学前须知

本版指引材料适合哪些同学?

仅适合正在学习或准备学习数据科学与人工智能，几乎完全没有Python、数据处理与可视化相关基础的学员，如果材料里内容你以前都系统学习过，只是因为不常使用遗忘，可以选择另外适合复习的材料版本。本教材学习目标是带你快速从0到1上手Python并掌握NumPy、Pandas、Matplotlib、Seaborn等数据处理与可视化内容，达到能边搜边写、尽快进入边学AI算法边做project的阶段。如果你的方向是Python脚本开发等其他领域，那么本材料抽取的知识结构和顺序并不适合。

注：学习本教材时，不要死记硬背，充分理解后能边搜边写就好了，计算机作为应用学科，写着写着就熟了、用着用着就牛了，加油！。

如何学编程，能事半功倍?

1. **围绕目标学，知识点是无穷的，我们学必要的**
   - 大部分学习目标，如果坚持按大致正确的路径学下去，即使走的是弯路，但最终也殊途同归，掌握它只是时间问题。互联网上学习资源并不稀奇，我们缺的是时间。所以虽然知识点是无穷的，但我们只学必要的，学习路上其实做加法容易，做减法难，新人最常见的坑是总担心学的不扎实、遗漏了什么重要知识点，所以学的宽、扣的细，导致迟迟不能建立起知识体系，更不谈融会贯通了，可能到最后也没有入门，实际上学的是否扎实主要看是否充分理解和用的多不多，很多旁枝末节问题大多后期会自然消解，如果以后遇不上的细节说明并不重要。
2. **站在编程语言设计者的角度充分理解其设计逻辑**，这样才能开悟，并越学越快，而不是单纯靠记忆积累。
   - 多思考并多尝试通过代码实验去感受编程的语法规则，即能通过代码测试得到答案的问题，就不搜索；
   - 习惯使用help和查官方文档，help类似Python的语法说明书，查官方文档可以更准确地debug，搜的博客经验可以参考，但往往写博客的人不会备注开发环境的详细版本配置，不同版本的经验不完全通用
3. **学以致用**
   - 多尝试去解决工作或生活中实际的问题，哪怕只解决了一部分或者陆续做了很久才解决，都可以有效激发学习兴趣和动力，对快速提升工程能力很有帮助，如果当下没有实际问题要解决时，“学以致用”的根本是多用，所以认真做coding作业也是OK的
4. **初学编程时，不复制粘贴**
   - 因为自己照着敲时也可能出现中英文字符混乱、缩进异常、拼写错误、少冒号等小错误，而这些小问题越早暴露就越早改正，debug和“写bug”的能力是全在平常的积累里，此时慢才是快。**所以，学习本文档时，多手动敲一敲**课程中的作业以及一些需要多次阅读才看懂的代码，当然，如果作业读完题后脑子里能清晰跳出答案，并且很有信心，那么也可以跳过，只要确保扎实掌握就行。

In [1]:
```
help(print) #help () 函数用于查看函数或模块用途的详细说明；help(print)也可以写为?print
```

如何高效提问?

1. 工作或学习沟通中，提问时应清楚说明问题背景，并无歧义地描述问题本身
2. 先思考后提问，提问前先搜索和思考，尝试无果后再考虑提问
3. 不钻牛角尖，比如，“为什么python中等号是==而不是=呢”，这类问题就像“为什么1+1=2”，新手单纯理解为语法规则就好，不必深究，成为顶级大牛时再回头“做科研”
4. 提问时尽可能讲出自己的理解，以便其他同学或老师能准确get并给出正确回复

Notebook使用

备注：jupyter notebook在非编辑状态（按ESC后）下按H可以查看并编辑快捷键，建议记下常用的。

- In[*]表示正在运行中

  打断方法：
  1.点击菜单Kernel并单击interrupt，如果不起作用再单击Restart；
  2.关闭页面重新打开

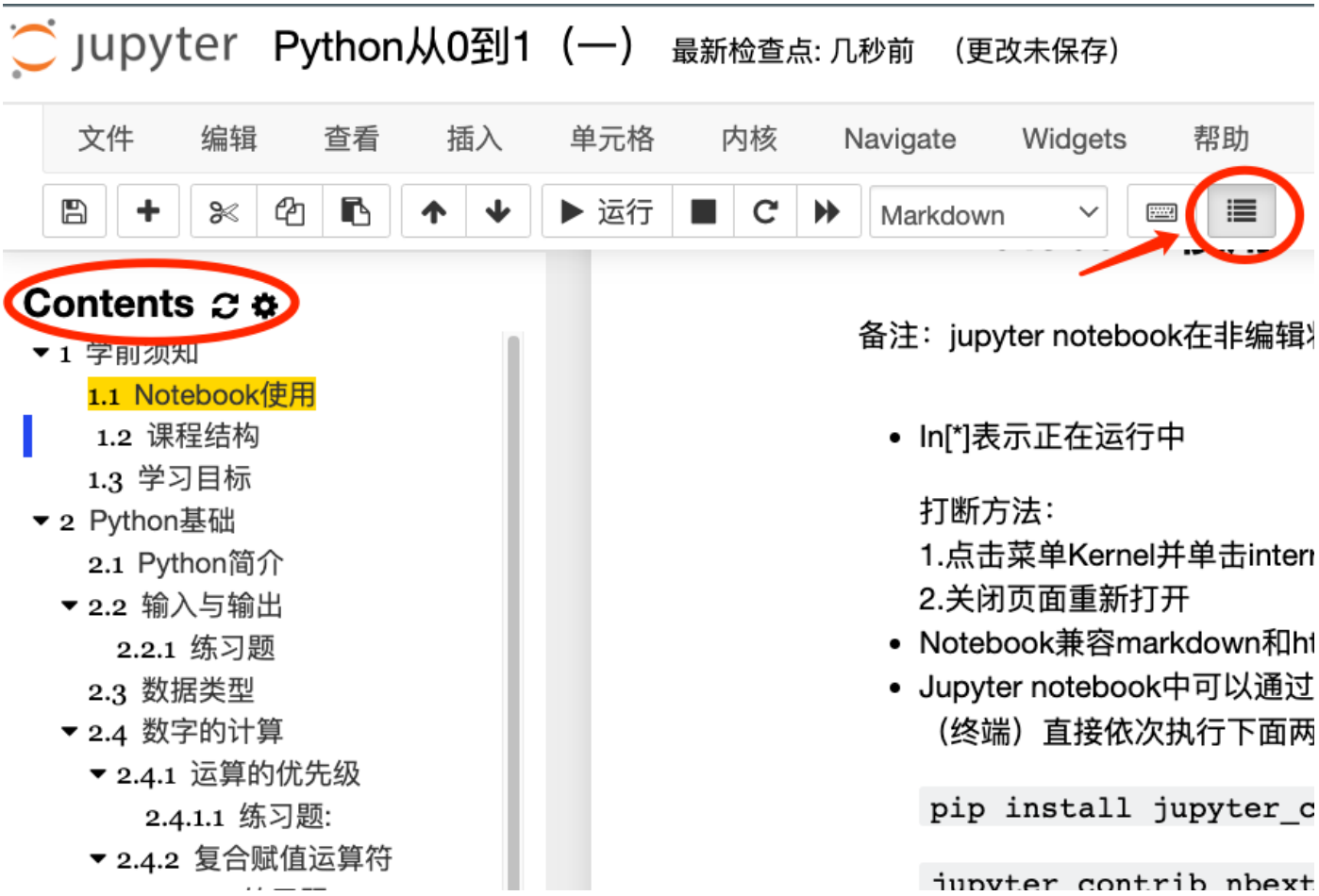
- Notebook兼容markdown和html语法，用法简单搜一下markdown的常见格式对应语法即可

- Jupyter notebook中可以通过安装nbextensions添加显示目录结构，阅读和编写会更方便些（**推荐安装**）； 如果需要，可以在jupyter新建terminals（终端）直接依次执行下面两行代码并重启后，在Nbextensions中开启Table of Contents即可：

```
pip install jupyter_contrib_nbextensions

jupyter contrib nbextension install --user
```

  安装后效果如下，在左侧可以显示Contents:



jupyter默认输出逻辑

In [2]:
```
# c 会默认输出
c = 2
c
```

In [3]:
```
# 只有e会被输出，而d不会被输出
d = 3
e = 1.45
```

```
d
e
```

jupyter重启

jupyter重启后，虽然页面上仍然显示结果，但此前已经执行的结果在后台是被完全清空的。

课程结构

我们AI编程阶段共设置7个章节：

章节	学习要求	难度	预计小时	目标	答案

Python从0到1（一）|掌握Python基础数据结构、循环、函数、切片等知识点，并完成作业练习|☆☆|5～7|掌握必要知识，完成常规作业，巩固知识|无 Python从0到1（二）|掌握进阶函数和面向对象编程相关内容，并完成作业练习|☆☆☆|5～7|知识学习及独立拓展新内容|无 「微博热搜预警分析系统1.0」|按提示独立学习部分新知识，并完成1个简化版舆情监测系统|☆☆☆☆|8～10|学会模块化独立写代码|部分提示 线性代数与NumPy|复习向量与矩阵、向量内积、矩阵乘法等线性代数知识的基本概念及其在Numpy中的表示与实现。了解向量化较循环计算带来的速度增益|☆☆|5～7|知识学习及作业|有 统计分析与Pandas|掌握Pandas主要数据结构及用法，理解数据IO，均值、方差、分位数、中位数、众数、偏度、峰度等概念及内涵，并用pandas进行实现，掌握Groupby、Apply等统计分析工具|☆☆|5～7|知识学习及作业|有 数据处理与可视化|Numpy、Pandas进行数据读取、数据处理，学习Matplotlib、Seaborn库进行可视化，掌握不同数据类型基本处理方法以及分析思路|☆☆|5～8|知识学习及实践|无 机器学习建模流程|参照指引搭建一个最简化的线性回归模型，来掌握建模流程|☆☆☆|6～9|掌握建模流程|参考提示

学习时间：

关于学习时间，这部分内容1～3周完成的学员都很正常，表里“预计小时”是基于历史学员反馈按实际有效时间预估的，如果时间相对分散，按适合自己的节奏规划、调整就好。

材料来源：

部分由布尔艺数导师原创，部分引用自互联网，然后由布尔艺数导师修改、编排。如果学习中有任何建议或反馈，比如哪些地方比较难以理解，或者材料有错误，**十分感谢**你愿意通过链接https://form.boolart.com/f/icBmbZ告知我们～以便我们进行更新调整

参考链接：

1. https://www.liaoxuefeng.com/wiki/1016959663602400
2. https://so.csdn.net/so/search?spm=1001.2100.3001.4498&q=python&t=&u=
3. https://www.runoob.com/python/python-tutorial.html
4. https://github.com/search?q=python
5. 其他书籍或文章会在对应引用位置注明

各小节的知识点材料里会明确提示要求掌握的程度，对应学习即可，接下来，我们正式开始学习！

Python基础

Python简介

- 交互模式与命令行模式
  执行python代码分为交互模式、命令行模式两种，在命令行模式下，可以直接运行一个.py文件，Python交

互模式的代码是输入一行，执行一行，而命令行模式下直接运行.py文件是一次性执行该文件内的所有代码。

- Python解释器
  Python代码通常保存为以.py为扩展名的文本文件。要运行代码，就需要Python解释器去执行.py文件。由于Python完全是开源的，所以任何人都可以编写Python解释器来执行Python代码。事实上，确实存在多种Python解释器，比如CPython、IPython、Jython等，解释器是用C语言开发的，所以叫CPython。在命令行下运行python就是启动CPython解释器；IPython是基于CPython之上的一个交互式解释器，也就是说，IPython只是在交互方式上有所增强，但是执行Python代码的功能和CPython是完全一样的;Jython是运行在Java平台上的Python解释器。
  - CPython用>>>作为提示符，而IPython用In [序号]:作为提示符。使用最广泛的是CPython。如果要和Java平台交互，通常不是用对应语言的Jython解释器，而是通过网络调用来交互，确保各程序之间的独立性。
  - Jupyter Notebook 默认保存的文件格式为 .ipynb，编写完成后也可以Download as 为.py格式
- Python语法中，**大小写敏感**
- Python语法**缩进有讲究**，不能乱写空格

In [4]:
```
a = 100
if a >= 0:
 print(a) # 如果缩进格式错误，jupyter可能会正常执行并红色提示，但在实际项目开发中，可能会导致报错
else:
    print(-a)
```

In [5]:
```
"""
多行注释
可以这样写
其实就是字符串， 只是没有进行操作
"""
```

In [6]:
```
# 和多行字符串没有什么区别
s = """这是一个
可以换行的字符串，
你有了这样一个字符串， 会比较方便，
注意后面有换行符的存在"""
```

In [7]:
```
s
```

In [8]:
```
print(s)
```

Python的语法比较简单，采用缩进方式，写出来的代码就像下面的样子：

```
# 打印一个数字的绝对值
a = 100
if a >=0:
    print(a)
else:
    print(-a)
```

Python 利用缩进来划分代码块

这种语法形式的优点是看起来整洁,美观,没有大量的括号,更便于阅读,看起来更像人类的自然语言.

可以强迫你写出格式化的代码,通常我们使用Tab键或者四个空格来完成缩进。

坏处是复制粘贴之后,代码的缩进会不一样,这时候必须手动重新调整.

在Python 中，对于类定义、函数定义、流程控制语句、异常处理语句等，行尾的冒号和下一行的缩进，表示下一个代码块的开始，而缩进的结束则表示此代码块的结束。

输入与输出

In [9]:
```
print('输出')
```

In [10]:
```
help(print)
```

从help的描述中可以看出：

- print也可以接受多个字符串，用逗号“,”隔开，就可以连成一串输出
- sep、end可以添加设置间隔和结尾字符格式

In [11]:
```
print('15+16=',15+16,sep='')
```

In [12]:
```
print('iron Man', 'Spideman','Hulk',sep='===>')#使用sep设置中间分隔符
```

In [13]:
```
print('iron Man', 'Spideman','Hulk',sep='===>',end='......')#使用end设置结尾，并且默认已经换行
print('Captain')
```

In [14]:
```
print("1234567890-------") # 会在一行显示
print("1234567890\n-------")# 使用 \n 换行
```

In [15]:
```
print("-"*50)
```

In [16]:
```
city=input() #Python提供了一个input()，可以让用户输入字符串，并存放到一个变量里。
```

成都

In [17]:
```
city=input('再输入不同于第一个的城市名称')
```

再输入不同于第一个的城市名称北京

In [18]:
```
print(city) # 此时打印出的city只出现了第二次输入的城市名，变量city的内容被覆盖了
```

北京

In [19]:
```
print('╔═══════════════════════════════════════════════╗')
print('╟  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ┆  ╢')
print('╟  ┆春┆滟┆江┆空┆江┆江┆人┆不┆白┆谁┆可┆玉┆此┆鸿┆昨┆江┆斜┆不┆  ╢')
print('╟  ┆江┆滟┆流┆里┆天┆畔┆生┆知┆云┆家┆怜┆户┆时┆雁┆夜┆水┆月┆知┆  ╢')
print('╟  ┆潮┆随┆宛┆流┆一┆何┆代┆江┆一┆今┆楼┆帘┆相┆长┆闲┆流┆沉┆乘┆  ╢')
```

```
print('╔  |水|波|转|霜|色|人|代|月|片|夜|上|中|望|飞|潭|春|沉|月|  ╗')
print('╠  |连|千|绕|不|无|初|无|待|去|扁|月|卷|不|光|梦|去|藏|几|  ╣')
print('╠春|海|万|芳|觉|纡|见|穷|何|悠|舟|徘|不|相|不|落|欲|海|人|  ╣')
print('╠江|平|里|甸|飞|尘|月|已|人|悠|子|徊|去|闻|度|花|尽|雾|归|  ╣')
print('╠花|，|，|，|，|，|，|，|，|，|，|，|，|，|，|，|，|，|，|  ╣')
print('╠月|海|何|月|汀|皎|江|江|但|青|何|应|捣|愿|鱼|可|江|碣|落|  ╣')
print('╠夜|上|处|照|上|皎|月|月|见|枫|处|照|衣|逐|龙|怜|潭|石|月|  ╣')
print('╠  |明|春|花|白|空|何|年|长|浦|相|离|砧|月|潜|春|落|潇|摇|  ╣')
print('╠  |月|江|林|沙|中|年|年|江|上|思|人|上|华|跃|半|月|湘|情|  ╣')
print('╠  |共|无|皆|看|孤|初|望|送|不|明|妆|拂|流|水|不|复|无|满|  ╣')
print('╠  |潮|月|似|不|月|照|相|流|胜|月|镜|还|照|成|还|西|限|江|  ╣')
print('╠  |生|明|霰|见|轮|人|似|水|愁|楼|台|来|君|文|家|斜|路|树|  ╣')
print('╠  |。|。|。|。|。|？|。|。|。|？|。|。|。|。|。|。|。|。|  ╣')
print('╠  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  ╣')
print('╚═══════════════════════════════════════════════════════════╝')
```

```
╔═══════════════════════════════════════════════════════════╗
╠  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  ╣
╠  |春|滟|江|空|江|江|人|不|白|谁|可|玉|此|鸿|昨|江|斜|不|  ╣
╠  |江|滟|流|里|天|畔|生|知|云|家|怜|户|时|雁|夜|水|月|知|  ╣
╠  |潮|随|宛|流|一|何|代|江|一|今|楼|帘|相|长|闲|流|沉|乘|  ╣
╠  |水|波|转|霜|色|人|代|月|片|夜|上|中|望|飞|潭|春|沉|月|  ╣
╠  |连|千|绕|不|无|初|无|待|去|扁|月|卷|不|光|梦|去|藏|几|  ╣
╠春|海|万|芳|觉|纡|见|穷|何|悠|舟|徘|不|相|不|落|欲|海|人|  ╣
╠江|平|里|甸|飞|尘|月|已|人|悠|子|徊|去|闻|度|花|尽|雾|归|  ╣
╠花|，|，|，|，|，|，|，|，|，|，|，|，|，|，|，|，|，|，|  ╣
╠月|海|何|月|汀|皎|江|江|但|青|何|应|捣|愿|鱼|可|江|碣|落|  ╣
╠夜|上|处|照|上|皎|月|月|见|枫|处|照|衣|逐|龙|怜|潭|石|月|  ╣
╠  |明|春|花|白|空|何|年|长|浦|相|离|砧|月|潜|春|落|潇|摇|  ╣
╠  |月|江|林|沙|中|年|年|江|上|思|人|上|华|跃|半|月|湘|情|  ╣
╠  |共|无|皆|看|孤|初|望|送|不|明|妆|拂|流|水|不|复|无|满|  ╣
╠  |潮|月|似|不|月|照|相|流|胜|月|镜|还|照|成|还|西|限|江|  ╣
╠  |生|明|霰|见|轮|人|似|水|愁|楼|台|来|君|文|家|斜|路|树|  ╣
╠  |。|。|。|。|。|？|。|。|。|？|。|。|。|。|。|。|。|。|  ╣
╠  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  ╣
╚═══════════════════════════════════════════════════════════╝
```

练习题

小练习：编写代码完成以下名片的显示

```
==================================
    姓名: dongGe
    QQ:xxxxxxxxxx
    手机号:131xxxxxx
    公司地址:北京市xxxx
==================================
```

In [ ]:

小练习：任意使用input输入a,b,c三个整数，计算a+b-c的结果, 并打印出来

In [ ]:

数据类型

计算机可以处理各种类型的数据，比如视频、图像、声音、文字、数字等，不同的数据，会定义不同的数据类型，在Python中，能够**直接处理**的数据类型有：

数据类型	type	解释说明

整数|int|- 浮点数|float|就是小数,之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的，比如，1.23x109和12.3x108是完全相等的。浮点数可以用数学写法，如1.23，3.14，-9.01，等等。但是对于很大或很小的浮点数，就必须用科学计数法表示，把10用e替代，1.23x109就是1.23e9，或者12.3e8，0.000012可以写成1.2e-5，等等 字符串|str|字符串是以单引号'或双引号"括起来的任意文本，如果'本身也是一个字符，那就可以用""括起来,如果字符串内部既包含'又包含"怎么办？可以用转义字符\来标识，**如果字符串里面有很多字符都需要转义，就需要加很多\，为了简化，Python还允许用r''表示''内部的字符串默认不转义** 布尔值|bool|布尔值和布尔代数的表示完全一致，一个布尔值只有True、False两种值,布尔值可以用and、or和not运算 空值|NoneType|空值用None表示。注意不同于0，因为0是有意义的，而None是一个特殊的空值。

Python还提供了列表、字典等多种数据类型，还允许创建自定义数据类型，后续会陆续讲到。

In [20]:
```
type(3)
```

Out[20]:
```
int
```

In [21]:
```
type(3.1415926)
```

Out[21]:
```
float
```

In [22]:
```
print(r'\\\\nr\\”)))a')
```

```
\\\\nr\\”)))a
```

In [23]:
```
print('''line1
... line2
... line3''')
#如果字符串内部有很多换行，用\n写在一行里不好阅读，为了简化，
#Python允许用'''...'''的格式表示多行内容
```

```
line1
line2
line3
```

数字的计算

下面以a=10 ,b=20为例进行计算

运算符	描述	实例
+	加	两个对象相加 a + b 输出结果 30
-	减	得到负数或是一个数减去另一个数 a - b 输出结果 -10
*	乘	两个数相乘或是返回一个被重复若干次的字符串 a * b 输出结果 200
/	除	x除以y b / a 输出结果 2

运算符	描述	实例
//	取整除	返回商的整数部分 9//2 输出结果 4 , 9.0//2.0 输出结果 4.0
%	取余	返回除法的余数 b % a 输出结果 0
**	幂	返回x的y次幂 a**b 为10的20次方， 输出结果 100000000000000000000

In [24]:
```
# Python 内置的基础运算符
# + - * / ** // %
```

In [25]:
```
3 ** 3
```

Out[25]:
```
27
```

In [26]:
```
9 / 6
```

Out[26]:
```
1.5
```

In [27]:
```
9 // 6
```

Out[27]:
```
1
```

In [28]:
```
9 % 6
```

Out[28]:
```
3
```

运算的优先级

和数学上差不多，乘除大于加减、指数运算大于乘除，层级较多时推荐使用小括号界定优先级

In [29]:
```
2+3*5
```

Out[29]:
```
17
```

In [30]:
```
(4*2)**3
```

Out[30]:
```
512
```

练习题:

一个苹果的价格是10元, 一个梨的价格是12元, 如果买6个苹果, 三个梨, 需缴纳交易税为22%, 总共需要花费多少钱?

In [ ]:
```

```

复合赋值运算符

运算符	描述	实例
+=	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

运算符	描述	实例
/=	除法赋值运算符	c /= a 等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a 等效于 c = c** a
//=	取整除赋值运算符	c //= a 等效于 c = c // a

In [31]:
```
k = 1
j = 9
j+=k   #等价于j = j+k
j
```

Out[31]:
```
10
```

练习题

用普通运算符和复合运算符分别计算以下问题：

有一个数字，数字的值为12，让这个数字在原有的基础上减去2，求减去后的结果是多少。

python中的比较运算符

运算符	描述	示例
==	检查两个操作数的值是否相等，如果是则条件变为真。	如a=3,b=3则（a == b) 为 true.
!=	检查两个操作数的值是否相等，如果值不相等，则条件变为真。	如a=1,b=3则(a != b) 为 true.
>	检查左操作数的值是否大于右操作数的值，如果是，则条件成立。	如a=7,b=3则(a > b) 为 true.
<	检查左操作数的值是否小于右操作数的值，如果是，则条件成立。	如a=7,b=3则(a < b) 为 false.
>=	检查左操作数的值是否大于或等于右操作数的值，如果是，则条件成立。	如a=3,b=3则(a >= b) 为 true.
<=	检查左操作数的值是否小于或等于右操作数的值，如果是，则条件成立。	如a=3,b=3则(a <= b) 为 true.

注意"等于""的运算符是两个等号"==","不等于""的运算符是"!="

逻辑运算符

运算符	逻辑表达式	描述	实例
and	x and y	布尔"与" - 如果 x 为 False，x and y 返回 False，否则它返回 y 的计算值。	(a and b) 返回 20。
or	x or y	布尔"或" - 如果 x 是 True，它返回 True，否则它返回 y 的计算值。	(a or b) 返回 10。
not	not x	布尔"非" - 如果 x 为 True，返回 False 。如果 x 为 False，它返回 True。	not(a and b) 返回 False

特殊运算符

- is/is not 判断是否指向同一个引用
- in/ not in 判定某个变量是否在给定容器中

math科学计算库

	python内置的一些计算函数：
abs(x)	返回x的绝对值，类型随x
max(n1, n2, ...)	返回最大值

python内置的一些计算函数：	
min(n1, n2, ...)	返回最小值
round(x [,n])	默认返回浮点数x的四舍五入值，如给出n值，则代表舍入到小数点后的n位。例如round(1.23456, 3)返回1.235

python还提供科学计算等库，例如math，导入math库后，部分常用的函数有：	
fabs(x)	返回x的绝对值，类型是浮点数
ceil(x)	取x的上入整数，如math.ceil(4.1)返回5
floor(x)	取x的下入整数，如math.floor(4.9)返回4
exp(x)	返回e的x次幂，e是自然常数
sqrt(x)	返回x的平方根，返回值是float类型
modf(x)	返回x的整数部分和小数部分，两部分的符号与x相同，整数部分以浮点型表示。例如math.modf(4.333)，返回元组(0.3330000000000002, 4.0)
log10(x)	返回以10为基数的x的对数，返回值类型是浮点数
log(x,y)	返回以y为基数的x的对数，返回值类型是浮点数
pow(x, y)	返回x的y次幂，即x**y

math库有映像就行，用到时忘记搜一下就好

小练习: 计算根号5加上17的平方的结果取10的对数的

$$\log_{10}\left(\sqrt{5}+17^2\right)$$

In [32]:
```
import math # 使用一个库前先导入
```

In [33]:
```
math.log(10,2)
```

Out[33]:
```
3.3219280948873626
```

变量与常量

变量

1. python是动态数据类型
2. python中的变量不需要声明, 直接赋值就可以使用
3. 变量在使用之前必须进行赋值
4. Python会根据你赋给的值自动判断变量的数据类型
   - 其实变量并没有什么类型
   - 变量只是指向了一个内存地址, 内存地址中储存了我们的数据, 这个数据是具有数据类型的
   - 变量可以重复赋值, 后面的值会覆盖前边的值
   - 改变变量的值, 其实就是改变了变量指向的内存地址!

要理解变量,一定要理解数据在内存中的储存形式,变量只是一个对在内存中储存的某一个东西的指引

变量指引的位置可以变化,这是变量的值就会发生变化

当我使用变量,实际上是相当于在使用变量指向的那个内存地址中的值.

变量命名规则：

1. 变量名的长度不受限制，但其中的字符必须是字母、数字、或者下划线_，而不能使用空格、连字符、标点符号、引号或其他字符。
2. 变量名的第一个字符不能是数字，而必须是字母或下划线。
3. Python区分大小写。
4. 不能将Python关键字用作变量名。

In [34]:
```
a=1 #变量a是一个整数。
```

In [35]:
```
t_007 = 'T007' #变量t_007是一个字符串。
```

In [36]:
```
Answer = True #变量Answer是一个布尔值True。
```

In [37]:
```
import keyword
keyword.kwlist # 这些关键字全部不能作为变量名使用；给关键字赋值也会报错
```

Out[37]:
```
['False',
 'None',
 'True',
 'and',
 'as',
 'assert',
 'async',
 'await',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

在Python中，等号=是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，例如：

In [38]:
```
a = 123 # a是整数
print(a)
a = 'ABC' # a变为字符串
print(a)
```

```
123
ABC
```

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言，赋值语句如下（// 表示注释）：

```
int a = 123; // a是整数类型变量
a = "ABC"; // 错误：不能把字符串赋给整型变量
```

和静态语言相比，动态语言更灵活，就是这个原因。不要把赋值语句的等号等同于数学的等号。比如下面的代码：

In [39]:
```
x = 10
x = x + 2
```

如果从数学上理解x = x + 2那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式x + 2，得到结果12，再赋给变量x。由于x之前的值是10，重新赋值后，x的值变成12。

最后，理解变量在计算机内存中的表示也非常重要。当我们写：

In [40]:
```
a = 'ABC'
```

时，Python解释器干了两件事情：

1. 在内存中创建了一个'ABC'的字符串；
2. 在内存中创建了一个名为a的变量，并把它指向'ABC'。

也可以把一个变量a赋值给另一个变量b，这个操作实际上是把变量b指向变量a所指向的数据，例如下面的代码：

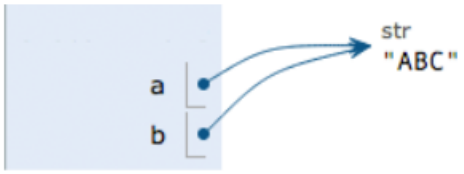
In [41]:
```
a = 'ABC'
b = a
a = 'XYZ'
print(b)
```

```
ABC
```

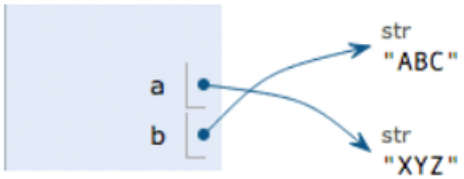
最后一行打印出变量 `b` 的内容到底是 `'ABC'` 呢还是 `'XYZ'`？如果从数学意义上理解，就会错误地得出 `b` 和 `a` 相同，也应该是 `'XYZ'`，但实际上 `b` 的值是 `'ABC'`，让我们一行一行地执行代码，就可以看到到底发生了什么事：

执行 `a = 'ABC'`，解释器创建了字符串 `'ABC'` 和变量 `a`，并把 `a` 指向 `'ABC'`：

执行 `b = a`，解释器创建了变量 `b`，并把 `b` 指向 `a` 指向的字符串 `'ABC'`：



执行 `a = 'XYZ'`，解释器创建了字符串'XYZ'，并把 `a` 的指向改为 `'XYZ'`，但 `b` 并没有更改：



所以，最后打印变量 `b` 的结果自然是 `'ABC'` 了。

Python支持多种数据类型，在计算机内部，可以把任何数据都看成一个“对象”，而变量就是在程序中用来指向这些数据对象的，对变量赋值就是把数据和变量给关联起来。

对变量赋值x = y是把变量x指向真正的对象，该对象是变量y所指向的。随后对变量y的赋值不影响变量x的指向。

> 注意：Python的整数没有大小限制，而某些语言的整数根据其存储长度是有大小限制的，例如Java对32位整数的范围限制在–2147483648–2147483647。
> Python的浮点数也没有大小限制，但是超出一定范围就直接表示为inf（无限大）。

常量

所谓常量就是不能变的变量，比如常用的数学常数π就是一个常量。在Python中，通常用全部大写的变量名表示常量：

In [42]:
```
PI = 3.14159265359
```

但事实上 PI 仍然是一个变量，Python根本没有任何机制保证 PI 不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量PI的值，也没人能拦住你。

整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的（除法难道也是精确的？是的！），而浮点数运算则可能会有四舍五入的误差。

解释一下整数的除法为什么也是精确的。在Python中，有两种除法，一种除法是 / ：

In [43]:
```
10 / 3
```

Out[43]:
```
3.3333333333333335
```

In [44]:
```
9/3 #除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数：
```

Out[44]:
```
3.0
```

还有一种除法是 `//` ，称为地板除，两个整数的除法仍然是整数：

In [45]:
```
10 // 3
```

Out[45]:
```
3
```

你没有看错，整数的地板除 `//` 永远是整数，即使除不尽。要做精确的除法，使用 `/` 就可以。 因为 `//` 除法只取结果的整数部分，所以Python还提供一个余数运算，可以得到两个整数相除的余数：

In [46]:
```
10 % 3
```

Out[46]:
```
1
```

无论整数做 `//` 除法还是取余数，结果永远是整数，所以，Python的语法和运算的规则决定了整数运算结果永远是精确的。

字符编码与格式化

字符编码

编码这节内容快速阅读，了解概念即可，阅读后只需要知道：

1. Python 3的字符串使用Unicode，直接支持多语言。当str和bytes互相转换时，需要指定编码。最常用的编码是 `UTF-8` 。Python当然也支持其他编码方式，比如把 `Unicode` 编码成 `GB2312` ,但这种方式纯属自找麻烦，如果没有特殊业务要求，请**牢记仅使用 `UTF-8` 编码**。
2. 能弄清 `ASCII` 、 `Unicode` 和 `UTF-8` 的关系

因为计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。最早的计算机在设计时采用8个比特（bit）作为一个字节（byte），所以，一个字节能表示的最大的整数就是255（二进制11111111=十进制255），如果要表示更大的整数，就必须用更多的字节。比如两个字节可以表示的最大整数是65535，4个字节可以表示的最大整数是4294967295。

由于计算机是美国人发明的，因此，最早只有127个字符被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为ASCII编码，比如大写字母A的编码是65，小写字母z的编码是122。

但是要处理中文显然一个字节是不够的，至少需要两个字节，而且还不能和 `ASCII` 编码冲突，所以，中国制定了 `GB2312` 编码，用来把中文编进去。

你可以想得到的是，全世界有上百种语言，日本把日文编到 `Shift_JIS` 里，韩国把韩文编到 `Euc-kr` 里，各国有各国的标准，就会不可避免地出现冲突，结果就是，在多语言混合的文本中，显示出来会有乱码。

因此， `Unicode` 字符集应运而生。 `Unicode` **把所有语言都统一到一套编码里**，这样就不会再有乱码问题了。 `Unicode` 标准也在不断发展，但最常用的是 `UCS-16` 编码，用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要4个字节）。现代操作系统和大多数编程语言都直接支持 `Unicode` 。

现在，捋一捋 ASCII 编码和 Unicode 编码的区别： ASCII 编码是1个字节，而 Unicode 编码通常是2个字节。

字母A用ASCII编码是十进制的65，二进制的 `01000001` ；

字符0用ASCII编码是十进制的48，二进制的 `00110000` ，注意字符 `'0'` 和整数 `0` 是不同的；

汉字 中 已经超出了 ASCII 编码的范围，用Unicode编码是十进制的 `20013` ，二进制的 `01001110 00101101` 。

你可以猜测，如果把 ASCII 编码的 A 用 Unicode 编码，只需要在前面补0就可以，因此，A的 Unicode 编码是 `00000000 01000001` 。

新的问题又出现了：如果统一成 Unicode 编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用 Unicode 编码比 ASCII 编码需要多一倍的存储空间，在存储和传输上就十分不划算。

所以，本着节约的精神，又出现了把 Unicode 编码转化为“可变长编码”的 UTF-8 编码。 UTF-8 编码把一个 Unicode 字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。如果你要传输的文本包含大量英文字符，用 UTF-8 编码就能节省空间：

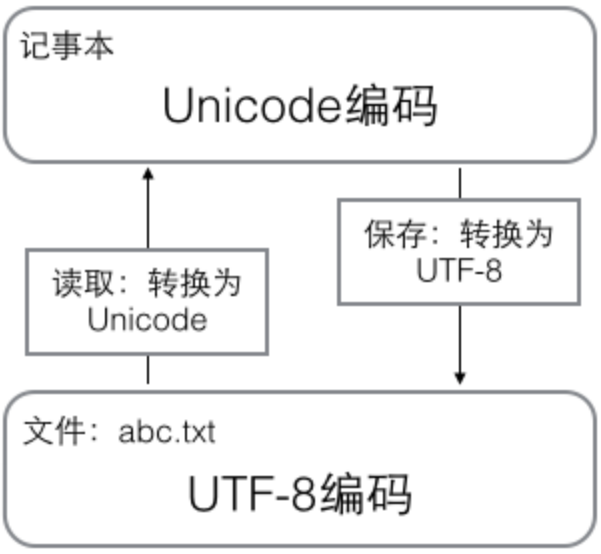
字符	ASCII	Unicode	UTF-8
A	01000001	00000000 01000001	01000001
中	x	01001110 00101101	11100100 10111000 10101101

从上面的表格还可以发现， UTF-8 编码有一个额外的好处，就是 ASCII 编码实际上可以被看成是 UTF-8 编码的一部分，所以，大量只支持 ASCII 编码的历史遗留软件可以在 UTF-8 编码下继续工作。

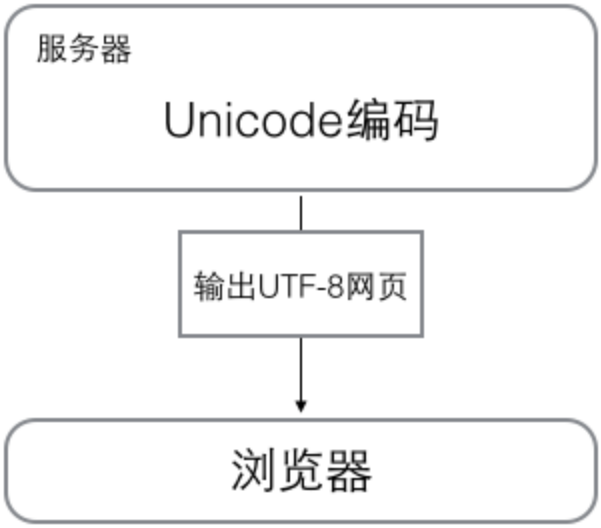
搞清楚了 ASCII 、 Unicode 和 UTF-8 的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

在计算机内存中，统一使用 Unicode 编码，当需要保存到硬盘或者需要传输的时候，就转换为 UTF-8 编码。

用记事本编辑的时候，从文件读取的 UTF-8 字符被转换为 Unicode 字符到内存里，编辑完成后，保存的时候再把 Unicode 转换为 UTF-8 保存到文件：



浏览网页的时候，服务器会把动态生成的 Unicode 内容转换为 UTF-8 再传输到浏览器：



所以你看到很多网页的源码上会有类似的信息，表示该网页正是用的UTF-8编码。

格式化输出

有时候我们需要根据变量变化地输出内容，比如'亲爱的xxx你好！你xx月的话费是xx，余额是xx'之类的字符串，而xxx的内容就是变量。

我们掌握两种格式化方法：

简洁通用的f-string方法

print() 函数使用以%开头的转换说明符对各种类型的数据进行格式化输出，具体请看下表。

转换说明符	解释	备注

%d、%i |转换为带符号的十进制整数|记住 %o |转换为带符号的八进制整数 %x、%X |转换为带符号的十六进制整数 %e |转化为科学计数法表示的浮点数（e 小写） %E |转化为科学计数法表示的浮点数（E 大写） %f、%F |转化为十进制浮点数|记住 %g |智能选择使用 %f 或 %e 格式|记住 %G |智能选择使用 %F 或 %E 格式|记住 %c |格式化字符及其 ASCII 码 %r |使用 repr() 函数将表达式转换为字符串|记住 %s |使用 str() 函数将表达式转换为字符串|记住

记不住的话也无妨，多搜几次就记住了

转换说明符（Conversion Specifier）只是一个占位符，它会被后面表达式（变量、常量、数字、字符串、加减乘除等各种形式）的值代替。例：

In [47]:
```
BoolArt = '布尔艺数'
print("%r的英文缩写是Boolart" % BoolArt)
```

'布尔艺数'的英文缩写是Boolart

In [48]:
```
BoolArt = '布尔艺数'
AI = 'AI从业者的职业伙伴'
print("%s是%s,他的公众号名称是%s" % (BoolArt,AI,BoolArt))
```

布尔艺数是AI从业者的职业伙伴,他的公众号名称是布尔艺数

我们也可以输出时指定最小输出宽度：

当使用转换说明符时，可以使用下面的格式指定最小输出宽度（至少占用多少个字符的位置）：

- %10d 表示输出的整数宽度至少为 10；

- %20s 表示输出的字符串宽度至少为 20。

In [49]:
```
n = 1234567
print("宽带为10:%10d" % n)
print("宽带为5:%5d" % n)
url = "http://boolart.com/"
print("宽度为45:%45s" % url)
print("宽度为20:%20s" % url)
```

```
宽带为10:   1234567
宽带为5:1234567
宽度为45:                             http://boolart.com/
宽度为20: http://boolart.com/
```

当然，我们也可以**指定对齐格式**，默认情况下，print() 输出的数据总是右对齐的。也就是说，当数据不够宽时，数据总是靠右边输出，而在左边补充空格以达到指定的宽度。Python 允许在最小宽度之前增加一个标志来改变对齐方式，Python 支持的标志如下：

标志	说明
-	指定左对齐
+	表示输出的数字总要带着符号；正数带+，负数带-。
0	表示宽度不足时补充 0，而不是补充空格。

- 对于整数，指定左对齐时，在右边补 0 是没有效果的，因为这样会改变整数的值。
- 对于小数，以上三个标志可以同时存在。
- 对于字符串，只能使用-标志，因为符号对于字符串没有意义，而补 0 会改变字符串的值。

In [50]:
```
n = 123456
# %09d 表示最小宽度为9，左边补0
print("n(09):%09d" % n)
# %+9d 表示最小宽度为9，带上符号
print("n(+9):%+9d" % n)
f = 140.5
# %-+010f 表示最小宽度为10，左对齐，带上符号
print("f(-+0):%-+010f" % f)
s = "Hello"
# %-10s 表示最小宽度为10，左对齐
print("s(-10):%-10s." % s)
```

```
n(09):000123456
n(+9):  +123456
f(-+0):+140.500000
s(-10):Hello     .
```

指定小数精度(了解即可)

对于小数（浮点数），print() 还允许指定小数点后的数字位数，也即指定小数的输出精度。

精度值需要放在最小宽度之后，中间用点号 . 隔开；也可以不写最小宽度，只写精度。具体格式如下：

`%m.nf`

`%.nf`

m 表示最小宽度， n 表示输出精度， . 是必须存在的。

In [51]:
```
f = 3.141592653
# 最小宽度为8，小数点后保留3位
print("%8.3f" % f)
# 最小宽度为8，小数点后保留3位，左边补0
print("%08.3f" % f)
# 最小宽度为8，小数点后保留3位，左边补0，带符号
print("%+08.3f" % f)
```

```
   3.142
0003.142
+003.142
```

str.format()方法

In [52]:
```
#str.format() 方法的基本用法
print(r'"Artificial intelligence"的缩写是字母{}和{}'.format('A', 'I'))
```

```
"Artificial intelligence"的缩写是字母A和I
```

花括号及之内的字符（称为格式字段）被替换为传递给 str.format() 方法的对象。花括号中的数字表示传递给 str.format() 方法的对象所在的位置:

In [53]:
```
print('{0} and {1}'.format('spam', 'eggs'))
print('{1} and {0}'.format('spam', 'eggs'))
```

```
spam and eggs
eggs and spam
```

使用关键字参数名引用值:

In [54]:
```
print('This {food} is {adjective}.'.format(
...     food='spam', adjective='absolutely horrible'))
```

```
This spam is absolutely horrible.
```

位置参数和关键字参数可以任意组合:

In [55]:
```
print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...     other='Georg'))
```

```
The story of Bill, Manfred, and Georg.
```

格式化输出的方法有多种，掌握以上两种方法可以覆盖大部分使用场景了。

In [ ]:
```
```

list和tuple

list

Python内置的一种数据类型是列表：list。list是一种有序的集合，可以随时添加和删除其中的元素。

比如，列出班里所有同学的名字，就可以用一个list表示：

In [56]:
```
classmates = ['Michael', 'Bob', 'Tracy']
```

```
classmates
```

Out[56]:
```
['Michael', 'Bob', 'Tracy']
```

变量 `classmates` 就是一个list。用len()函数可以获得list元素的个数：

In [57]:
```
len(classmates)
```

Out[57]:
```
3
```

用索引来访问list中每一个位置的元素，记得索引是从 `0` 开始的：

In [58]:
```
classmates[0]
```

Out[58]:
```
'Michael'
```

In [59]:
```
classmates[2]
```

Out[59]:
```
'Tracy'
```

In [60]:
```
classmates[3]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-60-81a27e3ce05f> in <module>
----> 1 classmates[3]

IndexError: list index out of range
```

当索引超出了范围时，Python会报一个IndexError错误，所以，要确保索引不要越界，记得最后一个元素的索引是 `len(classmates) - 1` 。

如果要取最后一个元素，除了计算索引位置外，还可以用 `-1` 做索引，直接获取最后一个元素：

In [61]:
```
classmates[-1]
```

Out[61]:
```
'Tracy'
```

以此类推，可以获取倒数第2个、倒数第3个，当然，倒数第4个就越界了，你可以写代码尝试下。

list是一个可变的有序表，所以，可以往list中追加元素到末尾：

In [62]:
```
classmates.append('Adam')
classmates.append('Adam')
```

In [63]:
```
classmates #重复执行上面cell的代码，classmates里可以重复追加
```

Out[63]:
```
['Michael', 'Bob', 'Tracy', 'Adam', 'Adam']
```

也可以把元素插入到指定的位置，比如索引号为1的位置：

In [64]:
```
classmates.insert(1, 'Jack')
```

In [65]:
```
classmates
```

Out[65]: ['Michael', 'Jack', 'Bob', 'Tracy', 'Adam', 'Adam']

要删除list末尾的元素，用 pop() 方法：

In [66]:
```
print('原classmates: ',classmates)
classmates.pop()
print('classmates.pop(): ',classmates)
#删除指定位置的元素，用pop(i)方法，其中i是索引位置:
classmates.pop(1)
print('classmates.pop(1)',classmates)
```

```
原classmates:  ['Michael', 'Jack', 'Bob', 'Tracy', 'Adam', 'Adam']
classmates.pop():  ['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
classmates.pop(1) ['Michael', 'Bob', 'Tracy', 'Adam']
```

In [67]:
```
classmates[1] = 'Sarah' #要把某个元素替换成别的元素，可以直接赋值给对应的索引位置
classmates
```

Out[67]: ['Michael', 'Sarah', 'Tracy', 'Adam']

list里面的元素的数据类型也可以不同，比如：

In [68]:
```
L = ['Apple', 123, True]
```

list元素也可以是另一个list，比如：

In [69]:
```
s = ['python', 'java', ['asp', 'php'], 'scheme']
len(s) #要注意s只有4个元素，其中s[2]又是一个list
```

Out[69]: 4

要拿到'php'可以写p[1]或者s[2][1]，因此s可以看成是一个二维数组，类似的还有三维、四维.....数组，不过很少用到。

In [70]:
```
s[2][1]
```

Out[70]: 'php'

如果一个list中一个元素也没有，就是一个空的list，它的长度为0：

In [71]:
```
L = []
len(L)
```

Out[71]: 0

tuple

另一种有序列表叫元组：tuple。tuple和list非常类似，但是tuple一旦初始化就不能修改，比如同样是列出同学的名字：

In [72]:
```
classmates = ('Michael', 'Bob', 'Tracy')
```

现在，classmates这个tuple不能变了，它也没有append()，insert()这样的方法。其他获取元素的方法和list是

一样的，你可以正常地使用classmates[0]，classmates[-1]，但不能赋值成另外的元素。

不可变的tuple有什么意义？因为tuple不可变，所以代码更安全。如果可能，能用tuple代替list就尽量用tuple。

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

In [73]:
```
t = (1, 2)
t
```

Out[73]:
```
(1, 2)
```

如果要定义一个空的tuple，可以写成()：

In [74]:
```
t = ()
t
```

Out[74]:
```
()
```

但是，要定义一个只有1个元素的tuple，如果你这么定义：

In [75]:
```
t = (1)
t
```

Out[75]:
```
1
```

In [76]:
```
type(t)
```

Out[76]:
```
int
```

定义的不是tuple，是1这个数！这是因为括号()既可以表示tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python规定，这种情况下，按小括号进行计算，计算结果自然是1。所以，只有1个元素的tuple定义时必须加一个逗号 ，，来消除歧义：

In [77]:
```
t = (1,)
t
```

Out[77]:
```
(1,)
```

In [78]:
```
type(t)
```

Out[78]:
```
tuple
```

最后来看一个“可变的”tuple：

In [79]:
```
t = ('a', 'b', ['A', 'B'])
t[2][0] = 'X'
t[2][1] = 'Y'
t
```

Out[79]:
```
('a', 'b', ['X', 'Y'])
```

这个tuple定义的时候有3个元素，分别是'a'，'b'和一个list。不是说tuple一旦定义后就不可变了吗？怎么后来又变了？

表面上看，tuple的元素确实变了，但其实变的不是tuple的元素，而是list的元素。tuple一开始指向的list并没有改成别的list，所以，tuple所谓的“不变”是说，tuple的每个元素，指向永远不变。即指向'a'，就不能改成指向'b'，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的！

条件判断

计算机之所以能做很多自动化的任务，因为它可以自己做条件判断。

比如，输入用户年龄，根据年龄打印不同的内容，在Python程序中，用 `if` 语句实现：

In [80]:
```
age = 20
if age >= 18:
    print('your age is', age)
    print('adult')
```

```
your age is 20
adult
```

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

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

In [81]:
```
age = 3
if age >= 18: #注意不要少写了冒号
    print('your age is', age)
    print('adult')
else:
    print('your age is', age)
    print('teenager')
```

```
your age is 3
teenager
```

当然上面的判断是很粗略的，完全可以用 `elif` 做更细致的判断：

In [82]:
```
age = 3
if age >= 18:
    print('adult')
elif age >= 6:
    print('teenager')
else:
    print('kid')
```

```
kid
```

`elif` 是 `else if` 的缩写，完全可以有多个 `elif` ，所以 `if` 语句的完整形式就是：

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

else:
    <执行4>

if 语句执行有个特点，它是从上往下判断，如果在某个判断上是 True ， 把该判断对应的语句执行后，就忽略掉剩下的 elif 和 else ， 所以下面的程序打印的是 teenager ：

In [83]:
```
age = 20
if age >= 6:
    print('teenager')
elif age >= 18:
    print('adult')
else:
    print('kid')
```

```
teenager
```

if 判断条件还可以简写，比如写：

In [84]:
```
if x:
    print('True')
```

```
True
```

只要 x 是非零数值、非空字符串、非空list等，就判断为 True ， 否则为 False 。

最后看一个有问题的条件判断。很多同学会用input()读取用户的输入，这样可以自己输入，程序运行得更有意思：

In [86]:
```
birth = input('year of birth: ')
if birth < 2000:
    print('00前')
else:
    print('00后')
```

```
year of birth: 2008
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-8dceb382635d> in <module>
      1 birth = input('year of birth: ')
----> 2 if birth < 2000:
      3     print('00前')
      4 else:
      5     print('00后')

TypeError: '<' not supported between instances of 'str' and 'int'
```

输入年份结果报错，这是因为 input() 返回的数据类型是 str ， **str 不能直接和整数比较**，必须先把 str 转换成整数。Python提供了 int() 函数来完成这件事情：

In [87]:
```
s = input('year of birth: ')
birth = int(s)
if birth < 2000:
    print('00前')
```

```
else:
    print('00后')
```

```
year of birth: 2008
00后
```

再次运行，就可以得到正确地结果。但是，如果输入 abc 呢？又会得到一个错误信息：

In [88]:
```
s = input('year of birth: ')
birth = int(s)
if birth < 2000:
    print('00前')
else:
    print('00后')
```

```
year of birth: abc
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-88-585697d71e65> in <module>
      1 s = input('year of birth: ')
----> 2 birth = int(s)
      3 if birth < 2000:
      4     print('00前')
      5 else:

ValueError: invalid literal for int() with base 10: 'abc'
```

原来int()函数发现一个字符串并不是合法的数字时就会报错，程序就退出了。

练习：if语句练习题 输入年份，判断是否是闰年

提示：

1. 能被400整除的年份
2. 能被4整除，但是不能被100整除的年份
3. 以上2种方法满足一种即为闰年

In [ ]:
```

```

循环

要计算1+2+3，我们可以直接写表达式： `1 + 2 + 3` ,但要计算1+2+3+...+10000，我们就需要循环语句。Python的循环有两种，一种是 `for...in` 循环，依次把list或tuple中的每个元素迭代出来;第二种循环是 `while` 循环，只要条件满足，就不断循环，条件不满足时退出循环。

`for...in`

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

执行这段代码，会依次打印names的每一个元素,所以 `for x in ...` 循环就是把每个元素代入变量x，然后执行缩进块的语句。

再比如我们想计算1-10的整数之和，可以用一个sum变量做累加：

In [ ]:
```
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print(sum)
```

如果要计算1-100的整数之和，从1写到100有点困难，幸好Python提供一个 range() 函数，可以生成一个整数序列，再通过list()函数可以转换为list。比如 range(11) 生成的序列是**从0开始**小于11的整数：

In [ ]:
```
list(range(11)) #注意该list包含0，共有11个元素
```

In [ ]:
```
sum = 0
for x in range(11): #用range(11)替换list
    sum = sum + x
print(sum)
```

while

while 循环，只要条件满足，就不断循环，条件不满足时退出循环。比如我们要计算100以内所有奇数之和，可以用 while 循环实现：

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

在循环内部变量n不断自减，直到变为-1时，不再满足while条件，循环退出。

break

在循环中， break 语句可以提前退出循环。例如，本来要循环打印1～10的数字：

In [ ]:
```
n = 1
while n <= 10:
    print(n)
    n = n + 1
print('END')
```

上面的代码可以打印出1~10。

如果要提前结束循环，可以用 break 语句：

In [ ]:
```
n = 1
while n <= 100:
    if n > 5: # 当n = 6时，条件满足，执行break语句
        break # break语句会结束当前循环
    print(n)
```

```
    n = n + 1
print('END')
```

continue

在循环过程中，也可以通过 continue 语句，跳过当前的这次循环，直接开始下一次循环。

In [ ]:
```
n = 0
while n < 5:
    n = n + 1
    print(n)
```

如果我们想只打印奇数，可以用 continue 语句跳过某些循环：

In [ ]:
```
n = 0
while n < 5:
    n = n + 1
    if n % 2 == 0: # 如果n是偶数，执行continue语句
        continue # continue语句会直接继续下一轮循环，后续的print()语句不会执行
    print(n)
```

循环是让计算机做重复任务的有效的方法。

break 语句可以在循环过程中直接退出循环，而 continue 语句可以提前结束本轮循环，并直接开始下一轮循环。这两个语句通常都必须配合if语句使用。

要特别注意，不要滥用 break 和 continue 语句。 break 和 continue 会造成代码执行逻辑分叉过多，容易出错。大多数循环并不需要用到 break 和 continue 语句，上面的两个例子，都可以试试改写循环条件或者修改循环逻辑，去掉 break 和 continue 语句。

练习题

小练习： 给定一个字符串, 通过循环将字符串反转，例如：输入'abc'，输出‘cba’

In [ ]:
```

```

小练习:打印1～20的整数中所有3的倍数

In [ ]:
```

```

break/continue小练习:

打印成绩表格中的所有学生分数

如果遇到不及格的分数则不打印,

如果遇到分数不在正常范围之内0-100,提示成绩异常

In [89]:
```
score = [68, 72, 85, 44, 38, 92, 87, 67, -32, 98, 88]
```

In [ ]:
```

```

dict和set

dict

Python内置了字典：dict的支持，dict全称dictionary，在其他语言中也称为map，使用键-值（key-value）存储，具有极快的查找速度。

举个例子，假设要根据同学的名字查找对应的成绩，如果用list实现，需要两个list：

In [90]:
```
names = ['Michael', 'Bob', 'Tracy']
scores = [95, 75, 85]
```

给定一个名字，要查找对应的成绩，就先要在names中找到对应的位置，再从scores取出对应的成绩，list越长，耗时越长。

如果用dict实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。用Python写一个dict如下：

In [91]:
```
d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
d['Michael']
```

Out[91]:
```
95
```

为什么dict查找速度这么快？因为dict的实现原理和查字典是一样的。假设字典包含了1万个汉字，我们要查某一个字，一个办法是把字典从第一页往后翻，直到找到我们想要的字为止，这种方法就是在list中查找元素的方法，list越大，查找越慢。

第二种方法是先在字典的索引表里（比如部首表）查这个字对应的页码，然后直接翻到该页，找到这个字。无论找哪个字，这种查找速度都非常快，不会随着字典大小的增加而变慢。

dict就是第二种实现方式，给定一个名字，比如'Michael'，dict在内部就可以直接计算出Michael对应的存放成绩的“页码”，也就是95这个数字存放的内存地址，直接取出来，所以速度非常快。

你可以猜到，这种key-value存储方式，在放进去的时候，必须根据key算出value的存放位置，这样，取的时候才能根据key直接拿到value。

把数据放入dict的方法，除了初始化时指定外，还可以通过key放入：

In [92]:
```
d['Adam'] = 67
d['Adam']
```

Out[92]:
```
67
```

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

In [93]:
```
d['Jack'] = 90
d['Jack']
```

Out[93]:
```
90
```

In [94]:
```
d['Jack'] = 88
d['Jack']
```

Out[94]: 88

如果key不存在，dict就会报错：

In [95]:
```
d
```

Out[95]:
```
{'Michael': 95, 'Bob': 75, 'Tracy': 85, 'Adam': 67, 'Jack': 88}
```

In [96]:
```
d['Thomas']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-96-bf27a9c462ee> in <module>
----> 1 d['Thomas']

KeyError: 'Thomas'
```

要避免key不存在的错误，有两种办法，一是通过in判断key是否存在：

In [97]:
```
'Thomas' in d
```

Out[97]:
```
False
```

二是通过dict提供的 `get()` 方法，如果key不存在，可以返回 `None` ，或者自己指定的value：

In [98]:
```
d.get('Thomas') #key不存在,返回None,返回None的时候Python的交互环境不显示结果。
```

In [99]:
```
d.get('Thomas', -1) #key不存在,返回-1
```

Out[99]:
```
-1
```

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

In [100…
```
d.pop('Bob')
```

Out[100…
```
75
```

In [101…
```
d
```

Out[101…
```
{'Michael': 95, 'Tracy': 85, 'Adam': 67, 'Jack': 88}
```

请务必注意，**dict内部存放的顺序和key放入的顺序是没有关系的。**

和list比较，dict有以下几个特点：

查找和插入的速度极快，不会随着key的增加而变慢； 需要占用大量的内存，内存浪费多。 而list相反：

查找和插入的时间随着元素的增加而增加； 占用空间小，浪费内存很少。 所以，dict是用空间来换取时间的一种方法。

dict可以用在需要高速查找的很多地方，在Python代码中几乎无处不在，正确使用dict非常重要，需要牢记的第一条就是dict的key必须是**不可变对象**。

这是因为dict根据key来计算value的存储位置，如果每次计算相同的key得出的结果不同，那dict内部就完全混乱了。这个通过key计算位置的算法称为哈希算法（Hash）。

要保证hash的正确性，作为key的对象就不能变。**在Python中，字符串、整数等都是不可变的，因此，可以放心地作为key。而list是可变的，就不能作为key**：

In [102…
```
key = [1, 2, 3]
d[key] = 'a list'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-102-bb63c40169bd> in <module>
      1 key = [1, 2, 3]
----> 2 d[key] = 'a list'

TypeError: unhashable type: 'list'
```

使用key-value存储结构的dict在Python中非常有用，选择不可变对象作为key很重要，最常用的key是字符串。

set

set和dict类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在set中，没有重复的key。

要创建一个set，需要提供一个list作为输入集合：

In [103…
```
s = set([1, 2, 3])
s
```

Out[103…
```
{1, 2, 3}
```

注意，传入的参数 `[1, 2, 3]` 是一个list，而显示的 `{1, 2, 3}` 只是告诉你这个set内部有1，2，3这3个元素，显示的顺序也不表示set是有序的。。

重复元素在set中自动被过滤：

In [104…
```
s = set([1, 1, 2, 2, 3, 3])
s
```

Out[104…
```
{1, 2, 3}
```

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

In [105…
```
s.add(4)
s
```

Out[105…
```
{1, 2, 3, 4}
```

In [106…
```
s.add(4)
s.add(4)
s
```

Out[106…
```
{1, 2, 3, 4}
```

In [107…
```
s.remove(4) #通过remove(key)方法可以删除元素
s
```

Out[107…
```
{1, 2, 3}
```

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

In [108…
```
s1 = set([1, 2, 3])
s2 = set([2, 3, 4])
```

In [109…
```
s1 & s2
```

Out[109…
```
{2, 3}
```

In [110…
```
s1 | s2
```

Out[110…
```
{1, 2, 3, 4}
```

set和dict的唯一区别仅在于没有存储对应的value，但是，set的原理和dict一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证set内部“不会有重复元素”。试试把list放入set，看看是否会报错。

再议不可变对象

上面我们讲了，str是不变对象，而list是可变对象。

对于可变对象，比如list，对list进行操作，list内部的内容是会变化的，比如：

In [111…
```
a = ['c', 'b', 'a']
a.sort()
a
```

Out[111…
```
['a', 'b', 'c']
```

而对于不可变对象，比如str，对str进行操作呢：

In [112…
```
a = 'abc'
a.replace('a', 'A')
```

Out[112…
```
'Abc'
```

In [113…
```
a
```

Out[113…
```
'abc'
```

虽然字符串有个 `replace()` 方法，也确实变出了 `'Abc'` ，但变量a最后仍是 `'abc'` ，应该怎么理解呢?

In [114…
```
help(a.replace)
```

```
Help on built-in function replace:

replace(old, new, count=-1, /) method of builtins.str instance
    Return a copy with all occurrences of substring old replaced by new.

      count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.
```

```
    If the optional argument count is given, only the first count occurrences are
    replaced.
```

我们先把代码改成下面这样：

In [115…
```
a = 'abc'
b = a.replace('a', 'A')
```

In [116…
```
b
```

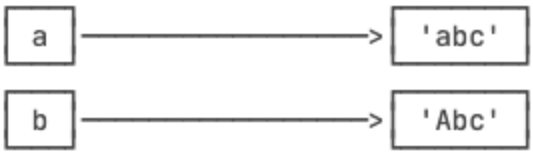
Out[116… `'Abc'`

In [117…
```
a
```

Out[117… `'abc'`

要始终牢记的是， `a` 是变量，而 `'abc'` 才是字符串对象！有些时候，我们经常说，对象 `a` 的内容是 `'abc'` ，但其实是指， `a` 本身是一个变量，它指向的对象的内容才是 `'abc'` ：

当我们调用 `a.replace('a', 'A')` 时，实际上调用方法 `replace` 是作用在字符串对象 `'abc'` 上的，而这个方法虽然名字叫 `replace` ，但却没有改变字符串 `'abc'` 的内容。相反， `replace` 方法创建了一个新字符串 `'Abc'` 并返回，如果我们用变量 `b` 指向该新字符串，就容易理解了，变量 `a` 仍指向原有的字符串 `'abc'` ，但变量b却指向新字符串 `'Abc'` 了：



所以，对于不变对象来说，调用对象自身的任意方法，也不会改变该对象自身的内容。相反，这些方法会创建新的对象并返回，这样，就保证了不可变对象本身永远是不可变的。

小练习

小练习：

有以下列表

L = ['最怕空气突然安静','最怕朋友突然的关心','最怕回忆','突然翻滚绞痛着','不平息','最怕忽然','听到你的消息']

M = ['想念如果会有声音','不愿那是悲伤的哭泣','终于让自己属于','我自己']

1. 将M列表中的歌词合并到L中,使歌词串联在一起.
2. 在列表中插入缺少的一句歌词,使顺序连续.
3. 查找列表中是否含有元素"最怕回忆"

4. 删除列表中的"最怕回忆"

5. 将歌词拼接成一个字符串

6. 将歌词翻转,使用join方法

In [118…
```
# 提示
movieName = ['加勒比海盗','骇客帝国','第一滴血','指环王','霍比特人','速度与激情']
del movieName[2] # del 关键字, 删除数据
```

In [119…
```
a = [1, 4, 2, 3]
a.reverse()
a
```

Out[119… `[3, 2, 4, 1]`

In [120…
```
a.sort(reverse=True)
```

In [121…
```
a
```

Out[121… `[4, 3, 2, 1]`

小应用: 使用for循环把列表中的元素拼接成字符串,使用逗号分隔

In [122…
```
names = ['raymond', 'rachel', 'matthew', 'roger',
         'betty', 'melissa', 'judith', 'charlie']
```

In [123…
```
################## 答案 ##########
# 空字符串用于拼接
S = ''
k = 1
for i in names:
    S += i
    # 在拼接一个逗号
    if k != len(names):# 说明是最后一次拼接
        S += ','

    k += 1
print(f"答案1结果: {S}")

# join方法
S2 = ','.join(names)
print(f"答案2结果: {S2}")
```

Out[123… `'raymond,rachel,matthew,roger,betty,melissa,judith,charlie'`

练习题:

dict = {"k1":"v1","k2":"v2","k3":"v3"}

1. 请循环遍历出所有的key
2. 请循环遍历出所有的value
3. 请循环遍历出所有的key和value,中间用冒号分割
4. 把所有value变成原来的值的两次重复,例如"v1"变成"v1v1"

In [ ]:

练习题:

有元组 T = ('A','B','E','F','A','B',"A",'B',"C")

T2 = ("B","E",'K','D')

去除元组T中的重复值,将结果赋值成Se

求Se 和 T2 的交集,并集

在Se中添加元素"Z"

清空Se

In [ ]:

函数

我们知道圆的面积计算公式为:

S = πr2

当我们知道半径r的值时，就可以根据公式计算出面积。假设我们需要计算3个不同大小的圆的面积:

In [ ]:
```
r1 = 12.34
r2 = 9.08
r3 = 73.1
s1 = 3.14 * r1 * r1
s2 = 3.14 * r2 * r2
s3 = 3.14 * r3 * r3
```

当代码出现有规律的重复的时候，你就需要当心了，每次写 `3.14 * x * x` 不仅很麻烦，而且，如果要把3.14改成3.14159265359的时候，得全部替换。

有了函数，我们就不再每次写 `s = 3.14 * x * x` ，而是写成更有意义的函数调用 `s = area_of_circle(x)`，而函数`area_of_circle` 本身只需要写一次，就可以多次调用。

基本上所有的高级语言都支持函数，Python也不例外。Python不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

调用函数

要调用一个函数，需要知道函数的名称和参数，比如求绝对值的函数abs，只有一个参数。可以直接从Python

的官方网站查看文档：

http://docs.python.org/3/library/functions.html#abs

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

In [ ]:
```
abs(-100.15)
```

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

In [ ]:
```
abs(1, 2)
```

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

In [ ]:
```
abs('a')
```

In [ ]:
```
max(1, 2,-3) #而max函数max()可以接收任意多个参数，并返回最大的那个
```

数据类型转换

Python内置的常用函数还包括数据类型转换函数，比如int()函数可以把其他数据类型转换为整数：

运行下面函数，观察下输出结果

```
int('123')
int(12.34)
float('12.34')
str(1.23)
bool(1)
bool('')
```

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

In [ ]:
```
a = abs # 变量a指向abs函数
a(-1) # 所以也可以通过a调用abs函数
```

定义函数

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

我们以自定义一个求绝对值的 `my_abs` 函数为例：

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

In [ ]:
```
my_abs(1-99.5)
```

请注意，函数体内部的语句在执行时，一旦执行到 return 时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有 return 语句，函数执行完毕后也会返回结果，只是结果为 None 。 return None 可以简写为 return 。

如果把my_abs()的函数单独保存为abstest.py文件，那么可以在该文件的当前目录下用 from abstest import my_abs 来导入my_abs()函数，注意abstest是文件名（不含.py扩展名）。当前了解函数可以单独保存后被调用即可，import的用法在后续章节中会详细介绍。

空函数

如果想定义一个什么事也不做的空函数，可以用 pass 语句：

In [ ]:
```
def nop():
    pass
```

pass 语句什么都不做，那有什么用？实际上 pass 可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个 pass ，让代码能运行起来。

pass 还可以用在其他语句里，比如：

In [ ]:
```
if age >= 18:
    pass
```

缺少了 pass ，代码运行就会有语法错误。

参数检查

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

In [ ]:
```
my_abs(1, 2)
```

但是如果参数类型不对，Python解释器就无法帮我们检查。试试 my_abs 和内置函数 abs 的差别：

In [ ]:
```
my_abs('A')
```

In [ ]:
```
abs('A')
```

当传入了不恰当的参数时，内置函数abs会检查出参数错误，而我们定义的 my_abs 没有参数检查，会导致if语句出错，出错信息和 abs 不一样。所以，这个函数定义不够完善。

让我们修改一下 my_abs 的定义，对参数类型做检查，只允许整数和浮点数类型的参数。数据类型检查可以用内置函数 isinstance() 实现：

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

```
    else:
        return -x
```

添加了参数检查后，如果传入错误的参数类型，函数就可以抛出一个错误：

In [ ]:
```
my_abs('A')
```

返回多个值

函数可以返回多个值，比如在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的坐标：

In [ ]:
```
import math #导入math包，用于后续代码引用math包里的sin、cos等函数。

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

然后，我们就可以同时获得返回值：

In [ ]:
```
x, y = move(100, 100, 60, math.pi / 6)
print(x, y)
```

但其实这只是一种假象，Python函数返回的仍然是单一值：

In [ ]:
```
r = move(100, 100, 60, math.pi / 6)
print(r)
```

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

备注：

定义时小括号中的参数，用来接收参数用的，称为 “形参”

调用时小括号中的参数，用来传递给函数用的，称为 “实参”

练习题

练习：编写一个函数

函数功能: 可以根据输入分数来输出不同的等级

学习成绩>=90分的同学返回 A ， 60-89分之间返回 B ， 60分以下返回 C 。

In [ ]:
```

```

函数的文档说明

In [125…
```
def test(a,b):
    """用来完成对2个数求和
```

```
    参数:
    a:我们的第一个参数用来相加
    b:这是相加的第二个参数
    return
    """
    return a + b
```

In [126…
```
# test?
```

In [127…
```
print(test.__doc__)
```

```
用来完成对2个数求和
    参数:
    a:我们的第一个参数用来相加
    b:这是相加的第二个参数
    return
```

函数等参数

注意：学习「函数等参数」这一小节时，不要使用碎片化时间，推荐集中时间专注阅读、理解

定义函数的时候，我们把参数的名字和位置确定下来，函数的接口定义就完成了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂逻辑被封装起来，调用者无需了解。

Python的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

位置参数

我们先写一个计算x2的函数：

In [128…
```
def power(x):
    return x * x
```

对于power(x)函数，参数x就是一个位置参数。

当我们调用power函数时，必须传入有且仅有的一个参数x：

In [129…
```
power(5)
```

Out[129…
```
25
```

现在，如果我们要计算x^3怎么办？可以再定义一个power3函数，但是如果要计算x^4、x^5.....怎么办？我们不可能定义无限多个函数。

你也许想到了，可以把 power(x) 修改为 power(x, n) ，用来计算x^n，说干就干：

In [130…
```
def power(x, n):
    s = 1
    while n > 0:
        n = n - 1
```

```
        s = s * x
    return s
```

对于这个修改后的power(x, n)函数，可以计算任意n次方：

In [131…
```
power(5, 3)
```

Out[131…
```
125
```

修改后的power(x, n)函数有两个参数：x和n，这两个参数都是位置参数，调用函数时，传入的两个值按照位置顺序依次赋给参数x和n。

默认参数

新的power(x, n)函数定义没有问题，但是，旧的调用代码失败了，原因是我们增加了一个参数，导致旧的代码因为缺少一个参数而无法正常调用：

In [132…
```
power(5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-132-4cd340f296c6> in <module>
----> 1 power(5)

TypeError: power() missing 1 required positional argument: 'n'
```

报错很清晰，少了参数n，由于我们经常计算x^2，所以，完全可以把第二个参数n的默认值设定为2：

In [133…
```
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

这样，当我们调用power(5)时，相当于调用power(5, 2)：

In [134…
```
power(5)
```

Out[134…
```
25
```

而对于n > 2的其他情况，就必须明确地传入n，比如 `power(5, 3)` 。

从上面的例子可以看出，默认参数可以简化函数的调用。设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；

二是如何设置默认参数。

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

使用默认参数有什么好处？最大的好处是能降低调用函数的难度。

举个例子，我们写个一年级小学生注册的函数，需要传入 `name` 和 `gender` 两个参数：

In [135…
```
def enroll(name, gender):
    print('name:', name)
    print('gender:', gender)
```

这样，调用enroll()函数只需要传入两个参数：

In [136…
```
enroll('Sarah', 'F')
```

```
name: Sarah
gender: F
```

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

In [137…
```
def enroll(name, gender, age=6, city='Beijing'):
    print('name:', name)
    print('gender:', gender)
    print('age:', age)
    print('city:', city)
```

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

In [138…
```
enroll('Sarah', 'F')
```

```
name: Sarah
gender: F
age: 6
city: Beijing
```

只有与默认参数不符的学生才需要提供额外的信息：

In [139…
```
enroll('Bob', 'M', 7)
```

```
name: Bob
gender: M
age: 7
city: Beijing
```

In [140…
```
enroll('Adam', 'M', city='Tianjin')
```

```
name: Adam
gender: M
age: 6
city: Tianjin
```

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

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用 `enroll('Bob', 'M', 7)` ，意思是，除了 `name` ， `gender` 这两个参数外，最后1个参数应用在参数 `age` 上， `city` 参数由于没有提供，仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用 `enroll('Adam', 'M', city='Tianjin')` ，意思是， `city` 参数用传进去的值，其他默认参数继续使用默

认值。

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个list，添加一个 END 再返回：

In [143…
```
def add_end(L=[]):
    L.append('END')
    return L
```

当你正常调用时，结果似乎不错：

In [144…
```
add_end([1, 2, 3])
```

Out[144…
```
[1, 2, 3, 'END']
```

In [145…
```
add_end(['x', 'y', 'z'])
```

Out[145…
```
['x', 'y', 'z', 'END']
```

当你使用默认参数调用时，一开始结果也是对的：

In [146…
```
add_end()
```

Out[146…
```
['END']
```

但是，再次调用add_end()时，结果就不对了：

In [147…
```
add_end()
```

Out[147…
```
['END', 'END']
```

In [148…
```
add_end()
```

Out[148…
```
['END', 'END', 'END']
```

很多初学者很疑惑，默认参数是 [] ，但是函数似乎每次都“记住了”上次添加了 'END' 后的list。

原因解释：

Python函数在定义的时候，默认参数L的值就被计算出来了，即 [] ，因为默认参数 L 也是一个变量，它指向对象 [] ，每次调用该函数，如果改变了L的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的 [] 了。

⚠️ 定义默认参数要牢记一点：默认参数最好指向不变对象！否则随着程序运行会出现很多奇怪的错误.

要修改上面的例子，我们可以用 None 这个不变对象来实现：

In [149…
```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
```

现在，无论调用多少次，都不会有问题：

In [150…
```
add_end()
add_end()
add_end()
```

Out[150…
```
['END']
```

为什么要设计 `str` 、 `None` 这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写函数时，参数如果可以设计一个不变对象，那就尽量设计成不变对象。

可变参数

在Python函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。

我们以数学题为例子，给定一组数字a，b，c……，请计算$a^2 + b^2 + c^2 + ……$。

要定义出这个函数，我们必须确定输入的参数。由于参数个数不确定，我们首先想到可以把a，b，c……作为一个list或tuple传进来，这样，函数可以定义如下：

In [151…
```
def calc(numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

但是调用的时候，需要先组装出一个list或tuple：

In [152…
```
calc([1, 2, 3])
```

Out[152…
```
14
```

In [153…
```
calc((1, 3, 5, 7))
```

Out[153…
```
84
```

如果利用可变参数，调用函数的方式可以简化成这样：

In [154…
```
# 函数的参数改为可变参数：
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

In [155…
```
calc(1, 2, 3)
```

Out[155…
```
14
```

In [156…
```
calc(1, 3, 5, 7)
```

Out[156…
```
84
```

定义可变参数和定义一个list或tuple参数相比，仅仅在参数前面加了一个 * 号。在函数内部，参数numbers接收到的是一个tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：

In [157…
```
calc()
```

Out[157…
```
0
```

如果已经有一个list或者tuple，要调用一个可变参数怎么办？可以这样做：

In [158…
```
nums = [1, 2, 3]
calc(nums[0], nums[1], nums[2])
```

Out[158…
```
14
```

这种写法当然是可行的，问题是太繁琐，所以Python允许你在list或tuple前面加一个*号，把list或tuple的元素变成可变参数传进去：

In [159…
```
calc(*nums)
```

Out[159…
```
14
```

*nums 表示把 nums 这个list的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

小练习

使用可变参数，写一个函数，不使用python自带的 sum 函数，计算一个列表的加和：

In [ ]:
```

```

关键字参数

可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。请看示例：

In [160…
```
def person(name, age, **kw):
    print('name:', name, 'age:', age, 'other:', kw)
```

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

In [161…
```
person('Michael', 30)
```

```
name: Michael age: 30 other: {}
```

也可以传入任意个数的关键字参数：

In [162…
```
person('Bob', 35, city='Beijing')
```

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

In [163…
```
person('Adam', 45, gender='M', job='Engineer')
```

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

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

和可变参数类似，也可以先组装出一个dict，然后，把该dict转换为关键字参数传进去：

In [164…
```
extra = {'city': 'Beijing', 'job': 'Engineer'}
```

In [165…
```
person('Jack', 24, city=extra['city'], job=extra['job'])
```

```
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

当然，上面复杂的调用可以用简化的写法：

In [166…
```
extra = {'city': 'Beijing', 'job': 'Engineer'}
person('Jack', 24, **extra)
```

```
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

**extra 表示把 extra 这个dict的所有key-value用关键字参数传入到函数的 **kw 参数， kw 将获得一个dict，注意 kw 获得的dict是 extra 的一份拷贝，对 kw 的改动不会影响到函数外的 extra 。

命名关键字参数

对于关键字参数，函数的调用者可以传入任意不受限制的关键字参数。至于到底传入了哪些，就需要在函数内部通过 kw 检查。

仍以 person() 函数为例，我们希望检查是否有 city 和 job 参数：

In [167…
```
def person(name, age, **kw):
    if 'city' in kw:
        # 有city参数
        pass
    if 'job' in kw:
        # 有job参数
        pass
    print('name:', name, 'age:', age, 'other:', kw)
```

但是调用者仍可以传入不受限制的关键字参数：

In [168…
```
person('Jack', 24, city='Beijing', addr='Chaoyang', zipcode=123456)
```

```
name: Jack age: 24 other: {'city': 'Beijing', 'addr': 'Chaoyang', 'zipcode': 123456}
```

如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收 city 和 job 作为关键字参数。这种方式定义的函数如下：

In [169…
```
def person(name, age, *, city, job):
    print(name, age, city, job)
```

和关键字参数 **kw 不同，命名关键字参数需要一个特殊分隔符 * ， * 后面的参数被视为命名关键字参数。

调用方式如下：

In [170…
```
person('Jack', 24, city='Beijing', job='Engineer')
```

```
Jack 24 Beijing Engineer
```

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符 * 了：

In [171…
```
def person(name, age, *args, city, job):
    print(name, age, args, city, job)
```

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错：

In [172…
```
person('Jack', 24, 'Beijing', 'Engineer')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-172-e7e3c3c0aaca> in <module>
----> 1 person('Jack', 24, 'Beijing', 'Engineer')

TypeError: person() missing 2 required keyword-only arguments: 'city' and 'job'
```

由于调用时缺少参数名 `city` 和 `job` ，Python解释器把前两个参数视为位置参数，后两个参数传给 `*args` ，但缺少命名关键字参数导致报错。

命名关键字参数可以有缺省值，从而简化调用：

In [173…
```
def person(name, age, *, city='Beijing', job):
    print(name, age, city, job)
```

由于命名关键字参数 `city` 具有默认值，调用时，可不传入 `city` 参数：

In [174…
```
person('Jack', 24, job='Engineer')
```

```
Jack 24 Beijing Engineer
```

使用命名关键字参数时，要特别注意，如果没有可变参数，就必须加一个 * 作为特殊分隔符。如果缺少 * ，Python解释器将无法识别位置参数和命名关键字参数：

In [175…
```
def person(name, age, city, job):
    # 缺少 *, city和job被视为位置参数
    pass
```

参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这5种参数都可以组合使用。但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数、命名关键字参数和关键字参数。

比如定义一个函数，包含上述若干种参数：

In [176…
```
def f1(a, b, c=0, *args, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)
```

```
def f2(a, b, c=0, *, d, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)
```

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

In [177…
```
f1(1, 2)
```

```
a = 1 b = 2 c = 0 args = () kw = {}
```

In [178…
```
f1(1, 2, c=3)
```

```
a = 1 b = 2 c = 3 args = () kw = {}
```

In [179…
```
f1(1, 2, 3, 'a', 'b')
```

```
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
```

In [180…
```
f1(1, 2, 3, 'a', 'b', x=99)
```

```
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
```

In [181…
```
f2(1, 2, d=99, ext=None)
```

```
a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}
```

通过一个tuple和dict，你更加方便的传入更重参数，调用上述函数：

In [183…
```
args = (1, 2, 3, 4)
kw = {'d': 99, 'x': '#'}
f1(*args, **kw)
```

```
a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}
```

In [184…
```
args = (1, 2, 3)
kw = {'d': 88, 'x': '#'}
f2(*args, **kw)
```

```
a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}
```

所以，对于任意函数，都可以通过类似 `func(*args, **kw)` 的形式调用它，无论它的参数是如何定义的。

⚠️虽然可以组合多达5种参数，但不要同时使用太多的组合，否则函数接口的可理解性很差。

小结

Python的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。

默认参数一定要用不可变对象，如果是可变对象，程序运行时会有逻辑错误！

要注意定义可变参数和关键字参数的语法：

`*args` 是可变参数， `args` 接收的是一个tuple；

`**kw` 是关键字参数， `kw` 接收的是一个dict。

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入： `func(1, 2, 3)`，又可以先组装list或tuple，再通过 `*args` 传入： `func(*(1, 2, 3))`；

关键字参数既可以直接传入： `func(a=1, b=2)`，又可以先组装dict，再通过 `**kw` 传入： `func(**{'a': 1, 'b': 2})`。

使用 `*args` 和 `**kw` 是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。

定义命名的关键字参数在没有可变参数的情况下不要忘了写分隔符*，否则定义的将是位置参数。

练习题

`mul` 函数允许计算两个数的乘积，请稍加改造，变成可接收一个或多个数并计算乘积：

In [186…
```
def mul(x, y):
    return x * y
```

In [ ]:
```

```

递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

举个例子，我们来计算阶乘 `n! = 1 x 2 x 3 x ... x n`，用函数 `fact(n)` 表示，可以看出：

$$fact(n)=n!=1\times2\times3\times\cdots\times(n-1)\times n=(n-1)!\times n=fact(n-1)\times n$$

所以， `fact(n)` 可以表示为 `n x fact(n-1)`，只有n=1时需要特殊处理。

于是， `fact(n)` 用递归的方式写出来就是：

In [187…
```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

上面就是一个递归函数。可以试试：

In [188…
```
fact(5)
```

Out[188…
```
120
```

如果我们计算fact(5)，可以根据函数定义看到计算过程如下：

```
===> fact(5)
===> 5 * fact(4)
===> 5 * (4 * fact(3))
===> 5 * (4 * (3 * fact(2)))
===> 5 * (4 * (3 * (2 * fact(1))))
===> 5 * (4 * (3 * (2 * 1)))
```

```
===> 5 * (4 * (3 * 2))
===> 5 * (4 * 6)
===> 5 * 24
===> 120
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试 `fact(10000)`：

In [189…

```
# fact(10000) # 如果没有报错就把数值再调大点
```

解决递归调用栈溢出的方法是通过尾递归优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指，在函数返回的时候，调用自身本身，并且，return语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的 `fact(n)` 函数由于 `return n * fact(n - 1)` 引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

In [190…

```
def fact(n):
    return fact_iter(n, 1)

def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

可以看到， `return fact_iter(num - 1, num * product)` 仅返回递归函数本身， `num - 1` 和 `num * product` 在函数调用前就会被计算，不影响函数调用。

`fact(5)` 对应的 `fact_iter(5, 1)` 的调用如下：

```
===> fact_iter(5, 1)
===> fact_iter(4, 5)
===> fact_iter(3, 20)
===> fact_iter(2, 60)
===> fact_iter(1, 120)
===> 120
```

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的 `fact(n)` 函数改成尾递归方式，也会导致栈溢出。

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。

Python标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

递归这个部分，初期理解到位很重要，暂时不会写没关系，推荐中期接触数据结构与算法时，把分治法、动态规划、贪心算法结合起来学习，想尝试练习的可以试试利用递归算法通过python实现获得指定项的斐波拉契数列。

局部变量与全局变量

局部变量

在函数内部定义的变量

In [191…

```
a = 200
def test1():
    a = 300
    print("在test1中a=", a)

def test2():
    a = 400
    print("在test2中a=", a)

test1()
test2()
print(a)
```

```
在test1中a= 300
在test2中a= 400
200
```

In [192…

```
def test1():
    m = 300  #这里的m就是局部变量
    return m+1
```

In [193…

```
m #报错，局部变量只在函数内使用
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-193-79bd9ad14f3a> in <module>
----> 1 m #报错，局部变量只在函数内使用

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

不同的函数，可以定义相同的名字的局部变量，但是各用个的不会产生影响

局部变量的作用，为了临时保存数据需要在函数中定义变量来进行存储，这就是它的作用

全局变量

如果一个变量，既能在一个函数中使用，也能在其他的函数中使用，这样的变量就是全局变量

In [194…

```
# 定义全局变量
a = 100
```

```
def test1():
    print(a)

def test2():
    b = 1000

    print(a+b)

# 调用函数
test1()
test2()
```

```
100
1100
```

全局变量和局部变量名字相同问题

In [195…

```
# 定义全局变量
a = 100
def test1():
    a = 300
    print("test1中a是: ", a)
def test2():
    print("test2中a是: ", a)
test1()
test2()
print(a)
```

```
test1中a是:  300
test2中a是:  100
100
```

修改全局变量

一个自然而然的需求, 能不能在函数内部修改全局变量呢?

In [196…

```
# 定义全局变量
a = 100
def test1():
    global a #声明为全局变量
    a = 300
    print("test1中a是: ", a)
def test2():
    print("test2中a是: ", a)
test1()
test2()
print(a)
```

```
test1中a是:  300
test2中a是:  300
```

```
300
```

总结一下:

1. 在函数外边定义的变量叫做全局变量
2. 全局变量能够在所有的函数中进行访问
3. 如果在函数中修改全局变量，那么就需要使用global进行声明，否则出错
4. 如果全局变量的名字和局部变量的名字相同，那么使用的是局部变量的，小技巧：强龙不压地头蛇

小练习

1. 写函数实现功能：传入一个列表，如果列表长度超过5，返回列表前5个元素组成的列表，如果长度小于5，使用0补齐元素，使得列表长度，并返回结果。

In [197…

```
l = [4, 5, 6, 7]

# return [4,5,6,7,0]

l = [1,2,3,4,5,6,7]

# return [1,2,3,4,5]
```

In [199…

```
############ 答案完成后再看：） #############
def func(x):
    return x[:5] + [0] * (5-min(len(x), 5))
```

1. 使用 while 循环，构造一个1, 3, 5, 7, ..., 99的列表.

In [200…

```
############ 答案完成后再看：） #############
L = []
n = 1
while n <= 99:
    L.append(n)
    n = n + 2

print(L)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 4
7, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91,
93, 95, 97, 99]
```