# 写在前面

我做了十几年的程序猿，码过代码、带过项目、做过产品经理、做过软件架构师。因为我是做通信设备软件的，面向底层操作系统，所以我的工作主要以C语言为主。Python在我的工作中通常用来写一些小脚本。最近一个项目的需要，我需要深入学习python。现在python很火，很多没有编程经验的也开始学起python了，我觉得这是很有意思的一件事情，就产生了把我的学习过程用文章和视频的方式记录下来的想法，希望这些东西能帮忙大家尤其是这些初学者。

由于我需要考虑这些从来没有接触过编程的初学者，所以我写的有些内容会比较基础，进度也可能会比较慢，有编程经验的同学可以跳着看。

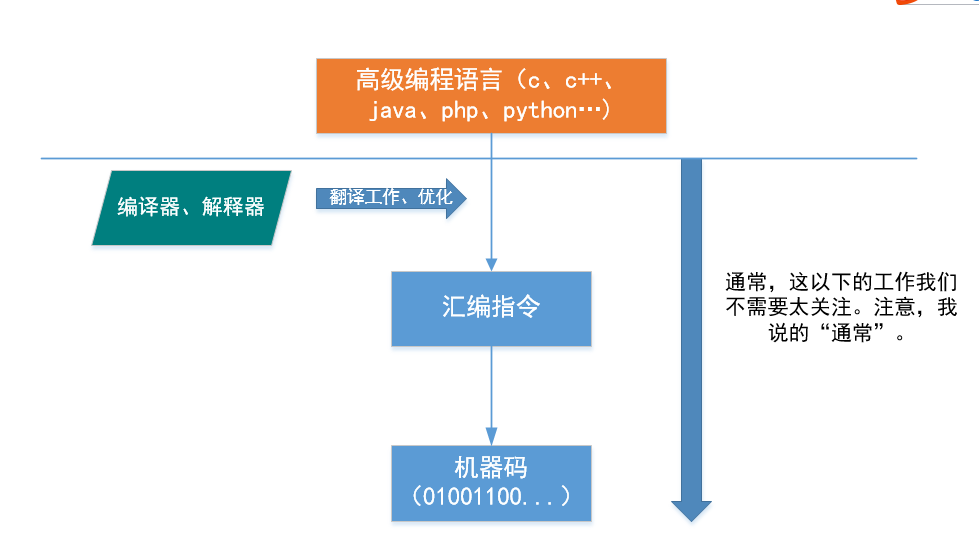
你可以叫我Tiger，准备好了吗？我们一起开启python的学习之旅吧！

# Python简介

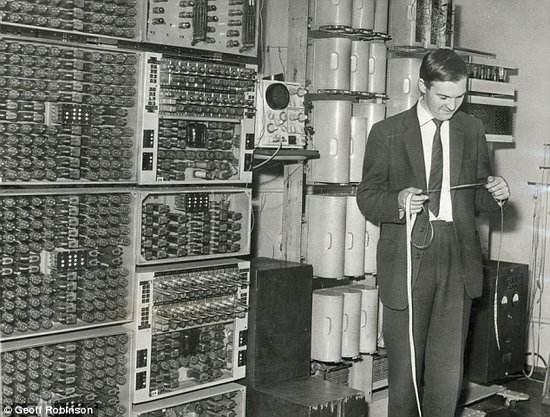
如果你从来没有接触过编程，那么需要先大体了解一下编程语言的历史背景。

1. 高级编程语言的诞生

计算机编程历经了从机器码到汇编再到高级编程语言的演进过程，下图描述了三者之间的逻辑关系：

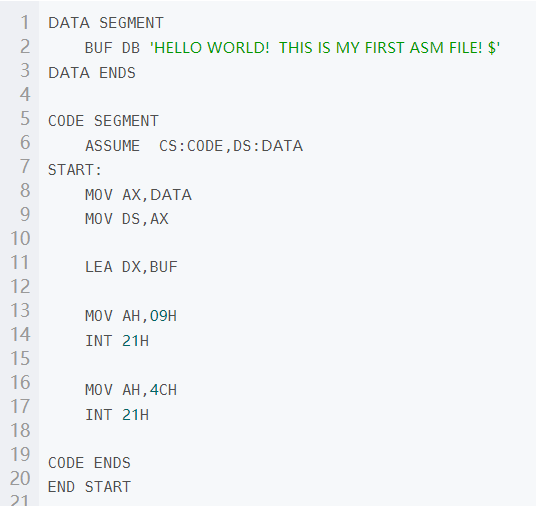


我们都知道，计算机只认识机器码，也就是一串对于人类来说毫无意义的0和1。这玩意儿对于机器来说是很友好的，但是对于人类来说就太残忍了。在没有编程语言之前，科学家们是通过打孔的方式来计算的，像这样：



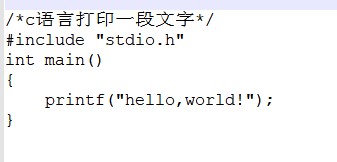
我很佩服这些科学家，因为他们拿着一根长长的打满孔的纸带，可以理解上面的意思。

后来，人类发明了汇编语言。这是一种编程语言了，但是它还不完全是现在我们接触到的编程语言，它介于人类和机器之间。要想玩转汇编语言，你首先得非常清楚计算机的计算原理，你得了解几十个不同的寄存器是怎么工作的，比如下面这段汇编代码，我相信很多写了多年代码的程序员都不一定能看懂。



汇编语言虽然不再是天书了，但对人类来说依然非常不友好。于是我们发明了“高级”编程语言，这里的高级是相对于汇编这种低级语言而言的。VB、C、C++、JAVA、PHP、Python…我们现在接触到的编程语言都是高级编程语言。

高级编程语言，定义了一大堆语法规则，这些语法规则是面向人类认知的，在人类看来是有逻辑意义的，易于理解。程序员基于这套语法规则编写的代码，再通过特殊的工具翻译成汇编指令，这种工具就是“编译器”或者“解释器”。



作为我们这一代程序员是幸福的，因为我们有了高级编程语言，我们不再需要把大量的精力放在研究机器是如何运行的，我们可以更多的关注我们代码需要实现的业务逻辑和功能上面。

好的，我们总结一下。

* 发明编程语言的原因，是因为机器码理解起来太困难。
* 我们现在接触到的编程语言基本上都是高级编程语言。
* 编程语言的本质，是人为定义的一套语法规则，我们学习编程首先要学习掌握这套语言规则。

1. Python的优势

回到python。

没错，python就是数十种高级编程语言中的一个而已。那为什么现在这么多人开始学习python呢？我的看法是：

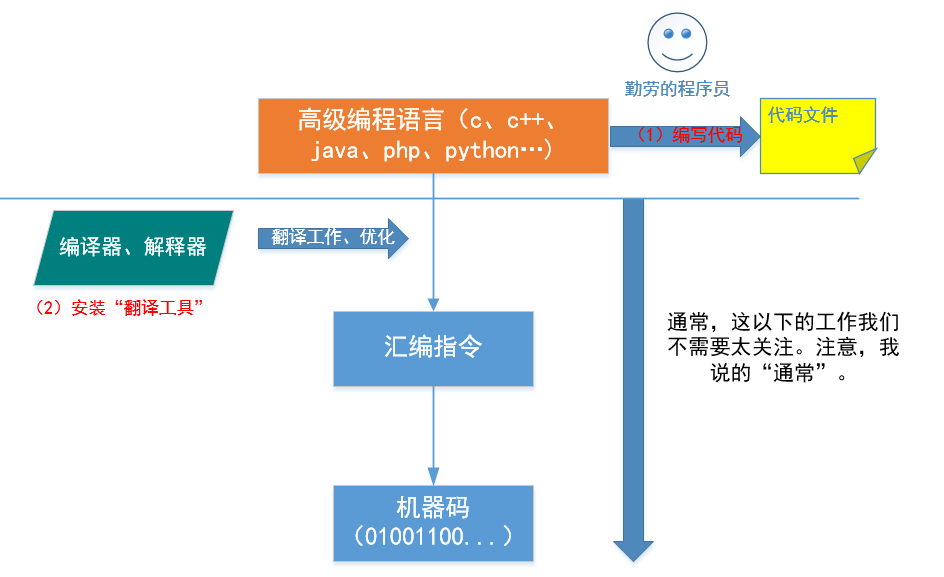
* **python足够简单**。尤其是对于没有编程经验的人来说，它入门足够简单。作为写了十几年C代码的老程序员，感触至深。你能想象让一个没有计算机基础的人去搞明白为什么内存会泄露吗？去搞明白什么是指针吗？
* **Python足够强大**。我一直认为python的强大在于其丰富的组件库，只要你能想到的功能，基本上都能找到现成的免费的开源库。作为一个老C程序员，当第一次看到python组件库，感动得泪流满面啊。想当年我们的系统，真的是从第一行C代码开始写的啊，没有任何可以拿来就用的库。
* **Python跟上了潮流**，或者说潮流选择了python。大数据、AI等等这些当红炸子鸡，都和python紧密联系在一起了，不火都不行。

好吧，废话不多说，我们开始python编程吧。

# 第一个Python程序

本节的任务，是完成我们的第一个python程序。

再次看看上一节提到的那张图，看看作为高级编程语言，我们如何编程。



首先，作为程序员的你，需要先遵循python定义的语法规则，编写一段代码并保存为一个代码文件。这仅仅是完成了编码工作，如何让这段代码在你的机器上运行起来呢？你还需要安装“翻译工具”，也就是python的解释器。

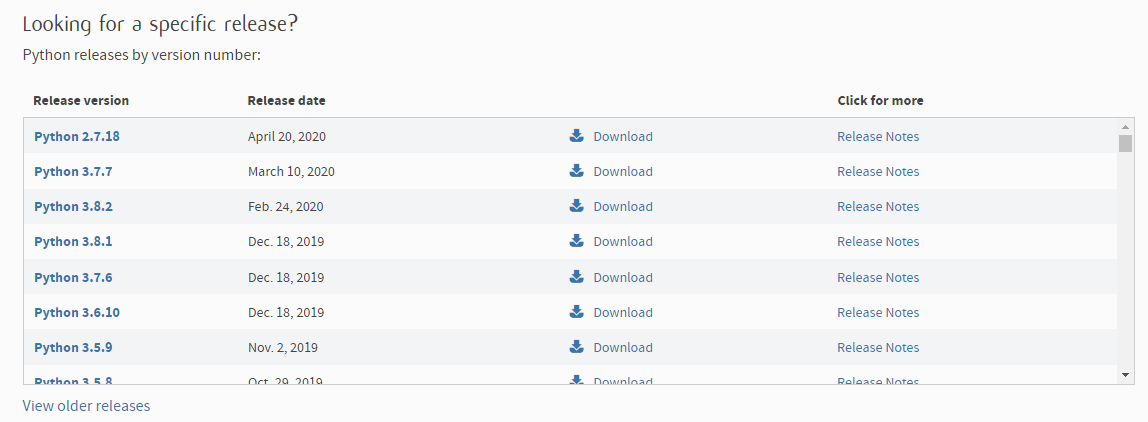
为了让我们的第一个python程序能顺利运行起来，我们先把解释器安装好。

## Python解释器安装

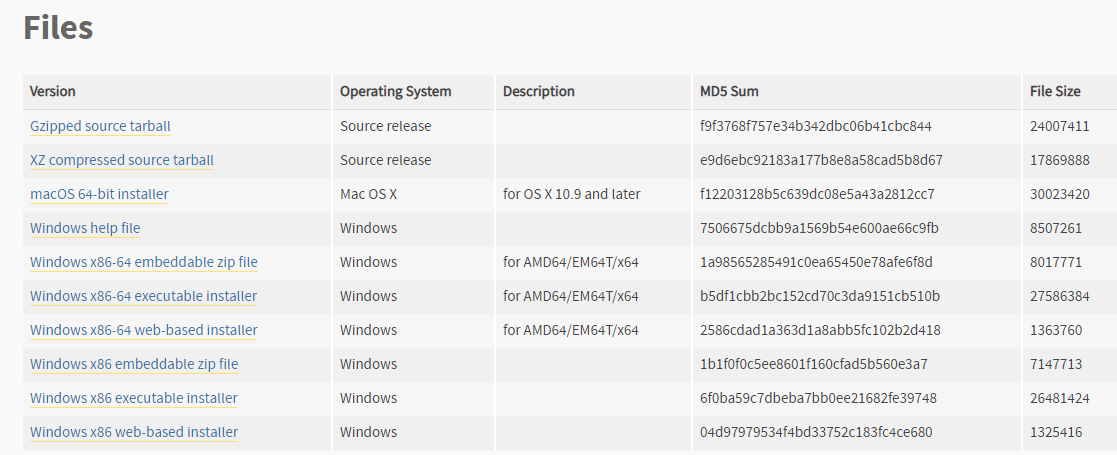
Python目前有两个大版本，python2.X和python3.X，这两个版本差异挺大，他们之间不兼容，也就是说你要想从python2.X升级到python3.X，是需要改代码的，它们之间存在语法差异。按照官网说法，python2.X会被python3.X取代，所以建议大家直接学习python3.X版本即可。

Python有多种解释器，CPython、IPython、PyPy、Jython…，它们的区别大家可以百度，这里我们不用太关注，以免被绕晕。直接去官网下载CPython即可。

Python官网：<https://www.python.org/downloads/>



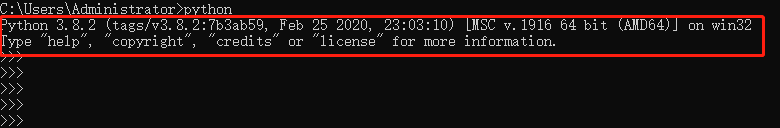
我下载的是最新的版本，Python3.8.2。



我的电脑是windows 64位操作系统，所以选择的是[Windows x86-64 executable installer](https://www.python.org/ftp/python/3.8.2/python-3.8.2-amd64.exe)。你也可以用linux，但是建议初学者还是用windows，可视化界面用起来方便。我们应该将主要的精力集中在python本身的学习上面，应尽量减少不熟悉的工具或者环境对我们学习过程的干扰。

Python的安装过程就非常简单了，按照提示默认一步一步安装即可。

安装完成后，Windows+R，输入 cmd，打开命令行工具。在命令行工具中输入python回车，如果出现python版本号则安装成功。

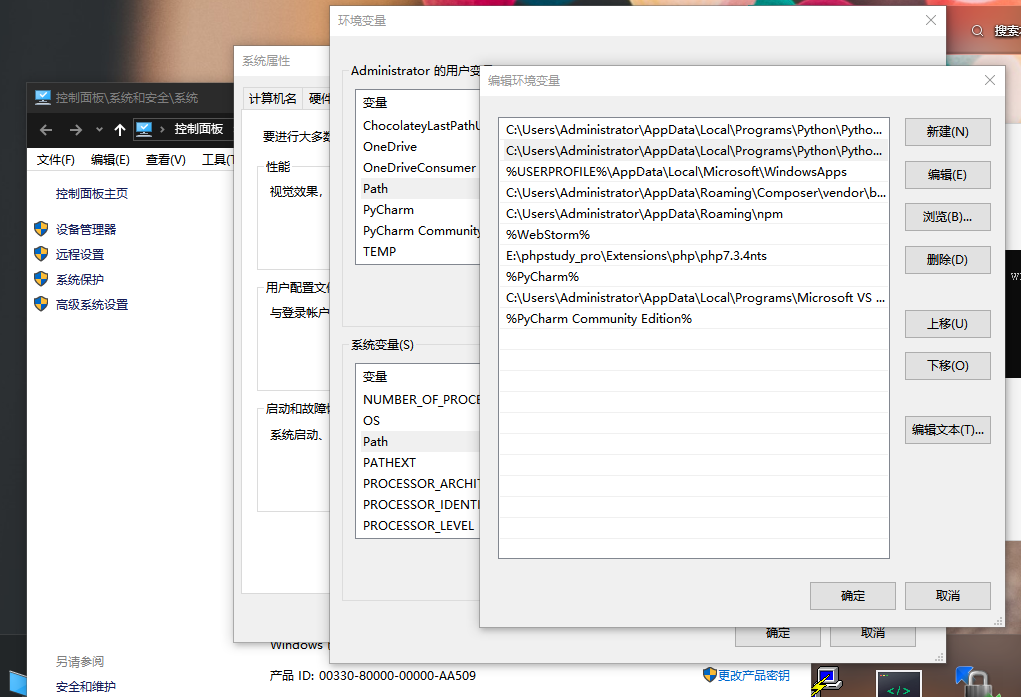


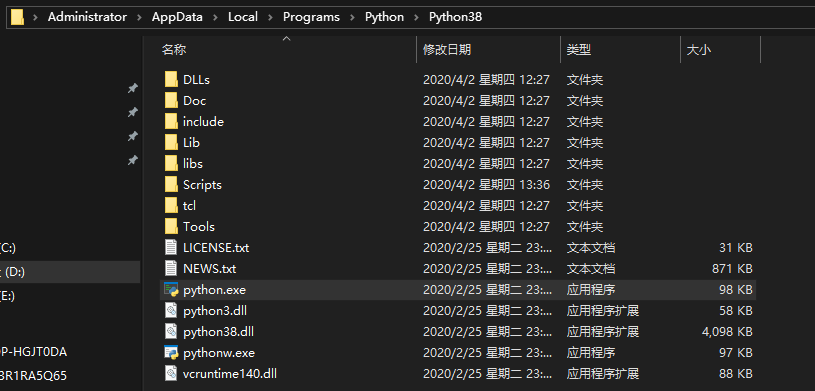
如果命令行提示找不到python命令，则可能是系统环境变量的设置问题。可以将python的安装目录添加到环境变量中。方法如下(我是windows10的操作系统，其它系统可能有些差别，不会的可以百度，很多设置环境变量的文章)：

右键点击“我的电脑”-> 属性 -> 高级系统设置 -> 环境变量 -> 用户变量 -> Path -> 新建。

输入我们安装的python可执行文件（python.exe）地址，我的在

C:\Users\Administrator\AppData\Local\Programs\Python\Python38





设置完环境变量后，再到命令行检查是否安装成功。

至此，我们的电脑上面就可以正常运行python代码程序了！

## 尝试写第一个python程序，并运行起来

我们开始尝试写第一个python程序，按照程序员的惯例，第一个程序通常叫“hello，world！”。

理论上，你可以使用任何文本编辑器来编辑代码，但是对于专业的程序员来说，他们会选择适合自己的专业代码编辑器，我们叫做集成开发环境IDE。给自己选择一个用着顺手的IDE是很重要的。为了不让工具干扰我们，我们先不安装专门的python IDE。你可以先使用任意一款文本编辑器（比如记事本、notepad++、ultraEdit等）编写下来这段代码。

# -\*-coding:utf-8-\*-

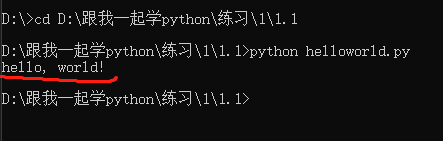
# file: ./3/helloworld.py

# 字符串打印

print("hello, world!")

然后保存为文件“helloworld.py”。

接下来我们需要运行这个python程序，打开命令行工具，cd到代码文件所在的目录。然后执行 > python helloworld.py，你可以在命令行里面看见打印出了“hello，world！”。



恭喜你，你已经成功编写并运行了第一个python程序！虽然这个程序看起来并没什么鸟用。

回过头来，我们看看我们写的那几行代码是什么意思。

# -\*-coding:utf-8-\*-

这一行是为源代码指定编码方式，默认情况下python3的编码方式就是utf-8。至于什么是字符编码，我们后面会专门安排一个章节来讲。大家现在只需要知道这行是用于指定字符编码方式即可。

# 字符串打印

这一行是代码的注释。“注释”是给程序员看的，便于程序员走读代码的时候理解代码逻辑。Python解释器在翻译代码的时候，会跳过这些“注释”语句。

除了可以使用#单行注释之外，也可以使用连续三个单引号或者双引号，进行多行注释：

# 这是单行注释的写法

# 这是单行注释的写法  
'''  
这是多行注释的写法  
这是多行注释的写法  
这是多行注释的写法  
这是多行注释的写法  
'''  
"""  
这是多行注释的写法  
这是多行注释的写法  
这是多行注释的写法  
这是多行注释的写法  
"""

print("hello, world!")

这一行是真正起作用输出“hello，world！”的代码。Print是python自带的一个函数，用于输出一串字符，括号里面的“hello，world！”是我们要输出的字符串值。后面我们有专门的章节讲解什么是函数。

## 安装IDE

作为专业的程序员，我们肯定不会使用普通的文本编辑器写代码的（喜欢炫技的除外）。一个功能强大的IDE可以极大提升编码的效率，同时降低我们编码的错误率。

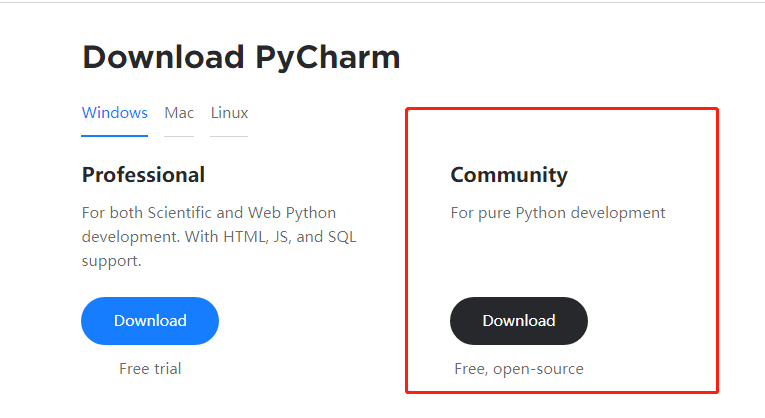
我使用的python IDE是pycharm，jetbrains公司专门针对python开发的一款专业IDE。这个工具的专业版是要付费的，社区版（community edition）是免费的，对于学习来说，免费版足够。我推荐大家使用pycharm。

当然也有很多人推荐使用微软的visual studio code，我不用后者的原因是因为它需要装一些插件才能使用，对于初学者来说没必要去折腾这些东西。我还是那个原则，尽量少花时间去折腾工具，让我们聚焦在python本身的学习上面。

Pycharm安装上就可以使用。

你需要先到pycharm的官网下载社区版：

<https://www.jetbrains.com/pycharm/download/#section=windows>



安装过程很简单了，按照默认配置一步步点就可以了。

后续我们所有的演示都会基于pycharm，大家在实际的练习中就可以学会如何使用pycharm，我们不单独教大家怎么使用该软件了。大家也可以百度学习。

至此，我们已经做好了正式学习python的一切准备。

# 变量

我们把前面的程序稍微改一下，来了解python中的变量。

# file: ./4/4\_1.py

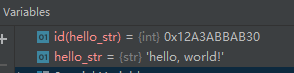
# 定义变量  
hello\_str = "hello, world!"  
  
# 字符串打印  
print(hello\_str)

这段代码实现的功能也是在终端打印出“hello,world!”这一字符串。与我们最开始那个程序不同的是，它定义了一个变量“hello\_str”用于存储这个字符串，然后再调用print函数输出。本节我们来学习什么是变量？

变量，顾名思义就是可以变化的一个数据，与其对应的不能改变的数据，叫做“常量”。变量和常量，是所有编程语言的一个基本概念。Python里面没有专门定义常量的语法，通常用变量来替代，所以我们不专门介绍常量。

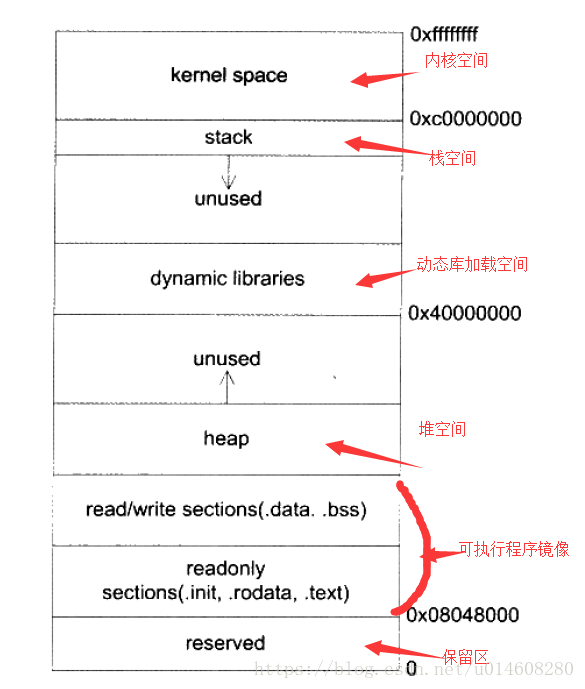
抽象了看，所有的程序，无论大小，其本质都是在操作一系列的数据按照我们预设的逻辑去运算。这些数据在运算过程中，会被临时存储在内存中，我们可以认为变量就是对这些存储空间的一个命名。我们可以在代码中通过使用变量，来达到操作对应数据的目的，而不需要感知这个数据具体是怎么被计算机存储的。

我们通过pycharm来调试该段代码，看看变量是如何存储数据的：



我们通过id(hello\_str)来获取变量hello\_str的地址，这个地址是我们看来很奇怪的一串值。其实这个值是一个内存地址，它指向的是一段内存空间的起始位置。“hello,world!”这一串字符就存储在这段内存空间中。

变量的存储空间是堆(heap)和栈(stack)。堆栈是有很大区别的，在C语言里面你需要非常清晰的搞清楚它们，但是python封装得更好，不需要太去深究它们。如果感兴趣，你可以参考下图，或者百度。

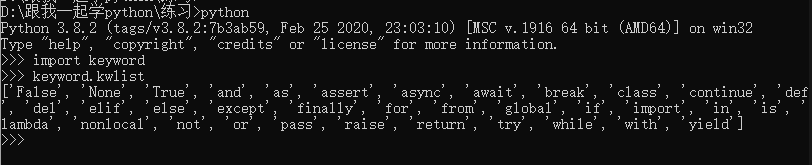


总结一下，变量是用来临时存储数据的，它本质上指向的是一段存储空间的起始地址。

Python里面，对于变量的命名有一些约束，如下：

* 变量的第一个字符必须是字母表中字母或下划线 \_ 。
* 变量的其他的部分由字母、数字和下划线组成。
* 标识符对大小写敏感。
* 变量不能采用python的保留字命名。

我们可以在windows命令行中采用下面的方法查询保留字：



理论上，python3可以支持采用中文字符命名变量。

下面我们通过一个有意思的例子来进一步深入理解python的变量。

# file: ./4/4\_2.py

score1 = 10  
score2 = 50  
  
# score1 score2的地址  
print('id(score1): %x, id(score2): %x, ' % (id(score1), id(score2)))  
  
score2 = score2 - score1  
  
# 字符串打印  
print('score2: ', score2)  
  
# score1 score2的地址  
print('id(score1): %x, id(score2): %x, ' % (id(score1), id(score2)))  
  
score3 = 40  
print('id(score3): %x' % (id(score3)))

这个例子的输出如下：

id(score1): 7ffc5331d7c0, id(score2): 7ffc5331dcc0,

score2: 40

id(score1): 7ffc5331d7c0, id(score2): 7ffc5331db80,

id(score3): 7ffc5331db80

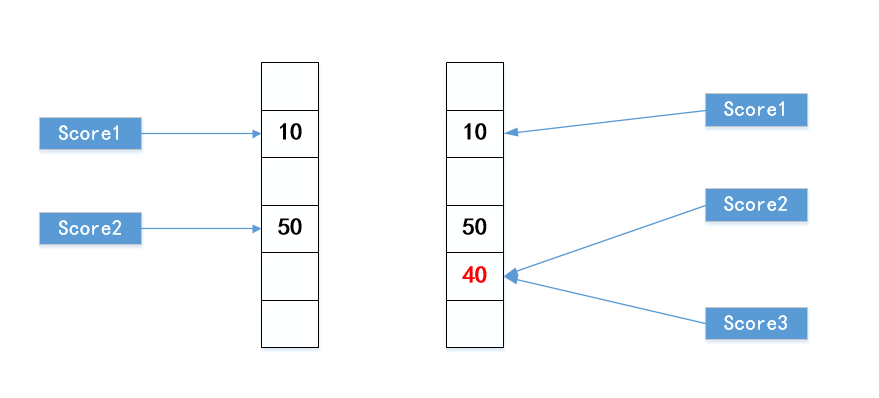
我们可以看到，在score2的值变为40之后，它指向了一个新的地址7ffc5331db80。而我们定义了一个新变量score3，给它赋值40，它居然也和score2指向了同一块地址空间。写惯了C语言的程序员对此会感到费解，但python就是这样的。

当我们给score1赋值为10时，python会先给10创建一个对象并分配一个存储空间，然后再将score1指向这个对象。

当score2做减法后，python也会先给40分配一个对象空间，然后将score2指向这个新的对象，所以我们看到score2的地址变了。

当我们给score3赋值为40时，由于40对应的对象存在，所以直接将score3指向了这个对象。所以我们看到score2和score3的地址相同。

如下图所示：



那么原来那个值为50的对象怎么处理呢？如果是C语言，需要程序员主动去将其释放，否则就会内存泄露。幸运的是，python为我们提供了自动垃圾回收机制（GC），当它发现这个对象没有被引用后会自动将其释放。关于垃圾回收机制，这是一个很大的话题，我们现在没必要去深究它，有兴趣的同学可以百度。

我们再来看一个有意思的例子：

# file: ./4/4\_3.py

score1 = 10

score2 = score1  
score1 = 20  
  
print('score1: ', score1)  
print('score2: ', score2)

它的输出是：

score1: 20

score2: 10

没有编程经验的同学会对此感到疑惑，不是score2=score1吗，为什么score1改变了，score2却没有改变？我们同样可以把变量的地址打印出来，就很好理解了。

# file: ./4/4\_3.py

score1 = 10  
print('id(score1): %x' % (id(score1)))  
  
score2 = score1  
print('id(score2): %x' % (id(score2)))  
  
score1 = 20  
print('id(score1): %x, id(score2): %x, ' % (id(score1), id(score2)))  
  
print('score1: ', score1)  
print('score2: ', score2)

它的输出是：

id(score1): 7ffc5331d7c0

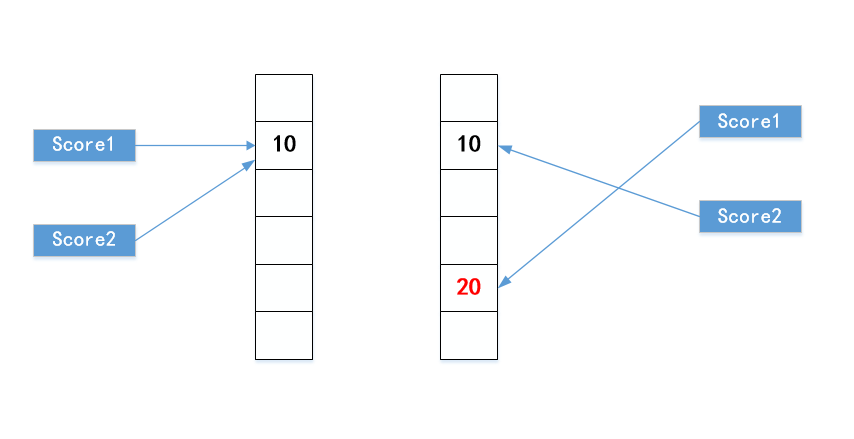
id(score2): 7ffc5331d7c0

id(score1): 7ffc5331d900, id(score2): 7ffc5331d7c0,

score1: 20

score2: 10

我们可以看到，在score2 = score1后，score2的确指向了score1的地址。但是我们改变score1的值为20后，score1指向了另外一块地址空间，而score2并没有跟着改变，所以score2依然是10.如下图所示：



我们最后再看一个例子，这个例子里面我们给变量赋了一个列表数据结构的值，关于数据结构，我们下一节会详细介绍，大家不用太关注。我们看看对于列表结构，它的变量是否也如同上面两个例子那样。

# file: ./4/4\_4.py

# 列表  
num\_list1 = [10, 20, 30, 40]  
num\_list2 = [10, 20, 30, 40]  
  
print('id(num\_list1): %x, id(num\_list2): %x' % (id(num\_list1), id(num\_list2)))  
  
num\_list1[0] = 50  
print('num\_list1: ', num\_list1)  
  
print('id(num\_list1): %x, id(num\_list2): %x' % (id(num\_list1), id(num\_list2)))

它的输出是：

id(num\_list1): 292bd87df00, id(num\_list2): 292bd9ba880

num\_list1: [50, 20, 30, 40]

id(num\_list1): 292bd87df00, id(num\_list2): 292bd9ba880

可以看出，在初始赋值时，虽然num\_list1和num\_list2的值是相同的，但是它们指向的地址空间并不相同。我们修改了num\_list1的值之后，num\_list1也并没有重新指向一个新的地址。所以，列表数据类型和我们上面两个例子中的变量处理是不一样的。Python会给列表对应的变量分配独立的地址空间，即便值相同，也不会多个变量复用。



对于不同数据类型的变量，python的处理方式是不一样的。也许你现在会觉得有点乱，没关系，下节我们学习了python的数据类型之后，你就能理解python的解释器为什么要这样区别处理了。

另外，从这些例子我们也能理解变量给我们带来的好处了，试想如果没有变量的话，程序员几乎没法写代码，因为你想要的那个数据一会儿存在A地址，一会儿又存在了B地址。但是有了变量的话，程序员只需要对这个变量进行操作即可，不需要关心它具体指向哪儿。

好了，下节我们学习python的数据类型。

# 数据类型

本节我们开始学习python的数据类型。

什么是数据类型呢？前面我们提过，所谓的编程，就是控制一系列的数据去完成我们预设的逻辑或者功能。所以，编程语言首先要定义一系列对“数据”的处理规则。这些处理规则包括：如何存储数据、数据的长度、数据的赋值、数据的读取、数据的显示、数据的比较等等。

不同类型的数据，它们的这些处理规则是不一样的。比如：整数和小数在内存中的存储方式肯定是不一样的；小数有精度的操作，而字符串肯定是没有的。

因此，编程语言需要对我们用到的所有数据进行分类，抽象出一些基本的类型，这就是编程语言定义的数据类型。

不同的编程语言所定义的数据类型其实大同小异，所以你只要理解了python的数据类型，其它编程语言的数据类型也基本都能搞明白。

Python3 中定义了七个标准的数据类型：

* Number（数字）
* String（字符串）
* Bytes（字节）
* List（列表）
* Tuple（元组）
* Set（集合）
* Dictionary（字典）

下面我们结合一些实例，依次学习这几个标准类型。

## Number（数字）

数学中的数字包括整数、小数，python中也对应定义了整型(int)和浮点型(float)，另外还定义了复数类型（complex）和布尔型(bool)。

* Int：它定义了一个整数类型。在C语言中，有很多不同的整型，如下表：

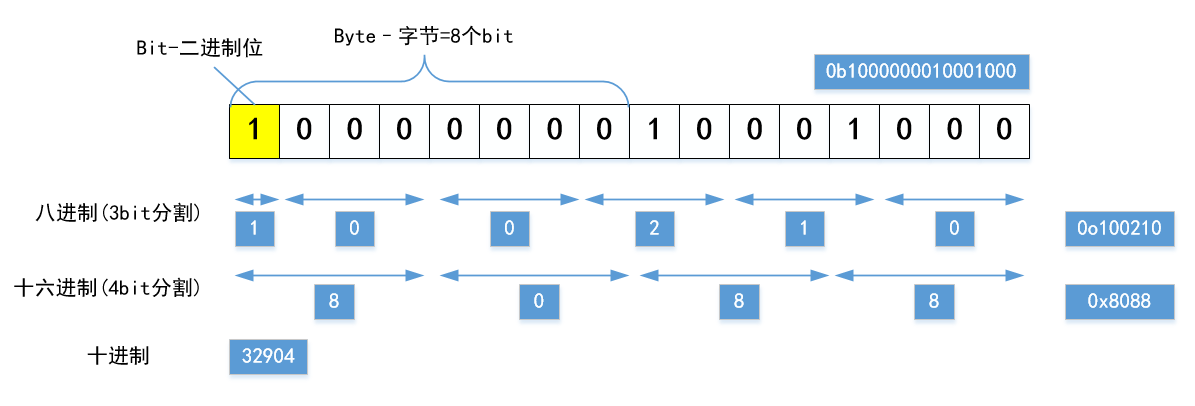


不同的类型，有不同的存储大小和取值范围，超出这个范围就会溢出报错。Python明显简化了这一类型，所有的整数类型，不论正负不论大小，全都归一为int类型。Python2还保留了long类型，python3把long类型也去掉了。

我们可以使用不同的方法表示一个int类型的数据，如下所示：

# int 类型

int\_1 = 100 # 10进制正数  
int\_2 = -69 # 10进制负数  
int\_3 = 0x77 # 16进制正数  
int\_4 = -0x24 # 16进制负数  
int\_5 = 0o70 # 8进制正数  
int\_6 = -0o70 # 8进制负数  
int\_7 = 0b10 # 2进制正数  
int\_8 = -0b10 # 2进制负数

上面提到了进制，我们先花几分钟了解一下“进制”的概念，看下面这张图：  


Python中使用0b开头表示二进制(bin)、0o开头表示八进制(oct)、0x开头表示十六进制(hex)。我们平时用得比较多的，是十进制(dec)、十六进制、二进制，八进制用得比较少。十六进制由于是满16进位，所以大于10的位用字母表示，10-15依次是a-f，不区分大小写。

进制是数值的不同表达方式。不同进制之间可以相互转换，参考下面的实例：

# file: ./6/6\_1.py  
  
# 进制表达和转换  
temp\_value = 32904  
  
print(" 十进制：{}\n 二进制：{}\n 八进制：{}\n 十六进制：{}\n"  
 .format(temp\_value, bin(temp\_value), oct(temp\_value), hex(temp\_value)))

输出为：

十进制：32904

二进制：0b1000000010001000

八进制：0o100210

十六进制：0x8088

* Float：它定义了一个小数类型。在编程语言中，我们习惯把小数叫做浮点数，所以float是浮点型。同样，python简化了浮点型的定义，不论数值大小以及精度，都归一为float类型。Python中有多种方式来表示一个浮点数，如下：

# float 类型

float\_1 = 17.18 # 正浮点数  
float\_2 = -17.18 # 负浮点数  
float\_3 = 7.99E+3 # 科学计数正数  
float\_4 = -7.99E+3 # 科学计数负数  
float\_5 = 7.99E-3 # 科学计数  
float\_6 = 7.99E3 # 科学计数  
float\_7 = 198. # 等同于198.0

* Complex：复数是由一个实数和一个虚数组合构成，表示为：x+yj，其中 x 是实数部分，y 是虚数部分。实数和虚数都是float类型数据。虚数部分必须有后缀j或者J。

# file: ./5/5\_2.py  
  
# complex 类型  
cpx\_1 = 123.2+34.6j  
  
print(cpx\_1)  
print('real: type %s, value %f' % (type(cpx\_1.real), cpx\_1.real))  
print('imag: type %s, value %f' % (type(cpx\_1.imag), cpx\_1.imag))

输出为：

(123.2+34.6j)

real: type <class 'float'>, value 123.200000

imag: type <class 'float'>, value 34.600000

* Bool：布尔类型只有两个值，True和False，它们对应值为1和0。Python2中没有真正的布尔类型，使用1和0替代。Python3中明确定义了关键词True和False，注意大小写。

# file: ./5/5\_3.py  
  
# bool 类型  
bool\_1 = True  
bool\_2 = False  
  
print(bool\_1)  
print(bool\_2)  
print(int(bool\_1)) # 强转为int类型  
print(int(bool\_2)) # 强转为int类型

输出为：

True

False

1

0

## String（字符串）

Python的字符串用单引号或者双引号括起来表示。

字符串由若干个有序字符组成，但是python中的字符串是不能改变的，也就是说我们不能通过索引去改变某个字符的值。比如：str[1] = ‘x’这样的操作是不被允许的。

### 转义字符

所谓转义字符，就是在其前面增加右斜杠\后，它并不代表其原本的字符含义，而是转义为另外的含义。比如‘\n’转义后表示一个换行符。

Python中用到的转义字符如下表所示：

| **转义字符** | **描述** |
| --- | --- |
| \(在行尾时) | 续行符 |
| \\ | 反斜杠符号 |
| \' | 单引号 |
| \" | 双引号 |
| \a | 响铃 |
| \b | 退格(Backspace) |
| \e | 转义 |
| \000 | 空 |
| \n | 换行 |
| \v | 纵向制表符 |
| \t | 横向制表符 |
| \r | 回车 |
| \f | 换页 |
| \oyy | 八进制数，yy代表的字符，例如：\o12代表换行 |
| \xyy | 十六进制数，yy代表的字符，例如：\x0a代表换行 |
| \other | 其它的字符以普通格式输出 |

# 转义字符

str1 = 'hello,\nworld!'  
# 尾部的\是续行符，表示下一行也算作本行的内容。续行符后面不能再有任何字符，包括空格和注释  
str2 = 'hello,\"world!\"' \  
 'I am Tiger.'  
  
print(str1)  
print(str2)

输出为：

hello,

world!

hello,"world!"I am Tiger.

对于续行符要注意，它后面不能再有任何字符，包括空格和注释，否则会报错。

### 截取

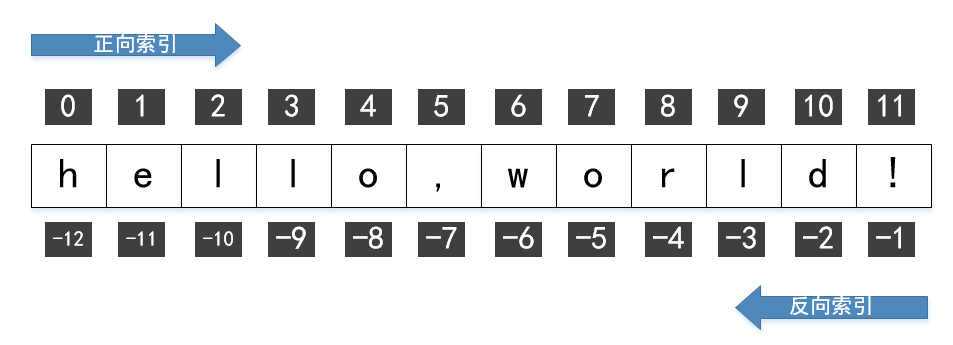
字符串截取就是从字符串中获取部分我们想到的字符，这是用得非常多的一个操作。

Python提供了非常灵活简单的字符串截取方式。

字符串截取语法如下：

变量[头下标:尾下标]

这里的下标，就是指的某个字符在字符串中对应的索引值。支持双向索引。我们还是以“hello，world！”这个字符串为例，其索引如下图所示：



正向索引，从0开始，往后递增；反向索引，从-1开始，往前递减。我们只需要明确要截取的子串对应的开始索引和结束索引，即可将其截取出来。

这里需要注意一点，python的截取语法，是“前闭后开”的。其截取的子串包含“头下标”对应的字符，但是却不包含“尾下标”对应的字符。这点需要特别注意，容易出错。

比如，我们要截取“hello”这个子串。[0:5]、[0:-7]、[-12:5]、[-12:-7]，这几种方式可以达到相同的效果。

下面列出了一些使用的例子，大家可以参考：

# 字符串截取

str3 = 'hello, world!'  
sub\_str1 = str3[0:5] # 截取 hello  
sub\_str2 = str3[-6:-1] # 截取 world  
print(sub\_str1)  
print(sub\_str2)  
print(str3[5]) # 输出第6个字符  
print(str3[:5]) # 输出第6个字符之前的所有字符  
print(str3[5:]) # 输出第6个字符及以后的所有字符  
print(str3[-5:]) # 输出倒数第5个字符及以后的所有字符

### 分割

有时候我们需要对一个字符串依照某种规则进行分割，得到若干个子串。比如，我们以逗号为分割标识，将“hello,world!”分割为两个子串“hello”和“world！”。

Python提供了split方法实现字符串分割功能，其语法为：

变量.split("分割标示符号"[分割次数])

# 字符串分割 split

sub\_str\_list = str3.split(',')  
print(sub\_str\_list)

输出为：

['hello', ' world!']

Split会返回一个列表结构，这个结构里面存储了分割之后的所有子串。如果找不到分割标识符号，则返回的列表中只有一个元素，就是原始字符串。

看下面的例子，我们指定分割次数后会怎么样：

# 字符串分割 split

sub\_str\_list = str3.split(',')  
print(str3.split('o'))  
print(str3.split('o', 1))

输出为：

['hell', ', w', 'rld!']

['hell', ', world!']

如果我们不指定分割次数，会分割为3个子串。如果我们指定只分割1次，则被分割成了2个子串。

Split只能满足一些简单固定的分割规则，对于比较复杂的规则，我们可以采用正则表达式，它的功能就非常强大了。后面我们会专门拿一个章节来讲解正则表达式，这里不展开了。

### 连接

我们通常使用下面两种方式来实现字符串的连接：

1. 通过加号+实现
2. 通过join方法实现

语法： str.join(sequence)

sequence -- 要连接的元素序列，可以是元组或者列表。

str1.join([s1, s2, s3])

它的连接结果是：

s1-str1-s2-str1-s3

如下实例所示：

# 字符串连接

str3=’ hello, world!’

str4 = ' I am Tiger.'  
print(str3 + str4 + " Glad to see you ! ")

print(''.join([str3, str4, " Glad to see you ! "]))

print('\*\*'.join(["aa", "bb", "cc"])) # aa\*\*bb\*\*cc

输出为：

hello, world! I am Tiger. Glad to see you !

hello, world! I am Tiger. Glad to see you !

aa\*\*bb\*\*cc

这两种方法实现的效果是一样的，但是他们的实现逻辑却有很大的区别。基于效率的考虑，如果连接的字符串超过2个，建议尽量采用join方法。

加号连接字符串，每连接一个字符串时，系统会分配一次内存，如果连接N个字符串，那么需要分配N-1次内存，性能消耗较大。

而join方法，则是在一开始会计算列表中所有字符串的总长度并一次分配好内存，所以它的性能会高一些。

### 替换

Python使用replace()函数来实现字符串的替换，其语法为：

str.replace(old, new[, max])

* old -- 将被替换的子字符串。
* new -- 新字符串，用于替换old子字符串。
* max -- 可选字符串, 替换不超过 max 次

执行成功后，返回替换后的新字符串。

比如下面的实例，我们将 “hello，world！”替换为 “hello，python！”：

# 字符串替换

str5 = 'python'  
print(str3.replace('world', str5))  
print(str3.replace('l', 't')) # 将l替换为t，不限制替换次数  
print(str3.replace('l', 't', 2)) # 将l替换为t，限制最多替换2次

输出为：

hello, python!

hetto, wortd!

hetto, world!

同样，我们也可以使用正则表达式来实现更加强大的字符串替换功能，正则表达式比较复杂，我们可以先往后放一放，继续下面的学习。

### 查找

Python提供了多种方式来实现字符串的查找，下面我们结合实例分别介绍。

* find()方法

语法 ： str.find(sub\_str, beg=0, end=len(string))

* + sub\_str– 需要查找的子串
  + beg -- 开始索引，默认为0。
  + end -- 结束索引，默认为字符串的长度。

如果查找成功，则返回子串开始的索引值，如果失败，则返回-1。

# 字符串查找

str6 = 'hello, python'  
sub\_str\_find = 'll'  
sub\_str\_not\_find = 'world'  
print(str6.find(sub\_str\_find)) # ‘ll’匹配到str6的起始索引是2，所以返回2  
print(str6.find(sub\_str\_find, 3)) # 指定从str6的索引为3开始，所以查找不到，返回-1  
print(str6.find(sub\_str\_not\_find)) # world字符串不在str6里面，返回-1

* index()方法

语法 ： str.index(sub\_str, beg=0, end=len(string))

* + sub\_str– 需要查找的子串
  + beg -- 开始索引，默认为0。
  + end -- 结束索引，默认为字符串的长度。

它和find（）方法是一样的，只不过如果查找不到，find（）方法返回-1，而index（）会抛出一个异常（Exception）。关于python异常的处理，我们在后面章节会介绍，这里不需要深究。

print(str6.index(sub\_str\_find, 3)) # 指定从str6的索引为3开始，所以查找不到，抛出一个

异常

抛出异常：

Traceback (most recent call last):

File "D:/跟我一起学python/练习/5/5\_4.py", line 54, in <module>

print(str6.index(sub\_str\_find, 3)) # 指定从str6的索引为3开始，所以查找不到，抛出一个异常

ValueError: substring not found

正式的代码里面，我们应该对这个异常进行处理，如果不处理程序会报错并退出。

* rfind()方法和rindex()方法

它们的使用方式和find()、index()一样，唯一的区别是它们是反向查找的（从右向左），返回的是子串最后一次出现的位置。

可以看看下面的实例：

str7 = 'hello, python, hello'

print(str7.find('ell')) # 正向查找，返回第一次查找到的索引  
print(str7.rfind('ell')) # 反向查找，返回第一次查找到的索引

它的输出为：

1

16

反向查找时，返回的是，从右到左匹配到的第一个子串的首字符索引。

需要注意的是，字符串查找返回的都是正向索引的值。这点不要和字符串截取中，反向索引的负数值搞混淆了。

我们同样可以使用正则表达式实现更加强大的字符串查找功能。

### 格式化输出

Python最常用的输出方式是print()，语法如下。

print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

参数说明：

file: 输出的目标文件流，缺省为系统的标准输出.  
sep: 插入到不同value之间的字符，缺省是空格符.  
end: 最后一个value之后的字符，缺省是\n换行.  
flush: 强制刷新标记.

我们主要修改第一个参数value，它的表达方式是： “格式化字符串”% 参数列表。

如果参数列表是多个，则用小括号括起来，“格式化字符串”% (参数1, 参数2, …)

“格式化字符串”由我们期望输出的字符串和一系列格式化符号组成，下面是格式化符号列表：

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

相匹配的，python提供了一系列辅助指令：

|  |  |
| --- | --- |
| **符号** | **功能** |
| \* | 定义宽度或者小数点精度 |
| - | 用做左对齐 |
| + | 在正数前面显示加号( + ) |
| <sp> | 在正数前面显示空格 |
| # | 在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X'(取决于用的是'x'还是'X') |
| 0 | 显示的数字前面填充'0'而不是默认的空格 |
| % | '%%'输出一个单一的'%' |
| (var) | 映射变量(字典参数) |
| m.n. | m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话) |

下面是一个实例：

# 格式化输出

fmt\_1ist = [10.2, 99, 'hello']  
print('this is example for fmt output \n %.4f, %#0x, %10s' % (fmt\_1ist[0], fmt\_1ist[1], fmt\_1ist[2]))

格式化输出本身是比较简单的，大家只要掌握了基本方法，需要的时候查上面的表即可。

另外，上面的方式如果参数过多的话，代码的可读性会非常差。Python提供了str.format方法，相对看起来要简单一些。这种方式也提供了很多格式控制符，大家可以自行百度，我们不一一列举了。

# str.format

print('this is example for fmt output \n {:.4f}, 0x{:x}, {:>10}'.format(fmt\_1ist[0], fmt\_1ist[1], fmt\_1ist[2]))

Python3.6之后又引入了一种新的格式化输出方法：f-string。它比上面方式要简单许多，这里不做介绍，大家可以去百度。

### 其它常用操作

字符串还有很多操作，我们不可能一一讲解，大家可以在需要使用的时候去查询相关的手册。

|  |  |
| --- | --- |
| 功能 | 方法 |
| str.strip() | 删除字符串两边的指定字符 |
| str.lstrip() | 删除字符串左边的指定字符 |
| str.rstrip() | 删除字符串右边的指定字符 |
| in | 是否包含指定字符串 |
| len(str) | 字符串长度 |
| str.lower() | 转换为小写 |
| str.upper() | 转换为大写 |
| str.swapcase() | 大小写互换 |
| str.capitalize() | 首字母大写 |
| str.center() | 将字符串放入中心位置可指定长度以及位置两边字符 |
| str.count() | 统计字符串中出现某个子串的次数 |
| str \* num | 使用\*来复制字符串，num表示复制次数 |
| str.startswith(prefix[,start[,end]]) | 是否以prefix开头 |
| str.endswith(suffix[,start[,end]]) | 以suffix结尾 |
| str.isalnum() | 是否全是字母和数字，并至少有一个字符 |
| str.isalpha() | 是否全是字母，并至少有一个字符 |
| str.isdigit() | 是否全是数字，并至少有一个字符 |
| str.isspace() | 是否全是空白字符，并至少有一个字符 |
| str.islower() | 是否全是小写 |
| str.isupper() | 是否全是大写 |
| str.istitle() | 是否是首字母大写的 |
| str.partition() | 分割，前中后三部分 |
| str.splitlines() | 根据换行执行分割 |
| str.zfill() | 返回指定长度的字符串，原字符串右对齐，前面填充0 |
|  |  |

## Bytes （字节）

### Bytes类型

Bytes是python3新增的一个数据类型，用于表示一个字节串，它是一个有序的序列。

通常有两种方式来构造一个bytes类型的对象：

1. 通过bytes()函数构造

bytes\_1 = bytes('hello', 'utf-8')  
bytes\_2 = bytes([1, 200, 80, 50])

1. 通过b后面跟字符串的方式

bytes\_3 = b'world'  
bytes\_4 = b'\x77\x6f\x72\x6c\x64'

我们在print一个bytes类型数据时，python会以/x的格式依次打印每个字节的值，以两位16进制来显示。但是python对于一些字符会直接字符编码转换，所以造成打印出来的结果看起来很混乱，比如：

bytes\_2 = bytes([1, 200, 80, 50])

print('bytes\_2:', bytes\_2)

输出结果为：

bytes\_2: b'\x01\xc8P2'

最后两个数值80、50，被转换为了字符P、2，看起来很混乱。

这时，我们可以写一个简单的方法，让它不做这种转换：

# bytes 按照16进制输出，强制不ascii转码  
def trans(s):  
 return "b'%s'" % ''.join('\\x%.2x' % x for x in s)

bytes\_2 = bytes([1, 200, 80, 50])

print('bytes\_2:', trans(bytes\_2))

输出结果为：

bytes\_2: b'\x01\xc8\x50\x32'

这样我们看到，bytes里面包含了一个一个的字节。

因为我们还没有学函数的概念，所以大家只要知道在输出的时候调用这个方法即可。

bytes类型，存储的是一系列的字节，它并不关注这些字节具体表示什么含义（字符、网络数据、图片、音视频等）。Bytes并不约束你如果使用这些字节数据，你可以按照你自己的功能逻辑做任意的转换。这个转换逻辑，不是bytes数据类型的功能范畴。

比如：对于字符，通常我们需要对其做一个编码转换，将字节类型转换为有意义的字符串。这个转换规则，就是字符编码，紧接着下一小节我们会介绍字符编码。

我们可以看到，bytes类型也是一种序列，所以它的大多数操作方法和String一致。

# 操作方法  
print(bytes\_3[0: 3])  
print(bytes\_1 + bytes\_3)  
print(b'h' in bytes\_1)  
print(bytes\_1.split(b'l'))  
print(bytes\_1.find(b'll'))  
print(bytes\_1.replace(b'l', b't'))

输出结果为：

b'wor'

b'helloworld'

True

[b'he', b'', b'o']

2

b'hetto'

是不是和string类型高度一致？ bytes类型和string类型的对比如下：

1. string的基本单位是字符，bytes的基本单位是字节；
2. 他们都是属于一种序列，所以对于序列的操作方法，对他们基本都适用；
3. String和bytes都是不可变类型，不能对其元素进行修改。

注意，虽然bytes通常会和string一起使用，但是bytes并不只是给string用，它本质上是一个字节串。Bytes适合那种面向二进制流的存储数据，比如图片、视频等多媒体，或者网络通信等二进制报文流。

### 字节序

字节序，顾名思义就是字节存储的顺序。大家可能觉得奇怪，字节不都是“从左到右”依次存储的吗？怎么会有字节序的问题？大家看看下面的例子：

# author: Tiger， wx ID：tiger-python

# file: ./6/6\_2.py  
  
# bytes 按照16进制输出，强制不ascii转码  
def trans(s):  
 return "b'%s'" % ''.join('\\x%.2x' % x for x in s)  
  
  
# 字节序  
byte\_1 = 'python'.encode('utf-8')  
print(trans(byte\_1))

print('Big endian: ', hex(int.from\_bytes(byte\_1, byteorder='big', signed=False)))  
print('Little endian: ', hex(int.from\_bytes(byte\_1, byteorder='little', signed=False)))

输出结果为：

b'\x70\x79\x74\x68\x6f\x6e'

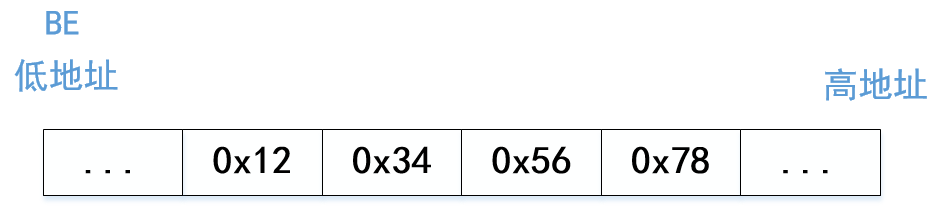
Big endian: 0x707974686f6e

Little endian: 0x6e6f68747970

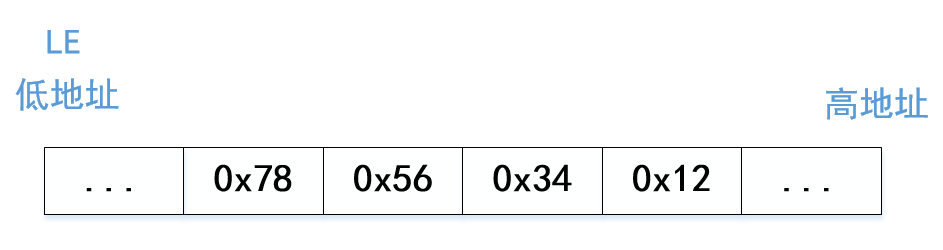
上面的实例中，我们将bytes类型b’python’强制转换为int类型，在转换过程中分别指定其字节序为big和little。从打印结果可以看出，这两种类型对应的输出结果完全相反。它们对应的就是大端字节序（Big endian，BE）和小端字节序（Little endian，LE）。

比如我要存储一个字节串：b’\x12\x34\x56\x78’:

大端字节序：从低地址到高地址，依次存储数据字节；



小端字节序：相反，从高地址到低地址，依次存储数据字节。



因为我们查看内存通常是由低位地址向高位地址看，所以大端字节序是更加符合我们的习惯的，而小端则相反。

为什么计算机会产生两种不同的字节序呢？

因为字节序是由CPU架构决定，而在计算机技术发展初期，CPU架构的两大阵营X86和PowerPC分别采用了完全相反的两种字节序，X86采用了LE，PowerPC采用了BE。所以，才会导致我们现在需要面对字节序的问题。

我们可以下面的方法获取当前cpu的字节序类型：

# 获取当前cpu的字节序类型

import sys  
print('endian of cur env:', sys.byteorder)

输出为：

endian of cur env: little

我使用的环境是X86的CPU，对应的是小端字节序。

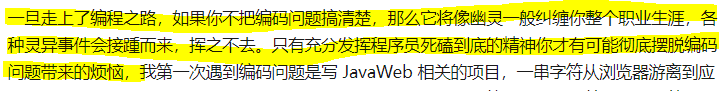
如果你的程序只会在本地运行，不会涉及到跨主机（跨不同类型CPU）的操作，那么你不需要关注字节序。反之，你需要特别关注字节序，因为它容易出错。

如果计算机A采用了BE架构的CPU，计算机B采用了LE架构的CPU。我们有一段程序，在计算机A发送一个bytes ： b’\x12\x34\x56\x78’给计算机B，那么计算机B解析出来的数据将是bytes ： b’\x78\x56\x34\x12’，这就完全错了。

在这种跨主机的数据传输中的字节序，我们通常称之为网络字节序，网络字节序和CPU无关，它是网络通信协议定义的一套规范。几乎所有的网络字节序都采用了大端字节序BE。计算机将数据发送给网络协议之前，需要统一转换为网络字节序，同样，接收端的计算机从网络接收到数据后，也会统一将其由网络字节序转换为本机字节序。这样，我们就解决了跨主机的字节序问题。Python的网络编程里面，我们还会涉及到字节序，到时候我们可以回头来看看。

### 字符编码

Python2.x的乱码问题一直被程序员所诟病，虽然Python3.X大体上已经解决了这个问题，但是作为入门python的基础，你还是需要把字符编码问题搞得非常清楚，否则你会被各种“乱码”搞崩溃。下面是引用的知乎上面一个回答，挺有意思：



* 什么是字符编码呢？

我们知道计算机是使用01这样的二进制串来存储数据的，这里的数据不仅仅是数字，而是所有数据，包括图片、视频、音频、文字等等。所以，我们需要有一个规则，来定义这些数据和二进制之间如何转换。

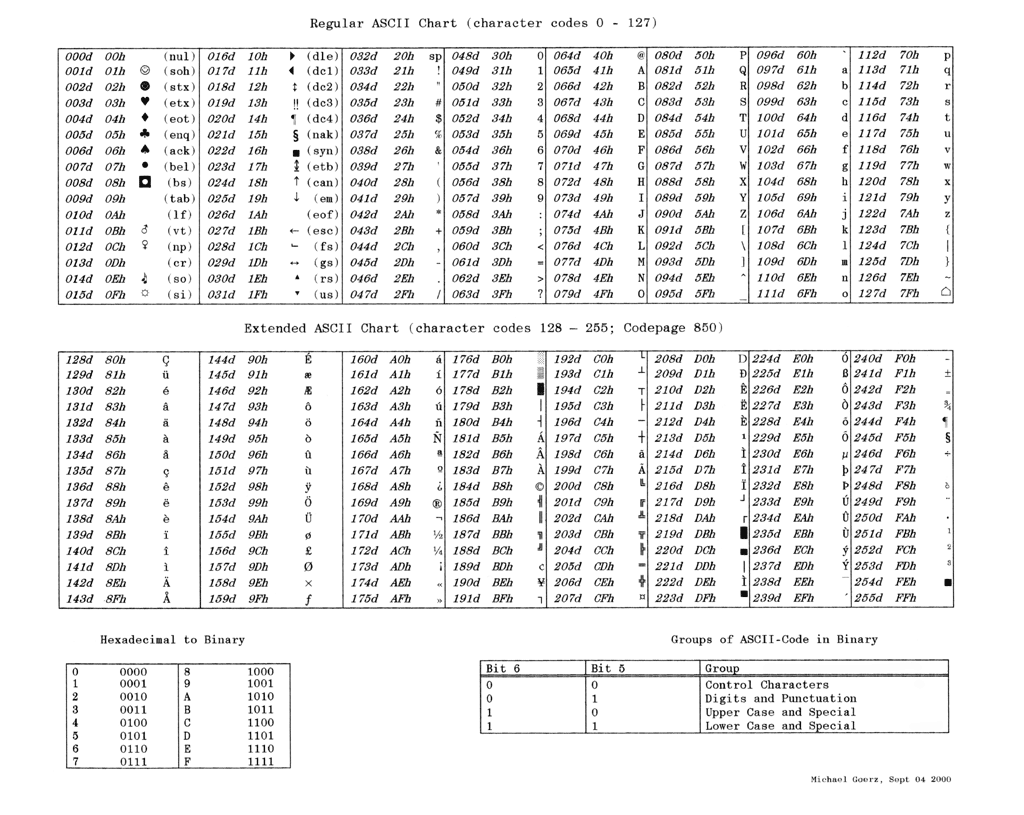
字符编码，就是一套标准（事实上有若干套标准），根据这些标准，我们将字符和二进制双向转换。

最早的字符编码标准，叫做ASCII码。

* ASCII码

计算机技术起源于美国，所以最早的字符编码标准ASCII（American Standard Code for Information Interchange)也是老美根据他们的语言体系制定的。在英语的世界里面，26个小写字母、26个大写字母、若干个标点符号、一些控制符号等就可以表示所有字符。

下面就是全部的ASCII编码，一共就只有256个符号的编码定义：



第一列是字符对应的十进制数值，比如001d；

第二列是字符对应的十六进制数值，比如0Ah；

第三列是对应的字符；

0~31还定义了第四列，用于表示一些特殊的控制符，比如008d对应的是退格（Backspace），我们敲键盘的Backspace退格键，就会产生008d这个数值在计算机中传递。

最早 ASCII 只定义了128个字符编码，包括96个显示字符和32个控制符号，一共128个字符，只占用了一个字节8bit位中的低7位。然而随着计算机慢慢普及，一些西欧字符在字符集中无法被表示，于是就有了下面那种扩展ASCII表，将字符集扩展到了256个。

* GB2312\GBK

ASCII码标准的指定可没有考虑中文的问题，于是当计算机在中国以及亚洲国家出现以后，中文编码成了一个问题。为了解决这个问题，我国制定了自己的字符编码标准GB2312(GB-国标)。GB2312收录的汉字基本覆盖了我们使用的绝大多数场景，但是对于繁体字和少数民族文字无法处理。于是后来又制定了GBK标准(K-扩展)，GBK在GB2312基础上制定，它除了收录27484个汉字以外，还收录了藏文、蒙文、维吾尔文等少数民族文字。

GBK采用两个字节来表示中文，同时它也兼容ASCII标准，当使用ASCII字符时可以自动转换为一个字节来表示。



* Unicode

不同国家都有自己的语言和文字，如果大家都自己制定自己的字符编码规范，那它们之间如何互通呢？为了解决这个问题，科学家们发明一种统一的字符编码-unicode（UniversalMultiple-OctetCodedCharacterSet，简称UCS），它几乎囊括了人类所有语系字符的编码。Unicode同样也是兼容ascii码的。

Unicode采用2字节编码，这一标准的2字节形式叫做UCS-2。但是我们知道2字节最多只能表示65535个字符编码，这对于一些特殊语系来说是完全不够的，比如中文字符就有几万个。所以，unicode还有一种4字节的形式，叫做UCS-4，它有足够的空间可以定义人类所有的文字符号。

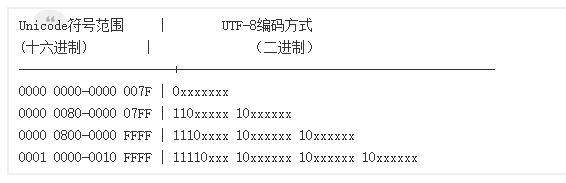
* UTF-8

Unicode是一个字符集，它仅定义了不同符号对应的编码值。但是这个字符该如何存储呢？对于ascii码，我们可以统一采用一个字节来存储不同的字符。Unicode字符也可以这样操作，但这样会带来一个存储效率的问题。

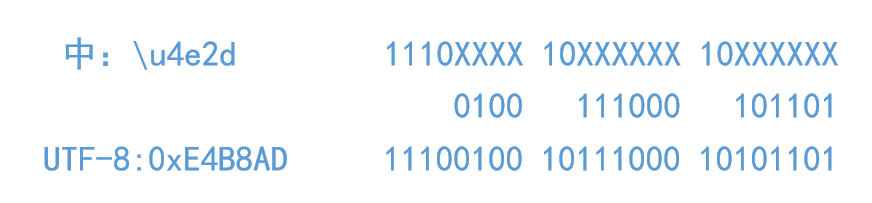
比如字符“a”，它对应的编码是61，一个字节就可以表示。如果我们采用4字节表示，那么有3个字节就浪费掉了。在英语系国家，这种浪费就显得非常严重，接近3/4的存储性能损失，这是不能被接受的。

基于这个原因，出现了各种unicode的实现方法，最著名的是UTF-8。

UTF-8以一个字节（8比特位）为一个编码单位，并且它是变长的。对于不同的unicode字符，它会采用不同的字节长度来存储。这样就可以大大提升存储性能。



比如汉字“中”，它的unicode表示和utf-8编码如下：



除了UTF-8，还有UTF-16，同理，它是以16比特-2字节为一个编码单位。

字符乱码的原因，基本都是字符编码不一致导致的。Python2默认采用ascii码，中文字符默认是不能转换的，所以我们在python2代码中如果涉及中文字符，必须在代码文件一开始就进行字符编码的声明：

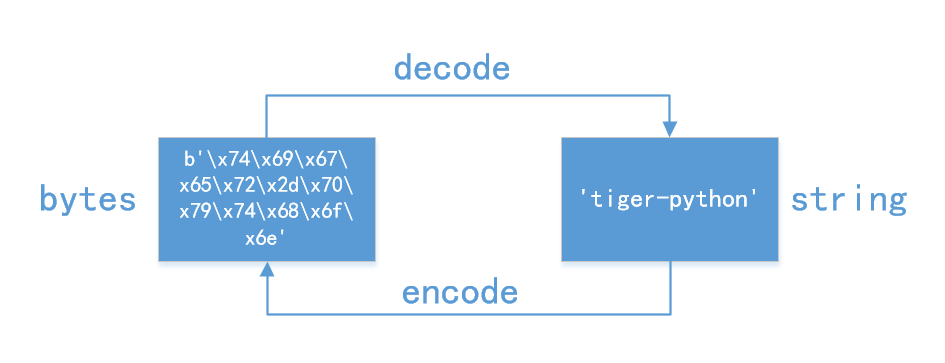
# -\*-coding:utf-8-\*-

Python2的字符编码问题备受诟病。在python3中，默认编码改成了utf-8，所以我们不声明编码类型，也是可以正常支持中文字符的。

字符编码需要端到端保持一致，不管是存储、计算、传输等环节都要设置一致，比如我们代码处理时采用的字符编码必须和数据库存储的编码方式一致。否则很容易出现乱码。

### Bytes和string之间的转换

Bytes和string之间的转换，其本质就是一个字符编码的过程。Python3提供了encode和decode方式来实现两者之间的灵活转换，其过程如下图：



Encode是字符编码过程，将字符转换为指定的编码，并以bytes类型存储；

Decode则相反，将bytes值转换为对应的字符。

语法如下：

str.encode(encoding='UTF-8',errors='strict')

str.decode(encoding='UTF-8',errors='strict')

encoding -- 要使用的编码方式，如"UTF-8"。

errors -- 设置不同错误的处理方案。默认为 'strict',意为编码错误引起一个UnicodeError。 其他可能得值有 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' 以及通过 codecs.register\_error() 注册的任何值。

下面是一个编解码的例子：

# author: Tiger， wx ID：tiger-python

# file: ./6/6\_3.py  
  
# bytes 按照16进制输出，强制不ascii转码  
def trans(s):  
 return "b'%s'" % ''.join('\\x%.2x' % x for x in s)  
  
  
# 字符编码  
encstr\_1 = 'tiger-python'  
encbytes\_1 = encstr\_1.encode('utf-8')  
print(trans(encbytes\_1))  
print(encbytes\_1.decode('utf-16')) # 编码方式不一致，造成乱码  
print(encbytes\_1.decode('utf-8'))

输出为：

b'\x74\x69\x67\x65\x72\x2d\x70\x79\x74\x68\x6f\x6e'

楴敧⵲祰桴湯

tiger-python

如果编解码的编码方式不一致，则出现乱码。

## List（列表）

列表是python中非常常用的一个数据结构，它的语法如下：

[item1, item2, item3, …]

由中括号将所有列表元素括起来，不同的元素之间通过逗号分隔。

列表中的元素item，支持几乎所有类型的数据，并且同一个列表中的所有元素可以是不同的数据类型。所以列表使用起来会非常灵活。用过C语言数组结构的同学应该知道，数组结构只能存储同一类型的元素，比如整型数组、字符串数组等等。另外，C语言的数组结构一旦初始化之后，是不能动态扩容的。C语言也可以实现列表功能，但它不是C语言的标准数据类型。相比较起来，Python的数据类型要强大和灵活得多。

列表本质上是一种序列，前面我们学习的string字符串本质上也是一种序列，还有下一节的tuple元组也是序列。我们来看看序列都有一些什么样的共性呢？

* 序列具备索引，正向索引和反向索引，前面字符串截取时我们学习过。
* 序列都支持切片（分割、截取）。
* 序列具备一些通用的操作（加、乘、检查成员）

通过下来的例子我们演示列表的常用操作：

# file: ./5/5\_5.py  
  
*"""  
演示列表的操作  
"""*# list 列表  
list\_1 = ['hello', 100, ['跟我一起学', 4]] # 支持不同类型的item，可以嵌套list  
list\_2 = ['python', '!']  
  
print(list\_1[0:2]) # 截取的方式和字符串一致  
print(list\_1[0:-2])  
print(list\_1[0: 1])  
print(list\_1[0])  
  
print(len(list\_1)) # 获取列表的长度  
  
# 列表连接  
list\_3 = list\_1 + list\_2  
print(list\_3)  
  
# 使用乘法让列表重复n次  
list\_4 = list\_2 \* 3  
print(list\_4)  
  
# 判断一个元素是否存在于列表中  
print('python' in list\_2) # True  
  
# 判断一个元素在列表中出现的次数  
print(list\_4.count('python'))  
  
# 获取列表中最大最小值,求和  
list\_5 = [1, 2, 4, 10, 90]  
print(max(list\_5))  
print(min(list\_5))  
print(sum(list\_5))  
  
# 列表的增删改操作  
list\_5.append(100) # 在列表尾增加元素100  
print(list\_5)  
  
list\_5.insert(1, 'insert\_obj') # 把元素插入到索引为1的位置  
print(list\_5)  
  
list\_6 = ['hello', 'python']  
list\_5.extend(list\_6) # 在列表后面追加另外一个列表  
  
list\_5[0] = 200 # 将索引为0的元素修改为200  
print(list\_5)  
  
del list\_5[0] # 删除索引为0的元素  
print(list\_5)  
  
list\_5.pop(2) # 移除索引为2的元素，如果不填写索引值，则默认移除列表最后一个元素  
print(list\_5)  
  
list\_5.remove('insert\_obj') # 移除一个元素，注意这里指定的是元素的值。如果列表中有多个相同的值，则只移除第一个匹配项  
print(list\_5)  
  
list\_5.clear() # 清空整个列表  
print(list\_5)  
  
# 列表的排序操作  
list\_7 = [100, 99, 27, 198, 3]  
  
list\_7.reverse() # 列表反向排列  
print(list\_7)  
  
list\_7.sort() # 列表升序排列  
print(list\_7)  
  
list\_7.sort(reverse=True) # 列表降序排列  
print(list\_7)

由于同一个列表可以支持不同的元素，所以某些列表操作会有一些限制，大家在使用时需要注意。比如一些数值操作，如sum(list)，它就无法支持list中包含字符串的情况，因为字符串没法求和。一个比较特殊的操作，最大值max和最小值min，它们支持字符串的比较，那么它们是按照什么规则来比较大小的呢？我们通过一小段代码测试一下。

# file: ./5/5\_6.py  
  
# max\min 如何比较字符串列表  
list\_1 = ['a', 'b', 'cat', '跟我一起学python']  
  
print(ord('a'), ord('b'), ord('c'), ord('跟'))  
  
print(max(list\_1))  
print(min(list\_1))

可以看出，对于字符串列表，是按照元素的首字符对应的ASCII编码值来比较大小的（参考字符编码章节）。如果同一个列表中混杂了数字和字符串，则无法比较，会抛出异常。

列表是python中用得最多的标准数据类型，后面我们在讲循环语句时还会介绍如果对列表进行迭代操作。大家应该对列表操作勤加练习，熟能生巧。

## Tuple（元组）

元组，也是一种序列结构，它和列表非常类似，但是它不能被改变。也就是说，我们不能对元组中的元素进行修改。元组的语法如下：

(item1, item2, item3, …)

由中括号将所有列表元素括起来，不同的元素之间通过逗号分隔。

元组的语法和列表是不是很像，唯一的差别就是它用的小括号，而列表用的中括号。其实不然，元组还有一些特殊的地方，比如下面两个元组的定义：

# file: ./5/5\_7.py  
  
# 元组的操作  
tuple\_1 = (1, 2, 100, 'python')  
tuple\_2 = 'hello', 'python', 100 # python中，未加括号的序列都默认是元组  
tuple\_3 = (98,) # 只包含一个元素的元组，需要加上逗号，否则小括号会被认为是运算符

元组的操作和列表几乎是一样的，我们就不一一列出了，大家可以参考列表去练习。

下面我们主要通过实例来看看元组“不可改变”的特性。

# 元组的元素不可改变

tuple\_4 = (1, 5, 10)  
# tuple\_4[0] = 3 # 会抛出异常

上面代码运行时会抛出异常：

TypeError: 'tuple' object does not support item assignment

明确提示，元组对象不支持元素的改变。

但是，元组元素可以是可变数据类型，如下操作是合法的：

# 元组的元素可以是可变数据类型

tuple\_5 = (100, 200, [300, 400])  
tuple\_5[2][0] = 500  
print(tuple\_5)

再看下面这个例子：

tuple\_4 = (1, 5, 10)

tuple\_4 = (100, 99)  
print(tuple\_4)

这个例子可以正常运行，并且输出我们预期的正确结果。这说明tuple\_4可以被改变吗？还记得我们在前面讲变量的哪些实例吗，我们用同样的方法把tuple\_4这个变量对应的内存地址打印出来，看看它究竟发生了什么变化？

tuple\_4 = (1, 5, 10)

print(id(tuple\_4))  
tuple\_4 = (100, 99)  
print(tuple\_4)  
print(id(tuple\_4))

输出结果为：

2812724308608

(100, 99)

2812723061120

我们可以看到，tuple\_4在重新赋值后，它对应的内存改变了。这个行为和数字、字符串是不是很像呢？没错，这一类数据类型我们称为“不可变数据类型”。

不可变数据类型对应的内存中的数据是不能被修改的，就像tuple\_4一样，它重新被赋值后，实际上是被指向了一块新的内存。原来的那块内存中的数据依然没有改变。

后面我们拿一个小节来总结一下可变数据类型和不可变数据类型。

## Dictionary（字典）

字典，也是Python中使用得比较广泛的一种数据类型。字典本质上是一种“键值对”（key-value）的集合。“键值对”这种数据的描述方式，更加符合我们对客观世界的认识。客观世界的数据，通常都存在一个名字-key，以及对应的值-value，使用“键值对”可以非常直观简便地表达这些数据。一些数据库技术也是基于键值对的数据存储方式，比如Redis、memcached。“键值对”在大数据领域也有广泛应用。

字典的结构定义和JSON基本一致，两者可以很方便的相互转换。

字典的语法如下：

{key1: value1, key2: value2, key3: value3, …}

字典使用大括号{}括起来，每个key-value之间使用逗号分割。Key和value之间使用冒号：分割。

同一个字典里面，key值需要保证唯一，不能重复，并且只能是不可变数据类型（数字、字符串、元组）。

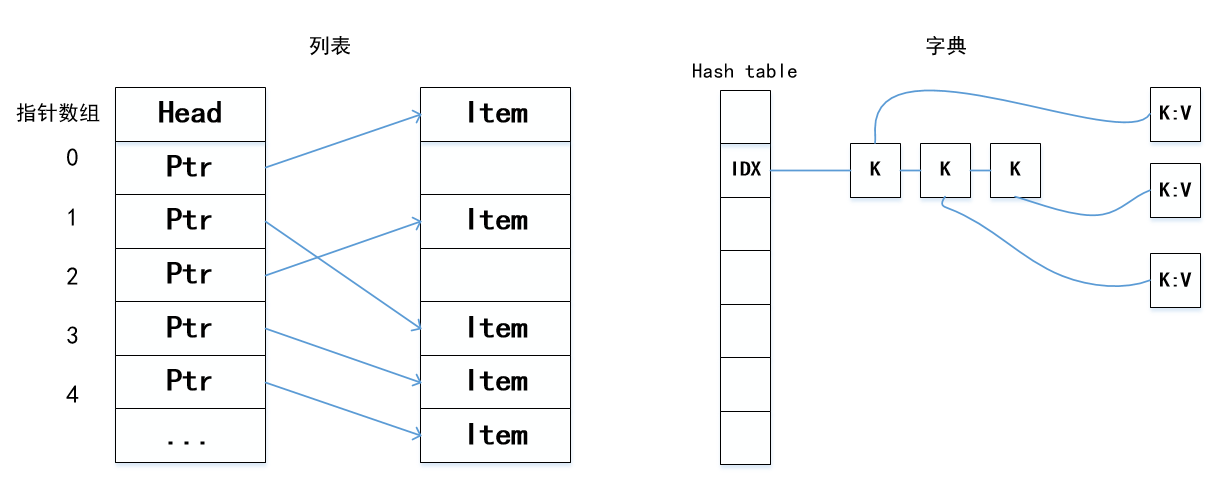
Value不要求唯一，并且可以是任意数据类型。

字典不是一种序列，它没有索引，并且是无序的，这一点和列表存在本质区别，大家要注意。

实质上，字典在python的底层实现中，是一种hash索引结构。所以字典结构在查询、新增、删除、修改上面表现出更加优异的时间性能，但是它会额外消耗部分内存去保存hash表，这是典型的空间换时间的设计。

而列表在python的底层实现中，是一种动态数组结构。该数组元素不会直接保存列表元素的数据，而是存储了列表元素的地址。地址的大小是固定的，所以它可以是一个数组结构。由于它是动态数组（支持resize内存空间），所以也可以支持扩容。同时，数组是连续存储的，所以列表存在索引，是有序的。

下图示意了字典和列表的区别，不同的python版本可能存在差异，但是大体类似。



列表采用了指针数组，它是一段连续内存，所以它可以像数组一样去索引的。而字典采用了hash表结构，key值是通过hash函数去命中到一个hash index，所以它只能通过key值去查找。

真正的实现中，不会像我画的这么简单。比如字典的hash表还需要考虑hash散列、hash冲突，但是这些细节并不影响我们理解字典和列表之间的本质区别。对于初学者，不建议大家现在去搞懂这些细节，可以留待后面再去深入理解。

下面我们通过一系列实例来熟悉字典的操作：

# file: ./5/5\_8.py  
  
# 字典  
dict\_1 = {'name': 'xiaowang', 'age': 20, 'city': 'Chengdu'}  
print('name: %s, age: %d, city: %s' % (dict\_1['name'], dict\_1['age'], dict\_1['city']))  
  
dict\_1['city'] = 'Beijing'  
print('name: %s, age: %d, city: %s' % (dict\_1['name'], dict\_1['age'], dict\_1['city']))  
  
dict\_2 = {}.fromkeys(('name', 'age', 'city')) # 批量根据key值初始化一个字典  
print(dict\_2)  
  
print(len(dict\_1))  
  
# 增加键值对  
dict\_1['sex'] = 'male'  
print(dict\_1)  
  
# 删除键值对  
del dict\_1['sex']  
print(dict\_1)  
  
# 批量更新  
dict\_3 = {'name': 'xiaoliu', 'age': 18, 'city': 'Chengdu', 'sex': 'female'}  
dict\_1.update(dict\_3)  
print(dict\_1)  
  
# 判断key是否在字典中存在  
print('name' in dict\_3)  
  
# 获取所有的key  
print(list(dict\_1.keys()))  
  
# 获取所有的value  
print(list(dict\_1.values()))  
  
# 获取所有的(键, 值) 元组列表  
print(list(dict\_1.items()))  
  
# 清空字典  
dict\_1.clear()  
print(dict\_1)

python提供了copy()方法，用于字典的浅复制（shallow copy）。与浅复制相对应的，还有深复制(deep copy)的概念。我们通过实例来理解一下二者的区别。浅复制和深复制的概念在很多编程语言中都涉及，大家应该理解并举一反三。

# file: ./5/5\_10.py  
  
# 深复制和浅复制  
dict\_1 = {'name': 'xiaowang', 'age': 20, 'score': {'yuwen': 90, 'shuxue': 100}}  
dict\_2 = dict\_1.copy()  
  
print('dict\_1:', dict\_1)  
print('dict\_2:', dict\_2)  
  
# 修改value  
print('modify 90->98'.center(64, '-'))  
dict\_1['score']['yuwen'] = 98  
print('dict\_1:', dict\_1)  
print('dict\_2:', dict\_2)  
  
# 深拷贝  
print('deep copy'.center(64, '-'))  
  
import copy  
dict\_3 = copy.deepcopy(dict\_1)  
print('dict\_1:', dict\_1)  
print('dict\_3:', dict\_3)  
print('modify 98->99'.center(64, '-'))  
dict\_1['score']['yuwen'] = 99  
print('dict\_1:', dict\_1)  
print('dict\_3:', dict\_3)

输出结果为：

dict\_1: {'name': 'xiaowang', 'age': 20, 'score': {'yuwen': 90, 'shuxue': 100}}

dict\_2: {'name': 'xiaowang', 'age': 20, 'score': {'yuwen': 90, 'shuxue': 100}}

-------------------------modify 90->98--------------------------

dict\_1: {'name': 'xiaowang', 'age': 20, 'score': {'yuwen': 98, 'shuxue': 100}}

dict\_2: {'name': 'xiaowang', 'age': 20, 'score': {'yuwen': 98, 'shuxue': 100}}

---------------------------deep copy----------------------------

dict\_1: {'name': 'xiaowang', 'age': 20, 'score': {'yuwen': 98, 'shuxue': 100}}

dict\_3: {'name': 'xiaowang', 'age': 20, 'score': {'yuwen': 98, 'shuxue': 100}}

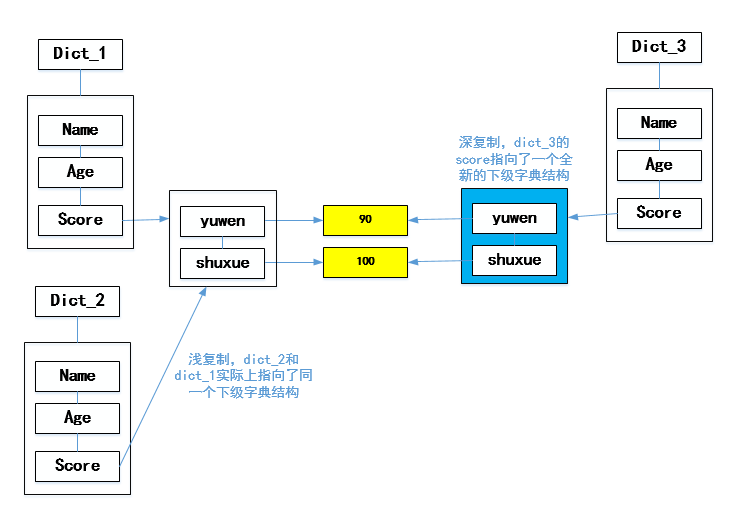
-------------------------modify 98->99--------------------------

dict\_1: {'name': 'xiaowang', 'age': 20, 'score': {'yuwen': 99, 'shuxue': 100}}

dict\_3: {'name': 'xiaowang', 'age': 20, 'score': {'yuwen': 98, 'shuxue': 100}}

所谓浅复制，其实是复制了内存的地址，并没有拷贝内存里面存的数据。而深复制，则是重新分配了一块内存，并把数据拷贝进去。所以，我们通过上面的实例可以看到，在浅复制的情况下，我们改变dict\_1的值，dict\_2依然会跟着变化，说明他们指向了同一块内存空间。而深复制的情况下，则互不影响，说明他们指向的是不同的内存空间。

通过下面的图，更加容易理解：



大家可以思考一下，为什么key为‘name’和‘age’的值不受浅复制的影响呢？

## Set（集合）

在python中，集合是一个无序的不可重复的元素组合，它是不是和前面介绍的字典的key值很像呢？集合的语法如下：

{item1, item2, item3,…} 或者 set(iterable)

集合中的元素必须保证唯一，不可重复。

对于空集合，必须使用set()来创建，而不是使用{}，因为这对应的是一个空字典。

Set的参数，是一个可迭代对象，后续我们在迭代器章节会详细介绍，目前我们学习到的序列类型结构，如字符串、元组、列表，都是可迭代的。参数中不能包含重复元素，否则自动覆盖。比如：

# file: ./5/5\_9.py  
  
# 集合 Set  
set\_1 = set('hello')  
print(set\_1)

输出结果为：

{'l', 'e', 'o', 'h'}

我们可以看到，它的输出是无序的，并且重复的元素‘l’算作一个元素，是不可重复的。

集合的这种特性很适合用来对字典的key赋值：

dict\_1 = {}.fromkeys(set\_1) # 批量根据key值初始化一个字典

print(dict\_1)

输出结果为：

{'h': None, 'l': None, 'e': None, 'o': None}

可以把集合理解为没有value只有key的字典。

集合中的元素必须是可hash的，也就是说元素是不可变的，这一点也和字典的key一样。比如列表就不能作为集合中的元素。

集合分为可变集合和不可变集合，使用frozenset()创建的是不可变集合，其它都是可变集合。不可变集合不支持增加、删除等操作。

下面通过实例来熟悉集合的一些操作：

# file: ./5/5\_9.py  
  
# 集合 Set  
set\_1 = set('hello')  
print(set\_1)  
print(len(set\_1))  
  
dict\_1 = {}.fromkeys(set\_1) # 批量根据key值初始化一个字典  
print(dict\_1)  
  
set\_2 = frozenset('hello') # 不可变集合  
print(set\_2)  
  
# 新增元素  
set\_1.add('t')  
print(set\_1)  
set\_1.update('k', 'p') # 批量添加多个  
print(set\_1)  
  
# 删除元素  
set\_1.remove('t')  
print(set\_1)  
  
# 判断元素是否存在  
print('t' in set\_1)  
print('t' not in set\_1)  
  
# 清空元素  
set\_1.clear()  
print(set\_1)  
  
# 集合操作  
set\_3 = set('hello')  
set\_4 = set('world')  
  
print(set\_3 - set\_4) # 补集  
print(set\_3 | set\_4) # 并集  
print(set\_3 & set\_4) # 交集  
print(set\_3 ^ set\_4) # 对称补集， 等同于 (set\_3 - set\_4) | (set\_4 - set\_3)  
  
set\_5 = {1, 2}  
set\_6 = {1, 2, 3}  
print(set\_5 < set\_6) # 子集判断  
print(set\_5 > set\_6) # 超集判断

集合提供了类似数学里面集合概念的一些操作，比如并集、交集、子集、超集等，这在某些特殊场景下使用起来会非常方便。

## 数据类型转换

数据类型转换，指的是通过某种方法，将一个数据由原来的类型转换为另外一个类型。比如，我们将字符串“123”转换为数字123，这就是一种数据类型的转换。

Python支持各种标准数据类型之间的转换，但并不是任意数据都可以转换的，所有的转换要符合“常理”，逻辑上应该是成立的。比如，你不应该试图将一个complex类型转换为int，因为python也不知该怎么转换。

下面我整理了python3数据类型之间转换的支持情况：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Int | Float | Bool | Complex | String | List | Tuple | Set | Dict |
| Int | - | Y | Y | N | Y | N | N | N | N |
| Float | Y | - | Y | N | Y | N | N | N | N |
| Bool | Y | Y | - | Y | Y | Y | Y | Y | Y |
| Complex | Y | Y | Y | - | Y | N | N | N | N |
| String | Y | Y | Y | Y | - | Y | Y | Y | Y |
| List | N | N | N | N | Y | - | Y | Y | Y |
| Tuple | N | N | N | N | Y | Y | - | Y | Y |
| Set | N | N | N | N | Y | Y | Y | - | Y |
| Dict | N | N | N | N | Y | Y | Y | Y | - |

下面代码列举了各种类型之间的转换方法：

# file: ./5/5\_11.py  
  
'''  
数据类型转换  
'''  
  
# 转换为int  
print('int'.center(30, '\*'))  
print(int(1.2)) # float -> int  
print(int('123')) # string -> int  
print(int(True)) # bool -> int  
  
# 转换为float  
print('float'.center(30, '\*'))  
print(float('1.2')) # string->float  
print(float(123)) # int->float  
print(float(False)) # bool->float  
  
# 所有基本类型都可以转换为bool  
print('bool'.center(30, '\*'))  
print(bool(1)) # int->bool  
print(bool(0.0)) # float->bool  
print(bool(0 + 0j)) # complex->bool  
print(bool('')) # string->bool, 空字符串为False,其它都是True  
print(bool([])) # list->bool, 空为False,其它都是True  
print(bool(())) # tuple->bool, 空为False,其它都是True  
print(bool({})) # dict->bool, 空为False,其它都是True  
print(bool(set())) # set->bool, 空为False,其它都是True  
  
# 转换为complex  
print('complex'.center(30, '\*'))  
print(complex(100)) # int->complex  
print(complex(1.2)) # float->complex  
print(complex(True)) # bool->complex  
print(complex('1.2+2.3j')) # string->complex  
  
# 所有基本类型都可以转换为string  
print('string'.center(30, '\*'))  
print(str(1)) # int->string  
print(str(1.2)) # float->string  
print(str(True)) # bool->string  
print(str(1.2 + 2.3j)) # complex->string其它都是True  
print(str(['hello', 100])) # list->string  
print(str(('hello', 100))) # tuple->string  
print(str({'name': 'xiaowang', 'age': 20})) # dict->string  
print(str({'name', 'age'})) # set->string  
  
# 转换为list  
print('list'.center(30, '\*'))  
print(list("hello")) # string->list  
print(list((100, 200, 300))) # tuple->list  
print(list({'name', 'age'})) # set->list  
print(list({'name': 'xiaowang', 'age': 20})) # dict->list， 只取key值  
  
# 转换为tuple  
print('tuple'.center(30, '\*'))  
print(tuple("hello")) # string->tuple  
print(tuple([100, 200, 300])) # list->tuple  
print(tuple({'name', 'age'})) # set->tuple  
print(tuple({'name': 'xiaowang', 'age': 20})) # dict->tuple， 只取key值  
  
# 转换为set  
print('set'.center(30, '\*'))  
print(set("hello")) # string->set  
print(set([100, 200, 300])) # list->set  
# print(set([100, 200, [300, 400]])) # list->set, list中包含可变数据类型，报异常  
print(set(('name', 'age'))) # tuple->set  
# print(set(('name', 'age', []))) # tuple->set,包含可变数据类型，报异常  
print(set({'name': 'xiaowang', 'age': 20})) # dict->set， 只取key值  
  
# 转换为dict的方法略微复杂一些  
print('dict'.center(30, '\*'))  
  
# string->dict  
user\_str = '{"name": "xiaowang", "city": "Chengdu", "age": 28}'  
import json  
print(json.loads(user\_str)) # 方式一、使用json转换，字符串格式需要严格按照json格式来  
print(eval(user\_str)) # 方式二、使用eval函数转换，eval有安全隐患，不建议使用  
import ast  
print(ast.literal\_eval(user\_str)) # 方式三、 使用ast.literal\_eval  
  
# list->dict  
user\_keys = ['name', 'city', 'age']  
user\_values = ['xiaowang', 'Chengdu', 28]  
print(dict(zip(user\_keys, user\_values))) # 方式一、需要用到zip  
  
user\_info = [  
 ["name", "xiaowang"],  
 ["city", "Chengdu"],  
 ["age", 28]  
 ]  
print(dict(user\_info)) # 方式二、二维列表  
  
  
# set->dict tuple->dict的方式和list->dict一样

## 可变数据类型和不可变数据类型

前面我们的学习中其实已经涉及到可变数据类型和不可变数据类型的概念了，这一节我们总结一下。

python的数据类型分为mutable（可变） 和 immutable （不可变）：

* mutable包括list、dictionary、set
* immutable包括 number、string、tuple

这里的可变与不可变，指的是该对象对应内存中的那块数据（value）是否允许改变。

如果是不可变类型数据，系统会重新分配一个对象，并将该对象的引用（可以理解为内存地址\指针）重新赋给变量。而可变类型则可以直接修改，不会生成新的对象。

在函数传参时，两者也存在差别。如果入参是不可变类型，那么是传值方式，如果入参是可变类型，那么传入的是对象的引用。这一点我们在讲函数的章节也会介绍到。

## 动态类型和静态类型

编程语言分为动态类型语言和静态类型语言。

静态类型：在编译阶段检查变量的数据类型，比如C、Java等。对于这种类型的语言，我们需要在代码中定义变量的数据类型，不管是显式的声明还是隐式的定义。静态类型语言的变量，一旦定义了数据类型，在运行时是不能动态改变的。比如C语言，它的典型用法如下：

void main()

{

int score1 = 0; // 变量score1在使用之前需要声明为int类型，否则编译报错

int score2 = 50;

score1 = score2 + 10;

// ......

}

动态数据：在运行阶段才检查变量的数据类型。Python就是一种动态数据语言，它的变量不需要声明数据类型，只有在运行时才动态确定其类型。通过上一节的几个小实验我们其实可以看出来，python的变量本质上是一个内存地址，本身没有类型，它的数据类型由其指向的对象来决定。比如下面的例子，在python中是可以正常运行的：

# 此时，tmp是一个整数类型  
tmp = 100  
print(type(tmp))  
  
# 此时，tmp是一个字符串类型，类型可以动态改变  
tmp = "hello, world!"  
print(type(tmp))

它的输出是：

<class 'int'>

<class 'str'>

这种动态类型的特性，让python使用起来会更加简单，代码会更加简洁。但是它也会带来一些负面影响，比如无法在编译阶段发现问题，同时动态类型会带来运行时的一部分性能损耗。两者各有利弊。我们在使用python编程时，也建议随时搞清楚变量对应的数据类型，这样可以规避一些隐藏的问题。

# 运算符

# 控制语句

# 异常

# 函数

# IO编程

# 模块

# 正则表达式