Python 学习笔记

变量的定义和使用

```
name = '玛丽亚'
print(name)
print('标识', id(name))
print('类型', type(name))
print('值', name)
```

- 当多次赋值后，变量名会指向新的空间

```
name = '玛丽亚'
print(name)
name = '楚留冰'
print(name)
```

数据类型

- 常用的数据类型
    - 整数类型 int
    - 浮点数类型 float
    - 布尔类型 bool
    - 字符串类型 str

整数类型

- 英文为 integer，简写为 int，可以表示正数、负数和零
- 整数的不同进制表示方式
    - 十进制 -> 默认的进制
    - 二进制 -> 以 `0b` 开头
    - 八进制 -> 以 `0o` 开头

- 十六进制 -> 以`0x`开头

浮点类型

- 浮点数由整数部分和小数部分组成
- 浮点数存储具有不精确性
  - 使用浮点数进行计算时，可能会出现小数位数不确定的情况

```
print(1.1 + 2.2) # 3.3000000000000003
print(1.1 + 2.1) # 3.2
```

  - 解决方案
    - 导入`decimal`模块

```
from decimal import Decimal
print(Decimal('1.1') + Decimal('2.2')) # 3.3
```

布尔类型

- 用来表示真或假的值
- True表示真，False表示假
- 布尔值可以转化为整数
  - True -> 1
  - False -> 0

```
print(True + 1) # 2
print(False + 1) # 1
```

字符串类型

- 字符串又被称为不可变的字符序列
- 可以使用单引号`''`、双引号`""`、三引号`''' '''`或`""" """`来定义
- 单引号和双引号定义的字符串必须在一行
- 三引号定义的字符串可以分布在连续的多行

```
str = '''人生苦短,
    我用Python'''
```

数据类型转换

函数名	作用	注意事项	举例
str()	将其他数据类型转成字符串	也可以用引号转换	str(123) '123'
int()	将其他数据类型转成整数	1. 文字类和小数类字符串，无法转化成整数 2. 浮点数转化成整数，抹零取整	int('123') int(9.8)
float()	将其他数据类型转成浮点数	1. 文字类无法转成浮点数 2. 整数转化成浮点数，末尾为.0	float('9.9') float(9)

注释

- 在代码中对代码的功能进行解释说明的标注行文字，可以提高代码的阅读性
- 注释的内容会被Python解释器忽略
- 通常包括三种类型的注释
    - 单行注释 -> 以`#`开头，直到换行结束
    - 多行注释 -> 并没有单独的多行注释标记，将一对三引号之间的代码称为多行注释
    - 中文编码声明注释 -> 在文件开头加上中文声明注释，用以指定源码文件的编码形式

```
# coding gbk
```

input 函数

- 作用：接收来自用户的输入
- 返回值类型：输入值的类型为str
- 值的存储：使用=对输入的值进行存储

```
present = input('大圣想要什么礼物呢？')
print(present)
```

运算符

算数运算符

- 标准算数运算符：加(`+`)、减(`-`)、乘(`*`)、除(`/`)、整除(`//`)
- 取余运算符：`%`
- 幂运算符：`**`
- 整除时，向下取整

```
print(9 // 4) # 2
print(-9 // -4) # 2
print(-9 // 4) # -3
print(9 // -4) # -3
```

- 取余运算时，余数 = 被除数 - 除数 * 商

```
print(9 % -4) # 9 - (-4) * (-3) = -3
print(-9 % 4) # (-9) - 4 * (-3) = 3
```

赋值运算符

- =
    - 执行顺序：右 -> 左
    - 支持链式赋值：`a=b=c=20`
    - 支持参数赋值：`+=`、`-=`、`*=`、`/=`、`//=`、`%=`
    - 支持系列解包赋值：`a,b,c = 20,30,40`

比较运算符

- 对变量或表达式的结果进行大小、真假等比较
- `>`、`<`、`>=`、`<=`、`!=`
- 对象 value 的比较：`==`
- 对象 id 的比较：`is`、`is not`

布尔运算符

- `and`：当两个运算数都为True时，运算结果才为true
- `or`：只要有一个运算数为True，运算结果就为True
- `not`：如果运算数为True，运算结果为False；如果运算数为False，运算结果为True

- `in`
- `not in`

```
s = 'helloworld'
print('w' in s) # True
print('a' not in s) # True
```

位运算

- 位与`&`：对应数位都是1，结果数位才是1，否则为0
- 位或`|`：对应数位都是0，结果数位才是0，否则为1
- 左移位运算符`<<`：高位溢出舍弃，地位补0
- 右移位运算符`>>`：低位溢出舍弃，高位补0

```
print(4 & 8) # 0
print(4 | 8) # 12
print(4 << 1) # 8
print(4 << 2) # 16
print(4 >> 1) # 2
print(4 >> 2) # 1
```

> 向左移动1位，相当于乘以2
>
> 向右移动1位，相当于除以2

运算符的优先级

- `**` ---> `*`、`/`、`//`、`%` ---> `<<`、`>>` ---> `&` ---> `|` ---> `>`、`<`、`>=`、`<=`、`==`、`!=` ---> `and` ---> `or` ---> `=`

顺序结构

- 程序从上到下顺序地执行代码，中间没有任何的判断和跳转，直到程序结束

对象的布尔值

- Python 一切皆对象，所有对象都有一个布尔值
    - 获取对象的布尔值，使用内置函数`bool()`
- 以下对象的布尔值为 False

- False
- 数值0
- None
- 空字符串
- 空列表

```
print(bool([])
print(bool(list()))
```

- 空元组

```
 print(bool(()))
 print(bool(tuple()))
```

- 空字典

```
 print(bool({}))
 print(bool(dict()))
```

- 空集合

```
 print(set())
```

分支结构

单分支结构

- 中文语义：如果...就...
- 语法结构：

```
if 条件表达式:
    条件执行体
```

双分支结构

- 中文语义：如果...不满足...就...
- 语法结构：

```
if 条件表达式:
    条件执行体1
else:
    条件执行体2
```

多分支结构

- 语法结构：

```
if 条件表达式1:
    条件执行体1
elif 条件表达式2:
    条件执行体2
elif 条件表达式N:
    条件执行体N
[else:]
    条件执行体N+1
```

嵌套 if 的使用

- 语法结构：

```
if 条件表达式:
    if 内层条件表达式2:
        内层条件执行体1
    else:
        内层条件执行体2
else:
    条件执行体
```

条件表达式

- 条件表达式是 if...else... 的简写
- 语法结构：`x if 判断条件 else y`
- 运算规则：如果判断条件的布尔值为**True**，条件表达式的返回值为**x**，否则条件表达式的返回值为**y**

```
a = int(input('请输入第一个整数: '))
b = int(input('请输入第二个整数: '))
print( str(a)+'大于等于'+str(b) if a>b else str(a)+'小于'+str(b) )
```

pass 语句

- 语句什么都不做，只是一个占位符，用在语法上需要语句的地方
- 生么时候使用?
    - 先搭建语法结构，还没想好代码怎么写的时候
- 与哪些语句一起使用?

    - **if语句的条件执行体**
    - **for...in语句的循环体**
    - **定义函数时的函数体**

range 函数

- 用于生成一个整数序列
- 创建 range() 函数的三种方式
    - `range(stop)`：**创建一个[0,stop)之间的整数序列，步长为1**
    - `range(start, stop)`：**创建一个[start,stop)之间的整数序列，步长为1**
    - `range(start, stop, step)`：**创建一个[start,stop)之间的整数序列，步长为step**
- **返回值是一个迭代器对象**
- range类型的优点：不管range对象表示的整数序列有多长，所有range对象占用的内存空间都是相同的，因为仅仅需要存储start，stop和step，只有当用到range对象时，才会计算序列中的相关元素
- `in` 与 `not in` 用于判断整数序列中是否存在（不存在）指定的整数

while 循环

- 语法结构：

```
ehile 条件表达式:
    条件执行体（循环体）
```

- 选择结构的if和循环结构while的区别
    - if时判断一次，条件为True执行一次
    - while是判断N+1次，条件为True执行N次
- 四步循环法
    - 初始化变量
    - 条件判断

- 条件执行体（循环体）
- 改变变量

for...in 循环

- in表达式从字符串、序列等中依次取值，又称为遍历
- for-in 遍历的对象必须是可迭代对象
- for-in 的语法结构：

```
for 自定义的变量 in 可迭代对象:
    循环体
```

- 循环体内不需要访问自定义变量，可以将自定义变量替换为下划线（_）

```
# 输出5次 人生苦短，我用Python
for _ in range(5)
    print('人生苦短，我用Python')
```

流程控制语句 break

- 用于结束循环结构，通常与分支结构if一起使用

流程控制语句 continue

- 用于结束当前循环，进入下一次循环，通常与分支结构中的if一起使用

else 语句

- 与else语句配合使用的三种情况
    - if ... else ... ： if表达式不成立时执行else
    - while ... else ... ： 没有碰到break时执行else
    - for ... else ... ： 没有碰到break时执行else

嵌套循环

- 循环结构中又嵌套了另外的完整的循环结构，其中内层循环作为外层循环的执行体执行

```
# 输出一个三行四列的矩形
for i in range(1, 4)
    for j in range(1, 5)
        print('*', end='\t') # 不换行输出
    print() # 换行
```

```
# 输出九九乘法表
for i in range(1, 9)
    for j in range(i, i+1)
        print(i, '*', j, '=', i*j, end='\t') # 不换行输出
    print() # 换行
```

二重循环中的 break 与 continue

- 二重循环中的 break 与 continue 用于控制本层循环

列表

为什么需要列表

- 变量可以存储一个元素，而列表是一个“大容器”，可以存储N多个元素，程序可以方便地对这些数据进行整体操作
- 列表相当于其它语言中的数组

列表对象的创建

- 列表需要使用方括号[]括起来，元素之间使用英文的逗号进行分隔
- 列表的创建方式
    - 使用中括号

```
lst = ['大圣', '娟子姐']
```

    - 调用内置函数 `list()`

```
lst2 = list(['大圣', '娟子姐'])
```

列表的特点

- 列表元素按顺序有序排序
- 索引映射唯一一个数据
- 列表可以存储重复的元素
- 任意数据类型可以混存
- 根据需要动态分配和回收内存

获取指定元素的索引

- 获取类表中指定元素的索引——`index()`
    - 如果列表中存在N个相同的元素，只返回相同元素中第一个元素的索引
    - 如果查询的元素在列表中不存在，则会输出 `ValueError`
    - 还可以在指定的start和stop之间进行查找

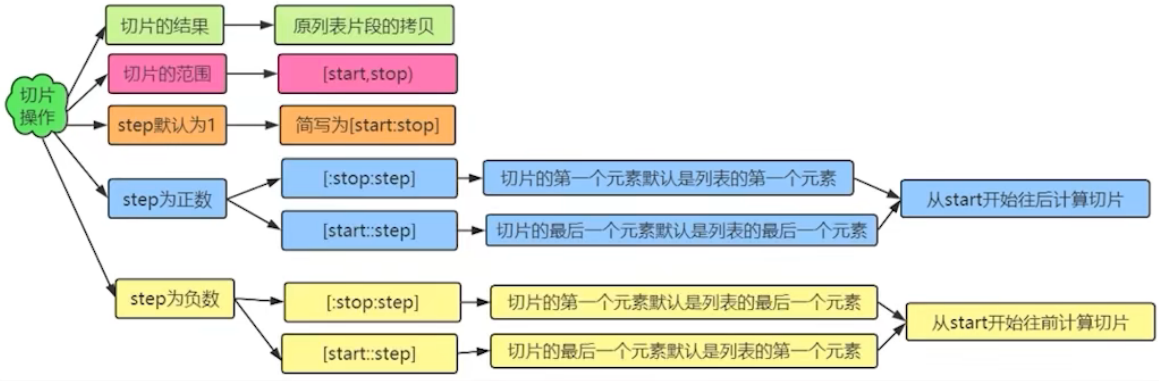
获取列表中指定的元素

- 获取列表中的单个元素
    - 正向索引从0到N-1，举例：`lst[0]`
    - 逆向索引从-N到-1，举例：`lst[-N]`
    - 指定索引不存在，抛出`IndexError`

获取列表中的多个元素——切片操作

- 语法格式：

```
列表名[start : stop : step]
```



列表元素的判断及遍历

- 判断指定元素在列表中是否存在

```
元素 in 列表名
```

```
元素 not in 列表名
```

- 列表元素的遍历

```
for 迭代变量 in 列表名:
    操作
```

列表元素的添加操作

方法	描述
append()	在列表的末尾添加一个元素
extend()	在列表的末尾至少添加一个元素
insert()	在列表的任意位置添加一个元素
切片	在列表的任意位置至少添加一个元素

列表元素的删除操作

方法	描述
remove()	一次删除一个元素 重复元素只删除第一个 元素不存在抛出ValueError
pop()	删除一个指定索引位置上的元素 指定索引不存在抛出IndexError 不指定索引，删除列表中最后一个元素
切片	一次至少删除一个元素 切片后会产生一个新的数组
clear()	清空列表
del	删除列表

列表元素的修改操作

- 为指定索引的元素赋予一个新值
- 为指定的切片赋予一个新值

```
lst = [10, 20, 30, 40];
lst[1] = 100;
lst[1:3] = [100, 200, 300]
```

列表元素的排序操作

- 调用 `sort()` 方法，列表中的所有元素默认按照从小到大的顺序进行排序，可以指定 `reverse = True` ，进行降序排序

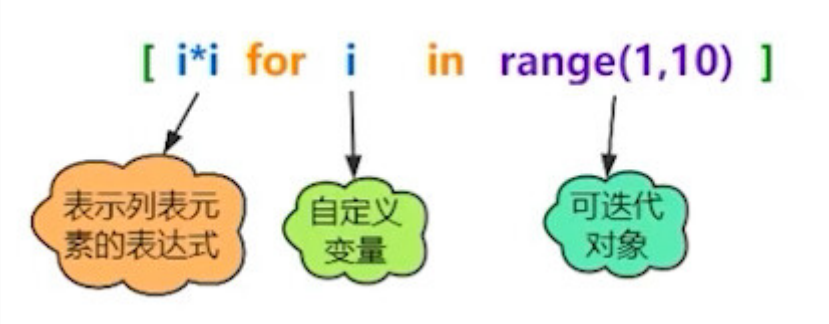
```
lst.sort()
lst.sort(reversr = True)
```

- 调用内置函数 `sorted()` ，可以指定 `reverse = True` ，进行降序排序，原列表不发生改变

```
lst2 = sorted(lst)
lst2 = sorted(lst, reverse = True)
```

列表生成式

- 语法格式：`[i*i for i in range(1, 10)]`

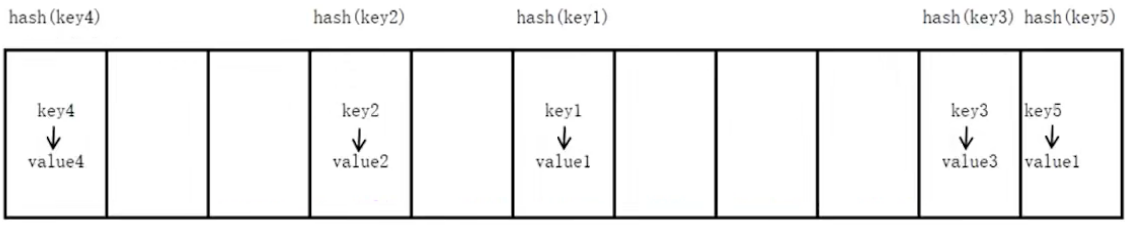


- ⚠️注意事项：“表示列表元素的表达式”中通常要包含自定义变量

字典

什么是字典

- Python 内置的数据结构之一，与列表一样是一个可变序列
- 以键值对的方式存储数据，字典是一个无序的序列
- 字典示意图



字典的实现原理

- 字典的实现原理与查字典类似，查字典是先根据部首或拼音查找对应的页码，Python 中的字典是**根据key查找value所在的位置**

字典的创建

- 最常见的方式：使用花括号 `{}`

```
scores = {"张三" : 100, "李四" : 98, "王五" : 45}
```

- 使用内置函数 `dict()`

```
dict(name='jack', age=20)
```

字典元素的获取

- 字典中元素的获取
  - 通过方括号 `[]` 获取，举例：`scores['张三']`
  - 通过 `get()` 方法获取，举例：`scores.get('张三')`
- [] 取值与使用 get() 取值的区别
  - [] 取值时，如果字典中不存在指定的key，抛出KeyError异常
  - get() 方法取值，如果字典中不存在指定的key，并不会抛出 KeyError 异常，而是返回None，可以通过参数设置默认的value，以便指定的key不存在时返回

```
scores.get('麻七', 99) # 99是在查找麻七所对应的value不存在时所提供的默认值
```

字典元素的增、删、改操作

- key 的判断
  - in ： 指定的key在字典中存在返回True
  - not in ： 指定的key在字典中不存在返回True
- 字典元素的删除

```
del scores['张三']
scores.clear() # 清空字典元素
```

- 字典元素的新增

```
scores['jack'] = 90
```

获取字典视图

方法	描述
keys()	获取字典中所有key
values()	获取字典中所有value
items()	获取字典中所有key,value对

字典元素的遍历

- 语法结构

```
for item in scores:
    print(item)
```

字典的特点

- 字典中所有元素都是一个 key-value 对，key 不允许重复，value 可以重复
- 字典中的元素是无序的
- 字典中的 key 必须是不可变对象
- 字典也可以根据需要动态地伸缩
- 字典会浪费较大的内存，是一种使用空间换时间的数据结构

字典生成式

- 内置函数 `zip()`
    - 用于将可迭代的对象作为参数，将对象中对应的元素打包成一个元组，然后返回由这些元组组成的列表

```
items = ['Fruits', 'Books', 'Others']
prices = [96, 78, 85]
lst = zip(items, prices)
print(list(lst))
```

- 字典生成式

```
{item.upper() : price for item,price in zip(items, prices)}
```

元组

什么是元组

- Python 内置的数据结构之一，是一个不可变序列

```
t = ('Python', 'hello', 90)
```

- 不可变序列与可变序列
    - 不可变序列：字符串、元组
        - 没有增、删、改操作
    - 可变序列：列表、字典
        - 可以对序列执行增、删、改操作，对象地址不发生改变

元组的创建方式

- 直接小括号 `()`

```
t = ('Python', 'hello', 90)
```

- 使用内置函数 `tuple()`

```
t = tuple(('Python', 'hello', 90))
```

- 只包含一个元素的元组需要使用逗号和小括号

```
t = (10, )
```

- 空元组

```
t1 = ()
t2 = tuple()
```

为什么要将元组设计成不可变序列

- 在多任务环境下，同时操作对象时不需要加锁
- 因此，在程序中尽量使用不可变序列
- ⚠️ **注意事项：**元组中存储的是对象的引用

    - 如果元组中对象本身是不可变对象，则不能再引用其他对象
    - 如果元组中的对象是可变对象，则可变对象的引用不允许改变，但数据可以改变

元组的遍历

- 元组是可迭代对象，可以使用 for...in 进行遍历

```
t = tuple(('Python', 'hello', 90))
for item in t:
    print(item)
```

集合

集合的概述与创建

- Python 语言提供的内置数据结构
- 与列表、字典一样都属于可变类型的序列
- 集合是没有 value 的字典
- 集合的创建方式
    - 直接创建 `{}`

```
s = {'Python', 'hello', 90}
```

    - 使用内置函数 `set()`

```
s = set(range(6))
print(s)
print(set([3, 4, 53, 56]))
print(set((3, 4, 43, 435)))
print(set('Python'))
print(set({124, 3, 4, 4, 5}))
print(set())
```

- 集合中的元素不允许重复，集合会自动去除重复的元素
- 集合中的元素是无序的

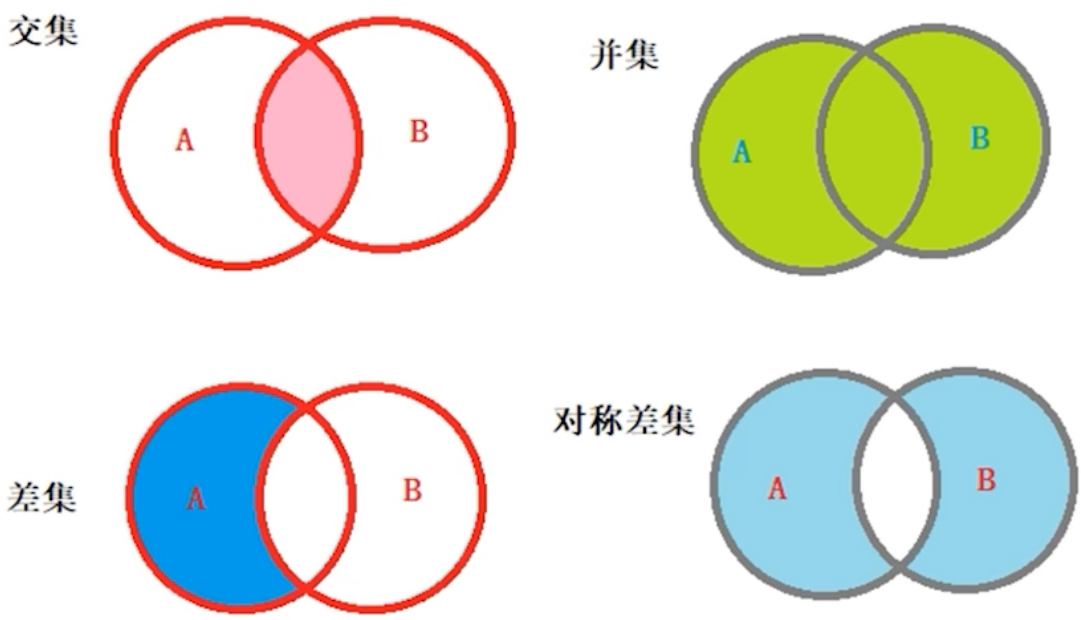
集合的相关操作

- 集合元素的判断操作
    - `in` 或 `not in`
- 集合元素的新增操作
    - 调用 `add()` 方法，一次添加一个元素
    - 调用 `update()` 方法，一次至少添加一个元素
- 集合元素的删除操作
    - 调用 `remove()` 方法，一次删除一个指定元素，如果指定的元素不存在，抛出 KeyError 异常
    - 调用 `discard()` 方法，一次删除一个指定元素，如果指定的元素不存在不抛异常
    - 调用 `pop()` 方法，一次删除一个任意元素，该方法没有参数
    - 调用 `clear()` 方法，清空集合

集合间的关系

- 两个集合是否相等
    - 可以使用运算符 `==` 或 `!=` 进行判断
- 一个集合是否是另一个集合的子集
    - 可以调用方法 `issubset()` 进行判断
- 一个集合是否是另一个集合的超集
    - 可以调用方法 `issuperset()` 进行判断
- 两个集合是否没有交集
    - 可以调用方法 `isdisjoint()` 进行判断
    - 没有交集为True
    - 有交集为False

集合的数据操作



- 求两个集合的交集
  - `intersection()`
  - `&`
- 求两个集合的并集
  - `union()`
  - `|`
- 求两个集合的差集
  - `differencr()`
  - `-`
- 求两个集合的对称差集
  - `symmetric_difference()`
  - `^`

集合生成式

```
{i*i for i in range(1, 10)}
```

- 将`{}`修改为`[]`就是列表生成式
- 没有元组生成式

列表、字典、元组、集合总结

数据结构	是否可变	是否重复	是否有序	定义符号
列表（list）	可变	可重复	有序	[]
元组（tuple）	不可变	可重复	有序	()
字典（dict）	可变	key不可重复 value可重复	无序	{key : value}
集合（set）	可变	不可重复	无序	{}

字符串

字符串的创建与驻留机制

- Python 中字符串是基本数据类型，是一个不可变的字符序列
- 字符串驻留机制：仅保存一份相同且不可变字符串的方法，不同的值被存放在字符串的驻留池中，Python 的驻留机制对相同的字符串只保留一份拷贝，后续创建相同字符串时，不会开辟新空间，而是把该字符串的地址赋给新创建的变量。
- 驻留机制的几种情况（交互模式）
    - 字符串的长度为0或1时
    - 符合标识符的字符串（含有数字、字母、下划线）
    - 字符串只在编译时进行驻留，而非运行时
    - [-5, 256] 之间的整数数字
- sys 中的 intern 方法强制2个字符串指向同一个对象
- PyCharm 对字符串进行了优化处理

字符串的常用操作

字符串的查询操作

方法名称	作用
index()	查找子串substr第一次出现的位置，如果查找的字串不存在，则抛出ValueError
rindex()	查找字串substr最后一次出现的位置，如果查找的字串不存在，则抛出ValueError
find()	查找子串substr第一次出现的位置，如果查找的字串不存在，则返回-1
dfind()	查找字串substr最后一次出现的位置，如果查找的字串不存在，则返回-1

字符串的大小写

方法名称	作用
upper()	把字符串所有字符都转换成大写字母
lower()	把字符串所有字符都转换成小写字母
swapcase()	把字符串中所有大写字母转成小写字母，把所有小写字母转成大写字母
capitalize()	把第一个字符转换成大写，其余字符转为小写
title()	把每个单词的第一个字符转换为大写，把每个单词的剩余字符转换为小写

字符串的内容对齐

方法名称	作用
center()	居中对齐，第一个参数指定宽度，第二个参数指定填充符，第二个参数是可选的，默认是空格，如果设置宽度小于实际宽度则返回原字符串
ljust()	左对齐，第一个参数指定宽度，第二个参数指定填充符，第二个参数是可选的，默认是空格，如果设置宽度小于实际宽度则返回原字符串
rjust()	右对齐，第一个参数指定宽度，第二个参数指定填充符，第二个参数是可选的，默认是空格，如果设置宽度小于实际宽度则返回原字符串
zfill()	右对齐，左边用0填充，给方法只接收一个参数，用于指定字符串的宽度，如果指定的宽度小于字符串的长度，返回字符串本身

字符串的劈分

方法名称	作用
split()	1.从字符串的左边开始劈分，默认的劈分字符是空格字符串，返回一个列表 2.以参数sep为劈分符进行劈分 3.通过参数maxsplit指定劈分字符串时的最大劈分次数，经过最后一次劈分后，剩余的字串会单独作为一部分
rsplit()	1.从字符串的右边开始劈分，默认的劈分字符是空格字符串，返回一个列表 2.以参数sep为劈分符进行劈分 3.通过参数maxsplit指定劈分字符串时的最大劈分次数，经过最后一次劈分后，剩余的字串会单独作为一部分

字符串判断的相关方法

方法名称	作用
isidentifier()	判断指定的字符串是不是合法的标识符
isspace()	判断指定的字符串是否全部由空白字符组成（回车、换行、水平制表符）
isalpha()	判断指定的字符串是否全部由字母组成
isdecimal()	判断指定的字符串是否全部由十进制数字组成
isnumeric()	判断指定的字符串是否全部由数字组成
isalnum()	判断指定的字符串是否全部由字母和数字组成

替换与合并

方法名称	作用
join()	将列表或元组中的字符串合并成一个字符串
replace()	第一个参数指定被替换的字串，第二个参数指定替换字串的字符串，该方法返回替换后得到的字符串，替换前的字符串不发生变化没调用该方法时可以通过第三个参数指定最大替换次数

字符串的比较操作

- **运算符**：`>`、`>=`、`<`、`<=`、`==`、`!=`
- **比较规则**：首先比较两个字符串中的第一个字符，如果相等则继续比较下一个字符，依次比较下去，直到两个字符串中的字符不相等时，其比较结果就是两个字符串的比较结果，两个字符串中所有后续字符将不再被比较
- **比较原理**：两个字符串比较时，比较的是其ordinal value(原始值)，调用内置函数`ord()`可以得到指定字符的ordinal value。与内置函数`ord()`对应的是内置函数`chr()`，调用内置函数`chr()`时指定ordinal value可以得到其对应字符

> == 比较的是value
>
> != 比较的是id

字符串的切片操作

- 字符串是不可变类型
- 不具备增、删、改等操作
- 切片操作将产生新的对象

```
s = "hello,Python"
print(s[1 : 5 : 1]) # ello
print(::2) # hloPto
print(s[::-1]) # nohtyP,olleh
```

格式化字符串

- 格式化字符串的两种方式
    - `%`做占位符
        - `%s` 字符串
        - `%i` 或 `&d` 整数
        - `%f` 浮点数

```
name = '张三'
age = 20
print('我叫%s,今年%d岁' % (name, age))
```

    - `{}`做占位符

```
name = '张三'
age = 20
print('我叫{0},今年{1}岁了，我真的叫{0}'.format(name, age))
print(f'我叫{name},今年{age}岁')
```

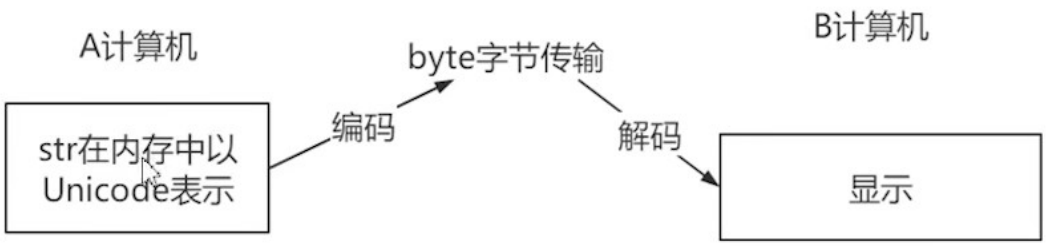
```
print('%10d' % 99) # 10表示宽度
print('%.3f' % 3.1415926) # .3表示保留三位小数

# 同时表示宽度和精度
print('%10.3f' % 3.1415926) # 表示总宽度为10.小数点后保留三位
```

```python
print('{0:.3}'.format(3.1415926)) # .3表示三位有效数字，0为占位符顺序，可以不写
print('{0:.3f}'.format(3.1415926)) # .3f表示三位小数
print('{:10.3f}'.format(3.1415926)) # 表示总宽度10，保留三位小数
```

字符串的编码与解码

- 为什么需要字符串的编码转换



- 编码与解码的方式
    - 编码：将字符串转换为二进制数据（bytes）
    - 解码：将bytes类型的数据转换成字符串类型

```python
s = '天涯共此时'

# 编码
print(s.encode(encoding='GBK')) # 在GBK这种编码格式中，一个中文占两个字节
print(s.encode(encoding='UTF-8')) # 在UTF-8这种编码格式中，一个中文占三个字节

# 解码
byte = s.encode(encoding='GBK')
print(byte.decode(encoding='GBK'))
```

> 编码格式和解码格式要一致

函数

函数的定义与调用

- 什么是函数
    - 函数就是执行特定任务以完成特定功能的一段代码
- 为什么需要函数
    - 复用代码
    - 隐藏实现细节

    - 提高可维护性
    - 提高可读性便于调试

- 函数的创建

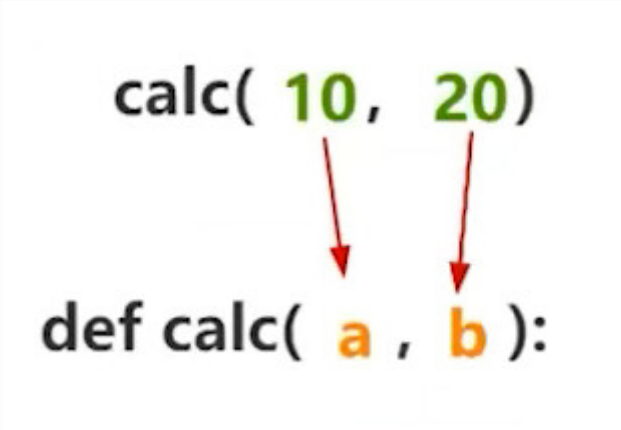
```
def 函数名 ([输入参数]) :
    函数体
    [return xxx]
```

- 函数的调用

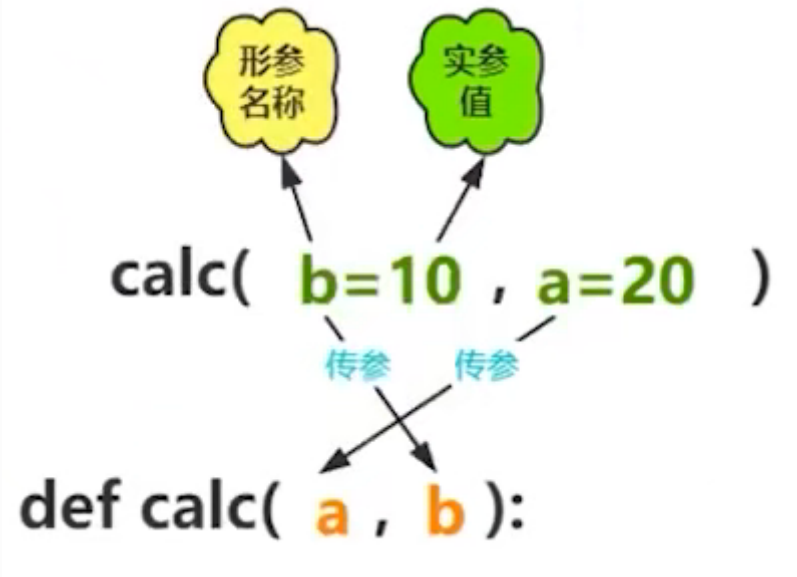
```
函数名([实际参数])
```

函数调用的参数传递

- 位置实参
    - 根据形参对应的位置进行实参传递



- 关键字实参
    - 根据形参名称进行实参传递



1. 实参名称与形参名称可以不一致
2. 在函数调用过程中，进行参数传递
    1. 如果是不可变对象，在函数体的修改不会影响实参的值
    2. 如果是可变对象，在函数体的修改会影响实参的值

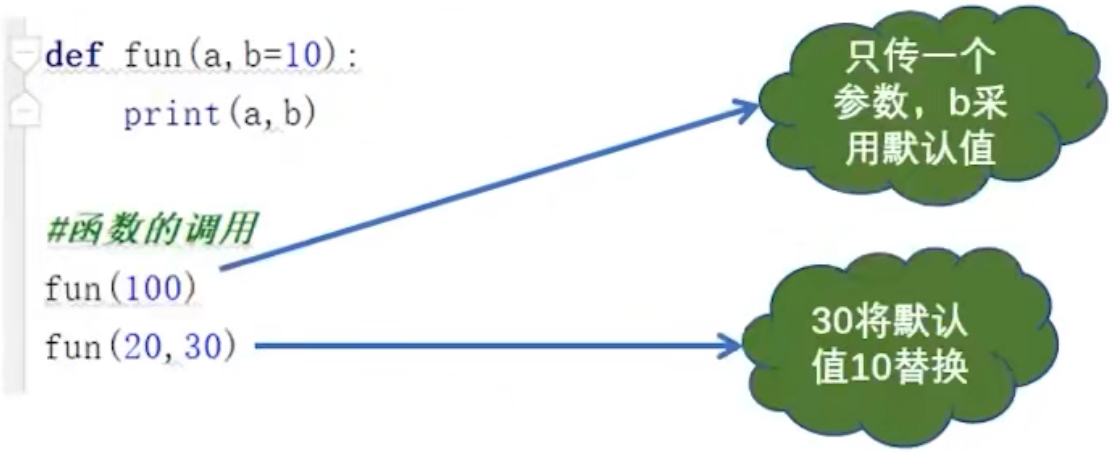
函数的返回值

- 如果函数没有返回值，return可以省略不写
- 函数的返回值只有1个时，直接返回
- 函数返回多个值时，结果为元组

函数参数定义

默认值参数

- 函数定义时，给形参设置默认值，只有与默认值不符的时候才需要传递实参



个数可变的位置形参

- 定义函数时，可能无法实现确定传递的位置实参的个数时，可以使用可变的位置形参
- 使用`*`定义个数可变的位置形参
- 结果为一个元组

```
def fun(*args):
    print(args)
fun(10)
fun(10,20,30)
```

```
Files\Python38\python.exe
 E:/dream/chapfile/d.py
(10,)
(10, 20, 30)
```

> 函数定义过程中，个数可变的位置形参只能是1个

个数可变的关键字形参

- 定义函数时，可能无法实现确定传递的位置实参的个数时，可以使用可变的关键字形参
- 使用 `**` 定义个数可变的关键字形参
- 结果为一个字典

```
def fun(**args):
    print(args)
fun(a=10)
fun(a=10,b=20,c=30)
```

```
 E:/dream/chapfile/d.py
{'a': 10}
{'a': 10, 'b': 20, 'c': 30}
```

> 1. 函数定义过程中，个数可变的关键字形参只能是1个
> 2. 在函数定义过程中，既有个数可变的关键字形参，也有个数可变的位置形参，要求个数可变的位置形参放在个数可变的关键字形参之前

函数的参数总结

序号	参数的类型	函数的定义	函数的调用	备注
1	位置实参		√	
	将序列中的每个元素都转换为位置实参		√	使用*
2	关键字实参		√	
	将字典中的每个键值对都转换为关键字实参		√	使用**
3	默认值形参	√		
4	关键字形参	√		使用*
5	个数可变的位置形参	√		使用*
6	个数可变的关键字形参	√		使用 **

变量的作用域

- 程序代码能访问该变量的区域
- 根据变量的有效范围可分为

    - 局部变量
        - 在函数内定义并使用的变量，只在函数内部有效
        - 局部变量使用`global`声明时，就会变成全局变量
    - 全局变量
        - 函数体外定义的变量，可作用于函数内外

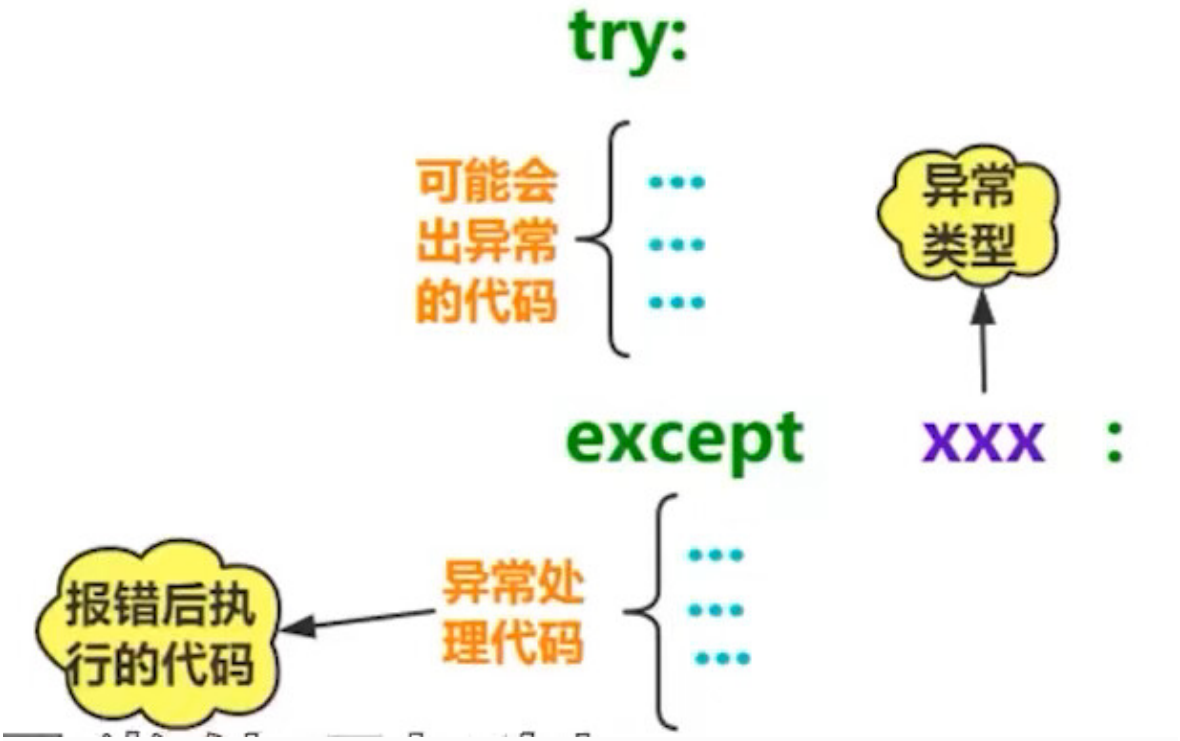
递归函数

- 什么时递归函数
    - 如果一个函数的函数体内调用了函数本身，这个函数就称为递归函数
- 递归的组成部分
    - 递归调用与递归终止条件
- 递归的调用过程
    - 每递归调用一次函数，都会在栈内存分配一个栈帧
    - 每执行完一次函数，都会释放相应的空间
- 递归的优缺点
    - 缺点：占用内存多，效率低下
    - 优点：思路和代码简单

异常

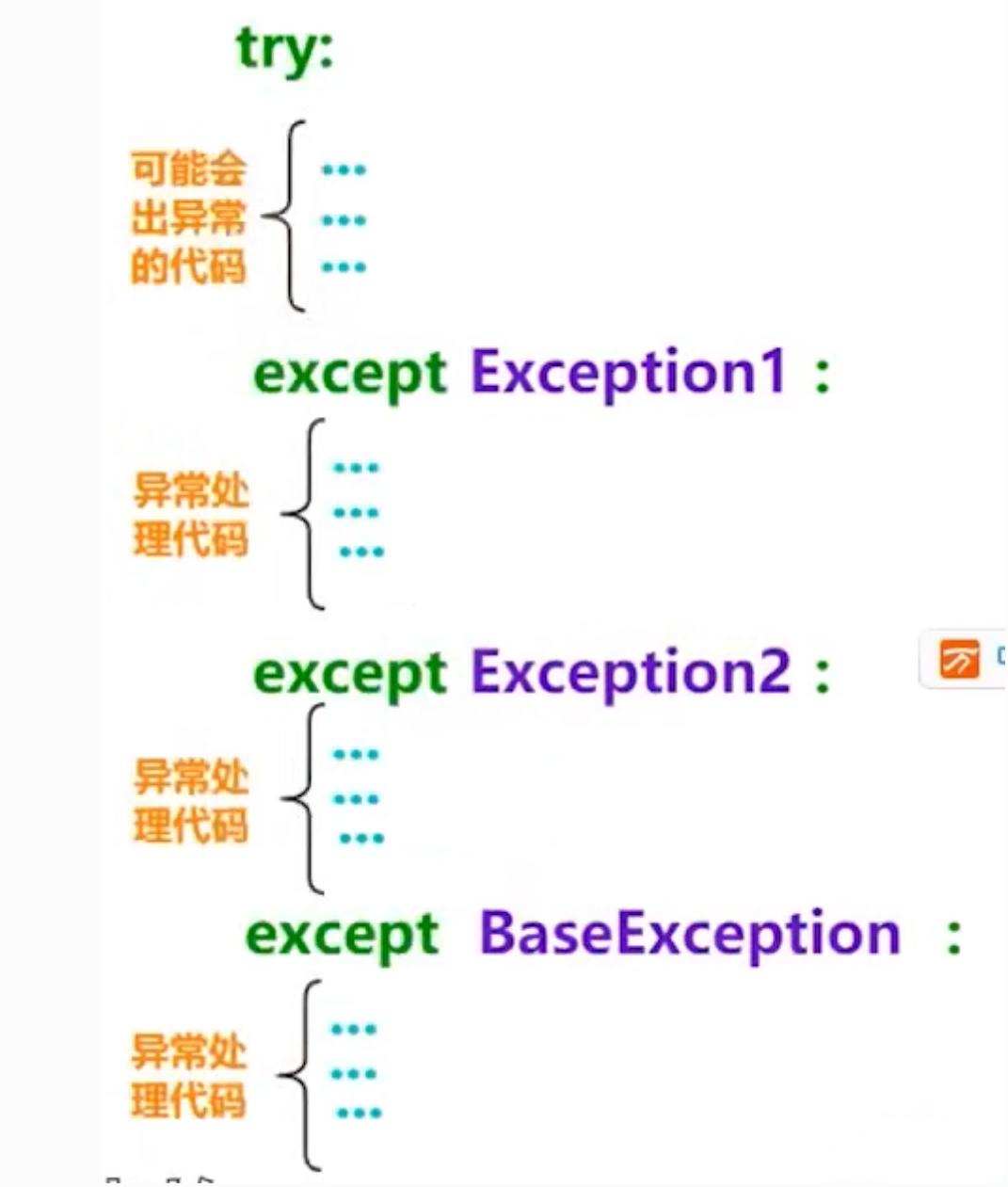
try-except

- Python 提供了异常处理机制，可以在异常出现时即使捕获，然后“内部消化”，让程序继续运行




```python
try:
    n1 = int(input('请输入一个整数: '))
    n2 = int(input('请输入另一个整数: '))
    result = n1 / n2
    print('结果为: ', result)
except ZeroDivisionError:
    print('除数不能为0！！！')
```

- 多个except结构
    - 捕获异常的顺序按照先子类后父类的顺序，为了避免遗漏可能出现的异常，可以在最后增加`BaseException`



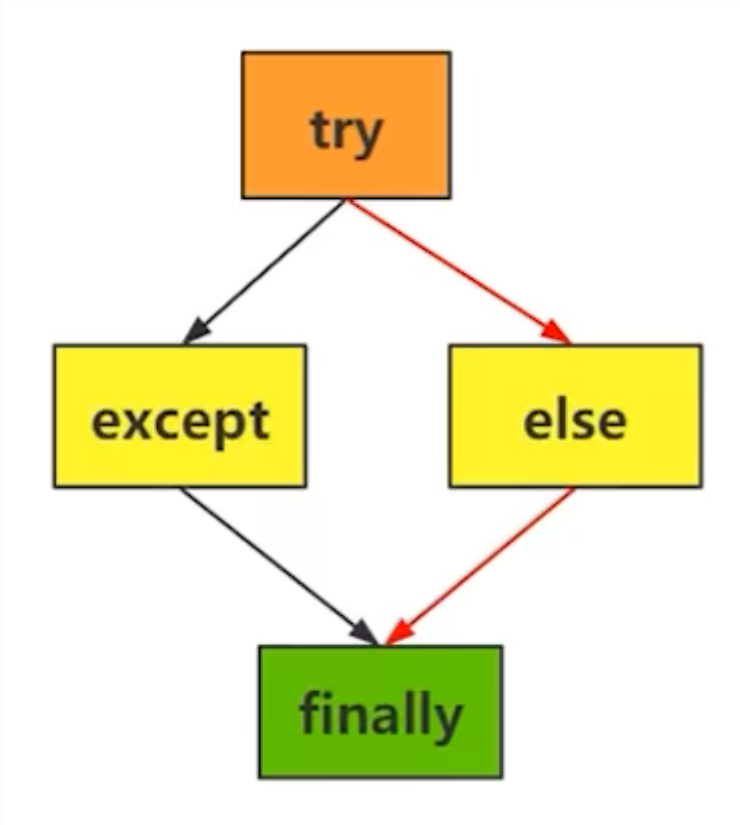

```python
try:
    n1 = int(input('请输入一个整数: '))
    n2 = int(input('请输入另一个整数: '))
    result = n1 / n2
    print('结果为: ', result)
except ZeroDivisionError:
    print('除数不能为0! ! ! ')
except ValueError:
    print('不能将字符串转为数字!!!')
except BaseException as e:
    print(e)
```

try-except-else 结构与 try-except-else-fianlly 结构

- try-except-else 结构
    - 如果try块没有抛出异常，则执行else块，如果try中抛出异常，则执行except块

```python
try:
    n1 = int(input('请输入一个整数: '))
    n2 = int(input('请输入另一个整数: '))
    result = n1 / n2
except BaseException as e:
    print('出错了')
    print(e)
else:
    print('结果为: ', result)
```

- try-except-else-fianlly 结构
    - finally模块无论是否发生异常都会被执行，常用来释放try块中申请的资源




```python
try:
    n1 = int(input('请输入一个整数: '))
    n2 = int(input('请输入另一个整数: '))
    result = n1 / n2
except BaseException as e:
    print('出错了')
    print(e)
else:
    print('结果为: ', result)
finally:
    print('无论是否发生异常，总会被执行的代码')
print('程序结束')
```

Python 中常见的异常类型

异常类型	描述
ZeroDivisionError	除（或取模）0（所有整数类型）
IndexError	序列中没有此索引（index）
KeyError	映射中没有这个键
NameError	未声明/初始化对象（没有属性）
SyntaxError	Python语法错误
ValueError	传入无效的参数

traceback模块的使用

- 使用traceback模块打印异常信息

```
import traceback
try:
    print('1.----------')
    num=10/0
except:
    traceback.print_exc()
```

```
1.----------
Traceback (most recent call last):
  File "E:/dream/chap11/demo1.py", line 7, in <module>
    num=10/0
ZeroDivisionError: division by zero
```

面向对象

类与对象

- 类是多个类似事物组成的群体的统称
- 数据类型
    - 不同的数据类型属于不同的类
    - 使用内置函数`type()`查看数据类型
- 对象
    - 100、99、520都是int类之下包含的相似的不同个例，这些个例的专业术语称为实例或对象

定义Python中的类

- 创建类的语法

```
class 类名 :
    pass
```

- 类的组成
    - 类属性
    - 实例方法
    - 静态方法
    - 类方法

对象的创建

- 对象的创建又称为类的实例化
- 语法

```
实例名 = 类名()
```

- 例子

```
stu = Studnet()
```

- 意义：有了实例，就可以调用类中的内容

类属性、类方法、静态方法

- 类属性：类中方法外的变量称为类属性，被该类的所有对象所共享
- 类方法：使用`@classmethod`修饰的方法，使用类名直接访问的方法
- 静态方法：使用`@staticmethod`修饰的方法，使用类名直接访问的方法

动态绑定属性和方法

- Python 是动态语言，在创建对象之后，可以动态地绑定属性和方法

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def eat(self):
```

```
        print(self.name + '在吃饭')

def show():
    print('我是一函数')

stu = Student('jack', 20)
stu.gender = '男' # 动态绑定属性
print(stu.name, stu.age, stu.gender)
stu.show = show # 动态绑定方法
stu.show()
```

封装

- 面向对象的三大特征
    - 封装：提高程序的安全性
        - 将数据（属性）和行为（方法）包装到类对象中。在方法内部对属性进行操作，在类对象的外部调用方法。这样无需关心方法内部具体实现的细节，从而隔离了复杂度
        - 在Python中没有专门的修饰符用于属性的私有，如果该属性不希望在类外被访问，在前面加两个 `_`
    - 继承：提高代码的复用性
    - 多态：提高程序的可扩展性和可维护性

继承及其实现方式

- 语法格式

```
class 子类类名(父类1, 父类2, ...):
    pass
```

- 如果一个类没有继承任何类，则默认继承object
- Python支持多继承
- 定义子类时，必须在其构造函数中调用父类的构造函数

方法重写

- 如果子类对继承自父类的某个属性或方法不满意，可以在子类中对其（方法体）进行重新编写
- 子类重写后的方法中可以通过 `super().xxx()` 调用父类中被重写的方法

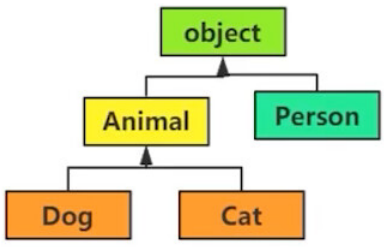
object 类

- object 类是所有类的父类，因此所有类都有object 类的属性和方法
- 内置函数`dir()`可以查看指定对象所有属性
- Object有一个`__str__()`方法，用于返回一个对于“对象的描述”，对应于内置函数`str()`，经常用于`print()`方法，帮我们查看对象的信息，所以我们经常会对`__str__()`进行重写

多态的实现

- 简单来说，多态就是“具有多种形态”，它指的是：即便不知道一个变量所引用的对象到底是什么类型，仍然可以通过这个变量调用方法，在调用过程中根据变量所引用对象的类型，动态决定调用哪个对象中的方法

```
class Animal(object):
    def eat(self):
        print('动物要吃东西')
class Dog (Animal):
    def eat(self):
        print('狗吃肉')
class Cat(Animal):
    def eat(self):
        print('猫吃鱼')
class Person(object):
    def eat(self):
        print('人吃五谷杂粮')
```




```
def fun(animal):
    animal.eat()

fun(Dog())
fun(Cat())
fun(Person())
```

```
狗吃肉
猫吃鱼
人吃五谷杂粮
```

- 静态语言实现多态的三个必要条件
    - 继承
    - 方法重写
    - 父类引用指向子类对象
- 动态语言不需要关心对象是什么类型，只关心对象的行为

特殊属性

名称	描述
__dict__	获得对象或实例对象所绑定的所有属性和方法的字典
__class__	获得对象所属的类
__bases__	获得父类的元组
__base__	获得父类
__mro__	获得类的层次结构
__subclasses__	获得子类的列表

特殊方法

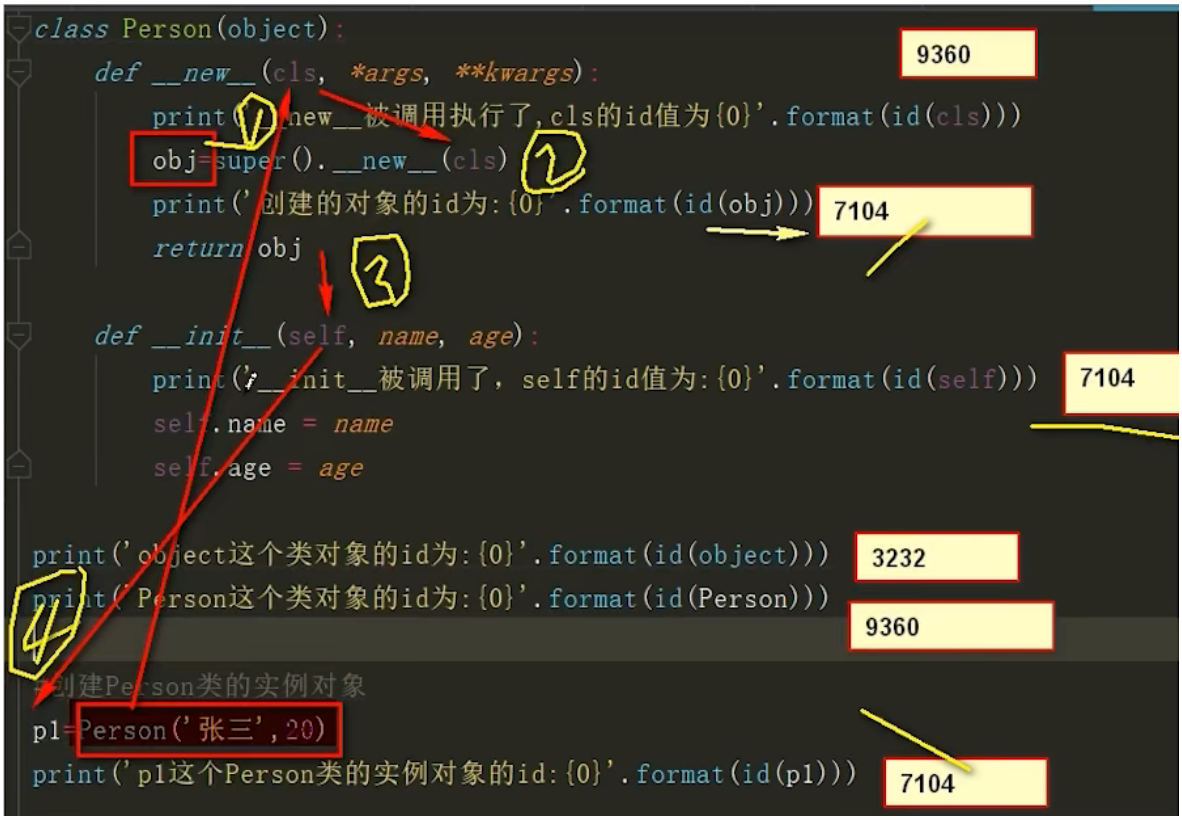
名称	描述
__len__()	通过重写__len__()方法，让内置函数len()的参数可以是自定义类型
__add__()	通过重写__add__()方法，可使自定义对象具有"+"功能
__new__()	用于创建对象
__init__()	对创建的对象进行初始化

new 与 init演示创建对象的过程

```
class Person:
    def __init__(self, name, age):
        print('__init__被调用了')
        self.name = name
        self.age = age

    def __new__(cls, *args, **kwargs):
        print('__new__被调用了')
        obj = super().__new__(cls)
        return obj

p = Preson('张三',  20)
```



类的赋值与浅拷贝

- 变量的赋值操作
  - 只是形成两个变量，实际上还是指向同一个对象
- 浅拷贝
  - Python 拷贝一般都是浅拷贝，拷贝时，对象包含的子对象内容不拷贝，因此，源对象与拷贝对象会引用同一个子对象

深拷贝

- 深拷贝
  - 使用 `copy` 模块的 `deepcopy()` 函数，递归拷贝对象中包含的子对象，源对象和拷贝对象所有的子对象也不相同

模块

模块化编程

- 模块
    - 模块英文为Modules
    - 函数与模块的关系
        - 一个模块中可以包含N个函数
    - 在Python中一个扩展名为.py的文件就是一个模块
    - 使用模块的好处
        - 方便其他程序和脚本的导入及使用
        - 避免函数名与变量名冲突
        - 提高代码的可维护性
        - 提高代码的可复用性

模块的导入

- 创建模块
    - 新建一个.py文件，名称尽量不要与Python自带的标准模块名称相同
- 导入模块

```
import 模块名称 [as 别名]
```

```
from 模块名称 import 函数/变量/类
```

以主程序方式运行

- 在每个模块的定义中都包含一个记录模块名称的变量`__name__`，程序可以检查该变量，以确定它们在哪个模块中执行。如果一个模块不是被导入到其他程序中执行，那么它可能在解释器的顶级模块中执行。顶级模块的`__name__`变量的值为`__main__`

```
if __name__ == '__main__':
    pass
```

Python 中的包

- 包是一个分层次的目录结构，它将一组功能相近的模块组织在一个目录下
- 作用
    - 代码规范
    - 避免模块名称冲突
- 包与目录的区别
    - 包含**init**.py文件的目录称为包
    - 目录里通常不包含**init**.py文件
- 包的导入

```
import 包名.模块名 [as 别名]
```

```
from 包名 import 模块名/函数/变量/类
```

Python 中常用的内容模块

模块名	描述
sys	与Python解释器及其环境操作的标准库
time	提供与时间相关的各种函数的标准库
os	提供访问操作系统服务功能的标准库
calendar	提供与日期相关的各种函数的标准库
urllib	用于读取来自网上（服务器）的数据的标准库
json	用于使用JSON序列化和反序列化对象
re	用于在字符串中执行正则表达式匹配和替换
math	提供标准算术运算函数的标准库
decimal	用于进行精确控制运算精度、有效数位和四舍五入操作的十进制运算
logging	提供了灵活地记录事件、错误、警告和调试信息等日志信息的功能

第三方模块的安装与使用

- 第三方模块的安装

```
pip install 模块名
```

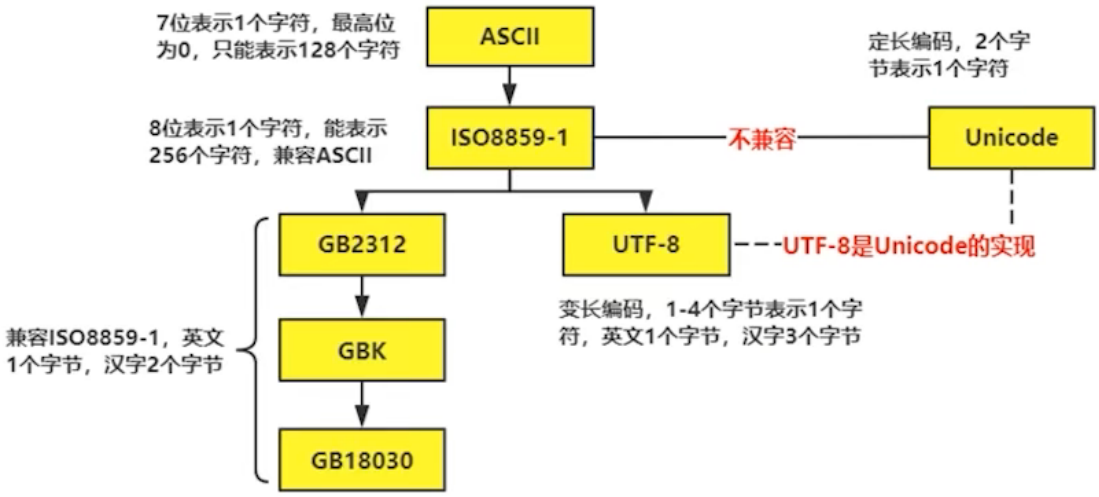
- 第三方模块的使用

```
import 模块名
```

文件读写

编码格式的介绍

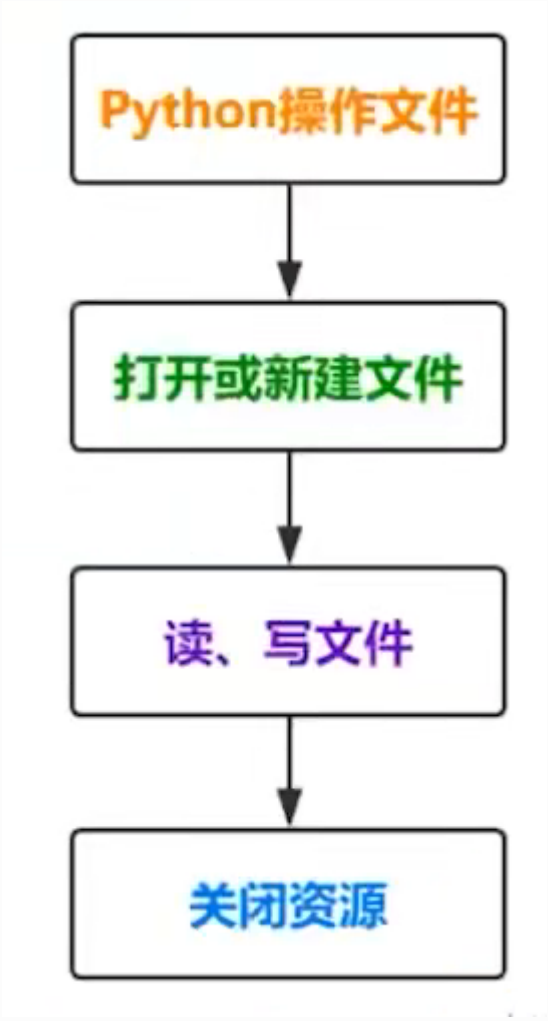
- 常见的字符编码格式
    - Python的解释器使用的是Unicode（内存）
    - .py文件在磁盘上使用UTF-8存储（外存）



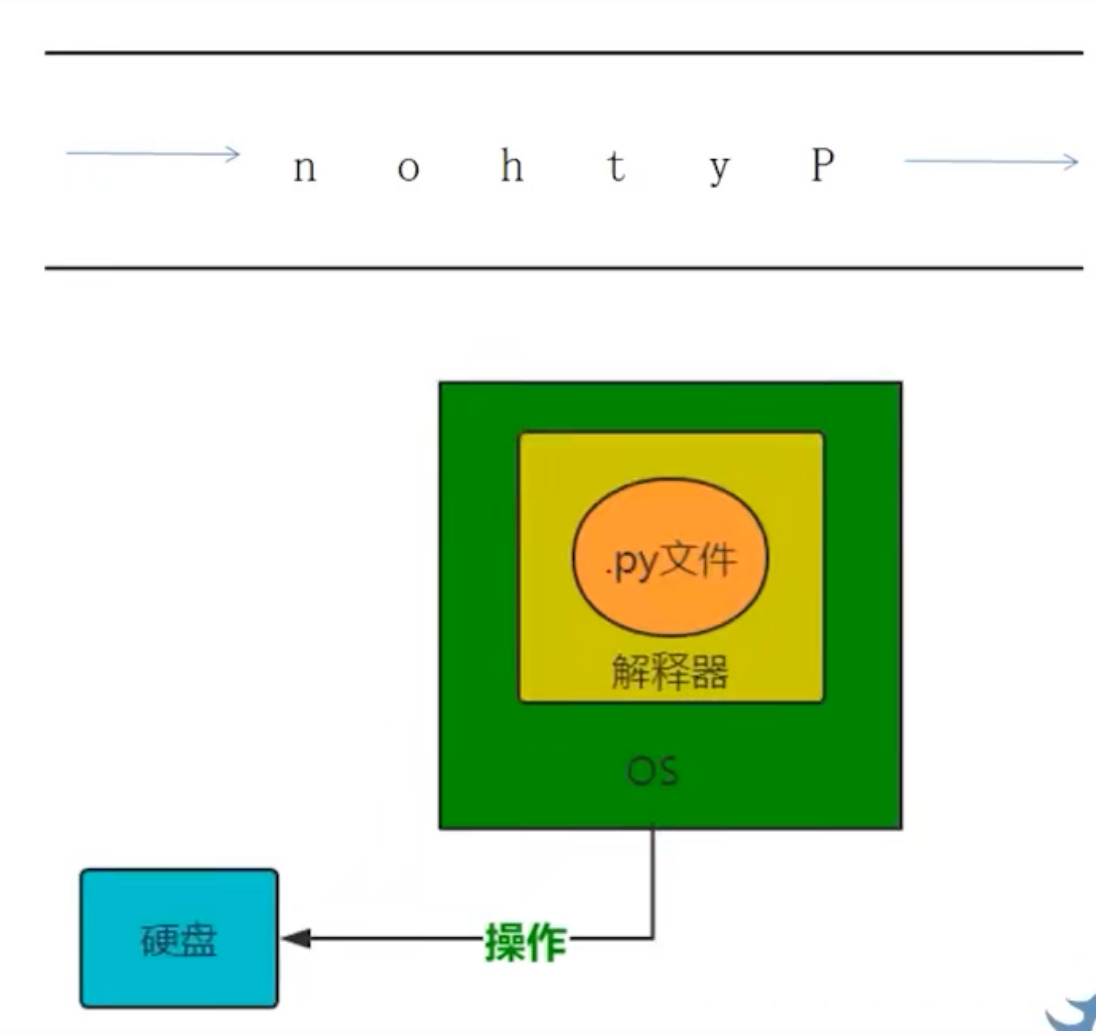
文件读写的原理

- 文件的读写俗称“IO操作”

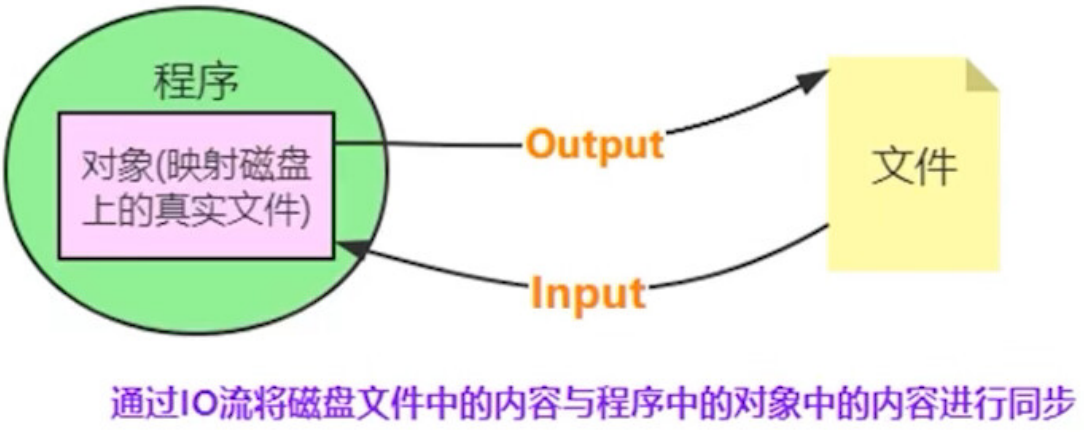
- 文件读写操作流程



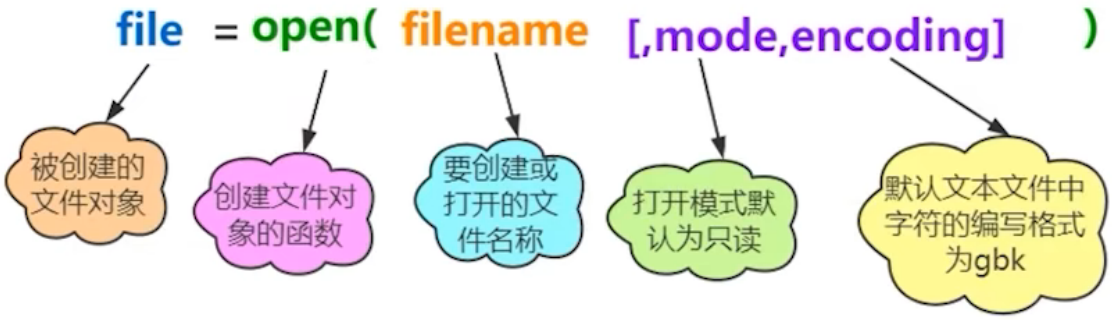
- 操作原理



- 内置函数 `open()` 创建文件对象



- 语法规则




```
file = open('a.txt', 'r')
print(file.readlines())
file.close()
```

常用的文件打开模式

- 文件的类型
    - 按文件中数据的组织形式，文件分为以下两大类
        - **文本文件**：存储的是普通“字符”文本，默认为unicode字符集，可以使用记事本程序打开
        - **二进制文件**：把数据内容用“字节”进行存储，无法用记事本打开，必须使用专用的软件打开，举例：mp3音频文件、jpg图片、doc文档等

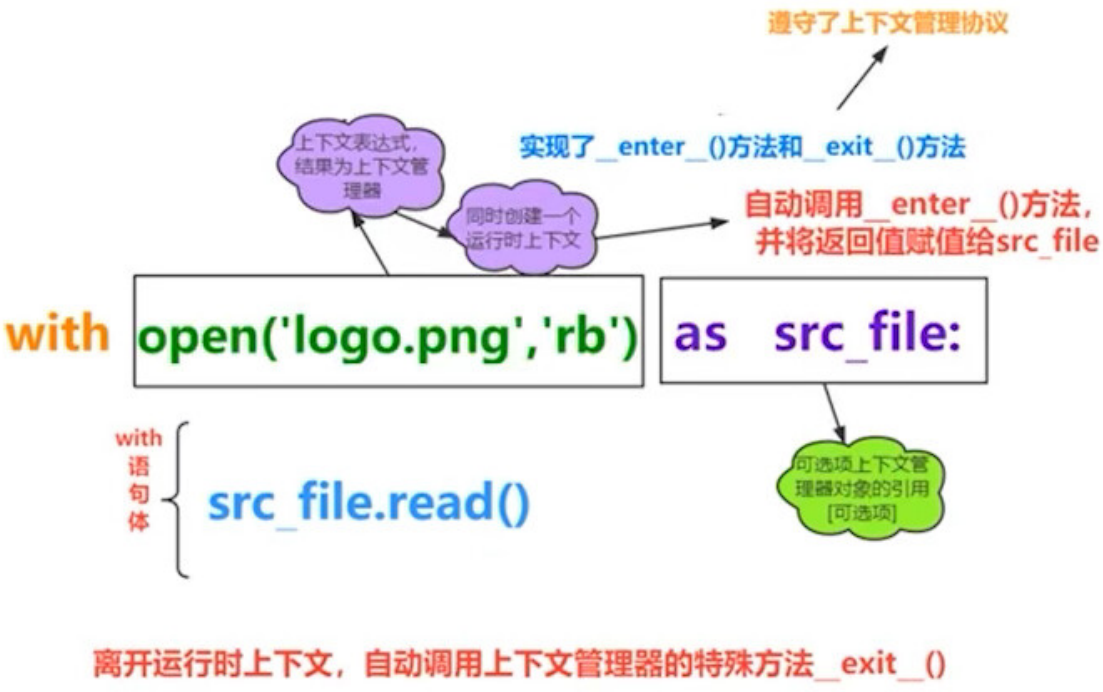
打开模式	描述
r	以只读模式打开文件，文件的指针将会放在文件的开头
w	以只写模式打开文件，如果文件不存在则创建，如果文件存在，则覆盖原有内容，文件指针在文件的开头
a	以追加模式打开文件，如果文件不存在则创建，文件指针在文件开头，如果文件存在，则在文件末尾追加内容，文件指针在文件末尾
b	以二进制方式打开文件，不能单独使用，需要与其他模式一起使用，如 rb 或 wb
+	以读写方式打开文件，不能单独使用，需要与其他模式一起使用，如 a+

文件对象的常用方法

方法名	说明
read([size])	从文件中读取size个字节或字符的内容返回，若省略size，则读取到文件末尾，即一次读取文件所有内容
readline()	从文本文件中读取一行内容
readlines()	把文本文件中每一行都作为独立的字符串对象，并将这些对象放入列表返回
write(str)	将字符串str内容写入文件
writelines(s_list)	将字符串列表s_list写入文本文件，不添加换行符
seek(offset[, whence])	把文件指针移动到新的位置，offset表示相对whence的位置 offset：为正往结束方向移动，为负往开始方向移动 whence：不同的值代表不同的含义 0：从文件开头计算(默认) 1：从当前位置开始计算 2：从文件末尾开始计算
tell()	返回文件指针的当前位置
flush()	把缓冲区的内容写入文件，但不关闭文件
close()	把缓冲区的内容写入文件，同时关闭文件，释放文件对象相关资源

with 语句

- with 语句可以自动管理上下文资源，不论什么原因跳出with块，都能确保文件正确关闭，以此来达到释放资源的目的



> 上下文管理器实现了**enter**()和**exit**()方法

os 模块的常用函数

- os模块是Python内置的与操作系统功能和文件系统相关的模块，该模块中的语句的执行结构通常与操作系统有关，在不同的操作系统上运行，得到的结果可能不一样
- os模块与os.path模块用于对目录或文件进行操作

```
os.system('notepad.exe')
os.system('calc.exe')
# 直接调用可执行文件
os.startfile('C:\\Program Files\\Tencent\\QQ\\Bin\\qq.exe')
```

- os模块操作目录相关函数

函数	说明
getcwd()	返回当前的工作目录
listdir(path)	返回指定路径下的文件和目录信息
mkdir(path[, mode])	创建目录
makedirs(path1/path2...[, mode])	创建多级目录
rmdir(path)	删除目录
removedirs(path1/path2...)	删除多级目录
chdir(path)	将path设置为当前工作目录

os.path 模块的常用方法

函数	说明
abspath(path)	用于获取文件或目录的绝对路径
exists(path)	用以判断文件或目录是否存在，如果存在返回True，否则返回False
join(path, name)	将目录与目录或者文件名拼接起来
splitext()	分离文件名和扩展名
basename(path)	从一个目录中提取文件名
dirname(path)	从一个路径中提取文件路径，不包括文件名
isdir(path)	用于判断是否为路径，不包括文件名
split(path)	分离目录和文件名
walk(path)	递归遍历目录下的所有文件

```
inport os
path = os.getcwd()
lst_files = os.walk(path)
```

```
for dirpath,dirname,filename in lst_files:
    '''print(dirpath)
    print(dirname)
    print(filename)
    print('--------------------------------')'''
    for dir in dirname:
        print(os.join(dirpath, dir))

    for file in filename:
        print(os.join(ditpath, file))

    print('--------------------------------')
```