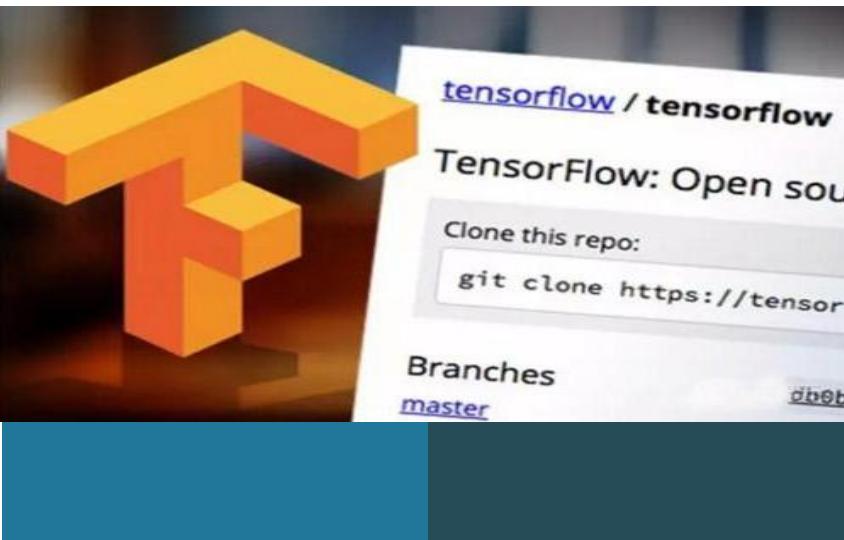




03 Python语言基础

西安科技大学 牟琦
muqi@xust.edu.cn



3.1 初识Python

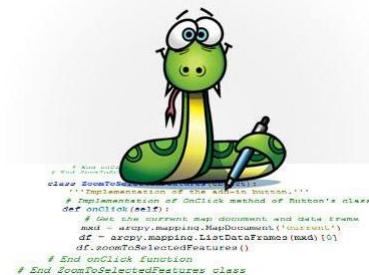
3.1 初识Python



- 1989, Guido Van Rossu



Monty Python's Flying Circus



```
# Start zoomToSelectedFeatures class
class ZoomToSelectedFeatures(object):
    """Implementation of the zoomToSelectedFeatures method.
    # Implementation of OnClick method of Python's class
    def onClick(self):
        # Get current map document and data frame
        md = arcpy.mapping.MapDocument("CURRENT")
        df = arcpy.mapping.ListDataFrames(md)[0]
        df.zoomToSelectedFeatures()
    # End onClick function
# End ZoomToSelectedFeatures class
```



西安科技大学

计算机科学与技术学院

■ 自由软件 (free software)

Richard Matthew Stallman:

- “**free software**” is a matter of **liberty, not price**. To understand the concept, you should think of “free” as in “**free speech**,” **not as in “free beer”**.
- Free software is a matter of the users' freedom to **run, copy, distribute, study, change and improve** the software.”



口 自由软件 (free software)

■ GNU

GNU's Not Unix!

反对使用专利软件，认为程序需附带源代码

■ copyleft

对源代码进行的所有的改进和修改，改动后的源代码必须公开
保证了自由软件传播的延续性。

■ GUL(General Public License,GNU通用公共许可协议)

可以自由地运行、拷贝、修改和再发行使用GPL授权的软件
但是不允许将修改后和衍生的代码做为私有的商业软件发布和销售



兰州交通大学

计算机科学与技术学院

■ 开源软件 (open source)

允许软件授权收费

软件本身可以以开源免费的方式提供，但是针对软件的服务和维护可以收费

开源社区可以接受来自商业公司的资金支持

■ 免费软件 (freeware)

免费提供给用户使用的软件，通常会有一些限制

源码不一定会公开

使用者也并没有复制、研究、修改和再散布的权利



■ 编译型语言

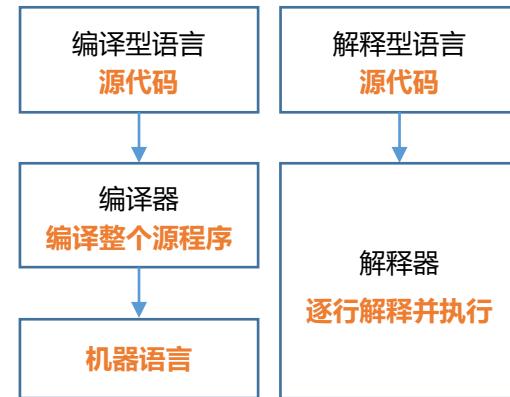
在程序执行之前，编译器将整个高级语言源程序翻译成**机器语言**
运行时直接运行机器语言（可执行程序）

■ 解释型语言

在程序运行时，解释器对源程序逐条翻译，并且逐条执行

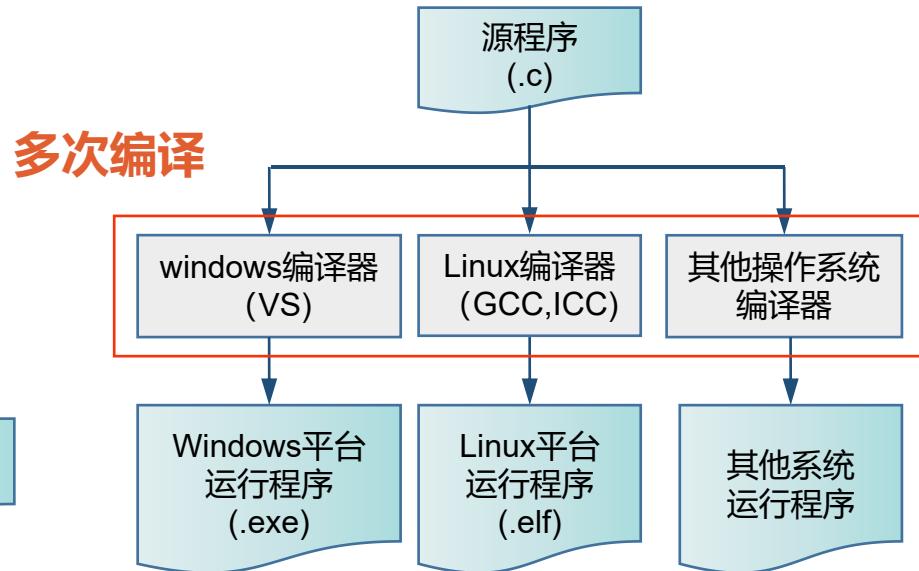
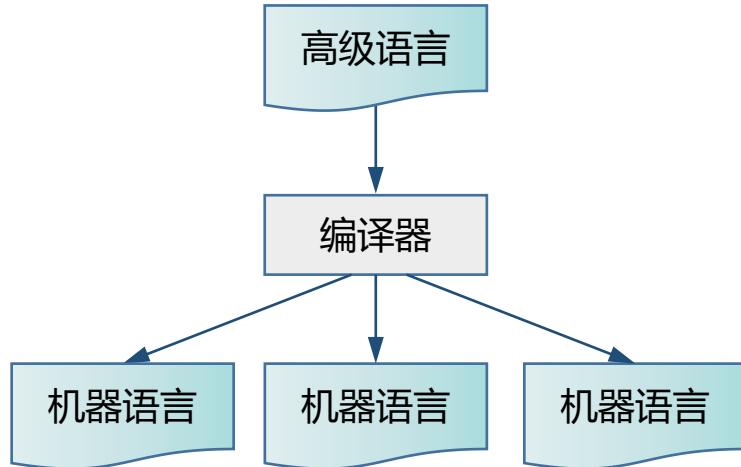
■ Java虚拟机

编译器首先将源代码.java编译成字节码文件.class
程序运行时，再由**Java虚拟机**将字节码翻译为机器语言



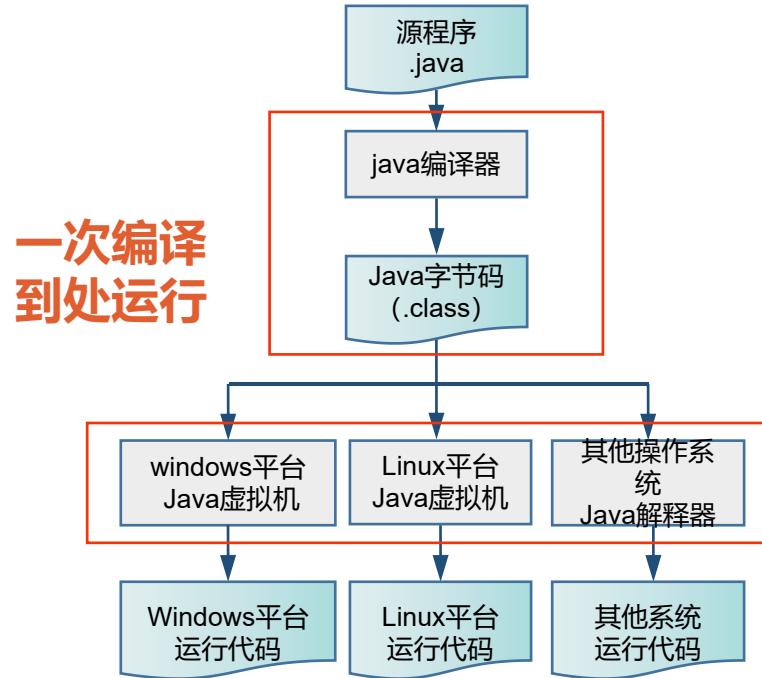
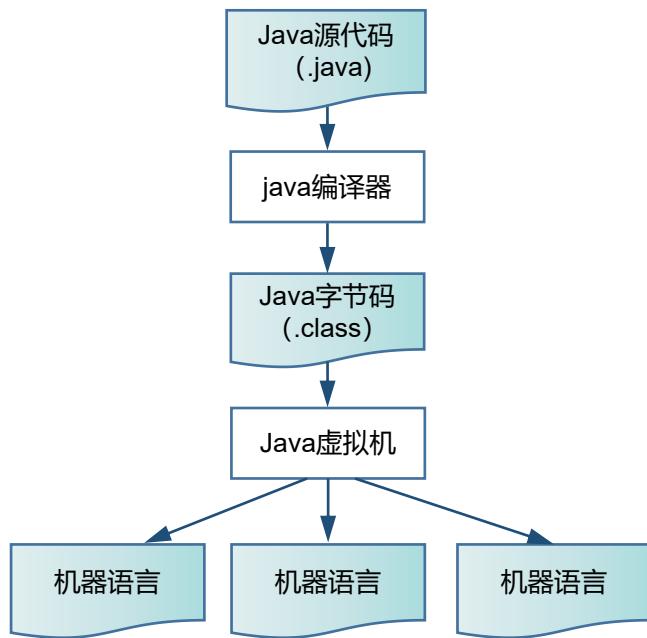
3.1 初识Python

□ C语言——编译型语言



3.1 初识Python

□ Java语言——虚拟机语言

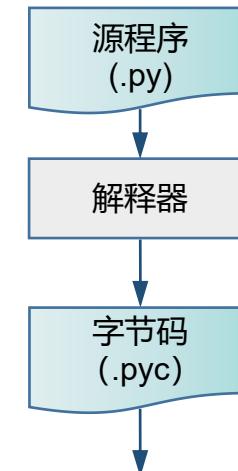


□ Python语言

- 程序首次执行时，Python解释器将源文件.**py**编译成字节码文件.**pyc**
- 当再次执行时，Python解释器加载.pyc文件，对字节码逐行解释执行
- Python会**自动检查时间戳**，当源程序发生了改动，会自动重新创建字节码文件

跨平台，且加快了程序的运行效率

Python中并没有独立的编译系统，仍属于**解释型语言**



■ Python的版本

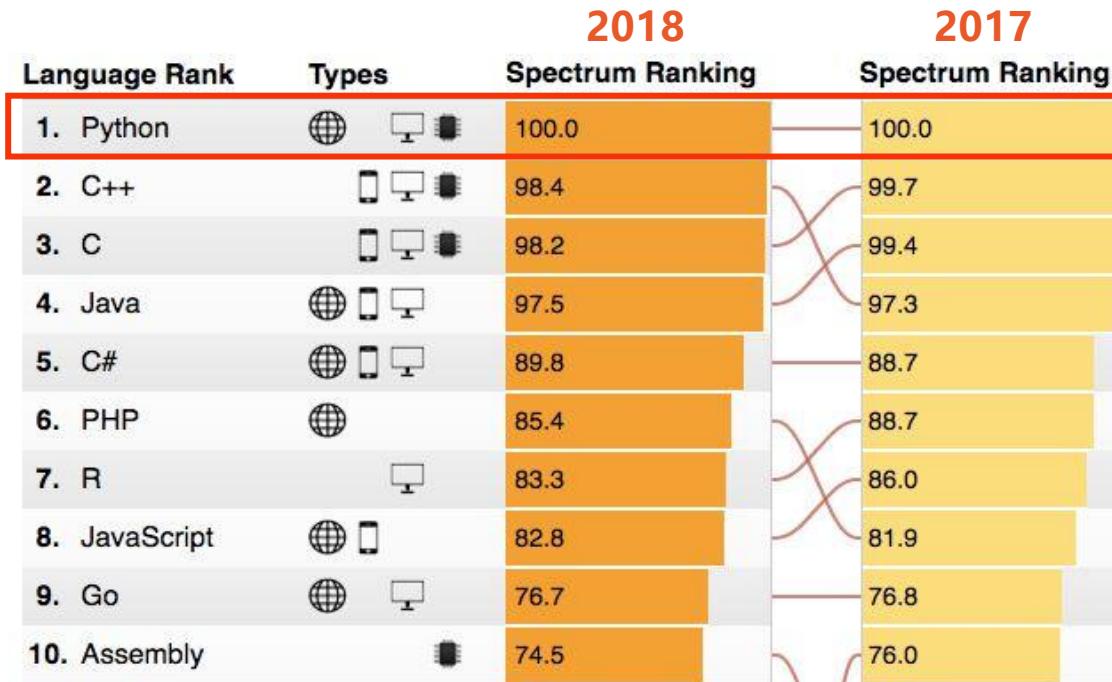
- 1991 V0.9.0
- 1994 V1.0, 新增函数式工具
- 2000-2002 V2.0 内存回收机制, 列表推导式
- 2008-2010 V2.6 V2.7 过渡版本, Python2.0+3.0混搭
- 2008.12 V3.0 不向下兼容

2to3转换工具: <python_root>/tools/scripts/2to3-script.py

```
python 2to3.py -w 文件路径
```



■ IEEE2018、2017顶级编程语言排行榜



■ IEEE2018顶级编程语言排行榜

□ 年度发展最快的编程语言

| Language Rank | Types | Trending Ranking | Trending Ranking |
|---------------|-------|------------------|------------------|
| 1. Python | 🌐💻📱 | 100.0 | 100.0 |
| 2. C++ | 💻🖱️📱 | 96.4 | 98.4 |
| 3. Java | 🌐💻🖱️ | 94.6 | 97.7 |
| 4. C | 💻🖱️📱 | 94.4 | 97.1 |
| 5. Go | 🌐💻 | 85.5 | 88.9 |
| 6. PHP | 🌐 | 80.9 | 87.6 |
| 7. JavaScript | 🌐🖱️ | 80.8 | 86.8 |
| 8. Scala | 🌐🖱️ | 78.6 | 85.4 |
| 9. Ruby | 🌐💻 | 77.2 | 80.2 |
| 10. Assembly | 🖱️ | 75.3 | 80.0 |
| 11. C# | 🌐🖱️💻 | 74.7 | 79.4 |
| 12. HTML | 🌐 | 73.5 | 75.8 |

□ 开源项目钟爱的编程语言

| Language Rank | Types | Open Ranking | Open Ranking |
|---------------|-------|--------------|--------------|
| 1. Python | 🌐💻📱 | 100.0 | 100.0 |
| 2. C++ | 💻🖱️📱 | 95.8 | 97.9 |
| 3. Java | 🌐💻🖱️ | 95.8 | 96.8 |
| 4. C | 💻🖱️📱 | 90.7 | 94.9 |
| 5. C# | 🌐🖱️💻 | 89.7 | 90.4 swift |
| 6. PHP | 🌐 | 88.5 | 88.3 |
| 7. HTML | 🌐 | 88.3 | 88.1 |
| 8. JavaScript | 🌐🖱️ | 88.3 | 85.7 |
| 9. Go | 🌐💻 | 81.9 | 84.7 |
| 10. R | 💻 | 80.5 | 83.2 |
| 11. Shell | 💻 | 80.3 | 82.7 |
| 12. Ruby | 🌐🖱️ | 79.8 | 82.4 |



■ IEEE2018顶级编程语言排行榜

□ 工作环境使用的编程语言

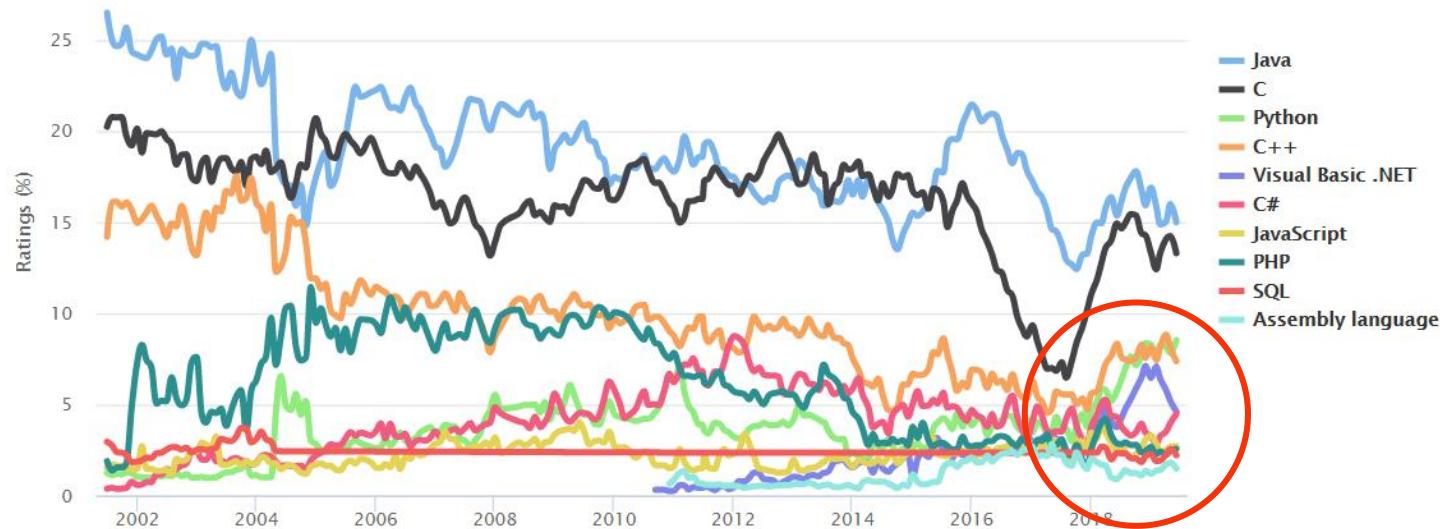
| Language Rank | Types | Jobs Ranking | Jobs Ranking |
|---------------|-------|--------------|--------------|
| 1. Python | 🌐💻📱 | 100.0 | 100.0 |
| 2. C | 💻💻📱 | 99.4 | 99.3 |
| 3. Java | 🌐💻💻 | 99.2 | 99.3 |
| 4. C++ | 💻💻📱 | 94.1 | 92.7 |
| 5. C# | 🌐💻💻 | 86.6 | 90.4 |
| 6. JavaScript | 🌐📱 | 85.8 | 86.6 |
| 7. PHP | 🌐 | 83.7 | 81.4 |
| 8. Assembly | _ASM | 83.7 | 81.0 |
| 9. HTML | 🌐 | 80.5 | 77.8 |
| 10. Scala | 🌐📱 | 76.7 | 77.7 |
| 11. Shell | 💻 | 76.3 | 76.7 |
| 12. Ruby | 🌐💻 | 75.7 | 75.3 |

□ 设计自由度

| Language Rank | Types | Custom Ranking | Custom Ranking |
|---------------|-------|----------------|----------------|
| 1. Python | 🌐💻📱 | 100.0 | 100.0 |
| 2. C | 💻💻📱 | 97.0 | 99.2 |
| 3. C++ | 💻💻📱 | 95.5 | 99.0 |
| 4. Java | 🌐💻💻 | 95.5 | 96.0 |
| 5. JavaScript | 🌐📱 | 82.3 | 88.9 |
| 6. PHP | 🌐 | 82.3 | 84.2 |
| 7. C# | 🌐💻💻 | 80.5 | 82.8 |
| 8. Assembly | _ASM | 75.5 | 81.8 |
| 9. Ruby | 🌐💻 | 75.0 | 81.4 |
| 10. Go | 🌐💻 | 74.8 | 77.8 |
| 11. Scala | 🌐📱 | 74.0 | 75.0 |
| 12. HTML | 🌐 | 73.0 | 74.1 |



■ TIOBE 编程语言排行榜



■ Python的特性——语法简洁，结构清晰，简单易学

Python

```
print("Hello, world !")
```

C

```
#include<studio.h>
int main()
{
    printf("Hello,world !\n");
    return 0;
}
```

Java

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



■ Python的特性——功能强大，资源丰富

- 丰富的**标准库**：网络、文件、GUI、数据库、文本、加密、爬虫、机器学习等
 - 庞大的**第三方库**：(>12万)：NumPy、SciPy、Matplotlib、Pandas、wordcloud
 - 开源、开放体系：世界上最大的、针对单一编程语言的计算生态
-
- 高可扩展性：**胶水语言**
 - 高可移植性：对**底层操作系统**的良好**兼容性**



3.1 初识Python

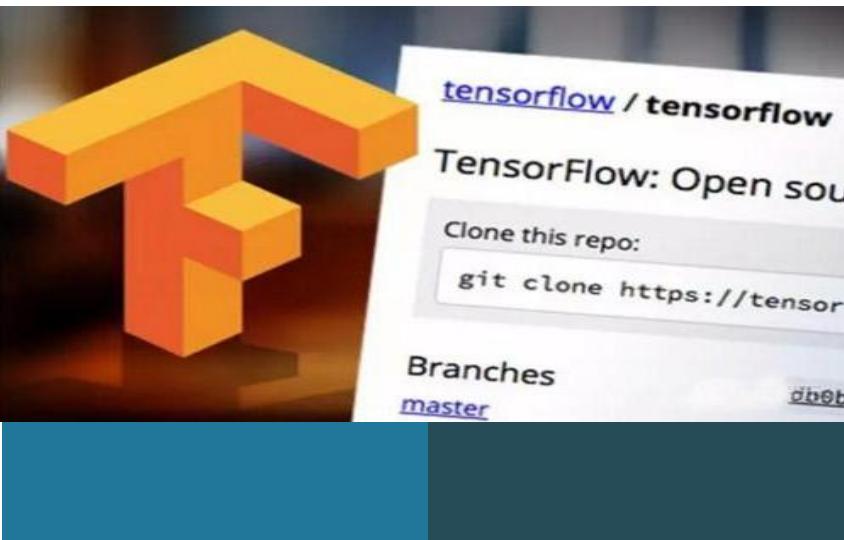


Life is short
you need Python



■ Why Python? Why Now?





3.2 第一个Python程序

3.2 第一个Python程序

□ 例：判断变量 num 是否是正数

```
# 判断变量num是否是正数
num=0
if num > 0:
    print ("num是正数")
else:
    print ("num可能是0")
    print ("num也可能是负数")
```

如果 $num > 0$

输出： num是正数

否则

输出： num可能是0

num也可能是负数



西安科技大学

计算机科学与技术学院

□ 注释语句 & 赋值语句

```
# 判断变量num是否是正数
num=0
if num > 0:
    print ("num是正数")
else:
    print ("num可能是0")
    print ("num也可能是负数")
```

注释语句

赋值语句

是对程序进行说明的语句，
在程序运行过程中不被执行

动态类型语言

- 不需要声明变量的语言
- 变量在使用前必须赋值
- 类型检查在运行阶段完成
- Python、JavaScript、Ruby.....

静态类型语言

- 必须声明变量
- 类型检查在编译阶段完成
- C、C++、Java.....



□ 条件语句 & 语句块

```
# 判断变量num是否是正数
num=0
if num > 0:
    print ("num是正数")
else:
    print ("num可能是0")
    print ("num也可能是负数")
```

条件语句

语句块

Python语句块直接通过代码的缩进来表示

```
if num > 0:
    print ("num是正数")
else:
    print ("num可能是0")
    print ("num也可能是负数")
```

改变语句缩进，程序逻辑发生变化

如果 num>0
输出： num是正数
否则
输出： num可能是0
输出： num也可能是负数



兰州交通大学

计算机科学与技术学院

■ C语言：悬挂else问题

C编译器是忽略缩进，
按照就近原则配对

```
if (a > 0)
    if (b > 0)
        printf("a和b都大于0");
else
    printf("a小于0");
```



```
if (a > 0)
    if (b > 0)
        printf("a和b都大于0");
else
    printf("a小于0");
```



■ Python语言

按照**缩进**来识别语句块，可以有效的避免其他语言中可能出现的**错误配对**问题

```
if a > 0:  
    if b > 0:  
        print ("a和b都大于0")  
else:  
    print ("a小于0")
```

PEP8规范中，
规定语句块的缩进为4个空格

代码缩错误

```
IndentationError: unexpected indent
```

常见错误：混用键盘上的**Tab键**和**空格键**，造成缩进不一致。



兰州交通大学

计算机科学与技术学院

3.2 第一个Python程序

□ 大小写敏感

```
#判断变量num是否是正数
num=0
Num=3
if num > 0:
    print ("num是正数")
else:
    print ("num可能是0")
    print ("num也可能是负数")
```

Num和num是不同的变量

可以将子句写在同一行上

```
num=0
if num > 0: print ("num是正数")
else: print ("num可能是0")
print ("num也可能是负数")
```

运行结果：

num可能是0
num也可能是负数

从新的一行开始打印

print()函数中**自动包含了换行**，
默认每次打印一行



兰州交通大学

计算机科学与技术学院

□ TIPS

Python语句可以以分号结尾

不同的语句可以写在同一行上，以**分号**隔开

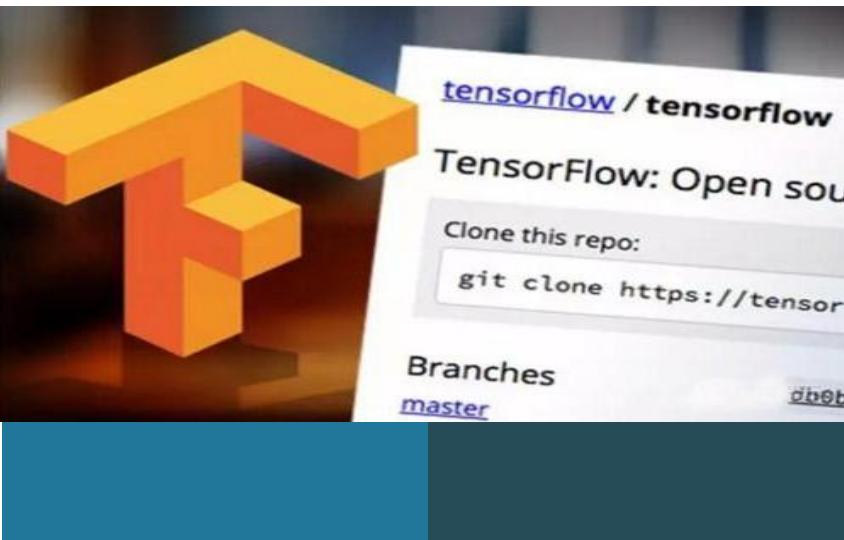
```
a=1  
b=2  
  
a=1;  
b=1;  
  
a=1;b=2  
  
a=1;b=2;
```

这些都是正确的



兰州交通大学

计算机科学与技术学院



3.3 输入和输出

Python语法初步：

- 使用 “#” 作为单行注释符。
- Python变量在使用之前不需要声明。
- 语句块通过代码的缩进来表示。
- 标识符是大小写敏感的。
- print()函数在输出中自动包含换行。



■ 输入函数：

input(提示信息)

接收用户的输入，并以**字符串**类型返回

在屏幕上输出提示信息，并等待键盘输入，
接收到的输入将被存储在字符串string中

提示信息

```
string = input("Please input some words:")
```



兰州交通大学

计算机科学与技术学院

□ 接收用户输入

■ 命令行

```
>>> string = input("Please input some words:")
Please input some words:

>>> string = input("Please input some words:")
Please input some words:Hello!
```

提示信息

用户输入

■ Jupyter Notebook

```
In [*]: string = input("Please input some words:")
```

Please input some words:

```
In [*]: string = input("Please input some words:")
```

Please input some words

Hello!

提示信息

用户输入



兰州科技大学

计算机科学与技术学院

3.3 输入和输出

```
# 判断变量num是否是正数
num=0
if num > 0:
    print ("num是正数")
else:
    print ("num可能是0")
    print ("num也可能是负数")
```

→ num=input("请输入一个整数: ")

类型错误

原因: input函数的返回值是字符串类型

TypeError: unorderable types: str() > int()

类型转换函数

→ num=int(input("请输入一个整数: "))

```
#判断变量num是否是正数
num=int(input("请输入一个整数: "))
print ("您输入的整数是: %d" %(num))
if num > 0:
    print ("num是正数")
else:
    print ("num可能是0")
    print ("num也可能是负数")
```

接收用户输入



兰州科技大学

计算机科学与技术学院

3.3 输入和输出

□ 接收用户输入

```
#判断变量num是否是正数
num=int(input("请输入一个整数: "))
print ("您输入的整数是: %d" %(num))
if num > 0:
    print ("num是正数")
else:
    print ("num可能是0")
    print ("num也可能是负数")
```

请输入一个整数: 2
num是正数

请输入一个整数: 0
num可能是0
num也可能是负数

请输入一个整数: -2
num可能是0
num也可能是负数



■ 输出函数

常量、变量、表达式

print (输出内容)

- 输出字符串常量

```
>>>print("Hello, Sir")
Hello, Sir
```

- 输出数学表达式

```
>>>print(1+2)
3
```

- 输出变量

```
>>>x = 12
>>>print(x)
12
```



兰州交通大学

计算机科学与技术学院

□ 格式化参数的使用

```
print ("您输入的整数是: %d" %(num))
```

 ↑ ↑
 格式化参数 变量

```
#判断变量num是否是正数
num=int(input("请输入一个整数: "))
print ("您输入的整数是: %d" %(num))
if num > 0:
    print ("num是正数")
else:
    print ("num可能是0")
    print ("num也可能是负数")
```



运行结果

□ 输入正数: 2

```
print ("您输入的整数是: %d" %(num))
```

请输入一个整数: 2
您输入的整数是: 2
num是正数

□ 输入0

请输入一个整数: 0
您输入的整数是: 0
num可能是0
num也可能是负数

□ 输入负数: -2

请输入一个整数: -2
您输入的整数是: -2
num可能是0
num也可能是负数



兰州交通大学

计算机科学与技术学院

3.3 输入和输出

■ 常用的格式化参数

| 符 号 | 描 述 | 符 号 | 描 述 |
|-----|---------------|-----|--------------------|
| %c | 格式化字符及其ASCII码 | %x | 格式化无符号十六进制数 |
| %s | 格式化字符串 | %X | 格式化无符号十六进制数（大写） |
| %d | 格式化整数 | %f | 格式化浮点数字，可指定小数点后的精度 |
| %u | 格式化无符号整型 | %e | 用科学计数法格式化浮点数 |
| %o | 格式化无符号八进制数 | %p | 用十六进制数格式化变量的地址 |



3.3 输入和输出

字符串格式化参数

```
>>>print("His name is %s" %( "Mike"))
His name is Mike
```

使用多个格式化参数

```
print(...%s...%s...%s" %(string1,string2,...,stringn))
```

```
>>>yourname="Mike"
>>>myname="Devid"
>>>print("Hello,%s! I am %s." %(yourname,myname))
Hello,Mike! I am Devid.
```



兰州科技大学

计算机科学与技术学院

3.3 输入和输出

□ 转义字符

| 转义字符 | 描述 | 转义字符 | 描述 |
|------|----------------|----------|--------------|
| \r | 回车 | \(行尾) | 续行符 |
| \n | 换行 | \\"反斜杠符号 | |
| \t | 横向制表符 | ' | 单引号 |
| \v | 纵向制表符 | " | 双引号 |
| \f | 换页 | \000 | 空 |
| \a | 响铃 | \oyyy | 八进制数yyy代表的字符 |
| \b | 退格 (Backspace) | \xyy | 十进制数yy代表的字符 |



3.3 输入和输出

■ 输出换行符

```
>>>print("纸上得来终觉浅, \n绝知此事要躬行。")  
纸上得来终觉浅,  
绝知此事要躬行。
```

转义字符

■ 输出引号

```
>>>print("使用转义字符输出一个双引号: \"")  
输出一个双引号: "
```

转义字符

■ 输出 \

```
>>>print("使用转义字符输出一个反斜杠: \\")  
使用转义字符输出一个反斜杠: \
```



3.3 输入和输出

■ 失效转义字符

```
>>>print("C:\MyProgram\rencent\num\test\score")
```

```
>>>print("C:\MyProgram\rencent\num\test\score")
ecentrogram
um      est\score
```

```
>>>print(r'C:\MyProgram\rencent\num\test\score')
C:\MyProgram\rencent\num\test\score
>>>print(R'C:\MyProgram\rencent\num\test\score')
C:\MyProgram\rencent\num\test\score
```



3.3 输入和输出

□ end参数

print(输出内容, end)

- 表示输出信息结束之后，附加的字符串
- 默认值是换行\n。
- 设置这个参数，可以改变输出效果

不设置end参数，默认为\n，换行

```
>>>print( "Python")
>>>print( "3.5")
Python
3.5
```

设置end参数为空串，连续输出

```
>>>print( "Python", end="")
>>>print( "3.5")
Python3.5
```

```
>>>print( "Python", end=" ")
>>>print( "3.5")
Python 3.5
```

空格

设置end参数为空格

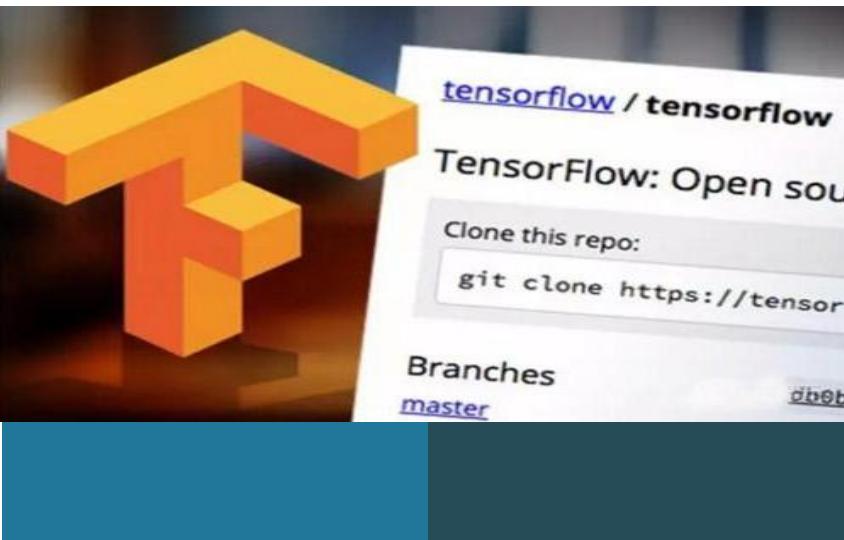


兰州交通大学

计算机科学与技术学院



3.4 常量、变量和表达式



3.4.1 数据类型、常量和变量

■ 基本数据类型

Python中支持**6种标准的数据类型**:

数字,字符串, 列表, 元组, 字典, 集合



□ 数字 (Numbers) —— 整型(Integer)

■ 正整数 `int_a=65536`

■ 负整数 `int_b=-200`

■ 零 `int_c=0`

Python3中的整数可以**任意大**，而不用担心位数不够而导致溢出的情况

```
>>>int_num=1234567899876543215578987651136
>>>print(int_num,type(int_num))
1234567899876543215578987651136 <class 'int'>
```



□ 数字 (Numbers) ——浮点数 (Float)

小数 3.14, -0.28

太阳直径: 1392000千米——>1.392×10⁹米——>1.392E9

浮点数

```
>>>float_sun=1.392E9
>>>print(float_sun,type(float_sun))
1392000000.0 <class 'float'>
```

浮点数

```
>>>print(1.392E9,";",1.392e9,";",0.1392E10)
1392000000.0; 1392000000.0; 1392000000.0;
```

整数

```
>>>print(type(1392000000))
<class 'int'>
```



□ 数字 (Numbers) —— 布尔值(Boolean)

True(真) —— 1 False(假) —— 0

true~~TRUE~~

false~~FALSE~~

□ 数字 (Numbers) —— 复数(Complex)

$a+bi$

$1+2i$

实部

虚部



兰州交通大学

计算机科学与技术学院

□ 字符串 (String)

- 使用成对的单引号或者双引号括起来。

```
"Python", "666", '3.5'
```

- 使用三重引号指定一个多行字符串。

```
>>> print('''  
这是一个文档字符串  
文档字符串可以跨越多行  
文档字符串可以使用三重单引号  
''')  
这是一个文档字符串  
文档字符串可以跨越多行  
文档字符串可以使用三重单引号
```

```
>>>print(""""  
这也是一个文档字符串  
使用三重双引号  
""")  
这也是一个文档字符串  
使用三重双引号
```



3.4.1 数据类型、常量和变量

多行注释：可以直接使用**多行字符串**，作为程序的注释

多行注释

"""

这个程序的功能是接收用户输入
并判断它是否大于0

"""

```
num=0
if num > 0:
    print ("num是正数")
else:
    print ("num可能是0")
    print ("num也可能是负数")
```



兰州交通大学

计算机科学与技术学院

■ 标识符 (Identifier) : 变量、函数、数组、文件、对象等的**名字**。

- 标识符的**第1个字符必须是字母或下划线**
- 其他字符可以由字母、下划线、或数字组成。
- 标识符长度任意。

test_data, test_data_1, _y, __, 1test, 1_test 

- 标识符是**大小写敏感的**。 score , Score
- 标识符不能与**Python关键字**重名。
- 在Python3中，标识符**支持非英语字符**，只要是Unicode字符集支持的字符都可以。建议尽量使用英文的标识符.

在给标识符命名时，应该尽量采用有意义的标识符。



Python关键字

```
>>>help("keywords")
Here is a list of the Python keywords. Enter any keyword
to get more help.
```

| | | | |
|----------|---------|----------|--------|
| False | def | if | raise |
| None | del | import | return |
| True | elif | in | try |
| and | else | is | while |
| as | except | lambda | with |
| assert | finally | nonlocal | yield |
| break | for | not | |
| class | from | or | |
| continue | global | pass | |



■ 常量 (constant)

- 数字、字符串、布尔值、空值等

2, -10086, 3.5, "Python", True, False, None

- Python中没有**命名常量**,不能给常量起一个名字



■ 变量 (variable)

- Python的变量**不需要声明**
- 其**数据类型**由**所赋的值**来决定
- 不同类型的**数字型**数据运算时，会自动的进行**类型转换**
bool<int<float<complex
- 自动的类型转换，仅存在于**数字型数据**之间

```
>>>print(1+True,1+False)  
2 1
```

```
>>>a=10  
>>>b=3.5  
>>>print(a+b,type(a+b))  
13.5<class 'float'>
```



■ 数据类型转换函数

| 函数 | 功 能 | 示 例 |
|--------------------------|-------------------|---------------------------|
| <code>int(x,base)</code> | 转换为整型 | <code>int("123")</code> |
| <code>float()</code> | 转换为浮点型 | <code>float("123")</code> |
| <code>bool()</code> | 转换为布尔型 | <code>bool(1)</code> |
| <code>str()</code> | 转换为字符串 | <code>str(123)</code> |
| <code>chr()</code> | 将整数转换为对应的ASCII字符 | |
| <code>ord()</code> | 将一个字符转换为对应的ASCII码 | |
| <code>complex()</code> | 转换为复数型 | <code>complex(1)</code> |



3.4.1 数据类型、常量和变量

```
#判断变量num是否是正数
num=int(input("请输入一个整数: "))
print ("您输入的整数是: %d" %(num))
if num > 0:
    print ("num是正数")
else:
    print ("num可能是0")
    print ("num也可能是负数")
```

向0取整

```
>>>print(int(3.6))
3
>>>print(int(-3.6))
-3
>>>print(int())
0
```

```
>>>a="11"
>>>b=int(a)+1
>>>print("a=",a ,type(a))
>>>print("b=",b ,type(b))
a= 11 <class 'str'>
b= 12 <class 'int'>
```

```
>>>print(chr(65))
A
>>>print(ord('A'))
65
```



□ 使用**help()**函数查看函数的用法

```
>>>help(int)
Help on class int in module builtins:
class int(object)
    int(x=0) -> integer
    int(x, base=10) -> integer
```

Convert a number **or** string to an integer, **or return 0 if no arguments are given.** If x **is** a number, **return** x.**_int_()**. For floating point numbers, this truncates towards zero.

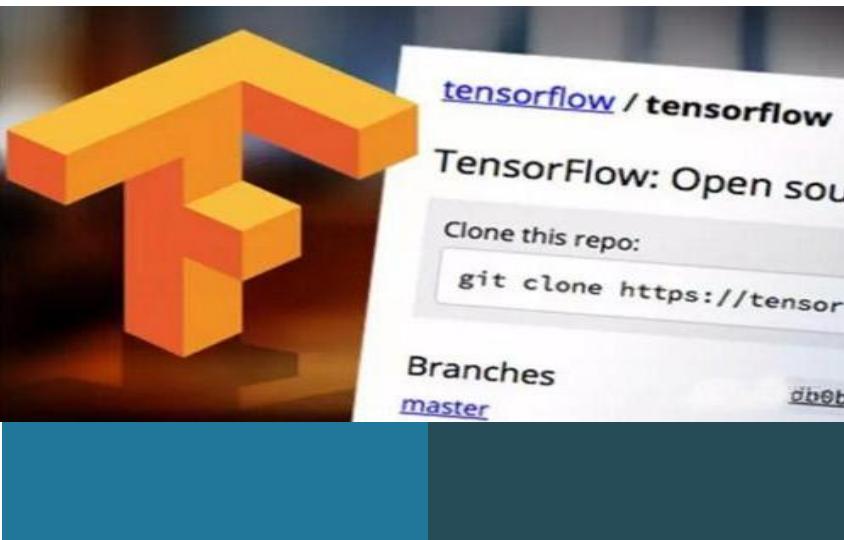
If x **is not** a number **or if** base **is** given, then x must be a string, bytes, **or** bytearray instance representing an integer literal **in** the given base. The literal can be preceded by '+' **or** '-' **and** be surrounded by whitespace. The base defaults to 10. Valid bases are 0 **and** 2-36.

Base 0 means to interpret the base **from** the string as an integer literal.

```
>>> int('0b100', base=0)
```

```
4
```





3.4.2 运算符和表达式

■ 运算符和表达式

- 运算符 (Operator) : 完成不同类型的常量、变量之间的运算
- 表达式 (Expression) : 由常量、变量和运算符组成



□ 算术运算符

| 算术运算符 | 描述 | 示例 |
|-------|------|---------------------------------------|
| + | 相加运算 | $1+2$ 的结果是3 |
| - | 相减运算 | $3-1$ 的结果是2 |
| * | 乘法运算 | $2*3$ 的结果是6 |
| / | 除法运算 | $7/2$的结果是3.5 (精确除法) |
| % | 求模运算 | $7\%2$的结果是1 (取余数) |
| ** | 幂运算 | $2^{**}3$ 的结果是8 |
| // | 整除运算 | $7//2$的结果是3 (向下取整) |



3.4.2 运算符和表达式

- 除法运算——`/` 结果是一个**浮点型的**精确数的值（即使是2个整数相除）

```
>>>print(7/2, 7.0/2,-7/2)  
3.5 3.5 -3.5
```

- 整除运算——`//` (**Floor**地板除法)

运算结果取**比商小的最大整数**

```
>>>print(7//2,-7//2)  
3 -4  
>>>print(7.0//2,-7//2.0)  
3.0 -4.0
```

如果操作数是浮点数，
那么结果仍然是**取整后的浮点数**

- 取余运算——`%`

```
>>>print(7%2)  
1  
>>>print(-7%2,7%-2)  
1 -1
```

余数的符号，是和**除数**一致的



□ 赋值运算符

| 赋值运算符 | 描述 | 示例 |
|-------|------|------------------------|
| = | 直接赋值 | $x=2$; 将2赋值到变量x中 |
| += | 加法赋值 | $x+=2$; 等同于 $x=x+2$ |
| -= | 减法赋值 | $x-=2$; 等同于 $x=x-2$ |
| *= | 乘法赋值 | $x*=2$; 等同于 $x=x*2$ |
| /= | 除法赋值 | $x/=2$; 等同于 $x=x/2$ |
| %= | 取模赋值 | $x\%=2$; 等同于 $x=x\%2$ |
| **= | 幂赋值 | $x**=2$; 等同于 $x=x**2$ |
| // | 整除赋值 | $x//=2$; 等同于 $x=x//2$ |



■ 连续赋值

```
>>>a=b=c=1
>>>print(a,b,c)
1 1 1
```

■ 多元赋值

```
>>>(x,y,z)=(1,2,'Python')
>>>print(x,y,z)
1 2 Python
```

```
>>>x=1
>>>y=2
>>>z='Python'
>>>print(x,y,z)
1 2 Python
```

例：交换x,y的值

```
>>>x,y=1,2
>>>x,y=y,x
>>>print(x,y)
2 1
```



□ 逻辑运算符

| 逻辑运算符 | 描述 |
|-------|-----|
| and | 逻辑与 |
| or | 逻辑或 |
| not | 逻辑非 |

True: 1 非0数值、非空集合
False: 0 值为0的数字、空集

操作数1 操作数2 and运算

| | | |
|-------|-------|-------|
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

操作数1 操作数2 or运算

| | | |
|-------|-------|-------|
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

操作数 not运算

| | |
|-------|-------|
| False | True |
| True | False |



□ 比较运算符

| 比较运算符 | 描述 |
|---------------------------|---------|
| <code>==</code> | 等于运算符 |
| <code>!=, <></code> | 不等于运算符 |
| <code><</code> | 小于运算符 |
| <code>></code> | 大于运算符 |
| <code><=</code> | 小于等于运算符 |
| <code>>=</code> | 大于等于运算符 |

```
>>>print(3<4)
True
>>>print(3==4)
False
>>>print(3<4<5)
True
```



3.4.2 运算符和表达式

□ **位运算符**: 直接对整数在内存中的**二进制位**进行操作

| 位运算符 | 描述 | 示例 |
|------|------|----------------------------------|
| & | 按位与 | $0\&0=0; 0\&1=0; 1\&0=0; 1\&1=1$ |
| | 按位或 | $0 0=0; 0 1=1; 1 0=1; 1 1=1$ |
| ^ | 按位异或 | $0^0=0; 0^1=1; 1^0=1; 1^1=0$ |
| ~ | 按位非 | $\sim 1=0; \sim 0=1$ |
| << | 位左移 | $1<<2 = 4$ |
| >> | 位右移 | $4>>1=2$ |



□ 字符串运算符

| 字符串运算符 | 描述 |
|--------|---------|
| + | 字符串连接 |
| * | 重复输出字符串 |

```
>>>py="Python"
>>>v2="2.0"
>>>v3="3.0"
>>>print(py+v2, py+v3)
Python2.0 Python3.0
```

```
>>>print("重要的话说三遍！"*3)
重要的话说三遍！重要的话说三遍！重要的话说三遍！
```



□ 字符串运算符

例：绘制分割线

```
>>>print("-"*40)
>>>print(" "*15+"这是分割线")
>>>print("-"*40)
```

这是分割线



兰州交通大学

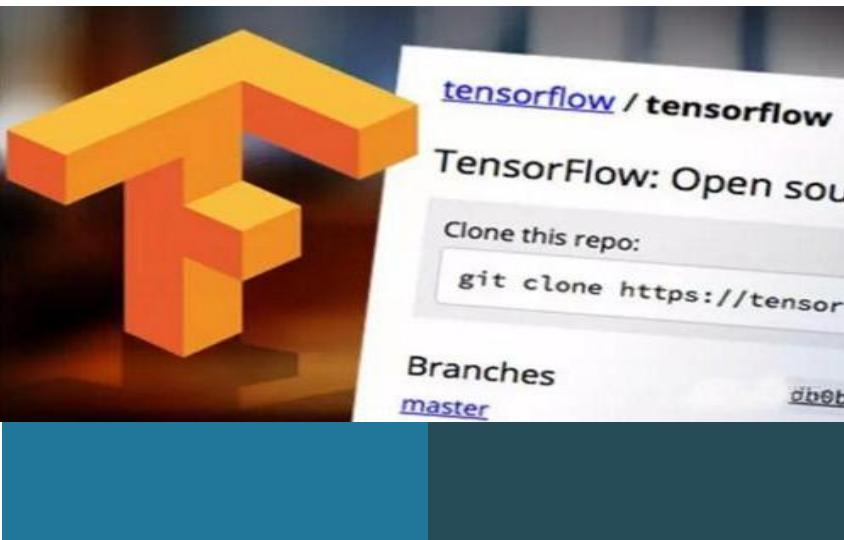
计算机科学与技术学院

□ 成员运算符

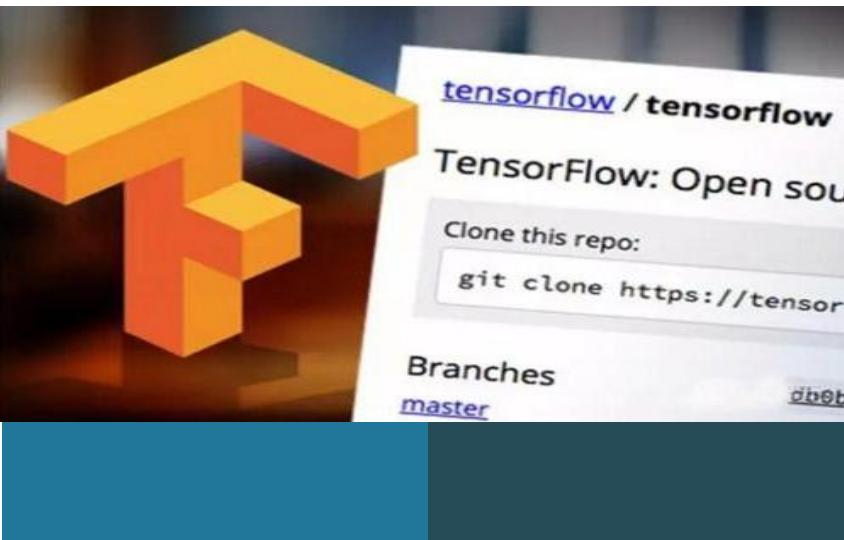
| 成员运算符 | 描述 |
|--------|--|
| in | 如果 序列 中包含给定的 元素 , 则返回True, 否则返回False。 |
| not in | 如果 序列 中不包含给定的 元素 , 则返回True, 否则返回False。 |

```
>>>str="Hello!"  
>>>'H' in str  
True  
>>>'h' in str  
False
```





3.5 程序控制语句

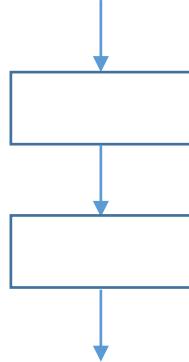


3.5.1 条件分支语句

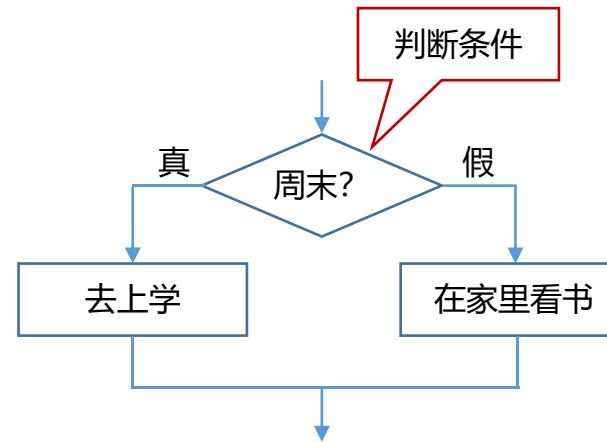
■ 程序控制语句：用来控制程序**执行的顺序**

□ **顺序结构**：各条语句一条一条**顺序执行**

□ **选择结构**：使程序根据**判断条件**而执行**不同的分支**



顺序结构

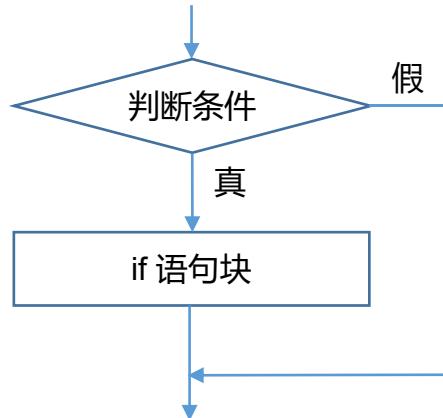


选择结构



if 语句

if 判断条件 :
 if语句块



```
x,y=3,5  
if x<y:  
    print("x is less than y.")  
    print("y is greater than x.")  
print("continue.....")
```

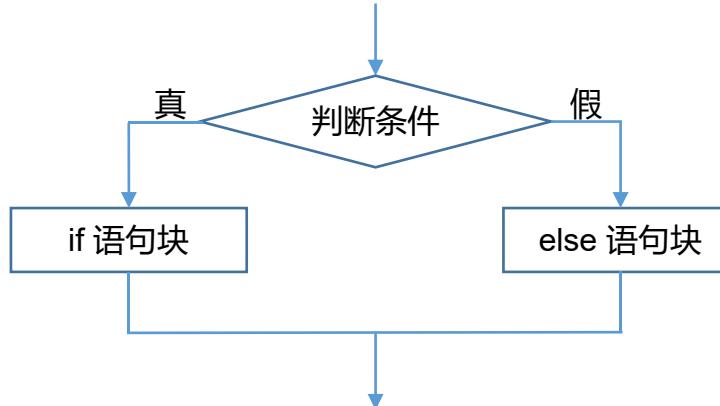
运行结果：

x is less than y.
y is greater than x.
continue.....



■ if-else 语句

```
if 判断条件 :  
    if语句块  
else:  
    else语句块
```



```
1 x=int(input("x:"))  
2 y=int(input("y:"))  
3 if x<y:  
4     print("x is less than y.")  
5 else:  
6     print("x is greater than y, or x is equal to y .")  
7     print("continue....")
```



3.5.1 条件分支语句

运行结果：

输入

x: 3

y: 5

输出

x is less than y.
continue....

x: 3

y: 3

x is greater than y, or x is equal to y .
continue....

x: 5

y: 3

x is greater than y, or x is equal to y .
continue....



兰州交通大学

计算机科学与技术学院

3.5.1 条件分支语句

■ if-elif-else语句

if 判断条件1:

语句块1

elif 判断条件2:

语句块2

elif 判断条件3:

语句块3

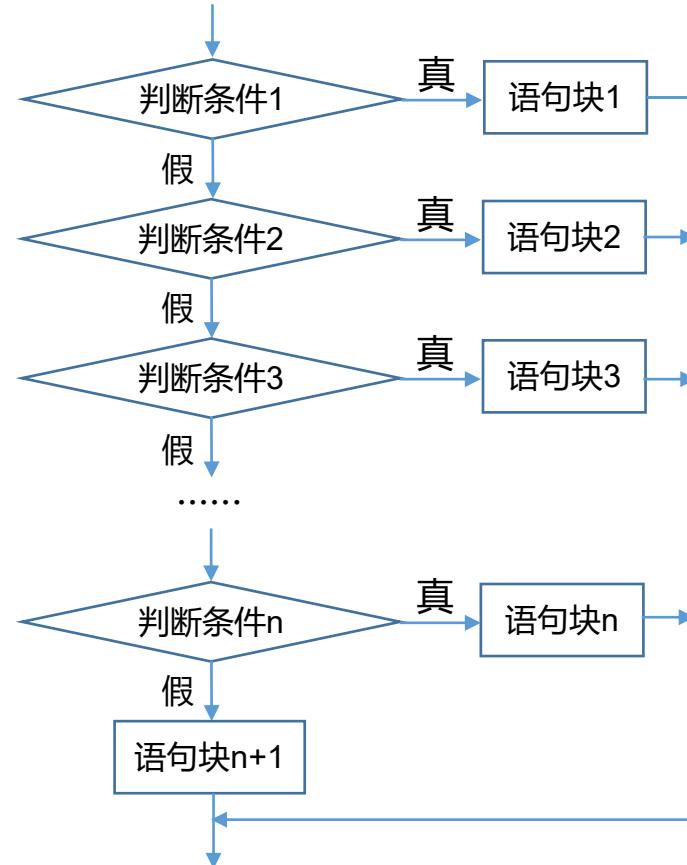
.....

elif 判断条件n:

语句块n

else:

语句块n+1



西安科技大学

计算机科学与技术学院

■ if-elif-else语句

```
1 x=int(input("x:"))
2 y=int(input("y:"))
3 if x<y:
4     print("x is less than y.")
5 elif x>y:
6     print("x is greater than y.")
7 else:
8     print("x is equal to y.")
9 print("continue....")
```



■ 条件表达式

表达式1 `if` 判断条件 `else` 表达式2

条件为真的结果

条件为假的结果

```
a,b=3,4  
print(a if a>b else b)
```

真

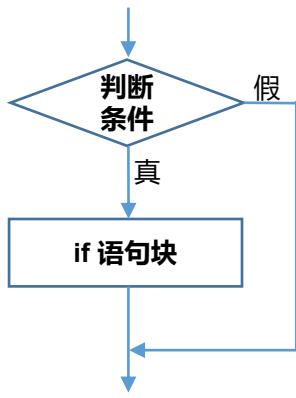
假



3.5.1 条件分支语句

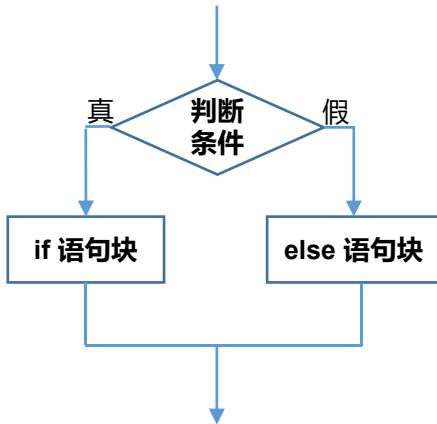
□ if 语句

if 判断条件：
if语句块



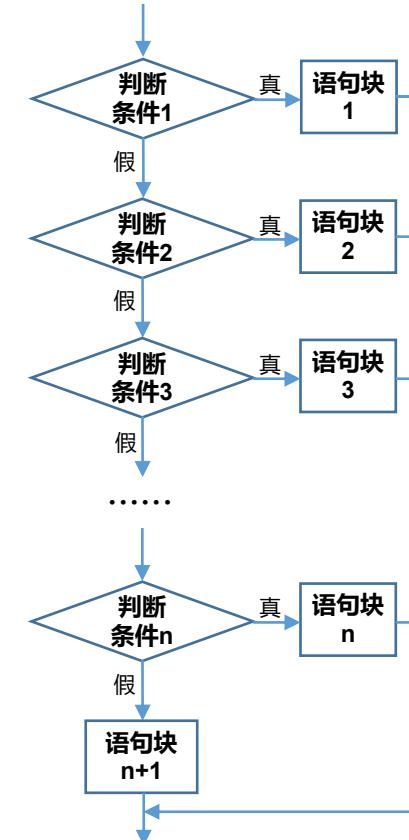
□ if-else语句

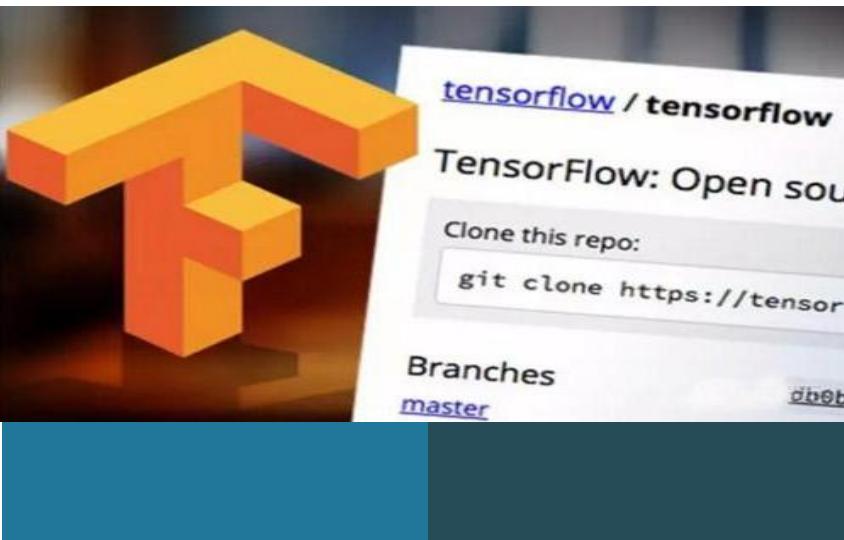
if 判断条件：
if语句块
else
else语句块



□ if-elif-else语句

if 判断条件1:
语句块1
elif 判断条件2:
语句块2
elif 判断条件3:
语句块3
.....
elif 判断条件n:
语句块n
else:
语句块n+1

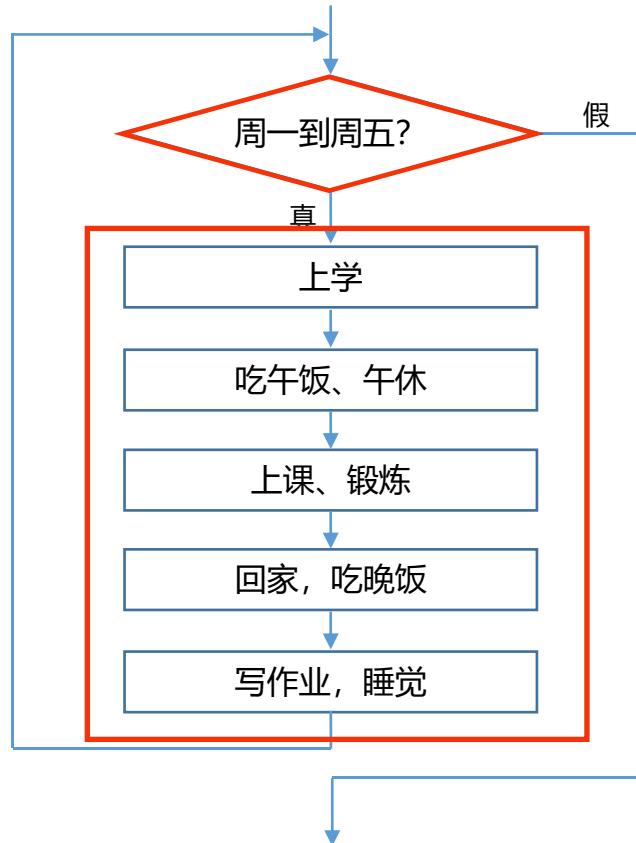




3.5.2 循环语句

■ 循环结构

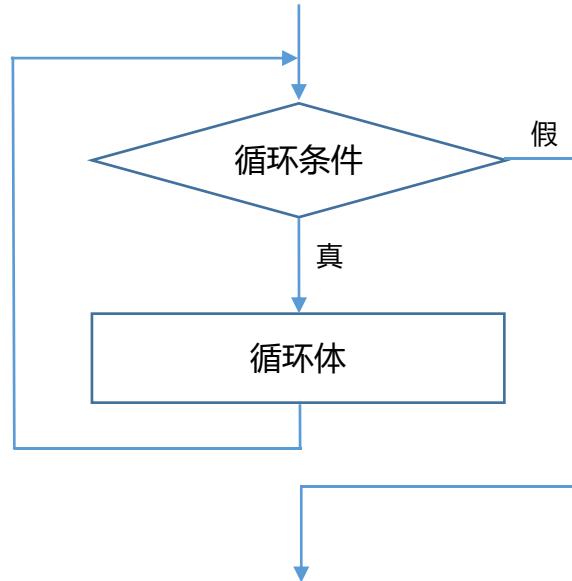
在**一定条件下**反复执行某段程序，
直到这个**条件不成立**为止。



■ while 语句

```
while 循环条件 :  
    循环体
```

死循环: 循环条件始终为真,
一直无法满足退出循环的条件。



3.5.2 循环语句

例：输出 $i=0,1,2$ 的值

```
1 i = 0
2 while i < 3:
3     print("i = %d" % i)
4     i += 1
5 print("continue....")
```

运行结果：

```
i = 0
i = 1
i = 2
continue.....
```



兰州交通大学

计算机科学与技术学院

例：实现1-100的累加和

```
1 #计算1--100的和
2 i=1
3 sum=0
4 while i<101:
5     sum += i
6     i+=1
7 print(sum)
```



#设置计数器初值
#设置累加和初值
#循环条件
#累加
#计数器+1
#输出累加和

运行结果：

5050



兰州交通大学

计算机科学与技术学院

■ for语句

for 标识符 **in** 可迭代对象:
 循环体

接受可迭代对象 (Iterable) 作为参数,
并对这个可迭代对象进行遍历。

```
>>>for c in "Python":  
        print(c)  
P  
y  
t  
h  
o  
n
```

```
>>>for c in "Python":  
        print(c, end="")  
Python
```

```
>>>for c in "Python":  
        print(c, end=" ")  
P y t h o n
```



3.5.2 循环语句

□ range()函数

range(起始数字, 结束数字, 步长)

- 前闭后开：整数序列中不包括**结束数字**
- **起始数字**省略时，默认从**0**开始
- **步长**省略时，默认为**1**

range(5) [0,1,2,3,4]

range(0,5) [0,1,2,3,4]

range(1,10,2) [1,3,5,7,9]

[1,3,5,7,9]

```
>>>for i in range(1,10,2):  
    print(i)
```

1
3
5
7
9

```
>>>for i in range(1,10,2):  
    print(i, end=";")  
1;3;5;7;9;
```



3.5.2 循环语句

例：实现1-100的累加和

```
1 #计算1到100的累加和
2 sum = 0 #设置初值
3 for i in range(101): #循环条件
4     sum+=i #累加
5 print("sum=",sum) #输出累加结果
```

运行结果：

```
sum= 5050
```



兰州交通大学

计算机科学与技术学院

■ **continue语句**: 终止**本次循环**, 开始下一次循环

例: 计算1-100之间的所有奇数的和。

```
#计算1到100的奇数和
sum = 0
for i in range(1,101):
    if i %2 == 0:
        continue
    sum += i
print("sum=",sum)
```

#设置初值
#循环条件
#如果是偶数, 则跳过本次循环, 不进行累加
#累加
#输出累加结果

运行结果:

```
sum= 2500
```



兰州交通大学

计算机科学与技术学院

3.5.2 循环语句

例：计算1-100之间的所有奇数的和。

```
#计算1到100的奇数和
sum = 0
for i in range(1,101,2):
    sum += i
print("sum=",sum)
```

#设置初值
#循环条件
#累加
#输出累加结果

运行结果：

```
sum= 2500
```



兰州交通大学

计算机科学与技术学院

■ break语句：跳出循环体，结束循环

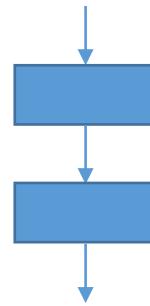
例：计算1-100的累加和。如果在计算过程中，累计和**超过3000**，则**终止计算**，并输出当前结果。

```
i=1          #设置初值
sum=0
while i<101:
    if sum<=3000:      #循环条件
        sum += i       #附加条件
        i+=1
    else:              #累积和>3000,跳出循环，终止计算
        break
print(sum)
```

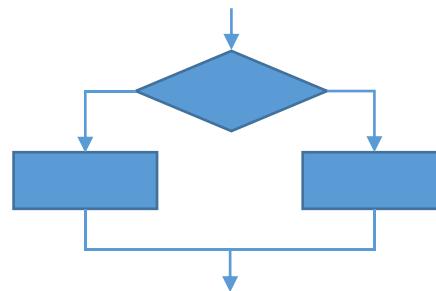


■ 程序流程控制

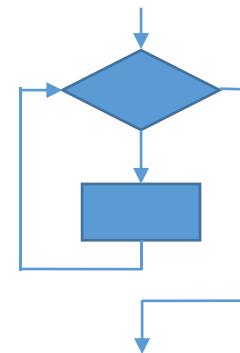
- 顺序结构：从上到下依次顺序执行。
- 选择结构：根据**判断条件**，来**选择**执行不同的任务。
- 循环结构：根据**循环条件**，来**重复**执行某段代码。



顺序结构



选择结构
条件分支语句

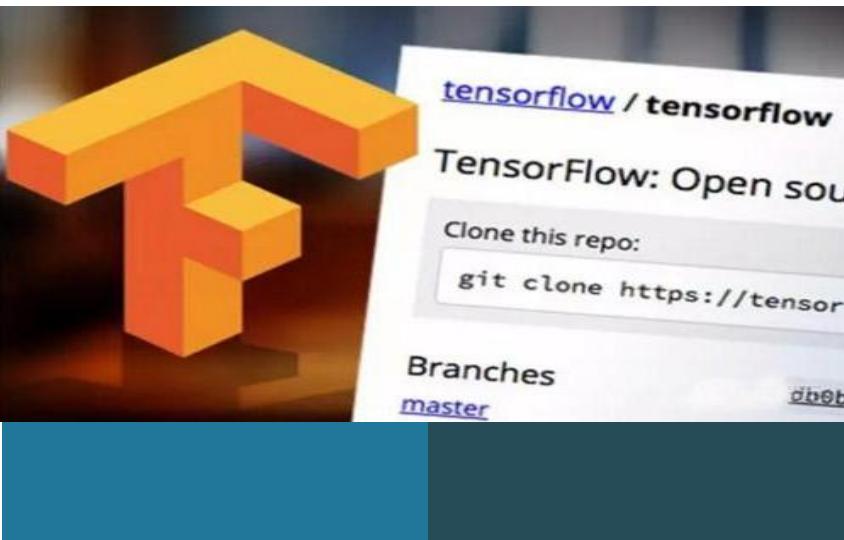


循环结构
for语句, while语句

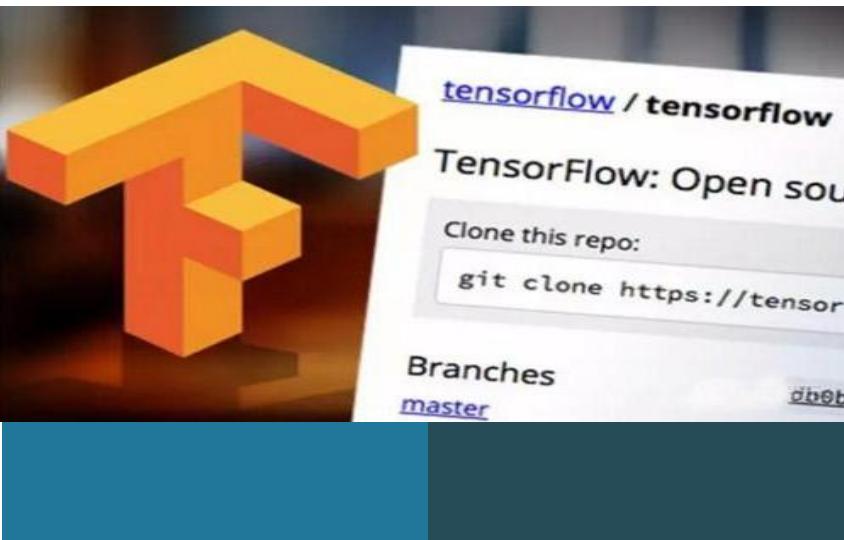


04 Python语言基础(2)

西安科技大学 牟琦
muqi@xust.edu.cn



4.1 内置数据结构



4.1.1 序列数据结构

■ 序列数据结构 (sequence)

- 成员是**有序**排列的
- 每个元素的位置称为**下标**或**索引**
- 通过索引访问序列中的成员
- Python中的序列数据类型有**字符串、列表、元组**

"abc" \neq "bca"

- Python中的列表和元组，可以存放**不同类型**的数据
- **列表**使用方括号[]表示，**元组**使用小括号()表示。

列表：[1,2,3]

元组：(1,2,3)



复旦科技大学

计算机科学与技术学院

4.1.1 序列数据结构

口 列表 (list)

```
lst_1=[1,2,3]
lst_2=[4]
lst_3=[[1,2,3],[4,5,6]]

lst_mix=[160612,"张明",18,[92,76,85]]
lst_empty=[ ]
```

口 元组 (tuple) : 一经定义，元组的内容不能改变。

```
>>>tup_1=(1,2,3)
>>>tup_empty=()
>>>tup_empty
()

>>>t1=(1)
>>>t2=(1,)
>>> print("t1=",t1,type(t1))
t1= 1 <class 'int'>
>>> print("t2=",t2,type(t2))
t2= (1,) <class 'tuple'>
```



4.1.1 序列数据结构

■ 索引（下标）：通过它访问序列中的元素

-6 -5 -4 -3 -2 -1 逆向索引
Python
正向索引: 0 1 2 3 4 5

-3 -2 -1 逆向索引
[1, 2, 3]
正向索引: 0 1 2

```
# 字符串索引
>>>str_py = "Python"
>>>print(str_py[0])
P
>>>print(str_py[-1])
n
```

```
#列表索引
>>> lst_1=[1, 2, 3]
>>> print(lst_1[1])
2
>>> print(lst_1[-2])
2
```



4.1.1 序列数据结构

切片：一次性从序列中获取多个元素

[开始位置:结束位置]

前闭后开: 切片不包括结束位置的元素

Python

正向索引: 0 1 2 3 4 5

```
# 字符串切片
>>>str_py="Python"
>>>print(str_py[1:5])
'ytho'
print(str_py[1:])
'ython'
print(str_py[:5])
'Pytho'
```

[1,2,3]

正向索引: 0 1 2

```
#列表切片
>>> list1=[1,2,3]
>>> list1[2:]
[3]
```

开始位置缺省，从第一个元素开始
结束位置缺省，到最后一个元素为止
开始位置和结束位置都缺省，取到整个序列



西安科技大学

计算机科学与技术学院

4.1.1 序列数据结构

例：混合列表切片

正向索引: 0 1 2 3

[160612, "张明", 18, [92, 76, 85]]

```
#列表切片
>>> lst_stud=[160612,"张明",18,[92,76,85]]

>>> list_stud[1:]
["张明",18,[92,76,85]]

>>> list_stud[:3]
[160612,"张明",18]
```



■ 打印——print()

```
>>>lst_1=[1,2,3]
>>>print(lst_1)          #打印整个列表
[1,2,3]
```

```
>>>tup_1=(1,2,3)
>>>print(tup_1)          #打印整个元组
(1,2,3)
```

```
>>>print(lst_1[0])       #打印列表中的元素
1
```

```
>>>print(tup_1[0:2])     #打印元组中的元素
(1,2)
```



■ 获取序列的长度——len(序列名称)

```
#获取字符串的长度
>>>len("Python")
6
>>>str="Python"
>>>len(str)
6
```

```
#获取列表的长度
>>>lst_1=[1,2,3]
>>>len(lst_1)
3
```

```
#获取二维列表的长度
lst_3=[[1,2,3],[4,5,6]]
>>>len(lst_3)
2
```

```
#获取混合列表的长度
>>>lst_mix=[160612,"张明",18,[92,76,85]]
>>>len(lst_mix)
4
```

```
# 获取元组的长度
>>>tup_1=(1,2,3)
>>>len(tup_1)
3
```



■ 更新列表 ——向列表中添加元素

□ append()

```
#向列表中追加元素
>>>lst_1=[1,2,3]
>>>lst_1.append(4)
>>>lst_1
[1,2,3,4]
```

□ insert()

```
#向列表中追加元素
>>>lst_1=[1,2,3]
>>>lst_1.insert(1,5)
>>>lst_1
[1,5,2,3]
```

由于元组一经定义后就不能更改了，因此**元组不支持更新操作**。



■ 更新列表——合并列表

□ extend()

```
#合并列表
>>>lst_1=[1,2,3]
>>>lst_2=[4]
>>>lst_1.extend(lst_2)
>>>lst_1
[1,2,3,4]
```

□ "+" 运算符

```
>>>lst_1=[1,2,3]
>>>lst_2=[4]
>>>lst_3=lst_1+lst_2
>>>lst_3
[1,2,3,4]
```



■ 更新列表——删除列表中的元素

□ del 语句

```
>>>lst_1=[1,2,3,4]
>>>del lst_1[1]          #删除下标为1的元素
>>>lst_1
[1,3,4]
```



■ 更新列表——排序

- `sort()`：对列表中的元素排序
- `reverse()`：对列表中的元素倒排序

```
>>>lst_1=[2,3,1,4]
>>>lst_1.sort()          #将lst_1中的元素按从小到大的顺序排列
>>>lst_1
[1,2,3,4]

>>>lst_1.reverse()      #将lst_1中的元素原地逆序
>>>lst_1
[4,3,2,1];
```



■ 遍历列表中的元素

```
#遍历列表中的元素
>>>lst_1=[1,2,3,4]
>>>for i in lst_1:
    print(i,end=" ")
1 2 3 4
```





4.1.2 字典和集合

■ 字典 (Dictionary)

- 每个字典元素都是一个**键/值对** (key/value)
 - **键**: 关键字
 - **值**: 关键字对应的取值
-
- 创建字典

```
>>>dic_score={"语文":80, "数学":85,"英语":78,"体育":90}  
  
>>>dic_employ={"name":"Mary", "age":26}  
  
>>>dic_employ={"name":{"first":"Mary", "last":"Smith"}, "age":26}
```



4.1.2 字典和集合

□ 打印字典、访问字典中的元素

```
>>>dic_score={"语文":80, "数学":85,"英语":78,"体育":90}  
  
>>>print(dic_score)  
{"语文":80, "数学":85,"英语":78,"体育":90}  
  
>>>print(dic_score["语文"])  
80  
  
>>>len(dic_score)  
4
```

□ 判断字典是否存在元素——in运算符

```
>>>dic_student = {'name':'张明','sex':'男','age':18,'score':98}  
>>> 'sex' in dic_student  
True
```



□ 遍历字典元素

- keys(): 返回字典中所有的关键字
- values(): 返回字典中所有的值
- items(): 返回字典中所有的键值对

例：遍历字典中所有的键

```
>>>dic_student = {'name':'张明','sex':'男','age':18,'score':98}

>>>for key in dic_student.keys():
    print(key,end=" ")
age name score sex
```

字典中的元素是无序的，因此每次显示的顺序可能不同



例：遍历字典中所有的值

```
>>>dic_student = {'name':'张明','sex':'男','age':18,'score':98}

>>>for value in dic_student.values():
    print(value, end=" ")
18 张明 98 男

>>>for item in dic_student.items():
    print(item)
('age', 18)
('name', '张明')
('score', 98)
('sex', '男')
```



4.1.2 字典和集合

□ 更新字典：添加元素、修改指定元素的取值

```
>>>dic_student={'name':'张明','sex':'男','age':18}

>>>dic_student['score'] = 98
>>>print(dic_student)
{'age': 18, 'name': '张明', 'score': 98, 'sex': '男'}
```



```
>>>dic_student['score']=90
>>>print(dic_student)
{'age': 18, 'name': '张明', 'score': 90, 'sex': '男'}
```



4.1.2 字典和集合

□ 合并字典：将另一个字典中的元素追加到字典中

```
>>>dic_student={'name':'张明','sex':'男','age':18}  
>>>dic_contact={'tel':13104415887,'email':'zhm@gmail.com'}  
  
>>>dic_student.update(dic_contact)  
  
>>>print(dic_student)  
{'email': 'zhm@gmail.com', 'age': 18, 'name': '张明', 'tel': 13104415887, 'sex': '男'}
```



4.1.2 字典和集合

□ 删除字典中的元素

pop (指定元素的关键字)

删除字典中的指定元素

clear()

清空字典中的所有元素

```
>>>dic_student={'name':'张明','sex':'男','age':18}

>>>dic_student.pop('sex')
>>>print(dic_student)
{'age': 18, 'name': '张明'}
```



```
>>>dic_student.clear()
>>>print(dic_student)
{}
```

另外，Python中还提供了了**del语句**，使用它可以删除字典中**指定的元素**，或**删除字典本身**。



■ 集合 (set) : 由一组**无序**排列的元素组成。

- 可变集合 (set) : 创建后可以添加、修改和删除其中的元素
- 不可变集合 (frozenset) : 创建后就不能再改变了

{1,2,3,4,5,4,5}

□ 创建集合

Python会自动的将集合中重复的元素清理掉

```
>>>set1={1,2,3,4,5,4,5}  
  
>>>print(set1)  
{1,2,3,4,5}  
  
>>>len(set1)  
5
```



西安科技大学

计算机科学与技术学院

□ 创建集合

set()

创建可变集合

frozenset()

创建不可变集合

```
>>>set_2=set("Python")
>>>print(set_2)
{'n', 'P', 'o', 't', 'h', 'y'}
```



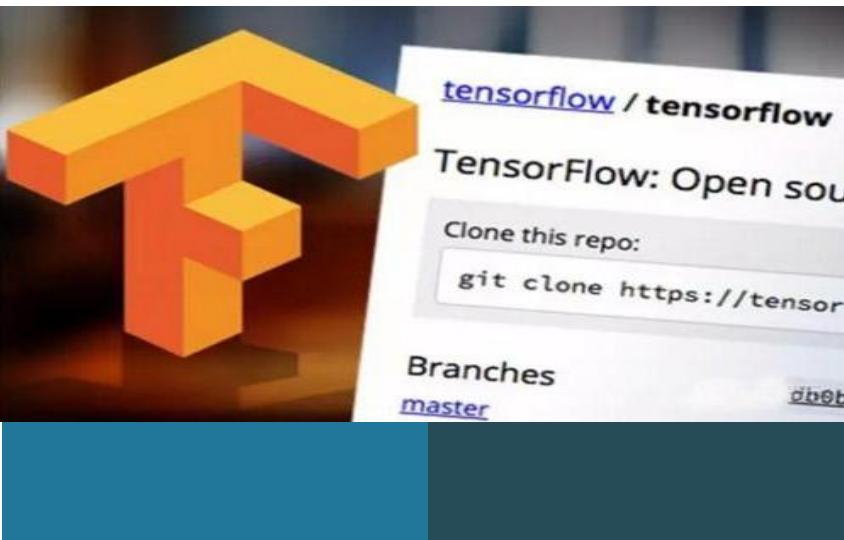
```
>>>set_3 = frozenset("Python")
>>>print(set_3)
frozenset({'n', 'P', 'o', 't', 'h', 'y'})
```

- 集合中的元素是无序的，因此**不能通过下标来访问**
- **打印集合、获取集合长度、遍历集合**的方法，与其他内置数据类型类似

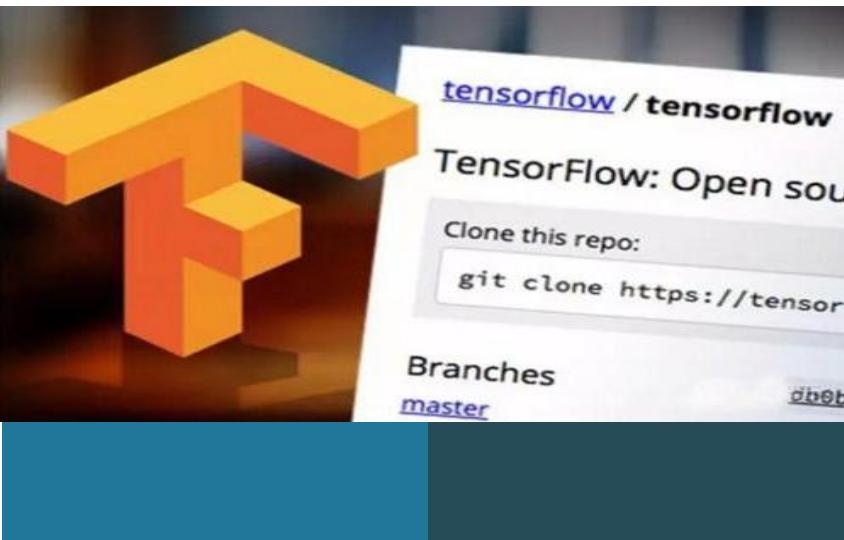


西安科技大学

计算机科学与技术学院



4.2 函数和模块



4.2.1 函数

■ 函数 (function)

- 实现某种**特定功能**的代码块
- 程序简洁，可重复调用、封装性好、便于共享
- 系统**函数**和**用户自定义函数**

Python3.6.2版本，一共提供了**68个内置函数**。



■ Python内置函数

- 数学运算函数
- 输入输出函数
- 类型转换函数
- 逻辑判断函数
- 序列操作函数
- 对象操作函数



□ 数学运算函数

| 函数 | 原型 | 具体说明 |
|----------|--------------|---|
| abs() | abs(x) | 返回x的 绝对值 |
| pow() | pow(x,y) | 返回x的y次 幂 |
| round() | round(x[,n]) | 返回浮点数x的 四舍五入 值，参数n指定保留的小数位数，默认为0 |
| divmod() | divmod(a,b) | 返回a除以b的 商和余数 ，返回一个元组。divmod(a,b)返回(a/b, a%b) |

```
>>>abs(-1)
1
>>>pow(2,3)
8
>>>round(3.1415,2)
3.14
>>>round(3.54)
4
>>>divmod(5,3)
(1,2)
```



口 常用Python内置函数

| 函数 | 描述 | 函数 | 描述 |
|---------|---------|---------|----------|
| len() | 返回长度 | list() | 转换为列表 |
| max() | 返回最大值 | help() | 显示帮助信息 |
| sum() | 返回总和 | dir() | 显示属性 |
| str() | 转换成字符串 | type() | 显示类型 |
| float() | 转换为浮点型 | range() | 返回一个整型列表 |
| int() | 转换为整型表示 | open() | 打开文件 |



■ 用户自定义函数

□ 定义函数

形式参数 / 形参

```
def 函数名 (参数列表):  
    函数体
```

- **形式参数**在定义时并没有确定的值，只是用来说 明函数的功能。在程序执行过程中，当函数被调 用时，形参才得到具体的值，并参与运算，求得 函数值
- **参数列表**可以为空，即函数可以没有参数；
- 如果函数中包含**多个参数**，参数之间用**逗号**分隔。
- **函数体**可以是一条语句，也可以是一个语句块。



4.2.1 函数

例：自定义函数

- 一个返回值

```
def add(a,b):  
    c=a+b  
    return c
```

- 多个返回值

```
def add_mul(a,b):  
    add=a+b  
    mul=a*b  
    return add,mul
```

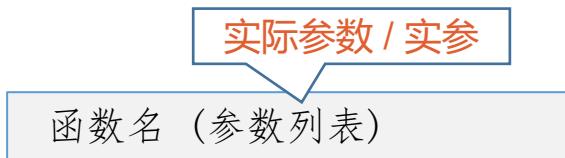
- 无返回值

```
def say_hello(your_name):  
    print("Hello,%s!" %your_name)
```



4.2.1 函数

□ 调用函数



定义函数：

```
def add(a,b):  
    c=a+b  
    return c
```

调用函数： add(1,2)

运行结果： 3



4.2.1 函数

□ 通过**多元赋值语句**，同时获取多个返回值

```
def add_mul(a,b):          #定义函数
    add=a+b
    mul=a*b
    return add,mul

x,y=add_mul(1,2)           #调用函数
print("add:",x,";mul:",y)
```

运行结果：

```
add: 3 ;mul: 2
```



兰州交通大学

计算机科学与技术学院

4.2.1 函数

□ 无形式参数

```
def say_hello():      # 定义函数
    print("Hello! ")  
  
say_hello()          # 调用函数
```

调用函数时，函数名后面必须有小括号，即使没有参数，这个小括号也不能省略。

运行结果：

```
Hello!
```



西安科技大学

计算机科学与技术学院

□ 变量的作用域

- **局部变量** (Local Variable) : 在函数中定义的变量, 仅在定义它的函数内部有效。
- **全局变量** (Global Variable) : 在函数体之外定义的变量, 在定义后的代码中都有效, 包括在它之后定义的函数体内。



4.2.1 函数

例：局部变量

```
def setNumber():
    a= 9
    a=a+1
    print("setNumber:", a)

setNumber()
```

运行结果：

```
setNumber: 10
```

定义局部变量的函数中，**只有局部变量是有效的**

```
>>>print(a)

-----
NameError                                 Traceback (most recent call last)
<ipython-input-9-5315f3e3adca> in <module>()
      1 print(a)

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



西安科技大学

计算机科学与技术学院

4.2.1 函数

例：全局变量&局部变量

```
a=100 #定义全局变量
def setNumber():
    a= 9 #定义函数
    a=a+1 #定义局部变量
    print("setNumber:", a) #打印局部变量

setNumber() #调用函数，打印局部变量
print(a) #打印全局变量
```

运行结果：

```
setNumber: 10
100
```

- 在函数外部定义的变量a是**全局变量**，当它与函数内部定义的局部变量同名时，在函数内部失效。
- 在函数内定义的变量a是**局部变量**，它只在函数体内局部有效，并不影响全局变量a的取值。



兰州交通大学

计算机科学与技术学院

□ 函数的参数——参数的传递

- 按值传递：形参和实参分别存储，相互独立。
- 在内部函数改变形参的值时，实参的值不会随之改变。

```
def func(num):          # 定义函数
    num+=1              # 函数内部改变形参的值
    print("num:", num)  # 打印形参

a=10
func(a)                # 调用函数， a为实参
print("a:", a)          # 输出变量a
```

运行结果：

```
num:11
a:10
```



兰州科技大学

计算机科学与技术学院

□ 函数的参数——参数的默认值

```
def 函数名 (参数1=默认值, 参数2=默认值...):  
    函数体
```

```
def add(a,b=2):  
    return a+b  
  
print(add(1))  
print(add(2,3))
```

运行结果：

```
3  
5
```

```
def add (a, b=1, c=2):  
    return a+b+c
```

当函数中有多个参数时，参数的默认值只能从最右面开始依次设置



兰州交通大学

计算机科学与技术学院

4.2.1 函数

□ 函数的参数——向函数内部批量传递数据

- 可以使用**列表、字典**变量作为参数，向函数内部批量传递数据。
- 当使用列表或字典作为函数参数时，在函数内部对列表或字典的元素所做的修改，**会改变实参的值**。

```
def sum(list):
    sum=0
    for x in list:
        sum+=x
    return sum

lst_1=[1,2,3,4,5]
print(sum(lst_1))
```

运行结果： 15



兰州交通大学

计算机科学与技术学院



4.2.2 模块、包和库

■ 模块 (Module)

- 模块是一个python文件 (.py)，拥有多个功能相近的函数或类。
- 便于代码复用，提高编程效率，提高了代码的可维护性。
- 避免函数名和变量名冲突。

■ 包 (Package)

- 为了避免模块名冲突，Python引入按目录来组织模块的方法。
- 一个包对应一个文件夹，将功能相近的模块（Python文件），放在同一个文件夹下。
- 在作为包的文件夹下有一个 `__init__.py` 文件。
- 子包：子目录中也有 `__init__.py` 文件。

■ 库 (Library)：具有相关功能的模块或包的集合。



复旦大学

计算机科学与技术学院

4.2.2 模块、包和库

例：包的结构

```
package_a
└── __init__.py
└── module_a1.py
└── module_a2.py
└── subpack_ab
    └── __init__.py
        ├── module_ab_1.py
        └── module_ab_2.py
```

这个包对应文件夹package_a，其中有一个__init__文件，和两个模块，还有一个子包；子包中也有2个模块。



4.2.2 模块、包和库

□ 导入模块/包/库

■ 导入整个包

PEP8 Python
编码规范推荐

import 名称 as 别名

```
import numpy as np  
np.random.random()
```

■ 导入包中指定的模块或子包

from 模块名 import 函数名 as 函数别名

```
from numpy import *  
random.random()
```



4.2.2 模块、包和库

□ 导入语句的作用域：

- 在程序顶部导入模块，作用域是全局的。
- 在函数的内部导入语句，作用域就是局部的。

建议导入的顺序

python 标准库/模块

python 第三方库/模块

自定义模块



兰州交通大学

计算机科学与技术学院

4.2.2 模块、包和库

□ 使用模块/包/库中的函数和变量

模块/包/库名. 函数名(参数列表)

模块/包/库名. 变量名

```
>>>import math          # 导入math模块
>>>math.pow(2,3)      # 计算2的3次幂
8.0

>>>from math import pow, sqrt   # 从math模块中只导入pow和sqrt函数
>>> pow(2,3)           # 计算2的3次幂
8.0
>>>sqrt(16)            # 计算16的开平方
4.0

>>>from math import sqrt as s  # 从math模块中导入sqrt函数并重命名为s
>>>s(16)
4.0
```



4.2.2 模块

□ 自定义模块

- 创建自定义模块：将常用函数的定义放在一个**.py文件**中。

mymodule.py

```
def 函数1():      # 定义函数1
    函数体1

def 函数2():      # 定义函数2
    函数体2
```

- 当函数较多时，可以**按照功能**将它们放在**不同的模块**中。
- 一个应用程序中可以定义多个模块。



兰州交通大学

计算机科学与技术学院

4.2.2 模块

□ 创建自定义模块

```
mymodule.py
def print_str(str):          # 打印字符串
    print(str)

def sum(a,b):                # 求和
    return a+b
```

□ 调用自定义模块

```
>>>import mymodule as mm      # 导入mymodule模块
>>>mm.print_str("Python")     #调用打印函数
Python
>>>mm.sum(2,3)               #调用求和函数
5
```



4.2.2 模块

■ Python标准库中的模块

- sys模块：提供有关Python运行环境的变量和函数。

sys模块中的常用变量

| 变量 | 功能 |
|--------------|---------------------|
| sys.platform | 获取当前操作系统 |
| sys.path | 获取指定模块搜索路径 |
| sys.argv | 获取当前正在执行的命令行参数的参数列表 |



兰州交通大学

计算机科学与技术学院

4.2.2 模块

- `sys.platform`: 获取当前操作系统。

```
>>>import sys  
>>>sys.platform  
linux2
```

```
>>>import sys  
>>>sys.platform  
win32
```

- `sys.path`: 获取指定模块搜索路径。

```
>>> import sys  
>>> sys.path  
['',  
'F:\\python3.6\\\\Lib\\\\idlelib',  
'F:\\python3.6\\\\python36.zip',  
'F:\\python3.6\\\\DLLs',  
'F:\\python3.6\\\\lib',  
'F:\\python3.6',  
'F:\\python3.6\\\\lib\\\\site-packages']
```



4.2.2 模块

- `sys.append()`: 添加指定模块搜索路径。

```
sys.path.append('路径')
```

```
>>>sys.path.append('F:\\myProgramme')
>>>sys.path
[',
 'F:\\python3.6\\Lib\\idlelib',
 'F:\\python3.6\\python36.zip',
 'F:\\python3.6\\DLLs',
 'F:\\python3.6\\lib',
 'F:\\python3.6',
 'F:\\python3.6\\lib\\site-packages',
 'F:\\myProgramme']
```

在退出Python环境后，使用`sys.path.append()`函数添加的路径会自动消失。



兰州交通大学

计算机科学与技术学院

4.2.2 模块

sys模块中的常用函数

| 函数 | 功能 |
|--|--|
| <code>sys.exit(n)</code> | 退出应用程序。 <code>n=0</code> , 正常退出; <code>n=1</code> , 异常退出。 |
| <code>sys.getdefaultencoding()</code> | 获取系统当前编码。 |
| <code>sys.setdefaultencoding()</code> | 设置系统默认编码 |
| <code>sys.getfilesystemencoding()</code> | 获取文件系统使用编码方式, Windows下返回'mbcs', mac下返回'utf-8'。 |



4.2.2 模块

□ platform模块：获取操作系统的详细信息和与Python有关的信息。

| 函数 | 功能 |
|-------------------------|--|
| platform.platform() | 获取操作系统名称及版本号信息 |
| platform.system() | 获取操作系统类型 |
| platform.version() | 获取操作系统的版本信息 |
| platform.processor() | 获取计算机的处理器信息 |
| platform.python_build() | 获取Python的版本信息，包括Python的主版本、编译版本号和编译时间等信息 |



4.2.2 模块

□ math模块：提供了常用的数学运算。

| 变量和函数 | 功能 |
|---------------|--------------------------------|
| math.e | 返回自然对数e的值 |
| math.pi | 返回π的值 |
| math.exp(x) | 返回e的x次幂 |
| math.fabs(x) | 返回x的绝对值 |
| math.ceil(x) | 返回大于等于x的最小整数 |
| math.floor(x) | 返回小于等于x的最大整数 |
| math.log(x,a) | 返回 $\log_a x$,如果不指定参数a，则默认使用e |
| math.log10(x) | 返回 $\log_{10} x$ |
| math.pow(x,y) | 返回x的y次幂 |
| math.sqrt(x) | 返回x的开平方 |

4.2.2 模块

□ random模块：生成随机数。

| 函 数 | 功 能 |
|----------------------|--|
| random.random() | 生成一个0到1的随机浮点数 |
| random.uniform(a, b) | 生成一个指定范围内的随机浮点数。其中a是下限， b是上限 |
| random.randint(a, b) | 生成一个指定范围内的随机整数。a是下限， b是上限 |
| random.choice(seq) | 从序列中随机获取一个元素。参数seq表示一个有序类型，可以是一个列表、元组或者字符串 |
| random.shuffle(x) | 将一个列表x中的元素打乱 |



4.2.2 模块

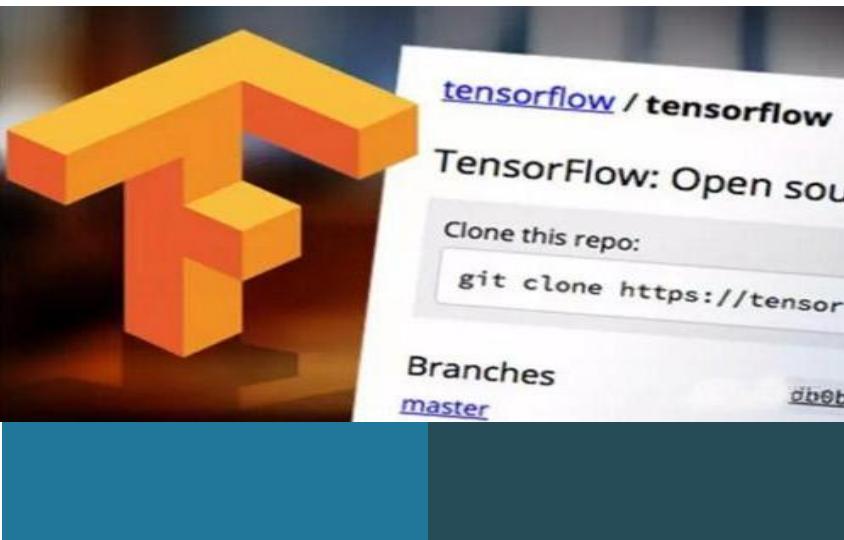
□ 小数和分数处理模块

- decimal模块：表示和处理小数。
- fractions模块：表示和处理分数。

□ 时间处理模块

- time
- datetime
- calendar

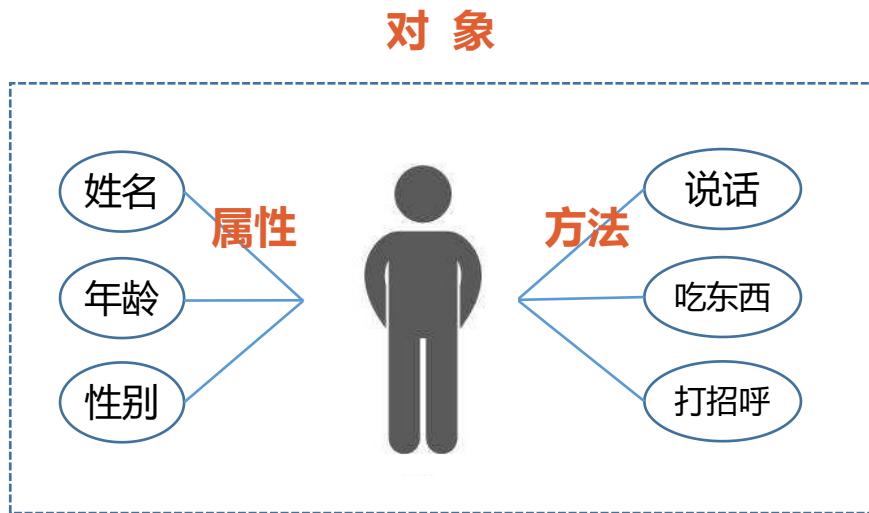




4.3 Python面向对象的编程

4.3 Python面向对象的编程

- 面向对象的程序设计 (Object-oriented programming, OOP)
- 面向过程的程序设计 (Process-oriented programming, POP)

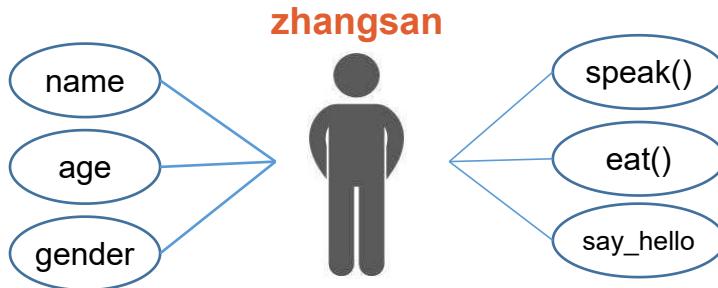


对象 (object) : 将属性和方法封装在一起。



4.3 Python面向对象的编程

■ 对象的属性和方法

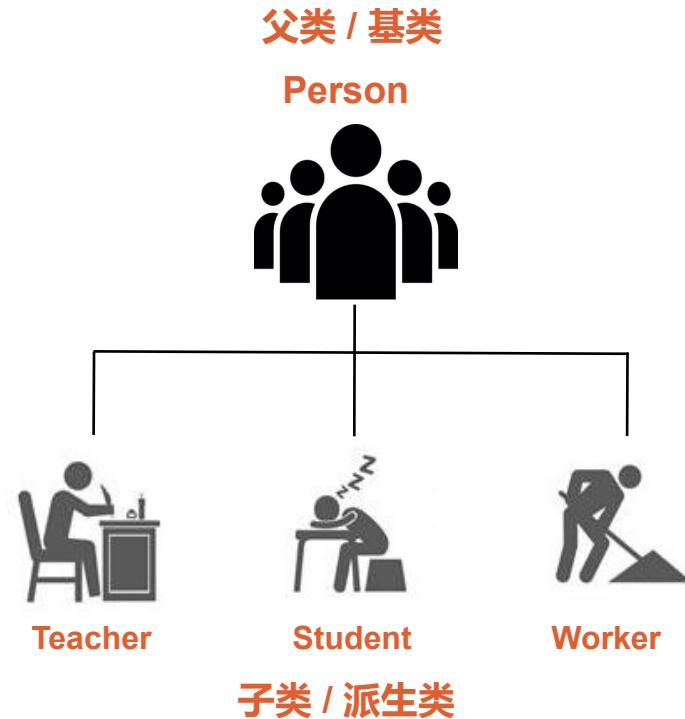


4.3 Python面向对象的编程

■ **类 (class)**: 具有相同的属性和方法的对象集合。

- 对象是类的**实例**
- 子类**继承**了父类的**全部属性和方法**，
并且也有自己**特有的属性和方法**。
- **继承**描述了类之间的**层次关系**

类和对象并非仅限于具体的事物，它也
可以是一种**抽象的概念或者规则**。



兰州交通大学

计算机科学与技术学院

4.3 Python面向对象的编程

□ 声明类

```
class 类名:  
    类属性=初值  
    方法(参数列表)
```

- 类属性是类中**所有对象共同拥有的属性**
- 它在内存中只存在**一个副本**
- 可以通过**类名**访问，也可以被类的所有**对象**访问
- 在类定义之后，可以通过类名**添加类属性**，新增的类属性也被**类和所有对象共有**。

□ 访问类属性

```
类名.类属性  
对象名.类属性
```

类的特殊属性

| 类属性 | 含义 |
|-------------------------|-------------|
| <code>__name__</code> | 类的名字 (字符串) |
| <code>__doc__</code> | 类的文档字符串 |
| <code>__bases__</code> | 类的所有父类组成的元组 |
| <code>__dict__</code> | 类的属性组成的字典 |
| <code>__module__</code> | 类所属的模块 |
| <code>__class__</code> | 类对象的类型 |



兰州交通大学

计算机科学与技术学院

4.3 Python面向对象的编程

□ 声明类

```
class 类名:  
    类属性=初值  
    方法(参数列表)
```

self, 参数2, 参数3, ...

- 方法中必须有一个参数self，而且它是参数列表中的第一个参数
- 由同一个类可以生成很多个对象，每一个对象，都有一个专属的self，代表这个对象自身

□ 创建对象

对象名 = 类名()

```
1 class Person():  
2     money=10000  
3     def say_hello(self):  
4         print("Hello!")  
5  
6 zhangsan = Person()  
7  
8 print(zhangsan.money)  
9 zhangsan.say_hello()
```

对象拥有Person类中的所有属性和方法

运行结果：

```
10000  
Hello!
```



兰州交通大学

计算机科学与技术学院

4.3 Python面向对象的编程

□ 创建对象之后，动态添加对象属性

```
1 class Person():
2     money=10000
3     def say_hello(self):
4         print("Hello!")
5
6 zhangsan = Person()
7
8 zhangsan.major="computer"
9 print("zhangsan's major is",zhangsan.major)
```

这是一个动态添加的实例属性，
它只属于zhangsan自己，如果重新创建其他的Person()对象，是
没有这个属性的。

运行结果：

```
zhangsan's major is computer
```



西安科技大学

计算机科学与技术学院

4.3 Python面向对象的编程

```
1 class Person():
2     money=10000
3     def say_hello(self):
4         print("Hello!")
5
6 zhangsan = Person()
7 zhangsan.major="computer"
8 print("zhangsan's major is",zhangsan.major)
9
10 lisi = Person()
11 print("lisi's major is",lisi.major)
```

运行结果：

```
zhangsan's major is computer
-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-5-5cf93aa47a05> in <module>()
      9
     10 lisi = Person()
--> 11 print("lisi's major is",lisi.major)

AttributeError: 'Person' object has no attribute 'major'
```

错误信息



4.3 Python面向对象的编程

```
1 class Person():
2     money=10000
3     def say_hello(self):
4         print("Hello!")
5
6 zhangsan = Person()
7 lisi=Person()
8
9 print(Person.money)
10 print(zhangsan.money)
11 print(lisi.money)
12
13 Person.money=9000
14
15 print(Person.money)
16 print(zhangsan.money)
17 print(lisi.money)
18
19 zhangsan.money=5000
20
21 print(Person.money)
22 print(zhangsan.money)
23 print(lisi.money)
```

类属性, 所有的实例共同享有。

运行结果:

| |
|-------|
| 10000 |
| 10000 |
| 10000 |
| 9000 |
| 9000 |
| 9000 |
| 9000 |
| 5000 |
| 9000 |

修改类属性, 所有实例的值都会发生变化。

zhangsan自己又创建了一个**实例属性**money, 它已经不是原来的类属性money了。

4.3 Python面向对象的编程

□ 删除对象

```
del 对象名
```

```
del zhangsan
```

在执行完这条语句后，zhangsan对象就不存在了，如果再访问zhangsan对象，就会出现错误提示：

```
>>>zhangsan.major
-----
NameError                                                 Traceback (most recent call last)
<ipython-input-28-2c73fb3b1bdb> in <module>()
----> 1 zhangsan.major

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



■ 构造函数：在创建对象时，用来完成初始化操作。

```
__init__(self, 参数2, 参数3, ...)
```

- 当创建对象时，系统会自动调用构造函数。
- 可以把对成员变量赋初值的语句写在构造函数中，从而保证每个变量都有合适的初始值。

■ 析构函数：在清除对象时，回收和释放对象所占用的资源。

```
__del__()
```

在Python中，构造函数和析构函数也**可以省略**。



复旦科技大学

计算机科学与技术学院

4.3 Python面向对象的编程

```
1 class Person:  
2     def __init__(self, name, age, gender="男"):  
3         self.name=name  
4         self.age=age  
5         self.gender=gender  
6     def __del__(self):  
7         print("Bye bye—from", self.name)  
8     def printInfo(self):  
9         print("姓名:", self.name, "年龄:", self.age, "性别:", self.gender)  
10  
11 zhangsan=Person("张三", 18)  
12 lisi=Person("李四", 19, "女")  
13  
14 zhangsan.printInfo()  
15 lisi.printInfo()  
16  
17 del zhangsan  
18 del lisi
```

运行结果：

```
姓名：张三 年龄：18 性别：男  
姓名：李四 年龄：19 性别：女  
Bye bye—from 张三  
Bye bye—from 李四
```



兰州科技大学

计算机科学与技术学院

■ 静态方法和类方法

□ 类方法

```
class 类名:  
    @classmethod  
    def 类方法名(cls,...):  
        方法体
```

□ 静态方法

```
class 类名:  
    @staticmethod  
    def 类方法名():  
        方法体
```

- 可以通过类名或对象名调用。
- 不能访问实例属性，但可以访问类属性。

- 可以通过类名或对象名调用。
- 不能访问实例属性，也不能直接访问类属性。
但是可以通过类名引用类属性。



■ 公有变量和私有变量

- 公有变量：可以在类的**外部**访问。
- 保护变量：只能允许其**本身**和**子类**进行访问。
- 私有变量：不允许在类的外部访问。

__xxx : 私有变量

_xxx : 保护变量

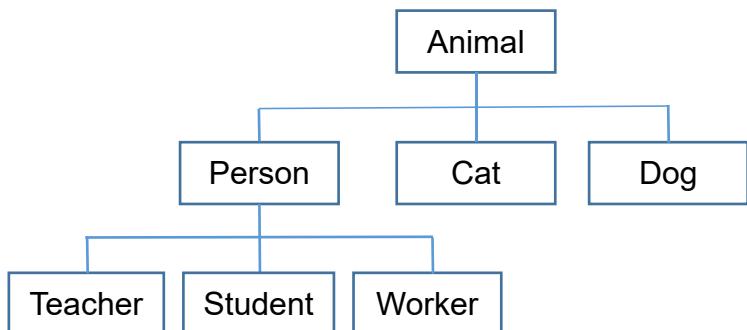
__xxx__ : 专有变量，方法



4.3 Python面向对象的编程

■ 继承(inheritance): 子类能够继承父类中所有**非私有的**成员变量和成员函数。

```
class 子类名(父类名 )  
    类属性=初值  
    方法(参数列表)
```



```
1 class Person():  
2     money=10000  
3     def say_hello(self):  
4         print("Hello!")  
5  
6 class Teacher(Person):  
7     pass  
8  
9 amy = Teacher()  
10  
11 print(amy.money)  
12 amy.say_hello()
```

运行结果:

```
10000  
Hello!
```



- **类**(class): 对具有**相同属性和方法**的一组对象的描述，它定义了所有对象所共有的属性和方法。
- **对象**(object): 是类中的一个具体的**实例**(instance)。
- **属性**(attribute): 是类和对象中的**变量**。
 - **类属性**: 定义在类的内部，方法的外部。是类中所有对象**共同拥有**的属性。
 - **实例属性**: 也叫做**成员变量**，在每个对象中都有自己的副本。



■ **方法**(method): 是在类中所定义的**函数**, 它描述了对象能执行的**操作**。

□ **实例方法/成员函数**: 只能通过**对象名**调用, 第一个参数必须是self。

构造函数和析构函数

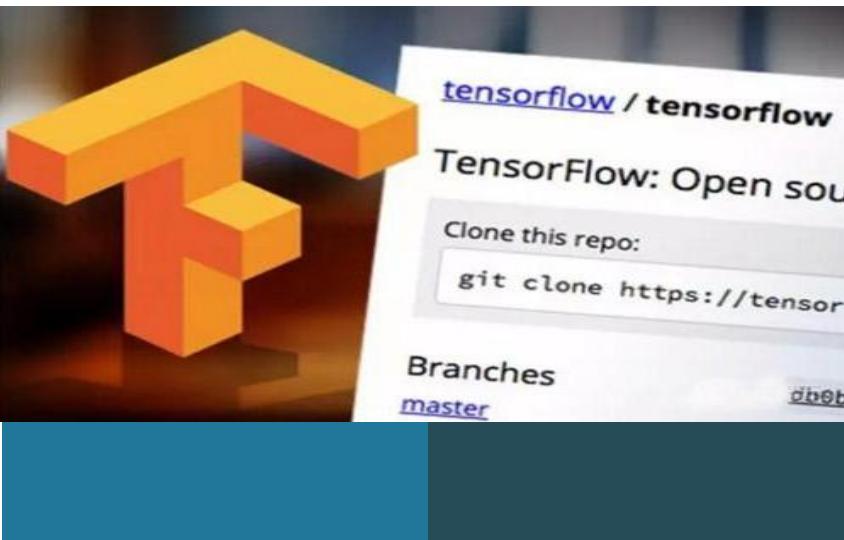
□ **类方法**: 可以通过**类名**或**对象名**调用, 第一个参数必须是 “cls” 。

□ **静态方法**: 通过类名或对象名调用, 没有 “self” 或 “cls” 参数。

■ **成员变量**: 公有变量、保护变量、和私有变量。

■ **继承**: 是**子类**自动的**共享父类中的属性和方法**的机制。





4.4 文件

■ 打开文件

文件对象 = `open (文件名, 访问模式)`

□ **绝对路径**: 从**盘符**开始的路径

C:\windows\system32\cmd.exe

C:\Users\Administrator\.jupyter_notebook_config.py

D:\Program Files\Anaconda3\lib\site-packages\ipykernel__main__.py

□ **相对路径**: 从**当前目录 (工作目录)**的路

径
D:\jupyter\example

\python\file.txt

D:\jupyter\example

..\file.txt

D:\jupyter\example

..\tensorflow\file.txt

D:\jupyter\example\python\file.txt

D:\jupyter\file.txt

D:\jupyter\tensorflow\file.txt



□ 获取当前路径

```
import os  
print (os.getcwd())
```

current working directory



□ 访问模式

文件对象 = open (文件名, 访问模式)

参数“访问模式”的可取值

| 访问模式 | 执行操作 |
|------|-----------------------|
| 'r' | 以只读方式打开文件 |
| 'w' | 以写入方式打开文件, 会覆盖已经存在的文件 |
| 'a' | 以写入方式打开文件, 在末尾追加写入 |
| '+' | 以可读写的方式打开文件 |
| 'b' | 以二进制模式打开文件 |
| 't' | 以文本模式打开文件(默认) |

| 访问模式 | 执行操作 |
|------|---------|
| 'rb' | 二进制读模式 |
| 'wb' | 二进制写模式 |
| 'ab' | 二进制追加模式 |



4.4 文件

例：在C盘根目录下，创建文本文件 `myfile.txt`，然后使用 `open()` 函数打开它：

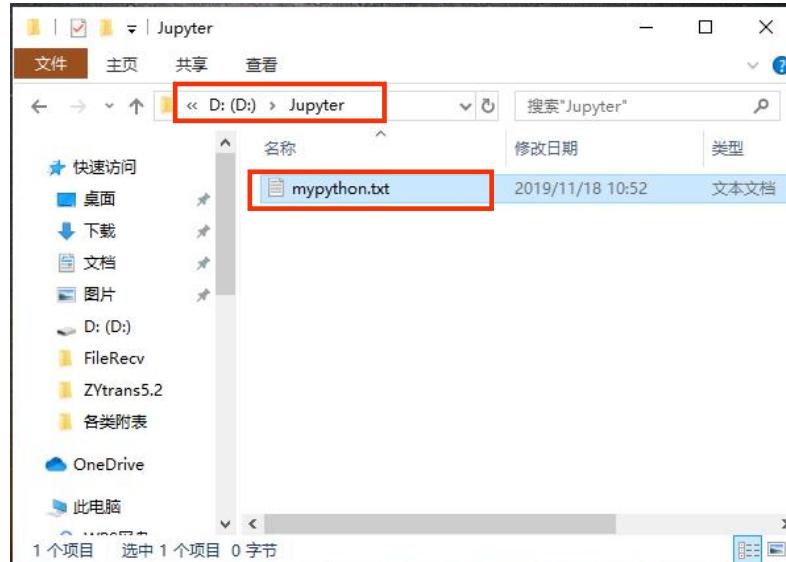
```
f = open("c:/myfile.txt")
```

使用 `open()` 函数成功打开一个文件之后，
它会返回一个**文件对象**；否则，就会返
回一个错误。

只有文件名，默认为**当前工作目录**

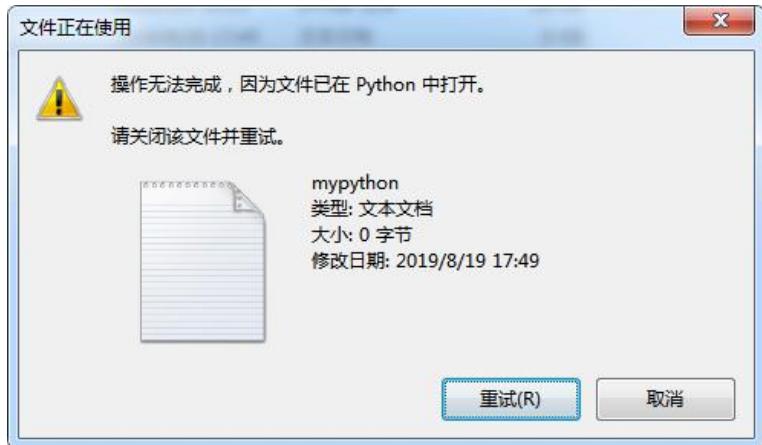
```
f = open("mypython.txt",'w')
```

访问模式是 "w"，如果在工作目录下找不到这个文件，
就会自动创建它。



4.4 文件

在资源管理器中**删除**上一步创建的文件 mypython.txt，会出现错误提示：



□ 关闭文件

文件对象.close()

f.close()

- Python有**垃圾回收机制**，会自动关闭不再使用的文件
- 在对文件进行了**写入操作**后，应该立刻关闭文件，以避免意外事故造成的错误

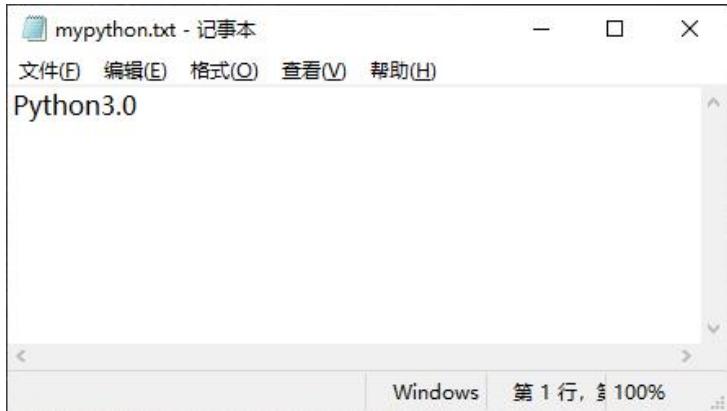


兰州交通大学

计算机科学与技术学院

□ 读取文件的内容

在资源管理器中，打开mypython.txt 文本文件，写入 “Python3.0”



□ read()方法：读取整个文档

文件对象.read()

```
>>>f = open("mypython.txt")
>>>f.read()
'Python3.0'
```



兰州交通大学

计算机科学与技术学院

例：读取文件

step1:

在资源管理中，在当前工作目录下，创建文本文件 **The Zen of Python.txt**，输入文本内容。

The Zen of Python.txt

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!



复旦科技大学

计算机科学与技术学院

4.4 文件

step2: 打开并读取文件内容

```
>>>fp=open("the zen of python.txt")
>>>fp.read()
"The Zen of Python, by Tim Peters\n\nBeautiful is better than ugly.
Explicit is better than implicit.\nSimple is better than complex.\n
Complex is better than complicated.\nFlat is better than nested.\n
Sparse is better than dense.\nReadability counts.\nSpecial cases
aren't special enough to break the rules.\nAlthough practicality
beats purity.\nErrors should never pass silently.\nUnless explicitly
silenced.\nIn the face of ambiguity, refuse the temptation to guess.
\nThere should be one-- and preferably only one --obvious way to
do it.\nAlthough that way may not be obvious at first unless you're
Dutch.\nNow is better than never.\nAlthough never is often better
than *right* now.\nIf the implementation is hard to explain, it's a
bad idea.\nIf the implementation is easy to explain, it may be a
good idea.\nNamespaces are one honking great idea -- let's do
more of those!"
```



❑ `readline()`方法：每次只读取文件中的一行

文件对象. `readline()`

```
>>>fp=open("the zen of python.txt")
>>>fp.readline()
"The Zen of Python, by Tim Peters\n"

>>>fp.readline()
"\n"

>>>fp.readline()
"Beautiful is better than ugly.\n"
>>>fp.readline()
"Explicit is better than implicit.\n"
>>>fp.readline()
"Simple is better than complex.\n"
```



□ 文件指针

read()

The Zen of Python.txt

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!



□ 文件指针

readline()

The Zen of Python.txt

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!



复旦科技大学

计算机科学与技术学院

4.4 文件

□ 读取文件的内容——指定字节数

文件对象. `read(字节数)`

文件对象. `readline(字节数)`



4.4 文件

"The Zen of Python, by Tim Peters\n\nBeautiful **is** better than ugly.\n**Explicit is** better than implicit.\nSimple **is** better than complex.\n**Complex is** better than complicated.\nFlat **is** better than nested.\nSparse **is** better than dense.\nReadability counts.\nSpecial cases aren't special enough to **break** the rules.\nAlthough practicality beats purity.\nErrors should never **pass** silently.\nUnless explicitly silenced.\nIn the face of ambiguity, refuse the temptation to guess.\nThere should be one-- **and** preferably only one --obvious way to do it.\nAlthough that way may **not** be obvious at first unless you're Dutch.\nNow **is** better than never.\nAlthough never **is** often better than *right* now.\nIf the implementation **is** hard to explain, it's a bad idea.\nIf the implementation **is** easy to explain, it may be a good idea.\nNamespaces are one honking great idea -- let's do more of those!"

```
>>>fp.readline(10)
'Complex is'
>>>fp.read(8)
' better '
```



■ 向文件中写入数据

write(写入内容)

- 在使用write()函数之前，要确保open()函数的**访问模式**，是**支持写入**的。
- 写入成功之后，会返回写入的**字符数**，是一个整数。

```
f = open("myfile.txt", "w")
f.write("Hello, World!")
```



4.4 文件

例：完整实例——打开文件，读取文件，写入文件，关闭文件

```
>>>f = open("myfile.txt","w")
>>>f.write("Hello!")
>>>f.close()

>>>f = open("myfile.txt")
>>>f.read()
'Hello!'
>>>f.close()

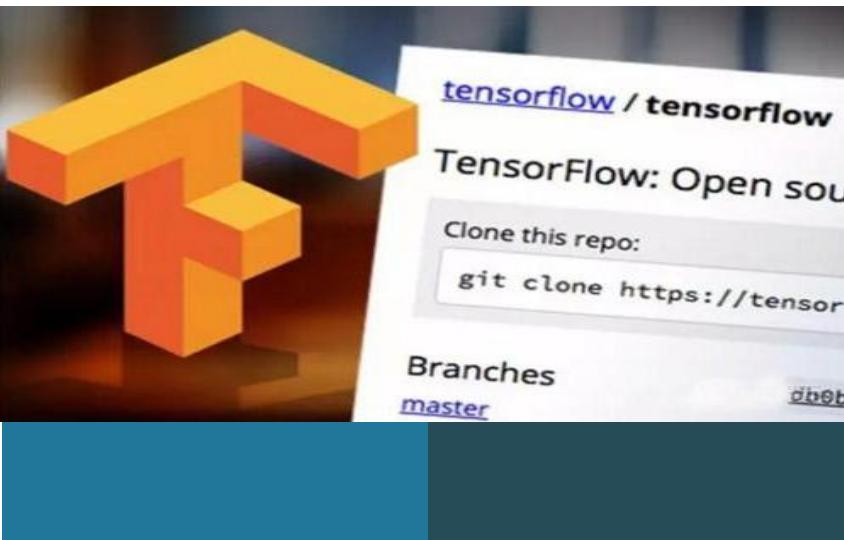
>>>f = open("myfile.txt","a")
>>>f.write("World!")
6
>>>f.close()
```

```
>>>f = open("myfile.txt")
>>>f.read()
'Hello!World!'
>>>f.close()

>>>f = open("myfile.txt","w")
>>>f.write("Python3.0")
9
>>>f.close()

>>>f = open("myfile.txt")
>>>f.read()
Python3.0
```





4.5 异常处理

■ **异常**: 程序运行时的**错误**, 对应一个**Python对象**。

□ try-except语句

try:

 语句块

except 异常1 as 错误原因:

 出现异常1后的处理代码

except 异常2 as 错误原因:

 出现异常2后的处理代码

- 在程序运行时, 解释器尝试执行try语句块中的所有代码。
- 如果语句块被执行完后没有异常发生, 就会忽略except后的代码。
- 当某个except所指定的异常发生后, 会忽略try子句中剩余的语句, 直接跳转到对应异常的处理代码处执行。



4.5 异常处理

例：异常处理

```
try:  
    aList = [0,1,2]  
    print(aList[3])  
    print("try语句块继续执行中.....")  
except IndexError as e:  
    print(e)  
    print("异常已处理")  
  
print("程序继续执行中.....")
```

运行结果：

```
list index out of range  
异常已处理  
程序继续执行中.....
```

```
try:  
    aList = [0,1,2]  
    print(aList[2])  
    print("try语句块继续执行中.....")  
except IndexError as e:  
    print(e)  
    print("异常已处理")  
  
print("程序继续执行中.....")
```

运行结果：

```
2  
try语句块继续执行中.....
```



兰州交通大学

计算机科学与技术学院

□ Python中常见的异常

- **IOError**: 输入/输出异常; (基本上是无法打开文件)
- **ImportError**: 无法导入模块或包; (出现这个异常基本上是路径问题或名称错误)
- **IndentationError**: 缩进错误;(代码没有正确对齐)
- **NameError**: 没有声明、或初始化对象
- **KeyError**: 试图访问字典里不存在的键
- **AttributeError**: 试图访问一个对象没有的属性
- **TypeError** : 类型不匹配
- **ValueError**: 传入一个调用者不期望的值, 即使值的类型是正确的



4.5 异常处理

□ Python异常类的继承关系

```

1  BaseException
2    +-- SystemExit
3    +-- KeyboardInterrupt
4    +-- GeneratorExit
5    +-- Exception
6      +-- StopIteration
7      +-- ArithmeticError
8        +-- FloatingPointError
9        +-- OverflowError
10       +-- ZeroDivisionError
11     +-- AssertionError
12     +-- AttributeError
13     +-- BufferError
14     +-- EOFError
15   +-- ImportError
16   +-- LookupError
17     +-- IndexError
18     +-- KeyError
19   +-- MemoryError
20   +-- NameError
21     +-- UnboundLocalError
22   +-- OSError
23     +-- BlockingIOError
24     +-- ChildProcessError
25     +-- ConnectionError
26       +-- BrokenPipeError
27       +-- ConnectionAbortedError
28       +-- ConnectionRefusedError
29       +-- ConnectionResetError
30     +-- FileExistsError
31     +-- FileNotFoundError
32     +-- InterruptedError
33     +-- IsADirectoryError
34     +-- NotADirectoryError
35     +-- PermissionError
36     +-- ProcessLookupError
37     +-- TimeoutError
38   +-- ReferenceError
39   +-- RuntimeError
40     +-- NotImplementedError
41   +-- SyntaxError
42     +-- IndentationError
43       +-- TabError
44   +-- SystemError
45   +-- TypeError
46   +-- ValueError
47     +-- UnicodeError
48       +-- UnicodeDecodeError
49       +-- UnicodeEncodeError
50       +-- UnicodeTranslateError
51   +-- Warning
52     +-- DeprecationWarning
53     +-- PendingDeprecationWarning
54     +-- RuntimeWarning
55     +-- SyntaxWarning
56     +-- UserWarning
57     +-- FutureWarning
58     +-- ImportWarning
59     +-- UnicodeWarning
60     +-- BytesWarning
61     +-- ResourceWarning

```

```

1  BaseException
2    +-- SystemExit
3    +-- KeyboardInterrupt
4    +-- GeneratorExit
5    +-- Exception
6      +-- StopIteration
7      +-- ArithmeticError
8        +-- FloatingPointError
9        +-- OverflowError
10       +-- ZeroDivisionError
11     +-- AssertionError
12     +-- AttributeError
13     +-- BufferError
14     +-- EOFError
15   +-- ImportError
16   +-- LookupError
17     +-- IndexError
18     +-- KeyError
19   +-- MemoryError
20   +-- NameError
21     +-- UnboundLocalError
22   +-- OSError
23     +-- BlockingIOError
24     +-- ChildProcessError
25     +-- ConnectionError
26       +-- BrokenPipeError
27       +-- ConnectionAbortedError
28       +-- ConnectionRefusedError
29       +-- ConnectionResetError
30     +-- FileExistsError
31     +-- FileNotFoundError
32     +-- InterruptedError
33     +-- IsADirectoryError
34     +-- NotADirectoryError
35     +-- PermissionError
36     +-- ProcessLookupError
37     +-- TimeoutError
38   +-- ReferenceError
39   +-- RuntimeError
40     +-- NotImplementedError
41   +-- SyntaxError
42     +-- IndentationError
43       +-- TabError
44   +-- SystemError
45   +-- TypeError
46   +-- ValueError
47     +-- UnicodeError
48       +-- UnicodeDecodeError
49       +-- UnicodeEncodeError
50       +-- UnicodeTranslateError
51   +-- Warning
52     +-- DeprecationWarning
53     +-- PendingDeprecationWarning
54     +-- RuntimeWarning
55     +-- SyntaxWarning
56     +-- UserWarning
57     +-- FutureWarning
58     +-- ImportWarning
59     +-- UnicodeWarning
60     +-- BytesWarning
61     +-- ResourceWarning

```



□ Python异常类的继承关系

```
1  BaseException
2  +-- SystemExit
3  +-- KeyboardInterrupt
4  +-- GeneratorExit
5  +-- Exception
6      +-- StopIteration
7      +-- ArithmeticError
8          +-- FloatingPointError
9          +-- OverflowError
10         +-- ZeroDivisionError
11     +-- AssertionError
12     +-- AttributeError
13     +-- BufferError
14     +-- EOFError
```

□ Exception

- 是所有**非系统退出类**异常类的基类
- 在编程时，可以通过捕获它，来避免程序遇到错误而退出

```
try:  
    语句块  
except exception as e:  
    异常处理语句块
```



□ finally子句：无论异常是否发生，都会执行

```
try:  
    语句块  
except 异常 as 错误原因:  
    出现异常后的处理代码  
finally:  
    语句块
```

- 如果try语句块中没有出现任何运行时错误，会跳过except语句块执行finally语句块的内容。
- 如果出现异常，则会先执行except语句块的内容，再执行finally语句块的内容。
- finally语句块经常用于**关闭资源**等清理工作。



4.5 异常处理

例：try-except-finally语句

```
try:  
    print("try语句块开始")  
    f = open('c:/test.txt')  
    print(f.read())  
    print("try语句块结束")  
except IOError as e:  
    print("except子句")  
    print (e)  
finally:  
    print("finally子句")  
    f.close()
```

运行结果：

try语句块开始
except子句
[Errno 2] No such file or directory: 'c:/test.txt'
finally子句

“找不到文件” 错误



复旦科技大学

计算机科学与技术学院

4.5 异常处理

在c盘根目录下创建文件test.txt，
并且写入 “Hello, Python! ”



```
try:  
    print("try语句块开始")  
    f = open('c:/test.txt')  
    print(f.read())  
    print("try语句块结束")  
except IOError as e:  
    print("except子句")  
    print (e)  
finally:  
    print("finally子句")  
    f.close()
```

运行结果：

```
try语句块开始  
Hello, Python!  
try语句块结束  
finally子句
```



兰州交通大学
计算机科学与技术学院



4.6 上下文管理器

4.6 上下文管理器

■ 异常处理

```
try:  
    f = open("mypython.txt")  
    print(f.read())  
except IOError as e:  
    print (e)  
finally:  
    f.close()
```

■ with语句

```
with open("mypython.txt") as f:  
    print(f.read())
```

- 使用with语句替代try-finally语句，代码更加的简洁清晰。
- 对于需要**对资源进行访问**的任务，无论在代码运行过程中，是否发生异常，都会执行必要的清理操作，**释放资源**。



■ with语句

```
with open("mypython.txt") as f:  
    print(f.read())
```

运行结果：

```
Python3.0
```

在with语句完成时，会**自动关闭文件**。如果再次读取这个文件，就会出现文件关闭的**错误提示信息**。

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

```
ValueError                                     Traceback (most recent call last)  
<ipython-input-5-9a7692027948> in <module>()  
      1 print(f.read())  
----> 2 ValueError: I/O operation on closed file.
```



■ 上下文管理器

- 上下文管理器是Python中的一种**协议**，它保证了每次代码执行的**一致性**
- 一旦进入上下文管理器，就一定会**按照规定的步骤**退出
- 如果合理的设计了退出上下文管理器的步骤，就能够很好的**处理异常**。
- 上下文管理器被最多用到的场景是**资源清理**操作。
- **实现上下文管理器**，只要在类定义时，实现**`_enter_()`方法**和**`_exit_()`方法**即可

```
class A():
    def __init__(self, val_a):
        self.a=val_a

    def __enter__(self):
        print("calss A's __enter__function.")

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("calss A's __exit__function.")
```



□ 实现上下文管理器

```
class A():

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

    def __enter__(self):
        print("calss A's __enter__function.")

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("calss A's __exit__function.")
```

如果with语句块正常结束，那么这三个参数全部是None。如果发生异常，那么这三个参数的值分别异常的类、实例和跟踪记录。

- 在with语句块**执行前**，首先会执行**enter()**方法
- 当with语句块**执行结束**，无论是否出现异常，都会调用**__exit__()**方法
- 通常将**清除、释放资源**的操作写在**__exit__()**方法中



4.6 上下文管理器

□ 使用with语句访问上下文管理器

上下文管理器对象，或函数(返回值是上下文管理器对象)

with 上下文管理器表达式 [as 变量]:
语句块

```
with open("mypython.txt") as f:  
    print(f.read())
```



例：模拟实现一个文件类

```
class File():

    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        print("执行__enter__()方法")
        self.f = open(self.filename, self.mode)
        return self.f

    def __exit__(self, *args):
        print("执行__exit__方法")
        self.f.close()
```



4.6 上下文管理器

以读的方式打开当前目录下的mypython文件，并输出文件内容。

```
with File('mypython.txt', 'r') as f:  
    print(f.read())
```

运行结果：
执行 `__enter__()` 方法
Python3.0
执行 `__exit__()` 方法

- 在执行with语句块之前，首先执行了`__enter__()`方法，然后再执行with语句块，最后执行`__exit__()`方法。
- 采用这种方式读取文件时，即使执行过程中出现了异常，`__exit__()`方法也会被执行，完成关闭文件的操作。



format 格式化输出

1. 常用方法 %

举例：

```
>>>print("今天是%d 年%d 月%d 日"%(2019,5,18))
今天是 2019 年 5 月 18 日
>>>print('第一个数字是%.5f,第二个数字是%d'%(0.678,10))
第一个数字是 0.67800,第二个数字是 10
```

2. 字符串的 format()方法

- 使用大括号 “{}” 代替 “%”
- str.format()

2.1 基本用法

(1) 不带编号

```
>>>print('{} {}'.format('hello','world'))
hello world
>>>print("今天是{}年{}月{}日".format(2019,5,18))
今天是 2019 年 5 月 18 日
```

(2) 带数字编号：参数个数没有限制，可以多次使用，顺序可以任意放置

```
>>>print('{0} {1}'.format('hello','world'))
hello world
>>> print('{0} {1} {0}'.format('hello','world'))
hello world hello
>>> print('{1} {1} {0}'.format('hello','world'))
world world hello
>>>print('她叫{0},今年{1}岁,她的朋友 lisa, 今年也{1}岁了'.format('Lucy',10))
她叫 Lucy,今年 10 岁,她的朋友 lisa, 今年也 10 岁了
```

(3) 带关键字

```
>>>print('她叫{name},今年{age}岁.'.format(age=10,name='Lucy'))
她叫 Lucy,今年 10 岁.
```

2.2 进阶用法

{ :填充字符 对齐方式 数据宽度 数据类型 }

(1) 填充：默认使用空格填充。=（只用于数字）在小数点后进行补齐

(2) 对齐：

| 符号 | 含义 |
|----|---------|
| < | 左对齐（默认） |
| > | 右对齐 |
| ^ | 中间对齐 |

举例：

宽度为 8，不够 8 位用！填充，右对齐，

```
>>>print('第一个数字是{:>8}'.format(123.98))
第一个数字是!!123.98
```

宽度为 8，不够 8 位默认用空格填充，右对齐，

```
print('第一个数字是{:>8}'.format(123.98)) >>第一个数字是 123.98
```

(3) 数据类型

| 符号 | 含义 |
|----|------|
| f | 浮点数 |
| b | 二进制 |
| d | 十进制 |
| o | 八进制 |
| x | 十六进制 |

举例：

第一个数字是浮点类型，小数点后保留 5 位有效数字；第二个数字是十进制整数

```
>>>print('第一个数字是{:.5f}, 第二个数字是{:d}'.format(0.678,10))
第一个数字是 0.67800, 第二个数字是 10
```

综合示例：

浮点类型，宽度为 8，小数点后保留 3 位有效数字，右对齐，不够 8 位宽度用“！”填充

```
>>>print('第一个数字是{:>8.3f}'.format(123.98))
>>第一个数字是!123.980
```

补充资料：matplotlib.tight_layout()

参考：https://matplotlib.org/stable/api/tight_layout_api.html

当画布中有多个子图时，tight_layout()会自动调整子图参数，使之填充整个图像区域。但是可能在某些情况下，使用tight_layout()函数的默认参数并不会达到我们想要的效果，这时就需要对它的参数进行设置，从而达到我们想要的布局效果。以下是matplotlib官网中的说明：

matplotlib.tight_layout

Routines to adjust subplot params so that subplots are nicely fit in the figure. In doing so, only axis labels, tick labels, axes titles and offsetboxes that are anchored to axes are currently considered.

Internally, this module assumes that the margins (left margin, etc.) which are differences between `Axes.get_tightbbox` and `Axes.bbox` are independent of Axes position. This may fail if `Axes.adjustable` is `datalim` as well as such cases as when left or right margin are affected by xlabel.

```
matplotlib.tight_layout.auto_adjust_sublpotpars(fig, renderer, nrows_ncols,
num1num2_list, subplot_list, ax_bbox_list=None, pad=1.08, h_pad=None, w_pad=None,
rect=None)
```

[source]

[Deprecated] Return a dict of subplot parameters to adjust spacing between subplots or `None` if resulting axes would have zero height or width.

Note that this function ignores geometry information of subplot itself, but uses what is given by the `nrows_ncols` and `num1num2_list` parameters. Also, the results could be incorrect if some subplots have `adjustable=datalim`.

Parameters: `nrows_ncols : tuple[int, int]`

Number of rows and number of columns of the grid.

`num1num2_list : list[tuple[int, int]]`

List of numbers specifying the area occupied by the subplot

`subplot_list : list of subplots`

List of subplots that will be used to calculate optimal subplot_params.

`pad : float`

Padding between the figure edge and the edges of subplots, as a fraction of the font size.

`h_pad, w_pad : float`

Padding (height/width) between edges of adjacent subplots, as a fraction of the font size. Defaults to `pad`.

`rect : tuple[float, float, float, float]`

[left, bottom, right, top] in normalized (0, 1) figure coordinates.

参数 `pad`, `h_pad`, `w_pad` 和 `rect` 用来指定子图的间距和位置，默认值如下：

`tight_layout(pad=1.08, h_pad=None, w_pad=None, rect=None)`

其中，参数 `rect` 指定子图区域，如果设置这个值，所有的子图都调整到这个矩形区域中；在课程的 6.1 小节中进行了详细的说明。

参数 `pad`, `h_pad` 和 `w_pad` 用来调整画布边缘距离和子图间距。

`pad`: 设置画布与子图之间的间距，默认为 1.08。

`h_pad`, `w_pad`: 设置相邻子图之间的间距。