

基本输出函数

1. print()

作用：输出运算结果；根据输出内容的不同，有三种用法

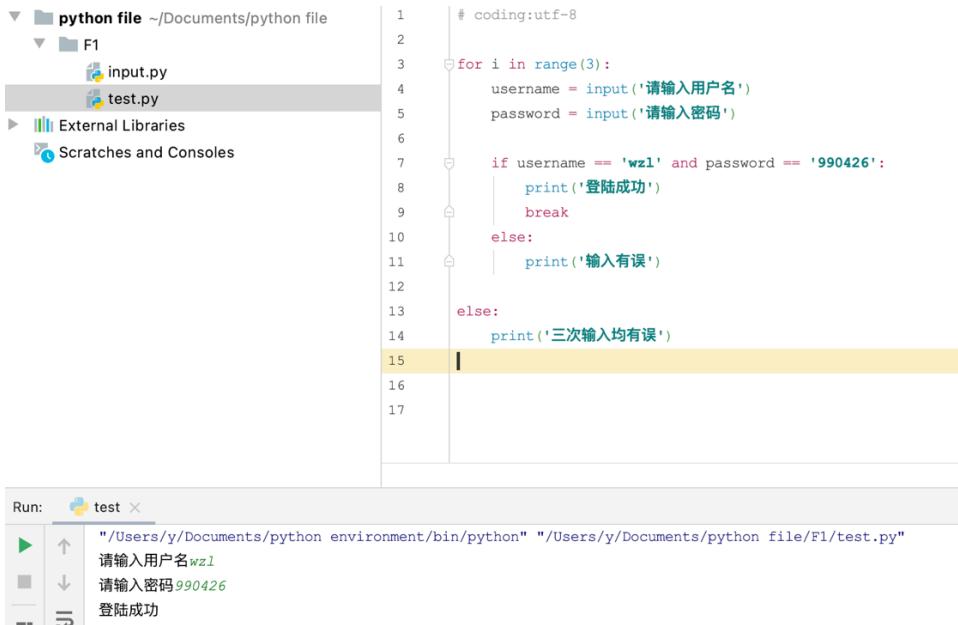
- 仅用于输出字符串： `print('待输出的字符串')`
 - 用于输出一个或多个变量： `print(变量 1, 变量 2, ……)`
 - 用于混合输出字符串和变量： `print('输出字符串模版'.format(变量 a,变量 b))`， 例如 `print('每人支付', everyone_pay, '元')`
 - 其中，输出字符串中用 {} 表示一个槽位
-
- (`print` 默认自动换行) —— 想要在同一行，`print(a, end="")`, `print(b) …..` 其中，“中间空格等于 ab 间隙；`\n` 表示换行，相当于回车键
 - `end='增加输出的结尾'`，例如 `%`

2. input() 函数

作用：从控制台获得用户的一行输入，无论用户输入什么内容，`input()` 函数都以字符串类型返回结果。

变量 = `input('提示性文字')`

注意：无论用户输入的是字符或是数字，`input()` 函数统一按照字符串类型输出。



```
# coding:utf-8
for i in range(3):
    username = input('请输入用户名')
    password = input('请输入密码')
    if username == 'wzl' and password == '990426':
        print('登陆成功')
        break
    else:
        print('输入有误')
else:
    print('三次输入均有误')
```

Run: test

/Users/y/Documents/python environment/bin/python "/Users/y/Documents/python file/F1/test.py"

请输入用户名wzl
请输入密码990426
登陆成功

语法元素的名称

命名规则：python 语言允许采用大写字母，小写字母，数字，下划线和汉字等字符及其组合给变量命名，但名字的首字符不能是数字，中间不能出现空格；标识符名称不能与 python 保留字相同。

注意：标识符对大小写敏感

控制台——`help('keywords')` 获取 保留字

数字类型

python 语言提供 3 种数字类型（表示数字或数值的数据类型）：整数、浮点数和复数，分别对应数学中的整数、实数和复数。

一个整数值可以表示为十进制、十六进制、八进制和二进制等不同进制类型。

十进制 180

十六进制（0x 或 0X 开头）：0xb4

八进制（0o 或 0O 开头）：0o264

二进制（0b 或 0B 开头）：0b10110100

- 整型 int
- 浮点型 float
- 内置函数 — type

1. 整型

- 就是整数，0 也是整数，但是特殊的整数
- int 既是整型的代表，又是定义整型的内置函数
- 定义一个整型，并不一定非要使用 int

```
count_100_01 = int(100)
```

```
count_100_02 = 100
```

```
100 100
```

2. 浮点型

- 就是小数，凡是带有小数点的类型，都可以认为是浮点类型
- float 既是浮点型的代表，又是浮点型定义的内置函数

```
pi_01 = float(3.14)
```

```
pi_02 = 3.14
```

- 定义 float 类型的时候，并不需要一定使用 float

3. 内置函数 --type

- 返回变量的类型
- type（已经被赋值的变量名或变量）

```
count = 1050
```

```
print(type(count))
```

```
print(type(3.1415926))
```

基本概念

头部注释

1. 脚本第一行用#号表示的信息就是头部注释，例如 `# coding: utf-8`
一般是被系统和解释器调用
2. 注释符号和注释内容
3. 常见头部注释：
国内常见 - `# coding: utf-8` 定义 coding 则告诉系统脚本是何编码

目前少见使用 - `#!/usr/bin/env` 定义#! 会去找指定路径下的 python 解释器

导入位置

1. 导入 是 将 python 的一些功能函数放到当前的脚本中使用
2. 不导入的功能无法直接在当前脚本使用（除了内置函数）
3. 为什么要导入 —— 借用别人的功能
4. 在头注释的下边
5. 例 - `import(内置导入函数) os(被导入模块)`

完整脚本

```
# coding:utf-8
```

```
import os
```

```
print('hello world')
```

代码的执行顺序

1. 如何执行 —— 自上而下，逐行执行
2. 什么是内置函数 —— 自动导入
3. 第一个内置函数 —— `print`
`print(object, end="")` 参考 基本输出函数

注释

1. 什么是 —— 在代码中，不会被 python 直接运行的语句
2. 为什么 —— 降低维护成本，方便二次理解
3. 三种用法
 - i. `#`,
 - ii. # 看到这行代码的人注意了，这是一个测试代码（和注释代码齐平；或者在下个代码的上一行）
 - iii. 三引号,
"""

这是三引号注释的第一种双引号形式，我们可以随意换行

```
"""
iii. 单引号,
'''
```

这是三引号注释的第二种单引号形式，功能和双引号完全一致
'''

引号注释一般用在两个地方：

1. 是整个脚本的开篇，作用在于给别人介绍一下这个脚本的作用；
2. 对于函数功能的解释

脚本执行的入口

1. 什么是 —— 就像赛车进入赛道需要一个入口；一般称为 主函数 main；
2. 代码脚本的写法 —— __name__ == '__main__' ; if __name__ == '__main__':
3. 通过程序入口判断程序是直接运行还是被调用
4. 并不一定需要脚本入口

变量和变量名

1. 赋值语句
2. 变量存在哪里 —— 内存里；每个变量被定义后存入一个内存块
3. 变量名规则 —— 语法元素的名称

字符串类型

1. 用 " 或 "" 包裹的信息就是 字符串
2. 可以包含任意信息
3. 用 str 来代表字符串类型，并且通过该函数可以定义字符串
4. 字符串是不可改变的！
5. 内置函数 id,
返回变量的内存地址
数字地址 = id (变量)
print(id(变量))
6. 内置函数 len
返回字符串的长度
无法返回数字类型的长度，因为数字类型没有长度
返回值 = len (字符串)
length = len ('python 是一门很好的语言')
print(length)

内置成员运算符 in 的使用

1. 用来判断你的数据中是否存在你想要的成员
'开发' in 'python 开发' —— True; False

内置函数 max

1. max 函数返回数据中最大的成员
2. max (数据) —— 成员值
print(max('今天是 1 月 3 日')) ? —— 月

3. 大小关系：
中文符号 > 字母> 数字> 英文符号
中文按照拼音的首字母来计算

内置函数 min

返回数据中最小的成员

字符串的累加

1. 字符串不是数字不能做减法，乘除法
2. 字符串的拼接，用 + 这个符号

```
a = '123'  
b = '456'  
c = a + b  
print(c) -> '123456'
```

布尔类型

1. 定义：真假的判断 即 布尔类型
2. 固定值：True ; False
3. 布尔值
4. 使用：bool 代表 布尔类型 也可以对结果进行真假判断

```
res = bool('name' in 'my name is hsy')  
print(res)  
-> True
```
5. 字符串和数字的布尔应用

```
int 0 -> False ; 非 0 -> True  
float 0.0 -> False ; 非 0.0 -> True  
str '' -> False ; 非空字符串 -> True
```
6. 在计算机中 0 1 上计算机的最原始形态，单个占空间也最小，故而经常会用 0 1 用来代替 True 和 False

空类型

1. 不属于任何类型
2. 固定值：None
3. 空类型 属于 False 的范畴
4. 如果不确定类型的时候 可以使用空类型

列表

1. 列表就是队列
2. 是各种数据类型的集合，也是一种数据结构
3. 列表是一种有序，且内容可重复的集合类型
4. 在 Python 中，list 代表 列表这种类型，也可以用来定义一个列表
5. 列表中的元素 存在于 一个[]中

```
In [1]: names_01 = list(['dewei', '小慕', 'dewei'])
```

```
In [2]: names_02 = ['dewei', '小慕', 'dewei']
```

6. 在 python 中，列表是一个无限制长度的数据结构

7.

列表中的类型

- ◆ str_array = ['dewei', 'haha', ' ', '']
- ◆ int_array = [1, 2, 3, 0, 10, 110]
- ◆ float_array = [1.1, 10.3, 0.1, 0.0, 3.1415926]
- ◆ bool_array = [True, False, False, True]
- ◆ none_array = [None, None, None]
- ◆ list_array = [[1,2,3], [1.2, 3.1]]
- ◆ mix_array = ['dewei', 1, 3.14, None, True]

8.

in, max, min 在列表中的使用

- ◆ 1 in [1 , 2 , 3 , 4] -> True; 10 in [1,2,3,4] -> False
- ◆ max([1,2,3,4]) -> 4
- ◆ min([1,2,3,4]) -> 1

9. max 和 min 在列表使用的时候，列表中的元素不能是多种类型，否则会报错

元组类型

1. 元组和列表一样，都是一种可以存储多种数据结构的队列
2. 元组也是一个有序的，且元素可以重复的集合
3. tuple 代表元组类型，也可以用来定义一个元组
4. 元组中的元素存在于一个 () 小括号中

```
names_01 = tuple(('wzl', 'hsy'))
```

```
names_02 = ('wzl', 'hsy')
```

如果元组中只有一个元素，后面要跟一个 ，

5. 元组是一个无限制长度的数据结构

6. 列表和元组的区别

- 元组比列表占用资源更小
- 列表是可变的，元组不是

7.

元组中的类型

- ◆ str_tuple = ('dewei', 'haha', '', '')
- ◆ int_tuple = (1, 2, 3, 0, 10, 110)
- ◆ float_tuple = (1.1, 10.3, 0.1, 0.0, 3.1415926)
- ◆ bool_tuple = (True, False, False, True)

元组中的类型

- ◆ none_tuple = (None, None, None)
- ◆ tuple_tuple = ((1, 2, 3), (1.2, 3.1))
- ◆ list_tuple = ([123, 456], [6789, 1234])
- ◆ mix_tuple = ('dewei', 1, 3.14, None, True)

```
tuple_array = [('a', 'b'), ('c', 'd'), ('e',)]
```

8. in, max, min 在元组中的使用和列表类似

字典

1. 字典是由多个键 (key) 以及其对应的值 (value) 所组成的一种数据类型
2. dict 代表字典，并且可以创建一个字典
3. 通过{}将一个个 key 与 value 存入字典中

```
a = dict()  
a = {}
```

```
person = {'name': 'dewei', 'age': 33}
```

4. key 支持字符串，数字和元组类型，但列表是不支持的
5. value 支持所有的 python 的数据类型

```
a = {'name': 'dewei', 'age': 30}  
b = {1: 'one', 2: 'two'}  
c = {(1, 2, 3): [1, 2, 3], (4, 5, 6): [4, 5, 6]}
```

6.

列表与元组中的字典

◆ dict_array = [{1:1, 2:2}, { 'one' : 1, 'two' : 2}]

◆ dict_tuple = ((1:1, 2:2), { 'one' : 1, 'two' : 2})

◆ 元组一旦创建，就不可改变

7. 字典中的每一个 key 一定是唯一的

数字的运算

1. 赋值运算符：

(取模就是取余数)

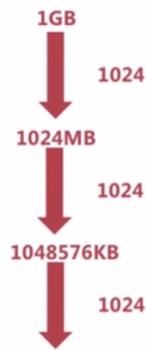
运算符	描述	举例
=	等于运算符	c = a + b
+=	加法运算符	c += a -> c = c + a
-=	减法运算符	c -= a -> c = c - a
*=	乘法运算符	c *= a -> c = c * a
/=	除法运算符	c /= a -> c = c / a
%=	取模运算符	c %= a -> c = c % a
**=	幂运算符	c **= a -> c = c ** a
//=	整除运算符	c // a -> c = c // a

2.

b kb mb gb 的转换

◆b kb mb gb 是计算机的计量单位

◆1024相差量



3. 字符串无法与字符串作乘法
字符串只可以和数字作乘法

```
name = 'xiaomu'  
print(name * 3)  
>> 'xiaomuxiaomuxiaomu'
```

4. 字典类型不支持乘法，元组支持

比较/身份运算符

1.

运算符	描述	举例
<code>==</code>	判断是否等于	<code>a == b</code>
<code>!=</code>	判断是否不等于	<code>a != b</code>
<code>></code>	判断是否大于	<code>a > b</code>
<code><</code>	判断是否小于	<code>a < b</code>
<code>>=</code>	判断是否大于等于	<code>a >= b</code>
<code><=</code>	判断是否小于等于	<code>a <= b</code>
<code><></code>	判断是否不等于	<code>a <> b</code>
<code>is</code>	判断两个对象存储单元是否相同	<code>a is b</code>
<code>is not</code>	判断两个对象存储单元是否不同	<code>a is not b</code>

(只有 python2 可以使用 `<>`)

2. 前面是比较运算符

`is` 和 `is not` 是身份运算符

3. 单元存储就是我们提过的内存块

4. 0~255

补充 - format

Python2.6 开始，新增了一种格式化字符串的函数 `str.format()`，它增强了字符串格式化的功能。

基本语法是通过 `{}` 和 `:` 来代替以前的 `%`。

`format` 函数可以接受不限个参数，位置可以不按顺序。

实例

```
>>>"{} {}".format("hello", "world")      # 不设置指定位置, 按默认顺序
'hello world'

>>> "{0} {1}".format("hello", "world")  # 设置指定位置
'hello world'

>>> "{1} {0} {1}".format("hello", "world")  # 设置指定位置
'world hello world'
```

也可以设置参数：

实例

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

print("网站名: {name}, 地址 {url}".format(name="菜鸟教程", url="www.runoob.com"))

# 通过字典设置参数
site = {"name": "菜鸟教程", "url": "www.runoob.com"}
print("网站名: {name}, 地址 {url}".format(**site))

# 通过列表索引设置参数
my_list = ['菜鸟教程', 'www.runoob.com']
print("网站名: {0[0]}, 地址 {0[1]}".format(my_list))  # "0" 是必须的
```

输出结果为：

```
网站名: 菜鸟教程, 地址 www.runoob.com
网站名: 菜鸟教程, 地址 www.runoob.com
网站名: 菜鸟教程, 地址 www.runoob.com
```

也可以向 `str.format()` 传入对象：

实例

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

class AssignValue(object):
    def __init__(self, value):
        self.value = value
my_value = AssignValue(6)
print('value 为: {}'.format(my_value)) # "0" 是可选的
```

输出结果为：

```
value 为: 6
```

补充 - append

append 每次只能添加一个值

更新列表

你可以对列表的数据项进行修改或更新，你也可以使用`append()`方法来添加列表项，如下所示：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

list = []          ## 空列表
list.append('Google') ## 使用 append() 添加元素
list.append('Runoob')
print list
```

认识对象

认识 python 中的对象

1. python 中一切都是对象
2. 每个对象都有各自的属性与方法
3. 对象里的特点就是它的属性，它的功能就是它的方法

字符串 capitalize()首字母大写方法

1. 功能：将字符串的首字母大写，其他字母小写

capitalize的用法

用法：

```
newstr = string.capitalize()
```

参数：

函数括弧内什么都不用填写

```
In [1]: name = 'xiaoMu'
```

```
In [2]: new_name = name.capitalize()
```

```
In [3]: print(new_name)
```

```
Xiaomu
```

2. 注意事项：

- 只对第一个字母有效
- 只对字母有效
- 已经是大写，则无效

字符串的 upper 函数/ lower 函数（作用则相反）

1. 将全体字母大写

2. 用法：

用法：

```
big_str = string.upper()
```

参数：

函数括弧内什么都不用填写

```
In [8]: name = 'xiaomu'
```

```
In [9]: big_name = name.upper()
```

```
In [10]: print(big_name)
```

```
XIAOMU
```

3. 注意事项：

- 只对字符串中的字母有效
- 已经是大写，则无效

字符串的 swapcase 函数

1. 功能： 将字符串中大小写字母进行转换
2. 用法：

用法：

```
newstr = string.swapcase()
```

参数：

函数括弧内什么都不用填写

```
In [11]: name = 'DeWei'  
In [12]: new_name = name.swapcase()  
In [13]: print(new_name)  
dEwEI
```

3. 注意事项：

只对字符串中的字母有效

字符串的 zfill 函数

1. 功能：
为字符串定义长度，如不满足，缺少的部分用 0 填补
2. 用法：

zfill的用法

用法：

```
newstr = string.zfill(width)
```

参数：

width: 新字符串希望的宽度

```
In [14]: name = 'xiaomu'  
In [15]: new_name = name.zfill(10)  
In [16]: print(new_name)  
0000xiaomu
```

3. 注意事项：

- 与字符串的字符无关
- 如果定义长度小于当前字符串长度，则不发生变化

字符串的 count 函数

1. 功能：
返回当前字符串中某个成员（元素）的个数

2. 用法：

用法：

```
inttype = string.count(item)
```

参数：

item: 查询个数的元素

```
In [17]: info = 'my name is dawei'
```

```
In [18]: print(info.count('e'))  
3
```

3. 注意事项：

- 查询元素不存在，则返回 0

字符串的 startswith 和 endswith 函数

1. 功能：

- startswith 是判断字符串开始位是否是某元素
- endswith 则判断字符串结尾是否是某元素

2. 用法：

用法：

string.startswith(item) -> item: 你想查询匹配的元素, 返回一个布尔值

string.endswith(item) -> item: 你想查询匹配的元素, 返回一个布尔值

```
In [20]: 'my name is dawei'.startswith('my')  
Out[20]: True
```

```
In [21]: 'my name is dawei'.endswith('my')  
Out[21]: False
```

字符串的 find 和 index 函数

1. 功能：

都是返回你想寻找的成员的位置

2. 用法：

用法：

string.find(item) -> item: 你想查询的元素, 返回一个整型

string.index(item) -> item: 你想查询的元素, 返回一个整型

或者报错

Ps: 字符串里的位置是从左向右, 以0开始的

```
In [22]: 'my name is dawei'.find('e')  
Out[22]: 6
```

```
In [23]: 'my name is dawei'.index('i')  
Out[23]: 8
```

3. 区别：

◆如果find找不到元素，会返回-1

◆如果index 找不到元素，会导致程序报错

字符串的 strip 函数

1. 功能：

strip 将去掉字符串左右两边的指定元素，默认是空格

2. 用法：

用法：

```
newstr = string.strip(item)
```

参数：

括弧里需要传一个你想去掉的元素，可不填写

```
In [25]: ' hello xiaomu '.strip()  
Out[25]: 'hello xiaomu'
```

```
In [26]: 'hello xiaomu'.strip('h')  
Out[26]: 'ello xiaomu'
```

3. 拓展知识：

- 传入的元素如果不在开头或结尾则无效
- lstrip 仅去掉字符串开头的指定元素或空格
- rstrip 仅去掉字符串结尾的指定元素或空格

字符串的 replace 函数

1. 功能：

将字符串中的 old (旧元素) 替换成 new (新元素)，并能指定替换的数量

2. 用法：

replace的用法

用法：

```
newstr = string.replace(old, new, max)
```

参数：

old：被替换的元素，

new：替代old的新元素，

max：可选，代表替换几个，默认全部替换全部匹配的old元素

```
In [29]: 'hello, dawei'.replace('dawei', 'xiaomu')  
Out[29]: 'hello, xiaomu'
```

```
In [30]: 'hello, xiaomu'.replace('l', '0', 1)  
Out[30]: 'he0lo, xiaomu'
```

字符串中返回 bool 类型的函数集合

1. isspace

功能：判断字符串是否是一个由空格组成的字符串

用法：

```
booltype = string.isspace () -> 无参数可传，返回一个布尔类型
```

```
In [31]: ' '.isspace()
Out[31]: True
```

```
In [32]: 'hello xiaomu'.isspace()
Out[32]: False
```

ps：由空格组成的字符串，不是空字符串：" != "

2. istitle

功能：判断字符串是否是一个标题类型

用法：

```
booltype = String.istitle () -> 无参数可传，返回一个布尔类型
```

```
In [33]: 'Hello Xiaomu'.istitle()
Out[33]: True
```

```
In [34]: 'hello xiaomu'.istitle()
Out[34]: False
```

ps：该函数只能用于英文

3. isupper / islower

功能：判断字符串中的字母是否都是大写 / 小写

用法：

```
booltype = string.isupper () -> 无参数可传，返回一个布尔类型
```

```
booltype = string.islower() -> 无参数可传，返回一个布尔类型
```

```
In [35]: 'hello xiaomu'.islower()
Out[35]: True
```

```
In [36]: 'hello xiaomu'.isupper()
Out[36]: False
```

ps：只检测字符串里的字母，对其他字符不做判断

字符的编码格式

1. 什么是编码格式 ——

有一定规则的规则，使用了这种规则，我们就能指定传输的信息是什么意思

2. 常见的编码格式

- gbk 中文编码
- ascii 英文编码
- utf-8 是一种国际通用的编码格式

字符串格式化的常用格式符

1. 定义：

◆用于对应各种数据类型的格式化符号-----格式化符号

符号	说明
%s	格式化字符串，通用类型
%d	格式化整型
%f	格式化浮点型
%u	格式化无符号整型（正整型）
%c	格式化字符

符号	说明
%o	格式化无符号八进制数
%x	格式化无符号16进制数
%e	科学计数法格式化浮点数

2. 例子：

```
7     print('%u' % -1)
8     print('%f' % 1.2)
9     print('%f' % 3.14)
L0
```

format2 ×

```
-1
1.200000
3.140000 I
```

3. format 形式

支持类型：

```
print('{:d}'.format(1))
print('{:f}'.format(1.2))
```

字符串的转义字符

1. 什么是转义字符

- 字符要转成其他含义的功能
- \+字符

2. python 中的转义字符

符号	说明
\n	换行，一般用于末尾，strip对其也有效
\t	横向制表符（可以认为是一个间隔符）
\v	纵向制表符（会有一个男性符号）
\a	响铃
\b	退格符，将光标前移，覆盖（删除前一个）
\r	回车
\f	翻页（几乎用不到，会出现一个女性符号）
\'	转义字符串中的单引号
\\"	转义字符串中的双引号
\\\	转义斜杠

列表元组的操作符

1. 累加与乘法：

```
In [7]: names = ['xiaomu', 'dewei', 'xiaowang']

In [8]: new_names = names + names

In [9]: print(new_names)
['xiaomu', 'dewei', 'xiaowang', 'xiaomu', 'dewei', 'xiaowang']

In [10]: names = ['xiaomu', 'dewei', 'xiaowang']

In [11]: new_names = names * 2

In [12]: print(new_names)
['xiaomu', 'dewei', 'xiaowang', 'xiaomu', 'dewei', 'xiaowang']
```

列表（元组）的添加 - insert 函数

1. 功能：

将一个元素添加到当前列表的指定位置中

2. 用法：

```
In [22]: fruits = ['苹果', '西瓜', '水蜜桃']

In [23]: fruits.insert(1, '水晶梨')

In [24]: fruits
Out[24]: ['苹果', '水晶梨', '西瓜', '水蜜桃']
```

3. insert 和 append 区别：

- ◆ append只能添加到列表的结尾，而insert可以选择任何一个位置
- ◆ 如果insert传入的位置列表中不存在，则将新元素添加到列表结尾
- ◆ 字符串，元组，列表 元素的位置是从 0 开始 计算的

列表（元组）的 count 函数

1. 功能：

返回当前列表中某个成员的个数

2. 用法：

用法：

```
inttype = list.count(item)
```

参数：

item：你想查询个数的元素

```
In [25]: fruits = ['苹果', '西瓜', '水蜜桃', '西瓜', '雪梨']  
In [26]: count = fruits.count('西瓜')  
In [27]: print(count)  
2
```

3. 注意事项：

- 如果查询的成员（元素）不存在，则返回 0
- 列表只会检查完整元素是否存在需要计算的内容

```
In [28]: fruits = ['苹果', '西瓜', '水蜜桃', '西瓜', '雪梨']  
In [29]: count = fruits.count('西')  
  
In [30]: count  
Out[30]: 0
```

列表的 remove 函数

1. 功能：

删除列表中的某个元素

2. 用法：

用法：

```
list.remove(item)
```

参数：

item：准备删除的列表元素

```
In [31]: drinks = ['雪碧', '可乐', '矿泉水']  
In [32]: drinks.remove('矿泉水')  
  
In [33]: drinks  
Out[33]: ['雪碧', '可乐']
```

3. 注意事项：

- 如果删除的成员（元素）不存在，会直接报错
- 如果被删除的元素有多个，只会删除第一个
- remove 函数不会返回一个新的列表，而是在原先的列表中对元素进行删除

4. python 内置函数 del：

```
del 把变量完全删除

In [34]: drinks = ['雪碧', '可乐', '矿泉水']

In [35]: del drinks

In [36]: print(drinks)
-----
NameError                                 Traceback (most recent call last)
<ipython-input-36-75a320656267> in <module>
      1 print(drinks)

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

5. 例子：

```
shops = ['可乐', '洗发水', '可乐', '牛奶', '牛奶', '牙膏', '牙膏']

print('我们的超市有这些内容：%s' % shops)
print('我们的可乐有%s件产品' % shops.count('可乐'))
print('我们的牛奶有%s件产品' % shops.count('牛奶'))
print('我们的牙膏有%s件产品' % shops.count('牙膏'))
print('我们的洗发水有%s件产品' % shops.count('洗发水'))
```

列表 reverse 函数

1. 功能：

对当前列表顺序进行反转

2. 用法：

```
list.reverse()
```

参数：

无参数传递

```
In [37]: drinks = ['雪碧', '可乐', '矿泉水']

In [38]: drinks.reverse()

In [39]: print(drinks)
['矿泉水', '可乐', '雪碧']
```

3. 例子：

```
students = [
    {'name': 'dewei', 'age': 33, 'top': 174},
    {'name': '小慕', 'age': 10, 'top': 175},
    {'name': 'xiaogao', 'age': 18, 'top': 188},
    {'name': 'xiaoyun', 'age': 18, 'top': 165}
]

print('当前的同学顺序是{}'.format(students))
students.reverse()
print('座位更换之后的顺序是{}'.format(students))
```

列表中的 sort 函数

1. 功能：

对当前列表按照一定规律进行排序

2. 用法：

用法：

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

参数：

key - 参数比较

reverse -- 排序规则 , reverse = True 降序 , reverse = False 升序 (默认)。

```
In [44]: books = ['python', 'django', 'web', 'flask', 'tornado']
```

```
In [45]: books.sort()
```

```
In [46]: print(books)
```

```
['django', 'flask', 'python', 'tornado', 'web']
```

按照首字母顺序排序

3. 注意事项：

列表中的元素类型必须相同，否则无法排序（报错）

列表的 clear 函数

1. 功能：

将当前列表中的数据清空

2. 用法：

```
list.clear() -> 该函数无参数，无返回值
```

```
In [49]: target = [1, 2, 3, 4, 5, 6]
```

```
In [50]: target.clear()
```

```
In [51]: print(target)
[]
```

列表的 copy 函数

1. 功能：

将当前的列表复制一份相同的列表，新列表与旧列表内容相同，但内存空间不同

2. 用法：

用法：

```
list.copy() -> 该函数无参数，返回一个一模一样的列表
```

```
In [52]: old_list = ['a', 'b', 'c']
```

```
In [53]: new_list = old_list.copy()
```

```
In [54]: print(new_list)
['a', 'b', 'c']
```

3. copy 与 2 次赋值的区别：

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

```
b = a
```

◆ 二次赋值的变量与原始变量享有相同内存空间

◆ copy函数创建的新列表与原始列表不是一个内存空间，不同享数据变更

- copy 是浅拷贝

4. 浅拷贝：

◆ 通俗的说，我们有一个列表a，列表里的元素还是列表，当我们拷贝出新列表b后，无论是a还是b的内部的列表中的数据发生了变化后，相互之间都会受到影响，- 浅拷贝

```
a = [[1, 2, 3], [5, 6, 7]]  
b = a.copy()  
  
b  
[[1, 2, 3], [5, 6, 7]]  
  
b[0].append(10)  
  
b  
[[1, 2, 3, 10], [5, 6, 7]]  
  
a  
[[1, 2, 3, 10], [5, 6, 7]]
```

5. 深拷：import copy

◆ 不仅对第一层数据进行了copy，对深层的数据也进行copy，原始变量和新变量完全不共享数据 - 深拷贝

```
a = [[1,2,3], [4,5,6]]  
b = copy.deepcopy(a)  
  
b  
[[1, 2, 3], [4, 5, 6]]  
  
b[0].append(10)  
  
b  
[[1, 2, 3, 10], [4, 5, 6]]  
  
a  
[[1, 2, 3], [4, 5, 6]]
```

列表的 extend 函数

1. 功能：

将其他列表或元组中的元素倒入到当前列表中

2. 用法：

```
list.extend(iterable) ->
```

参数：

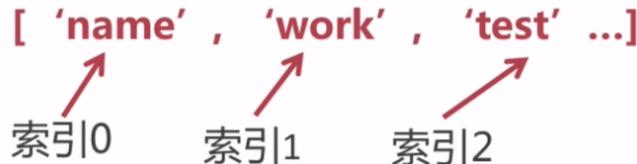
iterable 代表列表或元组，该函数无返回值

```
In [55]: students = ['deweい', 'xiaomu', 'xiaogang']
In [56]: new_students = ('xiaowang', 'xiaohong')
In [57]: students.extend(new_students)
In [58]: students
Out[58]: ['deweい', 'xiaomu', 'xiaogang', 'xiaowang', 'xiaohong']
```

索引/切片

索引与切片之列表

1. 什么是索引
 - 字符串, 列表和元组
 - 从最左边记录的位置就是索引
 - 索引用数字表示, 起始从 0 开始



字符串, 列表 (元组) 的最大索引是他们的长度 - 1

```
In [59]: I = ['name']      In [61]: I[1]
_____
In [60]: I[0]           IndexError
Out[60]: 'name'          <ipython-input-61-86e9f237d85c> in
                           ----> 1 I[1]

IndexError: list index out of range
```

2. 什么是切片

- 索引用来对单个元素进行访问, 切片则对一定范围内的元素进行访问
- 切片通过冒号在中括号内把相隔的两个索引查找出来

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers[3: 8])
>> [4, 5, 6, 7, 8]
```

- 切片规则为 : 左含, 右不含

```
print('获取列表完整数据: ', numbers[:])
print('另一种获取完整列表的方法: ', numbers[0:])
print('第三种获取列表的方法: ', numbers[0:-1])
print('列表的反序: ', numbers[::-1])
print('列表的反项获取', numbers[-3:-1])
print('步长获取切片:', numbers[0: 8: 2])
print('切片生成空列表', numbers[0: 0])
```

ps : 步长 -- 跳跃获取

- range 函数

```
>>> range(1, 5) #代表从1到5(不包含5)
[1, 2, 3, 4]
>>> range(1, 5, 2) #代表从1到5, 间隔2(不包含5)
[1, 3]
>>> range(5) #代表从0到5(不包含5)
[0, 1, 2, 3, 4]
```

3. 列表的索引, 获取与修改

- ◆ `list[index] = new_item`
- ◆ 数据的修改只能在存在的索引范围内
- ◆ 列表无法通过添加新的索引的方式赋值
- ◆ `list.index(item)`

```
In [65]: names = ['dewei', 'xiaoman']

In [66]: names.index('dewei')
Out[66]: 0

In [67]: names.index('xiaomu')
-----
ValueError                                     Traceback (most recent call last)
<ipython-input-67-8d084ff66f96> in <module>
      1 names.index('xiaomu')

ValueError: 'xiaomu' is not in list
```

• 索引和切片 元素修改

```
numbers[3] = 'code'
print(numbers)
numbers[2: 5] = ['a', 'b', 'c']
print(numbers)
```

4. 通过 `pop` 删除索引

用法 :

```
list.pop(index)
```

参数 :

`index`: 删除列表的第几个索引
 -> 函数会删除该索引的元素并返回
 -> 如果传入的`index`索引不存在则报错

```
In [75]: names = ['dewei', 'xiaomu']

In [76]: pop_item = names.pop(0)

In [77]: print('pop item:', pop_item, 'names:', names)
pop item: dewei names: ['xiaomu']
```

5. 通过 `del` 删除索引

```
del list[index]
➤ 直接删除 无返回值
➤ 如果index ( 索引 ) 不存在则报错
```

```
In [78]: names = ['dewei', 'xiaomu']

In [79]: del names[1]

In [80]: names
Out[80]: ['dewei']
```

6. 索引切片在元组中的特殊性
 - 可以和列表一样获取索引与切片索引
 - 元组函数 `index` 和列表用法完全一致
 - 无法通过索引修改和删除元素

索引与切片之字符串

1. 字符串的索引，获取
 - 索引规则与列表相同

```
name = 'dewei'  
name[0] -> d  
name[:2] -> de
```

- 切片和索引的获取与列表相同
 - 无法通过索引修改和删除
 - 字符串不可修改
2. 字符串的 `find` 与 `index` 函数
 - 功能：获取元素的索引位置
 - 用法：

```
string.index(item) -> item: 查询个数的元素, 返回索引位置  
string.find(item) -> item: 查询个数的元素, 返回索引位置
```

```
In [85]: info = 'my name is dewei'
```

```
In [86]: info.index('dewei')  
Out[86]: 11
```

```
In [87]: info.find('dewei')  
Out[87]: 11
```

- 区别：
 - ◆如果`find`找不到元素，会返回-1

◆如果`index` 找不到元素，会导致程序报错

字典添加修改数据的方法

1. []处理法
 - 字典没有索引
 - 添加或修改，根据 `key` 是否存在决定

```
In [1]: d = {'name': 'deweい'}
```

```
In [2]: d['name'] = 'xiaomu'
```

```
In [3]: print(d)  
{'name': 'xiaomu'}
```

2. 字典的内置函数 update

- 添加新的字典，如果新字典中有和原字典相同的 key，则该 key 的 value 会被新字典的 value 覆盖
- 用法：

`dict.update(new_dict)` --该函数无返回值

参数：

`new_dict: 新的字典`

```
In [4]: default_dict = {}
```

```
In [5]: new_dict = {'name': 'deweい'}
```

```
In [6]: default_dict.update(new_dict)
```

```
In [7]: default_dict
```

```
Out[7]: {'name': 'deweい'}
```

3. 字典的内置函数 setdefault

- 获取某个 key 的 value，如果 key 不存在于字典中，将会添加 key 并将 value 设为默认值
- 用法：

`dict.setdefault(key, value)`

参数：

`key 需要获取的key`

`value 如果key不存在，对应这个key存入字典的默认值`

```
In [8]: default_dict = {}
```

```
In [9]: value = default_dict.setdefault('name', '小慕')
```

```
In [10]: print('dict:', default_dict, 'value:', value)  
dict: {'name': '小慕'} value: 小慕
```

字典的 values 函数 / keys 函数（用法和 values 类似？）

1. 获取当前字典中所有键值对中的值 (value) / 键值 (key)

- 用法：

`dict.values()` -> 无需传参，返回一个value集合的伪列表

```
In [15]: my_dict = {'name': 'deweい', 'age': 33}
```

```
In [16]: my_dict.values()
```

```
Out[16]: dict_values(['deweい', 33])
```

```
print('{} | {} | {} | {}'.format(keys[0], keys[1], keys[2], keys[3]))
```

```
print('{} | {} | {} | {}'.format(values[0], values[1], values[2], values[3]))
```

字典 key 的获取

1. [] 的获取方法

- 用法：

```
In [17]: my_dict = {'name': 'dewei', 'age': 33}
```

```
In [18]: name = my_dict['name']
```

```
In [19]: print(name)  
dewei
```

- 字典+中括号内传 key，不进行赋值操作即为获取

2. 字典内置函数 get 获取方法

- 功能：获取当前字典中指定 key 的 value
- 用法：

```
dict.get(key, default=None)
```

参数：

- key：需要获取value的key
- default：key不存在则返回此默认值，默认是None，
我们也可以自定义

```
In [20]: my_dict = {'name': 'dewei', 'age': 33}
```

- In [21]: name = my_dict.get('name')

```
In [22]: print(name)  
dewei
```

3. []和 get 的区别

- []获取的 key 不存在，则直接报错
- get 获取的 key 不存在，则返回默认值
- 优先使用 get 函数

字典函数 copy

1. 功能：将当前字典复制一个新的字典

- 用法：

dict.copy() -> 该函数无参数，返回一个一模一样的内存地址不同的字典

```
In [33]: old_dict = {'name': 'dewei', 'age': 33}
```

```
In [34]: new_dict = old_dict.copy()
```

```
In [35]: id(new_dict) != id(old_dict)  
Out[35]: True
```

字典成员运算符

1. in 和 not in 在字典中的用法

```
In [39]: test_dict = {'name': 'xiaomu'}
In [40]: 'name' in test_dict
Out[40]: True

In [41]: 'name' not in test_dict
Out[41]: False
```

2. get

```
In [42]: test_dict = {'name': 'xiaomu'}

In [43]: bool(test_dict.get('name'))
Out[43]: True
```

字典中的末尾删除函数 —— popitem

1. 功能：

删除当前字典里末尾一组键值对并将其返回

2. 用法：

```
dict.popitem() -- 无需传参
>> 返回被删除的键值对，用元组包裹0索引是key，1索引是value

In [44]: my_dict = {'name': 'deweい', 'age': 33}

In [45]: my_dict.popitem()
Out[45]: ('age', 33)
```

3. 注意事项：

如果字典为 0，则直接报错

所有数据类型与布尔值的关系

1. 字符串，数字，列表，元组，字典，空类型与布尔值的关系总结：

- 每一种数据类型，自身的值都有表示 True 和 False
- not 对一切结果取反

数据类型	为True	为False
int	非0	0
float	非0.0	0.0
str	len(str) != 0	len(str) == 0 即 ''
list	len(list) != 0	len(list) == 0 即 []
tuple	len(tuple) != 0	len(tuple) == 0 即 ()
dict	len(dict) != 0	len(dict) == 0 即 {}
None	not None	None

集合

集合

1. 概念：

- 和字典长得很像，但集合里面的数据并不是成对的，例如 {1, 2, 3}
- 集合里面的数据是唯一且无序的

2. 创建方式

- set_1 = {1, 2, 3}
- 函数 set ()

集合的增删改

1. 集合的 add 函数

- 功能：用于集合中添加一个元素，若果集合中已存该元素则该函数不执行
- 用法：

用法：

set.add(item)

In [10]: a_set = set()

参数：

In [11]: a_set.add('dewei')

item：要添加到集合中的元素

In [12]: a_set

返回值：

Out[12]: {'dewei'}

无返回值

imoc

2. 集合的 update 函数

- 功能：加入一个新的集合（或列表，元组，字符串），如新集合内的元素在原集合中存在则无视
- 用法：

用法：

set.update(iterable)

In [13]: a_set = set()

参数：

iterable：集合，列表元组字符串

In [14]: a_set.update([3,4,5])

返回值：

In [15]: a_set

无返回值，直接作用于原集合

Out[15]: {3, 4, 5}

3. 集合的 remove 函数

- 功能：将集合中的某个元素删除，如元素不存在将会报错
- 用法：

用法：

set.remove(item) # 注意是元素不是索引

In [19]: a_set = {1,2,3}

参数：

item：当前集合中的一个元素

In [20]: a_set.remove(3)

返回值：

无返回值，直接作用于原集合

In [21]: a_set

Out[21]: {1, 2}

4. 集合的 clear 函数

- 功能：清空当前集合中的所有元素
- 用法：

用法：

set.clear()

In [22]: a_set = {1,2,3}

参数：

无

In [23]: a_set.clear()

返回值：

In [24]: a_set

无返回值，直接作用于原集合

Out[24]: set()

5. 用 del 删除集合

举例：

a_set = {1 , 2 , 3}

del a_set

print(a_set) # 报错

ps： 集合没有索引， 所以只能删除 set 自身

6. 重要说明：

◆ 集合无法通过索引获取元素

◆ 集合无获取元素的任何方法

◆ 集合只是用来处理列表或元组的一种临时类型，他不适合存储与传输

集合的交集 —— intersection 函数

1. 什么是交集：

a, b 两个集合分别拥有的相同的元素集，称为 a 和 b 的交集

2. 功能：

返回两个或更多集合中都包含的元素即交集

3. 用法：

a_set.intersection(b_set...)

参数：

b_set...： 与当前集合对比的1
或多个集合

In [32]: a_set = {'name', 'xiaomu', 'xiaoming'}

In [33]: b_set = {'xiaoming', 'xiaogang', 'xiao'hong'}

In [34]: a_inter = a_set.intersection(b_set)

In [35]: a_inter

Out[35]: {'xiaoming'}

返回值：

返回原始集合与对比集合的交集

4. 集合也可以重新变回列表：利用 list('set')函数

集合的并集 —— union 函数

1. 功能：

返回多个集合的并集，即包含了所有集合的元素，重复的元素只会出现一次

2. 用法：

用法：

`a_set.union(b_set...)`

参数：

`b_set...`：与当前集合对比
的1或多个集合

返回值：

返回原始集合与对比集合的并集

In [36]: `a_set = {'name', 'xiaomu', 'xiaoming'}`



In [37]: `b_set = {'xiaoming', 'xiaogang', 'xiaohong'}`



In [38]: `un = a_set.union(b_set)`

In [39]: `un`

Out[39]: `{'name', 'xiaogang', 'xiaohong', 'xiaoming', 'xiaomu'}`

集合的 `isdisjoint` 函数

1. 功能：

判断两个集合是否包含相同的元素，如果没有返回 True，则返回 False

2. 用法：

`a_set.isdisjoint(b_set)`

参数：

`b_set`：与当前集合用来判断的集合

返回值：

返回一个布尔值 True 或 False

In [57]: `a_set = {'name', 'xiaomu', 'xiaoming'}`

In [58]: `b_set = {'xiaoming', 'xiaogang', 'xiaohong'}`

in00c

In [59]: `result = a_set.isdisjoint(b_set)`

In [60]: `result`

Out[60]: `False`

字符串与数字间的转换

1. 什么？为什么？

- 将自身数据类型变新的数据类型，并拥有新的数据类型的所有功能的过程即为类型转换
- 例如 `a = '1'` # 无法做数字操作
- 为方便更好的帮助处理业务，将类型变为更适合业务场景的类型

2. 字符串与数字之间的转换的要求

- `str — number`：数字组成的字符串
- `number — str`：无要求

3. 字符串与数字之间的转换函数

原始类型	目标类型	函数	举例
整型	字符串	<code>str</code>	<code>new_str = str(123456)</code>
浮点型	字符串	<code>str</code>	<code>new_str = str(3.14)</code>
字符串	整型	<code>int</code>	<code>new_int = int(' 12')</code>
字符串	浮点型	<code>float</code>	<code>new_float = float('1.2')</code>

```
In [63]: int_str = '102983475'  
In [64]: new_int = int(int_str)  
In [65]: new_int  
Out[65]: 102983475
```

字符串与 bytes 的转换

1. 什么是 bytes
 - 二进制的数据流
 - 一种特殊的字符串
 - 字符串前 +b 标记

```
In [4]: bt = b'my name is dawei'
```

```
In [5]: type(bt)  
Out[5]: bytes
```

- 只支持英文
- 2. 字符串转 bytes 的函数 —— encode

用法：

```
string.encode(encoding='utf-8', errors='strict')
```

参数：

encoding: 转换成的编码格式，如ascii , gbk , 默认 utf-8

errors: 出错时的处理方法，默认strict ,

直接抛错误，也可以选择 ignore忽略错误

返回值：

返回一个比特 (bytes) 类型

```
In [11]: str_data = 'my name is dawei'
```

```
In [12]: byte_data = str_data.encode('utf-8')
```

```
In [13]: byte_data  
Out[13]: b'my name is dawei'
```

3. bytes 转 字符串的函数 —— decode

```
bytes.decode(encoding='utf-8', errors='strict')
```

参数：

encoding: 转换成的编码格式，如ascii , gbk , 默认 utf-8

errors: 出错时的处理方法，默认strict , 直接抛错误，也可以选择ignore忽略错误

返回值：

返回一个字符串类型

```
In [15]: byte_data = b'python is a good code'
```

```
In [16]: str_data = byte_data.decode('utf-8')
```

```
In [17]: str_data  
Out[17]: 'python is a good code'
```

元组，列表，集合间的转换

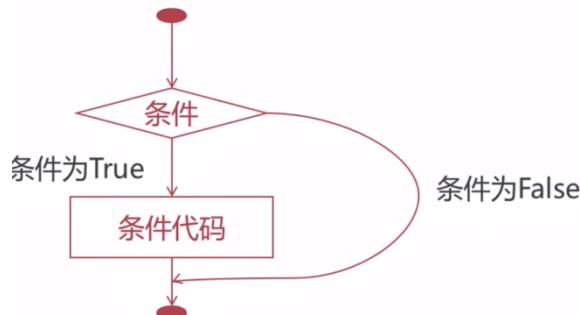
1. 利用函数

原始类型	目标类型	函数	举例
列表	集合	set	new_set = set([1,2,3,4,5])
列表	元组	tuple	new_tuple = tuple([1,2,3,4,5])
元组	集合	set	new_set = set((1,2,3,4,5))
元组	列表	list	new_list = list((1,2,3,4,5))
集合	列表	list	new_list = list({1,2,3,4,5})
集合	元组	tuple	new_tuple = tuple({1,2,3,4,5})

逻辑语句

python 的流程控制

1. 逻辑判断和逻辑语句
- 对于一件事正确与否
- 根据判断的结果做不同的事情，就是逻辑业务
- 对于条件满足的判断语句，就是条件语句



if 语句

- 用法

```
if bool_result : # 语法块  
    do          # 业务代码块 注意缩进
```

参数：

bool_result: 判断结果的真假，布尔类型
do：如果bool_result 为True时执行任意python代码

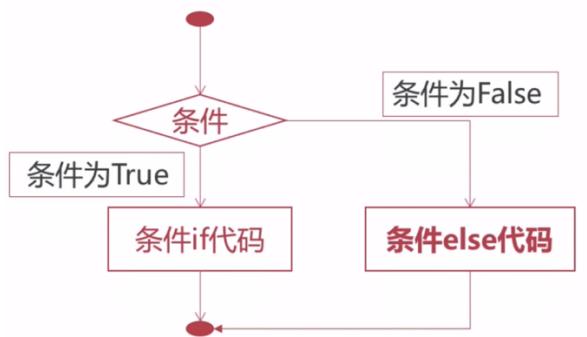
返回值：

if属于关键字，没有返回值

```
>>> dawei_status = 'hunger'  
>>> if dawei_status == 'hunger':  
...     print('Xiaomu invites Dawai to dinner')  
...  
Xiaomu invites Dawai to dinner
```

else 语句

1. else 就是对于 if 条件不满足的时候执行另一个代码块的入口



2. 用法：

```

if bool_result :
    do
else:
    elsedo # else语法块，需缩进
        # 缩进等级与do语法块一致
参数：
    elsedo: else语句对应的python代码块
返回值：
    else属于语法，没有返回值

>>> dawei_status = 'sleep'
>>> if dawei_status == 'hunger':
...     print('xiaomu invites dawei to dinner')
... else:
...     print('dawei will sleep')
...
dawei will sleep

```

elif 语句

1. 概念：

elif (或者如果) 对于命题的非第一次的多种判断，每一种判断条件对应一组业务代码

2. 用法：

```

if bool_result :
    do
elif bool_result:
    elifdo # 当前elif语句对应语法块
elif bool_result :
    elifdo # 缩进等级与do语法块一致
else:
    elsedo
参数：
    elifdo: 当前elif语句对应的python代码
返回值：
    elif属于语法，没有返回值

dawei_status = 'hunger'
if dawei_status == 'hunger':
    print('Xiaomu invites Dawai to dinner')
elif dawei_status == 'thirsty':
    print('xiaomu give dawei some drink')
elif dawei_status == 'sleepy':
    print('dawei want to sleep')
else:
    print('dawei status is good')

```

3. 条件语句的说明：

- 条件语句中满足一个条件后，将退出当前条件语句
- 每个条件语句中仅有且必须有一个 if 语句，可以有 0 个或多个 elif 语句，可以有 0 或 1 个 else 语句
- 每个条件语句 if 必须是第一个条件语句

循环与 for 循环

1. 概念：

- 周而复始地运动或变化

- python 中叫遍历

2. 功能：

通过 for 关键字将列表，元组，字符串，字典中的每个原按照序列顺序进行遍历（循环）

3. 用法：

```
for item in iterable : # for 循环语法块
    print(item)          # 每次循环对应的代码块
                        # 代码块需要缩进
```

参数：

 iterable: 可循环的数据类型 如列表 元组 字符串 字典
 item : iterable中的每一个元素（成员）

返回值：

 for 循环是语句，没有返回值，但在特定情况下有返回值

```
In [1]: l = ['deweい', 'xiaomu', 'xiaoman', 'xiaoming']

In [2]: for item in l:
...:     print(item)
...:
deweい
xiaomu
xiaoman
xiaoming
```

嵌套 for 循环

1. 概念：

for 循环中的 for 循环

```
In [17]: a = [1, 2, 3]

In [18]: b = [4, 5, 6]

In [19]: for i in a:
...:     for j in b:
...:         print(i + j, ' ', end=' ')
...:
...:
5 6 7 6 7 8 7 8 9
```

imooc

while 循环

1. 概念：

- 以一定条件为基础的循环，条件满足则无限循环，条件不满足退出循环
- 不依赖可迭代的数据类型，而 for 循环依赖

2. 用法：

```
while bool_result :
    do
```

参数：

 bool_result: 布尔类型，此处与if语法完全一致
 do : while循环体的代码块 # 需要缩进

返回值：

 while 循环是语句，没有返回值

```
In [8]: count = 1

In [9]: while count < 5:
...:     print(count, end=' ')
...:     count += 1
...:
1234
```

循环的继续与退出 continue and break

1. continue

- 功能：循环遇到 continue 将停止本次数据循环，进入下一次循环
- 用法：

```
while bool :
    continue
for item in iterable:
    continue
    print(item)
```

参数：
continue属于语法，不需要加（）即可执行
无参数
返回值：
continue是语法，没有返回值

举例：
count = 1
while count < 5:
 print(count)
 continue
 count += 1
>>1, 1 , 1 , 1 , 1

2. break :

- 功能：

使循环正常停止循环（遍历），这时如果循环配合了 else 语句，else 语句将不执行

- 用法：

```
while bool :
    break
for item in iterable:
    print(item)
    break
```

参数：
break属于语法，不需要加（）即可执行
无参数
返回值：
break是语法，没有返回值

举例：
count = 1
while count < 5:
 print(count)
 count += 1
 break
>>1

3.

- ◆continue 与 break 通常伴随着循环语句中的条件语句，满足某些条件可以继续执行，不满足某些条件提前结束循环
- ◆在while 循环中， break 语句优先于 while逻辑体的判断

4. 例子：

```
users = [  
    {'username': 'deweii', 'age': 33, 'top': 174, 'sex': '男'},  
    {'username': '小暮', 'age': 10, 'top': 175, 'sex': '男'},  
    {'username': 'xiaoyun', 'age': 18, 'top': 165, 'sex': '女'},  
    {'username': 'xiaogao', 'age': 18, 'top': 188, 'sex': '男'}  
]  
  
man = []  
  
for user in users:  
    if user.get('sex') == '女':  
        continue  
  
    man.append(user)  
print('%s 加入了帮忙的行列' % user.get('username'))
```

函数

函数的定义与使用

1. 函数的定义：
 - 将一件事情的步骤封装在一起并得到最终结果
 - 函数名代表了这个函数要做的事情
 - 函数体是实现函数功能的流程
 - 方法或功能
2. 函数的分类：
 - 内置函数： print, id, int, str, max, min, range. etc
 - 自定义函数
3. 函数的创建方法：
 - 通过关键字 def 的功能：
实现 python 中函数的创建
 - 定义过程：

```
def name(args...):  
    todo something..  
    返回值
```

- 例子：

```
In [1]: def say_hello():  
...:     print('hello xiaomu')  
...:  
  
In [2]: say_hello()  # 函数名+小括号执行函数  
hello xiaomu
```

4. 函数的返回 return：
 - 将函数结果返回的关键字
 - return 只能在函数体内使用
 - return 支持返回所有的 python 类型
 - 有返回值的函数可以直接赋值给一个变量
 - 例子：

```
def add(a, b):  
    c = a + b  
    return c  
result = add(a=1, b=1) # 参数按顺序传递  
print(result)  
>> 2
```

```

def capitalize(data):
    index = 0
    temp = ''

    for item in data:
        if index == 0:
            temp = item.upper()
        else:
            temp += item
    index += 1
return temp

```

函数的传参

1. 必传参数

- 函数中定义的参数没有默认值，在调用函数时如果不传入则报错
- 在定义函数的时候，参数后边没有等号与默认值
- 例子：

```

In [3]: def add(a, b):
...:     return a + b
...:

In [4]: result = add()
          Traceback (most recent call last)
<ipython-input-4-9a8e1c8be307> in <module>
      1 result = add()

TypeError: add() missing 2 required positional arguments: 'a' and 'b'

In [5]: result = add(1, 2)
          Out[5]: 3

```

2. 默认参数

- 在定义函数的时候，定义的参数含有默认值，通过赋值语句给他是一个默认的值
- 例子：

```

In [7]: def add(a, b=1):
...:     return a + b
...:

In [8]: result = add(1)
          def add(a, b=1)
          参数名，赋值等号，默认值
          In [9]: result
          Out[9]: 2

```

3. 不确定参数

- 没有固定的参数名和数量
- 例子：

```

def add(*args, **kwargs):
    ...
    add(1, 2, 3, name='dewei', age=33)

```

对应*args 对应**kwargs

◆ *args 代表：将无参数的值合并成元组

◆ **kwargs 代表将有参数与默认值的赋值语句合并成字典

4. 参数规则

```

def add(a, b=1, *args, **kwargs)
    ↑   ↑   ↑   ↑
    1   2   3   4

```

- ◆ 参数的定义从左到右依次是 必传参数，默认参数，可变元组参数？，可变字典参数
- ◆ 函数的参数传递非常灵活
- ◆ 必传参数与默认参数的传参多样化

函数的参数类型定义

1. 参数定义类型的方法：

```

参数名+冒号+类型函数      参数名+冒号+类型
                           ↓                         ↓
def person(name:str, age:int=33):
    print(name, age)

```

- ▶ 函数定义在python3.7之后可用
- ▶ 函数不会对参数类型进行验证

局部变量和全局变量

1. 全局变量

```

# coding:utf-8

name = 'dewei'          在python脚本最上层代码块的变量
def test():
    print(name)         全局变量可以在函数内被读取使用

```

2. 局部变量

```

# coding:utf-8

def test():
    name = 'dewei'    在函数体内定义的变量
    print(name)

print(name)              X 局部变量无法在自身函数以外使用

```

3. global

- 功能：

将全局变量可以在函数体内进行修改

- 用法：

```

# coding:utf-8

name = 'deweい'          定义一个全局变量

def test():
    global name
    name = '小慕'        定义函数
                        global + 全局变量名
                        函数体内给全局变量重新赋值

print(name)              小慕

```

- 工作中，不建议使用 global 对全局变量进行修改

递归函数

- 概念：
一个函数不停的将自己反复执行
- 递归的定义方法：

```

# coding:utf-8

def test(a):
    print(a)
    return test(a)      通过返回值 直接执行自身函数

```

- 例子：

```

def test():
    global count
    count += 1

    if count != 5:
        print('count条件不满足， 我要重新执行我自己！当前count是%s' % count)
        return test()
    else:
        print('count is %s' % count)

test()

```

```

count条件不满足， 我要重新执行我自己！当前count是1
count条件不满足， 我要重新执行我自己！当前count是2
count条件不满足， 我要重新执行我自己！当前count是3
count条件不满足， 我要重新执行我自己！当前count是4
count is 5

```

- 说明：
 - 内存溢出
 - 避免滥用递归

匿名函数 lambda

- 功能：
 - 定义一个轻量化的函数
 - 即用即删除，很适合需要完成一项功能，但是此功能只在此一处使用
- 用法：

无参数

```
f = lambda : value
```

```
f()
```

有参数

```
f = lambda x,y: x * y
```

```
f(3, 4)
```

3. 简单例子：

```
# coding:utf-8

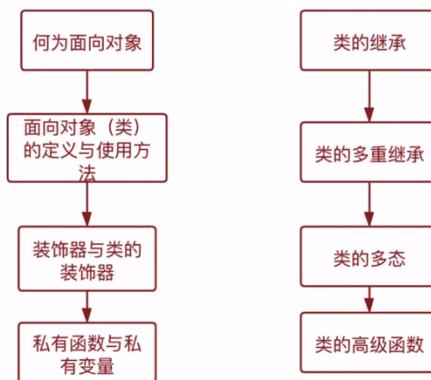
f = lambda: print(1)
f()

f1 = lambda x, y: x + y
print(f1(1, 2))
```

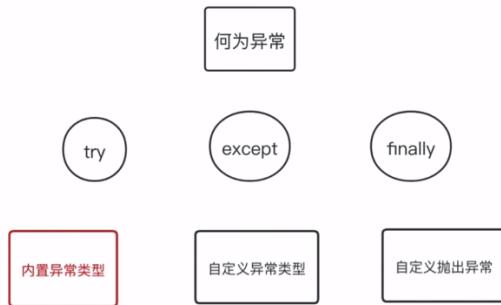
面对对象与异常

python 面向对象与异常

1. 面向对象：



2. 异常：



Bug 的由来及分类

1. 常见类型：

- 漏写末尾的冒号，if 语句，循环语句，else，etc.
- 缩进错误
- 英文符号写成中文，例如冒号，引号等
- 字符串拼接的时候，将字符串和数字拼在一起
- 没有定义变量
- == 比较运算符 和 = 赋值运算符的混淆

Python 的异常处理机制

1. try...except....else 结构

• try...except...else结构

- 如果try块中没有抛出异常，则执行else块，如果try中抛出异常，则执行except块

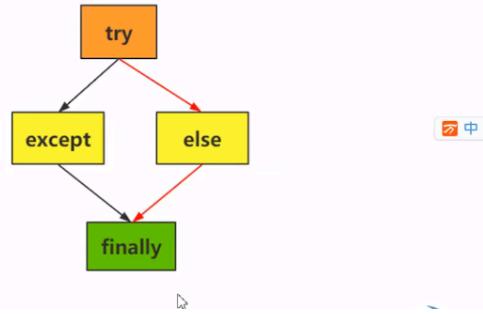
```
try:  
    n1=int(input('请输入一个整数:'))  
    n2=int(input('请输入另一个整数:'))  
    result=n1/n2  
except BaseException as e:  
    print('出错了')  
    print(e)  
else:  
    print('结果为:',result)
```

2. try...except...else...finally 结构

• try...except...else...finally结构

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

```
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('无论是否产生异常，总会被执行的代码')  
print('程序结束')
```



python 中常见的异常类型和处理机制

1. 常见：

序号	异常类型	描述
1	ZeroDivisionError	除(或取模)零 (所有数据类型)
2	IndexError	序列中没有此索引(index)
3	KeyError	映射中没有这个键
4	NameError	未声明/初始化对象 (没有属性)
5	SyntaxError	Python 语法错误
6	ValueError	传入无效的参数

2. traceback 模块：

- 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/dem01.py", line 7, in <module>  
    num=10/0  
ZeroDivisionError: division by zero
```

python 类和模块 (Class, Module)

1. python 是也是一个面对对象编程的语言，面对对象编程是一种设计思想，意味着我们把对象作为程序的基本单元，而每个对象包含自己的属性和方法
2. 在 python 中，使用类 (class) 来自定义对象
3. 每个类都有自己的属性和方法，例如一个人的身高，体重和年龄，都是属性；而吃饭，说话和洗澡，都是方法。（要注意：在 class 外部定语的可执行函数叫 function，类内部的函数叫做方法 method）
4. 类的定义以 class 为开头，类名的首字母要大写，冒号之后换行缩进紧跟着属性和方法的定义、属性无非就是一个变量的定义，而方法的定义和函数的定义是一样的，也是以 def 开头
5. 例子：

The screenshot shows two code snippets in a Python code editor.

Top Snippet:

```
# class
class Person:
    # attribute fields
    name = 'William'
    age = 45
    # method
    def greet(self):
        print("Hi, my name is " + self.name)
# Create an Object
p1 = Person()
# Call the method
p1.greet()
```

Output: Hi, my name is William

Bottom Snippet:

```
# Modify Object Properties
p1.age = 40
print(p1.age)

# Delete Object Properties
del p1.age

# Delete Objects
del p1
print(p1.age)
```

Output: 40

Traceback:

```
NameError                                 Traceback (most recent call last)
<ipython-input-3-305fcbbb238> in <module>
      7 # Delete Objects
      8 del p1
----> 9 print(p1.age)

NameError: name 'p1' is not defined
```

Buttons:

- SEARCH STACK OVERFLOW

类的参数 self

1. 要点：

- ◆ self 是类函数中的必传参数，且必须放在第一个参数位置
- ◆ self 是一个对象，他代表实例化的变量自身
- ◆ self 可以直接通过点来定义一个类变量 self.name = 'dewe'
- ◆ self 中的变量与含有self参数的函数可以在类中的任何一个函数内随意调用
- ◆ 非函数中定义的变量在定义的时候不用self

2. 例子：

```

class Person(object):
    name = None
    age = None

    def run(self):
        print(f'{self.name} 在奔跑')

    def jump(self):
        print(f'{self.name} 在跳跃')

xiaomu = Person()
xiaomu.jump()

```

构造函数的创建

1. 概念：

类中的一种默认函数，用来将类实例化的同时，将参数传入类中

2. 创建：

```

def __init__(self, a, b):
    self.a = a
    self.b = b

```

3. 例子：

A.

```

In [18]: class Test(object):
...:     def __init__(self, a):
...:         self.a = a
...:     def run(self):
...:         print(self.a)
...:

```

```
In [19]: t = Test(1)
```

```
In [20]: t.run()
1
```

(某种程度上，相当于给变量赋值)

B.

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的 `__init__` 方法，在创建实例的时候，就把 `name`，`score` 等属性绑上去：

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score
```

⚠ 注意：特殊方法“`__init__`”前后分别有两个下划线！！！

注意到 `__init__` 方法的第一个参数永远是 `self`，表示创建的实例本身，因此，在 `__init__` 方法内部，就可以把各种属性绑定到 `self`，因为 `self` 就指向创建的实例本身。

有了 `__init__` 方法，在创建实例的时候，就不能传入空的参数了，必须传入与 `__init__` 方法匹配的参数，但 `self` 不需要传，Python 解释器自己会把实例变量传进去：

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.name
'Bart Simpson'
>>> bart.score
59
```

C.

```
# -*- coding: utf-8 -*-
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'

lisa = Student('Lisa', 99)
bart = Student('Bart', 59)
print(lisa.name, lisa.get_grade())
print(bart.name, bart.get_grade())
```

▶ Run

Lisa A
Bart C

私有函数与私有变量

1. 概念：

- 无法被实例化后的对象调用的类中的函数与变量
- 类内部可以调用私有函数与变量
- 只希望类内部业务调用使用，不希望被使用者调用

2. 定义方法：

- 在变量或函数前添加__（两个下划线），变量或函数名后边无需添加
- 例子

```
class Person(object):
    def __init__(self, name):
        self.name = name
        self.__age = 33
    def dump(self):
        print(self.name, self.__age)
    def __cry(self):
        return 'I want cry'
```

Python 中的封装

1. 概念：

将不对外的私有属性或方法通过可对外使用的函数而使用（类中定义私有的，只有类内部使用，外部无法访问）

这样做的主要原因：保护隐私，明确区分内外

2. 例子：

```
class Parent(object):
    def __hello(self, data):
        print('hello %s' % data)

    def helloworld(self):
        self.__hello('world')

if __name__ == '__main__':
    p = Parent()
    p.helloworld()      hello world
```

装饰器

1. 概念：

- 也是一种函数
- 可以接受函数作为参数
- 可以返回函数
- 接收一个函数，内部对其进行处理，然后返回一个新函数，动态的增强函数功能
- 将 c 函数在 a 函数中执行，在 a 函数中可以选择执行或不执行 c 函数，也可以对 c 函数的结果进行二次加工处理
-

```
def a():
    def b():
        print('hello')
    b()
a()
b()
```

2. 定义方法：

```
def out(func_args):    外围函数
    def inter(*args, **kwargs):    内嵌函数
        return func_args(*args, **kwargs)
    return inter    外围函数返回内嵌函数
```

3. 用法：

- 将被调用的函数直接作为参数传入装饰器的外围函数括弧

```
def a(func):
    def b(*args, **kwargs):
        return func(*args, **kwargs)
    return b

def c(name):
    print(name)
a(c('dewei')) # dewei
```

- 将装饰器与被调用函数绑定在一起

@符号加装饰器函数放在被调用函数的上一行，被调用的函数正常定义，只需要直接调用被执行函数即可

```
@a
def c(name):
    print(name)

c('dewei')
```

4. 例子：

```
def check_str(func):
    def inner(*args, **kwargs):
        result = func(*args, **kwargs)
        if result == 'ok':
            return 'result is %s' % result
        else:
            return 'result is failed:%s' % result
    return inner

@test
def test(data):
    return data
```

thon_object ~/Pychar
object_decorator.py
object_init.py
object_private.py
ternal Libraries
atches and Consoles

object_decorator
/Users/zhangdewei/.virtualenvs/python_object/bin/python /Users/z
result is failed:no

hon_object ~/Pychar
object_decorator.py
object_init.py
object_private.py
ernal Libraries
atches and Consoles

object_decorator
/users/zhangdewei/.virtualenvs/python_object/bin/python /users/zhang
func: <function test at 0x10960c700>
args: () {'data': 'no'}
result is failed:no
args: () {'data': 'ok'}
result is ok