

因为最近python大火，并且python好像可以写自动化程序，所以去了解一下python的相关知识

python官方文档[Python 教程 — Python 3.10.6 文档](#)

也总结了关于python语法与java的区别

python简介

Python 是一门易于学习、功能强大的编程语言。它提供了高效的高级数据结构，还能简单有效地面向对象编程。Python 优雅的语法和动态类型以及解释型语言的本质，使它成为多数平台上写脚本和快速开发应用的理想语言。

第一章

变量

```
python定义变量 # 定义一个变量，用来记录钱包余额  
money = 50
```

注释

- 单行注释：以 **#开头，#右边** 的所有文字当作说明，而不是真正要执行的程序，起**辅助说明作用**

```
1 # 我是单行注释  
2 print("Hello World")
```

注意，#号和注释内容一般建议以一个空格隔开

- 多行注释：以 **一对三个双引号** 引起来 (**"""注释内容"""**) 来解释说明一段代码的作用使用方法

```
1 """  
2     我是多行注释  
3     诗名：悯农  
4     作者：李绅  
5 """  
6     print("锄禾日当午")  
7     print("汗滴禾下土")  
8     print("谁知盘中餐")  
9     print("粒粒皆辛苦")
```

需要注意的是 # 文字(中间)会有一个空格 (这是一种规范)

print函数

打印函数

```
print("买了冰淇淋花费10元，还剩余：" , 40, "元")
```

买了冰淇淋花费10元，还剩余：40 元

中间可以加逗号进行拼接

```
>>> str1='Python'  
>>> str2=' good'  
>>> str3=str1+str2    #字符串连接  
>>> print(str3)  
Python good  
>>> print(str1*2)    #输出字符串两次  
PythonPython  
>>> print(2*str1)  
PythonPython
```

数据类型

1. 在print语句中，直接输出类型信息：

```
print(type("黑马程序员"))  
print(type(666))  
print(type(11.345))  
  
test <  
D:\dev\Python\Python3.10  
<class 'str'>  
<class 'int'>  
<class 'float'>
```

str是string的缩写

2. 用变量存储type()的结果（返回值）：

```
1 string_type = type("黑马程序员")  
2 int_type = type(666)  
3 float_type = type(11.345)  
4 print(string_type)  
5 print(int_type)  
6 print(float_type)  
  
Run: test <  
D:\dev\Python\Python3.10.4\python  
<class 'str'>  
<class 'int'>  
<class 'float'>
```

根据type()函数得到数据类型，这个数据类型可以被变量所保存（我猜想保存的是字符串）

对于变量的类型也依然可以查看

查看的都是<字面量>的类型，能查看变量中存储的数据类型吗？

那当然：可以

```
name = "黑马程序员"
name_type = type(name)
print(name_type)
```

test ×
D:\dev\Python\Python3.10
<class 'str'>

不管我们查看的是变量的类型还是数据的类型，本质上其实都是**数据**的类型，因为变量里面存储的就是数据

数据类型转换

认识三个函数

int()

float()

str()

里面可以放相应的变量或字面量，其他需要注意的和java一样

标识符

给类。变量。方法起的名字就叫标识符

起名字的限制

标识符命名中，只允许出现：

- 英文
- 中文
- 数字
- 下划线（_）

这四类元素。

而且数字不能用在开头（和java一样）

python也是大小写敏感的（和java一样）

不能用关键字（和java一样）

算数运算符

算术（数学）运算符

运算符	描述	实例
<code>+</code>	加	两个对象相加 <code>a + b</code> 输出结果 30
<code>-</code>	减	得到负数或是一个数减去另一个数 <code>a - b</code> 输出结果 -10
<code>*</code>	乘	两个数相乘或是返回一个被重复若干次的字符串 <code>a * b</code> 输出结果 200
<code>/</code>	除	<code>b / a</code> 输出结果 2
<code>//</code>	取整除	返回商的整数部分 <code>9//2</code> 输出结果 4 , <code>9.0//2.0</code> 输出结果 4.0
<code>%</code>	取余	返回除法的余数 <code>b % a</code> 输出结果 0
<code>**</code>	指数	<code>a**b</code> 为10的20次方, 输出结果 1000000000000000000000000

需要注意的是python中仍然支持`+=`等复合运算符

字符串拓展

取出字符串中的字符

Python中的字符串有两种索引方式，从左往右以**0**开始，从右往左以**-1**开始。

```
>>> print(str1[0])      # 通过索引输出字符串第一个字符  
P  
>>> print(str1[2:5])    # 输出从第三个开始到第五个的字符  
tho  
>>> print(str1[0:-1])   # 输出第一个到倒数第二个的所有字符
```

字符串的三种定义方式

- 字符串在Python中有多种定义形式:

1. 单引号定义法: `name = '黑马程序员'`

2. 双引号定义法: `name = "黑马程序员"`

3. 三引号定义法: `name = """黑马程序员"""`

三引号定义法，和多行注释的写法一样，同样支持换行操作。

使用变量接收它，它就是字符串

不使用变量接收它，就可以作为多行注释使用。

需要注意的是第三种方法的支持换行操作

语法是这样的格式

```
name = """
我是
黑马
程序员
"""


```

并且对于引号里面有引号的情况，对于java来说我们可以用转义字符\来使用，python仍然可以，并且python同样支持以下两种方法

```
# 在字符串内 包含双引号
name = '"黑马程序员"'
# 在字符串内 包含单引号
name = "'黑马程序员'"
```

打印结果为

"黑马程序员"

'黑马程序员'|

字符串的拼接

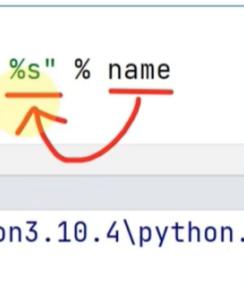
同样可以根据+进行字符串与字符串的拼接，整型浮点型不能通过+与字符串进行拼接

字符串格式化一

我们可以通过如下语法，完成字符串和变量的快速拼接。

```
name = "黑马程序员"
message = "学IT就来 %s" % name
print(message)
```

test ×
D:\dev\Python\Python3.10.4\python.
学IT就来 黑马程序员



其中的，%s

- % 表示：我要占位
- s 表示：将变量变成字符串放入占位的地方

所以，综合起来的意思就是：我先占个位置，等一会有个变量过来，我把它变成字符串放到占位的位置

那，数字类型呢？可以不可以占位？

那必须可以，我们来尝试如下代码：

```
1 class_num = 57
2 avg_salary = 16781
3 message = "Python大数据学科，北京%s期，毕业平均工资：%s" % (class_num, avg_salary)
4 print(message)
```

Run: test ×
D:\dev\Python\Python3.10.4\python.exe D:/python-learn/01_你好Python/test.py
Python大数据学科，北京57期，毕业平均工资：16781



多个变量占位
变量要用括号括起来
并按照占位的顺序填入

上面这个图片中的数字其实是整数类型，被转换成了字符串类型了而已

如果想直接用整型占位，可以用这些

格式符号	转化
%s	将内容转换成字符串，放入占位位置
%d	将内容转换成整数，放入占位位置
%f	将内容转换成浮点型，放入占位位置

字符串格式化方式二

通过语法：f"内容{变量}"的格式来快速格式化

```
name = "传智播客"
set_up_year = 2006
stock_price = 19.99
print(f"我是{name}, 我成立于: {set_up_year}, 我今天的股票价格是: {stock_price}")
```

D:\dev\Python\Python3.10.4\python.exe D:/python-learn/01_Python基础知识
我是传智播客, 我成立于: 2006, 我今天的股票价格是: 19.99 不做精度控制, 原样输出

这种方法不能做精度控制, 不能控制位数

f代表format

数据输入 (input函数)

我们前面学习过print语句（函数），可以完成将内容（字面量、变量等）输出到屏幕上。

在Python中，与之对应的还有一个input语句，用来获取键盘输入。

- 数据输出: print
- 数据输入: input

使用上也非常简单：

- 使用input()语句可以从键盘获取输入
- 使用一个变量接收（存储）input语句获取的键盘输入数据即可

```
print("请告诉我你是谁? ")
name = input()
print("Get! ! ! 你是: %s" % name)
```

n: test (1) >
D:\dev\Python\Python3.10.4\python.exe
请告诉我你是谁?
黑马程序员
Get! ! ! 你是: 黑马程序员

需要注意的是input语句里面可以填写参数（比如字符串）会给输入程序启动前给出提示的内容（参数中填写的字符串）

并且无论在input（键盘）中输入的是数字还是字符串，都把输入的内容当成字符串

第二章

布尔类型和比较运算符

布尔值

真True 值为1 (python特有)

假False 值为0 (和c一样)

运算符

运算符	描述	示例
==	判断内容是否相等，满足为True，不满足为False	如a=3,b=3，则(a == b) 为 True
!=	判断内容是否不相等，满足为True，不满足为False	如a=1,b=3，则(a != b) 为 True
>	判断运算符左侧内容是否大于右侧 满足为True，不满足为False	如a=7,b=3，则(a > b) 为 True
<	判断运算符左侧内容是否小于右侧 满足为True，不满足为False	如a=3,b=7，则(a < b) 为 True
>=	判断运算符左侧内容是否大于等于右侧 满足为True，不满足为False	如a=3,b=3，则(a >= b) 为 True
<=	判断运算符左侧内容是否小于等于右侧 满足为True，不满足为False	如a=3,b=3，则(a <= b) 为 True

if判断语句

单if

程序中的判断

if 要判断的条件：

条件成立时，要做的事情

和java的区别是条件没有括号括着，并且后面有冒号

并且python通过缩进判断代码块的归属关系!!!!

if else

程序中的判断

if 条件:

 满足条件时要做的事情1

 满足条件时要做的事情2

 满足条件时要做的事情3

 (省略)....

else:

 不满足条件时要做的事情1

 不满足条件时要做的事情2

 不满足条件时要做的事情3

 (省略)....

if elif else

和java中的else if一样

程序中的判断

if 条件1:

 条件1满足应做的事情
 条件1满足应做的事情

elif 条件2:

 条件2满足应做的事情
 条件2满足应做的事情

elif 条件N:

 条件N满足应做的事情
 条件N满足应做的事情

else:

 所有条件都不满足应做的事情
 所有条件都不满足应做的事情

总结

if中可以嵌套，但最重要的是python是根据缩进完成的配套，没有大括号这一说

第三章

while循环

程序中的循环

`while` 条件:

条件满足时，做的事情1

条件满足时，做的事情2

条件满足时，做的事情3

....(省略)....

只要条件满足
会无限循环执行

条件+:

通过缩进确定逻辑层次关系

同时while也支持嵌套哦

for循环

while与for的区别

除了while循环语句外，Python同样提供了for循环语句。

两者能完成的功能基本差不多，但仍有一些区别：

- while循环的循环条件是自定义的，**自行控制循环条件**
- for循环是一种“**轮询**”机制，是对一批内容进行“逐个处理”

for的语法

程序中的for循环

for 临时变量 in 待处理数据集:
循环满足条件时执行的代码

这个语法跟Java中的加强for一样

需要注意的是字符串也可以被循环处理 (这个是python特有的)

遍历字符串

```
# 定义字符串name  
name = "itheima"  
# for循环处理字符串  
for x in name:  
    print(x)
```

运行结果如下：

```
i  
t  
h  
e  
i  
m  
a
```

第四章

函数

函数是组织好的，可重复使用的，用来实现特定功能的代码段

为了得到一个针对特定需求，减少代码冗余，提高代码复用性

自定义函数的语法

函数的定义：

def 函数名(传入参数):
 函数体
 return 返回值

函数使用步骤：

先定义函数（写在前面）

后调用函数

注意事项：

参数不需要，可以省略

返回值不需要，可以省略（跟Java不太一样，Java需要单独写return）

函数的返回值None

演示：

```
def say_hello():
    print("Hello...")

# 使用变量接收say_hello函数的返回值
result = say_hello()
# 打印返回值
print(result)          # 结果None
# 打印返回值类型
print(type(result))   # 结果<class 'NoneType'>
```

None可以主动使用return返回，效果等同于不写return语句：

```
def say_hello():
    print("Hello...")
    return None

# 使用变量接收say_hello函数的返回值
result = say_hello()
# 打印返回值
print(result)          # 结果None
```

这个None是Python特别的返回值，一共有两种方式得到None，和Java中的null很像

并且None有很多应用的场景

None作为一个特殊的字面量，用于表示：空、无意义，其有非常多的应用场景。

- 用在函数无返回值上

- 用在if判断上

- 在if判断中，None等同于False

- 一般用于在函数中主动返回None，配合if判断做相关处理

```
def check_age(age):
    if age > 18:
        return "SUCCESS"
    return None

result = check_age(5)
if not result:
    print("未成年，不可进入")
```

返回值若为None的话为false，加上前面的not就为true了

- 用于声明无内容的变量上

- 定义变量，但暂时不需要变量有具体值，可以用None来代替

```
# 暂不赋予变量具体值
name = None
```

相当于java中的给的默认值

函数的文档注释

用法和Java的文档注释一样，不一样的是Python用多行注释代替了文档注释，并且写在了函数里面（Java是写在函数外面的）

语法规则：

```
def func(x, y):
    """
    函数说明
    :param x: 形参x的说明
    :param y: 形参y的说明
    :return: 返回值的说明
    """

    函数体
    return 返回值
```

通过多行注释的形式，对函数进行说明解释

- 内容应写在函数体之前

作用域

需要注意的是变量分为局部变量以及全局变量，如果在函数里面创建的变量叫局部变量，在函数外创建的叫全局变量，如果全局变量与局部变量有相同名字的变量，在函数内使用默认使用局部变量的值，函数内修改不会改变全局变量的值（这是跟Java不一样的）如果想修改全局变量的值，需要进行声明

声明的语法是

函数：

```
global 全局变量名称 # 声明了这个变量是全局变量，这样就一样啦
```

多返回值

语法

```
def test_return():
    return 1, 2

x, y = test_return()
print(x)      # 结果1
print(y)      # 结果2
```

按照返回值的顺序，写对应顺序的多个变量接收即可

变量之间用逗号隔开

支持不同类型的数据return

位置参数与关键字参数

```
def user_info(name, age, gender):
    print(f"姓名是:{name}, 年龄是:{age}, 性别是:{gender}")

# 位置参数 - 默认使用形式
user_info('小明', 20, '男')

# 关键字参数
user_info(name='小王', age=11, gender='女')
user_info(age=10, gender='女', name='潇潇')      # 可以不按照参数的定义顺序传参
user_info('甜甜', gender='女', age=9)
```

缺省参数（默认值）

缺省参数：缺省参数也叫默认参数，用于定义函数，为参数提供默认值，调用函数时可不传该默认参数的值（注意：所有位置参数必须出现在默认参数前，包括函数定义和调用）。

作用：当调用函数时没有传递参数，就会使用默认是用缺省参数对应的值。

```
def user_info(name, age, gender='男'):
    print(f'您的名字是{name}, 年龄是{age}, 性别是{gender}')

user_info('TOM', 20)
user_info('Rose', 18, '女')
```

注意：

函数调用时，如果为缺省参数传值则修改默认参数值，否则使用这个默认值

需要注意的是

```
# 缺省参数(默认值)
def user_info(name='小王', age, gender):
    print(f"姓名是:{name}, 年龄是:{age}, 性别是:{gender}")

user_info('小天', 13, '男')默认值必须写在最后面
```

不定长参数

位置传递

```
def user_info(*args):
    print(args)

# ('TOM',)
user_info('TOM')
# ('TOM', 18)
user_info('TOM', 18)
```

注意：

传进的**所有参数**都会被**args**变量收集，它会根据传进参数的位置合并为一个元组(tuple)，**args**是元组类型，这就是**位置传递**

关键字传递

```
def user_info(**kwargs):
    print(kwargs)

# {'name': 'TOM', 'age': 18, 'id': 110}
user_info(name='TOM', age=18, id=110)
```

注意：

参数是“**键=值**”形式的形式的情况下，所有的“**键=值**”都会被**kwargs**接受，同时会根据“**键=值**”组成**字典**.

取出数据是根据元组或者字典的规则取出的，其他操作同理

匿名函数

函数作为参数传递

```

如下代码：
def test_func(compute):
    result = compute(1, 2)
    print(result)

def compute(x, y):
    return x + y

test_func(compute)      # 结果: 3      test_func(compute)      # 结果: 2      test_func(compute)      # 结果: -1

```

函数compute，作为参数，传入了test_func函数中使用。

- test_func需要一个函数作为参数传入，这个函数需要接收2个数字进行计算，计算逻辑由这个被传入函数决定
- compute函数接收2个数字对其进行计算，compute函数作为参数，传递给了test_func函数使用
- 最终，在test_func函数内部，由传入的compute函数，完成了对数字的计算操作

所以，这是一种，**计算逻辑的传递，而非数据的传递。**

就像上述代码那样，不仅仅是相加，相减、相除、等**任何逻辑都可以自行定义并作为函数传入。**

函数本身是可以作为参数，传入另一个函数中进行使用的

将函数传入的作用在于：传入计算逻辑，而非平时传入数据

lambda匿名函数

函数的定义中

- def关键字，可以定义**带有名称**的函数
- lambda关键字，可以**定义匿名**函数（无名称）

有名称的函数，可以基于名称**重复使用**。

无名称的匿名函数，只可**临时使用一次**。

匿名函数的语法：

匿名函数定义语法：

lambda 传入参数： 函数体(一行代码)

- lambda是关键字，表示定义匿名函数
- 传入参数表示匿名函数的形式参数，如：x, y表示接收2个形式参数
- 函数体，就是函数的执行逻辑，要注意：只能写一行，无法写多行代码

lambda定义的函数主要是用于作为其他函数的参数进行传递

```

def test_func(compute):
    result = compute(1, 2)
    print(result)

```

```

test_func(lambda x, y: x + y)      # 结果: 3

```

注意事项：

匿名函数用于临时构建一个函数，只用一次的场景

匿名函数的定义中，函数体只能写一行代码，如果函数体要写多行代码，不可以用lambda匿名函数，可以使用def定义带名函数

第五章

数据容器

数据容器就是一种可以存储多个元素的Python数据类型（像Java中的数组与集合的合成体）

Python中的数据容器：

一种可以容纳多份数据的数据类型，容纳的每一份数据称之为1个元素

每一个元素，可以是任意类型的数据，如字符串、数字、布尔等。

数据容器根据特点的不同，如：

- 是否支持重复元素
- 是否可以修改
- 是否有序，等

分为5类，分别是：

列表（list）、元组（tuple）、字符串（str）、集合（set）、字典（dict）

我们将一一学习它们

list列表

基本的语法

列表的定义

基本语法：

```
1 # 字面量
2 [元素1, 元素2, 元素3, 元素4, ...]
3
4 # 定义变量
5 变量名称 = [元素1, 元素2, 元素3, 元素4, ...]
6
7 # 定义空列表
8 变量名称 = []
9 变量名称 = list()
```

列表内的每一个数据，称之为元素

- 以 [] 作为标识
- 列表内每一个元素之间用，逗号隔开

示例

```
1 my_list = ['itheima', 666, True]
2 print(my_list)
3 print(type(my_list))
```

```
['itheima', 666, True]
<class 'list'>
```

注意事项

注意：列表可以一次存储多个数据，且可以为不同的数据类型，支持嵌套

注意和Java一样，这里面也是可以嵌套的，嵌套的语法为

嵌套列表的定义

```
1 my_list = [ [1, 2, 3], [4, 5, 6] ]  
2 print(my_list)  
3 print(type(my_list))
```

```
[[1, 2, 3], [4, 5, 6]]  
<class 'list'>
```

list列表支持用下表索引取出相应的元素

如图，列表中的每一个元素，都有其位置下标索引，从前向后的方向，**从0开始，依次递增**

我们只需要按照下标索引，即可取得对应位置的元素。

```
1 # 语法： 列表[下标索引]  
2  
3 name_list = ['Tom', 'Lily', 'Rose']  
4 print(name_list[0]) # 结果： Tom  
5 print(name_list[1]) # 结果： Lily  
6 print(name_list[2]) # 结果： Rose
```

支持反向索引（python特有）

或者，可以反向索引，也就是从后向前：从-1开始，依次递减（-1、-2、-3.....）



```
1 # 语法： 列表[标号]  
2  
3 name_list = ['Tom', 'Lily', 'Rose']  
4 print(name_list[-1]) # 结果： Rose  
5 print(name_list[-2]) # 结果： Lily  
6 print(name_list[-3]) # 结果： Tom
```

list列表的方法

具体用法在[第六章-04-列表的常用操作方法](#)bilibili

编号	使用方式	作用
1	列表.append(元素)	向列表中追加一个元素
2	列表.extend(容器)	将数据容器的内容依次取出，追加到列表尾部
3	列表.insert(下标, 元素)	在指定下标处，插入指定的元素
4	del 列表[下标]	删除列表指定下标元素
5	列表.pop(下标)	删除列表指定下标元素
6	列表.remove(元素)	从前向后，删除此元素第一个匹配项
7	列表.clear()	清空列表
8	列表.count(元素)	统计此元素在列表中出现的次数
9	列表.index(元素)	查找指定元素在列表的下标 找不到报错ValueError
10	len(列表)	统计容器内有多少元素

tuple元组

思考：列表是**可以修改的**。

如果想要传递的信息，**不被篡改**，列表就不合适了。

元组同列表一样，都是可以封装多个、不同类型的元素在内。

但最大的不同点在于：

元组一旦定义完成，就不可修改

元组是不可以修改的

相当于只读不可写

元组定义方法：

元组定义：定义元组使用**小括号**，且使用**逗号**隔开各个数据，数据可以是不同的数据类型。

```
1 # 定义元组字面量
2 (元素, 元素, ..., 元素)
3 # 定义元组变量
4 变量名称 = (元素, 元素, ..., 元素)
5 # 定义空元组
6 变量名称 = ()          # 方式1
7 变量名称 = tuple()     # 方式2
```

注意是小括号，list的中括号

定义元组的注意事项：

注意事项

```
1 # 定义3个元素的元组
2 t1 = (1, 'Hello', True)
3
4 # 定义1个元素的元组
5 t2 = ('Hello',) # 注意，必须带有逗号，否则不是元组类型
```

注意：元组只有一个数据，这个数据后面要添加逗号

特别的：

元组的内容不可以修改，但是如果元组里面嵌套了list列表，list列表内的内容可以修改

```
1 # 尝试修改元组内容
2 t1 = (1, 2, ['itheima', 'itcast'])
3 t1[2][1] = 'best'
4 print(t1) # 结果: (1, 2, ['itheima', 'best'])
```

并且元组之间可以进行拼接

比如：tuple3=tuple1+tuple2

字符串

和其它容器如：列表、元组一样，字符串也可以通过下标进行访问

- 从前向后，下标从0开始
- 从后向前，下标从-1开始

```
1 # 通过下标获取特定位置字符
2 name = "itheima"
3 print(name[0]) # 结果i
4 print(name[-1]) # 结果a
```

同元组一样，字符串是一个：**无法修改**的数据容器。

所以：

- 修改指定下标的字符 (如：字符串[0] = “a”)
- 移除特定下标的字符 (如：del 字符串[0]、字符串.remove()、字符串.pop()等)
- 追加字符等 (如：字符串.append())

均无法完成。如果必须要做，只能得到一个新的字符串，旧的字符串是无法修改

同样支持正向索引以及反向索引

字符串相关的方法

编号	操作	说明
1	字符串[下标]	根据下标索引取出特定位置字符
2	字符串.index(字符串)	查找给定字符的第一个匹配项的下标
3	字符串.replace(字符串1, 字符串2)	将字符串内的全部字符串1，替换为字符串2 不会修改原字符串，而是得到一个新的
4	字符串.split(字符串)	按照给定字符串，对字符串进行分隔 不会修改原字符串，而是得到一个新的列表
5	字符串.strip() 字符串.strip(字符串)	移除首尾的空格和换行符或指定字符串
6	字符串.count(字符串)	统计字符串内某字符串的出现次数
7	len(字符串)	统计字符串的字符个数

序列的切片操作

序列是指：内容连续、有序，可使用下标索引的一类数据容器
列表、元组、字符串，均可以视为序列。



序列支持切片，即：列表、元组、字符串，均支持进行切片操作

切片：从一个序列中，取出一个子序列

语法：序列[起始下标:结束下标:步长]

表示从序列中，从指定位置开始，依次取出元素，到指定位置结束，得到一个新序列：

- 起始下标表示从何处开始，可以留空，留空视作从头开始
- 结束下标（不含）表示何处结束，可以留空，留空视作截取到结尾
- 步长表示，依次取元素的间隔
 - 步长1表示，一个个取元素
 - 步长2表示，每次跳过1个元素取
 - 步长N表示，每次跳过N-1个元素取
 - 步长为负数表示，反向取（注意，起始下标和结束下标也要反向标记）

注意，此操作**不会影响序列本身，而是会得到一个新的序列（列表、元组、字符串）**

set集合

set集合的特点是内容无序且不可重复

基本语法

基本语法：

```
1 # 定义集合字面量
2 {元素, 元素, ..., 元素}
3 # 定义集合变量
4 变量名称 = {元素, 元素, ..., 元素}
5 # 定义空集合
6 变量名称 = set()
```

和列表、元组、字符串等定义基本相同：

- 列表使用：[]
- 元组使用：()
- 字符串使用：""
- 集合使用：{}

因为set集合是无序的，所以不支持下标索引访问，但是集合和列表一样，是允许修改的
set集合的遍历

```
# 集合的遍历
# 集合不支持下标索引，不能用while循环
# 可以用for循环
set1 = {1, 2, 3, 4, 5}
for element in set1:
    print(f"集合的元素有: {element}")
```

集合常用方法

集合常用功能总结

编号	操作	说明
1	集合.add(元素)	集合内添加一个元素
2	集合.remove(元素)	移除集合内指定的元素
3	集合.pop()	从集合中随机取出一个元素
4	集合.clear()	将集合清空
5	集合1.difference(集合2)	得到一个新集合，内含2个集合的差集 原有的2个集合内容不变
6	集合1.difference_update(集合2)	在集合1中，删除集合2中存在的元素 集合1被修改，集合2不变
7	集合1.union(集合2)	得到1个新集合，内含2个集合的全部元素 原有的2个集合内容不变
8	len(集合)	得到一个整数，记录了集合的元素数量

字典

和java的map一样

是key-value结构

字典的定义语法

字典的定义，同样使用{}，不过存储的元素是一个个的：键值对，如下语法：

```
1 # 定义字典字面量
2 {key: value, key: value, ...., key: value}
3 # 定义字典变量
4 my_dict = {key: value, key: value, ...., key: value}
5 # 定义空字典
6 my_dict = {}          # 空字典定义方式1
7 my_dict = dict()      # 空字典定义方式2
```

字典不允许key重复，但是value可能会重复（和java一样）

字典的常用方法

编号	操作	说明
1	字典[Key]	获取指定Key对应的Value值
2	字典[Key] = Value	添加或更新键值对
3	字典.pop(Key)	取出Key对应的Value并在字典内删除此Key的键值对
4	字典.clear()	清空字典
5	字典.keys()	获取字典的全部Key，可用于for循环遍历字典
6	len(字典)	计算字典内的元素数量

字典的特点

经过上述对字典的学习，可以总结出字典有如下特点：

- 可以容纳多个数据
- 可以容纳不同类型的数据
- 每一份数据是KeyValue键值对
- 可以通过Key获取到Value，Key不可重复（重复会覆盖）
- 不支持下标索引
- 可以修改（增加或删除更新元素等）
- 支持for循环，不支持while循环

数据容器总结

数据容器可以从以下视角进行简单的分类：

- 是否支持下标索引
 - 支持：列表、元组、字符串 – 序列类型
 - 不支持：集合、字典 – 非序列类型
- 是否支持重复元素：
 - 支持：列表、元组、字符串 – 序列类型
 - 不支持：集合、字典 – 非序列类型
- 是否可以修改
 - 支持：列表、集合、字典
 - 不支持：元组、字符串

	列表	元组	字符串	集合	字典
元素数量	支持多个	支持多个	支持多个	支持多个	支持多个
元素类型	任意	任意	仅字符	任意	Key: Value Key: 除字典外任意类型 Value: 任意类型
下标索引	支持	支持	支持	不支持	不支持
重复元素	支持	支持	支持	不支持	不支持
可修改性	支持	不支持	不支持	支持	支持
数据有序	是	是	是	否	否
使用场景	可修改、可重复的一批数据记录场景	不可修改、可重复的一批数据记录场景	一串字符的记录场景	不可重复的数据记录场景	以Key检索Value的数据记录场景

1. 基于各类数据容器的特点，它们的应用场景如下：

- 列表：一批数据，可修改、可重复的存储场景
- 元组：一批数据，不可修改、可重复的存储场景
- 字符串：一串字符串的存储场景
- 集合：一批数据，去重存储场景
- 字典：一批数据，可用Key检索Value的存储场景

首先，在遍历上：

- 5类数据容器都支持for循环遍历
- 列表、元组、字符串支持while循环，集合、字典不支持（无法下标索引）

容器通用功能

功能	描述
通用for循环	遍历容器（字典是遍历key）
max	容器内最大元素
min()	容器内最小元素
len()	容器元素个数
list()	转换为列表
tuple()	转换为元组
str()	转换为字符串
set()	转换为集合
sorted(序列, [reverse=True])	排序, reverse=True表示降序 得到一个排好序的列表

回忆：函数是一个封装的代码单元，可以提供特定功能。

在Python中，如果将函数定义为class（类）的成员，那么函数会称之为：方法

```
def add(x, y):  
    return x + y
```

函数

```
class Student:
```

```
    def add(self, x, y):  
        return x + y
```

方法

方法和函数功能一样，有传入参数，有返回值，只是方法的使用格式不同：

函数的使用：`num = add(1, 2)`

方法的使用：`student = Student()`

`num = student.add(1, 2)`

面向对象啦

第六章

文件操作

文件读取操作

open()打开函数

在Python，使用open函数，可以打开一个已经存在的文件，或者创建一个新文件，语法如下

```
open(name, mode, encoding)
```

name：是要打开的目标文件名的字符串（可以包含文件所在的具体路径）。

mode：设置打开文件的模式（访问模式）：只读、写入、追加等。

encoding：编码格式（推荐使用UTF-8）

示例代码：

```
f = open('python.txt', 'r', encoding=" UTF-8")  
# encoding的顺序不是第三位，所以不能用位置参数，用关键字参数直接指定
```

注意事项

注意：此时的`f`是`open`函数的文件对象，对象是Python中一种特殊的数据类型，拥有属性和方法，可以使用对象.属性或对象.方法对其进行访问，后续面向对象课程会给大家进行详细的介绍。

mode常用的三种基础访问模式

模式	描述
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
w	打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，原有内容会被删除。 如果该文件不存在，创建新文件。
a	打开一个文件用于追加。如果该文件已存在，新的内容将会被写入到已有内容之后。 如果该文件不存在，创建新文件进行写入。

读操作相关方法

read()方法:

```
文件对象.read(num)
```

num表示要从文件中读取的数据的长度（单位是字节），如果没有传入num，那么就表示读取文件中所有的数据。

readlines()方法:

readlines可以按照行的方式把整个文件中的内容进行一次性读取，并且返回的是一个列表，其中每一行的数据为一个元素。

```
f = open('python.txt')  
content = f.readlines()  
  
# ['hello world\n', 'abcdefg\n', 'aaa\n', 'bbb\n', 'ccc']  
print(content)  
  
# 关闭文件  
f.close()
```

readline方法可以一次读取一行

```
# 读取文件 - readline()  
line1 = f.readline()  
line2 = f.readline()  
line3 = f.readline()  
print(f"第一行数据是: {line1}")  
print(f"第二行数据是: {line2}")  
print(f"第三行数据是: {line3}")
```

读取整个文件（这个方法目前还不知道为什么可以，f是文件类型也可以遍历欸）

```
# for循环读取文件行  
for line in f:  
    print(f"每一行数据是:{line}")
```

如果在一个程序中多次调用read或者readlines方法，会在上一次读取的结尾处继续读取下一次read，跟文件指针有关

with open语法

```
# with open 语法规操作文件    文件操作执行完之后会自动关闭该文件  
with open("D:/测试.txt", "r", encoding="UTF-8") as f:  
    for line in f:  
        print(f"每一行数据是: {line}")
```

操作汇总

操作	功能
文件对象 = open(file, mode, encoding)	打开文件获得文件对象
文件对象.read(num)	读取指定长度字节 不指定num读取文件全部
文件对象.readline()	读取一行
文件对象.readlines()	读取全部行，得到列表
for line in 文件对象	for循环文件行，一次循环得到一行数据
文件对象.close()	关闭文件对象
with open() as f	通过with open语法打开文件，可以自动关闭

文件写入操作

```
# 1. 打开文件
f = open('python.txt', 'w')

# 2. 文件写入
f.write('hello world')

# 3. 内容刷新
f.flush()
```

注意：

- 直接调用write，内容并未真正写入文件，而是会积攒在程序的内存中，称之为缓冲区
- 当调用flush的时候，内容会真正写入文件
- 这样做是避免频繁的操作硬盘，导致效率下降（攒一堆，一次性写磁盘）

需要注意的是close方法内置了flush方法，也就是说close之前先进行了一次flush操作

注意事项

- w模式，文件不存在，会创建新文件
- w模式，文件存在，会清空原有内容

文件追加写入操作

注意：

- a模式，文件不存在会创建文件
- a模式，文件存在会在最后，追加写入文件

写入方法和w模式一样

第七章

异常

当检测到一个错误时，解释器就无法继续执行了，出现了报错信息，这就是BUG

异常就是程序运行的过程中出现了错误

世界上没有完美的程序，任何程序在运行的过程中，都有可能出现异常，也就是出现bug导致程序无法完美运行下去。

我们要做的，不是力求程序完美运行。

而是在力所能及的范围内，对可能出现的bug，进行提前准备、提前处理

这种行为我们称之为：异常处理（捕获异常）

当我们的程序遇到了BUG，那么接下来有两种情况：

①整个程序因为一个BUG停止运行

②对BUG进行提醒，整个程序继续运行

显然在之前的学习中，我们所有的程序遇到BUG就会出现①的这种情况，也就是整个程序直接奔溃。

但是在真实工作中，我们肯定不能因为一个小的BUG就让整个程序全部奔溃，也就是我们希望的是达到②的这种情况那这里我们就需要使用到捕获异常

捕获异常的作用在于：提前假设某处会出现异常，做好提前准备，当真的出现异常的时候，可以有后续手段。

捕获所有异常

基本语法

基本语法：

```
try:  
    可能发生错误的代码  
except:  
    如果出现异常执行的代码
```

快速入门

需求：尝试以`r`模式打开文件，如果文件不存在，则以`w`方式打开。

```
try:  
    f = open('linux.txt', 'r')  
except:  
    f = open('linux.txt', 'w')
```

下面这个也是捕获所有的异常哦

```
# 捕获所有异常  
try:  
    1 / 0  
except Exception as e:  
    print("出现异常了")
```

捕获指定的异常

捕获指定异常

基本语法:

```
try:  
    print(name)  
except NameError as e:  
    print('name变量名称未定义错误')
```

注意事项

- ① 如果尝试执行的代码的异常类型和要捕获的异常类型不一致，则无法捕获异常。
- ② 一般try下方只放一行尝试执行的代码。

上面这个e就是相当于java异常那个e

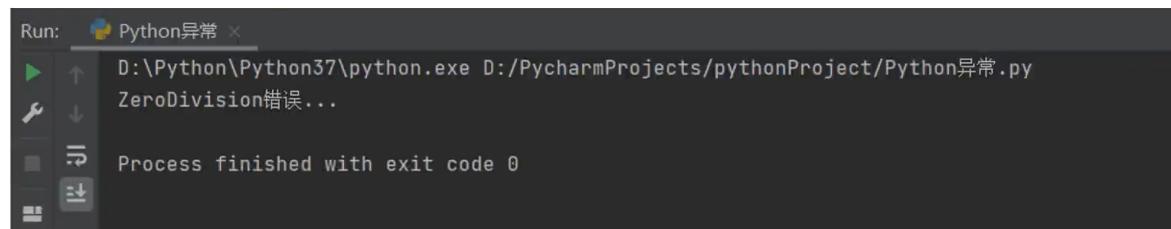
捕获多个异常

捕获多个异常

当捕获多个异常时，可以把要捕获的异常类型的名字，放到except 后，并使用元组的方式进行书写。

```
try:  
    print(1/0)  
except (NameError, ZeroDivisionError):  
    print('ZeroDivision错误...')
```

执行结果:



```
Run: Python异常  
D:\Python\Python37\python.exe D:/PycharmProjects/pythonProject/Python异常.py  
ZeroDivision错误...  
Process finished with exit code 0
```

异常else

异常else

else表示的是如果没有异常要执行的代码。

```
try:  
    print(1)  
except Exception as e:  
    print(e)  
else:  
    print('我是else， 是没有异常的时候执行的代码')
```

执行结果：



```
Run: Python异常  
D:\Python\Python37\python.exe D:/PycharmProjects/pythonProject/Python异常.py  
1  
我是else， 是没有异常的时候执行的代码  
Process finished with exit code 0
```

异常的finally

异常的finally

finally表示的是无论是否异常都要执行的代码，例如关闭文件。

```
try:  
    f = open('test.txt', 'r')  
except Exception as e:  
    f = open('test.txt', 'w')  
else:  
    print('没有异常， 真开心')  
finally:  
    f.close()
```

异常的传递

异常的传递

异常是具有传递性的

当函数func01中发生异常，并且没有捕获处理这个异常的时候，异常会传递到函数func02，当func02也没有捕获处理这个异常的时候main函数会捕获这个异常，这就是异常的传递性.

提示：

当所有函数都没有捕获异常的时候，程序就会报错

```
def func01(): 异常在func01中没有被捕获
    print("这是func01开始")
    num = 1 / 0
    print("这是func01结束")

def func02(): _____
    print("这是func02开始")
    func01() ←
    print("这是func02结束")

def main(): 异常在mian中被捕获
    try:
        func02() ←
    except Exception as e:
        print(e)

main()
```

和java的一样

执行结果是

func2 开始执行

func1 开始执行

python的模块

和c语言很像嘿嘿

什么是模块

Python 模块(Module)，是一个 Python 文件，以 .py 结尾。模块能定义函数，类和变量，模块里也能包含可执行的代码。

模块的作用： python中有很多各种不同的模块，每一个模块都可以帮助我们快速的实现一些功能，比如实现和时间相关的功能就可以使用time模块
我们可以认为一个模块就是一个工具包，每一个工具包中都有各种不同的工具供我们使用进而实现各种不同的功能。

大白话：模块就是一个Python文件，里面有类、函数、变量等，我们可以拿过来用（导入模块去使用）

模块的导入方式

模块在使用前需要先导入 导入的语法如下：

```
[from 模块名] import [模块 | 类 | 变量 | 函数 | *] [as 别名]
```

常用的组合形式如：

- import 模块名
- from 模块名 import 类、变量、方法等
- from 模块名 import *
- import 模块名 as 别名
- from 模块名 import 功能名 as 别名

基本语法：

```
import 模块名  
import 模块名1, 模块名2
```

```
模块名.功能名()
```

案例：导入time模块

```
# 导入时间模块  
import time  
  
print("开始")  
# 让程序睡眠1秒(阻塞)  
time.sleep(1)  
print("结束")
```

也可以只导入功能（方法/函数）

from 模块名 import 功能名

基本语法：

from 模块名 import 功能名

功能名()

案例：导入time模块中的sleep方法

```
# 导入时间模块中的sleep方法
from time import sleep

print("开始")
# 让程序睡眠1秒(阻塞)
sleep(1)
print("结束")
```

这种方法可以直接用方法名调用方法

导入一个模块下的所有功能

from 模块名 import *

基本语法：

```
from 模块名 import *
```

功能名()

案例：导入time模块中所有的方法

```
# 导入时间模块中所有的方法
```

```
from time import *
```

```
print("开始")
```

```
# 让程序睡眠1秒(阻塞)
```

```
sleep(1)
```

```
print("结束")
```

*表示全部

这样的方式也可以直接用方法名直接调用方法

as 定义别名

基本语法:

```
# 模块定义别名  
import 模块名 as 别名
```

```
# 功能定义别名  
from 模块名 import 功能 as 别名
```

案例:

```
# 模块别名  
import time as tt  
  
tt.sleep(2)  
print('hello')
```

```
# 功能别名  
from time import sleep as sl  
sl(2)  
print('hello')
```

因为有些模块名字实在是太长了，我们可以用as给它改个名字

上面只导入功能的模式也可以改名

```
from time import sleep as sl  
print("你好")  
sl(5)  
print("我好")
```

自定义模块

案例：新建一个Python文件，命名为my_module1.py，并定义test函数

```
my_module1.py
1 def test(a, b):
2     print(a + b)
3

text_my_module.py
1 import my_module1
2
3 my_module1.test(10, 20)
4
5
6
```

如果不想在导入模块中执行导入模块中的语句时

可以加入这段话

```
if __name__ == '__main__':
    test(1, 2)
```

if __main__ == "__main__" 表示，只有当程序是直接执行的才会进入
if内部，如果是被导入的，则if无法进入

这句话的意思是这个运行程序是主函数才会执行，如果是导入进去的话就不执行

__all__变量

__all__

如果一个模块文件中有`__all__`变量，当使用`from xxx import *`导入时，只能导入这个列表中的元素

```
my_module1.py
1 __all__ = ['test_A']
2
3 def test_A():
4     print('testA')
5
6 def test_B():
7     print('testB')

text_my_module.py
1 from my_module1 import *
2
3 test|          这里只能使用test_A函数
4 f test_A()
5 ^ and ^ will move caret down and up in the editor Next Tip
6
7
```

上面的只能导入test_A方法，没有导入test_B方法

__all__变量可以控制import *的时候哪些功能可以被导入(因为星号代表的是全部，跟英文表达意思是一样的哦)

all变量是一个列表，里面可以添加方法名称

python的包

基于Python模块，我们可以在编写代码的时候，导入许多外部代码来丰富功能。

但是，如果Python的模块太多了，就可能造成一定的混乱，所以我们可以用python的包管理机制

从物理上看，包就是一个文件夹，在该文件夹下包含了一个_init_.py文件，该文件夹可用于包含多个模块文件从逻辑上看，包的本质依然是模块

_init_的作用

创建包会默认自动创建这个文件，通过这个文件来表示一个文件夹是Python的包，而非普通的文件夹。

和java的包一样

通过import导入

The screenshot shows a PyCharm interface. On the left, there's a file tree with a package named 'my_package' highlighted by a red box. Inside 'my_package', there are two files: '_init_.py' and 'my_module1.py'. On the right, there's a code editor with the following Python code:

```
# 创建一个包
# 导入自定义的包中的模块，并使用
import my_package.my_module1
import my_package.my_module2

my_package.my_module1.info_print1()
my_package.my_module2.info_print2()
```

通过from导入

The screenshot shows a PyCharm interface. On the left, there's a file tree with a package named 'my_package' highlighted by a red box. Inside 'my_package', there are two files: '_init_.py', 'my_module1.py', and 'my_module2.py'. On the right, there's a code editor with the following Python code:

```
# my_package.my_module1.info_print1()
# my_package.my_module2.info_print2()

from my_package import my_module1
from my_package import my_module2
my_module1.info_print1()
my_module2.info_print2()

# 通过__all__变量，控制import *
```

甚至可以采取这种方式

```
from my_package.my_module1 import info_print1
from my_package.my_module2 import info_print2
info_print1()
info_print2()
```

导入第三方包

第一种方法

首先我们得下载这个包哦，相当于java的jar包

打开我们许久未见的：命令提示符程序，在里面输入：

pip install 包名称

即可通过网络快速安装第三方包

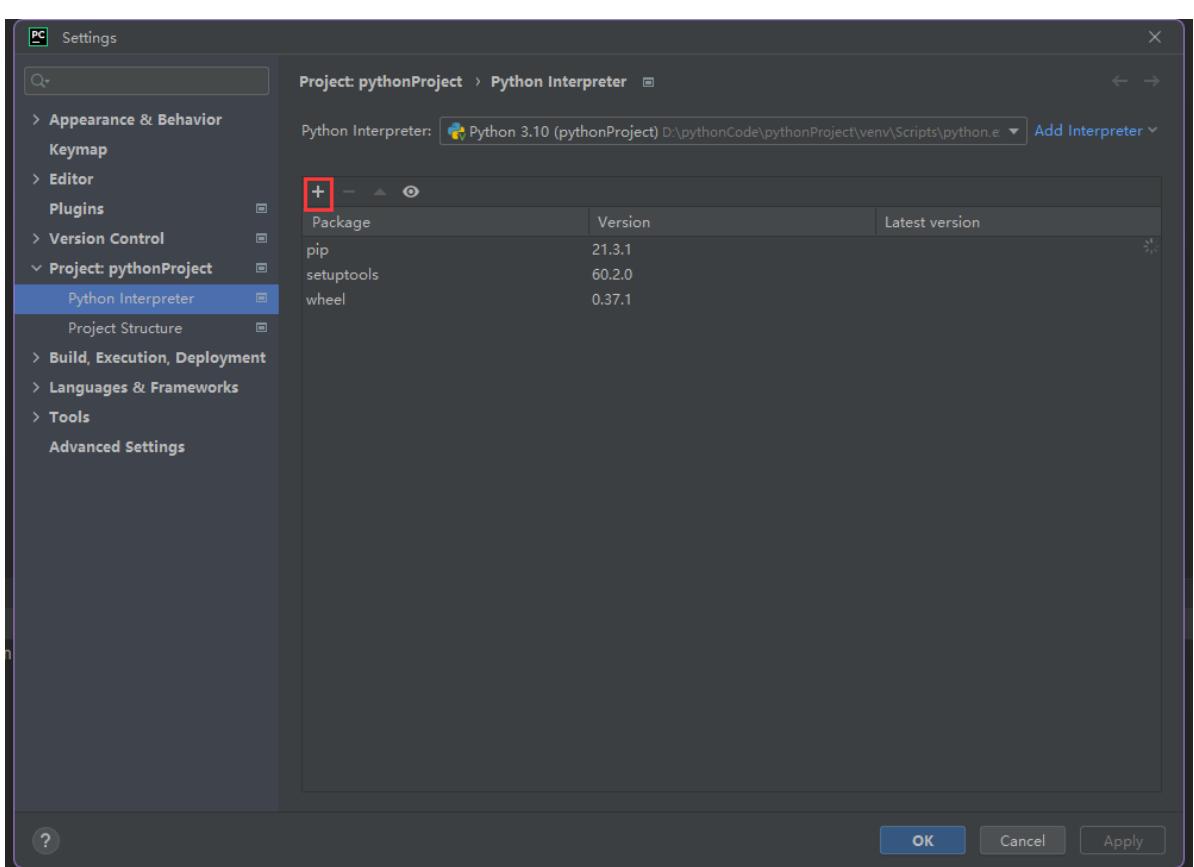
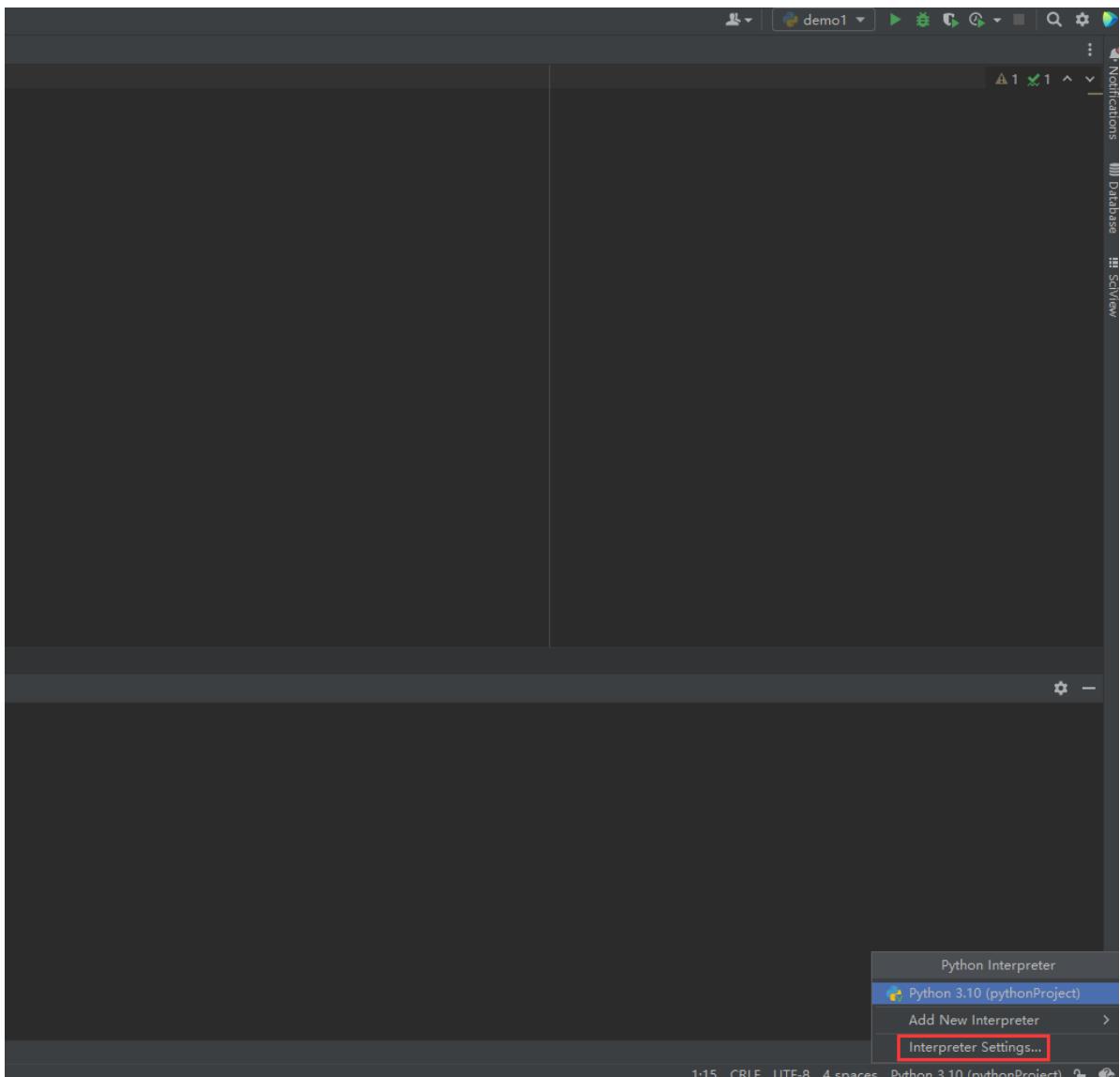


上面的下载速度太慢啦，可以配置镜像

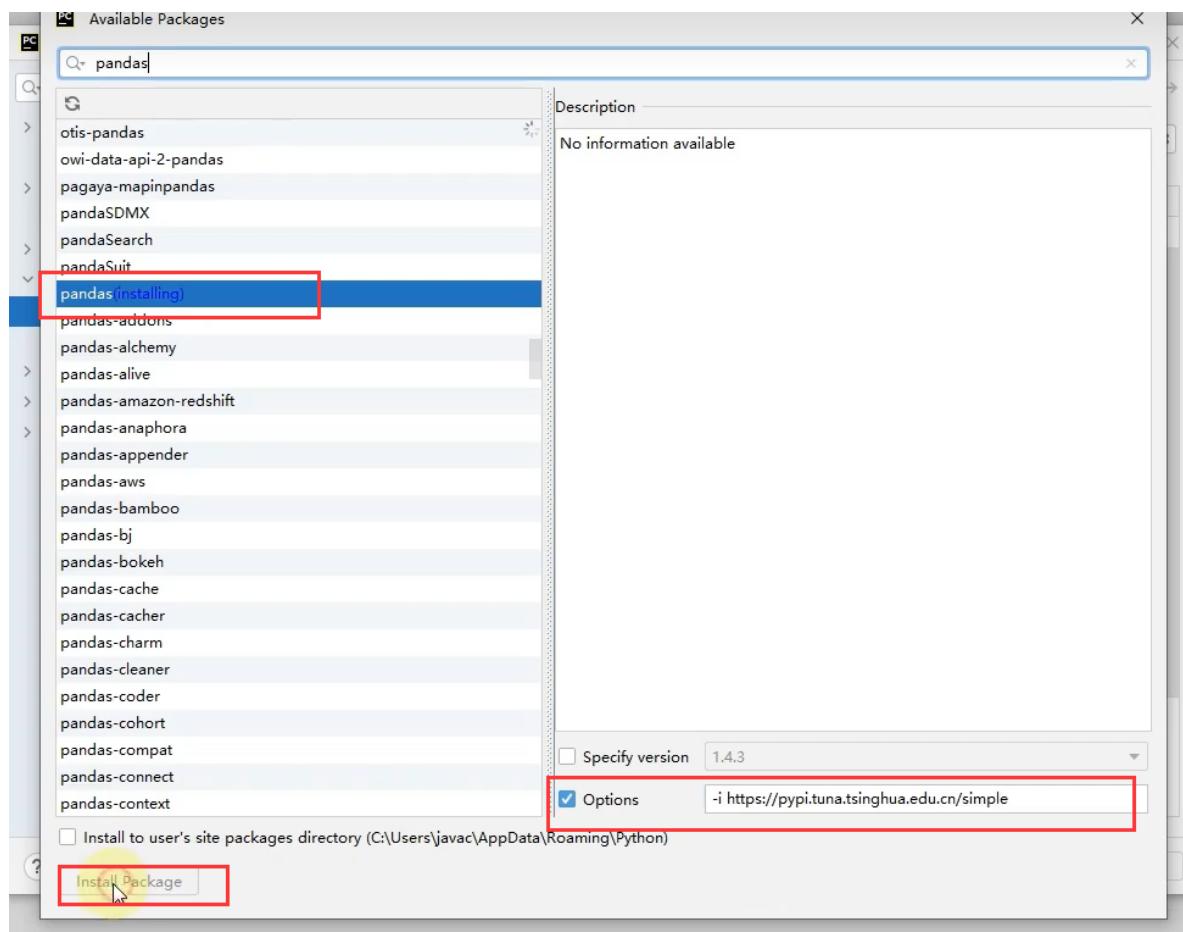
```
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple 包名
```

第二种方法

pycharm内置下载



之后就可以搜索第三方包下载了



第八章

python案例

python数据和json数据的相互转化

- Python数据和Json数据的相互转化

```
# 导入json模块
import json

# 准备符合格式json格式要求的python数据
data = [{"name": "老王", "age": 16}, {"name": "张三", "age": 20}]

# 通过 json.dumps(data) 方法把python数据转化为了 json数据
data = json.dumps(data)

# 通过 json.loads(data) 方法把json数据转化为了 python数据
data = json.loads(data)
```