

# 环境

## 安装包安装

如果你想用python的安装包的话，直接去官网下载：[点我下载](#)

### Windows

<https://www.python.org/downloads/windows>

安装方式：[双击exe文件](#) 安装即可，如安装过程中有 [添加环境变量](#) 选项，勾选即可，否则安装完成后需要自行配置环境变量。

### Linux

安装Python之前需要安装（否则有的地方还需要安装，会报错，需要重新编译）

```
yum -y install libffi-devel

yum -y groupinstall "Development tools"

yum -y install zlib-devel bzip2-devel openssl-devel ncurses-devel sqlite-devel
readline-devel tk-devel
```

#### CentOS一键安装Python脚本

需要将对应的Python安装包放在脚本同目录下，并修改脚本中安装的Python的文件名（无文件后缀）

下载地址：<https://www.python.org/downloads/source/>

选择对应版本，这里以Python-3.8.0为例，[点我下载](#)

```
source ~/.bashrc
# #!/bin/bash

CURRENT_PATH=`pwd`
PYTHON_VERSION='Python-3.8.0'
PYTHON_PATH=${CURRENT_PATH}/${PYTHON_VERSION}
echo 当前安装的Python版本是:${PYTHON_VERSION}
yum -y install libffi-devel
yum -y install yum-utils
tar xf ${PYTHON_VERSION}.tgz
cd ./${PYTHON_VERSION}
./configure --prefix=${PYTHON_PATH}
make && make install
ln -s ${PYTHON_PATH}/python /usr/local/bin/python3
ln -s ${PYTHON_PATH}/bin/pip3 /usr/local/bin/pip3
python3 -V
```

```
# 注意：如果和Anaconda混用的话，并且以此版本为系统python版本
# 需要将 ~/.bashrc 中的 Anaconda 环境删除，然后 source ~/.bashrc，否则用的是 Anaconda 的版本
```

## Anaconda

### Windows安装Anaconda

使用Anaconda集成环境，便于不同版本的项目创建不同的虚拟环境。

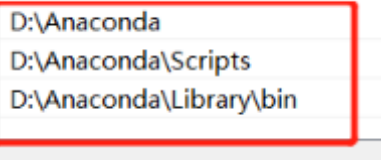
Anaconda下载地址：[点我下载](#)

Miniconda下载地址：[点我下载](#)

下载后安装，记得勾选：

```
Add Anaconda3 to my PATH environment variable
```

或者手动将Anaconda的环境变量添加进系统PATH中



D:\Anaconda  
D:\Anaconda\Scripts  
D:\Anaconda\Library\bin

安装完成后，打开cmd控制台，输入 `conda -V` 查看是否安装成功。

控制台输出以下内容，表示成功。

```
conda 23.7.2
```

### Linux安装Anaconda

```
# 下载
wget -c https://repo.anaconda.com/archive/Anaconda3-2023.07-2-Linux-x86_64.sh
# 给执行权限
chmod 775 Anaconda3-2023.07-2-Linux-x86_64.sh
# 执行安装过程 安装过程中要按下enter和输入yes，常规操作
# 运行
bash Anaconda3-2023.07-2-Linux-x86_64.sh
# 安装完成后，添加环境变量
vim ~/.bashrc
# 在文件末尾加上anaconda的路径
export PATH=/root/anaconda3/bin:$PATH
# 激活环境变量
source ~/.bashrc
# 验证是否安装成功
conda --version
```

本人通过虚拟机安装时遇到过 `name or service not known` 错误，和上面安装无关，这是我虚拟机的问题，这里记录下解决方式。

```
vi /etc/resolv.conf
# 在后面添加
nameserver 8.8.8.8
# 保存退出
```

## 配置清华镜像源

配置清华镜像源后，国内下载依赖包速度会大大增加。

使用 conda 配置添加channels信息。

```
# 使用conda配置添加channels信息
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/r
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/pro
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/msys2
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-
forge
conda config --add channels
https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/pytorch/
# 配置下载的时候显示channels相关的url信息
conda config --set show_channel_urls yes
```

conda 恢复channel的默认设置

```
conda config --remove-key channels
conda config --add channels defaults
```

## 创建python虚拟环境

打开cmd控制台，输入以下命令：

```
# 其中env_name表示虚拟环境的名称
# 3.8 表示使用的python版本
conda create --name env_name python=3.8
# 创建完成后，进入(激活)某个虚拟环境
conda activate env_name
```

以上操作也可使用Anaconda可视化界面软件完成。

Anaconda安装完成后，自带可视化软件。

## Jupyter配可切换置虚拟环境

```
# 1、在base环境，终端执行以下命令
conda install nb_conda_kernels

# 2、创造个新环境，并在该环境安装ipykernel
conda create -n env_name python=3.8
conda activate env_name
conda install ipykernel

# 3、在base环境启动 jupyter notebook
jupyter notebook

# 4、然后，就可以在内核切换不同的jupyter环境
```

## 常用Conda命令

```
# 查看conda版本，验证是否安装
conda --version

# 更新至最新版本，也会更新其它相关包
conda update conda

# 更新所有包
conda update --all

# 更新指定的包
conda update package_name

# 创建名为env_name的新环境，并在该环境下安装名为package_name 的包，可以指定新环境的版本号
# 例如: conda create -n python2 python=python2.7 numpy pandas，创建了python2环境，python版本为2.7，同时还安装了numpy pandas包
conda create -n env_name package_name

# 从原始cmd环境切换至base虚拟环境
activate

# 切换至env_name环境
conda activate env_name

# 退出环境
conda deactivate

# 显示所有已经创建的环境
conda info -e

# 复制old_env_name为new_env_name
conda create --name new_env_name --clone old_env_name

# 删除环境
conda remove --name env_name --all

# 查看所有已经安装的包
conda list

# 在当前环境中安装包
conda install package_name

# 在指定环境中安装包
conda install --name env_name package_name

# 删除指定环境中的包
conda remove --name env_name package

# 删除当前环境中的包
conda remove package
```

## 1. 个人常用

```
# 安装、卸载
conda create -n tensorflow2 python=3.9
conda remove -n tensorflow2 --all# 清理
conda clean --all## 更新
conda update conda
conda update -y --all
```

## 2. 安装

```
conda install pandas
conda install numpy
conda install tqdmconda install matplotlib
conda install scikit-learnpip install swifter
```

## 更新相关命令

```
conda update conda
conda update anaconda
conda update -y --all
conda update anaconda-navigator
conda update -n base conda
```

## conda环境相关命令

```
conda create -n env_name python=3.7 #创建python版本3.7, 名字为env_name虚拟环境
conda remove -n env_name --all      #删除名字为env_name虚拟环境
conda activate                       #激活默认base环境
conda activate env_name              #激活名为env_name的环境
conda deactivate                     #关闭当前环境
conda env list                       #显示所有的虚拟环境
conda info --envs                    #显示所有的虚拟环境
```

## 安装、卸载包相关

```
conda list                           #查看当前环境已经安装的包
conda list -n env_name               #查看env_name虚拟环境下安装的package
conda install pck_name               #安装名为pck_name的包
conda update pck_name                #更新名为pck_name的包
conda uninstall pck_name             #卸载名为pck_name的包
```

## pip管理安装包

```
pip list                             #列出当前缓存的包
pip purge                            #清除缓存
pip remove                           #删除对应的缓存
pip help                             #帮助
pip install xxx                      #安装xxx包
pip uninstall xxx                    #删除xxx包
pip show xxx                         #展示指定的已安装的xxx包
```

```
pip check xxx          #检查xxx包的依赖是否合适
```

---

以上安装依赖包的命令使用pip也可以。

## 常用pip命令

```
# 要是觉得自己的pip版本有点低，想要升级一下的话
pip install --upgrade pip

# 安装第三方的包
pip install package-name

# 安装指定版本的第三方的包
pip install package-name==版本号

# 卸载某个包
pip uninstall package_name

# 更新某个包
pip install --upgrade package_name
# 或者是
pip install -U package_name

# 查看某个包的信息
pip show -f package_name

# 查看需要被升级的包
pip list -o

# 查看兼容问题
# 在下载安装一些标准库的时候，需要考虑到兼容问题
# 一些标准库的安装可能需要依赖其他的标准库，会存在版本相冲突等问题
# 先用下面这条命令行来检查一下是否有冲突的问题存在
pip check package_name

# 指定国内源来安装
pip install -i https://pypi.douban.com/simple/ package_name

# 想要下载某个包到指定的路径下，下载包但是不安装
pip download package_name -d "某个路径"

# 生成requirements.txt，记录依赖包版本
pip freeze > requirements.txt

# 批量安装第三方库
pip install -r requirements.txt
```

## pip换源

---

如果你没有用Anaconda，而是直接使用安装包安装的python。

## pip配置稳定的国内镜像源

清华: <https://pypi.tuna.tsinghua.edu.cn/simple>

阿里云: <http://mirrors.aliyun.com/pypi/simple/>

华为云: <https://repo.huaweicloud.com/repository/pypi/simple>

中国科技大学 <https://pypi.mirrors.ustc.edu.cn/simple/>

华中理工大学: <http://pypi.hustunique.com/>

山东理工大学: <http://pypi.sdutlinux.org/>

豆瓣: <http://pypi.douban.com/simple/>

### 方法1 (永久更改)

进入我的电脑, 输入

```
%APPDATA%
```

按下回车, 自动转到一个文件夹

在这里, 我们新建一个文件夹, 命名为pip

在这个文件夹里, 新建pip.ini文件, 内容:

```
[global]
time-out = 60
index-url = https://pypi.tuna.tsinghua.edu.cn/simple/

[install]
trusted-host = tsinghua.edu.cn
```

### 方法2 (永久更改)

```
pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple/
```

### 方法3 (临时性使用)

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

## Pycharm

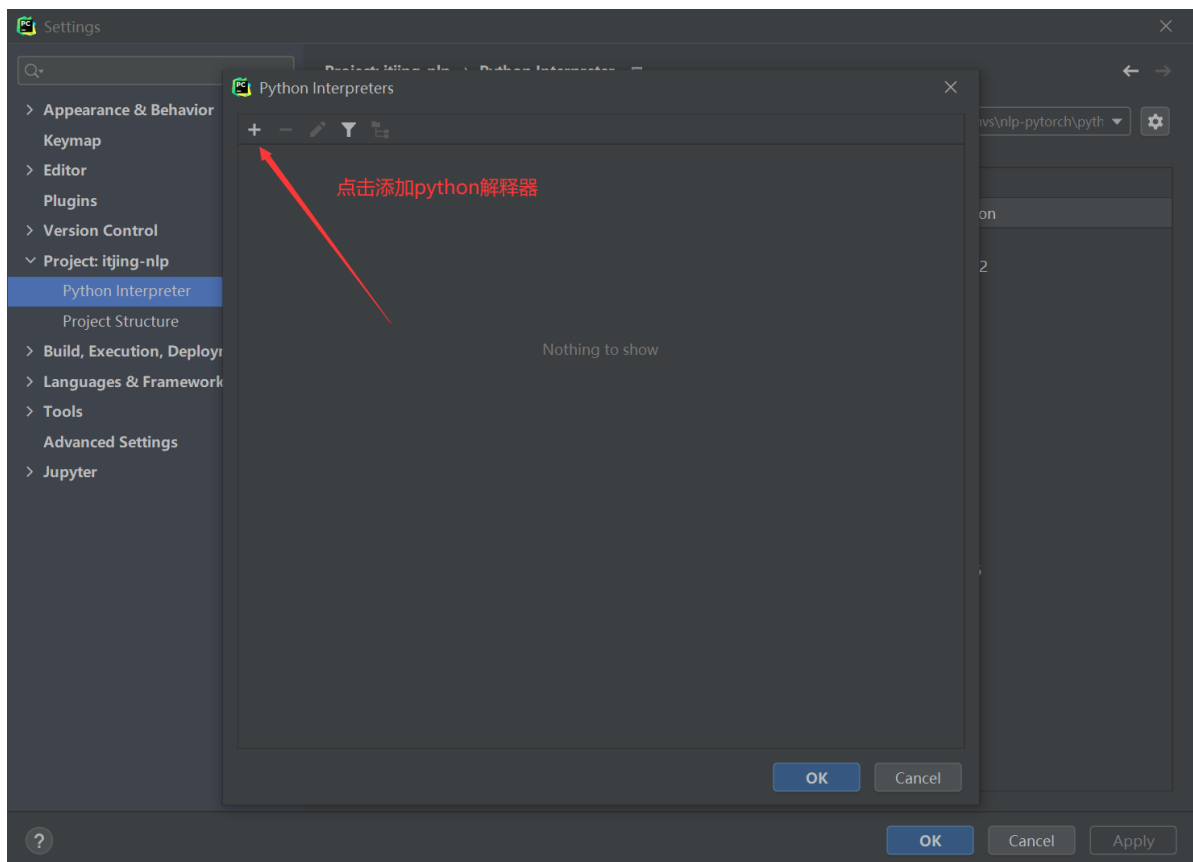
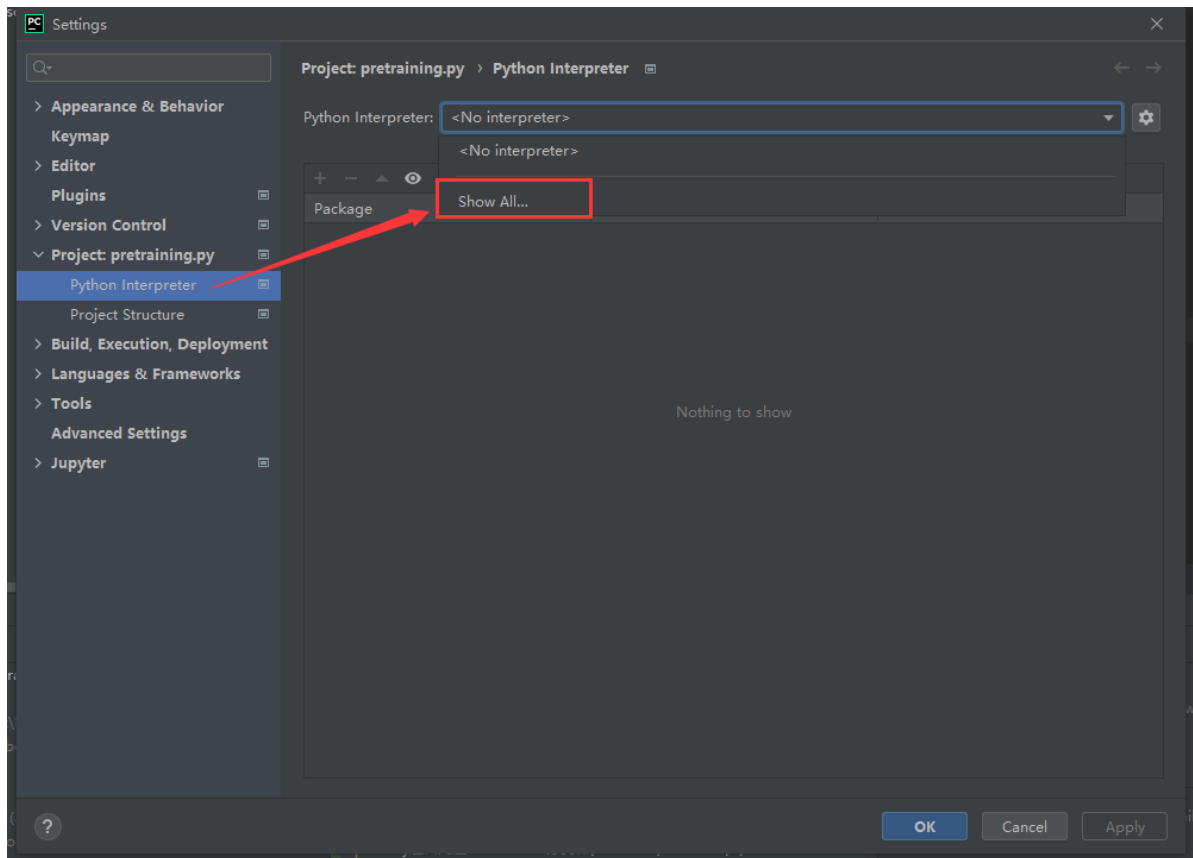
Pycharm中文网: [点我查看](#)

Pycharm历史版本下载 (建议2021.3.1): [点我去下载](#)

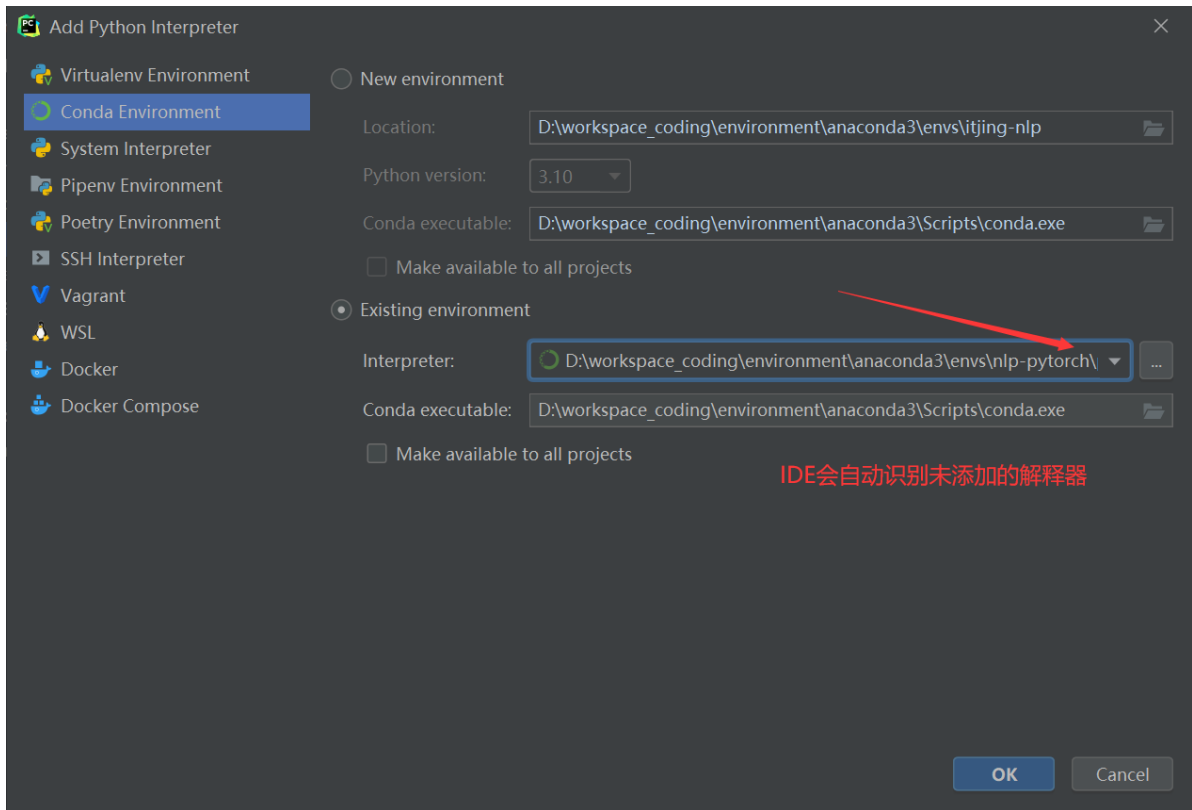
Pycharm破解: [点我查看](#)

# Pycharm设置python解释器

使用Anaconda虚拟环境



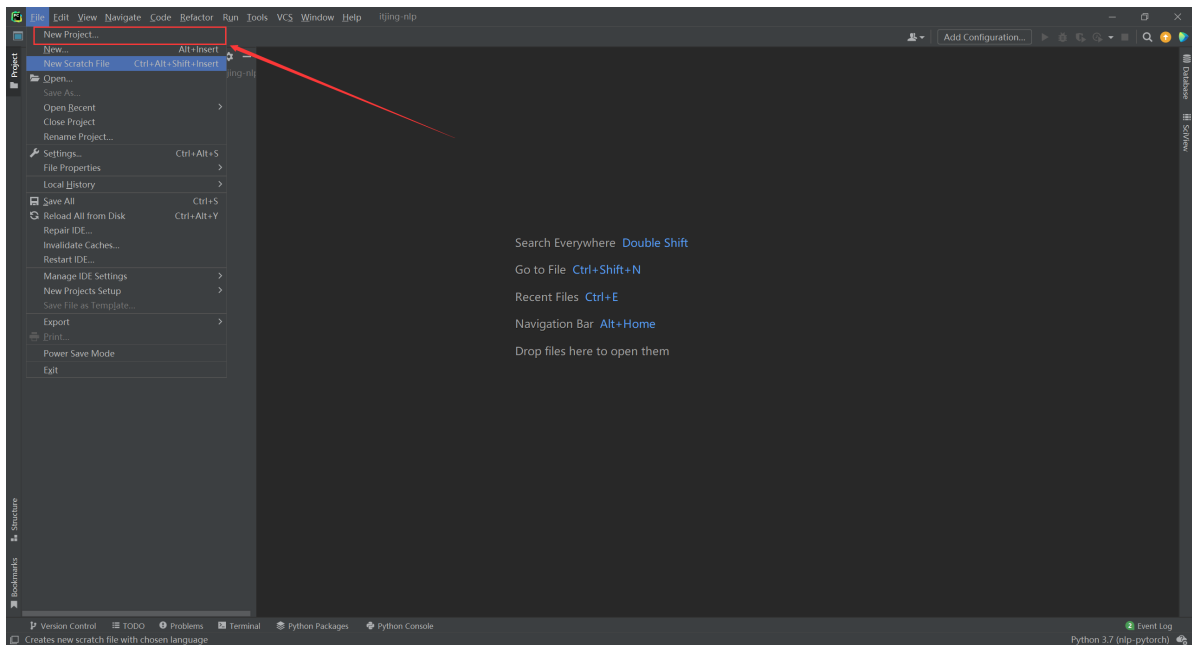


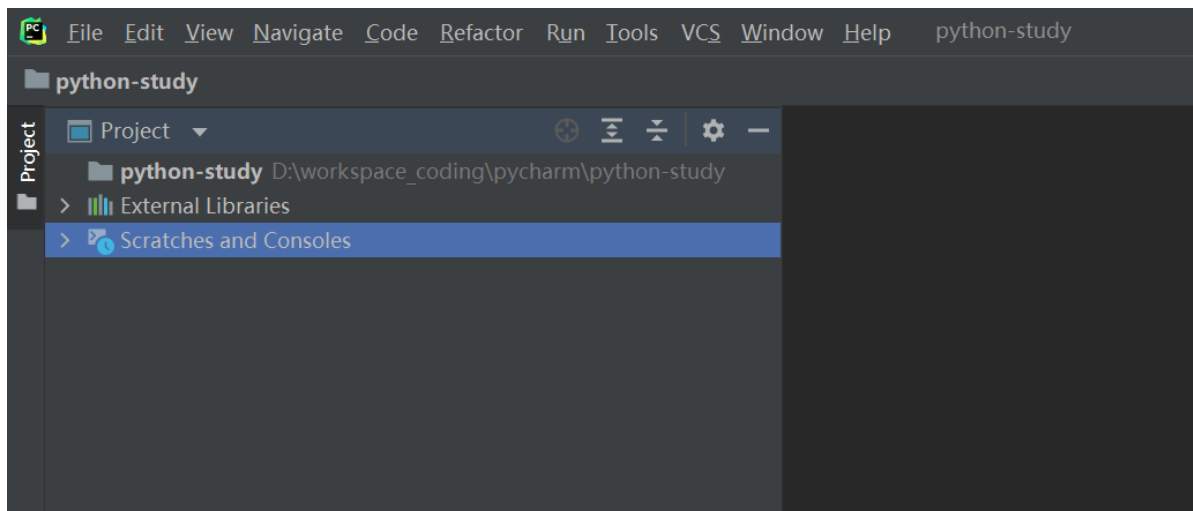
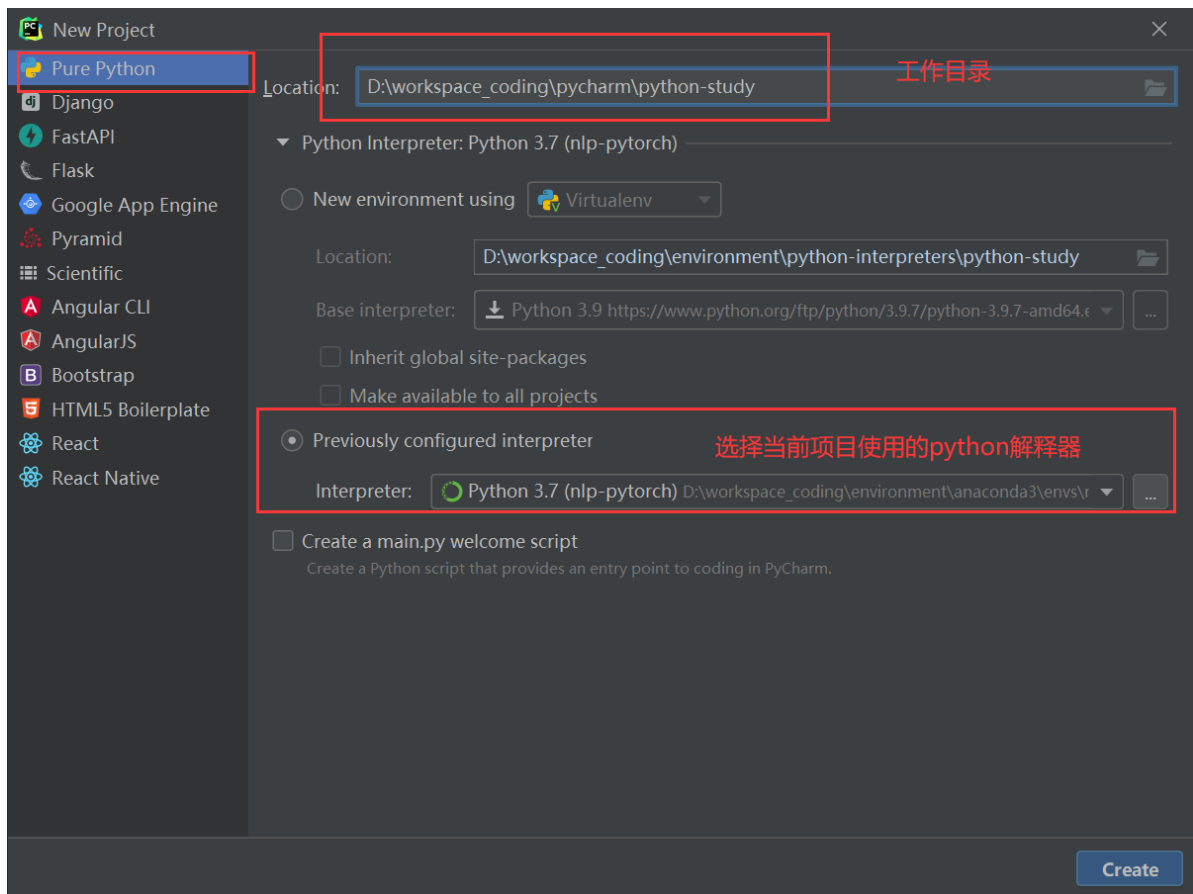


选择好了以后，将对应的python解释器设置为当前项目的主解释器。

## New Project

### 新建项目





# Python

## 资源

备忘清单

中文文档

## 基础语法

## Pycharm设置模板头

File | Settings | Editor | File and Code Templates | Python Script

```
# -*- coding: utf-8 -*-
'''
-----
@Time      : ${YEAR}-${MONTH}-${DAY} ${HOUR}:${MINUTE}
@author    : lijing
@File      : ${NAME}.py
@Description:
-----
'''
```

## 系统关键字

```
import keyword
print(keyword.kwlist)

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

```
'yield'  
]  
...
```

## 文件编码

默认情况下，3.x 源码文件都是 UTF-8 编码，字符串都是 Unicode 字符。也可以手动指定文件编码。

注意: 该行标注必须位于文件第一行

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

或者

```
# encoding: utf-8
```

## 标识符

- 第一个字符必须是英文字母或下划线 `_`。
- 标识符的其他部分由字母、数字和下划线组成。
- 标识符对大小写敏感。

注：从 3.x 开始，非 ASCII 标识符也是允许的，但不建议。

## 注释

单行注释采用 `#`，多行注释采用 `'''` 或 `"""`。

```
# 这是单行注释
```

```
'''
```

```
这是多行注释
```

```
这是多行注释
```

```
'''
```

```
"""
```

```
这也是多行注释
```

```
这也是多行注释
```

```
"""
```

## 行与缩进

Python 最具特色的就是使用缩进来表示代码块，不需要使用大括号 `{}`。

缩进的空格数是可变的，但是同一个代码块的语句必须包含相同的缩进空格数。

缩进不一致，会导致运行错误。

## 多行语句

Python 通常是一行写完一条语句，但如果语句很长，可以使用反斜杠 `\` 来实现多行语句。

```
total = item_one + \
        item_two + \
        item_three
```

## 同一行写多条语句

Python 可以在同一行中使用多条语句，语句之间使用分号;分割。

```
import sys; x = 'hello world'; sys.stdout.write(x + '\n')
```

## print 输出

print 默认输出是换行的，如果要实现不换行需要在变量末尾加上 `end=""` 或别的非换行符字符串

```
print('lijing') # 默认换行
print('lijing', end = "") # 不换行
```

## import 与 from...import

在 Python 用 import 或者 `from...import` 来导入相应的模块

将整个模块导入，格式为：`import module_name`

从某个模块中导入某个函数，格式为：`from module_name import func1`

从某个模块中导入多个函数，格式为：`from module_name import func1, func2, func3`

将某个模块中的全部函数导入，格式为：`from module_name import *`

## 运算符

### 算术运算符

运算符	描述
+	加
-	减
*	乘
/	除
%	取模
**	幂
//	取整除（返回商的整数部分（向下取整））

## 比较运算符

运算符	描述
==	等于
!=	不等于
>	大于
<	小于
>=	大于等于
<=	小于等于

## 赋值运算符

运算符	描述
=	简单的赋值运算符
+=	加法赋值运算符
-=	减法赋值运算符
*=	乘法赋值运算符
/=	除法赋值运算符
%=	取模赋值运算符
**=	幂赋值运算符
//=	取整除赋值运算符

## 位运算符

运算符	描述
&	按位与运算符：参与运算的两个值，如果两个相应位都为1，则该位的结果为1，否则为0
	按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1
^	按位异或运算符：当两对应的二进位相异时，结果为1
~	按位取反运算符：对数据的每个二进制位取反，即把1变为0，把0变为1。~x类似于-x-1
<<	左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0
>>	右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数

## 逻辑运算符

运算符	逻辑表达式	描述
and	x and y	布尔"与", 如果x为False, x and y返回False, 否则它返回y的计算值
or	x or y	布尔"或", 如果x是True, 它返回x的值, 否则它返回y的计算值
not	not x	布尔"非", 如果x为True, 返回False。如果x为False, 它返回True

## 成员运算符

运算符	描述
in	如果在指定的序列中找到值返回True, 否则返回False
not in	如果在指定的序列中没有找到值返回True, 否则返回False

## 身份运算符

运算符	描述	实例
is	is是判断两个标识符是不是引用自一个对象	x is y, 类似 id(x) == id(y), 如果引用的是同一个对象则返回True, 否则返回False
is not	is not是判断两个标识符是不是引用自不同对象	x is not y, 类似 id(a) != id(b)。如果引用的不是同一个对象则返回结果True, 否则返回False

## 数据类型

### bool布尔类型

Python中布尔值使用常量 `True` 和 `False` 来表示, 注意大小写

### string字符串

Python中单引号和双引号使用完全相同。

使用三引号( `'''` 或 `"""` )可以指定一个多行字符串

```

print('lijing')
print("lijing")
print('''li
jing''')
print("""li
jing""")

# lijing
# lijing
# li
# jing
# li
# jing

```

```

print('li\njing') # 表示换行
print(r'li\njing') # 加r表示反斜杠不发生转义，正常输出
print('this', 'is', 'lijing') # 逐个输出字符串

# li
# jing
# li\njing
# this is lijing

```

字符串可以用 '+' 运算符连接在一起，用 '\*' 运算符重复。

```

str = 'lijing'
print(str + '你好') # 连接两个字符串
print(str * 5) # 打印该字符串5次

# lijing你好
# lijinglijinglijinglijinglijing

```

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

```

str = 'lijing'
print(str[1]) # 打印该字符串中第二个元素
print(str[2:-1]) # 打印该字符串中第三个元素到倒数第二个元素

# i
# jin

```

Python中的字符串不能改变。

```

str = 'lijing'
str[0] = 's'
print(str[0])

```

```

Traceback (most recent call last):
  File "D:/workspace_coding/pycharm/python-study/test.py", line 12, in <module>
    str[0] = 's'
TypeError: 'str' object does not support item assignment

```

字符串的截取的语法格式如下： 变量[头下标:尾下标:步长]



```
str = 'welcome to study python'
print(str[0:13:2])

# wloet t
```

## 列表(list)

python中切片操作是左闭右开的。

切片操作即：[start:stop]

List (列表) 是 Python 中使用最频繁的数据类型。

- 列表可以完成大多数集合类的数据结构实现。列表中元素的类型可以不相同，它支持数字，字符串甚至可以包含列表（所谓嵌套）。
- 列表是写在方括号 `[]` 之间、用逗号分隔开的元素列表。
- 和字符串一样，列表同样可以被索引和截取，列表被截取后返回一个包含所需元素的新列表。
- 列表截取的语法格式如下： `变量[头下标:尾下标]`
- 列表中的元素是可以改变的，很多操作和字符串类似

```
a = ['nlp', 'lijing', 'gitee', 'github', 'python']
b = ['php']
print(a[2]) # 输出第三个元素
a[0] = 'conda' # 将列表a中的第一个元素替换
print(a[0]) # 打印a中第一个元素
print(a[0:3]) # 打印a中第一个到第三个元素。
print(a[2:-1]) # 打印a中第三个到倒数第二个元素
print(a + b) # 将列表a, b相加
print(b * 3) # 将列表b乘以三

# lij
# gitee
# conda
# ['conda', 'lijing', 'gitee']
# ['gitee', 'github']
# ['conda', 'lijing', 'gitee', 'github', 'python', 'php']
# ['php', 'php', 'php']
```

### 列表常见的操作

#### 1、增加 `append`

```
names = ['java', 'python']
names.append('php')
print(names)

# ['java', 'python', 'php']
```

#### 2、插入 `insert` / `extend`

```
names = ['java', 'python']
names.insert(2, "php")
print(names)
name = ['pytorch', 'pandas', 'conda']
names.extend(name)
print(names)

# ['java', 'python', 'php']
# ['java', 'python', 'php', 'pytorch', 'pandas', 'conda']
```

### 3、直接删除 `del`

```
names = ['java', 'python']
del names[1]
print(names)

# ['java']
```

### 4、删除元素 `pop`

```
names = ['java', 'python', 'c', 'c++', 'c#']
print(names.pop()) # 默认删除最后一个元素并返回被删除的值
print(names.pop(0)) # 删除指定索引元素并返回被删除的值
print(names)

# c#
# java
# ['python', 'c', 'c++']
```

### 5、删除指定的元素 `remove`

```
names = ['java', 'python', 'c', 'c++', 'c#']
names.remove('c#')
print(names)

# ['java', 'python', 'c', 'c++']
```

### 6、清空 `clear`

```
names = ['java', 'python', 'c', 'c++', 'c#']
names.clear()
print(names)

# []
```

### 7、修改

```
names = ['java', 'python', 'c', 'c++', 'c#']
names[-1] = 'php'
print(names)

# ['java', 'python', 'c', 'c++', 'php']
```

### 8、查操作 `index` / `count`

```
names = ['java', 'python', 'c', 'c++', 'c#', 'python']
print(names.index('python')) # 返回从左开始匹配到的第一个python的索引
print(names.count('python')) # 返回元素python的个数
print('python' in names) # 判断python字符串是否在names列表中

# 1
# 2
# True
```

## 9、切片操作

```
names = ['java', 'python', 'c', 'c++', 'c#', 'php', '.net']
print(names[0:2]) # ['java', 'python']
print(names[4:-1]) # ['c#', 'php']
print(names[0:-1]) # ['java', 'python', 'c', 'c++', 'c#', 'php']
print(names[4:]) # ['c#', 'php', '.net']
print(names[0:4:2]) # 步长为2 ['java', 'c']
print(names[0:2]) # ['java', 'c', 'c#', '.net']
print(names[-1:-6:-1]) # ['.net', 'php', 'c#', 'c++', 'c']
print(names[0:]) # ['java', 'python', 'c', 'c++', 'c#', 'php', '.net']
```

## 10、反转

```
names = ['java', 'python', 'c', 'c++', 'c#', 'php', '.net']
print(names[::-1]) # ['.net', 'php', 'c#', 'c++', 'c', 'python', 'java']
print(names[::-2]) # ['.net', 'c#', 'c', 'java']
names.reverse()
print(names) # ['.net', 'php', 'c#', 'c++', 'c', 'python', 'java']
```

## 11、排序

```
nums = [4, 3, 53, 23, 44, 234, 2, 4]
nums.sort()
print(nums)

# [2, 3, 4, 4, 23, 44, 53, 234]
```

## 12、循环列表

```
names = ['java', 'python', 'c', 'c++', 'c#', 'php', '.net']
for item in names:
    print(item, ' ', end='')

# java python c c++ c# php .net
```

## 元组(tuple)

`tuple` 和 `list` 十分相似，但是 `tuple` 是不可变的，即不能修改 `tuple`，元组通过 `圆括号` 中用逗号分割的项定义。

- 支持索引和切片操作
- 可以使用 `in` 查看一个元素是否在 `tuple` 中。
- 空元组 `()`
- 只含有一个元素的元组 `("a",)`，需要加个逗号

优点: `tuple` 比 `list` 速度快; 对不需要修改的数据进行‘写保护’, 可以是代码更安全

`tuple`与`list`可以相互转换, 使用内置的函数 `list()` 和 `tuple()`。

```
l = [1, 2, 3]
print(l) # [1, 2, 3]
t = tuple(l)
print(t) # (1, 2, 3)
l = list(t)
print(l) # [1, 2, 3]
```

元组最通常的用法是用在打印语句, 如下例:

```
name = "lijing"
age = 18
print("Name: %s; Age: %d" % (name, age))

# Name: lijing; Age: 18
```

## 元组的操作

索引、长度、切片、循环 等 ---> 参照列表

## 字典(dict)

关键点:

- 键与值用冒号 `:` 分开
- 项与项用逗号 `,` 分开

特性:

- key-value结构;
- key必须是不可变类型, 唯一性;
- 可存放多个value, 可修改, 不唯一;
- 无序;
- 查询速度快, 不受dict大小影响;

## 字典的创建

方式一

```
# 直接给出key和value
info = {
    'name': 'lijing',
    'num': '202309',
    'gender': '男'
}
print(info)

# {'name': ' lijing', 'num': '202309', 'gender': '男'}
```

方式二

```
# key和value分别赋值
keys = [1, 2, 3, 4, 5]
dict = {}.fromkeys(keys)
print(dict)
dict = {}.fromkeys(keys, 100)
print(dict)

# {1: None, 2: None, 3: None, 4: None, 5: None}
# {1: 100, 2: 100, 3: 100, 4: 100, 5: 100}
```

## 字典添加键值

`d[key] = value`，如果字典中已有 `key`，则为其赋值为 `value`，否则添加新的键值对 `key:value`；

```
info = {
    'name': 'lijing',
    'num': '202309',
    'gender': '男'
}
info['age'] = 18
print(info)
info['num'] = '202308'
print(info)

{'name': 'lijing', 'num': '202309', 'gender': '男', 'age': 18}
{'name': 'lijing', 'num': '202308', 'gender': '男', 'age': 18}
```

`setdefault(key,[default])`

若字典中有key，则返回value值，若没有key，则加上该key，默认值None

```
info = {
    'name': 'lijing',
    'num': '202309',
    'gender': '男'
}
print(info.setdefault('name')) # lijing
print(info.setdefault('age')) # None
print(info) # {'name': 'lijing', 'num': '202309', 'gender': '男', 'age': None}
print(info.setdefault('address', "lianyungang")) # lianyungang
print(info) # {'name': 'lijing', 'num': '202309', 'gender': '男', 'age': None, 'address': 'lianyungang'}
```

## 字典的复制 copy()

`copy()`返回字典的一个副本(浅拷贝)

浅拷贝是指只拷贝对象的最外层，不会拷贝内层的对象。当使用赋值运算符 `=` 或者 `copy()` 方法进行浅拷贝时，两个对象会共享内部对象，也就是说，修改副本中内部对象的值会影响原始对象中的相应内部对象。

```
original = {"name": "Alice", "address": {"city": "Shanghai", "district": "Pudong"}}
shallow_copy = original.copy()
shallow_copy["name"] = "Bob"
shallow_copy["address"]["city"] = "Beijing"
print(original) # {'name': 'Alice', 'address': {'city': 'Beijing', 'district': 'Pudong'}}
print(shallow_copy) # {'name': 'Bob', 'address': {'city': 'Beijing', 'district': 'Pudong'}}
```

`dict1.update(dict2)`

把dict2的元素加入到dict1中去，键字重复时会覆盖dict中的键值

```
dict1 = {1: "one", 2: "two", 3: "three"}
dict2 = {1: "first", 4: "forth"}
dict1.update(dict2)
print(dict1) # {1: 'first', 2: 'two', 3: 'three', 4: 'forth'}
```

## 字典的删除 pop/del/clear

`pop(key, [default])`：若字典中key键存在，删除并返回 `dict[key]`，若不存在，且未给出 `default` 值，引发KeyError异常

`del d[key]`：使用 `del d[key]` 可以删除键值对

`clear()`：删除字典中所有元素

## 查操作

```
info = {
    'name': 'lijing',
    'num': '202309',
    'gender': '男'
}
print("name" in info) # True
print(info.get("name")) # lijing
print(info.get("age", 18)) # 返回字典dict中键key对应值，如果字典中不存在此键，则返回default的值
                             (default默认值为None)
print(info.keys()) # dict_keys(['name', 'num', 'gender'])
print(info.values()) # dict_values(['lijing', '202309', '男'])
print(info.items()) # dict_items([('name', 'lijing'), ('num', '202309'), ('gender', '男')])
```

## 字典的遍历

```
info = {
    'name': 'lijing',
    'num': '202309',
    'gender': '男'
}
for item in info:
    print(item)
...
name
num
gender
...
```

```

for item in info.items():
    print(item)
'''
('name', 'lijing')
('num', '202309')
('gender', '男')
'''

for i, j in info.items():
    print(i, j)
'''
name lijing
num 202309
gender 男
'''

```

## 集合(set)

- 里面的元素不可变，不能在集合中存放列表或字典，而字符串、元组、数字等不可变类型可以存放
- 天生去重，在集合中无法存放相同的元素
- 无序。不能像列表一样通过索引来标记其元素在列表中的位置，例如{1, 2, 3}和{2, 1, 3}是同一个集合。

## 集合的创建

```

info = {'lijing', 18, '男'}
print(info) # {18, 'lijing', '男'}
print(type(info)) # <class 'set'>

```

## 列表转集合

```

list = ['lijing', 18, '男']
info = set(list)
print(info) # {18, 'lijing', '男'}
print(type(info)) # <class 'set'>

```

## 集合的新增add

```

info = {'lijing', 18, '男'}
info.add('python')
print(info) # {'lijing', 18, '男', 'python'}

```

## 集合的删除discard/remove/pop

```

info = {'lijing', 18, '男'}
# remove方法用于删除集合中指定的元素,如果该元素不存在于集合中,则会抛出KeyError异常。
# discard方法也用于删除集合中指定的元素,但是如果该元素不存在于集合中,则不会抛出异常,而是直接忽略。
info.discard('lijing')
print(info) # {'男', 18}
info.remove(18)
print(info) # {'男'}
info.pop()
print(info) # set()

```

## 集合的遍历

```
info = {'lijing', 18, '男'}
for item in info:
    print(item)

# 18
# lijing
# 男
```

## 序列

序列类型是指容器内的元素从0开始的索引顺序访问，一次可以访问一个或者多个元素；列表、元组和字符串都是序列。

序列的三个主要特点是

- 索引操作符和切片操作符
- 索引可以得到特定元素
- 切片可以得到部分序列

```
numbers = ["zero", "one", "two", "three", "four"]

print(numbers[1]) # one
print(numbers[-1]) # four
print(numbers[:]) # ['zero', 'one', 'two', 'three', 'four']
print(numbers[3:]) # ['three', 'four']
print(numbers[:2]) # ['zero', 'one']
print(numbers[2:4]) # ['two', 'three']
print(numbers[1:-1]) # ['one', 'two', 'three']
```

切片操作符中的第一个数（冒号之前）表示切片开始的位置，第二个数（冒号之后）表示切片到哪里结束。

如果不指定第一个数，Python就从序列首开始。如果没有指定第二个数，则Python会停止在序列尾。

注意，返回的序列从 开始位置开始 ，刚好在结束 位置之前结束 。即 开始位置是包含在序列切片中的，而结束位置被排斥在切片外 。

可以用负数做切片。负数用在从序列尾开始计算的位置。

## Python-判断和循环

### 判断语句

#### if判断语句

该语句的语法格式如下：

```
# 单条件判断：
if condition_1:
    result_1
else:
```



```
result_2

# 多条件判断:
if condition_1:
    result_1
elif:
    result_2
elif:
    result_3
else:
    result_4
```

## 循环语句

### while 循环

```
while 判断条件:
    执行语句
```

### continue、break的用法

- continue是终止本次循环
- break是终止循环

### for语句

for循环可以遍历任何序列的所有元素

```
for 变量 in 序列:
    执行语句
```

## Python-函数

---

### 函数定义的一般格式

```
# 函数定义的一般格式
def 函数名 (参数列表):
    函数体

# 实例
def hello():
    print('Hello World')

hello()
# Hello World
```

## 函数调用

```
# 函数功能：打印该字符串
def println(str):
    print(str)
    return

# 调用函数
println('调用上述函数')
println('打印这段字符串')
println('多次调用函数')

# 调用上述函数
# 打印这段字符串
# 多次调用函数
```

## 默认参数

- 函数定义时，默认参数必须在位置形参的后面。
- 函数调用时，指定参数名的参数，叫关键参数。
- 而在函数定义时，给参数名指定值的时候，这个参数叫做默认参数。
- 关键参数，和默认参数两个参数写法一样，区别在于：  
关键参数是在函数调用时，指定实参的参数名，也可以说指定值的参数名。  
默认参数是在函数定义时，指定参数名的值。

写法：

```
def (a, b=100):
    pass
```

定义时，有默认参数的话，调用时，这个实参可以不写。如果实参不写的话，这个形参的参数值是他的默认值。

## 动态参数

当需要一个函数能处理很多参数，超过已声明的参数数量，这时就需要动态参数。

与上述两中参数不同的是，该参数声明不需要命名。

**\*args**

一个'\*'的参数会以元组(tuple)的形式导入，存放未命名的变量参数

```
# 函数定义
def print_info(arg1, *var_tuple):
    print('输出: ')
    print(arg1)
    print(var_tuple)

# 调用
print_info(34, 45, 32, 12)
```

```
# 输出:
# 34
# (45, 32, 12)

# 如果函数调用时没有指定参数，动态参数则表现为空元组。
```

## **\*\*kwargs**

还有一种动态参数，加了两个星号则以字典的形式导入

```
# 函数定义
def print_info(arg1, **var_dict):
    print('输出: ')
    print(arg1)
    print(var_dict)

# 调用
print_info(34, a=45, b=32, c=12)

# 输出:
# 34
# {'a': 45, 'b': 32, 'c': 12}
```

## **匿名函数**

- python可以使用lambda来创建匿名函数
- 所谓匿名，即不再使用def这样的标准语句来专门定义函数
- lambda的主体是一个表达式，而不是一个代码块。仅仅能在lambda中封装有限的逻辑进去。
- lambda的语法只包含一个语句，格式如下：

```
lambda [arg1[,arg2,...,argN]]:expression
```

```
# 用def格式写
def function1(x, y):
    return x * y

print(function1(2, 3)) # 6

# 用匿名函数写
function2 = lambda x, y: x * y

print(function2(3, 4)) # 12
```

## **Python-类和对象**

---

## 定义类

```
class 类名(object):  
    n个类属性...  
    n个类方法...
```

- **class**：必须的关键字
- **类名**：自定义名称，遵循标准驼峰写法
- **括号**：python3表示继承某个类
- **object**：超类，继承所有

例如：

```
class MyClass:  
  
    name = "lijing"  
  
    def hello(self):  
        return 'hello python'
```

## 类构造方法

Python 类有一个名为 `__init__()` 的方法，该方法是一个特殊的类实例方法，称为构造方法（或构造函数）。

手动添加构造方法的语法规则如下：

```
def __init__(self,...):  
    代码块
```

注意：

- 方法名开头和结尾各有 2 个下划线，且中间不能有空格。
- `__init__()` 方法可以包含多个参数，但必须包含一个名为 self 的参数，且必须作为第一个参数。
- 即便不手动为类添加任何构造方法，Python 也会自动为类添加一个仅包含 self 参数的默认构造方法。

## self 参数

Python 只是规定，无论是构造方法还是实例方法，最少要包含一个参数，并没有规定该参数的具体名称。

之所以将其命名为 self（self 不是 python 关键字），只是程序员之间约定俗成的一种习惯，遵守这个约定。

self 参数的具体作用：

- self 代表的是类的实例，代表当前对象的地址。而 self.class 则指向类。
- self 所表示的都是实际调用该方法的对象（即谁调用该方法，那么 self 就代表谁）。

## 类的实例化

创建类对象的过程，称为类的实例化。

对已定义好的类进行实例化，其语法格式如下：

```
类名(参数)
```

定义类时，如果没有手动添加 `__init__()` 构造方法，又或者添加的 `__init__()` 中仅有一个 `self` 参数，则创建类对象时的参数可以省略不写。

## 类对象访问变量或方法

访问类中实例变量的语法格式如下：

```
类对象名.变量名
```

调用类中方法的语法格式如下：

```
类对象名.方法名(参数)
```

## 类方法和类静态方法

类方法需要在方法上加上 `@classmethod`

类静态方法需要在方法上加上 `@staticmethod`

一般来说，要使用某个类的方法，需要先实例化一个对象再调用方法。

而使用 `@staticmethod` 或 `@classmethod`，就可以不需要实例化，直接 `类名.方法名()` 来调用。

### classmethod(类方法)

使用 `@classmethod` 装饰器定义的方法在类和实例之间共享。

类方法的第一个参数通常被约定为 `cls`，它表示类本身，而不是实例。

通过类方法，可以访问类的属性和调用其他类方法。类方法可以通过类或实例进行调用。

类方法通常用于执行与类相关的操作，而不依赖于具体的实例。

### staticmethod(静态方法)

使用 `@staticmethod` 装饰器定义的方法不与类或实例绑定，它们是类中的普通函数。

静态方法与类方法不同，它们不能访问类的属性或调用其他类方法。

静态方法与类和实例无关，可以通过类或实例进行调用。

静态方法通常用于执行与类和实例无关的操作，它们可以在类的内部作为一种组织和封装代码的方式。

关键区别：

`classmethod` 可以访问和修改类的属性，可以调用其他类方法，第一个参数为类本身(通常命名为 `cls`)。

`staticmethod` 不能访问或修改类的属性，也不能调用其他类方法，它与类和实例无关。

在选择使用`classmethod`还是`staticmethod`时，要根据具体情况考虑方法是否需要访问或修改类的属性，以及是否需要调用其他类方法。

如果需要访问类的属性或调用其他类方法，则应选择`classmethod`。如果方法与类和实例无关，则可以选择`staticmethod`。

## Python-模块和包

### 模块导入

想使用 Python 源文件，只需在另一个源文件里执行 `import` 语句，语法如下：

```
import 模块名字 # 导入整个模块，这种导入方式比较占用内存

import 模块名字 as xx # 这里是导入整个模块的同时给它取一个别名，因为有些模块名字比较长，用一个缩写的别名代替在下次用到它时就比较方便

from 模块名字 import 方法 # 从一个模块里导入方法，要用到模块里的什么方法就从这个模块里导入那个方法，这样占用的内存就比较少，当然也能用别名

from package.modules import 方法 # 从一个包的模块里导入方法 这个方法跟上面那种基本一样，占用的内存也比较少

from 模块名字 import *
# 表示导入模块中所有的不是以下划线(_)开头的名字都导入到当前位置
# 大部分情况下Python程序不应该使用这种导入方式，因为*不知道导入什么名字，
# 很有可能会覆盖掉之前已经定义的名字。而且可读性极其的差，在交互式环境中导入时没有问题
# 如果使用* 的方式进行了导入，这时只想使用里面的某个或某些功能时，可以使用__all__来进行约束
# __all__只是用来约束* 方式的，其他方式导入的话，不会生效
```

### 包的导入操作

包的导入分为 `import` 和 `from...import...` 两种

但是无论哪种方式，在导入时必须都遵循一个原则：凡是在导入时带点的，点的左边必须是一个包，否则非法

包的本质就是文件夹，导入包就相当于导入包下的 `__init__.py` 文件

模块(module)其实就是py文件，里面定义了一些函数、类、变量等。

包(package)是多个模块的聚合体形成的文件夹，里面可以是多个py文件，也可以嵌套文件夹。

包的 `__all__` 定义在 `__init__.py` 文件中，模块的 `__all__` 定义在模块文件的开头

# 基本操作与格式规范

## 命名规范

变量和标识符，变量和标识符在命名规则上都遵循以下原则

- 只能由字母、数字和下划线组成，且不能以数字开头
- 命名要做到见名知意：一般来说，只要是自定义的名字都可以被称为标识符，包括变量、函数名、类名、模块名、包名，所以变量其实只是标识符的一个子类。

变量常见的命名风格有三种：

- 单词全部纯小写，单词之间使用下划线隔开，例如：data\_science
- 小驼峰式命名，第一个单词字母全小写，其他单词首字母大写，例如：dataScience
- 大驼峰式命名，每个单词的首字母均大写，例如：DataScience
- 以上这些命名规范不仅仅适用于变量，同时也适用于函数、类等；
- 在Python中，函数和变量名一般采用第一种方式，类名采用第三种方式，第二种方式在Java声明方法时比较常见。
- 切记，标识符的名字不能和Python的关键字冲突(可以使用keyword.kwlist查看Python所有的关键字)

## 格式化输出

拼接符 '+'

注意：使用+，则变量必须为字符串类型

```
print('变量名: ' + 变量)
```

拼接符 ','

```
print('变量名: ', 变量)
```

## 格式化符号

### 整数输出：

%o — oct 八进制

%d — dec 十进制

%x — hex 十六进制

```
print('%o' % 20)    # 以八进制输出20
print('%d' % 20)    # 以十进制输出20
print('%x' % 20)    # 以十六进制输出20

# 24
# 20
# 14
```

### 浮点数输出

```
print('%f' % 1.11)  # 默认保留6位小数
print('%.1f' % 1.11) # 取1位小数
print('%e' % 1.11)  # 默认6位小数，用科学计数法
print('%.3e' % 1.11) # 取3位小数，用科学计数法
print('%g' % 1111.1111) # 默认6位有效数字
print('%.7g' % 1111.1111) # 取7位有效数字
print('%.2g' % 1111.1111) # 取2位有效数字，自动转换为科学计数法

# 1.110000
# 1.1
# 1.110000e+00
# 1.110e+00
# 1111.11
# 1111.111
# 1.1e+03
```

## format的用法

format的用法很常见，并且实用。

format就是变量之间的映射，它有三种形式

- （1）不带编号，即“{}”
- （2）带数字编号，可调换顺序，即“{1}”、“{2}”
- （3）带关键字，即“{a}”、“{tom}”

```
print('{} {}'.format('hello','world')) # 不带字段
print('{0} {1}'.format('hello','world')) # 带数字编号
print('{0} {1} {0}'.format('hello','world')) # 打乱顺序
print('{1} {1} {0}'.format('hello','world'))
print('{a} {b} {a}'.format(b='hello',a='world')) # 带关键字

# hello world
# hello world
# hello world hello
# world world hello
# world hello world
```



## 格式化字符串

- f-string, 亦称为格式化字符串常量 (formatted string literals), 是Python3.6新引入的一种字符串格式化方法。
- f-string在形式上是以 f 或 F 修饰符引领的字符串 (f'xxx' 或 F'xxx'), 以大括号 {} 标明被替换的字段;
- f-string在本质上并不是字符串常量, 而是一个在运行时运算求值的表达式。

先尝试一下str、int类型的变量

```
name = "lijing"
age = 18
print(f"姓名: {name}, 年龄: {age}")

# 姓名: lijing, 年龄: 18
```

再试一下字典

```
one_dict = {"name": "lijing", "age": 18, "hobby": ["running", "singing"]}
print(f"姓名: {one_dict['name']}, 爱好: {one_dict['hobby']}")

# 姓名: lijing, 爱好: ['running', 'singing']
```

## 断言

Python 的断言语句是一种调试辅助功能, 不是用来处理运行时错误的机制。

assert 在条件为 False 的时候触发, 后面的内容是报错信息。

```
import sys
assert sys.version_info >= (3, 7), "请在Python3.7及以上环境执行"
```

如果这个项目要求最低是 Python3.7 的环境, 那么如果使用 Python3.6 来运行这个项目, 就会出现这个错误信息。

## Python中的下划线

Python有很多地方使用下划线。在不同场合下, 有不同含义: 比如 `_var` 表示内部变量; `__var` 表示私有属性; `__var__` 表示系统 (魔术) 方法; 这些含义有的是程序员群体的约定, 如 `_var`; 有的是Python解释器规定的形式, 如 `__var`。

目前常见的用法有五种:

- `_` 用于临时变量
- `var_` 用于解决命名冲突问题
- `_var` 用于保护变量
- `__var` 用于私有变量
- `__var__` 用于系统 (魔术) 方法

### for循环中的\_

for循环中 `_` 作为临时变量用。下划线来指代没什么意义的变量。

例如在如下函数中, 只关心函数执行次数, 而不关心具体次序的情况下, 可以使用 `_` 作为参数。

```
nums = 13
for _ in range(nums):
    ...
```

## 元组拆包中的\_

元组拆包，赋值的时候可以用\_来表示略过的内容。

如下代码忽略北京市人口数，只取得名字和区号。

```
city, _, code = ('Beijing', 21536000, '010')
print(city, code)

# Beijing 010
```

如果需要略过的内容多于一个的话，可以使用\*开头的参数，表示忽略多个内容。

如下代码忽略面积和人口数，只取得名字和区号

```
city, *_ , code = ('Beijing', 21536000, 16410.54, '010')
print(city, code)

# Beijing 010
```

## 大数字表示形式

\_ 也可用于数字的分割，这在数字比较长的时候常用。

```
a = 9_999_999_999
print(a)

# 9999999999
```

a的值自动忽略了下划线。这样用\_分割数字，有利于便捷读取比较大的数。

## var\_用于解决命名冲突问题

变量后面加一个下划线。主要用于解决命名冲突问题，编程中遇时Python保留的关键字时，需要临时创建一个变量的副本时，都可以使用这种机制。

```
def type_obj_class(name, class_):
    pass
```

以上代码中出现的 `class` 是Python的保留关键字，直接使用会报错，使用下划线后缀的方式解决了这个问题。

## \_var用于保护变量

如果使用通配符从模块中导入所有名称，则Python不会导入带有前导下划线的名称（除非模块定义了覆盖此行为的 `--all--` 列表）：

前面一个下划线，后面加上变量，这是仅供内部使用的“保护变量”。比如函数、方法或者属性。

这种保护不是强制规定，而是一种程序员的约定，解释器不做访问控制。

一般来讲这些属性都作为实现细节而不需要调用者关心，随时都可能改变，编程时虽然能访问，但是不建议访问。

这种属性，只有在导入时，才能发挥保护作用。而且必须是 `from XXX import *` 这种导入形式才能发挥保护作用。

比如在下例汽车库函数tools.py里定义的“保护属性”：发动机型号和轮胎型号，这属于实现细节，没必要暴露给用户。

当使用 `from tools import *` 语句调用时，其实际并没有导入所有 `_` 开头的属性，只导入了普通 `drive` 方法。

```
_moto_type = 'xxx'
_wheel_type = 'xxx'

def drive():
    _start_engine()
    _drive_wheel()

def _start_engine():
    print('start engine %s'%_moto_type)

def _drive_wheel():
    print('drive wheel %s'%_wheel_type)
```

### 突破保护属性

之所以说是“保护”并不是“私有”，是因为Python没有提供解释器机制来控制访问权限。依然可以访问这些属性：

```
import tools
tools._moto_type = 'xxx'
tools.drive()
```

以上代码，已越过“保护属性”。

此外，还有两种方法能突破这个限制，一种是将“私有属性”添加到tool.py文件的 `__all__` 列表里，使 `from tools import *` 也导入这些本该隐藏的属性。

```
__all__ = ['drive', '_moto_type', '_wheel_type']
```

另一种是导入时指定“受保护属性”名。

```
from tools import drive, _start_engine
_start_engine()
```

甚至是，使用 `import tools` 也可以轻易突破保护限制。

所以可见，“保护属性”是一种简单的隐藏机制，只有在 `from tools import *` 时，由解释器提供简单的保护，但是可以轻易突破。

这种保护更多地依赖程序员的共识：不访问、修改“保护属性”。

除此之外，有没有更安全的保护机制呢？

有，就是下一部分讨论的私有变量。

## \_\_var用于私有变量

私有属性解决的之前的保护属性保护力度不够的问题。

变量前面加上两个下划线，类里面作为属性名和方法都可以。

两个下划线属性由 Python的改写机制 来实现对这个属性的保护。

看下面汽车例子中，品牌为普通属性，发动机为“保护属性”，车轮品牌为“私有属性”。

```
class Car:
    def __init__(self):
        self.brand = 'xxx'
        self._moto_type = 'xxx'
        self.__wheel_type = 'xxx'

    def drive(self):
        print('Start the engine %s, drive the wheel %s, I get a running %s car'%
              (self._moto_type,
               self.__wheel_type,
               self.brand))
```

实例化一个car，用 `var(car)` 查看下具体属性值

```
['_Car__wheel_type', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_moto_type', 'brand', 'drive']
```

可见，实例化 `car` 中，普通属性 `self.brand` 和保护属性 `self._moto_type` 都得以保存，两个下划线的私有属性 `__wheel_type` 没有了。

取而代之的是 `_Car__wheel_type` 这个属性。

这就是改写机制（Name mangling）。

两个下划线的属性，被改写成带有类名前缀的变量，这样子类很难命名一个和如此复杂名字重名的属性，保证了属性不被重载，保证了其的私有性。

### 突破私有属性

这里“私有变量”的实现，是从解释器层面给与的改写，保护了私有变量。

但是这个机制并非绝对安全，依然可以通过 `obj._ClassssName__private` 来访问 `__private` 私有属性。

```
car.brand = 'xxx'
car._moto_type = 'xxx'
car._Car__wheel_type = 'xxx'
car.drive()
```

## \_\_var\_\_ 用于系统（魔术）方法

变量前面两个下划线，后面两个下划线。

这是Python当中的系统（魔术）方法，一般是给系统程序调用的。

例如上例中的 `__init__` 就是类的初始化系统（魔术）方法。

指的是一般都是python自身调用的方法，即系统方法，程序猿是不应该直接调用这一类方法的。

总结

模式	举例	含义
单前导下划线	<code>_var</code>	命名约定，仅供内部使用。通常不会由Python解释器强制执行（通配符导入除外），只作为对程序员的提示。
单末尾下划线	<code>var_</code>	按约定使用以避免与Python关键字的命名冲突。
双前导下划线	<code>__var</code>	当在类上下文中使用时，触发"名称修饰"。由Python解释器强制执行。
双前导和双末尾下划线	<code>__var__</code>	表示Python语言定义的特殊方法。避免在你自己的属性中使用这种命名方案。
单下划线	<code>_</code>	有时用作临时或无意义变量的名称（"不关心"）。也表示Python REPL中最近一个表达式的结果。