

飞扬研发第一次例会

四川大学飞扬俱乐部

胡宗尧 2023.11.5

目录

研发第一次例会

目录

Part1. Python 光速入门

Part2. Git 入门

Part3. GitHub 使用完全教程

Part1. Python 光速入门

写在前面：只是带你过一遍 python 的基础语法，目的就是为了让每个读者看完以后，都有能力**安装并使用第三方模块**，从而快速获得写 python 的快感，大大降低学习曲线。

建议「**先快速上手，找个好玩的东西做，然后再考虑往深里学**」。比如我这篇教程好多都是照搬廖雪峰的python教程的内容，讲的真的很好，但是这一系列文章没有几个礼拜你啃不完啊，等你啃完了，学习的热情早就没了。

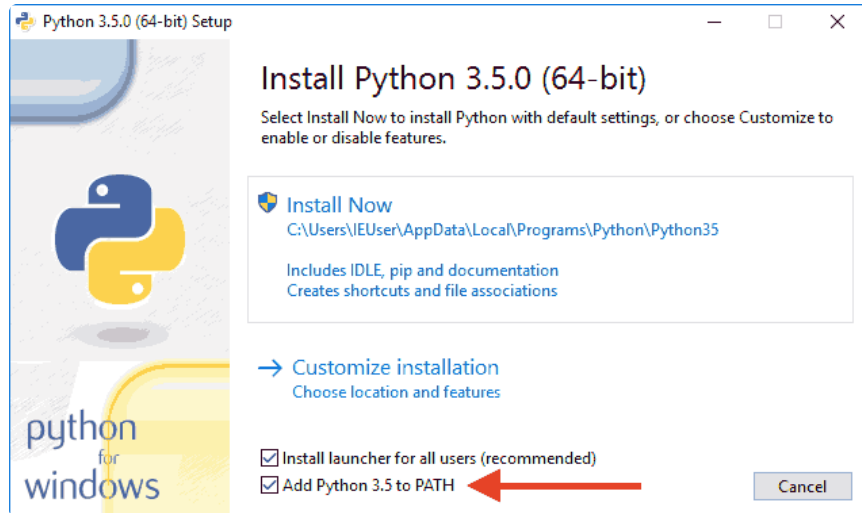
0. 安装 python 与开发环境推荐

安装 PYTHON

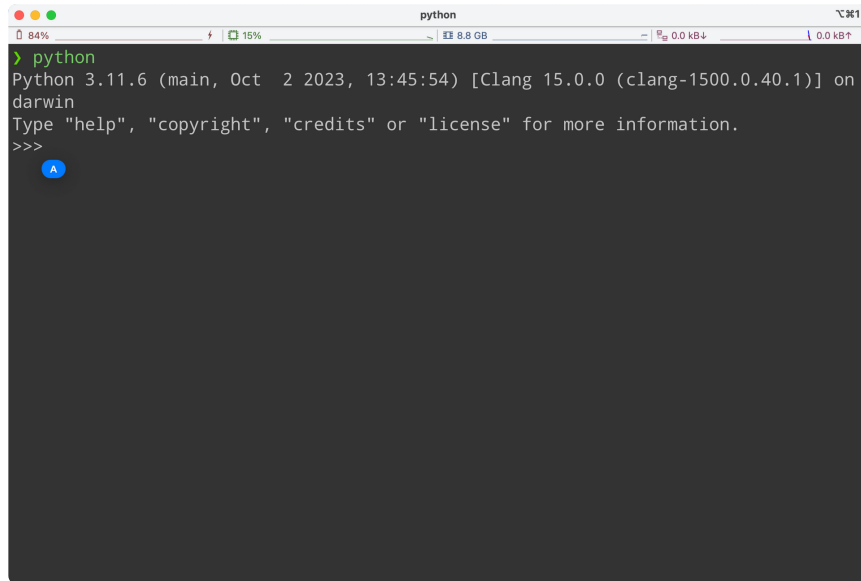
python 的安装非常简单，只需要两步：

第一步：

在官网下载最新的 python 安装包，安装时一定要记得勾选`Add Python3.11 to PATH`



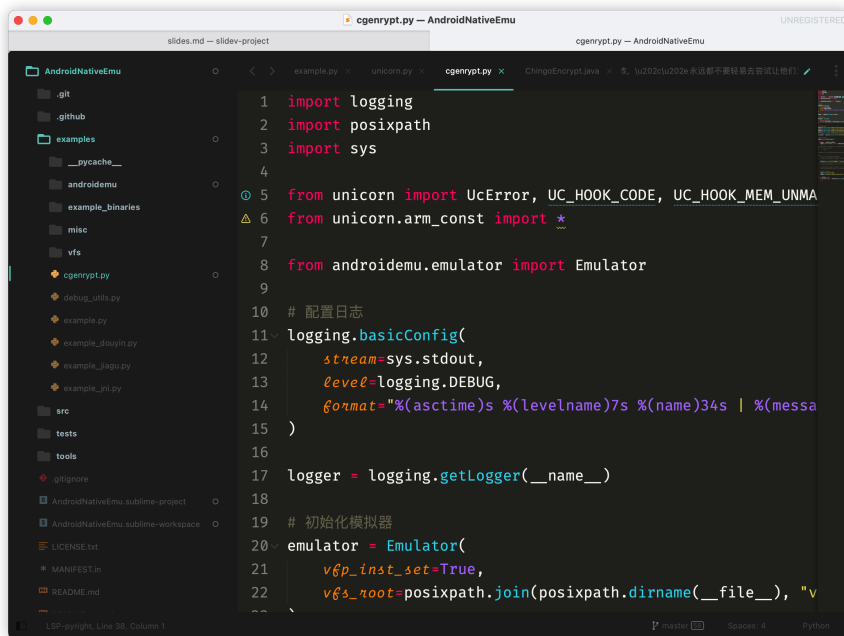
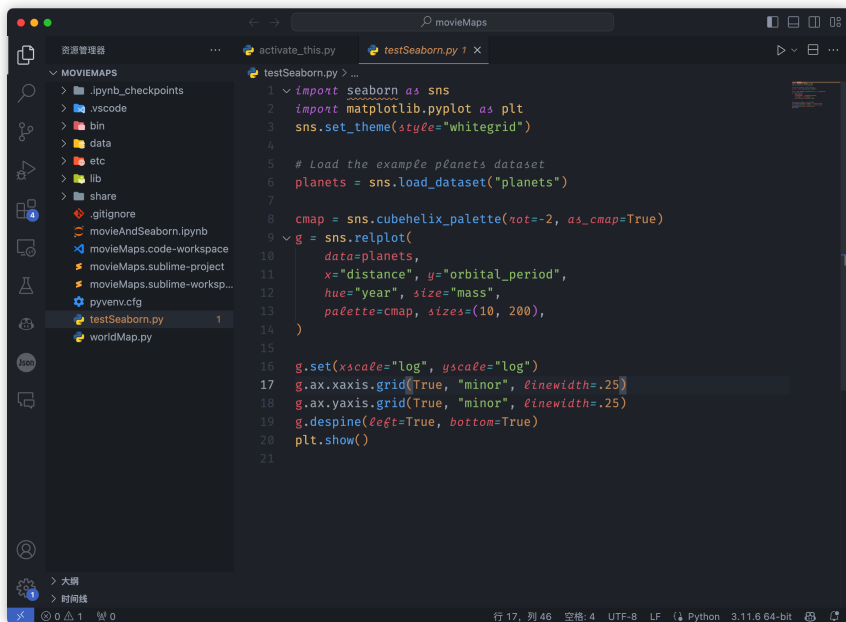
第二步：菜单栏里打开命令提示符，输入 python 后回车，如果出现`>>>`，即所谓的「python 解释器」或者「python 交互环境」，则可视为安装成功。



开发环境配置

请不要用Word和Windows的记事本。Word 保存的不是纯文本文件；记事本没有代码高亮，连IDLE都不如

- 微软出品的Visual Studio Code：最火的编辑器，没有之一，开源免费。
- 高颜值的Sublime Text4：同样跨平台，比 vscode 启动更快，缺点是不开源。



1. Python 是什么?

- 写起来优雅、快速：完成同一个任务，C 语言要写 *C语言*

1000 行代码，Java 只需要写 100 行，而 Python 可能只要 20 行。

- 解释性语言：运行起来很慢
- 丰富的第三方模块：开源社区非常活跃，贡献了很多强大的第三方库

python 解释器中的 python 语句是一行一行的：

```
>>> print("hello world!")
hello world!
>>> print(100+200)
300
>>>
```

```
#include <stdio.h>
int main()
{
    printf("Hello World!");
    return 0;
}
```

Java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

python

```
print("Hello World!")
```

1. Python 是什么?

- 写起来优雅、快速：完成同一个任务，C 语言要写 1000 行代码，Java 只需要写 100 行，而 Python 可能只要 20 行。
- 解释性语言：运行起来很慢
- 丰富的第三方模块：开源社区非常活跃，贡献了很多强大的第三方库

python 解释器中的 python 语句是一行一行的：

```
>>> print("hello world!")
hello world!
>>> print(100+200)
300
>>>
```

C语言

```
#include <stdio.h>
int main()
{
    printf("Hello World!");
    return 0;
}
```

Java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

python

```
print("Hello World!")
```

1. Python 是什么?

- 写起来优雅、快速：完成同一个任务，C 语言要写 *C语言*

1000 行代码，Java 只需要写 100 行，而 Python 可能只要 20 行。

- 解释性语言：运行起来很慢
- 丰富的第三方模块：开源社区非常活跃，贡献了很多强大的第三方库

python 解释器中的 python 语句是一行一行的：

```
>>> print("hello world!")
hello world!
>>> print(100+200)
300
>>>
```

```
#include <stdio.h>
int main()
{
    printf("Hello World!");
    return 0;
}
```

Java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

python

```
print("Hello World!")
```


解释性语言

意味着你的`.py`文件中的代码在执行时会一行一行地翻译成 CPU 能理解的机器码，这个翻译过程非常耗时，所以很慢。而 C 程序是运行前直接编译成 CPU 能执行的机器码，所以非常快。

此外，运行`.py`文件和在 Python 交互式环境下直接运行 Python 代码一点点不同。Python 交互式环境会把每一行 Python 代码的结果自动打印出来，但是，直接运行 Python 代码却不会。

但是，看在开发 python 如此舒适的分上，运行慢一点完全可以接受啊。



2. 基本语法详解

“#”是注释的意思，它后面的文字会被 Python 忽略。

2.0 变量

```
>>> a = 100
>>> b = 3.1415926
>>> c = "一串字符串，里面可以是任意文本"
>>> a + b # 103.1415926
>>> a = 10 # a的值可以改变
>>> a + b # 13.1415926
>>> # python 强大的计算能力：
>>> a = -8080
>>> b = 1.234e9
>>> b
1234000000.0
>>> 2**100 # 2的100次方
1267650600228229401496703205376
>>> b**0.5 # 对b开根
35128.33614050059
>>> a + b
1233991920.0
>>> b/a
-152722.77227722772
>>>
```

那么如果我想让 c 的内容是 `"hello"` 怎么办？注意到 c 头尾两个字符是 python 默认「定义字符串」用的双引号，难道这样写吗？

```
>>> c = "hello"
File "<stdin>", line 1
    c = "hello"
        ^^^^^
SyntaxError: invalid syntax
```

- 包裹字符串的双引号可以换成单引号，这样就没有歧义了：`c = 'hello'`
- 使用转义字符 `\"`

```
>>> c = "\"hello\""
>>> c
'"hello"'
```

2. 基本语法详解

“#”是注释的意思，它后面的文字会被 Python 忽略。

2.0 变量

```
>>> a = 100
>>> b = 3.1415926
>>> c = "一串字符串，里面可以是任意文本"
>>> a + b # 103.1415926
>>> a = 10 # a的值可以改变
>>> a + b # 13.1415926
>>> # python 强大的计算能力：
>>> a = -8080
>>> b = 1.234e9
>>> b
1234000000.0
>>> 2**100 # 2的100次方
1267650600228229401496703205376
>>> b**0.5 # 对b开根
35128.33614050059
>>> a + b
1233991920.0
>>> b/a
-152722.77227722772
>>>
```

那么如果我想让 c 的内容是 `"hello"` 怎么办？注意到 c 头尾两个字符是 python 默认「定义字符串」用的双引号，难道这样写吗？

```
>>> c = ""hello""
File "<stdin>", line 1
    c = ""hello""
        ^^^^^
SyntaxError: invalid syntax
```

- 包裹字符串的双引号可以换成单引号，这样就没有歧义了：`c = 'hello'`
- 使用转义字符 `\"`

```
>>> c = "\"hello\""
>>> c
'"hello"'
```

转义字符对照表

转义字符	意义	ASCII
<code>\0</code>	空字符(NUL)	0
<code>\t</code>	制表符	9
<code>\n</code>	换行符	10
<code>\"</code>	代表一个双引号字符	34
<code>'</code>	代表一个单引号	39
<code>\\</code>	代表一个反斜线字符'	92

```
>>> print("\\")  
\  

```

数据类型

我们可以使用`type()`来查看变量的数据类型：

```
>>> type(a) # 整数
<class 'int'>
>>> type(b) # 浮点数
<class 'float'>
>>> type(c) # 字符串
<class 'str'>
>>> type(True) # 布尔值
<class 'bool'>
```

请不要把赋值语句的等号等同于数学的等号

```
x = 10
x = x + 2 # 12
```

「布尔值」一定要注意 True 和 False 的首字母大写！

布尔值经常会用到条件判断语句中，后面会提到。

```
>>> 3 > 2
True
>>> 3 > 5
False
>>> True and True # 与运算
True
>>> True and False
False
>>> False and False
False
>>> 5 > 3 and 3 > 1
True
>>> True or True # 或运算
True
>>> True or False
True
>>> False or False
False
>>> 5 > 3 or 1 > 3
True
>>> not True # 非运算
False
>>> not False
True
>>> not 1 > 2
True
```

数据类型

我们可以使用`type()`来查看变量的数据类型：

```
>>> type(a) # 整数
<class 'int'>
>>> type(b) # 浮点数
<class 'float'>
>>> type(c) # 字符串
<class 'str'>
>>> type(True) # 布尔值
<class 'bool'>
```

请不要把赋值语句的等号等同于数学的等号

```
x = 10
x = x + 2 # 12
```

「布尔值」一定要注意 True 和 False 的首字母大写！

布尔值经常会用到条件判断语句中，后面会提到。

```
>>> 3 > 2
True
>>> 3 > 5
False
>>> True and True # 与运算
True
>>> True and False
False
>>> False and False
False
>>> 5 > 3 and 3 > 1
True
>>> True or True # 或运算
True
>>> True or False
True
>>> False or False
False
>>> 5 > 3 or 1 > 3
True
>>> not True # 非运算
False
>>> not False
True
>>> not 1 > 2
True
```

数据类型

我们可以使用`type()`来查看变量的数据类型：

```
>>> type(a) # 整数
<class 'int'>
>>> type(b) # 浮点数
<class 'float'>
>>> type(c) # 字符串
<class 'str'>
>>> type(True) # 布尔值
<class 'bool'>
```

请不要把赋值语句的等号等同于数学的等号

```
x = 10
x = x + 2 # 12
```

「布尔值」一定要注意 True 和 False 的首字母大写！

布尔值经常会用到条件判断语句中，后面会提到。

```
>>> 3 > 2
True
>>> 3 > 5
False
>>> True and True # 与运算
True
>>> True and False
False
>>> False and False
False
>>> 5 > 3 and 3 > 1
True
>>> True or True # 或运算
True
>>> True or False
True
>>> False or False
False
>>> 5 > 3 or 1 > 3
True
>>> not True # 非运算
False
>>> not False
True
>>> not 1 > 2
True
```

数据类型

我们可以使用`type()`来查看变量的数据类型：

```
>>> type(a) # 整数
<class 'int'>
>>> type(b) # 浮点数
<class 'float'>
>>> type(c) # 字符串
<class 'str'>
>>> type(True) # 布尔值
<class 'bool'>
```

请不要把赋值语句的等号等同于数学的等号

```
x = 10
x = x + 2 # 12
```

「布尔值」一定要注意 True 和 False 的首字母大写！

布尔值经常会用到条件判断语句中，后面会提到。

```
>>> 3 > 2
True
>>> 3 > 5
False
>>> True and True # 与运算
True
>>> True and False
False
>>> False and False
False
>>> 5 > 3 and 3 > 1
True
>>> True or True # 或运算
True
>>> True or False
True
>>> False or False
False
>>> 5 > 3 or 1 > 3
True
>>> not True # 非运算
False
>>> not False
True
>>> not 1 > 2
True
```


数据类型

我们可以使用`type()`来查看变量的数据类型：

```
>>> type(a) # 整数
<class 'int'>
>>> type(b) # 浮点数
<class 'float'>
>>> type(c) # 字符串
<class 'str'>
>>> type(True) # 布尔值
<class 'bool'>
```

请不要把赋值语句的等号等同于数学的等号

```
x = 10
x = x + 2 # 12
```

「布尔值」一定要注意 True 和 False 的首字母大写！

布尔值经常会用到条件判断语句中，后面会提到。

```
>>> 3 > 2
True
>>> 3 > 5
False
>>> True and True # 与运算
True
>>> True and False
False
>>> False and False
False
>>> 5 > 3 and 3 > 1
True
>>> True or True # 或运算
True
>>> True or False
True
>>> False or False
False
>>> 5 > 3 or 1 > 3
True
>>> not True # 非运算
False
>>> not False
True
>>> not 1 > 2
True
```

2.1 输入输出

输出

```
print("100+200 =", 100+200)
# 100+200 = 300
```

输入

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

在命令行中运行：

```
~/myWorkspace> python hello.py
please enter your name: Michael
hello, Michael
```

2.2 LIST（数组）、TUPLE（元组）与 DICT（字典）

```
>>> # `len()`函数可以获得 list 元素的个数:
>>> numbers = [2, 4, 7, 3, 9, 10]
>>> len(numbers)
6
>>> classmates = ['Michael', 'Bob', 'Tracy']
>>> classmates
['Michael', 'Bob', 'Tracy']
>>> classmates.append('Adam')
>>> classmates
['Michael', 'Bob', 'Tracy', 'Adam']
>>> classmates.insert(1, 'Jack')
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
>>> classmates.pop()
'Adam'
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy']
>>> classmates.pop(1)
'Jack'
>>> classmates
['Michael', 'Bob', 'Tracy']
>>> classmates[1] = 'Sarah'
>>> classmates
['Michael', 'Sarah', 'Tracy']
```

用「索引」来访问 list 中每一个位置的元素，记得索引是从 0 开始的：

```
>>> classmates[0]
'Michael'
>>> classmates[1]
'Bob'
>>> classmates[2]
'Tracy'
>>> classmates[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

字符串也可以用索引：

```
>>> c = "hello"
>>> c[0]
'h'
>>> c[-1]
'o'
>>> c[1]
'e'
>>> c[2]
'l'
>>> c[3]
'l'
>>> c[4]
'o'
```

2.2 LIST（数组）、TUPLE（元组）与 DICT（字典）

```
>>> # `len()`函数可以获得 list 元素的个数:
>>> numbers = [2, 4, 7, 3, 9, 10]
>>> len(numbers)
6
>>> classmates = ['Michael', 'Bob', 'Tracy']
>>> classmates
['Michael', 'Bob', 'Tracy']
>>> classmates.append('Adam')
>>> classmates
['Michael', 'Bob', 'Tracy', 'Adam']
>>> classmates.insert(1, 'Jack')
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
>>> classmates.pop()
'Adam'
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy']
>>> classmates.pop(1)
'Jack'
>>> classmates
['Michael', 'Bob', 'Tracy']
>>> classmates[1] = 'Sarah'
>>> classmates
['Michael', 'Sarah', 'Tracy']
```

用「索引」来访问 list 中每一个位置的元素，记得索引是从 0 开始的：

```
>>> classmates[0]
'Michael'
>>> classmates[1]
'Bob'
>>> classmates[2]
'Tracy'
>>> classmates[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

字符串也可以用索引：

```
>>> c = "hello"
>>> c[0]
'h'
>>> c[-1]
'o'
>>> c[1]
'e'
>>> c[2]
'l'
>>> c[3]
'l'
>>> c[4]
'o'
```

2.2 LIST（数组）、TUPLE（元组）与 DICT（字典）

```
>>> # `len()`函数可以获得 list 元素的个数:
>>> numbers = [2, 4, 7, 3, 9, 10]
>>> len(numbers)
6
>>> classmates = ['Michael', 'Bob', 'Tracy']
>>> classmates
['Michael', 'Bob', 'Tracy']
>>> classmates.append('Adam')
>>> classmates
['Michael', 'Bob', 'Tracy', 'Adam']
>>> classmates.insert(1, 'Jack')
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
>>> classmates.pop()
'Adam'
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy']
>>> classmates.pop(1)
'Jack'
>>> classmates
['Michael', 'Bob', 'Tracy']
>>> classmates[1] = 'Sarah'
>>> classmates
['Michael', 'Sarah', 'Tracy']
```

用「索引」来访问 list 中每一个位置的元素，记得索引是从 0 开始的：

```
>>> classmates[0]
'Michael'
>>> classmates[1]
'Bob'
>>> classmates[2]
'Tracy'
>>> classmates[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

字符串也可以用索引：

```
>>> c = "hello"
>>> c[0]
'h'
>>> c[-1]
'o'
>>> c[1]
'e'
>>> c[2]
'l'
>>> c[3]
'l'
>>> c[4]
'o'
```

2.2 LIST（数组）、TUPLE（元组）与 DICT（字典）

```
>>> # `len()`函数可以获得 list 元素的个数:
>>> numbers = [2, 4, 7, 3, 9, 10]
>>> len(numbers)
6
>>> classmates = ['Michael', 'Bob', 'Tracy']
>>> classmates
['Michael', 'Bob', 'Tracy']
>>> classmates.append('Adam')
>>> classmates
['Michael', 'Bob', 'Tracy', 'Adam']
>>> classmates.insert(1, 'Jack')
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
>>> classmates.pop()
'Adam'
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy']
>>> classmates.pop(1)
'Jack'
>>> classmates
['Michael', 'Bob', 'Tracy']
>>> classmates[1] = 'Sarah'
>>> classmates
['Michael', 'Sarah', 'Tracy']
```

用「索引」来访问 list 中每一个位置的元素，记得索引是从 0 开始的：

```
>>> classmates[0]
'Michael'
>>> classmates[1]
'Bob'
>>> classmates[2]
'Tracy'
>>> classmates[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

字符串也可以用索引：

```
>>> c = "hello"
>>> c[0]
'h'
>>> c[-1]
'o'
>>> c[1]
'e'
>>> c[2]
'l'
>>> c[3]
'l'
>>> c[4]
'o'
```

元组(TUPLE)

Tuple 和 List 几乎一样，也是一个有序列表。

定义方式也非常类似：

```
numbers_list = [2, 4, 7, 3, 9, 10]
tuple_list = (2, 4, 7, 3, 9, 10)
```

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

换句话说，Tuple无法被更改，只能查看 Tuple中的元素！

这样做的好处在于，tuple 不可变，所以代码更安全。如果可能，能用 tuple 代替 list 就尽量用 tuple。

字典(DICT)

字典以键值对（key-value）的形式存储数据：

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

通过 key 放入：

```
>>> d['Adam'] = 67
>>> d['Adam']
67
>>> d['Jack'] = 90
>>> d['Jack']
90
>>> d['Jack'] = 88
>>> d['Jack']
88
>>> d['Thomas'] # 如果key不存在，dict就会报错：
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Thomas'
```

元组(TUPLE)

Tuple 和 List 几乎一样，也是一个有序列表。

定义方式也非常类似：

```
numbers_list = [2, 4, 7, 3, 9, 10]
tuple_list = (2, 4, 7, 3, 9, 10)
```

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

换句话说，Tuple无法被更改，只能查看 Tuple中的元素！

这样做的好处在于，tuple 不可变，所以代码更安全。如果可能，能用 tuple 代替 list 就尽量用 tuple。

字典(DICT)

字典以键值对（key-value）的形式存储数据：

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

通过 key 放入：

```
>>> d['Adam'] = 67
>>> d['Adam']
67
>>> d['Jack'] = 90
>>> d['Jack']
90
>>> d['Jack'] = 88
>>> d['Jack']
88
>>> d['Thomas'] # 如果key不存在，dict就会报错：
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Thomas'
```


元组(TUPLE)

Tuple 和 List 几乎一样，也是一个有序列表。

定义方式也非常类似：

```
numbers_list = [2, 4, 7, 3, 9, 10]
tuple_list = (2, 4, 7, 3, 9, 10)
```

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

换句话说，Tuple无法被更改，只能查看 Tuple中的元素！

这样做的好处在于，tuple 不可变，所以代码更安全。如果可能，能用 tuple 代替 list 就尽量用 tuple。

字典(DICT)

字典以键值对（key-value）的形式存储数据：

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

通过 key 放入：

```
>>> d['Adam'] = 67
>>> d['Adam']
67
>>> d['Jack'] = 90
>>> d['Jack']
90
>>> d['Jack'] = 88
>>> d['Jack']
88
>>> d['Thomas'] # 如果key不存在，dict就会报错：
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Thomas'
```

2.3. 条件判断

用 if 语句实现输入用户年龄，根据年龄打印不同的内容：

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

注意：

- `>=` 在 python 中用 `>=` 替代
- 不要少写了冒号！！

```
age = 3
if age >= 18:
    print('adult')
elif age >= 6:
    print('teenager')
else:
    print('kid')
```

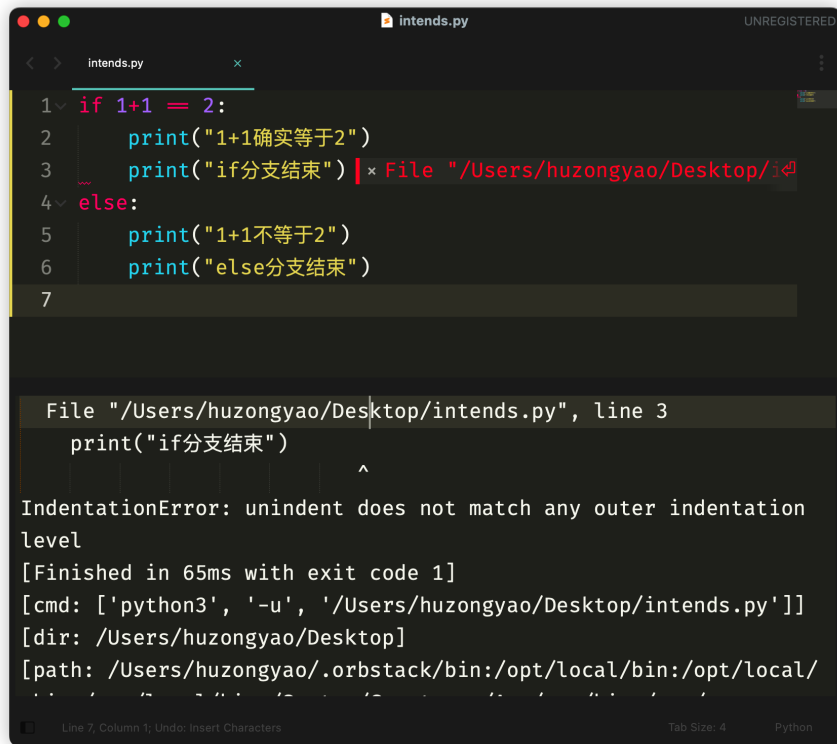
一般形式（条件判断算出来的就是之前提到的布尔值）

```
if <条件判断1>:
    <执行1>
elif <条件判断2>:
    <执行2>
elif <条件判断3>:
    <执行3>
else:
    <执行4>
```

其中 elif, else 都是可有可无的，就是说删了 elif 这个条件判断分支，代码也不会报错。

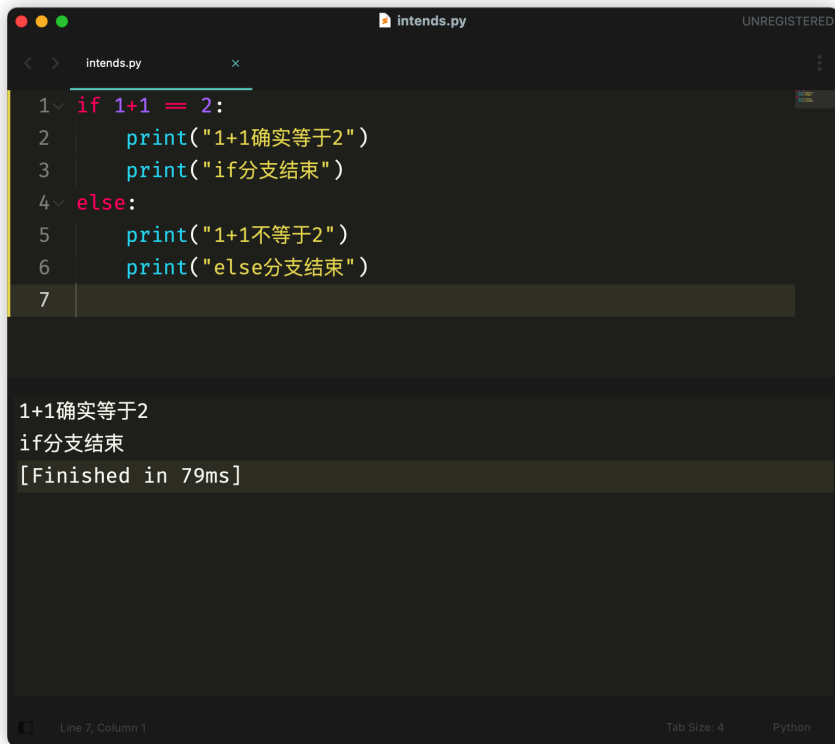
2.4. PYTHON 的严格缩进机制

这两段代码有什么不同？ — Python的缩进非常严格【大坑】。



```
1 if 1+1 == 2:
2     print("1+1确实等于2")
3     print("if分支结束")
4 else:
5     print("1+1不等于2")
6     print("else分支结束")
7
```

File "/Users/huzongyao/Desktop/intends.py", line 3
print("if分支结束")
^
IndentationError: unindent does not match any outer indentation level
[Finished in 65ms with exit code 1]
[cmd: ['python3', '-u', '/Users/huzongyao/Desktop/intends.py']]
[dir: /Users/huzongyao/Desktop]
[path: /Users/huzongyao/.orbstack/bin:/opt/local/bin:/opt/local/



```
1 if 1+1 == 2:
2     print("1+1确实等于2")
3     print("if分支结束")
4 else:
5     print("1+1不等于2")
6     print("else分支结束")
7
```

1+1确实等于2
if分支结束
[Finished in 79ms]

2.5. 循环

为了让计算机能计算成千上万次的重复运算，我们就需要循环语句。

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

执行这段代码，会依次打印 names 的每一个元素：

```
Michael
Bob
Tracy
```

所以`for x in ...`循环就是把每个元素代入变量 x，然后执行后面的语句。

`range(101)`是使用 python 提供的`range()`函数，生成 0-100 的整数序列（0-100 嘛，101 个数），你可以理解为用`range(10)`代替了`[0,1,2,3,4,5,6,7,8,9]`

```
sum = 0
# 生成涵盖0~100的数组
for x in range(101):
    sum = sum + x
print(sum)
# 5050
```

除了 for 循环外，还有 while 循环：

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

这里`while n > 0:`表示，只要满足 $n > 0$ ，循环就会一直进行下去，直到发现 $n \leq 0$ 了，才退出循环

3. 函数

3.0 函数简介

上面其实已经出现了很多函数了，但都是 python 内置的

- `len()`: 用于计算数组的长度
- `range()`: 生成整数序列
- `print()`: 把东西打印到控制台
- `type()`: 来查看变量的数据类型

Q: `list.pop()` 不是删除 List 末尾的一个元素嘛？那 `pop` 算函数吗？

A: `pop()` 单独用会报错啊！只有 `pop()` 前面加一个点，变成 `numbers.pop()` 才能正常运行。这是因为 `pop` 是「数组」这个数据类型提供的一个「方法」，如果你深入了解的话，会在面向对象中学到。虽然长得像函数，定义也很像，但是得做出区分。

3.1 使用函数

试试看直接调用 python 的内置函数之一，abs 函数：

```
>>> abs(100)
100
>>> abs(-20)
20
>>> abs(12.34)
12.34
```

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，但给出了两个：

```
>>> abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
```

如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报 ``TypeError`` 的错误，并且给出错误信息：``str` 是错误的参数类型``：

```
>>> abs('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

而 max 函数 ``max()`` 可以接收任意多个参数，并返回最大的那个：

```
>>> max(1, 2)
2
>>> max(2, 3, 1, -5)
3
```

3.2 定义函数

我们以自定义一个求绝对值的 my_abs 函数为例：

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

很好理解，因为 def 就是 define（定义）的意思。就是这里有个 return 需要理解一下，众所周知，一个函数不能有两个返回值，高中数学教的对吧？所以，一个函数只要执行了一个 return，就停下来了，不可能执行其他 return 了。

使用方法：

```
>>> # 调用函数计算两点之间的距离  
>>> distance = calculate_distance(1, 2, 4, 6)  
>>> print("两点之间的距离为:", distance) # 打印结果: 5.0  
>>> my_abs(-9)  
9
```

如果函数有多个输入也是一样的（我们把函数的输入称为参数）：

```
def calculate_distance(x1, y1, x2, y2):  
    """  
    计算两点之间的距离
```

参数：

x1 (float): 第一个点的 x 坐标
y1 (float): 第一个点的 y 坐标
x2 (float): 第二个点的 x 坐标
y2 (float): 第二个点的 y 坐标

返回值：

float: 两点之间的距离
"""

计算 x 轴上的差值

x_diff = x2 - x1

计算 y 轴上的差值

y_diff = y2 - y1

计算两点之间的直线距离

```
distance = (x_diff ** 2 + y_diff ** 2) ** 0.5  
return distance
```

4. 使用模块

Python 本身就内置了很多非常有用的模块，只要安装完毕，这些模块就可以立刻使用。

4.0 用 MATH 模块

游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的坐标：

```
# 导入math包，import就是进口的意思
import math

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

「import math」语句表示导入 math 包，并允许后续代码引用 math 包里的「sin()」、「cos()」等函数。

发现什么问题没有？之前不是说函数不可能有两个返回值吗？这是什么？「return nx, ny」这不是两个吗？

```
>>> r = move(100, 100, 60, math.pi / 6)
>>> print(r)
(151.96152422706632, 70.0)
```

原来返回值是一个 tuple！

但是在 tuple 里面一个个取值太麻烦了，于是 python 提供了一个可以让写代码舒适度提高很多的语法：

```
>>> x, y = move(100, 100, 60, math.pi / 6)
>>> print(x, y)
151.96152422706632 70.0
```

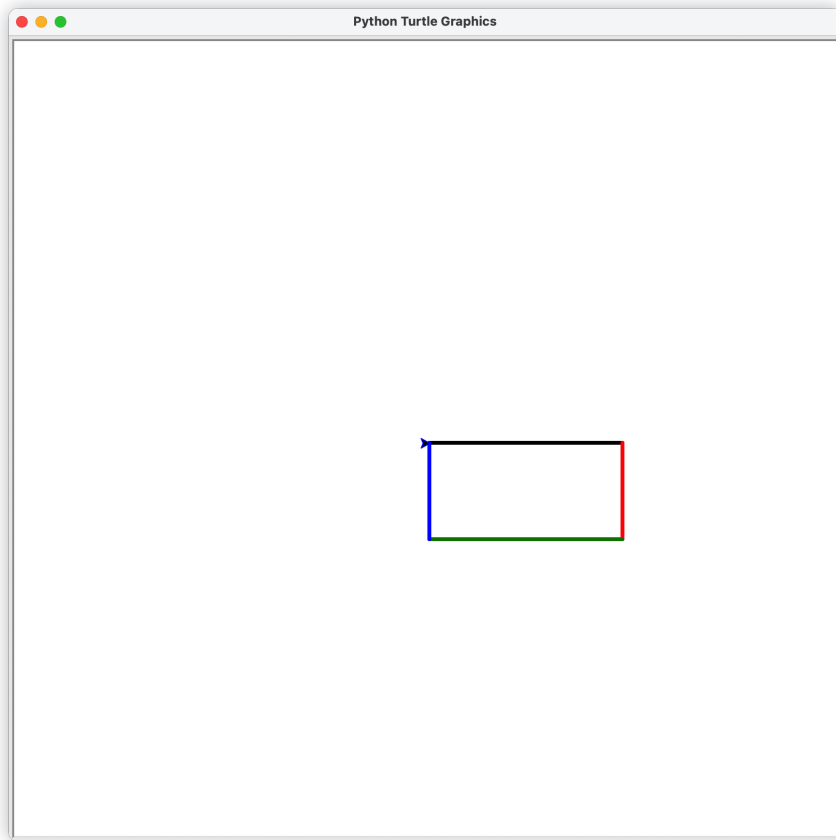

4.1 使用 TURTLE 模块

`turtle` 就是鼎鼎有名的海龟作图模块。

```
# 导入turtle包
import turtle
# 设置笔刷宽度:
turtle.width(4)
# 前进:
turtle.forward(200)
# 右转90度:
turtle.right(90)
# 设置笔刷颜色:
turtle.pencolor('red')
turtle.forward(100)
turtle.right(90)

turtle.pencolor('green')
turtle.forward(200)
turtle.right(90)

turtle.pencolor('blue')
turtle.forward(100)
turtle.right(90)
# turtle.调用done()使得窗口等待被关闭, 否则将立刻关闭窗口:
turtle.done()
```



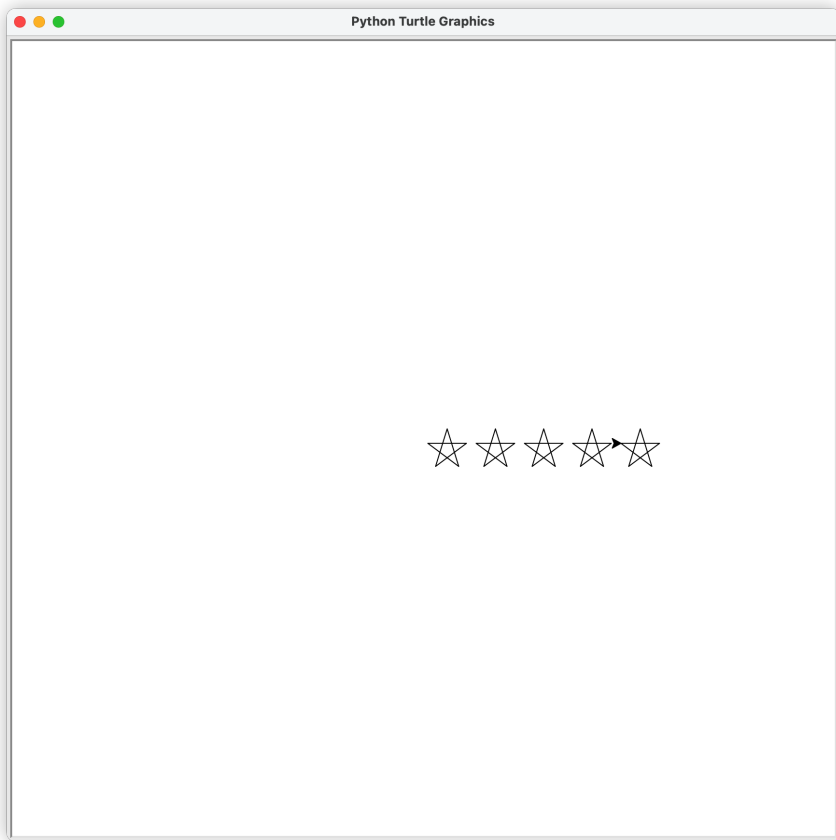
‘Turtle’ 包本身只是一个绘图库，但是配合 Python 代码，比如循环、函数等就可以绘制各种复杂的图形：

```
import turtle

def drawStar(x, y):
    """
    这一个在(x, y)坐标画出一个五角星的函数
    """
    # 抬起画笔，不留轨迹
    turtle.penup()
    # 去给定坐标
    turtle.goto(x, y)
    # 落下画笔，开始绘画
    turtle.pendown()
    # 设置箭头朝向（0代表向正右方）
    turtle.setheading(0)
    for i in range(5):
        turtle.forward(40)
        # 向右旋转144度
        turtle.right(144)

for x in range(0, 250, 50):
    drawStar(x, 0)

turtle.done()
```



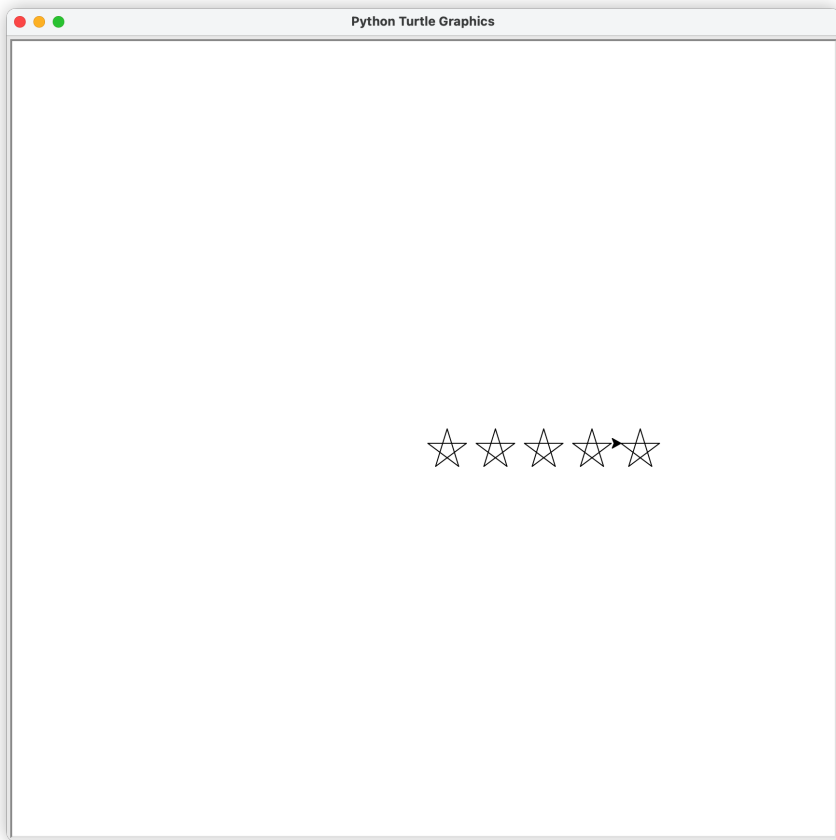
‘Turtle’ 包本身只是一个绘图库，但是配合 Python 代码，比如循环、函数等就可以绘制各种复杂的图形：

```
import turtle

def drawStar(x, y):
    """
    这一个在(x, y)坐标画出一个五角星的函数
    """
    # 抬起画笔，不留轨迹
    turtle.penup()
    # 去给定坐标
    turtle.goto(x, y)
    # 落下画笔，开始绘画
    turtle.pendown()
    # 设置箭头朝向（0代表向正右方）
    turtle.setheading(0)
    for i in range(5):
        turtle.forward(40)
        # 向右旋转144度
        turtle.right(144)

for x in range(0, 250, 50):
    drawStar(x, 0)

turtle.done()
```



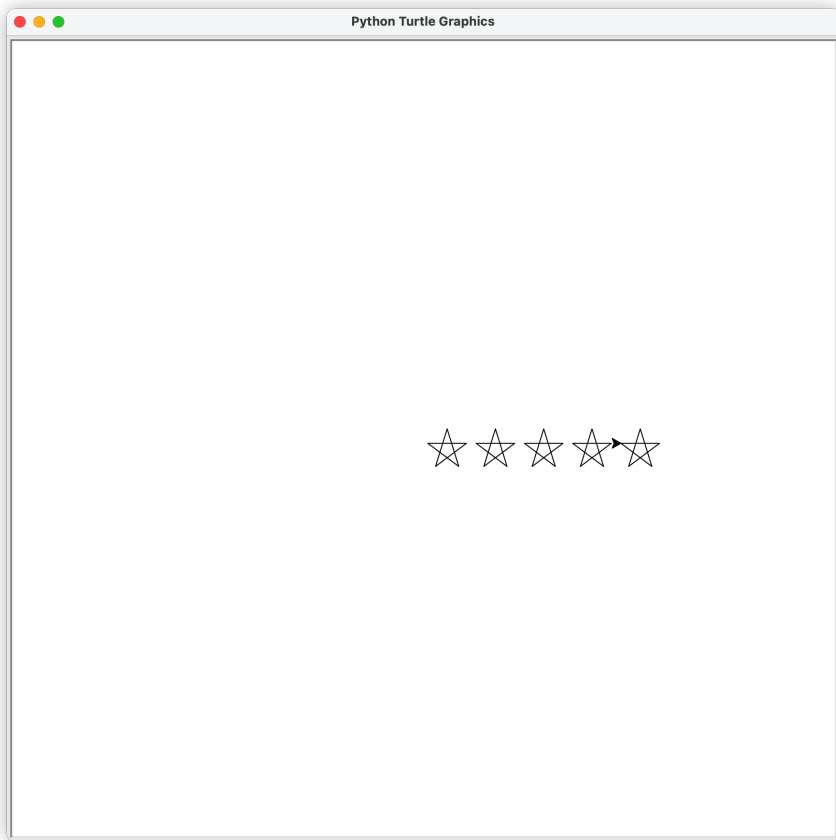
‘Turtle’包本身只是一个绘图库，但是配合 Python 代码，比如循环、函数等就可以绘制各种复杂的图形：

```
import turtle

def drawStar(x, y):
    """
    这一个在(x, y)坐标画出一个五角星的函数
    """
    # 抬起画笔，不留轨迹
    turtle.penup()
    # 去给定坐标
    turtle.goto(x, y)
    # 落下画笔，开始绘画
    turtle.pendown()
    # 设置箭头朝向（0代表向正右方）
    turtle.setheading(0)
    for i in range(5):
        turtle.forward(40)
        # 向右旋转144度
        turtle.right(144)

for x in range(0, 250, 50):
    drawStar(x, 0)

turtle.done()
```



4.2 安装第三方模块

所有的第三方模块都会在PyPI上注册。

比如爬虫必备的 requests 库。没有安装的时候，运行 import 语句会报错：

```
>>> import requests
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'requests'
>>>
```

打开命令提示符（终端），输入这行命令以后，如果没有报错就是安装完成了。

```
~$ pip install requests
...
Successfully installed certifi-2023.7.22
charset-normalizer-3.3.1 idna-3.4
requests-2.31.0 urllib3-2.0.7
```

注意！！如果出现红色的字，或者没有看到 `Successfully installed requests`，说明安装失败了，此时 90%以上的可能是“网络问题”，Pypi 服务器有点不太稳定。因此我们可以用 Pypi 镜像站代替。

- 清华：<https://pypi.tuna.tsinghua.edu.cn/simple/>
- 阿里云：<http://mirrors.aliyun.com/pypi/simple/>
- 中国科技大学：<https://pypi.mirrors.ustc.edu.cn/simple/>
- 华中科技大学：<http://pypi.hustunique.com/simple/>
- 上海交通大学：<https://mirror.sjtu.edu.cn/pypi/web/simple/>
- 豆瓣：<http://pypi.douban.com/simple/>

```
~$ pip install -i https://pypi.tuna.tsinghua.edu.cn/simple requests
```

安装成功后再次 import 就能看到结果了：

```
>>> import requests
>>>
```

Part2. Git 入门

Part3. GitHub 使用完全教程