

Python编程基础

2023/2/27

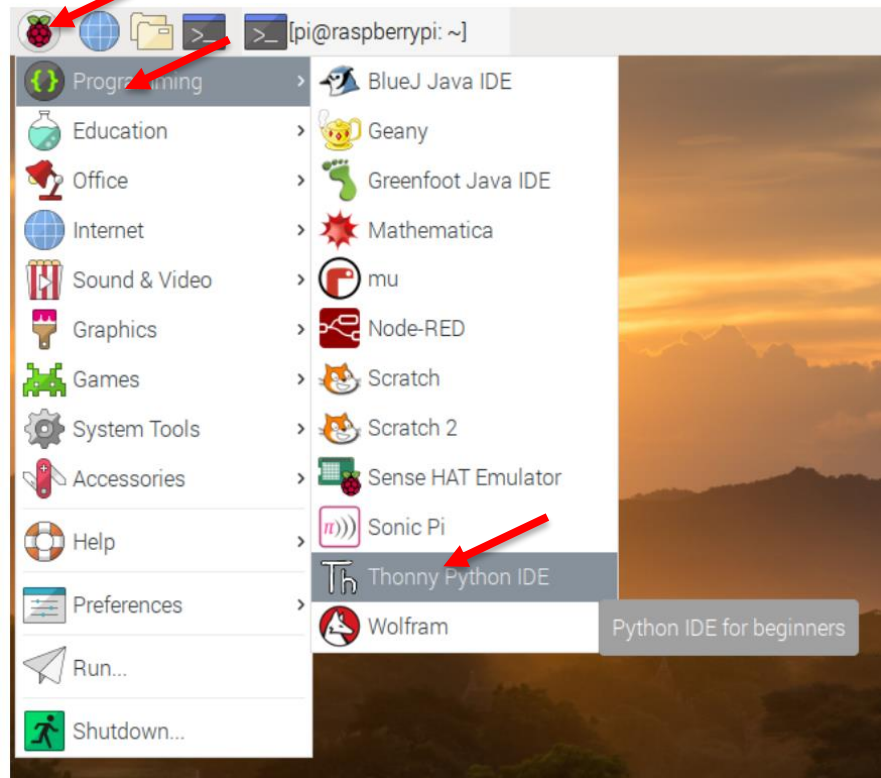
电子系统导论教学团队

Python 在树莓派中的使用

看一看课堂上在哪里进行python编程——
Raspberry Pi OS

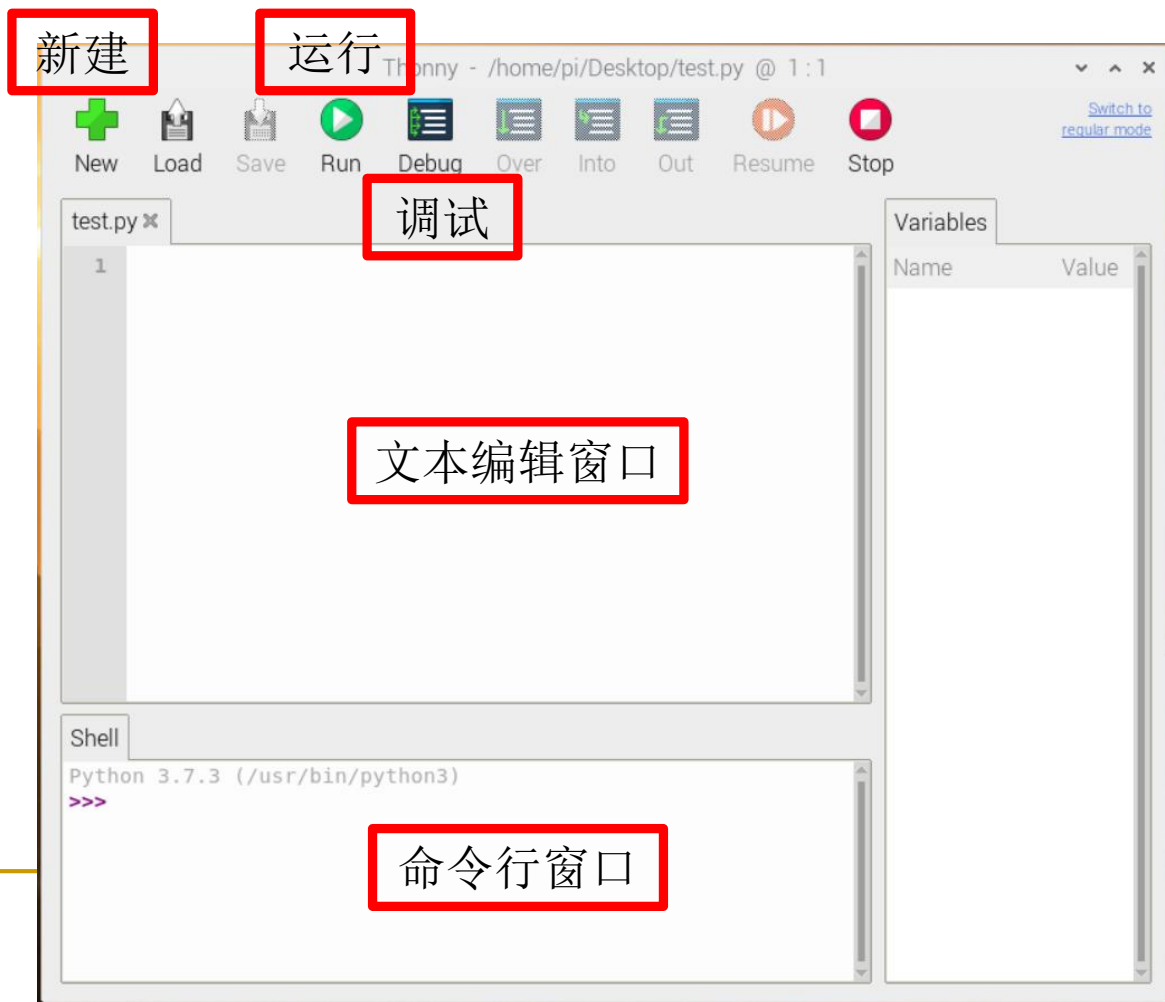
树莓派系统里Python IDE的使用

- 在Programming界面可以看到很多IDE
- Thonny是树莓派4自带Python的集成开发环境
- 建议将Thonny设为Python脚本文件的默认打开方式，方便日后的开发和学习。
- 注意，保证每次只有一个IDE环境在运行，同时保证只有一个程序在运行！



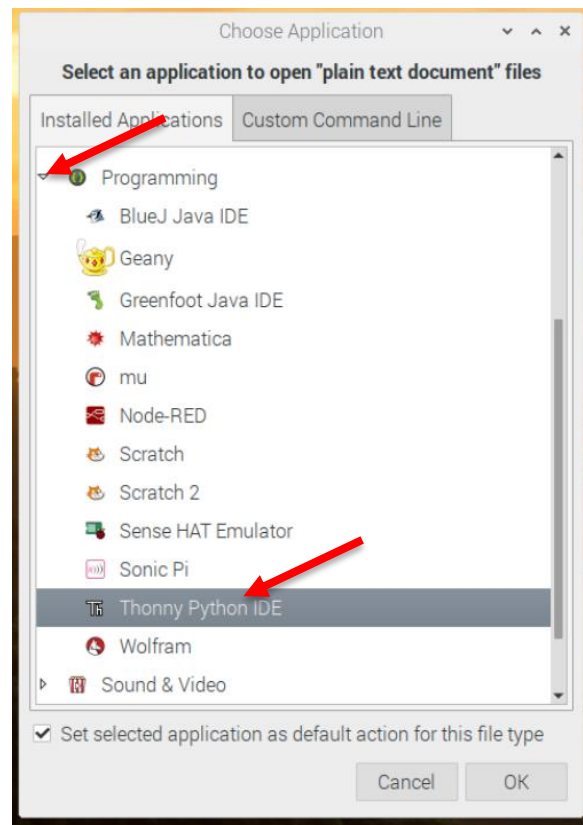
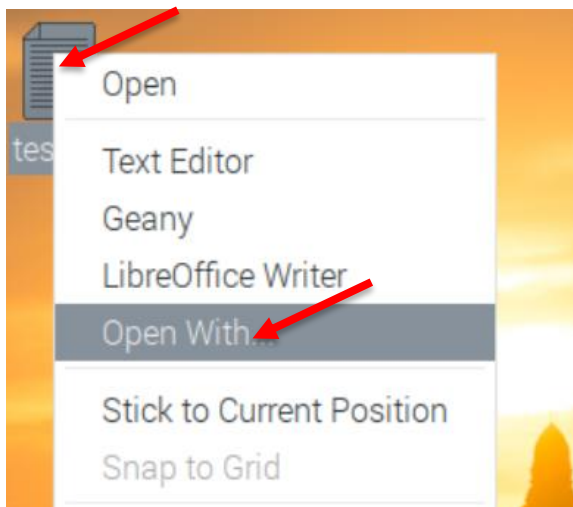
Thonny的使用

- 打开后，选择新文件，进入程序文本编辑器



Thonny的使用

- 最后我们将Thonny设置为python脚本的默认打开方式
- 右键选择任意一个.py程序脚本文件
- 选择Open With-Open With...
- 在窗口中选择Thonny Python IDE
- 以后双击.py文件即用Thonny打开



Python PC端开发环境搭建

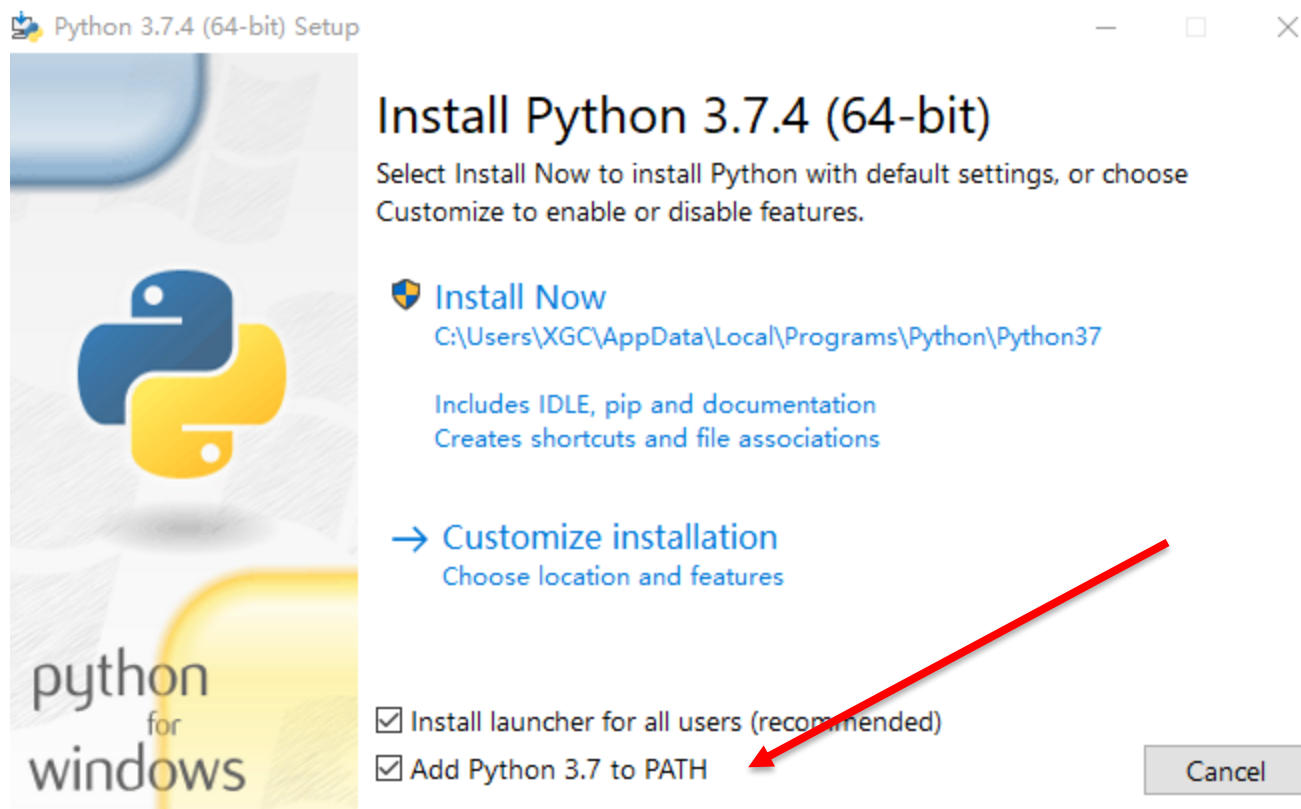
看一看自己的电脑怎么进行python开发——
Windows & Ubuntu & Mac OS

Python 安装（Windows）：方法一

- 打开浏览器访问<http://www.python.org/download/>
- 在下载列表中选择Window平台安装包，包格式为：
python-3.7.X.msi 文件（3.7.X 为要安装的版本号）
- 要使用安装程序 python-3.7.X.msi, Windows系统必须支持Microsoft Installer 2.0搭配使用。Windows XP和更高版本已经有MSI，很多老机器也可以安装MSI。
- 下载后，双击下载包，进入Python安装向导，只需使用默认设置一直点击"下一步"直到安装完成即可。

Python 安装（Windows）：方法一

- 安装过程中特别要注意选上Add python 3.7 to PATH



Python 安装（Windows）：方法一

- Python默认会安装到C:\Users\用户\AppData\Local\Programs\Python\Python37目录下，然后打开命令提示符窗口，输入python后，出现图示窗口，说明安装成功。

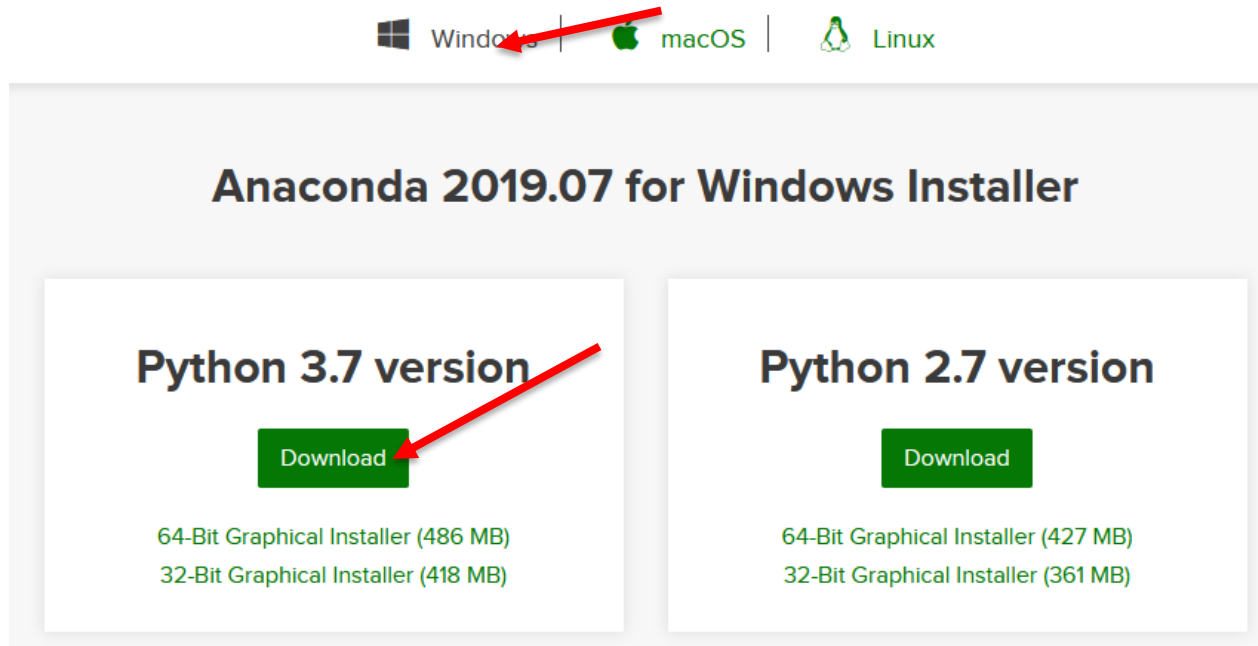
（也可直接在对应安装目录中打开交互式界面）

 C:\Users\XGC\AppData\Local\Programs\Python\Python37\python.exe

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

Python 安装（Windows）：方法二

- 浏览器访问<https://www.anaconda.com/distribution/>
在下载界面中选择Window平台、Python 3.7 version



- 下载后，双击下载包，进入Python安装向导，只需使用默认设置一直点击"下一步"直到安装完成即可。

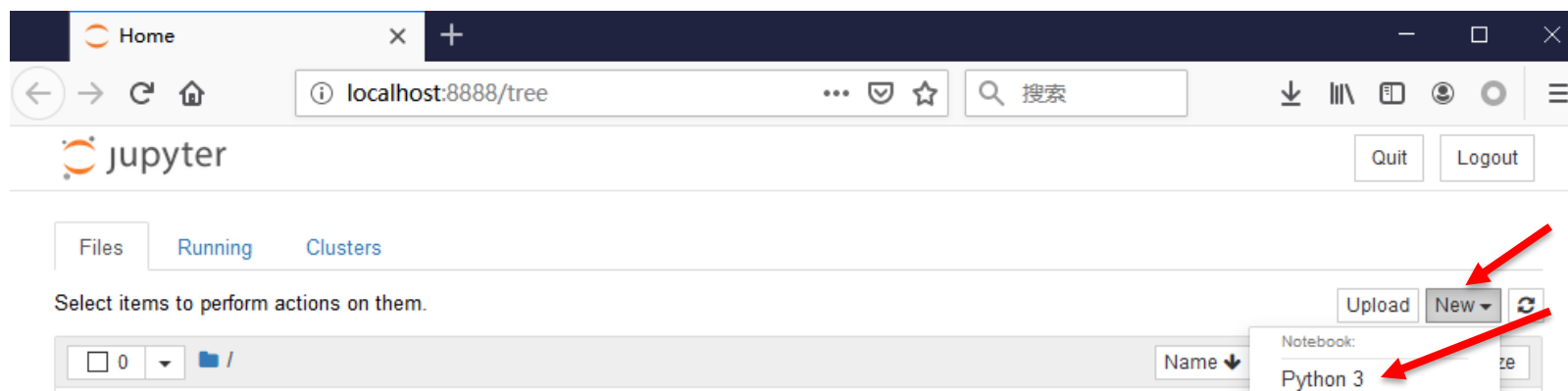
Python 安装（Windows）：方法二

- 安装完成之后，启动Anaconda Prompt，输入python，出现图示窗口

```
Anaconda Prompt - python

(base) C:\Users\XGC>python
Python 3.7.3 (default, Mar 27 2019, 17:13:21) [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- 或者启动Jupyter Notebook，在浏览器上使用python



Python 集成开发环境

- Python 可直接在交互式界面编写代码，但是编写的代码无法保存。使用集成开发环境（IDE: Integrated Development Environment）：



Spyder: 若安装了Anaconda，安装成功后就自带的IDE（推荐）



PyCharm edu在Windows和Ubuntu上均可使用，下载地址：<https://www.jetbrains.com/pycharm-edu/>

Python集成开发环境

- PyCharm edu在Windows下的安装不做过多说明。
- Ubuntu软件中心无PyCharm edu，需进入PyCharm官网下载。

Python3 安装（Linux）

- 以Ubuntu为例；
- 初次使用Ubuntu系统可能因操作失误使系统出现问题，所以建议先安装虚拟机VMware，再在虚拟机上安装Ubuntu系统；
- 在VMware上安装好Ubuntu系统后，安装VMware Tools，便可实现主机和VMware间文件传送和文字内容的复制粘贴。（后续会用到）

Python3 安装（Linux）

- Ubuntu14.04及之后的版本上自带了Python3，打开终端，输入python3（此处为小写），出现图示内容，即证明已安装Python3

```
Python 3.6.8 (default, Aug 20 2019, 17:12:48)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Python3 安装（Linux）

- 如果安装了较早版本的Ubuntu版本，则需要自行安装Python
- Python可在ubuntu自带的软件中心中安装（可以避免版本不兼容等问题）

Ubuntu安装Python集成开发环境

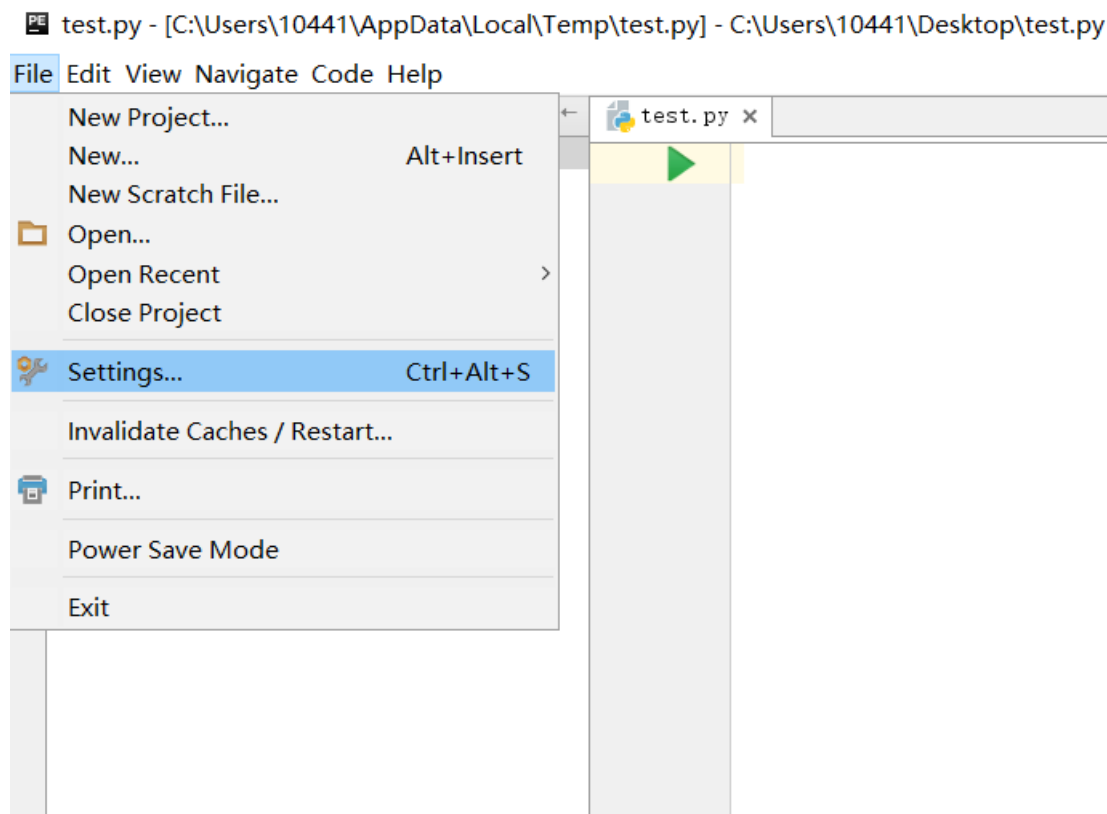
- PyCharm下载完成后，进入文件下载目录，右击文件，选择“*提取到此处*”，会生成一个文件夹；
- 进入该文件夹下的*bin*目录；
- 右击空白处，选择“*在终端打开*”；
- 在终端输入“*sh ./pycharm.sh*”即可打开PyCharm。

Python3 安装（Mac）

- 最近的Mac系统都自带Python环境。
- 也可以在链接 <http://www.python.org/download/> 上下载最新版安装。

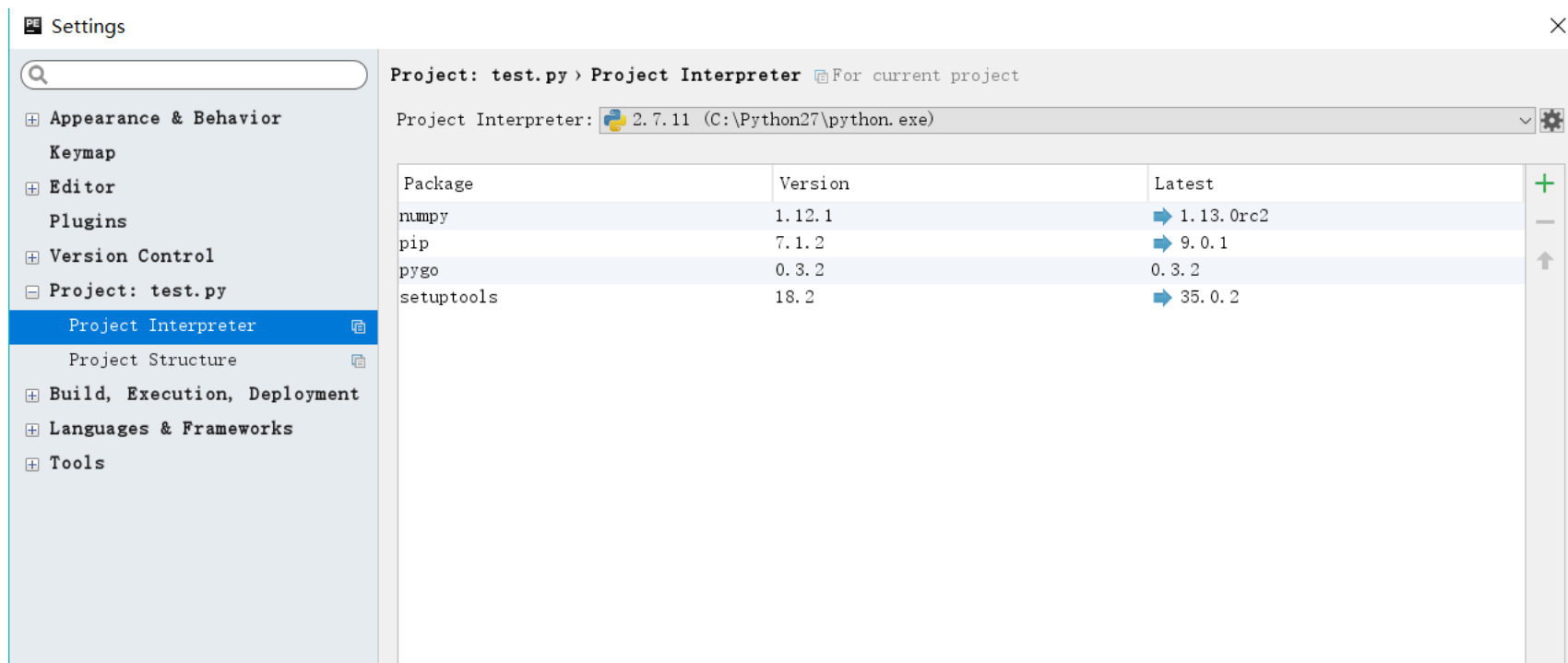
Python中第三方包（package）的安装方法

- Pycharm中可直接安装第三方的包。
- 如图，选择file目录下的settings

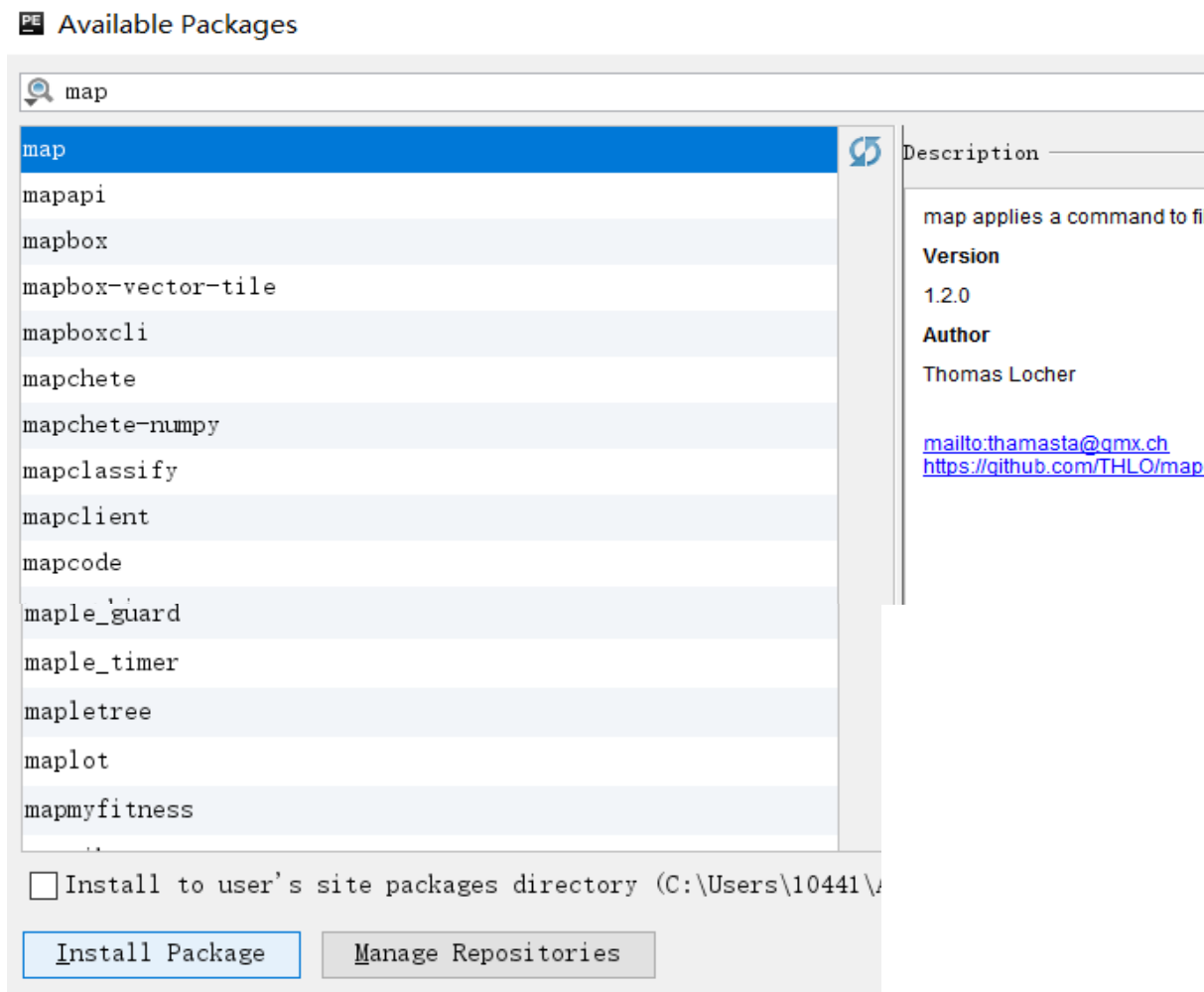


■ setting栏中选择Project Interpreter

会显示当前已安装的包， 点击最右侧的“+”号添加新的包。



- 输入需要安装的包的名称，找到后点击最下方Install Package

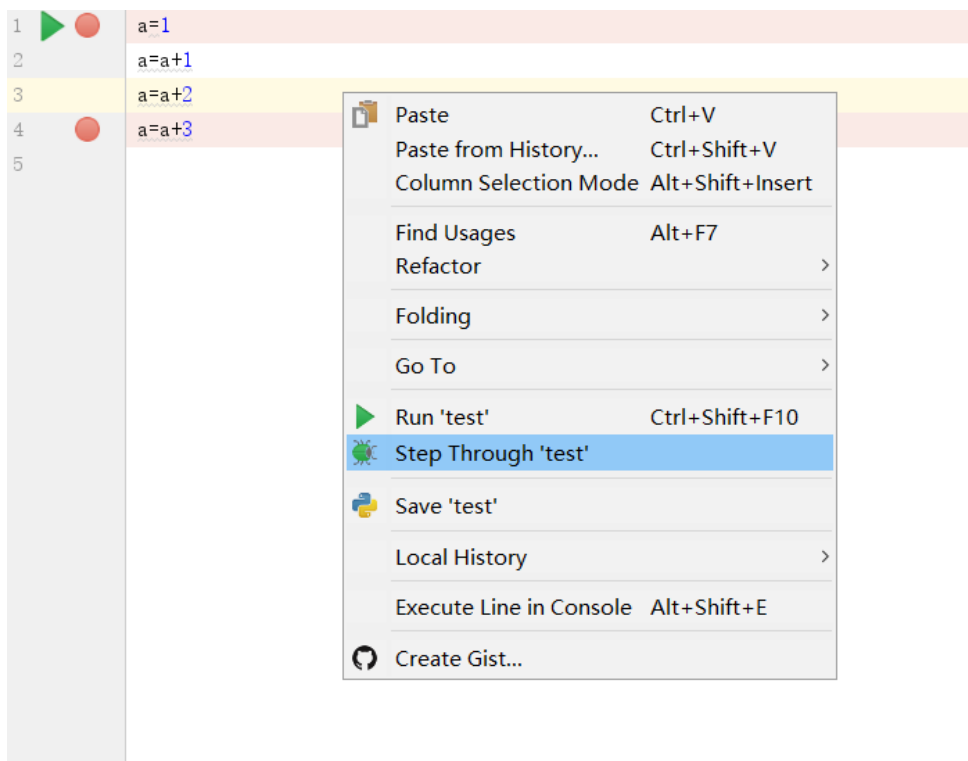


终端里使用pip安装第三方包

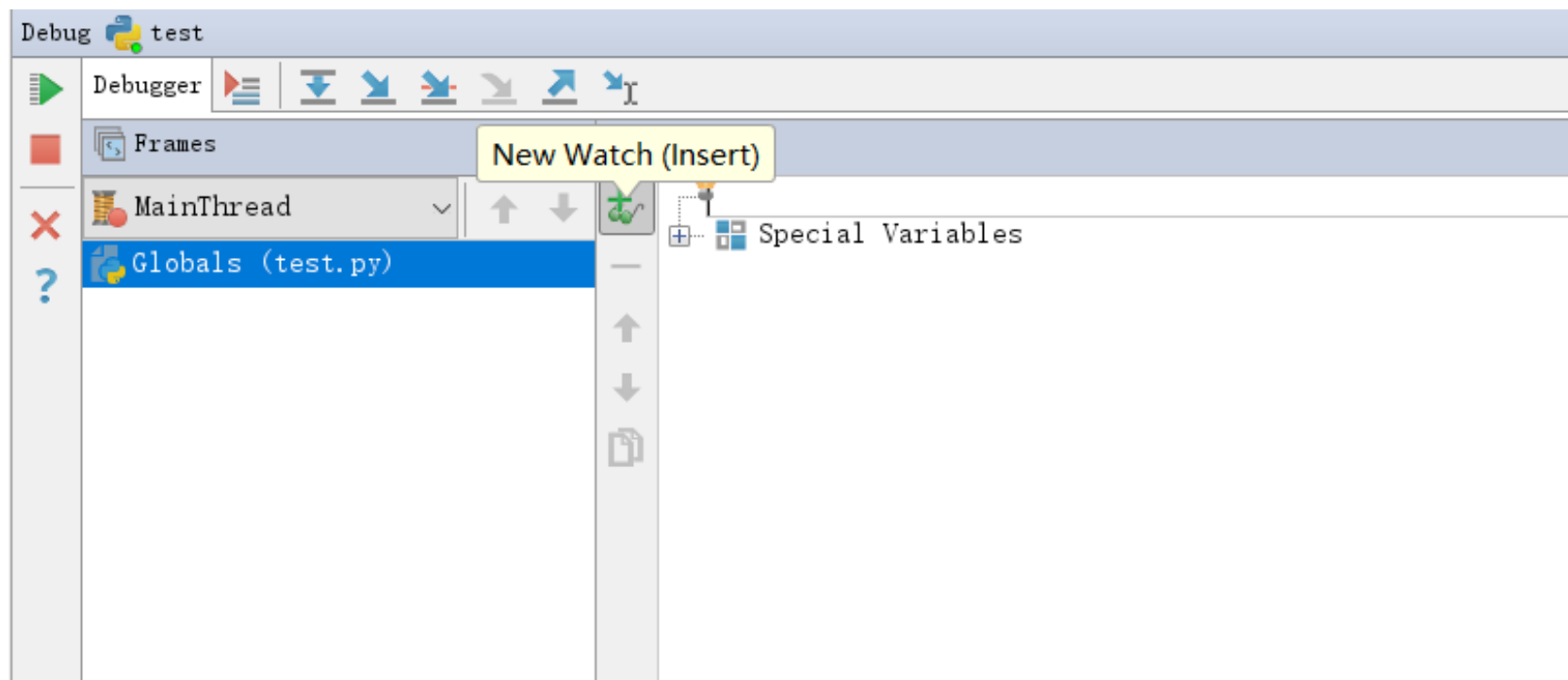
- 安装: `pip install PackageName`
- 更新: `pip install -U PackageName`
- 移除: `pip uninstall PackageName`
- 搜索: `pip search PackageName`
- 帮助: `pip help`
- 参考链接: <http://blog.sciencenet.cn/blog-442719-863840.html>

Pycharm中添加断点并调试

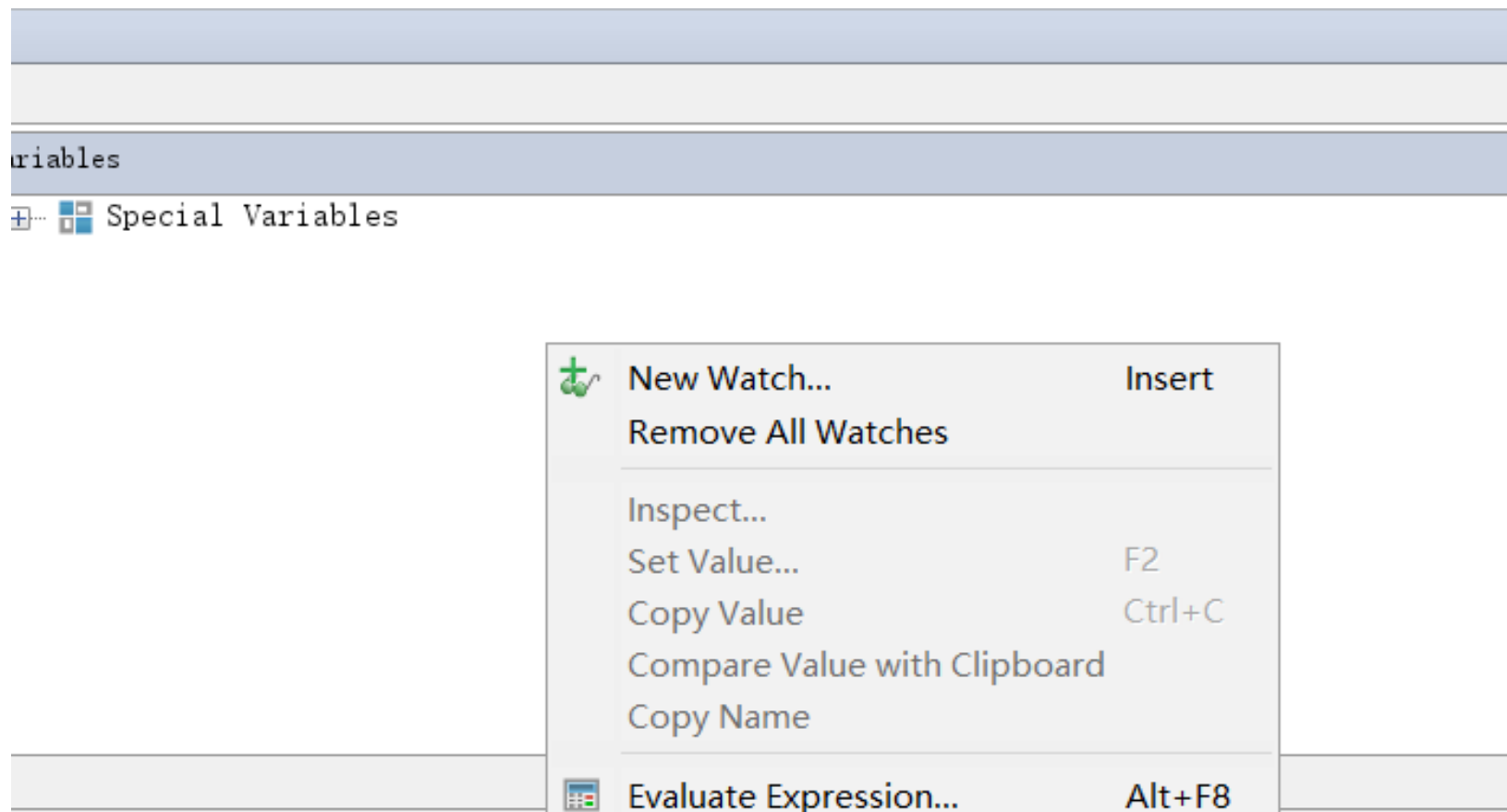
- 1.左击代码行左端添加断点；
- 2.右击代码界面选择Step Through



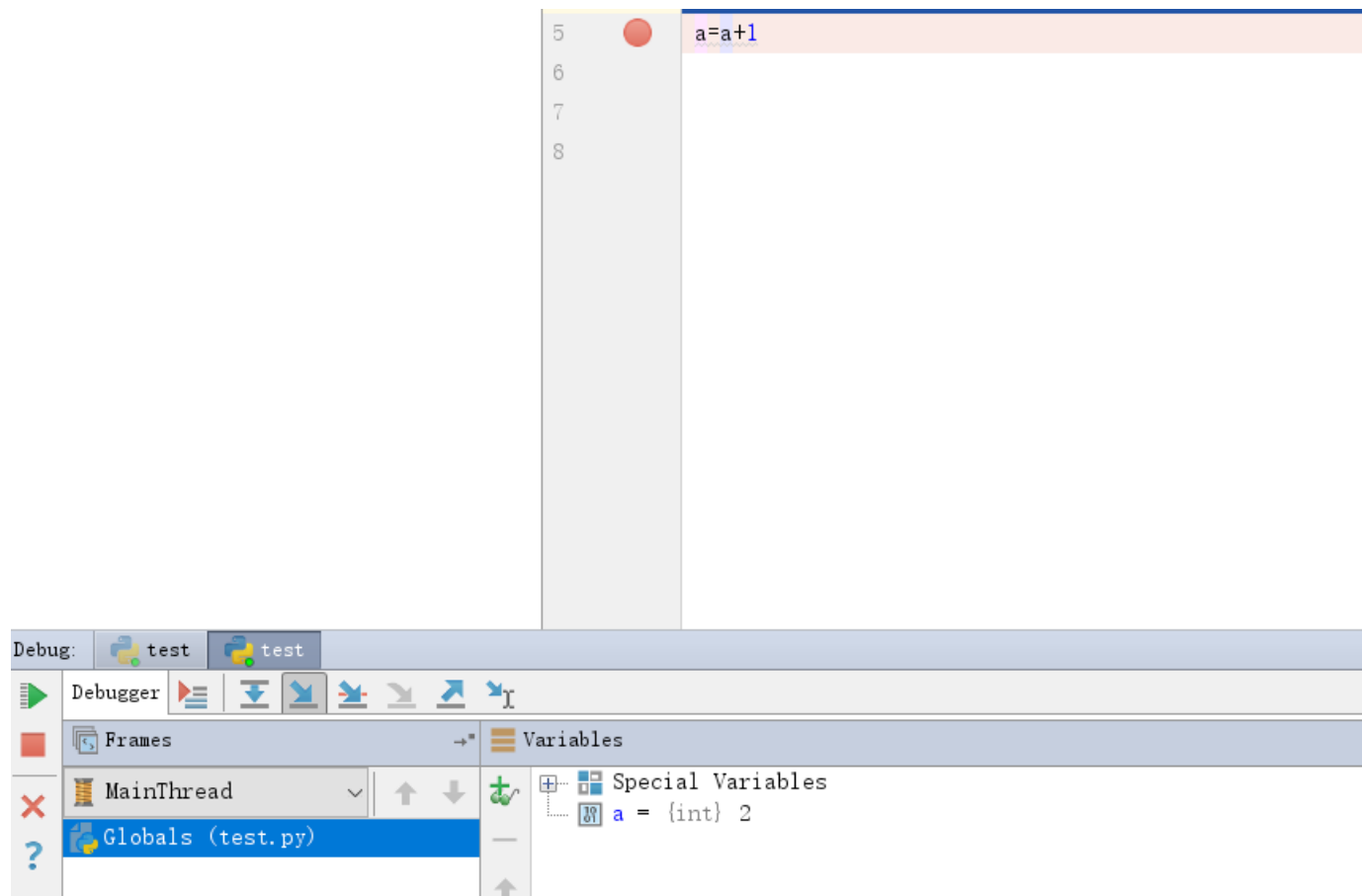
- 3.左击最下方debug界面New Watch（insert）添加要查看的变量；



- 或者直接右击debug界面选择New Watch（insert）添加要查看的变量；



- 4. 点击debug界面上方向下箭头执行下一步；
- 5. 点击debug界面左侧红色“x”号即可退出。



Python中的基本语法

Python

■ Python是一门解释性语言

编译型语言在程序执行之前，有一个单独的编译过程，将程序翻译成机器语言，以后执行这个程序的时候，就不用再进行翻译了

解释型语言，是在运行的时候将程序翻译成机器语言，所以运行速度相对于编译型语言要慢

行与缩进

- Python程序的最基本的组成元素是语句，一条语句可以占有一个物理行，过长的语句可以占有多个物理行，此时这多个物理行组成了一个逻辑行，它们在物理上虽然跨越多行，但是逻辑上是属于同一部分。每个物理行的结尾可以是注释，#之后到物理行结尾为止的所有字符都是注释部分，Python解释器将忽略注释部分。

```
a="我是一个物理行"
```

```
a="""我是一个逻辑行
```

```
因为我一条语句便跨越了2个物理行"""
```

行与缩进

■ 跨行的逻辑行

1. `\`可以将两个相邻的物理行连接成一个逻辑行，这需要一个条件就是连接的第一个物理行必须没有注释，`\`添加到第一个物理行的后面，注意：`\`前面的空格会被当成是物理行的内容；
2. `[]`、`{}`、`()`可以跨越物理行；
3. 三重引号字符串常量（包括单引号和双引号）时，也可以跨越多行。

行与缩进

- 缩进是Python表示语句块的唯一方法

标准Python风格是每个缩进级别是使用4个空格或者使用Tab制表符，不过同一段代码中的缩进级别需要保持一致，**一段代码不能既用四个空格，又用tab键**，在大多数代码编辑器中建议使用tab缩进，简单方便。

注释

- 注释有多种，有单行注释，多行注释，批量注释，中文注释也是常用的

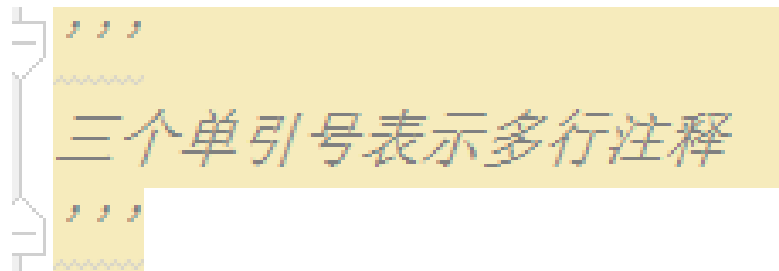
常被用作单行注释符号，在代码中使用#时，它右边的任何数据都会被忽略，当做是注释。

```
>>>print ('1') #打印1
```

```
1
```


注释

- 多行注释是用三引号''' '''包含的



The diagram shows a code block with a yellow background. It contains three lines of text: the first line is three single quotes '''', the second line is a wavy line representing a comment, and the third line is the text '三个单引号表示多行注释' (Three single quotes represent multi-line comments). The entire block is enclosed in a larger set of triple single quotes, with the closing quotes on the bottom line.

```
'''  
~~~~~  
三个单引号表示多行注释  
~~~~~  
'''
```

数据类型和变量

- 整数 其中十六进制用0x前缀和0-9, a-f表示
e.g. 1,100,-100,0xff00
- 浮点数 用科学计数法表示(较大或较小)
e.g. 1.23e9
- 字符串 以单引号(')或双引号(")括起来的任意文本
e.g. 'abc'

数据类型和变量

- 如果字符串内部既包含‘又包含“可以用转义字符\来标识

e.g.

```
>>> print('I\'m "OK!"')  
I'm "OK!"
```

所表示的内容为 I'm "OK"!

- 用r"表示"内部的字符串默认不转义

e.g.

```
>>> print(r'\\t\\')
```

 输出 `\\t\\`

- 布尔值

一个布尔值只有True、False两种值

数据类型和变量

■ 空值

用**None**表示(空值并非0)

■ 变量

变量在程序中用一个变量名表示，变量名必须是大小写英文、数字和_的组合，且不能用数字开头

e.g. `t_007 = 'T007'` 表示变量t_007是一个字符串

e.g. `Answer = True` 表示变量Answer是一个布尔值True

List和tuple

■ List

List(列表)是一种有序的集合，可以随时添加和删除其中的元素

e.g. `>>> classmates=['Michael','Bob','Tracy']`

用索引来访问list中每一个位置的元素，索引从0开始

```
>>> print(classmates[0])
```

当索引超出了范围时，Python会报IndexError的错误

List和tuple

■ List

可以往**List**中追加元素到末尾

```
>>> classmate=['Michale','Bob','Tracy']  
>>> classmate.append('Adam')  
>>> classmate  
['Michale', 'Bob', 'Tracy', 'Adam']
```

可以把元素插入到指定的位置

```
>>> classmate.insert(1,'Jack')  
>>> classmate  
['Michale', 'Jack', 'Bob', 'Tracy', 'Adam']
```

List和tuple

■ List

删除List末尾的元素

```
>>> classmate.pop()  
'Adam'  
>>> classmate  
['Michale', 'Jack', 'Bob', 'Tracy']
```

List和tuple

■ List

要删除指定位置的元素，用`pop(i)`方法，`i`是索引位置

```
>>> classmate.pop(1)
'Jack'
>>> classmate
['Michale', 'Bob', 'Tracy']
```

把某个元素替换掉，可以直接赋值给对应的索引位置

```
>>> classmate[1]='Sarah'
>>> classmate
['Michale', 'Sarah', 'Tracy']
```


List和tuple

■ Tuple

Tuple(元组)和**List**非常类似，但是**tuple**一旦初始化就不能修改

当定义一个**tuple**时，在定义的时候，**tuple**的元素就必须被确定下来

Dict和set

■ Dict

Dict全称dictionary(字典), 在其他语言中也称为map, 使用键-值 (key-value) 存储, 具有极快的查找速度

```
>>> d={'Micheal':95, 'Bob':85, 'Tracy':75}
>>> d['Micheal']
95
```

Dict和set

数据放入**Dict**的方法，除了初始化时指定外，还可以通过**key**放入

```
>>> d['Adam'] = 67
>>> d['Adam']
67
```

一个**key**只能对应一个**value**，所以多次对一个**key**放入**value**，后面的值会把前面的值冲掉

如果**key**不存在，就会报错

Dict和set

避免key不存在的错误

一是通过in判断key是否存在

```
>>> 'Thomas' in d  
False
```

二是通过Dict提供的get方法，如果key不存在，可以返回None，或者自己指定的value

```
>>> d.get('Thomas')  
>>> d.get('Thomas', -1)  
-1
```

(返回None的时候Python的交互式命令行不显示结果)

Dict和set

要删除一个key，用pop(key)方法，对应的value也会从Dict中删除

```
>>> d.pop(' Bob' )  
85  
>>> d  
{' Micheal' : 95, ' Tracy' : 75, ' Adam' : 67}
```

务必注意， Dict内部存放的顺序和key放入的顺序是没有关系的

Dict和set

set(集合)和**Dict**类似，也是一组**key**的集合，但不存储**value**，由于**key**不能重复，所以在**set**中没有重复的**key**
创建一个**set**，需要提供一个**List**作为输入集合

```
>>> s=([1, 2, 3])  
>>> s  
[1, 2, 3]
```

Dict和set

重复元素在**set**中自动被过滤

```
>>> s=set([1, 1, 2, 2, 3, 3])  
>>> s  
set([1, 2, 3])
```

通过**add(key)**方法可以添加元素到**set**中，可以重复添加，但不会有效果

```
>>> s.add(4)  
>>> s  
set([1, 2, 3, 4])
```

Dict和set

通过`remove(key)`方法可以删除元素

```
>>> s.remove(4)
>>> s
set([1, 2, 3])
```

set可以看成数学意义上的无序和无重复元素的集合，因此，两个**set**可以做数学意义上的交集、并集等操作

Dict和set

```
>>> s1=set([1, 2, 3])
>>> s2=set([2, 3, 4])
>>> s1&s2
set([2, 3])
>>> s1|s2
set([1, 2, 3, 4])
```

set和**dict**的唯一区别仅在于没有存储对应的**value**，但是，**set**的原理和**dict**一样，所以，同样不可以放入可变对象

类(class)

- 类简单看做是数据以及由存取、操作这些数据的方法所组成的一个集合
- 类的优点：
 - 类对象是多态的：也就是多种形态，这意味着我们可以对不同的类对象使用同样的操作方法，而不需要额外写代码。
 - 类的封装：封装之后，可以直接调用类的对象，来操作内部的一些类方法，不需要让使用者看到代码工作的细节。
 - 类的继承：类可以从其它类或者元类中继承它们的方法，直接使用。

类(class)

■ 定义类的语法:

```
>>> class Iplaypython:  
...     def fname(self, name):  
...         self.name=name
```

第一行，语法是**class** 后面紧接着，类的名字，最后别忘记“冒号”，这样来定义一个类。

类的名字，首字母，有一个不成文的规定：最好是大写，这样便于在代码中识别区分每个类。

第二行开始是类的方法，大家看到了，和函数非常相似，但是与普通函数不同的是，它的内部有一个“**self**”参数，它的作用是针对对象自身的引用。

条件判断和循环

■ 条件判断

用if语句实现

```
age=20
if age>=18:
    print('your age is', age)
print('adult')
```

根据Python的缩进规则，如果if语句判断是True，就把缩进的两行print语句执行了，否则，什么也不做

也可以给if添加一个else语句，意思是，如果if判断是False，不要执行if的内容，去执行else

条件判断和循环

■ 条件判断

elif **elif**是**else if** 的缩写

if是从上往下判断，如果在某个判断上是**True**，把该判断对应的语句执行后，就忽略掉剩下的**elif**和**else**

```
age=20
if age>=6:
    print('teenager')
elif age>=18:
    print('adult')
else:
    print('kid')
```

打印的是teenager

条件判断和循环

■ for...in循环

依次把list或tuple中的每个元素迭代出来

```
names=['Michael','Bob','Tracy']  
for name in names:  
    print(name)
```

所以for x in ...循环就是把每个元素代入变量x，然后执行缩进块的语句

条件判断和循环

■ While循环

只要条件满足，就不断循环，条件不满足时退出循环

```
sum=0
n=99
while n>0:
    sum=sum+n
    n=n-2
print(sum)
```

条件判断和循环

■ Break

与C语言中的break类似，常与if连用

```
for x in range(1,4):  
    print(x,"for语句")  
    if x>2:  
        print(x,"if语句")  
        break  
else:  
    print(x,"else语句")
```


条件判断和循环

■ Continue

与C语言中的continue类似

```
for x in range(1, 4):  
    print(x, 'for语句')  
    continue  
    print(x, 'continue语句后')  
else:  
    print(x, 'else语句')
```

函数

■ 自带函数

Python内置了许多的自带函数可以直接引用
在 <http://docs.python.org/3/library/functions.html>
可以查询所有的自带函数

函数

■ 调用函数

要调用一个函数，需要知道函数的名称和参数

比如求绝对值的函数**abs**，只有一个参数。

也可以在交互式命令行通过**help(abs)**查看**abs**函数的帮助信息。

调用函数的时候，如果传入的参数数量不对，会报**TypeError**的错误，并且**Python**会明确地告诉你：**abs()**有且仅有1个参数，但传入了两个。

函数

■ 数据类型转换

```
>>> int('123')
```

```
123
```

```
>>> int(12.34)
```

```
12
```

```
>>> float('12.34')
```

```
12.34
```

函数

■ 函数名

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”。

```
>>> a=abs#变量a指向abs函数
```

```
>>> a(-1)#通过a调用abs函数
```

```
1
```

函数

■ 定义函数

定义一个函数要使用**def**语句，依次写出函数名、括号、括号中的参数和冒号:，然后，在缩进块中编写函数体，函数的返回值用**return**语句返回

```
def my_abs(x):  
    if x >= 0:  
        return x  
    else:  
        return -x
```

函数体内部的语句在执行时，一旦执行到**return**时，函数就执行完毕，并将结果返回

函数

■ 空函数

如果想定义一个什么事也不做的空函数，可以用**pass**语句

```
def nop():  
    pass
```

实际上**pass**可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个**pass**，让代码能运行起来，**pass**还可以用在其他语句里

```
if age >= 18:  
    pass
```

缺少了**pass**，代码运行就会有语法错误

函数

■ 参数检查

调用函数时，如果参数个数不对，Python解释器会自动检查出来，并抛出**TypeError**。

但是如果参数类型不对，Python解释器就无法帮我们检查。当传入了不恰当的参数时，内置函数**abs**会检查出参数错误，而我们定义的**my_abs**没有参数检查，所以，这个函数定义不够完善。

修改一下**my_abs**的定义，对参数类型做检查，只允许整数和浮点数类型的参数

函数

数据类型检查可以用内置函数`isinstance`实现

```
def my_abs(x):  
    if not isinstance(x, (int, float)):  
        raise TypeError('bad operand type')  
    if x >= 0:  
        return x  
    else:  
        return -x
```

函数

■ 返回多个值

```
import math

def move(x, y, step, angle=0):
    nx = x + step * math.cos(angle)
    ny = y - step * math.sin(angle)
    return nx, ny
```

这样我们就可以同时获得返回值

函数

```
>>> x, y = move(100, 100, 60, math.pi / 6)
```

```
>>> print (x, y)
```

```
151.961524227 70.0
```

其实这只是一种假象，Python函数返回的仍然是单一值

返回值是一个**tuple**，但是，在语法上，返回一个**tuple**可以省略括号，而多个变量可以同时接收一个**tuple**，按位置赋给对应的值，所以，Python的函数返回多值其实就是返回一个**tuple**，但写起来更方便。

函数

■ 默认参数

```
def power(x, n=2):  
    s = 1  
    while n > 0:  
        n = n - 1  
        s = s * x  
    return s
```

即把第二个参数默认值设定为2

当我们调用`power(5)`时，相当于调用`power(5, 2)`

对于 $n > 2$ 的情况，就必须明确地传入 n ，如`power(5, 3)`

函数

设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错；

二是当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

定义默认参数要牢记一点：默认参数必须指向不变对象

函数

■ 可变参数

可变参数就是传入的参数个数是可变的。

```
def calc(*numbers):  
    sum = 0  
    for n in numbers:  
        sum = sum + n * n  
    return sum
```

定义可变参数和定义list或tuple参数相比，仅仅在参数前面加了一个*号，在函数内部，参数numbers接收到的是一个tuple。

函数

如果已经有一个**list**或者**tuple**，要调用一个可变参数，可在**list**或**tuple**前面加一个*号，把**list**或**tuple**的元素变成可变参数传进去。

```
>>> nums = [1, 2, 3]
```

```
>>> calc(*nums)
```

```
14
```

函数

■ 关键字参数

可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。

而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。

关键字参数有什么用？它可以扩展函数的功能。比如，在person函数里，我们保证能接收到name和age这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

函数

```
def person(name, age, **kw):  
    print('name:', name, 'age:', age, 'other:', kw)
```

函数person除了必选参数name和age外，还接受关键字参数kw，在调用该函数时，可以只传入必选参数

```
>>> person('Michael', 30)  
name: Michael age: 30 other: {}
```

函数

也可以传入任意个数的关键字参数

```
>>> person('Bob', 35, city='Beijing')
```

```
name: Bob age: 35 other: {'city': 'Beijing'}
```

```
>>> person('Adam', 45, gender='M', job='Engineer')
```

```
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

模块导入与函数引入

■ import modname （推荐方式）

模块是指一个可以交互使用，或者从另一Python 程序访问的代码段。只要导入了一个模块，就可以引用它的任何公共的函数、类或属性。模块可以通过这种方法来使用其它模块的功能。

用import语句导入模块，就在当前的名称空间(namespace)建立了一个到该模块的引用。这种引用必须使用全称，也就是说，当使用在被导入模块中定义的函数时，必须包含模块的名字。所以不能只使用 funcname，而应该使用 modname.funcname

模块导入与函数引入

- `from modname import funcname`

- `from modname import fa, fb, fc`

- `from modname import *`

函数引入与前述模块导入的区别：**funcname** 被直接导入到本地名字空间去了，所以它可以直接使用，而不需要加上模块名的限定

* 表示，该模块的所有公共对象(**public objects**)都被导入到当前的名称空间，也就是任何只要不是以”_”开始的东西都会被导入。

此时**modname**没有被定义，所以**modname.funcname**这种方式不起作用。并且，如果**funcname**如果已经被定义，它会被新版本（该导入模块中的版本）所替代。

Matplotlib使用

- Matplotlib 是 Python 的绘图库。
- 安装方式

树莓派打开 Terminal 窗口安装二进制包：

```
sudo apt install python3-matplotlib
```

Anaconda自带matplotlib包；

运行import matplotlib，如果没有报错则证明安装成功。

- Matplotlib用户手册：
<https://matplotlib.org/contents.html>

Matplotlib使用

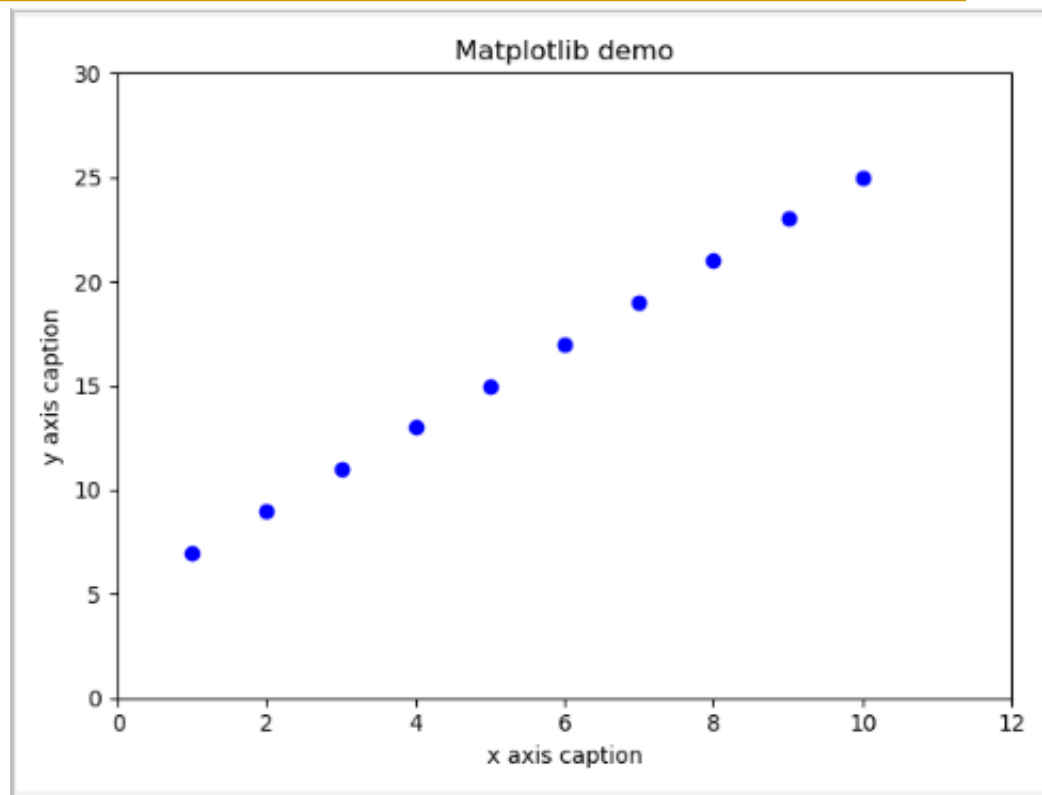
■ 示例:

(线性图的离散显示)

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.arange(1, 11)
y = 2 * x + 5
```

```
plt.title("Matplotlib demo")
plt.xlabel("x axis caption")
plt.ylabel("y axis caption")
plt.plot(x, y, "ob")
plt.axis([0, 12, 0, 30])
plt.show()
```



设置图像标题

设置横轴 (x轴) 标签

设置纵轴 (y轴) 标签

plt.plot() 绘制折线图, 括号内为横纵坐标与显示设置

设置横纵坐标区间

图形显示

Matplotlib使用

- `plot(x, y, "ob")` 中的“ob”格式字符串设置显示格式，‘o’为圆标记，‘b’为蓝色。（其他格式与绘图可自学了解）
- 在一幅图上显示多个图表——`subplot()`

#建立 subplot 网格（高为2，宽为1），激活第一个 subplot并绘制第一个图像

```
plt.subplot(2, 1, 1)
```

```
plt.plot()
```

将第二个 subplot 激活，并绘制第二个图像

```
plt.subplot(2, 1, 2)
```

```
plt.plot()
```

展示图像

```
plt.show()
```

程序的异常处理

■ 什么是异常？

异常即是一个事件，该事件会在程序执行过程中发生，影响了程序的正常执行。

一般情况下，在Python无法正常处理程序时就会发生一个异常。

异常是Python对象，表示一个错误。

当Python脚本发生异常时我们需要捕获处理它，否则程序会终止执行。

程序的异常处理

■ 异常处理

捕捉异常可以使用`try/except`语句。

`try/except`语句用来检测`try`语句块中的错误，从而让`except`语句捕获异常信息并处理。

如果你不想在异常发生时结束你的程序，只需在`try`里捕获它。

语法：

以下为简单的`try....except...else`的语法：

程序的异常处理

```
try:
<语句>           #运行别的代码
except <名字>:
<语句>           #如果在try部份引发了'name'异常
except <名字>, <数据>:
<语句>           #如果引发了'name'异常，获得附加的数据
else:
<语句>           #如果没有异常发生
```

程序的异常处理

- **try**的工作原理是，当开始一个**try**语句后，**python**就在当前程序的上下文中作标记，这样当异常出现时就可以回到这里，**try**子句先执行，接下来会发生什么依赖于执行时是否出现异常。
- 如果当**try**后的语句执行时发生异常，**python**就跳回到**try**并执行第一个匹配该异常的**except**子句，异常处理完毕，控制流就通过整个**try**语句（除非在处理异常时又引发新的异常）。

程序的异常处理

- 如果在**try**后的语句里发生了异常，却没有匹配的**except**子句，异常将被递交到上层的**try**，或者到程序的最上层（这样将结束程序，并打印缺省的出错信息）。
- 如果在**try**子句执行时没有发生异常，**python**将执行**else**语句后的语句（如果有**else**的话），然后控制流通过整个**try**语句。

程序的异常处理

■ finally子句

finally子句是无论是否检测到异常，都会执行的一段代码。我们可以丢掉**except**子句和**else**子句，单独使用**try...finally**，也可以配合**except**等使用。

程序的异常处理

代码如下：

```
>>> import syslog
>>> try:
...     f = open("/root/test.py")
... except IOError,e:
...     syslog.syslog(syslog.LOG_ERR,"%s"%e)
... else:
...     syslog.syslog(syslog.LOG_INFO,"no exception caught\n")
... finally:
>>>     f.close()
```

程序的异常处理

■ 断言（assert）

`assert expression[,reason]`

`assert`是断言的关键字。执行该语句的时候，先判断表达式`expression`，如果表达式为真，则什么都不做；如果表达式不为真，则抛出异常

```
>>> assert len("love")==len("like")
>>> assert 1==1
>>> assert 1==2,"1 is not equal 2!"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: 1 is not equal 2!
```

常见的报错

异常	描述
NameError	尝试访问一个没有声明的变量
ZeroDivisionError	除数为0
SyntaxError	语法错误
IndexError	索引超出序列范围
KeyError	请求一个不存在的字典关键字
IOError	输入输出错误（比如你要读的文件不存在）
AttributeError	尝试访问未知的对象属性
ValueError	传给函数的参数类型不正确，比如给int()函数传入字符串形

文件处理（IO）

IO在计算机中指Input和Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。

无处不在的IO

- 比如你打开浏览器，访问百度首页，浏览器这个程序就需要通过网络IO获取百度的网页。浏览器首先会发送数据给百度服务器，告诉它我想要首页的HTML，这个动作是往外发数据，叫Output，随后百度服务器把网页发过来，这个动作是从外面接收数据，叫Input。所以，通常，程序完成IO操作会有Input和Output两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有Input操作，反过来，把数据写到磁盘文件里，就只是一个Output操作。
- IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。

输出

- 用**print**加上字符串，就可以向屏幕上输出指定的文字。比如输出'hello, world'，用代码实现如下：

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

- **print**语句也可以跟上多个字符串，用逗号“,”隔开，就可以连成一串输出：

```
print('flash cat', 'cute dog', 'fly')
```

- `print`会依次打印每个字符串，遇到逗号“,”会输出一个空格
- `print`也可以打印整数，或者计算结果：

```
>>> print(300)
300
>>> print(100+200)
300
```

- 我们可以把计算`100 + 200`的结果打印得更漂亮一点

```
>>> print('100 + 200 =', 100+200)
100 + 200 = 300
```

- 注意，对于`100 + 200`，Python解释器自动计算出结果300，但是，`'100 + 200 ='`是字符串而非数学公式，Python把它视为字符串

输入

- 如果要从用户从电脑输入一些字符怎么办？Python提供了一个**input**，可以让用户输入字符串，并存放到一个变量里。比如输入用户的名字：

```
>>> name=input()  
Michael
```

- 当输入**name = input()**并按下回车后，Python交互式命令行就在等待输入了。这时，用户可以输入任意字符，然后按回车后完成输入。

- 输入完成后，不会有任何提示，Python交互式命令行又回到>>>状态了。那刚才输入的内容到哪去了？答案是存放到name变量里了。可以直接输入name查看变量内容：

```
>>> name  
'Michael'
```

raw_input可以显示一个字符串来提示用户输入，于是可以把代码改成：

```
name=input('please enter your name:')  
print('hello,', name)
```

变量

- 在计算机程序中，变量不仅可以为整数或浮点数，还可以是字符串，因此，**name**作为一个变量就是一个字符串。
- 要打印出**name**变量的内容，除了直接写**name**然后按回车外，还可以用**print**语句：

```
>>> print(name)  
Michael
```

同步IO&异步IO

- 由于CPU和内存的速度远远高于外设的速度，所以，在IO编程中，就存在**速度严重不匹配**的问题。
- 比如要把100M的数据写入磁盘，CPU输出100M的数据只需要0.01秒，可是磁盘要接收这100M数据可能需要10秒，如何解决这个问题呢？方法有两种：同步IO;异步IO。
- 同步IO:CPU**等待**IO执行，程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，紧接着往下执行
- 异步IO:CPU**不等待**IO的执行去执行其他任务，后续代码可立即接着执行，此项称为异步IO（性能更高效，但是编程模型复杂）

- 实质上，同步IO和异步IO的关系就好似在商场吃饭，但是自己点的单还需要一会时间才能做好，如果自己在餐厅等待就是同步IO；但是如果选择在等餐间隙去商场里边逛一下，那么就是异步IO。但是现在问题出现了，如何在异步IO里边去通知系统可以继续执行后续代码？这就好似自己在商场里边逛的时候餐厅如何通知自己的单已经做好了？
- 普通的代码是无法完成异步IO的：

```
do_some_code()
f = open('/path/to/file', 'r')
r = f.read() # <== 线程停在此处等待IO操作结果
# IO操作完成后线程才能继续执行：
do_some_code(r)
```

- 同步IO的模型是无法实现异步IO模型的，异步IO模型需要一个消息循环，在消息循环中，主线程不断的重复读取消息处理消息这个过程。

```
loop = get_event_loop()
while True:
    event = loop.get_event()
    process_event(event)
```

消息模型其实早在应用在桌面应用程序中了。一个GUI程序的主线程就负责不停地读取消息并处理消息。所有的键盘、鼠标等消息都被发送到GUI程序的消息队列中，然后由GUI程序的主线程处理。

和异步IO相关的还有python3.4版本中的asyncio标准库，感兴趣的同学可以去做更多的了解，在这里不多赘述

文件读写

- 读写文件是最常见的IO操作。Python内置了读写文件的函数，用法和C是兼容的
- 在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

文件的打开

- 以读文件的模式打开一个文件对象，使用Python内置的`open()`函数，传入文件名和标示符；
- 例如利用下列代码：

```
>>> f = open('/Users/michael/test.txt', 'r')
```

如果文件不存在，`open()`函数就会抛出一个`IOError`的错误，并且给出错误码和详细的信息告诉你文件不存在：

```
>>> f=open('/Users/michael/notfound.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: '/Users/michael/notfound.txt'
>>>
```

文件的读取

- 如果文件打开成功，接下来，调用`read()`方法可以一次读取文件的全部内容，Python把内容读到内存，用一个`str`对象表示：

```
>>> f.read()  
'Hello, world!'
```

最后一步是调用`close()`方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

```
>>> f.close()
```

Try...finally 函数

- 由于文件读写时都有可能产生IOError，一旦出错，后面的f.close()就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用try ... finally来实现：

- try:

```
f = open('/path/to/file', 'r')
```

```
print f.read()
```

```
finally:
```

```
if f:
```

```
f.close()
```

With 语句

- Python引入了with语句来自动帮用户调用close()方法

```
with open('/path/to/file', 'r') as f:  
    print f.read()
```

这和前面的try ... finally是一样的，但是代码更佳简洁，并且不必调用f.close()方法

Read(size)& readline ()

- 调用**read()**会一次性读取文件的全部内容,因此为保护内存,可采用反复调用**read(size)**的方法,每次只读取**size**个字节的内容。
- 调用**readline()**可以每次读取一行内容,调用**readlines()**一次读取所有内容并按行返回**list**。因此,要根据需要决定怎么调用。
- 如果文件很小, **read()**一次性读取最方便; 如果不能确定文件大小, 反复调用**read(size)**比较保险; 如果是配置文件, 调用**readlines()**最方便

二进制文件

- 前面讲的默认都是读取文本文件，并且是**ASCII**编码的文本文件。要读取二进制文件，比如图片、视频等等，用'**rb**'模式打开文件即可：

```
>>> f = open('/Users/michael/test.jpg', 'rb')
>>> f.read()
'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' # 十六进制表示的字节
```

文件的写入

- 写文件和读文件是一样的，唯一区别是调用`open()`函数时，传入标识符'`w`'或者'`wb`'表示写文本文件或写二进制文件：

```
>>> f = open('/Users/michael/test.txt', 'w')  
>>> f.write('Hello, world!')  
>>> f.close()
```

写入语句里的with语句

- 可反复调用**write()**来写入文件，但是务必要调用**f.close()**来关闭文件。写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用**close()**方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用

```
with open('/Users/michael/test.txt', 'w') as f: 下
    f.write('Hello, world!')
```

- 要写入特定编码的文本文件，请效仿**codecs**的示例，写入**unicode**，由**codecs**自动转换成指定编码

操作文件和目录

- 如果用户要操作文件、目录，可以在命令行下面输入操作系统提供的各种命令来完成。比如**dir**、**cp**等命令
- 如果要在**Python**程序中执行这些目录和文件的操作怎么办？其实操作系统提供的命令只是简单地调用了操作系统提供的接口函数，**Python**内置的**os**模块也可以直接调用操作系统提供的接口函数。
 -

操作文件和目录的函数一部分放在os模块中，一部分放在os.path模块中。查看、创建和删除目录可以这么调用

```
# 查看当前目录的绝对路径:
```

```
>>> os.path.abspath('.')
```

```
'/Users/michael'
```

```
# 在某个目录下创建一个新目录,
```

```
# 首先把新目录的完整路径表示出来:
```

```
>>> os.path.join('/Users/michael', 'testdir')
```

```
'/Users/michael/testdir'
```

```
# 然后创建一个目录:
```

```
>>> os.mkdir('/Users/michael/testdir')
```

```
# 删掉一个目录:
```

```
>>> os.rmdir('/Users/michael/testdir')
```

序列化（Pickle模块）

- 在程序运行的过程中，所有的变量都是在内存中，比如，定义一个dict:

```
d = dict(name='Bob', age=20, score=88)
```

- 可以随时修改变量，比如把name改成'Bill'，但是一旦程序结束，变量所占用的内存就被操作系统全部回收如果没有把修改后的'Bill'存储到磁盘上，下次重新运行程序，变量又被初始化为'Bob'。
- 我们把变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling

序列化之后，可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即**unpickling**

Python提供两个模块来实现序列化：**cPickle**和**pickle**。这两个模块功能是一样的，区别在于**cPickle**是C语言写的，速度快，**pickle**是纯Python写的，速度慢。用的时候，先尝试导入**cPickle**，如果失败，再导入**pickle**：

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

首先，尝试把一个对象序列化并写入文件：

```
>>> d = dict(name='Bob', age=20, score=88)
>>> pickle.dumps(d)
"(dp0\nS'age'\np1\nI20\nsS'score'\np2\nI88\nsS'name'\np3\nS'Bob'\np4\ns."
```

`pickle.dumps()`方法把任意对象序列化成一个`str`，然后就可以把这个`str`写入文件。或者用另一个方法`pickle.dump()`直接把对象序列化后写入一个`file-like Object`：

```
>>> f = open('dump.txt', 'wb')
>>> pickle.dump(d, f)
>>> f.close()
```


当用户要把对象从磁盘读到内存时，可以先把内容读到一个str，然后用pickle.loads()方法反序列化出对象，也可以直接用pickle.load()方法从一个file-like Object中直接反序列化出对象。打开另一个Python命令行来反序列化刚才保存的对象：

```
>>> f = open('dump.txt', 'rb')
>>> d = pickle.load(f)
>>> f.close()
>>> d
{'age': 20, 'score': 88, 'name': 'Bob'}
```

它们

Python多线程

- 在介绍多线程之前，我们先理解一下Python的单线程；
- 在单线程模式里，我想做听音乐和看电影两件事儿，那么一定要先排一下顺序。：

```
from time import ctime, sleep
def music():
    for i in range(2):
        print("I was listening to music. %s" % ctime())
        sleep(1)
def move():
    for i in range(2):
        print("I was at the movies! %s" % ctime())
        sleep(5)
if __name__ == '__main__':
    music()
    move()
    print("all over %s" % ctime())
```

Python多线程

- 我们先听了一首音乐，通过for循环来控制音乐的播放了两次，每首音乐播放需要1秒钟，`sleep()`来控制音乐播放的时长。接着我们又看了一场电影每一场电影需要5秒钟，因为太好看了，所以我也通过for循环看两遍。在整个休闲娱乐活动结束后，我们通过 `print` (“all over %s” %`ctime()`) 查看运行时间，结果如下：

```
I was listening to music. Thu Sep 28 11:08:00 2017
I was listening to music. Thu Sep 28 11:08:01 2017
I was at the movies! Thu Sep 28 11:08:02 2017
I was at the movies! Thu Sep 28 11:08:07 2017
all over Thu Sep 28 11:08:12 2017
[Finished in 12.1s]
```

Python多线程

- 其实，`music()`和`move()`更应该被看作是音乐和视频播放器，至于要播放什么歌曲和视频应该由我们使用时决定。所以，我们对上面代码做了改造：

```
import threading
from time import ctime, sleep
def music(func):
    for i in range(2):
        print("I was listening to %s. %s" % (func, ctime()))
        sleep(1)
def move(func):
    for i in range(2):
        print("I was at the %s! %s" % (func, ctime()))
        sleep(5)
if __name__ == '__main__':
    music(u'爱情买卖')
    move(u'阿凡达')
    print("all over %s" % ctime())
```

Python多线程

- 上面的程序中我们对music()和move()进行了传参处理。体验中国经典歌曲和欧美大片文化。运行结果如下：

```
I was listening to 爱情买卖. Thu Sep 28 11:16:49 2017
I was listening to 爱情买卖. Thu Sep 28 11:16:50 2017
I was at the 阿凡达! Thu Sep 28 11:16:51 2017
I was at the 阿凡达! Thu Sep 28 11:16:56 2017
all over Thu Sep 28 11:17:01 2017
[Finished in 12.1s]
```

Python多线程

- 科技在发展，时代在进步，我们的CPU也越来越快，同时干多个活都没问题的；于是，操作系统就进入了多任务时代。我们听着音乐吃着火锅的不再是梦想。
- python提供了两个模块来实现多线程thread 和threading，thread 有一些缺点，在threading 得到了弥补，为了不浪费大家时间，所以我们直接学习threading 就可以了。
- 继续对上面的例子进行改造，引入threading来同时播放音乐和视频。

Python多线程

```
import threading
from time import ctime, sleep
def music(func):
    for i in range(2):
        print("I was listening to %s. %s" % (func, ctime()))
        sleep(1)
def move(func):
    for i in range(2):
        print("I was at the %s! %s" % (func, ctime()))
        sleep(5)

threads=[]
t1=threading.Thread(target=music, args=(u'爱情买卖',))
threads.append(t1)
t2=threading.Thread(target=move, args=(u'阿凡达',))
threads.append(t2)

if __name__ == '__main__':
    for t in threads:
        t.setDaemon(True)
        t.start()
    print("all over %s" % ctime())
```

Python多线程

- `import threading`

首先导入`threading` 模块，这是使用多线程的前提。

- `threads = []`

```
t1 = threading.Thread(target=music,args=(u'爱情买卖',))  
threads.append(t1)
```

创建了`threads`数组，创建线程`t1`,使用`threading.Thread()`方法，在这个方法中调用`music`方法`target=music`，`args`方法对`music`进行传参。把创建好的线程`t1`装到`threads`数组中。接着以同样的方式创建线程`t2`，并把`t2`也装到`threads`数组。

Python多线程

- `for t in threads:`
 `t.setDaemon(True)`
 `t.start()`

最后通过**for**循环遍历数组。（数组被装载了**t1**和**t2**两个线程）。

- `setDaemon()`

`setDaemon(True)`将线程声明为守护线程，功能是设置子线程随主线程的结束而结束，必须在**start()** 方法调用之前设置。设置以后，子线程启动后，父线程也继续执行下去，当父线程执行完最后一条语句**print “all over %s” %ctime()**后，没有等待子线程，直接就退出了，同时子线程也一同结束。通常用于需要自动终止子线程的应用场景。

- `start()`

开始线程活动。

Python多线程

I was listening to 爱情买卖. Thu Sep 28 11:27:23 2017

I was at the 阿凡达! Thu Sep 28 11:27:23 2017all over Thu Sep 28 11:27:23 2017

[Finished in 0.1s]

- 从执行结果来看，子线程（`muisc`、`move`）和主线程（`print "all over %s" %ctime()`）都是同一时间启动，但由于主线程执行完结束，所以导致子线程也终止。

Python多线程

■ 继续调整程序

```
import threading
from time import ctime, sleep
def music(func):
    for i in range(2):
        print("I was listening to %s. %s" % (func, ctime()))
        sleep(1)
def move(func):
    for i in range(2):
        print("I was at the %s! %s" % (func, ctime()))
        sleep(5)

threads=[]
t1=threading.Thread(target=music, args=(u'爱情买卖',))
threads.append(t1)
t2=threading.Thread(target=move, args=(u'阿凡达',))
threads.append(t2)

if __name__=='__main__':
    for t in threads:
        t.setDaemon(True)
        t.start()
    t.join()
    print("all over %s" % ctime())
```

Python多线程

■ `t.join()`

我们只对上面的程序加了个`join()`方法，用于等待线程终止。`join`的作用是，在子线程完成运行之前，这个子线程的父线程将一直被阻塞。

注意：`join()`方法的位置是在`for`循环外的，也就是说必须等待`for`循环里的两个进程都结束后，才去执行主进程。

■ 运行结果：

```
I was listening to 爱情买卖. Thu Sep 28 11:32:59 2017
I was at the 阿凡达! Thu Sep 28 11:32:59 2017
I was listening to 爱情买卖. Thu Sep 28 11:33:00 2017
I was at the 阿凡达! Thu Sep 28 11:33:04 2017
all over Thu Sep 28 11:33:09 2017
[Finished in 10.1s]
```

Python多线程

```
I was listening to 爱情买卖. Thu Sep 28 11:32:59 2017
I was at the 阿凡达! Thu Sep 28 11:32:59 2017
I was listening to 爱情买卖. Thu Sep 28 11:33:00 2017
I was at the 阿凡达! Thu Sep 28 11:33:04 2017
all over Thu Sep 28 11:33:09 2017
[Finished in 10.1s]
```

- 从执行结果可看到，**music** 和**move** 是同时启动的。开始时间**32分59秒**，直到调用主进程为**33分09秒**，总耗时为**10秒**。从单线程时减少了**2秒**，这些时间看起来似乎不能明显的体现出多线程能节省时间，我们可以把**music**的**sleep()**的时间调整为**4秒**。

Python多线程

```
def music(func):  
    for i in range(2):  
        print("I was listening to %s. %s" % (func, ctime()))  
        sleep(4)
```

■ 执行结果如下：

```
I was listening to 爱情买卖. Thu Sep 28 11:43:07 2017  
I was at the 阿凡达! Thu Sep 28 11:43:07 2017  
I was listening to 爱情买卖. Thu Sep 28 11:43:11 2017  
I was at the 阿凡达! Thu Sep 28 11:43:12 2017  
all over Thu Sep 28 11:43:17 2017  
[Finished in 10.1s]
```

Python多线程

- 子线程启动**43分7秒**，主线程运行**43分17秒**。
- 虽然**music**每首歌曲从**1秒**延长到了**4**，但通多程线的方式运行脚本，总的时间没变化。如果用单线程，则需要**18s**的时间。只要多线程的前提下，听歌时间小于**5秒**，就不会影响父线程的关闭时间。（父线程只等待最后**1**个子线程关闭，父线程才关闭）

多线程读写锁

- 读写锁一般用于多个读者,1个或多个写者同时访问某种资源的时候。多个读者之间是可以共享资源的,但是写者与读者之间,写者与写者之间是资源互斥的。简单说来就类似一个在笔记本上记录,当一个人已经占用了笔记本,那么另一个人只能等待前者写完再操作。
- 读写锁与一般锁最大的区别是对同一共享资源多个线程的读取行为是并行的,同时保持该资源同一时刻只能由一个写进程独占,且写请求相对读请求有更高的优先级以防止**writer starvation**。(一般锁同一时刻只能由一个线程独占,不论是读进程还是写进程,即读写都是串行的,而读写锁读是并行的,写是串行的)

- 使用读写锁的注意事项：
- 慎用**promote**！读写锁一般都有提权函数**promote()**用于将一个已经获取读锁的线程进一步提权获得写锁，这样做很容易导致程序死锁。例如，两个均已经获取读锁的线程A和B同时调用**promote**函数尝试获得写权限，线程A发现存在读线程B，需要等待B完成以获取写锁，线程B发现存在读线程A，需要等待线程A完成以获取写锁，循环等待发生，程序死锁。因此，当且仅当你能确定当前仅有一个读线程占有锁时才能调用**promote**函数。一个已经获取读锁的线程提权最好的办法是先释放读锁，然后重新申请写锁
- 使用多个锁时保证加解锁顺序相反。
- 获得锁的线程用完后一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以需要**try...finally**来确保锁一定会被释放。

■ 参考代码：

```
balance = 0
lock = threading.Lock()

def run_thread(n):
    for i in range(100000):
        # 先要获取锁:
        lock.acquire()
        try:
            # 放心地改吧:
            change_it(n)
        finally:
            # 改完了一定要释放锁:
            lock.release()
```

参考代码网址：

- <https://majid.info/blog/a-reader-writer-lock-for-python/>
- <https://github.com/azraelxyz/rwlock/blob/master/rwlock/rwlock.py>

参考资料

- Python3基础教程：可以在线编程，说明完备，适合快速入门
<http://www.runoob.com/python3/python3-tutorial.html>
- 廖雪峰Python教程：
<https://www.liaoxuefeng.com/wiki/1016959663602400>
- Python官方中文教程
<https://docs.python.org/zh-cn/3/tutorial/index.html>
- Python学习视频：<http://www.jikexueyuan.com/course/python/>
- 改善Python程序的91个建议：<https://zhuanlan.zhihu.com/p/26155739>

编程作业

- 每个班级教师布置一道编程题。
- **每位同学**按照要求在Elearning上提交实验报告和code。
- 实验报告模板已上传到Elearning。

致谢

- 本课件由以下同学协助编写
 - 谌达 (16210720026)
 - 袁渊源 (16307130267)
 - 夏潇 (16300720052)
 - 田耀光 (16300720044)
 - 蒋凯帆 (16300720051)