目　录

引入

0.1 版本需求

Python 为 3.9 版本，自 3.4 以来改动的语法可忽略不计，除非更新到 4.0。

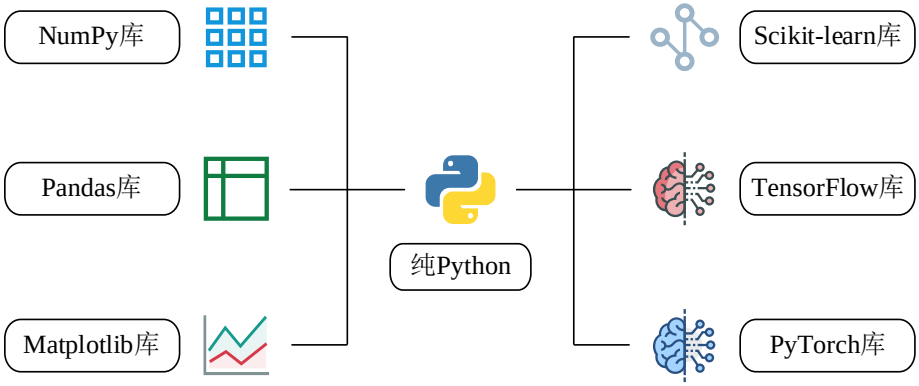
0.2 视频特点

- 本视频分辨率为 1080P，请调高分辨率；
- 视频简介与置顶评论中均附有讲义链接，此讲义为原创，请勿商用。
- 适宜人群：有一定 C 语言或 Matlab 编程基础的同学。

0.3 视频 UP 主

- UP 的本科为三峡大学（原电力部 6 所直属高校之一，超强电气型），硕士是中南大学（计算机、自动化、临床、护理等热门专业均属 A 类学科）。
- 如果课件中有纰漏，请在视频评论区反馈。

0.4 深度学习的相关库



① NumPy 包为 Python 加上了关键的数组变量类型，弥补了 Python 的不足；
② Pandas 包在 NumPy 数组的基础上添加了与 Excel 类似的行列标签；
③ Matplotlib 库借鉴 Matlab，帮 Python 具备了绘图能力，使其如虎添翼；
④ Scikit-learn 库是机器学习库，内含分类、回归、聚类、降维等多种算法；
⑤ TensorFlow 库是 Google 公司开发的深度学习框架，于 2015 年问世；
⑥ PyTorch 库是 Facebook 公司开发的深度学习框架，于 2017 年问世。

0.5 深度学习的基本常识

- **人工智能**是一个很大的概念，其中一个最重要的分支就是机器学习；
- **机器学习**的算法多种多样，其中最核心的就是神经网络；
- **神经网络**的隐藏层若足够深，就被称为深层神经网络，也即深度学习；
- **深度学习**包含深度神经网络、卷积神经网络、循环神经网络等。

一、变量类型与输出语句

1.1 编程基础

- 敲代码时切换到英文输入法，所有符号均为英文状态；
- Python 的注释以 # 开头为标志，注释单独成行，或放在某语句右侧；
- Python 是动态输入类型的语言，像 Matlab 一样，变量类型是动态推断的；
  静态类型的 C 语言须声明变量类型，如 int a=1，而 Python 只需要 a=1；
- 编程语言中的 a=1 的含义是——将数值 1 赋给变量 a；
- Python 里换行符（回车）可以替代分号（;），所以一般不出现分号；
- Python 里四个空格是一个缩进，不要随意在某行代码的开头输入空格。

1.2 变量类型

- 像 C 语言和 Matlab 一样，变量名由字母、数字、下划线组成（但不能以数字开头），字母区分大小写，变量名不能与内置的函数同名。
- 根据变量是否可以充当容器，将变量类型分为基本类型和高级类型。
  基本变量类型：字符串、数字、布尔型；
  高级变量类型：集合、元组、列表、字典。

七种变量类型，示例如下。

In [1]:
```
# 字符串（str）
str_v = "a real man"
```

In [2]:
```
# 数字（int 或 float）
num_v = 415411
```

In [3]:
```
# 布尔型（bool）
bool_v = True
```

In [4]:
```
# 集合（set）
set_v = {1, 2, 3, 1}
```

In [5]:
```
# 元组（tuple）
tuple_v = (1, 2, 3)
```

In [6]:
```
# 列表（list）
list_v = [1, 2, 3]
```

In [7]:
```
# 字典（dict）
dict_v = {'a':1, 'b':2 , 'c':3 }
```

1.3 输出语句

Python 语言的标准输出方法是：print(变量名)。

Jupyter 中增加一种独特的输出方法，即在一个代码块内的最后一行写上变量名，即可输出该变量的数值。

以上两种输出方法完全等效，如示例所示。

```
In [1]:
# 字符串
str_v = "a really man"
str_v
Out [1]: 'a really man'

In [1]:
# 字符串
str_v = "a really man"
print(str_v)
a really man

In [2]:
# 数字
num_v = 415411
num_v
Out [2]: 415411

In [2]:
# 数字
num_v = 415411
print(num_v)
415411

In [3]:
# 布尔型
bool_v = Ture
bool_v
Out [3]: True

In [3]:
# 布尔型
bool_v = Ture
print(bool_v)
True

In [4]:
# 集合
set_v = {1, 2, 3, 1}
set_v
Out [4]: {1, 2, 3}

In [4]:
# 集合
set_v = {1, 2, 3, 1}
print(set_v)
{1, 2, 3}

In [5]:
# 元组
tuple_v = (1, 2, 3)
tuple_v
Out [5]: (1, 2, 3)

In [5]:
# 元组
tuple_v = (1, 2, 3)
print(tuple_v)
(1, 2, 3)

In [6]:
# 列表
list_v = [1, 2, 3]
list_v
Out [6]: [1, 2, 3]

In [6]:
# 列表
list_v = [1, 2, 3]
print(list_v)
[1, 2, 3]

In [7]:
# 字典
dict_v = {'a':1, 'b':2 , 'c':3 }
dict_v
Out [7]: {'a': 1, 'b': 2, 'c': 3}

In [7]:
# 字典
dict_v = {'a':1, 'b':2 , 'c':3 }
print(dict_v)
{'a': 1, 'b': 2, 'c': 3}
```

这里你可以暂时不理解变量之间的区别，但必须理解一点：print 函数的原理是**输出仅一个单独的变量**，它只有这一种用法，其它的各种用法要么是输出了 f 字符串，要么是输出了元组，后续讲到字符串或元组时会重点提到输出语句。

二、基本变量类型

2.1 字符串

（1）字符串的结构

字符串用引号括起来，双引号和单引号都可以。示例代码如下。

In [1] :
```
str1 = 'A real man, he knows what he needs to do.'
str1
```
Out [1] : 'A real man, he knows what he needs to do.'

In [2] :
```
str2 = "A real man, he knows what he needs to do."
str2
```
Out [2] : 'A real man, he knows what he needs to do.'

当字符串变量内含有单引号，就用双引号来表示该字符串；

当字符串变量内含有双引号，就用单引号来表示该字符串；

（2）输出语句的经典用法

想在字符串中插入其它变量，可使用“f 字符串”的方法，代码示例如下。

In [1] :
```
str1 = "money path"
str2 = "doctor"
```

In [2] :
```
str3 = f"You ruined his {str1}. You ruined his {str2}."
str3
```
Out [2] : 'You ruined his Money Path. You ruined his doctor.'

正是有了 f 字符串，才能实现如下输出，这也是 print 最常见的使用场景。

In [3] :
```
answer = 0.98      # 经过一系列运算，测试集给出准确率为 98%
print(f"测试集的准确率为: {answer}")
```
测试集的准确率为: 0.98

注意，这里的 print 依然只输出了一个单独的变量，即一个单独的 f 字符串。

（3）输出语句的转义字符

在字符串中添加转义字符，如换行符\n 与制表符\t，可增加 print 的可读性。

同时，转义字符只有在 print 里才能生效，单独对字符串使用无效。

In [1] :
```
# 构建字符串
message = "Shop sells:\n\tlitchi, \n\tfritters, \n\tfried fish."
```

In [2] :
```
# 单独输出字符串
message
```
Out [2] : 'Shop sells:\n\tlitchi, \n\tfritters, \n\tfried fish.'

In [3] :
```
# 转义字符在 print 里生效
print(message)
```
```
Shop sells:
	litchi,
	fritters,
	fried fish.
```

2.2 数字

（1）整数型数字和浮点型数字

数字有两种数据类型，分别是整数（int）和浮点数（float）。这里暂时不用太过区分二者，进入下一节课《NumPy 数组库》时才要注意区分。

（2）常用运算符

常用运算符如表 2-1 所示。

表 2-1 常见的数字运算符

运算符	含义	输入	输出
+、-、*、/	加、减、乘、除	1*2+3/4	2.75
**	幂	2**4	16
()	修正运算次序	1*(2+3)/4	1.25
//	取整	28//5	5
%	取余	28%5	3

2.3 布尔型

布尔型只有两个值（True 和 False），通常是基于其它变量类型来进行生成。

（1）基于基本变量类型生成

- 对字符串作比较，使用等于号==与不等号!=；
- 对数字作比较，使用大于>、大于等于>=、等于==、小于<、小于等于<=。

In [1]:
```
# 字符串——检查某字符串的值
str_v = 'cxk'
print(str_v == 'chicken')
print(str_v != 'chicken')
```
```
False
True
```

In [2]:
```
# 数字——检查某数字是否在某范围
num_v = 3
print(num_v > 5)
print(num_v == 5)
print(num_v < 5)
```
```
False
False
True
```

（2）基于高级变量类型生成

In [3]:
```
# 集合——检查某变量是否在该集合中
set_v = {1, 2, 3}
print(2 in set_v)
print(2 not in set_v)
```
```
True
False
```

In [4]:
```
# 元组——检查某变量是否在该元组中
tuple_v = (1, 2, 3)
print(2 in tuple_v)
print(2 not in tuple_v)
```
```
True
False
```

In [5]:
```
# 列表——检查某变量是否在该列表中
list_v = [1, 2, 3]
print(2 in list_v)
print(2 not in list_v)
```
```
True
False
```

In [6]:
```
# 字典——检查某变量是否在该字典中
dict _v = {'a':1, 'b':2, 'c':3}
print( 2 in dict _v.values() )
print( 2 not in dict _v.values() )     # 字典的.values()方法见 3.5 小节
```
```
True
False
```

（3）同时检查多个条件

and 的规则是，两边全为 **True** 则为 **True**，其它情况均为 **False**；

or 的规则是，两边有一个是 **True** 则为 **True**，其他情况为 **False**。

In [1] :
```
# 先产生两个布尔值
T = True
F = False
```

In [2] :
```
# and 示例
T and F
```

Out [2] : False

In [3] :
```
# or 示例
T or F
```

Out [3] : True

除了 **and** 和 **or**，还在一个布尔值前面加上 **not**，如 **not True** 就是 **False**。

2.4 判断语句

bool 值通常作为 **if** 判断的条件，**if** 判断的语法规则为

```
if 布尔值:
    情况一
elif 布尔值:
    情况二
else:
    其它情况
```

注意事项：

- Python 的循环、判断、函数和类中均不使用 end 来表示代码块的结束；
  Python 常常利用缩进（即四个空格）来表示代码块的范围；
  每一个判断条件的最后有一个冒号，不要遗漏。
- **if** 语句中，**if**:只出现 1 次，**elif**:可出现 0 至∞次，**else**:可出现 0 或 1 次。

示例如下。

In [1] :
```
bool1 = False
bool2 = False
bool3 = False
if bool1:
    print('当 bool1 为 True，此行将被执行')
elif bool2:
    print('否则的话，当 bool2 为 True，此行将被执行')
elif bool3:
    print('否则的话，当 bool3 为 True，此行将被执行')
else:
    print('否则的话，此行将被执行')
```

当以上条件测试均不满足，此行将被执行。

2.5 基本变量间的转换

字符串、整数、浮点数、布尔型四者之间可以无缝切换。

- 转换为字符串使用 str 函数；
- 转换为整数型数字使用 int 函数；
- 转换为浮点型数字使用 float 函数；
- 转换为布尔型使用 bool 函数。

示例如下。

In [1]:
```
# 定义变量
str_v = '123'
int_v = 123
float_v = 123.0
bool_v = True
```

In [2]:
```
# 转化为字符串
print( str( int_v   ) )
print( str( float_v ) )
print( str( bool_v ) )
```

```
123
123.0
True
```

In [3]:
```
# 转化为整数型变量
print( int( str_v   ) )
print( int( float_v ) )
print( int( bool_v ) )
```

```
123
123
1
```

In [4]:
```
# 转化为浮点型变量
print( float( str_v   ) )
print( float( float_v ) )
print( float( bool_v ) )
```

```
123.0
123.0
1.0
```

In [5]:
```
# 转化为布尔型变量
print( bool( str_v   ) )
print( bool( int_v   ) )
print( bool( float_v ) )
```

```
True
True
True
```

注：其它变量转为布尔型变量时，只有当字符串为空、数字为 0、集合为空、元组为空、列表为空、字典为空时，结果才为 False。

三、高级变量类型

高级变量类型，即集合、元组、列表、字典，它们有一个共同的特点：作为容器，它们可以随意容纳任意变量（甚至在同一个容器内包含 7 种变量类型）。

3.1 集合

集合是无序的、不可重复的元素的组合。

可以用两种方式创建集合：通过 set 函数或使用大括号，示例如下。

```
In [1]: # 使用 set() 函数将列表转化为集合
        set([9,1,4,1,3,1])
Out [1]: {1, 3, 4, 9}

In [2]: # 使用大括号创建
        {'中','南','大','学','与','湖','南','大','学','是','带','专'}
Out [2]: {'与', '专', '中', '南', '大', '学', '带', '是', '湖'}
```

注意，请勿用大括号创建空集合，否则会被误认为是字典。

集合出现次数稀少，它更多的是被看作是字典的索引（即数据的标签）。考虑到后面我们会学习 Pandas 库，可替代集合，因此这里就不再讨论集合了，以后在别人代码里遇到了知道是集合就可以。

3.2 元组

元组（UP 最爱），是一个超厉害的变量类型（请不要将之与列表一同提起）。

（1）创建元组

有两种方式可以创建元组，一种是规范括号法，一种是省略括号法。

In [1]：
```
# 规范的括号法
(1, 2, 3)
```
Out [1]： (1, 2, 3)

In [2]：
```
# 省略括号法（核心）
1, 2, 3
```
Out [2]： (1, 2, 3)

（2）输出语句中的元组法

高级变量类型都可以容纳所有的变量类型，如示例所示。

In [1]：
```
# 一个释放自我的元组
'a', 1, True, {1,2,3},(1,2,3),[1,2,3],{'a':1, 'b':2, 'c':3}
```
Out [1]： ('a', 1, True, {1, 2, 3}, (1, 2, 3), [1, 2, 3], {'a': 1, 'b': 2, 'c': 3})

因此，如果你在 print 里忽然见到逗号，不必大惊小怪，示例如下。

In [2]：
```
# 元组法替代 f 字符串
answer = 98
print(f'最终答案为:{answer}')      # f 字符串法
print('最终答案为:', answer)       # 元组法
```
最终答案为:98
最终答案为: 98

元组法输出相对于 f 字符串输出有个缺点，即输出的元素之间含有一个空格。

（3）元组拆分法

In [1]：
```
# 元组拆分法——极速创建新变量
a,b,c = 1,2,3
print(c,b,a)
```
3 2 1

In [2]：
```
# 元组拆分法——极速交换变量值
a,b = 1,2
b,a = a,b
print(a,b)
```
2 1

In [3]：
```
# 元组拆分法——只要前两个答案
values = 98, 99, 94, 94, 90, 92
a, b, *rest = values
a, b, rest
```
Out [3]： (98, 99, [94, 94, 90, 92])

希望你可以跟我一起感受到元组带来的快乐！

3.3 列表

（1）创建列表

列表由若干个有序的变量组成，其中的元素之间可以无任何关系。

列表出现的标志是中括号，列表里的变量使用逗号分隔，示例如下。

In [1]：
```
# 一个全是字符串的列表
list1 = ['Bro.chicken', '30 months', 'Marry Pd']
list1
```
Out [1]：['Bro.chicken', '30 months', 'Marry Pd']

In [2]：
```
# 一个全是数字的列表
list2 = [11, 45, 14]
list2
```
Out [2]：[11, 45, 14]

In [3]：
```
# 一个释放自我的列表
list3 = ['cxk', 666, True, set([1,2,3]), (1,2,3), [1,2,3], {'a':1, 'b':2, 'c':3}]
list3
```
Out [3]：['cxk', 666, True, {1, 2, 3}, (1, 2, 3), [1, 2, 3], {'a': 1, 'b': 2, 'c': 3}]

由 list3 可见，列表可以容纳各种变量类型，其代价是——列表要单独存储每一个元素的变量类型，列表越大越占空间。但是请仔细想想，深度学习真会出现类似 list3 的这种列表吗？我想不会。

下一节课我们将介绍 NumPy 库，其中的 NumPy 数组仅接纳一种变量类型。

（2）访问与修改某个元素

访问列表元素时使用中括号，索引由 0 开始，示例如下。

In [4]：
```
list3 = ['cxk', 666, True, set([1,2,3]), (1,2,3), [1,2,3], {'a':1}]
print(list3[0])
print(list3[5])
```
```
cxk
[1, 2, 3]
```

当想访问列表倒数第一个元素时，可使用 a[-1]；当想访问倒数第二个元素，可使用 a[-2]；以此类推。代码示例如下。

In [5]：
```
list4 = ['a', 'b', 'c']
print( ( list4[-3] , list4[-2] , list4[-1] ) )
```
```
('a', 'b', 'c')
```

可以通过访问某列表元素的方式对其数值进行修改。

In [6]：
```
list4 = ['a', 'b', 'c']
list4[-1] = 5
list4
```
Out [6]：['a', 'b', 5]

（3）切片——访问部分元素

切片，就是列表的一部分。在学习列表索引时，可结合图 3-1 所示。

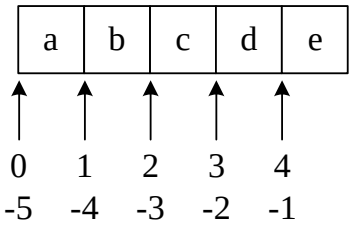


图 3-1 索引负责其指向区域的右侧一个单元格

当明确知道从第 x 个元素切到第 y 个元素，示例为

In [1]:
```
list_v = ['a', 'b', 'c', 'd', 'e']
print( list_v )
print( list_v[ 1 : 4 ] )     # 从索引[1]开始，切到索引[4]之前
print( list_v[ 1 :   ] )     # 从索引[1]开始，切到结尾
print( list_v[   : 4 ] )     # 从列表开头开始，切到索引[4]之前
```

```
['a', 'b', 'c', 'd', 'e']
['b', 'c', 'd']
['b', 'c', 'd', 'e']
['a', 'b', 'c', 'd']
```

当明确切除列表的开头与结尾，如示例所示。

In [2]:
```
list_v = ['a', 'b', 'c', 'd', 'e']
print( list_v )
print( list_v[ 2 : -2 ] )     # 切除开头 2 个和结尾 2 个
print( list_v[   : -2 ] )     # 切除结尾两个
print( list_v[ 2 :   ] )      # 切除开头两个
```

```
['a', 'b', 'c', 'd', 'e']
['c']
['a', 'b', 'c']
['c', 'd', 'e']
```

当明确隔几个元素采样一次时，示例如下。

In [3]:
```
list_v = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
print( list_v[   :   : 2 ] )     # 每 2 个元素采样一次
print( list_v[   :   : 3 ] )     # 每 3 个元素采样一次
print( list_v[ 1 : -1 : 2 ] )    # 切除一头一尾后，每 2 个元素采样一次
```

```
['a', 'c', 'e', 'g']
['a', 'd', 'g']
['b', 'd', 'f']
```

值得注意的是，对列表进行切片后得到一个新的对象，与原列表变量相互独立，如示例所示。但是若存储几百万条数据，这无疑对电脑是一种全新的考验。

In [1]:
```
# 创建 list_v 的切片 cut_v
list_v = [1, 2, 3]
cut_v = list_v[ 1 :]
cut_v
```
Out [1]: [2, 3]

In [2]:
```
# 修改 cut_v 的元素
cut_v[1] = 'a'
cut_v
```
Out [2]: [2, 'a']

In [3]:
```
# 输出 list_v，其不受切片影响
list_v
```
Out [3]: [1, 2, 3]

为此，NumPy 的切片被设定为是原对象的一个视图，不会在内存中创建一个新对象，改变 NumPy 切片上的对象会影响原 NumPy 数组，因此其比 Matlab 更 6。那 NumPy 真想提取部分切片用于备份怎么办?届时使用 .copy() 方法即可。

（4）列表元素的添加

列表可以使用 + 和 * 来添加原列表，示例如下。

In [1]:
```
list1 = [1,2,3]
print( list1 + [4] )            # 列表尾部添加一个元素
print( list1 + [10,11,12] )     # 与另一个列表连接
```
```
[1, 2, 3, 4]
[1, 2, 3, 10, 11, 12]
```

In [2]:
```
print( list1 * 2 )      # 复制两倍的自己
print( list1 * 4 )      # 复制四倍的自己
```
```
[1, 2, 3, 1, 2, 3]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

从这里可以看出，列表完全没有任何数组的特征，加减乘除油盐不进。看来真的只能把列表看作为一个容器而已。

在 Numpy 中如何实现数组元素添加的功能呢？使用 numpy.concatenate 函数。基于此，NumPy 的加减乘除得到解放，可以像 Matlab 一样对数组做数学运算。

3.4 字典

（1）创建字典

字典可以理解为升级版的列表，每个元素的索引都可以自己定，示例如下。

In [1]:
```
list_v = [    90,    95,    100 ]
dict_v = { 'a':90, 'b':95, 'c':100 }
print( list_v[1] )
print( dict_v['b'] )
```
```
95
95
```

接下来创建一个与列表完全等效的特殊字典，如示例所示。

In [2]:
```
list_v = [    90,   95,   100 ]
dict_v = { 0: 90, 1: 95, 2: 100 }
print( list_v[1] )
print( dict_v[1] )
```
```
95
95
```

字典中的元素值可以是任何变量，如示例所示。

In [3]:
```
dict_v = {
      0: 'Chicken',
      1: 123,
      2: True,
      3: set([1,2,3]),
      4: (1,2,3),
      5: [1,2,3],
      6: {'a':1}
}

print ( dict_v[0], dict_v[1], dict_v[2], dict_v[3],
        dict_v[4], dict_v[5], dict_v[6])
```
```
Chicken  123  True  {1, 2, 3}  (1, 2, 3)  [1, 2, 3]  {'a': 1}
```

字典中的索引只能是数字或者字符串，且一般都是字符串，如示例所示。

In [4]:
```
dict_v = {
        'zero' : 123,
             1: True,
}
print( dict_v['zero'], dict_v[1])
```
```
123  True
```

用官方的说法，字典的索引叫做“键”，字典的索引值叫“值”。

（2）字典元素的修改、添加与删除

现以学科名称为字典索引，以学科实力为索引值，示例如下。

In [1]:
```
# 原字典（某大学对外招生宣传学科名单）
CSU = {
      '冶金工程': 'A+',
      '矿业工程': 'A+',
      '护理学': 'A+'
}
CSU
```

Out [1]:
```
{'冶金工程': 'A+',
 '矿业工程': 'A+',
 '护理学': 'A+'}
```

In [2]:
```
# 添加元素（添加热门专业）
CSU['计算机科学与技术'] = 'A-'
CSU['控制科学与工程'] = 'A-'
CSU['临床医学'] = 'A-'
CSU['交通运输'] = '双一流学科'
CSU['数学'] = '双一流学科'
CSU
```

Out [2]:
```
{'冶金工程': 'A+',
 '矿业工程': 'A+',
 '护理学': 'A+',
 '计算机科学与技术': 'A-',
 '控制科学与工程': 'A-',
 '临床医学': 'A-',
 '交通运输': '双一流学科',
 '数学': '双一流学科'}
```

In [3]:
```
# 删除元素
del CSU['冶金工程']
del CSU['矿业工程']
CSU
```

Out [3]:
```
{'护理学': 'A+',
 '计算机科学与技术': 'A-',
 '控制科学与工程': 'A-',
 '临床医学': 'A-',
 '交通运输': '双一流学科',
 '数学': '双一流学科'}
```

3.5 循环语句

（1）for 循环遍历列表

for 循环可以让循环变量依次取遍列表中的每个元素，其语法规则为

```
for 循环变量 in 列表:
    循环体
```

for 循环语句依托列表来进行循环，循环变量依次取遍列表中的元素。

示例如下。

In [1]:
```
# 先分开表扬，最后再说句额外的话
schools = ['中南大学', '湖南大学', '三峡大学', '长江大学']
for school in schools:
    message = f"{school}, you are a great school! "
    print(message)
print("I can't wait to visit you!")
```

Out [1]:
```
中南大学, you are a great school!
湖南大学, you are a great school!
三峡大学, you are a great school!
长江大学, you are a great school!
I can't wait to visit you!
```

（2）for 循环遍历字典

for 循环遍历字典时，既可以遍历索引，也可以遍历值，更可以都遍历。

In [1]:
```
# 两电一邮的 A+学科
schools={'成电':'电科、信通', '西电':'电科', '北邮':'信通'}
```

In [2]:
```
# 循环键
for k in schools.keys():
    print('两电一邮包括', k)
```

```
两电一邮包括 成电
两电一邮包括 西电
两电一邮包括 北邮
```

In [3]:
```
# 循环值
for v in schools.values():
    print('A+学科是', v)
```

```
A+学科是 电科、信通
A+学科是 电科
A+学科是 信通
```

In [4]:
```
# 循环键值对
for k, v in schools.items():
    print(k, '的 A+学科是', v)
```

```
成电 的 A+学科是 电科、信通
西电 的 A+学科是 电科
北邮 的 A+学科是 信通
```

以上 print 输出的都是一个元组变量，用于替代 f 字符串。

（3）while 循环

for 循环用于遍历高级变量类型中的元素，而 **while** 循环用于不断运行，直到布尔条件从 **True** 转为 **False**。其语法规则为

```
while bool:
    循环体
```

示例如下。

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

```
1
2
3
4
5
```

（4）continue 与 break

continue 用于中断本轮循环并进入下一轮循环，在 **for** 和 **while** 中均有效。

break 用于停止循环，跳出循环后运行后续代码，在 **for** 和 **while** 中均有效。

In [1]:
```
# break 的演示
a = 1
while True:
    if a > 5:
        break
    print(a)
    a += 1
```

```
1
2
3
4
5
```

In [2]:
```
# continue 的演示
a = 0
while a < 5:
    a += 1
    if a == 3:
        continue
    print(a)
```

```
1
2
4
5
```

3.6 列表推导式

这个语法不要求掌握，只求能看懂即可（有些演示代码里可能会出现）。

基础用法如示例所示，这两种写法可以表达同一个意思。

In [1] :
```
# 求平方——循环
value = [ ]
for i in [1,2,3,4,5]:
    value = value + [i**2]
print(value)
```
[1, 4, 9, 16, 25]

In [2] :
```
# 求平方——列表推导式
value = [ i**2  for i in [1,2,3,4,5] ]
print(value)
```
[1, 4, 9, 16, 25]

In [2]的第二行读作：value = i^2，i 取[1,2,3,4,5]，则 value = [1^2, 2^2, 3^2, 4^2, 5^2]。

当然，列表推导式可以加一个 **if** 语句，如示例所示

In [1] :
```
# 求平方——循环
value = [ ]
for i in [1,2,3,4,5]:
    if i < 4:
        value = value + [i**2]
print(value)
```
[1, 4, 9]

In [2] :
```
# 求平方——列表推导式
value = [ i**2  for i in [1,2,3,4,5]  if i < 4 ]
print(value)
```
[1, 4, 9]

In [2]的第二行读作：value = i^2，i 取[1,2,3,4,5]，但 i < 4 时生效，则 value = [1^2, 2^2, 3^2]。

3.7 高级变量间的转换

集合、元组、列表、字典四者之间可以无缝切换，需要用到四个函数

- 转换为集合使用 set 函数；
- 转换为元组使用 tuple 函数；
- 转换为列表使用 list 函数；
- 转换为字典使用 dict 函数。

In [1]:
```
# 定义变量
set_v = {1,2,3}
tuple_v = (1,2,3)
list_v = [1,2,3]
dict_v = { 'a':1 , 'b':2 , 'c':3 }
```

In [2]:
```
# 转化为集合
print( set( tuple_v ) )
print( set( list_v ) )
print( set( dict_v.keys() ) )
print( set( dict_v.values() ) )
print( set( dict_v.items() ) )
```

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

In [3]:
```
# 转化为元组
print( tuple( set_v ) )
print( tuple( list_v ) )
print( tuple( dict_v.keys() ) )
print( tuple( dict_v.values() ) )
print( tuple( dict_v.items() ) )
```

```
(1, 2, 3)
(1, 2, 3)
('a', 'b', 'c')
(1, 2, 3)
(('a', 1), ('b', 2), ('c', 3))
```

In [4]:
```
# 转化为列表
print( list( set_v ) )
print( list( tuple_v ) )
print( list( dict_v.keys() ) )
print( list( dict_v.values() ) )
print( list( dict_v.items() ) )
```

```
(1, 2, 3)
(1, 2, 3)
('a', 'b', 'c')
(1, 2, 3)
(('a', 1), ('b', 2), ('c', 3))
```

In [5]:
```
# 转化为字典
print( dict( zip( {'a','b','c'} , set_v   ) ) )
print( dict( zip( ('a','b','c') , tuple_v ) ) )
print( dict( zip( ['a','b','c'] , list_v  ) ) )
```

```
{'a': 1, 'b': 2, 'c': 3}
{'a': 1, 'b': 2, 'c': 3}
{'a': 1, 'b': 2, 'c': 3}
```

注：在使用 dict 函数时，需搭配 zip 函数，zip 可将两个容器内的元素配对。

四、函数

函数可以避免大段的重复代码，其格式为

```
def 函数名(输入参数):
    ''' 文档字符串 '''
    函数体
    return 输出参数
```

文档字符串用于解释函数的作用，查看某函数文档字符串的方法是.__doc__。

第四行的 **return** 可省略（一般的函数不会省略），若省略，则返回 **None**。

4.1 吞吐各个类型的变量

函数的输入参数与输出参数均可以为任意的变量类型，示例如下。

In [1]:
```
# 函数的输入与输出
def my_func(v):
    ''' 我的函数 '''
    return v
```

In [2]:
```
str_v = my_func( "cxk" )          # 字符串
str_v
```
Out [2]: 'cxk'

In [3]:
```
num_v = my_func( 123 )            # 数字
num_v
```
Out [3]: 123

In [4]:
```
bool_v = my_func( True )          # 布尔型
bool_v
```
Out [4]: True

In [5]:
```
set_v = my_func( {1,2,3} )        # 集合
set_v
```
Out [5]: {1, 2, 3}

In [6]:
```
tuple_v = my_func( (1,2,3) )      # 元组
tuple_v
```
Out [6]: (1, 2, 3)

In [7]:
```
list_v = my_func( [1,2,3] )       # 列表
list_v
```
Out [7]: [1, 2, 3]

In [8]:
```
dict_v = my_func( {'a':1, 'b':2 , 'c':3 } )     # 字典
dict_v
```
Out [8]: {'a': 1, 'b': 2, 'c': 3}

函数内部的空间是独立的，函数内部的变量叫做形式参数，不影响外界的实际参数。在刚刚的例子中，In [1]函数体内的 v 就是形式参数，它在外界的实际空间中是不存在的，只在调用函数的过程中会在函数空间内临时存在。

4.2 吞吐多个变量

（1）吞吐多个普通参数

输入多个值本质是输入了一个元组，输出多个值本质是输出了一个元组。

In [1]:
```
# 吞吐多个值（借助元组）
def my_counter(a,b):
    '''加法器和乘法器'''
    return a+b, a*b
```

In [2]:
```
x,y = my_counter(5,6)
x,y
```

Out [2]: (11, 30)

（2）吞吐一个任意数量的参数

当然，还可以输入一个任意数量的参数，如下示例所示。

In [1]:
```
# 传入任意数量的参数（利用元组拆分法）
def menu(*args):
    '''菜单'''
    return args
```

In [2]:
```
info = menu('荔枝', '油饼', '香精煎鱼', '香翅捞饭')
info
```

Out [2]: ('荔枝', '油饼', '香精煎鱼', '香翅捞饭')

（3）吞吐多个普通参数，并附带一个任意数量的参数

当然，普通参数与任意数量参数可以同时出现，但请把后者放在最右侧。

In [1]:
```
# 同时传入普通参数和任意数量的参数（后者只能出现一个）
def her_hobbies(name, *hobbies):
    return name, hobbies
```

In [2]:
```
n,h = her_hobbies('cxk', 'singing', 'dancing', 'rap', 'basketball')
print( f"{n} likes {h}." )
```

Out [2]: cxk likes ('singing', 'dancing', 'rap', 'basketball').

（4）吞吐多个普通参数，并附带一个任意数量的键值对参数

除了上述示例，甚至可以输入多个键值对，如下示例所示。

In [1]:
```
# 对各专业评价
def evaluate(in1, in2, **kwargs):
    ''' 先对计算机类评价，再对通信类评价，也可自行补充 '''
    kwargs ['计算机类'] = in1
    kwargs ['通信工程'] = in2
    return kwargs
```

In [2]:
```
# 规矩评价法
eva1 = evaluate( '打代码的', '拉网线的' )
eva1
```

Out [2]: {'计算机类': '打代码的', '通信工程': '拉网线的'}

In [3]:
```
# 额外补充法
eva2 = evaluate(
    '打代码的',
    '拉网线的',
    电子工程 = '焊电路的',
    能源动力 = '烧锅炉的'
)
eva2
```

Out [3]:
```
{'电子工程': '焊电路的',
 '能源动力': '烧锅炉的',
 '计算机类': '打代码的',
 '通信工程': '拉网线的'}
```

在上述示例中，函数的输入形参 kwargs 的两个**会让 Python 创建一个名为 kwargs 的空字典，并将键值对放入其中。

4.3 函数的关键字调用

函数可以顺序调用，也可以关键字调用，关键字方式更广泛，如示例所示。

In [1]:
```
def my_evaluate1(college, major, evaluate):
    '''对某大学某专业的评价'''
    message = f"{college}的{major}{evaluate}。"
    return message
```

In [2]:
```
# 顺序调用
info = my_evaluate1('三峡大学', '电气工程', '挺厉害')
info
```

Out [2]:
```
'三峡大学的电气工程挺厉害。'
```

In [3]:
```
# 关键字调用
info = my_evaluate1('三峡大学', evaluate='也还行', major='水利工程')
info
```

Out [3]:
```
'三峡大学的水利工程也还行。'
```

4.4 输入参数的默认值

如果有些参数绝大部分情况是不变的，那么可以给其设置一个默认值。

In [1]:
```
# 函数的默认值
def my_evaluate2(college, level='带专'):
    message = f"{college}，你是一所不错的{level}!"
    return message
```

In [2]:
```
info = my_evaluate2('中南大学')      # 遵循默认值
info
```

Out [2]:
```
中南大学，你是一所不错的带专!
```

In [3]:
```
info = my_evaluate2('铁道学院', level='职高')      # 打破默认值
info
```

Out [3]:
```
铁道学院，你是一所不错的职高!
```

五、类

5.1 创建和使用类

- 类的本质：在一堆函数之间传递参数；
- 根据约定，类的名称需要首字母大写；
- **类中的函数叫方法**，一个类包含一个__init__方法 + 很多自定义方法，__init__特殊方法前后均有两个下划线，每一个类中都必须包含此方法。

示例如下。

In [1]:
```
# 类的示范
class Counter:
    '''一台可以加减的计算器'''

    def __init__(self,a,b):                # 特殊方法
        ''' a 和 b 公共变量，也是 self 的属性'''
        self.a = a                         # 公共变量 a 是 self 的属性
        self.b = b                         # 公共变量 b 是 self 的属性

    def add(self):                         # 自定义的加法方法
        '''加法'''
        return self.a + self.b

    def sub(self):                         # 自定义的乘法方法
        '''减法'''
        return self.a - self.b
```

- 函数内部的变量与外部是在两个空间，为了使自定义方法能在类里互通，需要一个 self 作为舰船，将需要互通的变量作为 self 的属性进行传递；因此，特殊方法__init__旨在使用舰船 self 来承载公共变量 a 和 b。
- __init__特殊方法后的括号里要写上舰船 self 以及公共变量 a 和 b，而自定义方法后的括号就只需要写舰船 self 即可。

类的编写完成后，可以创造很多的实例出来。

In [2]:
```
# 创建类的实例
cnt = Counter(5,6)      # 创建类的一个实例 cnt
print( cnt.a, cnt.b )   # 访问属性
print( cnt.add() )      # 调用方法
```

```
5 6
11
```

5.2 属性的默认值

前面讲过，属性即公共变量，上一个实例里的属性即 a 和 b。

可以给 self 的属性一个默认值，此时默认值不用写进__init__后面的括号里。

In [1]:
```
# 带有默认值的参数
class Man:
    '''一个真正的 man'''

    def __init__(self,name,age):
        '''公共变量'''
        self.name = name
        self.age = age
        self.gender = 'man'      # 一个带有默认值的属性

    def zwjs(self):
        '''自我介绍'''
        print(f"大家好！我是{self.name}，今年{self.age}岁了！")
```

In [2]:
```
# 创建与使用类
cxk = Man('鸡哥',24)
cxk.name, cxk.age      # 访问属性
```

Out [2]: ('鸡哥', 24)

In [3]:
```
cxk.zwjs()             # 调用方法
```

Out [3]: 大家好！我是鸡哥，今年 24 岁了！

In [4]:
```
cxk.gender             # 访问默认值
```

Out [4]: 'man'

In [5]:
```
# 修改默认值的数值
cxk.gender = 'Neither man nor woman'
cxk.gender
```

Out [5]: 'Neither man nor woman'

5.3 继承

继承：在某个类（父类）的基础上添加几个方法，形成另一个类（子类）。

- 父类从无到有去写属性和方法，第一行是 **class** 类名:
- 子类可继承父类的属性和方法，第一行是 **class** 类名(父类名):

子类在特殊方法里使用 super()函数，就可以继承到父类的全部属性与方法。

In [1]:
```
# 父类（也就是前面的加减计算器）
class Counter:
    '''一台可以加减的计算器'''

    def __init__(self,a,b):
        '''公共变量'''
        self.a = a
        self.b = b

    def add(self):
        '''加法'''
        return self.a + self.b

    def sub(self):
        '''减法'''
        return self.a - self.b
```

In [2]:
```
# 子类（在加减的基础上，添加乘除功能）
class Counter2(Counter):
    '''一台可以加减乘除的高级计算器'''

    def __init__(self,a,b):
        '''引用父类的属性'''
        super().__init__(a,b)       # 继承父类（superclass）

    def mul(self):
        '''乘法'''
        return self.a * self.b

    def div(self):
        '''除法'''
        return self.a / self.b
```

In [3]:
```
test = Counter2(3,4)
print( test.sub() )          # 调用父类的方法
print( test.mul() )          # 调用自己的方法
```

```
-1
12
```

如果想要在子类中修改父类中的某个方法，可以直接在子类里写一个同名方法，即可实现覆写，你也可以把覆写说成“变异”。

5.4 掠夺

继承只能继承一个类的方法，但如果想要得到很多其它类的方法，则需要掠夺功能。有了掠夺功能，一个类可以掠夺很多其它的类，示例如下。

In [1]:
```
# 一个无辜的类
class Counter:
    '''一台可以加减的计算器'''

    def __init__(self,a,b):
        '''公共变量'''
        self.a = a
        self.b = b

    def add(self):
        '''加法'''
        return self.a + self.b

    def sub(self):
        '''减法'''
        return self.a - self.b
```

In [2]:
```
# 掠夺者
class Amrc:
    '''一台本身只能乘除的计算器，现欲掠夺加减功能'''

    def __init__(self,c,d):
        '''公共变量'''
        self.c = c
        self.d = d
        self.cnt = Counter(c,d)      # 掠夺 Counter 类的方法

    def mul(self):
        '''乘法'''
        return self.c * self.d

    def div(self):
        '''除法'''
        return self.c / self.d
```

In [3]:
```
test = Amrc(3,4)                # 创建实例
print( test.mul() )             # 自己的方法
print( test.cnt.add() )         # 抢来的方法
```

```
12
7
```

掠夺的本质是，将另一类的实例当作 self 的属性用，这样一来，被掠夺的类的数量就不设上限。此外，假如掠夺者和被掠夺者里都有 add 函数，掠夺者里的方法用 test.add()，被掠夺者里的方法用 test.cnt.add()，于是就不必覆写。