
x / y	x与y之商	/=
x // y	x整除y	//=
x % y	x除以y的余数	%=
x ** y	x的y次幂,即x^y	**=

增强赋值操作符可简化代码表达:

> `x op = y` 等价于 `x = x op y` ,其中op为增强复制操作符

函数	描述	输出
abs(x)	x的绝对值,若x为复数则返回模	数值
divmod(x, y)	返回包含整除,求余结果的二元组形式(也称为元组类型)	(x//y,x%y)
pow(x, y[, z])	(X**y)%z,即 pow(x,y),它与x**y相同	数值
round(x[, ndigits])	对x四舍五入,保留 ndigits 位小数.round(x)返间四舍五入的整数	数值
max(x1,x2,...,xn)	值x1,x2,...,xn的最大值,n没有限定co	数值
min(x1,x2,...,xn)	值x1,x2,...,xn的最大值,n没有限定	数值

特别的, `pow()` 函数的第三个参数在加解密算法中十分重要,见如下对比:

```
>>> pow(3, pow(3, 999), 10000) # 幂运算和模运算同时进行,速度较快
4587
>>> pow(3, pow(3, 999))%10000 # 计算机较难运行,两种表述算术意义上等价
4587
```

内置数字类型转换

函数	合法输入	输出
int(x)	浮点数,整数,字符串	整数
float(x)	浮点数,整数,字符串	浮点数
complex(实部[,虚部])	实部:整数,浮点数,字符串;虚部:整数,浮点数(不允许字符串)	复数

> `int()` 会对浮点数进行截尾,不进行四舍五入操作;
>
> `int()` , `float()` 不允许输入复数.

`math` 库

常用数学常数:

常数	数学表示	描述
math.pi	π	圆周率
math.e	e	自然对数
math.inf	∞	正无穷大
math.nan		非浮点数标记,NaN

数值函数:

函数	数学表示	描述
math.fabs(x)	$\mid x \mid$	x的绝对值
math.fmod(x,y)	$x \% y$	x求余y
math.fsum([x,y,...])	$x + y + \cdots$	浮点数精确求和
math.ceil(x)	$\lceil x \rceil$	向上取整
math.floor(x)	$\lfloor x \rfloor$	高斯取整
math.factorial(x)	$x!$	x的阶乘
math.gcd(a,b)		a,b的公约数

函数	数学表示	描述
math.isfinite(x)		当x不是无穷大或NaN时为真
math.isnan(x)		当x是NaN时为真

幂对数函数:

函数	数学表示	描述
math.pow(x,y)	x^y	x的y次幂
math.exp()	e^x	e的x次幂
math.sqrt()	$\sqrt{x}$	x的平方根
math.log(a[,b])	$\log_b a$	a以b为底的对数,b省略时为$\ln x$

三角运算:

函数	数学表示	描述
math.degrees(x)		弧度转角度
math.radians(x)		角度转弧度
math.hypot(x,y)	$\sqrt{x^2+y^2}$	坐标(x,y)离原点距离

字符串类型和基本操作

字符串的表示

- 单引号表示 `'hello'`
- 双引号表示 `"hello"`
- 三引号表示 `'''hello'''`

1. 单引号或双引号都可以作为字符串的一部分,例如

```
'you can print "hello" by "print("hello")"' # 反正就是可以套娃啦
```

2. 三引号可用于表示多行字符串,也可以用于表示多行注释(必须以 `'''` 开头)

```
>>> print('''
        hello
        world
        ''')
                # 注意:此处为空行, '''后的第一个字符为换行符"\n"
hello
world
```

字符串的输入

输入语法:

```
<变量1> = input(<提示字符串>)
```

例如:

```
>>> name = input("请输入您的名字 ")
请输入您的名字 | # 将会打印提示并产生一个光标接受输入
```

字符串操作

基本字符串操作符:

操作符	描述
x + y	连接两个字符串x与y
x * n 或 n * x	复制n次字符串x
x in s	如果x是s的子串, 返回True
str[i]	否则返回False 索引, 返回第t个字符
str[N:M]	切片, 返回索引第N到第M的子串, 其中不包含M

特殊格式化控制字符:

操作符符	作用
\a	蜂鸣, 响铃.
\b	回退, 向后退一格.
\f	换页.
\n	换行, 光标移动到下行首行.
\r	回车, 光标移动到本行首行.
\t	水平制表.
\v	垂直制表.
\0	NULL,什么都不做.

内置字符串处理函数

函数	描述
len(x)	返回字符串 x 的长度, 也可返回其他组合数据类型元素个数
str(x)	返回任意类型 x 所对应的字符串形式

函数	描述
chr(x)	返回Ascii编码x 对应的单字符
ord(x)	返回单字符表示的Asciii编码
hex(x)	返回整数 x 对应十六进制数的小写形式字符串
oct(x)	返回整数 x 对应八进制数的小写形式字符串

> ASCII码表中: 大写字母 `A~Z` 对应 `65~90` ,小写字母 `a~z` 对应 `97~122`

内置字符串处理方法

↓---**编者水平有限,该段内容存在问题,有待研究,请跳过**---

python中对数据的处理可分为函数与方法.函数常以 `<库>.<函数>(待处理对象)` 的形式实现

```
>>> math.floor(23.4)
23
```

方法常为面向对象的,其通常以 `<对象>.<方法>(方法选项)`

```
>>> "Hello world".center(20,'😀')
'😀😀😀😀Hello world😀😀😀😀😀'
```

当然,方法与函数都可看作为函数,只不过他们的定义方式有所不同.

↑---**编者水平有限,该段内容存在问题,有待研究,请跳过**---

常用的内置字符串处理方法(str表示字符串变量)

方法	描述
str.lower()	返回字符串str的副本, 全部字符小写
str.upper()	返回字符串str的副本, 全部字符大写
str.islower()	当str所有字符都是小写时为真
str.isprintable()	当str所有字符都是可打印的时候为真
str.isnumeric()	当str所有字符都是数字的时候为真
str.isspace()	当str所有字符都是空格,返回True, 否则返回False
str.endswith(suffix [,start[,end]])	str[start: end]以suffix结尾返回True, 否则返回False
str.startswith(prefix[,start[,end]])	str[start: end]以prefix开始返回True, 否则返回False
str.split(sep=None, maxsplit=-1)	返回一个列表,由str根据sep被分隔的部分构成
str.count(sub[,start[,end]])	返回str[start: end]中sub子串出现的次数

方法	描述
str.replace(old, new[, count])	返回字符串str的副本, 所有old子串被替换为new, 如果count给出, 则前count次old出现被替换
str.center(width[,fillchar])	字符串居中函数,返回长度为width得字符串,str位于中间位置,以fillchar填充两边
str.strip([chars])	返回字符串str的副本, 在其左侧和右侧去掉chars中列出的字符
str.zfill(widlh)	返回字符串str的副本, 长度为width,不足部分在左侧补0,如果左侧为"+","-"则从第二个字符开始补0
str.format()	返回字符串str的一种排版格式
str.join(iterable)	返回一个新字符串, 由组合数据类型terable变址的每个元素组成, 元素间用str分隔

例子:

```
>>> "Python is an excellent language.".split()
['Python','is','an','excellent','language'.]

>>> "PYTHON".center(40,'=')
'=================PYTHON=================='

>>> "123".zfill(40)
'0000000000000000000000000000000000000123'

>>> "-123".zfill(40)
'-000000000000000000000000000000000000123'
```

字符串的 format() 格式化

format() 方法的使用

基本格式:

```
<模板字符串>.format(<以逗号分割的参数(字符串或数值表达式)>)
```

该方法的基本思想是将 format() 方法中用逗号分隔的参数按照序号关系替换到模板字符串的槽中,槽用大括号 {} 表示.大括号中可以指定使用参数的序号,按照序号替换.注意 format() 中的序号从0开始.
实例:

```
>>> "{1} {0}".format("hello", "world")
"world hello"
```

槽的格式控制

我们可以在槽中输入各种控制字符以进行字符格式化:

```
{ <参数序号> : <填充><对齐><宽度><千分位分隔符><浮点精度或字符串最大长度><数值类型>}
```

示例:

```
>>> "{:😀^20}".format("hello world")
'😀😀😀😀hello world😀😀😀😀😀'
```

```
>>> "{0:😀^20,.3f}".format(12345.67890)
' 😀😀😀😀😀12,345.679😀😀😀😀😀'
```

- **引导符号 :** 用 `:` 来引导后续格式控制标记
- **填充** 指定宽度(必须指定对齐方式)内除了参数外的字符采用什么方式表示, 默认采用空格,可以通过填充替换,
- **对齐** `<` 左对齐, `>` 右对齐, `^` 居中对齐
- **宽度** 指当前槽的设定输出字符宽度, 如果该槽对应的format()参数长度比 < 宽度 > 设定值大, 则使用参数实际长度;如果该值的实际位数小于指定宽度, 则位数将以**填充**字符串进行补充.
- **千分位分隔符** 以逗号作为千分位分割符(输入 `,`)
- **精度** 以 `.` 开头后加数字.对于浮点数,表示输出的有效位数,对于字符串表示最大输出长度(超出部分被截尾)
- **类型**
    - 对于整数类型:
        - b: 输出整数的二进制方式.
        - C: 输出整数对应的Unicode字符.
        - d: 输出整数的十进制方式.
        - 0: 输出整数的八进制方式.
        - X: 输出整数的小写十六进制方式.
        - X: 输出整数的大写十六进制方式.
    - 对于浮点数类型:
        - e: 输出浮点数对应的小写字母e的指数形式.
        - E: 输出浮点数对应的大写字母E的指数形式.
        - f: 输出浮点数的标准浮点形式.
        - %: 输出浮点数的百分形式.

第四章 程序的控制结构

if 条件语句

```
if <condition_1>: # 注意添加":"
    <statement_block_1>
elif <condition_2>: # 注意添加":"
    <statement_block_2>
else: # 注意添加":"
    <statement_block_3>
```

1. 如果 `<condition_1>` 为 True 将执行 `<statement_block_1>` 块语句
2. 如果 `<condition_1>` 为False, 将判断 `<condition_2>`
3. 如果 `<condition_2>` 为 True 将执行 `<statement_block_2>` 块语句
4. 如果 `<condition_2>` 为False, 将执行 `<statement_block_3>` 块语句

`if` 常用的操作符:

操作符	描述
<	小于
<=	小于或等于
>	大于
>=	大于或等于

操作符	描述
==	等于, 比较两个值是否相等
!=	不等于

循环语句

for 循环

```
for <variable> in <sequence>:
    <statements>
else:
    <additional_statements>
```

当for循环正常执行之后, 程序会继续执行else语句中的内容,若循环非正常退出则不执行.

while 循环

```
while <expr>:
    <statement>
else:
    <additional_statement>
```

当while循环正常退出(此时 `<expr>` 的值为 `False`)后,将会执行else语句中的内容.

特别的,我们可以通过创造永真的条件 `<expr>` 来实现无限循环,示例:

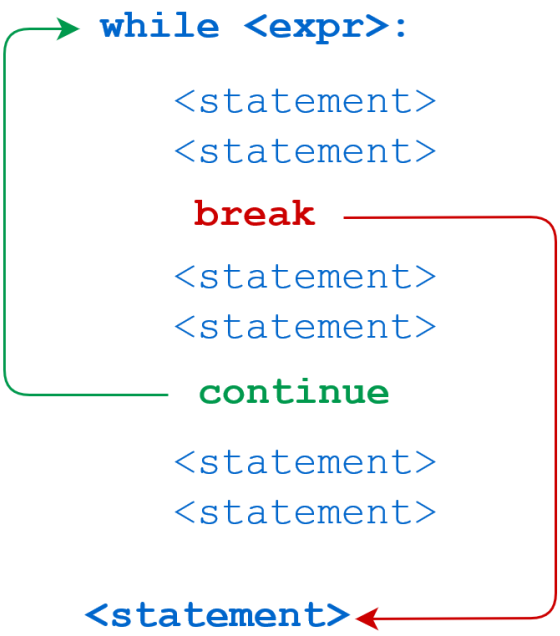
```
# 制作一个不断更新的时钟
import time

while True: # 条件永远为真,循环一直进行,直至人为中断
    time.sleep(1) # 时间等待一秒(更新间隔为1s)
    print('\r', # 刷新输入(原理类似于进度条)
        time.strftime('%Y-%m-%d %H:%M:%S',
                    time.localtime(time.time())
                    ), # 获取时间并格式化
        end=''
        )
```

循环保留字: `break` , `continue`

循环结构有两个保留字: `break` 和 `continue` , 它们用来辅助控制循环执行. 具体作用如下:

- `break` 语句可以跳出 for 和 while 的循环体. 如果你从 for 或 while 循环中终止, 任何对应的循环 else 块将不执行.

- `continue` 语句被用来告诉 Python 跳过当前循环块中的剩余语句, 然后继续进行下一轮循环.



函数: `range()`

在for循环中可用 `range()` 指定循环遍历的范围.用法:

```
range([起始数值, ]结束数值[, 步长])
```

以下 n,a,b,x 均为整数

- `range(n)` (仅指定结束数值)表示取**整数**$1, 2, 3, \cdots, n-1$
- `range(a, b)` (指定起始数值)表示取**整数**$a, a+1, \cdots, b-1$
- `range(a, b, x)` 表示取**整数**$a, a+x, \cdots, a+kx$(其中$a+kx \leq b$)

`random` 库的使用

函数	描述
seed(a=None)	初始化随机数种子, 默认值为当前系统时间
random()	生成一个 [0.0,1.0)之间的随机小数
randint(a, b)	生成一个[a,b]之间的整数
getrandbits(k)	生成一个K比特长度的随机整数
randrange(start, stop[, step])	生成一个[start, stop)之间以step为步数的随机整数
uniform(a, b)	生成一个[a, b]之间的随机小数
choice(seq)	从序列类型, 例如列表中随机返回 - 个元素
shuffle(seq)	将序列类型中的元素随机排列, 返回打乱后的序列
sample(pop, k)	从pop类型中随机选取k个元素,以列表类型返回

> 生成随机数之前可以通过 `seed()` 函数指定随机数种子,随机数种子一般是一个整数, 只要种子相同, 每次生成的随机序列也相同. 这种情况便于测试和同步数据

异常处理

```
try:
    <执行代码>
except <异常1> as <异常变量1>:
    <发生错误1时执行的代码>
except <异常2> as <异常变量2>:
    <发生异常2时执行的代码>
except:
    <发生其他异常时执行的代码>
else:
    <没有异常时执行的代码>
finally:
    <无论是否发生异常均执行的代码>
```

可以使用 `except...as..:` 语句实现将错误信息返回至一个变量的效果

```
>>> try:
        a = []
        print(a[1])
    except Exception as err:
        print(err)

'list index out of range' # 索引超出范围
```

可以实现将所有错误信息返回至 `err` 并打印.
以下是常见的错误代码:

异常代码	异常原因
AttributeError	属性错误, 特性引用和赋值失败时会引发属性错误
NameError	试图访问的变量名不存在
SyntaxError	语法错误, 代码形式错误
Exception	所有异常的基类, 因为所有python异常类都是基类Exception的其中一员, 异常都是从基类Exception继承的, 并且都在exceptions模块中定义.
IOError	一般常见于打开不存在文件时会引发IOError错误, 也可以解理为输出输入错误
KeyError	使用了映射中不存在的关键字(键)时引发的关键字错误
IndexError	索引错误, 使用的索引不存在, 常索引超出序列范围, 什么是索引
TypeError	类型错误, 内建操作或是函数应于在了错误类型的对象时会引发类型错误
ZeroDivisonError	除数为0, 在用除法操作时, 第二个参数为0时引发了该错误
ValueError	值错误, 传给对象的参数类型不正确, 像是给int()函数传入了字符串数据类型的参数.

异常抛出

使用 `raise` 保留字来抛出异常,常用于try语句中自定义返回错误:

```
raise [Exception [, args [, traceback]]]
```

示例:

```
>>> raise Exception('出错了啦つ‿‿⊂,快去debug')
# 错误提示↓
----------------------------------------
Exception
Traceback (most recent call last)
<ipython-input-76-c07bf7fa8af7> in <module>
----> 1 raise Exception('出错了啦つ‿‿⊂,快去debug')

Exception: 出错了啦つ‿‿⊂,快去debug
```

也可以结合try语句使用

```
>>> try:
        raise Exception('咦?`(*>﹏<*)′′')
    except Exception as err:
        print(err)

咦?`(*>﹏<*)′
```

第五章 函数与代码复用

定义函数

Python 定义函数使用 `def` 保留字, 一般格式如下:

```
def 函数名(参数列表):
    '''函数帮助信息'''
    函数体
    return <变量>
```

> **注意:**
>
> 1. 函数代码块以 def 关键词开头, 后接函数标识符名称和圆括号 ().
> 2. 任何传入参数和自变量必须放在圆括号中间, 圆括号之间可以用于定义参数.
> 3. 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明.
> 4. 函数内容以冒号起始, 并且缩进.
> 5. `return [表达式]` 结束函数, 选择性地返回一个值给调用方. 不带表达式的return相当于返回 None.

调用函数

函数的调用过程

1. 调用程序在调用处暂停执行.
2. 在调用时将实参复制给函数的形参.
3. 执行函数体语句.
4. 函数调用结束给出返回值, 程序回到调用前的暂停处继续执行.

函数的调用方式

```
<函数名>(参数)
```

例如:

```
>>> def pra(age):
        print(age)

    pra(13)
13
```

参数传递

在 python 中, 类型属于对象, 变量是没有类型的:

```
a = [1,2,3]
a = "hello"
```

以上代码中, [1,2,3] 是 List 类型, "hello" 是 String 类型, 而变量 a 是没有类型, 她仅仅是一个对象的引用(一个指针), 可以是指向 List 类型对象, 也可以是指向 String 类型对象.

可更改(mutable)与不可更改(immutable)对象

在 python 中, strings, tuples, 和 numbers 是不可更改的对象, 而 list,dict 等则是可以修改的对象.

- **不可变类型**: 变量赋值 a=5 后再赋值 a=10, 这里实际是新生成一个 int 值对象 10, 再让 a 指向它, 而 5 被丢弃, 不是改变 a 的值, 相当于新生成了 a.
- **可变类型**: 变量赋值 la=[1,2,3,4] 后再赋值 la[2]=5 则是将 list la 的第三个元素值更改, 本身la没有动, 只是其内部的一部分值被修改了.

**python 函数的参数传递: **

- **不可变类型**: 类似 C++ 的值传递, 如 整数、字符串、元组. 如 fun(a), 传递的只是 a 的值, 没有影响 a 对象本身. 如果在 fun(a))内部修改 a 的值, 则是新生成来一个 a. 示例(`id()` 函数可显示内存地址):

  ```
  def change(a):
      print(id(a))   # 指向的是同一个对象
      a=10
      print(id(a))   # 一个新对象

  a=1
  print(id(a))
  change(a)
  ```

  返回:

  ```
  94127348335360
  94127348335360
  94127348335648
  ```

  可以看见在调用函数前后, 形参和实参指向的是同一个对象(对象 id 相同), 在函数内部修改形参后, 形参指向的是不同的 id.

- **可变类型**: 类似 C++ 的引用传递, 如 列表, 字典. 如 fun(la), 则是将 la 真正的传过去, 修改后 fun 外部的 la 也会受影响.示例:

  ```
  def changeme( mylist ):
      "修改传入的列表"
      mylist.append([1,2,3,4])
      print ("函数内取值: ", mylist)
      return

  # 调用changeme函数
  mylist = [10,20,30]
  changeme( mylist )
  print ("函数外取值: ", mylist)
  ```

返回:

```
函数内取值:  [10, 20, 30, [1, 2, 3, 4]]
函数外取值:  [10, 20, 30, [1, 2, 3, 4]]
```

传入函数的和在末尾添加新内容的对象用的是同一个引用. 故输出结果相同.

鉴于传递的特征,我们可以使用 `global <变量>` 声明来将局部变量的改变应用到全局变量中.

参数

以下是调用函数时可使用的正式参数类型:

- 必需参数
- 关键字参数
- 默认参数
- 不定长参数

必需参数

```
def func(a):
    print(a)

func('hello')
```

该函数中参数 `a` 已被调用,故使用该函数时必须输入参数 `a` ,这种参数称为**必须参数**.必需参数须以正确的顺序传入函数. 调用时的数量必须和声明时的一样.

关键字参数

关键字参数和函数调用关系紧密, 函数调用使用关键字参数来确定传入的参数值. 使用关键字参数允许函数调用时参数的顺序与声明时不一致, 因为 Python 解释器能够用参数名匹配参数值. 使用方法:

```
def func(a):
    print(a)

func(a='hello')
```

默认参数(可选参数)

调用函数时, 如果没有传递参数, 则会使用默认参数. 可以通过指定参数的默认值,来避免没有该参数输入时发生错误.

```
def func(a, b=10):
    print(a+b)
# 传入b的值
func(1,2)
# 不传入b,b使用默认值
func(1)
```

返回:

```
3
11
```

不定长参数(可变数量参数)

当我们需要一个函数能处理比当初声明时更多的参数. 这些参数叫做不定长参数.

- **带一个 `*` 的不定长参数:** 通过在最后一个参数前增加 `*` 实现.

  特点:

1. 不定长参数一定为最后一个参数
2. 该参数将会以 元组 的形式传入函数
3. 如果在函数调用时没有指定参数, 它就是一个空元组. 我们也可以不向函数传递未命名的变量

示例:

```
def mean(a,*b):
    '求平均值'
    for i in b:
        a+=i
    print(a/(len(b)+1))
mean(1)
mean(1,2,3,4))
```

返回:

```
1.0
2.5
```

- **带两个 * 的不定长参数:** 通过在最后一个参数前增加 ** 实现.

区别:

1. 该参数将会以 字典 的形式传入函数

```
def printinfo( arg1, **vardict ):
    "打印任何传入的参数"
    print ("输出: ")
    print (arg1)
    print (vardict)

# 调用printinfo 函数
printinfo(1, a=2,b=3)
```

返回:

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

- * **单独出现的不定长参数** 声明函数时, 参数中星号 * 可以单独出现,*如果单独出现星号, * 后的参数必须用关键字参数方式传入.*

示例:

```
>>> def f(a,b,*,c):
        return a+b+c

>>> f(1,2,3)   # 报错
------------------------------------------
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: f() takes 2 positional arguments but 3 were given

>>> f(1,2,c=3) # 正常
6
```

匿名函数

定义方法:

```
<函数名> = lambda <参数列表>: <表达式>
```

特点:

1. lambda 只是一个表达式, 函数体比 def 简单很多.
2. lambda的主体是一个表达式, 而不是一个代码块. 仅仅能在lambda表达式中封装有限的逻辑进去.
3. lambda 函数拥有自己的命名空间, 且不能访问自己参数列表之外或全局命名空间里的参数.

代码复用与模块化设计

模块化设计指通过函数或对象的封装功能将程序划分成主程序、子程序和子程序间关系的表达. 模块化设计是使用函数和对象设计程序的思考方法, 以功能块为基本单位, 一般有以下两个基本要求.

1. 紧耦合: 尽可能合理划分功能块, 功能块内部耦合紧密.
2. 松耦合: 模块间关系尽可能简单, 功能块之间耦合度低.

> 使用函数只是模块化设计的必要不充分条件, 根据计算需求合理划分函数十分重要. 一般来说, 完成特定功能或被经常复用的一组语句应该采用函数来封装, 并尽可能减少函数间参数和返回值的数量.

函数的递归

我们看看数学中的递归:

$$f(x) = \begin{cases} 1 & , x = 0 \\ f(x+1) & , \text{otherwise} \end{cases}$$

$x = 0$称为递归的基例,我们可以用python函数实现以上的思想:

例1: 计算阶乘
数学表述:

$$n! = \begin{cases} 1 & , n = 0 \\ n(n-1)! & , \text{otherwise} \end{cases}$$

python实现:

```
def fact(n):
    if n == 0:
        return 1 # 基例
    else:
        return n*fact(n-1)

print(fact(10))
```

输出:

```
10
```

例2: 汉诺塔问题
有三个立柱A、B、C. A柱上穿有大小不等的圆盘N个, 较大的圆盘在下, 较小的圆盘在上. 要求把A柱上的圆盘全部移到C柱上, 保持大盘在下、小盘在上的规律(可借助B柱). 每次移动只能把一个柱子最上面的圆盘移到另一个柱子的最上面. 请输出移动过程.
递归逻辑:
A, B, C三个圆柱, 分别为初始位, 过渡位, 目标位, 设A柱为初始位, C位为最终目标位

1. 将最上面的n-1个圆盘从初始位移动到过渡位
2. 将初始位的最底下的一个圆盘移动到目标位
3. 将过渡位的n-1个圆盘移动到目标位
   python实现:

```
def move(n,a,b,c):   #n为圆盘数，a代表初始位圆柱，b代表过渡位圆柱，c代表目标位圆柱
        if n==1:
                print(a,'-->',c)
        else:
                move(n-1,a,c,b)
        '''将初始位的n-1个圆盘移动到过渡位，此时初始位为a，上一级函数的过渡位b即为
        本级的目标位，上级的目标位c为本级的过渡位'''
                print(a,'-->',c)

                move(n-1,b,a,c)
         '''将过渡位的n-1个圆盘移动到目标位，此时初始位为b，上一级函数的目标位c即为
         本级的目标位，上级的初始位a为本级的过渡位'''
```

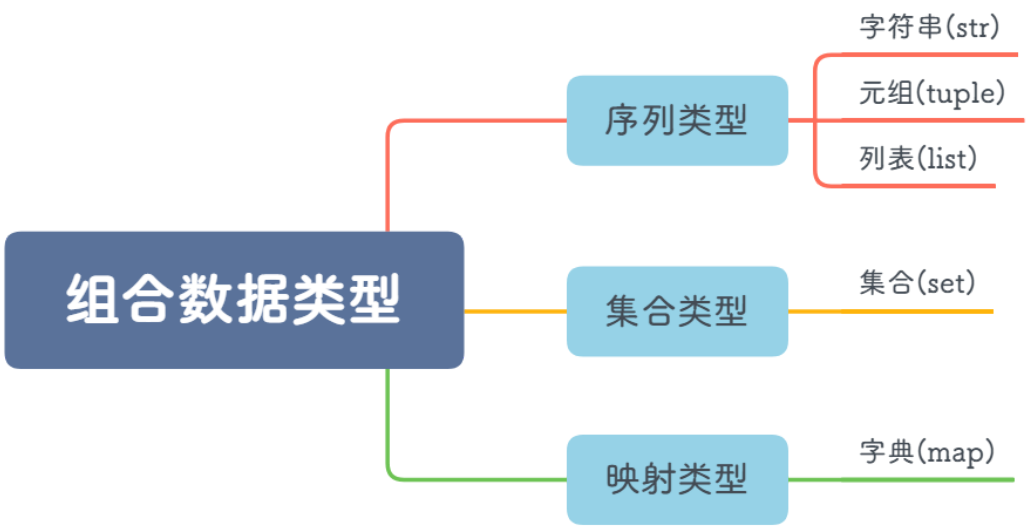
递归特点

1. 递归必须设计基例
2. 递归深度不应超出1000层,若有需要可通过以下代码更改最大层数.

```
import sys
sys.setrecursionlimit(<新的层数限制>)
```

第六章 组合数据类型

序列类型



序列类型特点:

1. 所有**序列类型**都可以使用相同的索引体系
2. 所有序列类型均支持成员关系操作符 `in` ,长度计算函数 `len()` ,切片 `[M:N:step]`
3. 元素自身亦可以为序列类型

序列类型的通用操作符或函数:

操作符	描述
x in s	如果x是s的元素,返回True,否则返回False

操作符	描述
x not in s	如果x不足s的元素，返回True, 否则返回False
s+t	连接s和t
s * n 或 n* s	将序列s复制n次
s[i]	索引,返回序列的第i个元素
s[i: j]	分片,返回包含序列s第i到j个元素的子序列(不包含第j个元素)
s[i: j: k]	步骤切片，返回包含序列s第i到j个元素以k为步数的子序列
len(s)	序列s的元素个数(长度)
min(s)	序列s中的最小元素
max(s)	序列s中的最大元素
s.index(x[, i[, j]])	序列s中从1开始到j位置中第一次出现元素x的位置
s.count(x)	序列s中出现x的总次数

元组

元组采用逗号 `,` 和圆括号(非必须) `()` 来表示:

```
>>> emotion = ('😀','😟','😔')
    print(type(emotion))
'tuple'
```

> **注意:** 元组一旦被创建就不能修改

列表

列表具有可修改的特性故除了序列操作,还有以下操作:

方法	描述
list.append(x)	把一个元素添加到列表的结尾, 相当于 `a[len(a):] = [x]` .
list.extend(L) 或 list += L	通过添加指定列表的所有元素来扩充列表, 相当于 `a[len(a):] = L` .
list.insert(i, x)	在指定位置插入一个元素. 第一个参数是准备插入到其前面的那个元素的索引, 例如 `a.insert(0, x)` 会插入到整个列表之前, 而 `a.insert(len(a)` , x) 相当于 `a.append(x)` .
list.remove(x)	删除列表中值为 x 的第一个元素. 如果没有这样的元素, 就会返回一个错误.
list.pop([i])	从列表的指定位置移除元素, 并将其返回. 如果没有指定索引, a.pop()返回最后一个元素. 元素随即从列表中被移除.

方法	描述
list.clear()	移除列表中的所有项, 等于 `del a[:]` .
list.index(x)	返回列表中第一个值为 x 的元素的索引. 如果没有匹配的元素就会返回一个错误.
list.count(x)	返回 x 在列表中出现的次数.
list.sort()	对列表中的元素进行排序.
list.reverse()	倒排列表中的元素.
list.copy()	返回列表的浅复制, 等于 `a[:]` .

列表的复制

从上表可以知道,我们可以使用 `list.copy()` 来复制列表,不妨考虑如下两种情况:

情况 1

```
list1 = [1, 2, 3]
list2 = list1
print(id(list1), id(list2))
list1.append(4)
print(list1, list2)
```

输出

```
139674627313168 139674627313168
[1, 2, 3, 4] [1, 2, 3, 4]
```

我们发现这种赋值方式将会导致两个列表变量指向同一个内存指针,对其中一个修改,将会导致两个变量均发生变化.

情况 2

```
list1 = [1, 2, 3]
list2 = list1.copy()
print(id(list1), id(list2))
list2.append(4)
print(list1, list2)
```

输出:

```
139674626873936 139674626888560
[1, 2, 3] [1, 2, 3, 4]
```

`list.copy()` 方法将会产生一个新的列表变量,其指向不同的内存指针,两者互相独立不相互改变

关于这两种情况,我们可以稍微引申一下:

```
a = 1
b = a
print(id(a), id(b))
b = a + 1
print(id(a), id(b))
```

输出:

```
93874524533504 93874524533504
93874524533504 93874524533536
```

当我们未对变量 b 进行修改时, b与a指向相同的地址,但当我们对b作出修改,此时a,b不再关联.

列表的递推公式

每个列表递推公式都在 for 之后跟一个表达式, 然后有零到多个 for 或 if 子句. 返回结果是一个根据表达从其后的 for 和 if 上下文环境中生成出来的列表. 如果希望表达式推导出一个元组, 就必须使用括号. 语法如下:

```
#单变量递推
[f(x) for x in <遍历范围> if <服从条件>]
#多变量递推
[f(x,y) for x in <x遍历范围>
        for y in <y遍历范围>
        if <x服从条件>
        if <y服从条件>]
```

例 1
小明和小红丢骰子,问小明的点数比小红大的概率是多少?

解 小明和小红丢骰子丢出的点数相互独立,故他们可取以下值:

```
# 小明
A = [x for x in range(1,7)] # = [1, 2, 3, 4, 5, 6]
# 小红
B = [x for x in range(1,7)] # = [1, 2, 3, 4, 5, 6]
```

设小明点数比小红大的事件为H,则

```
>>> H = [(x,y) for x in A for y in B if x>y]
[(2, 1),
(3, 1),
(3, 2),
(4, 1),
(4, 2),
(4, 3),
(5, 1),
(5, 2),
(5, 3),
(5, 4),
(6, 1),
(6, 2),
(6, 3),
(6, 4),
(6, 5)]
```

可以求出事件H发生的概率为:

```
>>> print('概率 = ',len(H)/len([(x,y) for x in A for y in B]))
概率 =  0.4166666666666667
```

列表的排序 `list.sort()` 方法与 `sorted()` 函数

列表的排序有两种处理方式,分别是 `list.sort()` 方法与 `sorted()` 函数.他们有如下区别

> ** `.sort()` 与 `sorted()` 区别: **
>
> `sort` 是应用在 list 上的方法, `sorted` 可以对所有可迭代的对象进行排序操作. list 的 `sort` 方法返回的是对已经存在的列表进行操作, *无返回值*, 而内建函数 `sorted` 方法返回的是一个*新的 list*, 而不是在原来的基础上进行的操作.

关于该区别,有以下辨识:

```
ls = [1,2,4,3]
print(ls.sort())
```

该程序的输出应为 `None` ,猜错的小伙伴要再去看看上面的区别啦😜. `sort()` 方法是没有返回值的,结果咧,你 `print` 了个寂寞.

`list.sort()` 方法语法

```
list.sort(key=None, reverse=False)
```

- **key** 通过函数来指定可迭代对象中的一个元素来进行排序;
- **reverse** 决定排序是否反向,默认为 `False`

示例:

```
list = [(2, 2), (3, 4), (4, 1), (1, 3)]
list.sort(key=lambda x:x[1]) # sort将会改变原列表(详见上述区别)
print(list)
```

返回:

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

`sorted()` 函数语法

```
sorted(iterable, key=None, reverse=False)
```

使用方法同上.

`del` 语句

使用 `del` 语句可以从一个列表中依索引而不是值来删除一个元素. 这与使用 pop() 返回一个值不同. 可以用 `del` 语句从列表中删除一个切片, 或清空整个列表(del a[:])而不删除该列表变量. 例如:

```
>>> a = [1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
```

集合

集合的定义类似于数学上的定义,有以下特点:

1. 集合是无序组合,没有索引与位置的概念,不支持切片;
2. 集合中的元素是不能重复的(可利用该特征过滤重复元素);
3. 打印出的集合会按照一定规律排列,不一定与定义顺序一致.

集合的表示与生成

使用大括号 `{}` 表示的数组称为集合,例如:

```
{'😒', '😀', (1, 2), 1234, 'hello world'}
```

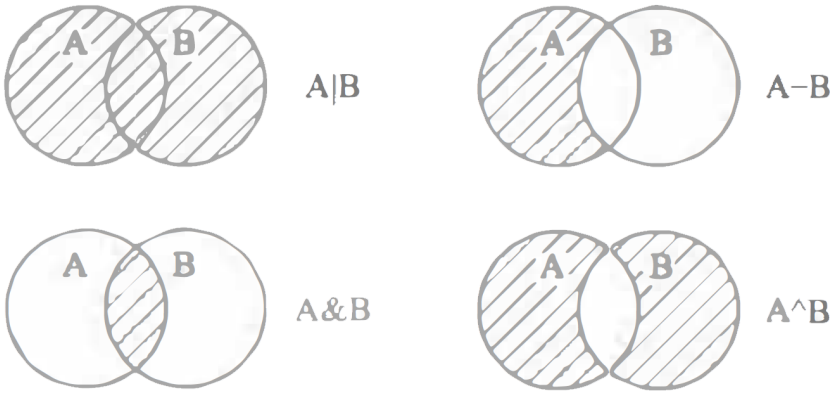
为一个集合.

使用 `set()` 来生成集合,示例如下:

```
>>> w = set('apple')
{'e', 'p', 'a', 'l'} # 注意单独输入字符串将被看作一个序列并拆分字符
```

集合类型的操作符

操作符	描述
S - T 或 S.difference(T)	$S-T$
S -= T 或 S.difference_update(T)	$S=S-T$
S & T 或 S.intersection(T)	$S \cap T$
S&=T 或 S.intersectionupdate(T)	$S=S \cap T$
S^T 或 s.symmetric_ difference(T)	$S \cup T-S \cap T$
S^=T 或 s.symmetric_difference_update(T)	$S=S \cup T-S \cap T$
S|T 或 S.union(T)	$S \cup T$
S|=T 或 S.update(T)	$S=S \cup T$
S<=T 或 S.issubset(T)	当$S \subseteqq T$ 时为真
S<T	当$S \subsetneqq T$ 时为真
S>=T 或 S.issuperset(T)	当$S \supseteqq T$ 时为真
S>T	当$S \supsetneqq T$ 时为真



集合类型的操作函数或方法

操作函数或方法	描述
S.add(x)	如果数据项x不在集合S中, 将x增加到s
S.clear()	移除S中的所有数据项
S.copy()	返回集合S的一个副本

操作函数或方法	描述
S.pop()	随机返回集合S中的一个元素, 如果S为空, 产生KeyError异常
S.discard(x)	如果x在集合S中, 移除该元素
S.remove(x)	如果x在集合S中, 移除该元素
S.isdisjoint(T)	如果集合S与T没有相同元素, 返回True
len(S)	返同集合S的元素个数
X in S	如果x是S的元素, 返回True, 否则返回False
x not in S	如果x不是S的元素, 返回True, 否则返回False

字典

在python中我们可以把字典看作**键值对**的集合(他们是不同的数据结构,字典不属于集合),
特点:

1. 序列是以连续的整数为索引, 与此不同的是, 字典以关键字为索引, 关键字可以是任意不可变类型, 通常用字符串或数值.
2. 字典键值对之间没有顺序也不能重复
3. **字典的键必须是** 元组 , 字符串 , 数字 **;不允许使用列表,字典,集合作为键.**

字典的创建

1. 使用 `{}` 创建空字典

```
dictionary = {}
```

2. 使用函数 `dict()` 直接从键值对元组列表中构建字典

```
>>> dict([('萝卜', '🥕'), ('香蕉', '🍌'), ('苹果', '🍏')])
{'萝卜': '🥕', '香蕉': '🍌', '苹果': '🍏'}
```

3. 使用递推公式生成键和值

```
>>> parrot =  {str(x)+'个萝卜🥕': str(x)+'个坑' for x in ['一', '两', '三']}
>>> print(parrot)

 {'一个萝卜🥕': '一个坑', '两个萝卜🥕': '两个坑', '三个萝卜🥕': '三个坑'}
```

4. 当键只是简单的**字符串**, 使用键参数指定键值

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

字典的访问

通过索引符号实现:

```
<字典变量>[<键>]
```

当找不到该键时,将会返回 `KeyError` .该访问方式也可以修改或添加键值对

```
<字典变量>[<键>] = <键值>
```

字典的操作

函数和方法	描述
<d>.keys()	返回所有的键信息
<d>.values()	返回所有的值信息
<d>.items()	返回所有的键值对
<d>.get(<key>,<default>)	键存在则返回相应值, 否则返回默认值
<d>.pop(<key>, <default>)	键存在则返回相应值, 同时删除键们对, 否则返回默认值
<d>.popitem()	随机从字典中取出一个键值对, 以元组(key, value)形式返回
<d>.clear()	删除所有的键值对
del <d>[<key>]	删除字典中某一个键值对
<key> in <d>	如果键在字典中则返回True, 否则返回False

需要注意的是 `.keys()`，`.values()`，`.items()` 他们的输出类型不是列表,我们可以使用 `list()` 函数来返回列表:

```
dictionary = dict([('萝卜', '🥕'), ('香蕉', '🍌'), ('苹果', '🍏')])
print(dictionary.keys())
print(dictionary.values())
print(dictionary.items())

print(type(dictionary.keys()))
print(type(dictionary.values()))
print(type(dictionary.items()))
```

输出:

```
dict_keys(['萝卜', '香蕉', '苹果'])
dict_values(['🥕', '🍌', '🍏'])
dict_items([('萝卜', '🥕'), ('香蕉', '🍌'), ('苹果', '🍏')])

<class 'dict_keys'>
<class 'dict_values'>
<class 'dict_items'>
```

注意观察他们的输出类型.

例子 : 词频统计
可以利用字典键不重复的特性进行词频统计,建立(单词:词频)的映射,思路如下:

```
for word in <string>:
    counts[word] = count.get(word, 0) + 1
```

遍历

在字典中遍历时, 键值对可以使用 `items()` 方法同时解读出来:

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

在序列中遍历时, 索引位置和对应值可以使用 `enumerate()` 函数同时得到:

```
>>> for i, v in enumerate(['🚀', '🦾', '🤚']):
...     print(i, v)
...
0 🚀
1 🦾
2 🤚
```

同时遍历两个或更多的序列, 可以使用 `zip()` 组合:

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

要反向遍历一个序列, 首先指定这个序列, 然后调用 `reversed()` 函数:

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

第七章 文件和数据格式化

文件的打开关闭

python使用 `open()` 方法实现文件的打开,打开文件之前需保证文件不被占用(否则返回 `OSError`),若文件被python占用,需使用 `close()` 解除占用.语法格式:

```
<变量名> = open(<filename>, <mode>)
```

`<mode>` 处用于指定文件的打开方式,有如下打开方式:

文件的打开模式	含义
'r'	只读校式, 如果文件不存在, 返回异常FileNotFoundError,默认值
'w'	覆盖写模式, 指针会从第一行行首开始覆写,文件不存在则创建, 存在则完全疫盖
'x'	创建马模式, 文件不存在则创建, 存在则返同异常FileExistsError

文件的打开模式	含义
'a'	追加写模式, 文件不存在则创建, 存在则在文件最后追加内容
'b'	二进制文件模式
't'	文本文件模式, 默认值
'+'	与r/w/x/a 一同使用, 在原功能基础上增加同时读写功能

以上打开方式可以组合使用,打开后记得使用 `close()` 方法释放.
示例:

```
file = open('とある科学の超電磁砲.txt', 'r')
<程序体>
file.close()
```

文件的读写

> 注意文件读写过程中,每一行都可能需要换行符.读取时记得删除换行符,写入时记得按照需求添加换行符.换行符是真实存在的字符,只不过看不见而已啦.

读取

操作方法	含义
<file>.readall()	读入整个文件内容, 返回 - 个字符串或字节流
<file>.read(size=-1)	从文件中读入整个文件内容, 如果给出参数, 读入前size长度的字符串或字节流
<file>.readline(size=-1)	从文件中读入一行内容, 如果给出参数, 读入该行前size长度的字符串或字节流
<file>.readlines(hint=-1)	从文件中读入所有行, 以每行为元素形成一个列表, 如果给出参数, 读入hint行

我们常用以下方式(python此时将文件看作一个行序列,通过遍历进行读取)来逐行读入并修改,这样可以避免一次性读入过多内容影响内存性能:

```
file = input('とある科学の超電磁砲.txt', 'r')
for line in file:
    print(line)
file.close()
```

这样就可以把一本超炮打印出来啦...

写入

写入方法	含义
<file>.write(s)	向文件写入一个字符串或字节流
<file>. writelines(Lines)	将一个元素全为字符串的列表写入文件

写入方法	含义
<file>.seek(offset)	改变当前文件操作指针的位置, offset的值: 0-文件开头; l一当前位置: 2一文件结尾