**第一个Python程序**

***第一个Python程序：***

***python源程序的基本概念：***

1、Python源程序就是**一个特殊格式的文本文件**，可以**使用任意文本编辑软件**做Python的开发

2、Python程序的 **文件扩展名** 通常是 **.py**

***认识错误（BUG）：***

·编写的程序**不能正常执行**，或者执行的**结果不是我们期望的**

·俗称BUG，是程序员在开发时非常常见的，常见错误主要有

1、手误

2、对已经学习过的知识理解还存在不足

3、对语言还有需要学习和提升的内容

·在学习语言时，不仅要学会语言的语法，而且还要学会如何认识错误和解决错误的方法。

**※每一个程序员都是在不断地修改错误中成长的**

NameError: 名称错误

SyntaxError: 语法错误

IndentationEror: 缩进错误

***执行Python程序的三种方式：***

**解释器python/python3**

python解释器

python xxx.py python3 xxx.py

其他解释器

·CPython——官方版本的C语言实现

·Jython——可以运行在Java平台

·IronPython——可以运行在.NET和Mono平台

·PyPy——Python实现的，支持JIT即时编译

**交互式运行Python程序**

·直接在终端中运行的解释器，而不输入要执行的文件名

·在Python的Shell中直接输入Python代码，会立即看到程序执行结果

·IPython是一个Python的交互式shell：

·支持自动补全

·自动缩进

·支持 bash shell 命令

·内置了许多很有用的功能和函数

·Ipython是基于BSD开源的

·ipython ipython3

**Python的IDE——PyCharm**

**程序安装：**

**1、程序文件目录**

·将安装包解压缩，并且移动到/opt目录下

·所有的相关文件都保存在解压缩的目录中

**2、配置文件目录**

·启动PyCharm后，会在用户家目录下建立一个.PyCharmxxx的隐藏目录

·保存PyCharm相关的配置信息

**3、快捷方式文件**

·/usr/share/applications/jetbrains-pycharm.desktop

·在ubuntu中，应用程序启动的快捷方式通常都保存在/usr/share/applications目录下

**程序卸载：**

·要卸载PyCharm只需要做以下两步工作：

**·删除解压缩目录**

$ sudo rm –r /opt/pycharm-2016.3.1/

**·删除家目录下用于保存配置信息的隐藏目录**

$ rm –r ~/.PyCharm2016.3/

**·如果不在使用PyCharm还需要将/usr/share/applications下的jetbrains-pycharm.desktop删掉**

**设置专业版启动图标：**

·在专业版中，选择菜单Tools/Create Desktop Entry…

·设置图标时，需要勾选Create the entry for all users

**多文件项目演练：**

·开发**项目**就是开发一个**专门解决一个复杂业务功能的软件**

·通常每一个项目就具有一个 **独立的专属目录**，用于保存**所有和项目相关的文件**，一个项目通常会包含 **很多源文件**

·对于商业项目而言，通常在一个项目中，只有一个可以 **直接执行的Python源程序**

**注释**

***注释的作用：***

使用自己熟悉的语言，在程序中对某些代码进行标注说明，增强程序的可读性

***单行注释（行注释）：***

·以 # 开头，# 右边的所有东西都被当做说明文字，而不是真正要执行的程序，只起到辅助说明作用

# 这是第一个单行注释

print(“hello python”)

·为了保证代码的可读性，# 后面建议先添加一个空格，然后再编写相应的说明文字

***在代码后面增加的单行注释：***

·在程序开发时，同样可以使用 # 在代码的后面（旁边）增加说明性的文字

·但是，需要注意的是，**为了保证代码的可读性，注释和代码之间至少要有两个空格**

print(“hello python”) # 输出 `hello python`

***多行注释（块注释）：***

·如果希望编写的**注释信息很多，一行无法显示**，就可以使用多行注释

·要在Python程序中使用多行注释，可以用 **一对连续的三个引号**（单引号和双引号都可以）

‘‘‘

这是一个多行注释

在多行注释之间，可以写很多很多的内容

’’’

print(“hello python”)

***什么时候需要使用注释：***

1、**注释不是越多越好，**对于一目了然的代码，不需要添加注释

2、**对于复杂的操作，**应该在操作开始前写上若干行注释

3、对于**不是一目了然的代码，**应在其行尾添加注释（为了提高可读性，注释应该至少离开代码2个空格）

4、绝不要描述代码，假设阅读代码的人比你更懂Python，他只是不知道你的代码要做什么

**算术运算符**

***算术运算符：***

·算术运算符是**运算符的一种**

·是完成基本的算术运算使用的符号，用来处理四则运算

运算符 描述 实例

+ 加 10 + 20 = 30

- 减 10 – 20 = -10

\* 乘 10 \* 20 = 200

/ 除 10 / 20 = 0.5

// 取整除 返回除法的整数部分（商）9//2输出4

% 取余数 返回除法的余数9 % 2 = 1

\*\* 幂 又称次方、乘方， 2 \*\* 3 = 8

·在Python中 \* 运算符还可以用于字符串，计算结果就是字符串重复指定次数的结果

>>>“ - ” \* 50

>>>`--------------------------------------------------`

***算术运算符的优先级：***

·和数学中的运算符的优先级一直，在Python中进行数学计算时

·先乘除后加减

·同级运算符是 从左至右 计算

·可以使用 （） 调整计算的优先级

运算符 描述

\*\* 幂（最高优先级）

\* / % // 乘、除、取余数、取整数

+ - 加法、减法

**程序执行原理**

***计算机的三大件：***

**CPU:**

·中央处理器，是一块超大规模的集成电路

·负责 **处理数据/计算**

**内存：**

·**临时** 存储数据（断电之后，数据会消失）

·速度快、空间小、价格高

**硬盘：**

·**永久** 存储数据

·速度慢、空间大、价格低

***程序执行的原理：***

1、程序**运行之前**，程序是 **保存在硬盘** 中的

2、当要运行一个程序时

·操作系统会首先让**CPU**把程序复制到 **内存** 中

·**CPU**执行 **内存** 中的 **程序代码**

※**程序要执行，首先要被加载到内存**

***Python程序执行原理：***

1、操作系统首先让**CPU**吧**Python解释器**的程序复制到**内存**中

2、**Python解释器**根据语法规则，**从上向下**让**CPU**翻译**Python程序中的代码**

3、**CPU**负责执行翻译完成的代码

***程序的作用：***

程序就是 **用来处理数据** 的！

**变量的基本使用**

※**程序就是用来处理数据的，而变量就是用来存储数据的**

***变量定义：***

·在Python中，每个变量**在使用前都必须赋值**，变量**赋值以后**该变量**才会被创建**

·等号（=）用来给变量赋值

·= 左边是一个变量名

·= 右边是存储在变量中的值

**变量名 = 值**

※**变量定义之后，后续就可以直接使用了**

·可以用 其他变量的计算结果 来定义变量

·变量定义后，后续就可以直接使用了

·**变量名** 只有在 **第一次出现才** 是 **定义变量**

·**变量名**再次出现，不是定义变量，而是直接使用之前定义过的变量

***变量的类型：***

**·在内存中创建一个变量**，会包括：

1、变量的名称

2、变量保存的数据

3、变量存储数据的类型

4、变量的地址（标示）

**·变量一共有4种数据类型：**

1、 str —— 字符串

2、 bool —— 布尔（真假）

3、 int —— 整数

4、 float —— 浮点数（小数）

·在Python中定义变量是 **不需要指定类型**，Python可以根据 = 等号右侧的值，自动推导出变量中存储数据的类型

·数据类型可分为 **数字型** 和 **非数字型**

·数字型

·整型（int）

·浮点型（float）

·布尔型（bool）

·真 True 非0数 ——非零即真

·假 False 0

·复数型（complex）

·主要用于科学计算，例如：平面场问题，波动问题、电感电容等问题

·非数字型

·字符串

·列表

·元组

·字典

**·使用type( )函数可以查看一个变量的类型**

type(name)

***不同类型变量之间的计算：***

**1、数字型变量 之间可以直接计算**

·在Python中，两个数字型变量时可以直接进行 算数运算的

·如果变量是 bool 型，在计算时

·True 对应的数字是 1

·False 对应的数字是 0

**2、字符串变量 之间可以使用 + 拼接生成新的字符串**

**3、字符串变量 可以和 整数 使用 \* 重复拼接相同的字符串**

**4、数字型变量 和 字符串 之间 不能进行其他计算**

***变量的输入：***

·所谓 输入，就是 用代码 获取 用户通过 键盘 输入的信息

·在Python中，如果要获取用户在键盘上的输入信息，需要使用到 **input()** 函数

**1、关于函数**

·一个 **提前准备好的功能**（别人或者自己写的代码），**可以直接使用**，而 **不用关心内部的细节**

**2、input 函数实现键盘输入**

·在Python中可以使用input函数从键盘等待用户的输入

·用户输入的 **任何内容** Python都认为是一个 **字符串**

**字符串变量 = input(“提示信息：”)**

**3、类型转换函数**

函数 说明

int(x) 将 x 转换为一个整数

float(x) 将 x 转换为

**※ 两个函数是能够嵌套使用的 print = float(input(“数字”))**

***变量的格式化输出：***

·在Python中可以使用print函数将信息输出到控制台

·如果希望输出文字信息的同时，**一起输出 数据**，就需要使用到 **格式化操作符**

·**%** 被称为 **格式化操作符**，专门用于处理字符串中的格式

·包含 **%** 的字符串，被称为 **格式化字符串**

·**%** 和不同的 **字符** 连用，**不同类型的数据** 需要使用 **不同的格式化字符**

格式化字符 含义

**%s**  字符串

**%d** 有符号十进制整数，%06d表示输出的整数显示位数

**%f**  浮点数，%.02f表示小数点后只显示两位

**%%**  输出 %

**·语法格式**：

print(“格式化字符串” % 变量1)

print(“格式化字符串” % (变量1，变量2，……))

**变量的命名**

***标识符和关键字：***

***标识符：***

标识符就是程序员定义的变量名、函数名

**名字** 需要有 **见名知义** 的效果

**·**标识符可以有 **字母、下划线** 和 **数字** 组成

**·不能以数字开头**

**·不能与关键字重名**

**关键字：**

**·关键字** 就是在 Python内部已经使用的标识符

**·关键字** 具有特殊的功能和含义

**·**开发者 **不允许定义和关键字相同的名字的标识符**

通过命令查看Python中的关键字

>>>import keyword

>>>print(keyword.kwlist)

·**import关键字**可以导入一个“**工具包**”

***变量的命名规则：***

**命名规则** 可以被视为一种 **惯例**，并无绝对与强制

目的是为了 **增加代码的表示和可读性**

※python中的 **标识符** 是 **区分大小写的**

1、在定义变量时，为了保证代码格式，“ **=** ” 的左右应该各保留一个空格

2、在Python中，如果 变量名 需要由 二个 或者 多个单词 组成时，可以按照以下方式命名：

a、每个单词都使用小写字母

b、单词与单词之间使用 **\_ 下划线** 连接

c、例如：first\_name 、last\_name 、qq\_number 、qq\_password

**驼峰命名法：**

·当 变量名 是由二个或者多个单词组成时，还可以利用驼峰命名法来命名

·小驼峰式命名法

·第一个单词以小写字母开始，后续单词的首字母大写

·例如： firstName 、lastName

·大驼峰式命名法

·每一个单词的首字母都采用大写字母

·例如： FirstName 、LastName 、CamelCase

**判断（if）语句**

**判断的定义：**

·如果 **条件满足**，才能做某件事情

·如果 **条件不满足**，就做另外一件事情，或者什么也不做

正是因为有了判断语句，才使得程序世界丰富多彩，充满变化！

※**判断语句** 又被称为“分支语句”，正是因为有了判断，才让程序有了很多的分支

**if语句体验：**

**if判断语句基本语法：**

if 要判断的条件：

条件成立时，要做的事情

……

注意：代码的缩进为一个teb键，或者4个空格，建议使用空格。

·Python开发中，tab和空格不要混用！

**※可以把整个if语句看成一个完整的代码块**

**else处理条件不满足的情况：**

if 要判断的条件：

条件成立时，要做的事情

……

else：

条件不成立是，要做的事情

……

※if和else语句以及各自的缩进部分共同是一个 **完成的代码块**

**逻辑运算：**

·在程序开发中，通常 **在判断条件时**，会需要同时判断多个条件

·只有多个条件都满足，才能够执行后续代码，这个时候需要使用 **逻辑运算符**

· **逻辑运算符** 可以把 **多个条件** 按照 **逻辑** 进行 **连接**，变成 **更复杂的条件**

·Python中的 **逻辑运算符** 包括：**与and/或or/非not** 三种

**and：（条件1 and 条件2）**

**·与/并且**

**·两个条件同时满足，返回 True**

**·只要有一个不满足，就返回 False**

**or：（条件1 or 条件2）**

**·或/或者**

**·两个条件只要有一个满足，就返回 True**

**·两个条件都不满足，返回 False**

**not：（not 条件）**

**·非/不是 （条件取反）**

**if语句进阶：**

**elif：**

·在开发中，使用if可以 判断条件

·使用else可以处理 条件不成立 的情况

·但是，如果希望 在增加一些条件，条件不同，需要执行的代码也不同 时，就可以使用elif

if 条件1：

条件1满足执行的代码

……

elif 条件2：

条件2满足时，执行的代码

……

elif 条件3：

条件3满足时，执行的代码

……

else：

以上条件都不满足时，执行的代码

……

·对比逻辑运算符的代码

if 条件1 and 条件2：

条件1满足 **并且** 条件2满足 执行的代码

……

※elif和else都必须和if联合使用，而不能单独使用

※可以将if、elif和else以及各自缩进的代码，看成一个 **完整的代码块**

if的嵌套：

※elif的应用场景是：**同时** 判断 **多个条件**，所有条件 **平级**

·在开发中，使用if进行条件判断，如果希望 **在条件成立的执行语句中** 再 **增加条件判断**，就可以使用 **if的嵌套**

·**if的嵌套** 的应用场景就是：**在之前条件满足的前提下，再增加额外的判断**

·**if的嵌套** 的语法格式，**除了缩进之外** 和之前的没有区别

if 条件1：

条件1满足执行的代码

……

if条件1基础上的条件2：

条件2满足时，执行的代码

……

条件2不满足的处理

else：

条件2不满足时，执行的代码

……

else：

条件1不满足是，执行的代码

……

**随机数的处理：**

·在Python中，要使用随机数，首选需要导入 **随机数** 的 **模块**

**import random**

·导入模块后，可以直接在 模块名称 后面敲一个 . 然后按tab键，会提示该模块中包含的所有函数

·random.randint(a, b),返回[a, b] 之间的整数，包含a 和 b

random.randint(12, 20) # 生成的随机数n： 12 <= n <= 20

random.randint(20, 20) # 结果永远是 20

random.randint(20, 10) # 该语句是错误的，下限须小于上限

**循环**

**程序的三大流程：**

**顺序** —— **从上向下**，顺序执行代码

**分支** —— 根据条件判断，决定执行代码的 **分支**

**循环** —— 让 **特定代码 重复** 执行

**while循环基本使用：**

·循环的作用就是让 **指定的代码** 重复执行

·while 循环最常用的应用场景就是 **让执行的代码** 按照 **指定的次数 重复** 执行

**while语句基本语法：**

**初始条件设置（i=0） —— 通常是重复执行的 计数器**

**while 条件（判断 计数器 是否到达 目标次数）：**

条件满足时，做的事情1

条件满足时，做的事情2

条件满足时，做的事情3

……（省略）……

**处理条件(i = i + 1)[计数器 +1]**

·while语句以及缩进部分是一个 **完整的代码块**

※循环结束后，之前定义的计数器条件的数值是依旧存在的

**死循环：**

由于程序员的原因，忘记 在循环内部 修改循环的判断条件，导致循环持续执行，程序无法终止！

**赋值运算符：**

·在Python中，使用 = 可以给变量赋值

·在算数运算时，为了简化代码的编写，Python还提供一系列的 与 **算术运算符** 对应的 **赋值运算符**

**※赋值运算符中间不能使用空格**

运算符 描述 实例

= 简单的赋值运算符 c=a+b将a+b的结果赋值为c

+= 加法赋值运算符 c += a 等效于 c = c + a

-= 减法赋值运算符 c -= a 等效于 c = c – a

\*= 乘法赋值运算符 c \*= a 等效于 c = c \* a

/= 除法赋值运算符 c /= a 等效于 c = c / a

// 取整除赋值运算符 c //= a 等效于 c = c // a

%= 取余数赋值运算符 c %= a 等效于 c = c % a

\*\*= 幂赋值运算符 c \*\*= a 等效于 c = c \*\* a

***Python中的计数方法：***

**·自然计数法（从 1 开始）**—— 更符合人类的习惯

**·程序计数法（从 0 开始）**—— 几乎所有的程序语言都选择从0开始计数

**※在编写程序时，除非需求的特殊要求，否则 循环 的计数都是从0开始**

**循环计算：**

※在程序开发中，通常会遇到 **利用循环 重复计算** 的需求

1、在while上方定义一个变量，用于 **存放最终计算结果**

2、在循环体内部，每次循环都用 **最新的计算结果**，**更新** 之前定义的变量

***break和continue：***

break和continue是专门在循环中使用的关键字

**·break 某一条件满足时**，退出循环，不再执行后续重复的代码

**·continue某一条件满足时**，不执行后续重复的代码

**※break和continue只针对 当前所在循环 有效**

**·使用continue时，条件处理部分的代码，在continue之前**

**while循环嵌套：**

**·while嵌套就是：while里面还有while**

while 条件1：

条件满足时，做的事情

……（省略）……

while 条件2：

条件满足时，做的事情

……（省略）……

处理条件2

处理条件1

**print函数的末尾：**

·在默认情况下，**print**函数输出内容之后，会自动在内容末尾增加换行

·如果不希望末尾增加换行，可以在**print**函数输出内容的后面增加，**end=””**

·其中””中间可以指定print函数输出内容后，希望继续显示的内容

# 向控制台输出内容结束后，不会换行

print(“\*”,end=””)

# 单纯的换行

print(“”)

**※ end=”” 表示向控制台输出内容结束之后，不会换行**

字符串中的转义字符：

\t 在控制台输出一个 **制表符**，协助在输出文本时 垂直方向 保持对齐

\n 在控制台输出一个 **换行符**

**※制表符 的功能是在不使用表格的情况下在 垂直方向 按列对齐文本**

\\ 反斜杠符号 \’ 单引号 \” 双引号

\n 换行 \t 横向制表符 \r 回车

**函数基础**

***函数的基本感念：***

·所谓 **函数** ，就是把 **具有独立功能的代码块** 组织为一个小模块，在需要的时候 **调用**

·函数的使用包含两个步骤：

1、定义函数 —— **封装** 独立的功能

2、调用函数 —— 享受 **封装** 的成果

·**函数的作用**，在开发程序时，使用函数可以提高编写的效率以及代码的 **重用**

***函数的基本使用：***

**函数的定义：**

def 函数名( ):

函数封装的代码

……

1、**def** 是英文 **define** 的缩写

2、**函数名称**应该能够表达**函数封装代码** 的功能，方便后续使用

3、**函数名称**的命名应该 **符合 标识符命名规则**

**·**可以由 **字母、下划线** 和 **数字** 组成

**·不能以数字开头**

**·不能与关键字重名**

***函数调用：***

通过 **函数名( )** 即可完成对函数的调用

·定义好函数之后，只表示这个函数封装了一段代码而已

·如果不主动调用函数，函数是不会主动执行的

·不能将 **函数调用** 放在 **函数定义** 的上方，因为在 **使用函数名** 调用函数之前，必须保证Python已经知道函数的存在

***PyCharm的调试工具：***

**·F8 Step Over** 可以单步执行代码，会把函数调用看作是一行代码直接执行

**·F7 Step Into** 可以单步执行代码，如果是函数，会进入函数内部

***函数的文档注释：***

·在开发中，如果希望给函数添加注释，应该在 **定义函数** 的下方，使用 **连续的三对引号**

·在 **连续的三对引号** 之间编写对函数的说明文字

·在 **函数调用** 位置，使用快捷键 **CTRL+Q**可以查看函数的说明

※因为 **函数体相对比较独立，函数定义的上方**，应该和其他代码（包括注释）保留 **两个空行**

***函数的参数：***

**函数参数的使用：**

·在函数名的后面的小括号内部填写 **参数**

·多个参数之间使用 **，**分隔

**参数的作用：**

·**函数**，把 具 **有独立功能的代码块** 组织为一个小模块，在需要的时候 **调用**

·**函数的参数**，增加函数的 **通用性**，针对 **相同的数据处理逻辑**，能够 **适应更多的数据**

1、在函数 **内部**，把参数当做 **变量** 使用，进行需要的数据处理

2、函数调用时，按照函数定义的 **参数顺序**，把 **希望在函数内部处理的数据**，**通过参数**传递

**形参和实参：**

**·形参：定义** 函数时，小括号中的参数，是用来接收参数用的，在函数内部 **作为变量使用**

**·实参：调用** 函数时，小括号中的参数，是用来吧数据传递到 **函数内部** 用的

***函数的返回值：***

·在程序开发中，有时候，会希望 **一个函数执行结束后，告诉调用者一个结果**， 以便调用者针对具体的结果做后续的处理

·**返回值** 是函数 **完成工作** 后，最后 给调用者的 **一个结果**

·在函数中使用 **return** 关键字可以返回结果

·调用函数一方，可以 **使用变量** 来 **接收** 函数的返回结果

def sum\_2\_num(num1, num2):

return = num1 + num2

result = sum\_2\_num(10, 20)

print(“计算结果是 %d” % result)

函数的嵌套调用：

·一个函数里面 **又调用** 了 **另一个函数**，这就是 **函数嵌套调用**

·如果函数 test2 中，调用了另一个函数 test1

·那么执行到调用test1 函数时，会先把函数 test1 中的任务都执行完

·才会回到 test2 中调用函数 test1 的位置，继续执行后续代码

※工作中针对需求的变化，应该冷静思考，**不要轻易修改之前已经完成的，能够正常执行的函数！**

***使用模块中的函数：***

**※模块是 Python 程序构架的一个核心概念**

·**模块** 就好比是 **工具包**，想要使用这个工具包中的工具，就需要**导入 import** 这个模块

·每一个以扩展名 **py** 结尾的 Python 源代码文件都是一个**模块**

·在模块中定义的 **全局变量、函数** 都是模块能够提供给外界直接使用的工具

·可以 **在一个Python文件** 中 **定义 变量 或者 函数**

·然后在 **另外一个文件中** 使用 import 导入这个模块

·导入之后，就可以使用 ***模块名.变量/模块名.函数*** 的方式，使用这个模块中定义的变量或者函数

※**模块** 可以让 **曾经编写过的代码** 方便的被 **复用！**

***模块名也是一个标识符：***

**·**标识符可以由 **字母、下划线** 和 **数字** 组成

**·不能以数字开头**

**·不能与关键字重名**

※如果在Python文件起名时，**以数字开头** 是无法在PyCharm中通过导入这个模块的

**高级变量类型**

***非数字型变量：***

·在Python中，所有 **非数字型变量** 都支持以下特点

1、都是一个 **序列 sequence**，也可以理解为 **容器**

2、**取值[]**

3、**遍历 for in**

4、**计算长度、最大/最小值、比较、删除**

5、**链接 + 和 重复 \***

6、**切片**

***列表(List)：***

·List（列表）是 Python中使用 **最频繁** 的数据类型，在其他语言中通常叫做 **数组**

·专门用于存储 **一串信息**

·列表用 **[]** 定义，数据 之间使用 **,** 分隔

·列表的 **索引** 从 **0** 开始

·**索引** 就是数据在 **列表** 中的位置编号，**索引** 又可以被称为 **下标**

※从列表中取值时，如果 **超出索引范围**，程序会报错

***列表常用操作：***

·在 ipython3 中定义一个 列表，例如：name\_list = []

·输入 name\_list.按下 TAB 键，ipython 会提示 **列表**的 **方法**

In [1]:name\_list. name\_list.remove

name\_list.append name\_list.count name\_list.insert

name\_list.clear name\_list.extend name\_list.pop

name\_list.sort name\_list.copy name\_list.index

序号 分类 关键字/函数/方法 说明

1 增加 列表.insert(索引，数据) 在指定位置插入数据

列表.append(数据) 在末尾追加数据

列表.extend(列表2) 将列表2数据追加到列表

2 修改 列表.[索引]=数据 修改指定索引的数据

3 删除 del 列表[索引] 删除指定索引的数据

列表.remove[数据] 删除第一个出现指定数据

列表.pop 删除末尾数据

列表.pop(索引) 删除指定索引数据

列表.clear 清空列表

4 统计 len(列表) 列表长度

列表.count(数据) 数据在列表中出现的次数

5 排序 列表.sort() 升序排序

列表.sort(reverse=True) 降序排序

列表.reverse() 逆序、反转

·使用**del**关键字（delete）同样可以删除列表中元素

·**del**关键字本质上是用来 **将一个变量从内存中删除的**

·如果使用del关键字变量从内存中删除，后续的代码就不能再使用这个变量了

※在日常开发中，要从列表删除数据，建议 **使用列表提供的方法**

关键字、函数和方法：

**·关键字** 是Python内置的，具有特殊意义的标识符

**In [1]:import keyword**

**In [2]:print(keyword.kwlist)**

**In [3]:print(len(keyword.kwlist))**

**※关键字后面不需要使用括号**

·**函数** 封装独立功能，可以直接调用

**函数名（参数）**

**※函数需要死记硬背**

**·方法** 和函数类似，同样是封装了独立的功能

**·方法** 需要通过 **对象** 来调用，表示针对这个 **对象**要做的操作

**对象.方法名（参数）**

**※在变量后面输入 .然后选择针对这个变量要执行的操作，记忆起来比函数要简单很多**

***循环遍历：***

·**遍历** 就是 **从头到尾依次** 从 **列表** 中获取数据

·在 **循环体内部** 针对 **每一个元素**，执行相同的操作

·在 Python中为了提高列表的遍历效率，专门提供的 **迭代遍历**

·使用 for 就能够实现迭代遍历

# for 循环内部使用的变量 in 列表

for name in name\_list():

print(name) #循环内部针对列表元素进行操作

·尽管Python的 列表 中可以 存储不同类型的数据

·但是在开发中，更多的应用场景是

1、**列表** 存储相同类型的数据

2、通过 **迭代遍历**，在循环体内部，针对列表中的每一项元素，执行相同的操作

***元组(Tuple):***

·Tuple(元组)与列表类似，不同之处在于元组的 **元素不能修改**

**·元组** 表示多个元素组成的序列

**·元组** 在Python开发中，有特定的应用场景

·用于存储 **一串 信息**，**数据** 之间使用 **,** 分隔

·元组用 ( ) 定义

·元组的 **索引** 从 0 开始

·索引 就是数据在 元组 中的位置编号

info\_tuple = (“zhangsan”, 18, 1.75)

**创建空元组：**

tuple = ()

**元组中 只包含一个元素** 时，需要 **在元素后面添加逗号**

info\_tulpe = (50, )

**元组常用操作：**

·在ipython3中定义一个 元组，例如：info = ()

·输入info. 按下 TAB 键，ipython会提示 元组 能够使用的函数如下：

info.count info.index

**循环遍历：**

·**取值** 就是从 **元组** 中获取存储在指定位置的数据

·**遍历** 就是 **从头到尾 依次** 从 **元组** 中获取数据

# for 循环内部使用的变量 in 元组

for item in info:

print(item)

·在Python中，可以使用 for 循环遍历所有非数字型类型的变量：**列表、元组、字典** 以及 **字符串**

·在实际开发中，除非 **能够确认元组中的数据类型**，否则针对元组的循环遍历需求并不是很多

**应用场景：**

·**函数的 参数** 和 **返回值**，一个函数可以接收 **任意多个参数** 或者 **依次返回多个数据**

·格式化字符串，格式化字符串后的( )本质上就是一个元组

·让 **列表不可以被修改**，以保护数据安全

info = (“zhangsan”, 18)

print(“%s 的年龄是 %d” % info)

**元组和列表之间的转换：**

·使用 list 函数可以把元组转换成列表

list(元组)

·使用tuple函数可以把列表转换成元组

tuple(列表)

***字典(dictionary)：***

**字典的定义：**

·dictionary(字典)是 **除列表以外** Python之中 **最灵活** 的数据类型

·字典同样可以用来 **存储多个数据**

·通常用于存储 **描述一个 物体 的相关信息**

·和列表的区别

·**列表** 是 **有序** 的对象集合

·**字典** 是 **无序** 的对象集合

·字典用**{ }**定义

·字典使用 **键值对** 存储数据，键值对之间使用 **,** 分隔

**·键** key 是索引

**·值** value 是数据

**·键** 和 **值** 之间使用 **:** 分隔

**·键必须是唯一的**

**·值** 可以取任何数据类型，但 **键** 只能使用 **字符串、数 字或 元组**

xiaoming = {“name”: “小明”,

“age”: 18

“gender”: True

“height”: 1.75}

**字典常用操作：**

·在 ipython3中定义一个 字典，例如xiaoming = {}

·输入xiaoming.按下TAB键，ipython会提示 字典 能够使用的函数如下：

In [1]: xiaoming.

xiaoming.clear xiaoming.items xiaoming.setdefault

xiaoming.copy xiaoming.keys xiaoming.update

xiaoming.fromkeys xiaoming.pop xiaoming.values

xiaoming.get xiaoming.popitem

**循环遍历：**

·**遍历** 就是 **依次** 从 **字典** 中获取所有键值对

for 循环内部使用的’ key 的变量’ in 字典

for k in xiaoming:

print(“%s ：%s” % (k, xiaoming[k]))

※实际开发中，针对字典的循环遍历需求并不多

**应用场景：**

·使用 **多个键值对**，存储 **描述一个 物体 的相关信息**

·将 **多个字典** 放在 **一个列表** 中，再进行遍历，在循环体内部针对每一个字典进行 **相同的处理**

***字符串的定义：***

·**字符串** 就是 **一串字符**，是编程语言中表示文本的数据类型

·在Python中可以使用 **一对双引号 “** 或者 **一对单引号 ‘** 定义一个字符串

·虽然可以使用 \” 或者 \’ 做字符串转义，但实际中：

·如果字符串内部需要使用 “ ，可以使用 ‘ 定义字符串

·如果字符串内部需要使用 ‘ ，可以使用 “ 定义字符串

·可以使用 **索引** 获取一个字符串中 **指定位置的字符**，索引计数从**0**开始

·也可以使用 for **循环遍历** 字符串中每一个字符

**※大多数编程语言都是用 “ 来定义字符串**

String = “Hello Python”

for c in string:

print(c)

***字符串的常用操作：***

·在ipython3中定义一个 **字符串**，例如：hello\_str = “”

·输入hello\_str.按下TAB键，ipython会提示 **字符串** 能够使用的所有方法

※正式因为Python内置提供的方法足够多，才使得在开发时，能够针对字符串进行更加灵活的操作！应对更多的开发需求！

1、判断类型 -- 9

string.isspace() 如果string中只包含空格，则返回True

string.isalnum() 如果string至少有一个字符并且所有字符都是字母或者数字则返回True

string.isalpha() 如果string至少有一个字符并且所有字符都是字母则返回True

string.isdecimal() 如果string只包含数字则返回True，全角数字

string.isdigit() 如果string只包含数字则返回True，全角数字、⑴、\u00b2

string.isnumeric() 如果string只包含数字则返回True，全角数字，汉字数字

string.istitle() 如果string是标题化的（每个单词的首字符大写）则返回True

string.islower() 如果string中包含至少一个区分大小写的字符，并且所有这些（区分大小写的）字符都是小写，则返回True

string.isupper() 如果string中包含至少一个区分大小写的字符，并且所有这些（区分大小写的）字符都是大写，则返回True

2、查找和替换 -- 7

string.startswith(str) 检查字符串是否是以str开头，是则返回True

string.endswith(str) 检查字符串是否是以str结束，是则返回True

string.find(str,start=0,end=len(string)) 检测str是否包含在string中，如果start和end指定范围，则检查是否包含在指定范围内，如果是 返回开始的索引值，否则返回 -1

string.rfind(str,start=0,end=len(string)) 类似于find()函数，不过是从右边开始查找

string.index(str,start=0,end=len(string)) 跟find()方法类似，只不过如果str不在string会报错

string.rindex(str,start=0,end=len(string)) 类似于index()，不过是从右边开始

string.replace(old\_str,new\_str,num=string.count(old)) 把string中的old\_str替换成new\_str，如果num指定，则替换不超过num次

3、大小写转换 -- 5

string.capitalize() 把字符串的第一个字符大写

string.title() 把字符串的每个单词首字母大写

string.lower() 转换string中所有大写字符为小写

string.upper() 转换string中的小写字母为大写

string.swapcase() 翻转string中的大小写

4、文本对齐 -- 3

string.ljust(width) 返回一个原字符串左对齐，并使用空格填充至长度width的新字符串

string.rjust(width) 返回一个原字符串右对齐，并使用空格填充至长度width的新字符串

string.center(width) 返回一个原字符串居中，并使用空格填充至长度width的新字符串

5、去除空白字符 -- 3

string.lstrip() 截掉string左边（开始）的空白字符

string.rstrip() 截掉string右边（结束）的空白字符

string.strip() 截掉string左右两边的空白字符

6、拆分和连接 -- 5

string.partition(str) 把字符串string分成一个3元素的元组（str前面，str，str后面）

string.rpartition(str) 类似于partition()函数，不过是从右边开始查找

string.split(str=””,num) 以str为分隔符切片string，如果num有指定值，则仅分离num+1个子字符串，str默认包含’\r’,’\t’,’\n’和空格

string.splitlines() 按照行（’\r’,’\n’,’\r\n’）分隔，返回一个包含各行作为元素的列表

string.join(seq) 以string作为分隔符，将seq中所有的元素（的字符串表示）合并为一个新的字符串

***字符串的切片：***

·**切片** 方法适用于 **字符串、列表、元组**

·**切片** 使用 **索引值** 来限定范围，从一个大的 **字符串** 中 **切出** 小的 **字符串**

·**列表** 和 **元组** 都是 **有序** 的集合，都能够 **通过索引值** 获取到对应的数据

·**字典** 是一个 **无序** 的集合，是使用 **键值对** 保存数据

**字符串[开始索引:结束索引:步长]**

1、指定的区间属于 **左闭右开** 型 **[开始索引，结束索引） =>开始索引 >= 范围 < 结束索引**

·从 **起始** 位置开始，到 **结束 位的前一位** 结束（**不包含结束体本身**）

2、从头开始，**开始索引 数字可以省略，冒号不能省略**

3、到末尾结束，**结束索引 数字可以省略，冒号不能省略**

4、步长默认为1，如果连续切片，**数字和冒号都可以省略**

***公共方法：***

**Python内置函数：**

函数 描述 备注

len(item) 计算容器中元素个数

del(item) 删除变量 del有两种方式

max(item) 返回容器中元素最大值 字典只针对key比较

min(item) 返回容器中元素最小值 字典只针对key比较

※**字符串** 比较符合以下规则： “0”<“A”<“a”

**切片：**

表达式 结果 支持的数据类型

**“0123456789”[::-2] “97531” 字符串、列表、元组**

·**切片** 使用 **索引值** 来限定范围，从一个大的 **字符串** 中 **切出** 小的 **字符串**

·**列表** 和 **元组** 都是 **有序** 的集合，都能够 **通过索引值** 获取到对应数据

·**字典** 是一个 **无序** 的集合，使用 **键值对** 保存数据

**运算符：**

运算符 表达式 结果 描述

+ [1,2]+[3,4] [1,2,3,4] 合并

\* [“HI!”] \* 2 [“HI!”,”HI!”]重复

in 3 in (1,2,3) True 元素是否存在

not in 4 not in (1,2,3) Ture 元素是否不存在

>>= == <<= (1,2,3) < (2,2,3) True 元素比较

·in 在对 **字典** 操作时，判断的是 **字典的键**

·in 和 not in被称为 **成员运算符**

***完整的for循环语法：***

·在Python中完整的**for循环** 的语法为：

for 变量 in 集合:

循环体代码

else:

没有通过break退出循环，循环结束后，会执行的代码

应用场景：

·在 **迭代遍历** 嵌套的数据类型时，例如 一个列表包含了多个字典

·需求：要判断某一个字典中是否存在指定的值

·如果 **存在**，提示并退出循环

·如果 **不存在**，在 **循环整体结束后**，希望 得**到一个统一的提示**

**名片管理系统**

***系统需求：***

1、程序启动，显示名片管理系统欢迎界面，并显示功能菜单

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

欢迎使用【名片管理系统】 V1.0

1. 新建名片
2. 显示全部
3. 查询名片
4. 退出系统

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2、用户用数字选择不同的功能

3、根据功能选择，执行不同的功能

4、用户名片需要记录用户的 **姓名、电话、QQ、邮件**

5、如果查询到指定的名片，用户可以选择 **修改** 或者 **删除** 名片

**步骤：**

1、框架搭建

2、新增名片

3、显示所有名片

4、查询名片

5、查询成功后修改、删除名片

6、让 Python程序能够直接运行

***框架搭建：***

·搭建名片管理系统 **框架结构**

1.准备文件，确定文件名，保证能够 **在需要的位置** 编写代码

2.编写 **主运行循环**，实现基本的 **用户输入和判断**

**文件准备：**

1.新建 card\_main.py 保存 **主程序功能代码**

·程序的入口

·每一次启动名片管理系统都要通过main这个文件启动

2.新建 card\_tools.py 保存 **所有名片功能函数**

·将对名片的 **新增、查询、修改、删除** 等功能封装在不同的函数中

**编写主运行循环：**

·在 card\_main中添加一个无限循环

while True：

# TODO(小明) 显示系统菜单

action = input(“请选择操作功能：”)

print(“你选择的操作是：%s” % action)

# 根据用户输入决定后续的操作

if action in [“1”, “2”, “3”]:

pass

elif action == “0”:

print(“欢迎再次使用【名片管理系统】”)

break

else:

print(“输入错误，请重新输入！”)

**·字符串判断**

if action in [“1”, “2”, “3”]:

if action == “1” or action == “2” or action == “3”:

1.使用in针对 **列表** 判断，避免使用 or 拼接复杂的逻辑条件

2.没有使用 int 转换用户输入，可以避免 **一旦用户输入的不是数字**，导致程序运行出错

**pass：**

·pass就是一个空语句，不做任何事情，一般用作占位

·是为了保持程序结果的完整性

**无限循环：**

·在开发软件时，如果 **不希望程序执行后** 立即退出

·可以在程序中增加一个 **无限循环**

·**由用户来决定** 退出程序的时机

**TODO注释：**

·在 **#** 后跟上 **TODO**，用于标记需要去做的工作

# TODO(作者/邮件) 显示系统菜单

***保存名片数据的结构：***

**程序就是用来处理数据的，而变量就是用来存储数据的**

·使用 **字典** 记录 **每一张名片** 的详细信息

·使用 **列表** 统一记录所有的 **名片字典**

**定义名片列表变量：**

·在card\_tools文件的顶部增加一个 **列表变量**

# 所有名片记录的列表

card\_list = []

1.**所有名片相关操作**，都需要使用这个列表，所以应该 **定义在程序的顶部**

2.**程序刚运行时，没有数据**，所以是 **空列表**

***LINUX上的Shebang符号（#！）：***

·#！这个符号叫做Shebang说着Sha-bang

·Shebang通常在Unix系统脚本中 **第一行开头** 使用

·指明 **执行这个脚本文件** 的 **解释程序**

**使用Shebang的步骤：**

·1、使用 which 查询 python3 解释器所在路径

$ which pychon3

·2、修改要运行的 主Python文件，在第一行增加以下内容

$ #! /usr/bin/python3

·3、修改 主python文件 的文件权限，增加执行权限

$ chmod +x cards\_main.py

·4、在需要时执行程序即可

$ ./cards\_main.py

**变量进阶**

***变量的引用：***

·**变量** 和 **数据** 都是保存在 **内存** 中的

·在 Python 中 **函数** 和 **参数传递** 以及 **返回值** 都是靠 **引用** 传递的

**引用的概念：**

·**变量** 和 **数据** 是分开存储的

·**数据** 保存在内存中的一个位置

·**变量** 中保存着数据在内存中的地址

·**变量** 中 **记录数据的地址**，就叫做 **引用**

·使用 **id()** 函数可以参看变量中保存数据所在的 **内存地址**

※如果变量已经被定义，当给一个变量赋值的时候，本质上是 **修改了数据的引用**

·变量 **不再** 对之前的数据引用

·变量 **改为** 对新赋值的数据引用

·变量的名字类似于 **标签纸** 贴在 **数据** 上

**函数的参数和返回值的传递：**

在Python中，函数的 **实参/返回值** 都是靠 **引用** 传递来的

***可变和不可变类型：***

**·不可变类型**，内存中的数据不允许被修改：

·数字类型：int,bool,float,complex,long(2,x)

·字符串 str

·元组 tuple

**·可变类型**，内存中的数据可以被修改：

·列表 list

·字典 dict

**※字典的key只能使用不可变类型的数据**

**注意**：

1、**可变类型**的数据变化，是通过 **方法** 来实现的

2、如果给一个可变类型的变量，赋值了一个新的数据，**引用会修改**

·变量 **不再** 对之前的数据引用

·变量 **改为** 对新赋值的数据引用

**哈希（hash）：**

·Python中内置有一个名字叫做**hash(o)**的函数

·接收一个 **不可变类型** 的数据作为 **参数**

·返回结果是一个 **整数**

·**哈希** 是一种 **算法**，其作用就是提取数据的 **特征码(指纹)**

·**相同的内容** 得到 **相同的结果**

·**不同的内容** 得到 **不同的结果**

·在Python中，设置字典的 **键值对** 时，会首先对**key**进行**hash** 已决定如何在内存中保存字典的数据，以方便 **后续** 对字典的操作：**增、删、改、查**

·键值对的key必须是不可变类型数据

·键值对的value可以是任意类型的数据

***局部变量和全局变量：***

·**局部变量** 是在 **函数内部** 定义的变量，**只能在函数内部使用**

·**全局变量** 是在 **函数外部定义** 的变量（没有定义在某一个函数）内，**所有函数** 内部 **都可以使用这个变量**

※在其他的开发语言中，大多 **不推荐使用全局变量** ——可变范围太大，导致程序不好维护！

***局部变量：***

·**局部变量** 是在 **函数内部** 定义的变量，**只能在函数内部使用**

·函数执行结束后，**函数内部的局部变量**，**会被系统回收**

·不同的函数，可以定义相同的名字的局部变量，但是各用各的不会产生影响

**局部变量的作用：**

·在函数内部使用，临时 保存 **函数内部需要使用的数据**

**局部变量的什么周期：**

·所谓 **生命周期** 就是变量从 **被创建** 到 **被系统回收**的过程

·**局部变量** 在 **函数执行时** 才会被创建

·**函数执行结束后** 局部变量 **被系统回收**

·**局部变量在生命周期** 内，可以用来存储 **函数内部临时使用到的数据**

***全局变量：***

·**全局变量** 是在 **函数外部定义** 的变量，所有函数内部都可以使用这个变量

※函数执行时，**需要处理变量时** 会：

1、**首先** 查找 **函数内部** 是否存在 **指定名称 的局部变量，如果有，直接使用**

2、如果没有，查找 **函数外部** 是否存在 **指定名称 的全局变量，如果有，直接使用**

3、如果还没有，程序报错！

**1>函数不能直接修改 全局变量的引用**

·**全局变量** 是在 **函数外部定义** 的变量（没有定义在某一个函数内），**所有函数** 内部 **都可以使用这个变量**

·在函数内部，可以 **通过全局变量的引用获取对应的数据**

·但是，**不允许直接修改全局变量的引用**——使用赋值语句修改全局变量的值

**2>在函数内部修改全局变量的值**

·如果在函数中需要修改全局变量，需要使用**global**声明

**3>全局变量定义的位置**

·为了保证所有的函数都能够正确使用到全局变量，应该 **将全局变量定义在其他函数的上方**

**4>全局变量命名的建议**

·为了避免 **局部变量和全局变量**出现混淆，在定义全局变量时，有些公司会有一些开发要求

·全局变量名前应该增加 **g\_** 或者**gl\_** 的前缀

**※具体的要求格式，各公司要求可能会有些差异**

***※代码结构示意图***

**·shebang**

**·import 模块**

**·全局变量**

**·函数定义**

**·执行代码**

***函数参数和返回值的作用：***

函数根据 **有没有参数** 以及 **有没有返回值**，可以 **相互组合**，一共有 **4种** 组合形式

1、无参数，无返回值

2、无参数，有返回值

3、有参数，无返回值

4、有参数，有返回值

※定义函数时，**是否接收参数，或者是否返回结果**，是根据 **实际的功能需求** 来决定的

1、如果函数 **内部处理的数据不确定**，就可以将外界的数据以参数传递到函数内部

2、如果希望一个函数 **执行完成后，向外界汇报执行结果**，就可以增加函数的返回值

***函数的返回值 进阶：***

·在程序开发中，有时候，会希望 **一个函数执行结束后，告诉调用者一个结果**，以便调用者针对具体的结果做后续的处理

·**返回值** 是函数 **完成工作后**，**最后** 给调用者的 **一个结果**

·在函数中使用**return**关键字可以返回结果

·调用函数一方，可以 **使用变量** 来 **接收** 函数的返回值

***函数的参数 进阶：***

**不可变和可变的参数：**

·无论传递的参数是 **可变** 还是 **不可变**

·只要 **针对参数** 使用 **赋值语句**，会在 **函数内部** 修改 **局部变量的引用**，**不会影响到 外部变量的引用**

·如果传递的参数是 **可变类型**，在函数内部，使用 **方法** 修改了数据的内容，**同样会影响到外部的数据**

·在Python中，列表变量调用 **+=** 本质上是在执行列表变量的**extend**方法，不会修改变量的引用(**会影响到外部的数据**)

***缺省参数：***

·定义函数时，可以给 **某个参数** 指定一个 **默认值**，具有默认值的参数就叫做 **缺省参数**

·调用函数时，如果没有传入 **缺省参数** 的值，则在函数内部使用定义函数时指定的 **参数默认值**

·函数的缺省参数，**将常见的值设置为参数的缺省值**，从而 **简化函数的调用**

**指定函数的缺省参数：**

·在参数后使用赋值语句，可以指定参数的缺省值

1、缺省参数，需要使用 **最常见的值** 作为默认值！

2、如果一个参数的值 **不能确定**，则不应该设置默认值，具体的数值在调用函数时，由外界传递！

**缺省参数的注意事项：**

**1>缺省参数的定义位置**

**·必须保证 带有默认值的缺省参数 在参数列表末尾**

**2>调用带有多个缺省参数的函数**

·在 **调用函数时**，如果有 **多个缺省参数，需要制定参数名，**这样解释器才能够知道参数的对应关系！

***多值参数：***

**定义支持多值参数的函数**

·有时可能需要 **一个函数** 能够处理的参数 **个数** 是不确定的，这个时候，就可以使用 **多值参数**

·Python中有 **两种** 多值参数

·参数名前增加 **一个 \*** 可以接收 **元组**

·参数名前增加 **两个 \*** 可以接收 **字典**

·一般在给多值参数命名时，**习惯** 使用以下两个名字

·**\*args** —— 存放 **元组** 参数，前面有**一个 \***

·**\*\*kwargs** —— 存放 **字典** 参数，前面有**两个 \***

·args是arguments 的缩写，有变量的含义

·kw是keyword的缩写，kwargs 可以记忆 **键值对参数**

※**多值参数** 的应用会经常出现在网络上一些大牛开发的框架中，知道多值参数，**有利于我们能够读懂大牛的代码**

***元组和字典的拆包：***

·在调用带有多值参数的函数时，如果希望：

·将一个 **元组变量**，直接传递给 args

·将一个 **字典变量**，直接传递给kwargs

·就可以使用 **拆包**，简化参数的传递，**拆包** 的方式是：

·在 **元组变量前**，增加 **一个 \***

·在 **字典变量前**，增加 **两个 \***

***函数的递归：***

※**函数调用自身的 编程技巧 称为递归**

**递归函数的特点：**

·**一个函数 内部 调用自己**

·函数内部可以调用其他函数，在函数内部也可以调用自己

·**代码特点**

1、函数内部的 **代码** 是相同的，只是针对 **参数** 不同，**处理的结果不同**

2、当 **参数满足一个条件** 时，函数不再执行

**·这个非常重要**，通常被称为递归的出口，否则 **会出现死循环**

※递归是一个 **编程技巧**，在处理 **不确定的循环条件时**，格外有用，例如：**遍历整个文件目录的结构**

**面向对象（OOP）基本概念**

***面向对象基本概念：***

·我们之前学习的编程方式就是 **面向过程** 的

·**面向过程** 和 **面向对象**，是两种不同的 **编程方式**

·对比 **面向过程**的特点，可以更好地了解什么是 **面向对象**

**过程和函数（科普）：**

·**过程** 是早期的一个变成概念

·**过程** 类似于函数，只能执行，但是没有返回值

·**函数** 不仅能执行，还可以返回结果

**面向过程** 和 **面向对象 基本概念：**

**1)面向过程——怎么做？**

1.把完成某一个需求的 **所有步骤 总头到尾** 逐步实现

2.根据开发需求，将某些 **功能独立** 的代码 **封装** 成一个又一 个 **函数**

3.最后完成的代码，就是顺序地调用 **不同的函数**

**特点：**

1.注重 **步骤与过程**，不注重职责分工

2.如果需求复杂，代码会变得很复杂

3.**开发复杂项目，没有固定的套路，开发难度很大！**

2）**面向对象——谁来做？**

**※**相比较函数，**面向对象** 是 **更大** 的 **封装**，根据 **职责** 在 **一 个对象中封装 多个方法**

1.在完成某一个需求前，首先确定**职责——要做的事情(方法)**

2.根据 **职责** 确定不同的 **对象**，在 **对象** 内部封装不同的 **方 法（多个）**

3.最后完成的代码，就是顺序地让 **不同的对象** 调用 **不同的 方法**

**特点：**

1.注重 **对象和职责**，不同的对象承担不同的职责

2.更加适合应对复杂的需求变化，**是专门应对复杂项目开发， 提供的固定套路**

3.**需要在面向过程基础上，再学习一些面向对象的语法**

***类和对象：***

**类和对象的基本概念：**

**·类** 和 **对象** 是 **面向对象编程的 两个 核心概念**

**类：**

·**类**是对一群具有**相同 特征** 或者 **行为** 的事物的一个统称， 是抽象的，**不能直接使用**

·**特征** 被称为 **属性**

·**行为** 被成为 **方法**

·**类** 就相当于制造飞机时的**图纸**，是一个 **模板**，是 **负责创 建对象的**

**对象：**

·**对象** 是 **由类创建出来的一个具体存在**，可以直接使用

·由 **哪一个类** 创建出来的 **对象**，就拥有在**哪一个类** 中的

**·属性**

**·方法**

·**对象** 就相当于用 **图纸 制造** 的飞机

**※**在程序开发中，应该 **先有类，再有对象**

**类和对象的关系：**

·**类是模板，对象** 是根据 **类** 这个模板创建出来的，应该 **先 有类，再有对象**

·**类** 只有一个，而 **对象** 可以有很多个

·**不同的对象** 之间 **属性** 可能会各不相同

·**类** 中定义了什么 **属性和方法**，**对象** 中就有什么属性和方 法，**不可能多，也不可能少**

**类的设计：**

在程序开发中，要设计一个类，通常需要满足三个要素：

1.**类名** 这类事物的名字，满足大驼峰命名法

2.**属性** 这类事物具有什么样的特征

3.**方法** 这类事物具有什么样的行为

**大驼峰命名法：**

1.每一个单词的首字母大写

2.单词与单词之间没有下划线

**类名的确定：**

**名词提炼法** 分析 **整个业务流程**，出现的 **名词**，通常就是 找到的类

**属性和方法的确定：**

·对 **对象的特征描述**，通常可以定义成 **属性**

·**对象具有的行为（动词）**，通常可以定义成 **方法**

**※**需求中没有涉及的属性或方法在设计类时，不需要考虑

***面向对象基础语法：***

**dir内置函数（知道）：**

·在 Python中，**对象几乎是无所不在的**，我们之前学习的 **变 量、数据、函数 都是对象**

1.在 **标识符/数据** 后输入一个 ”.”，然后按下**TAB键** ipython会提示该对象能够调用的 **方法列表**

2.使用内置函数dir传入 **标识符/数据**，可以查看对象内 的 **所有属性及方法**

**※**\_\_方法名\_\_格式的方法是 Python提供的 **内置方法/属性**

**方法名 作用**

**\_\_new\_\_** **创建对象**时，会被 **自动** 调用

**\_\_init\_\_** **对象被初始化**时，会被 **自动** 调用

**\_\_del\_\_** **对象被从内存中销毁**前，会被 **自动** 调用

**\_\_str\_\_** 返回**对象的描述信息**，print函数输出使用

**定义简单的类（只包含方法）：**

※面向**对象** 时 **更大** 的 **封装**，在 **一个类中封装 多个方法**， 这样**通过这个类创建出来的对象**，**就可以直接调用这些方法了**

**定义只包含方法的类：**

·在Python中要定义一个只包含方法的类，语法格式如下：

class类名：

def 方法1(self,参数列表)

pass

def 方法2(self,参数列表)

pass

·**方法** 的定义格式和之前学习过的 **函数** 几乎一样

·区别在于第一个参数必须是self

**※类名** 的 **命名规则** 要符合 **大驼峰命名法**

**创建对象：**

·当一个类定义完成后，要使用这个类来创建对象，语法格式

对象变量 =类名（）

**引用概念的强调：**

※在面向对象发开中，引用的概念是同样适用的！

·在 Python中使用类 **创建对象之后**，tom**变量中** 仍然记录 的是**对象在内存中的地址**

·也就是tom**变量引用** 了 **新建的猫对象**

·使用print输出 **对象变量**，默认情况下，是能够输出这个 变量 **引用的对象** 是 **由哪一个类创建的对象**，以及 **在内存中 的地址（十六进制表示）**

※在计算机中，通常使用 **十六进制** 表示 **内存地址**

·十进制 和 十六进制 都是用来表达数字的，只是表示的方 式不一样

·十进制 和 十六进制 的数字之间可以来回转换

·**%d**可以以 **10进制** 输出数字

·**%x**可以以 **16进制** 输出数字

**方法中的self参数：**

**给对象增加参数：**

·在Python中，要 **给对象设置属性**，非常容易，**但是不推荐 使用**

·因为：对象属性的封装应该封装在类的内部

·只需要在**类的外部的代码**中直接通过 **.**设置一个属性即可

※这种方法虽然简单，但是不推荐使用！

tom.name = “Tom”

…

lazy\_cat.name = “大懒猫”

**使用self在方法内部输出每一只猫的名字：**

※由 哪一个对象 调用的方法，方法内的self就是 哪一个对 象的引用

·在类封装的方法内部，self就表示**当前调用方法是自己**

·**调用方法时**，程序要不需要传递self参数

·**在方法内部**

·可以通过self，**访问对象的属性**

·也可以通过self，**调用其他的对象方法**

class Cat:

def eat(self):

print(“%s 爱吃鱼” % self.name)

tom = Cat( )

tom.name = (“Tom”)

tom.eat( )

lazy\_cat = Cat( )

lazy\_cat.name = “大懒猫”

lazy\_cat.eat( )

·在 **类的外部**，通过 **变量名.** 访问对象的 **属性和方法**

·在**类封装的方法中**，通过**self.** 访问对象的**属性和方法**

**初始化方法：**

·在日常开发中，不推荐在 **类的外部** 给对象增加属性

·如果在运行时，**没有找到属性，程序会报错**

·对象应该包含有哪些属性，应该**封装在类的内部**

**初始化方法：**

·当使用 类名()创建对象时，会 自动 执行操作：

1.为对象在内存中 **分配空间**——创建对象

2.为对象的属性 **设置初始值**——初始化方法(init)

·这个 **初始化方法** 就是\_\_init\_\_方法，\_\_init\_\_是对象 的 **内置方法**

※\_\_init\_\_方法是 **专门** 用来定义一个类 **具有那些属性 的方法！**

**在初始化方法内部定义属性：**

·在\_\_init\_\_方法内部使用 self.属性名 =属性的初始值 就可以 **定义属性**

·定义属性之后，再使用Cat类创建的对象，都会拥有该 属性

class Cat:

def \_\_init\_\_(self):

print(“这是一个初始化方法”)

self.name = “Tom”

def eat(self)：

print(“% s 爱吃鱼”% self.name)

tom = Cat( ):

tom.eat( )

**初始化的同时设置初始值：**

·在开发中，如果希望在 **创建对象的同时，就设置对象的 属性，**可以对\_\_init\_\_方法进行**改造**

1.把希望设置的属性值，定义成 \_\_init\_\_ 方法的参数

2.在方法内部使用self.属性 =形参 接受外部传递的 参数

3.在创建对象时，使用 类名（属性1，属性2…）调用

class Cat:

def \_\_init\_\_(self,name):

print(“初始化方法 %s”%name)

self.name = name

…

tom = Cat(“Tom”)

…

lazy\_cat = Cat(“大懒猫”)

…

**内置方法和属性：**

**方法名 作用**

**\_\_del\_\_** **对象被从内存中销毁**前，会被 **自动** 调用

**\_\_str\_\_** 返回**对象的描述信息**，print函数输出使用

**\_\_del\_\_方法（知道）：**

·当使用 类名()创建对象时，为对象 **分配完空间后**，**自 动** 调用 \_\_del\_\_ 方法

·当一个 **对象被从内存中销毁** 前，会 **自动** 调用 \_\_del\_\_ 方法

**·应用场景**

·\_\_init\_\_改造初始化方法，可以让创建对象更加灵活

·\_\_del\_\_如果希望在对象被销毁前，在做一些事情， 可以考虑一下\_\_del\_\_方法

**·生命周期**

·一个对象从调用 类名（）创建，生命周期开始

·一个对象的\_\_del\_\_方法一旦被调用，生命周期结束

·在对象的生命周期内，可以访问对象属性，或者让对 象调用方法

**\_\_str\_\_方法：**

·在Python中，使用print输出 **对象变量**，默认情况下 会输出这个变量 **引用的对象** 是 **由哪一个类创建的对象**， 以及 **在内存中的地址（十六进制表示）**

·如果在开发中，希望使用print输出 **对象变量** 时，能 够打印机 自定义的内容，就可以利用\_\_str\_\_这个方法

**※**\_\_str\_\_方法必须返回一个字符串

class Cat:

def \_\_init\_\_(self,new\_name):

self.name = new\_name

print(“%s 来了” % self.name)

def \_\_del\_\_(self):

print(“%s 去了” % self.name)

def \_\_str\_\_(self):

return “我是小猫 %s” % self.name

tom = Cat( )

print(tom)

**面向对象 封装**

***封装：***

1、**封装** 是面向对象变成的一大特点

2、面向对象编程的 **第一步** ——将 **属性** 和 **方法 封装** 到一个抽象的 **类** 中

3、**外界** 使用 **类** 创建 **对象**，然后 **让对象调用方法**

4、**对象方法的细节** 都被 **封装** 在 **类的内部**

**※**一个对象的 **属性** 可以是 **另外一个类创建的对象**

***定义没有初始值的属性：***

在定义属性时，如果 **不知道设置什么初始值**，可以设置为 **None**

·**None关键字** 表示 **什么都没有**

·表示一个 **空对象**，**没有方法和属性**，**是一个特殊的常量**

·可以将 **None**赋值给任何一个变量

***身份运算符：***

身份运算符用于 **比较** 两个对象的 **内存地址** 是否一致 —— **是否是对同一个对象的引用**

·在 Python中针对 None比较时，建议使用 is判断

**is**：is是判断两个标识符是不是引用同一个对象，xisy，类似id(x) == id(y)

**is not**：is not是判断两个标识符是不是引用不同对象，x is not y,类似 id(x) != id(y)

**is 与 ==区别：**

**is**用于判断 **两个变量 引用对象是否为同一个**

==用于判断 **引用变量的值** 是否相等

>>> a = [1, 2, 3]

>>> b = [1, 2, 3]

>>> b is a

false

>>> b == a

True

**私有属性和私有方法**

***应用场景及定义方式：***

**应用场景：**

·在实际开发中，**对象** 的 **某些属性或方法** 可能只希望 **在对象的内部被使用**，而 **不希望在外部被访问到**

·**私有属性** 就是 **对象** 不希望公开的 **属性**

·**私有方法** 就是 **对象** 不希望公开的 **方法**

**定义方式：**

·在 **定义属性或方法时**，在 **属性名或者方法名前** 增加 **两个下划线**，定义的就是 **私有** 属性或方法

***伪私有属性和方法（科普）：***

※在日常开发中，**不要使用这种方式**，访问对象的 **私有属性** 或**私有方法**

Python中，并没有 **真正意义** 的 **私有**

·在给 **属性、方法** 命名时，实际时对 **名称** 做了一些特殊处理，使得外界无法访问到

·**处理方式：**在 **名称** 前面加上 **\_类名 => \_类名\_\_名称**

**继承**

***面向对象三大特征：***

1、**封装** 根据 **职责** 将 **属性** 和 **方法 封装** 到一个抽象的 **类** 中

2、**继承实现代码的重用**，相同的代码不需要重复的编写

3、**多态** 不同的对象调用相同的方法，产生不同的执行结果，**增加代码的灵活度**

***单继承：***

**继承的概念：子类** 拥有 **父类** 的所有 **方法** 和 **属性**

**继承的语法：**

**class 类名(父类名)：**

**pass**

· **子类** 继承自 **父类**，可以直接 **享受** 父类中已经封装好的方法，不需要再次开发

·**子类** 中应该根据 **职责**，封装 **子类特有的 属性和方法**

**专业术语：**

·Dog类时Animal类的**子类**，Animal类是Dog类的**父类**，Dog类从Animal类**继承**

·Dog类是Animal类的**派生类**，Animal类是Dog类的**基类**，Dog类从Animal类**派生**

**继承的传递性：**

·C 类从 B类继承，B类又从 A类继承

·那么 C 类就具有 B类和 A 类的所有属性和方法

**子类** 拥有 **父类** 以及 **父类的父类** 中封装的所有 **属性** 和 **方法**

***方法的重写：***

·**子类** 拥有 **父类** 的所有 **方法** 和 **属性**

· **子类** 继承自 **父类**，可以直接 **享受** 父类中已经封装好的方法，不需要再次开发

·当**父类**的方法实现不能满足**子类**需求时，可以对方法进行**重写**

**重写** 父类方法有两种情况：

1、**覆盖** 父类的方法

2、对父类方法进行 **扩展**

**覆盖父类的方法：**

·如果在开发中，**父类的方法实现**和**子类的方法实现**，**完全不同**

·就可以使用**覆盖**的方式，在子类中 **重新编写**父类的方法实现

※具体的实现方式，就相当于在 **子类中** 定义了一个 **和父类同名的方法并且实现**

重写之后，在运行时，**只会调用** 子类中重写的方法，而不再会调用 **父类封装的方法**

**对父类方法进行扩展：**

·如果在开发中，**子类的方法实现** 中 **包含** 父类的方法实现

·**父类原本封装的方法实现** 是 **子类方法的一部分**

·就可以使用 **扩展** 的方式

1、在子类中 **重写** 父类的方法

2、在需要的位置使用**super( ).父类方法**，来调用父类方法

3、其他的位置针对子类的需求，编写 **子类特有的代码实现**

**关于super：**

·在 Python中 super是一个 特殊的类

·super( )就是使用 super类创建出来的对象

·**最常** 使用的场景就是在 **重写父类方法时**，调用 **在父类中封装 的方法实现**

***父类的私有属性和私有方法：***

1、**子类对象不能** 在自己的方法内部，**直接**访问 父类的 **私有属性** 或 **私有方法**

2、**子类对象** 可以通过 **父类** 的 **共有方法间接** 访问到 **私有属性** 或 **私有方法**

·**私有属性、方法** 是对象的隐私，不对外公开，**外界** 以及 **子类** 都不能直接访问

·**私有属性、方法** 通常用于做一些内部的事情

***多继承：***

·**子类** 可以拥有 **多个父类**，并且具有 **所有父类** 的 **属性和方法**

**语法：**

class 子类名(父类名1，父类名2…)

pass

***多继承的使用注意事项：***

※如果 **父类之间**存在 **同名的属性或者方法**，应该 **尽量避免** 使用多继承！

**Python中的 MRO—— 方法搜索顺序（知道）：**

·Python中针对类提供一个**内置属性\_\_mro\_\_**可以查看 **方法 搜索顺序**

·MRO 主要用于 **在多继承时判断 方法、属性 的调用 路径**

print(C.\_\_mro\_\_)

·在搜索方法时，时按照\_\_mro\_\_的输出结果 **从左至右** 的顺序查 找的

·如果在当前类中 **找到方法，就直接执行，不再搜索**

·如果 **没有找到，就查找下一个类** 中是否有对应的方法，**如果 找到，就直接执行，不再搜索**

·如果找到最后一个类，还没有找到方法，程序报错

***新式类与旧式（经典）类：***

※object是Python为所有对象提供的 **基类**，提供有一些内置的属性和方法，可以使用dir函数查看

·**新式类**：以 **object**为基类的类，**推荐使用**

·**经典类**：不以**object**为基类的类，**不推荐使用**

·在Python3.x中定义类时，如果没有指定父类，会 **默认使用object**作为该类的 **基类** —— Python3.x中定义的类都是 **新式类**

· 在Python2.x中定义类时，如果没有指定父类，则不会以 object作为 **基类**

※**新式类** 和 **经典类** 在多继承时 ——**会影响到方法的搜索顺序**

**※在定义类时，如果没有父类，建议统一继承自object**

class 类名(object):

pass

**多态**

***面向对象的三大特征：***

1、**封装** 根据 **职责** 将 **属性和方法封装** 到一个抽象的 **类** 中

**·定义类的准则**

2、**继承实现代码的重用**，相同的代码不需要重复的编写

**·设计类的技巧**

**·子类针对自己特有的需求，编写特定的代码**

3、**多态** 不同的 **子类对象**，调用相同的 **父类方法**，产生不同的 结果

**·多态 可以 增加代码的灵活度**

**·以 继承 和 重写父类方法 为前提**

**·是调用方法的技巧，不会影响到类的内部设计**

**类属性和类方法**

***类的结构：***

**术语——实例：**

1、使用面向对象开发，**第1步** 是设计 **类**

2、使用 **类名()** 创建对象，**创建对象** 的动作有两步：

**·1、在内存中为对象 分配空间**

**·2、调用初始化方法 \_\_init\_\_ 为 对象初始化**

3、对象创建后，**内存** 中就有了一个对象的 **实实在在** 的存在— —实例

因此，通常也会把：

1、创建出来的 **对象** 叫做 **类** 的**实例**

2、创建对象的 **动作** 叫做 **实例化**

3、**对象的属性** 叫做 **实例属性**

4、**对象调用的方法** 叫做 **实例方法**

·**每一个对象** 都有自己 **独立的内存空间**，**保存各自不同的属性**

·**多个对象的方法，在内存中只有一份**，在调用方法时，**需要把 对象的引用** 传递到方法内部

***类是一个特殊的对象：***

**Python中 一切皆对象：**

·class AAA：定义的类属于 **类对象**

·obj1= AAA()属于 **实例对象**

·在程序运行时，**类** 同样 **会被加载到内存**

·在Python中，**类** 是一个特殊的对象 —— **类对象**

·在程序运行时，**类对象** 在内存中 **只有一份**，使用 **一个类** 可 以创建出 **很多个对象实例**

·除了封装 **实例** 的 **属性** 和 **方法**外，**类对象** 还可以拥有自己 的 **属性** 和 **方法**

**1、类属性**

**2、类方法**

·通过 **类名.** 的方式可以 **访问类的属性** 或者 **调用类的方法**

***类属性和实例属性：***

**概念和使用：**

·**类属性** 就是给 **类对象** 中定义的 **属性**

·通常用来记录 **与这个类相关** 的特征

·**类属性 不会用于** 记录 **具体对象的特征**

***属性的获取机制：***

·在Python中 **属性的获取** 存在一个 **向上查找机制**

·因此，要访问类属性有两种方式：

1、**类名.类属性**

2、**对象.类属性（不推荐）**

※如果使用 **对象.类属性 =值** 赋值语句，只会 **给对象添加一个属性**，而不会影响到 **类属性的值**

***类方法和静态方法：***

**类方法：**

**·类属性** 就是针对 **类对象** 定义的属性

·使用 **赋值语句** 在 **class**关键字下方可以定义 **类属性**

·**类属性** 用于记录 **与这个类相关** 的特征

·**类方法** 就是针对 **类对象** 定义的方法

·在 **类方法** 内部可以直接访问 **类属性**或者调用其它**类方法**

语法如下：

**@classmethod**

**def 类方法名(cls):**

**pass**

·类方法需要用 **修饰器@classmethod**来标识，**告诉解释器这是一个类方法**

·类方法的 **第一个参数** 应该是 cls

·由**哪一个类**调用的方法，方法内的cls就是**哪一个类的引用**

·这个参数和 **实例方法** 的第一个参数是 self类似

·**提示** 使用其他名称也可以，不过习惯使用cls

·通过 **类名.** 调用 **类方法**，**调用方法时**，不需要传递 cls参数

·**在方法内部：**

·可以通过 cls. **访问类的属性**

·也可以通过 cls. **调用其他的类方法**

※在类方法内部，可以直接使用 **cls**访问 **类属性** 或者**调用类方法**

***静态方法：***

·在开发时，如果需要在 **类** 中封装一个方法，这个方法：

·既 **不需要** 访问 **实例属性** 或者调用 **实例方法**

·也 **不需要**访问 **类属性** 或者调用 **类方法**

·这个时候，可以把这个方法封装成一个 **静态方法**

语法如下：

**@staticmethod**

**def 静态方法名():**

**pass**

·**静态方法** 需要用 **修饰器@staticmethod**来标识，**告诉解释器 这是一个静态方法**

·通过 **类名.** 调用 **静态方法**

**单例**

***单例设计模式：***

**·设计模式：**

·**设计模式** 是 **前人工作的总结和提炼**，通常，被人们广泛流 传的设计模式都是针对 **某一特定问题** 的成熟的解决方案

·使用 **设计模式** 是为了可重用代码、让代码更容易被他人理 解、保证代码可靠性

·**单例设计模式：**

·**目的** —— 让**类**创建的对象，在系统中**只有唯一的一个实例**

·每一次执行 **类名()**返回的对象，**内存地址是相同的**