**python3教程 – 廖雪峰**

**1 Python解释器**

**1.1 CPython**

当我们从Python官方网站下载并安装好Python 3.x后，我们就直接获得了一个官方版本的解释器：CPython。这个解释器是用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。

CPython是使用最广的Python解释器。教程的所有代码也都在CPython下执行。

**1.2 IPython**

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。好比很多国产浏览器虽然外观不同，但内核其实都是调用了IE。

CPython用**>>>**作为提示符，而IPython用**In [序号]:**作为提示符。

**1.3 PyPy**

PyPy是**另一个Python解释器**，它的目标是执行速度。PyPy采用JIT技术，对Python代码进行动态编译（注意不是解释），所以可以显著提高Python代码的执行速度。

绝大部分Python代码都可以在PyPy下运行，但是PyPy和CPython有一些是不同的，这就导致相同的Python代码在两种解释器下执行可能会有不同的结果。如果你的代码要放到PyPy下执行，就需要了解PyPy和CPython的不同点。

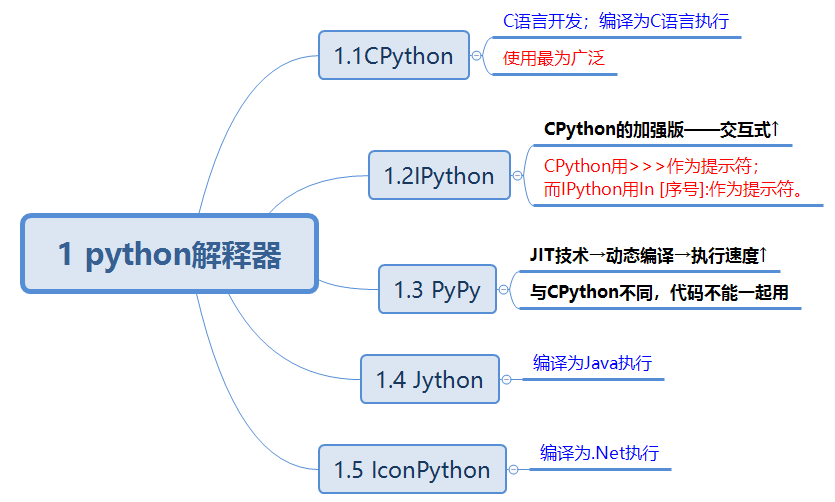
**1.4 Jython**

Jython是运行在Java平台上的Python解释器，可以直接把Python代码编译成Java字节码执行。

**1.5 IronPython**

IronPython和Jython类似，只不过IronPython是运行在微软.Net平台上的Python解释器，可以直接把Python代码编译成.Net的字节码。

**1.6 小结**

Python的解释器很多，但使用最广泛的还是CPython。如果要和Java或.Net平台交互，最好的办法不是用Jython或IronPython，而是通过网络调用来交互，确保各程序之间的独立性。

本教程的所有代码只确保在CPython 3.x版本下运行。请务必在本地安装CPython（也就是从Python官方网站下载的安装程序）。

**2 第一个Python程序**

**2.1 命令行模式和Python交互模式**

**2.1.1 命令行模式(cmd,console)**

在Windows开始菜单选择“命令提示符”，就进入到命令行模式，它的提示符类似C:\>：

┌────────────────────────────────────────────────────────┐

│Command Prompt - □ x │

├────────────────────────────────────────────────────────┤

│Microsoft Windows [Version 10.0.0] │

│(c) 2015 Microsoft Corporation. All rights reserved. │

│ │

│C:\> \_ │

│ │

└────────────────────────────────────────────────────────┘

**2.1.2 Python交互模式**

在命令行模式下敲命令python，就看到类似如下的一堆文本输出，然后就进入到Python交互模式，它的提示符是>>>。

┌────────────────────────────────────────────────────────┐

│Command Prompt - python - □ x │

├────────────────────────────────────────────────────────┤

│Microsoft Windows [Version 10.0.0] │

│(c) 2015 Microsoft Corporation. All rights reserved. │

│ │

│C:\> python │

│Python 3.7 ... on win32 │

│Type "help", ... for more information. │

│>>> \_ │

│ │

└────────────────────────────────────────────────────────┘

**2.2 建立文件**

文件必须要以.py结尾，此外，文件名只能是英文字母、数字和下划线的组合。

**2.3 输入和输出Input/Output(IO)**

**2.3.1 输出 print()**

用print()在括号中加上**字符串**，就可以向屏幕上输出指定的文字。比如输出'hello, world'，用代码实现如下：(单引号和双引号均可)

>>> print('hello, world')

print()函数也可以接受多个字符串，用逗号“,”隔开，就可以连成一串输出：(print中逗号“,”会转换成空格” ”)

>>> print('The quick brown fox', 'jumps over', 'the lazy dog')

The quick brown fox jumps over the lazy dog

print()中也可以直接输入计算：

a, b = 1, 5

print((a + b) \*\* a)

**2.3.2 输入input()**

**input(‘str’)**会弹出str再让你输入，python3中input()得到的是字符串，

name = input('please enter your name: ')

print('hello,', name)

**3 Python基础**

**3.1 数据类型和变量**

**3.1.1 数据类型**

|  |  |
| --- | --- |
| 整数(integer, int) | 有时候用十六进制表示整数比较方便，十六进制用0x前缀和0-9，a-f表示，例如：0xff00，0xa5b4c3d2，等等。 |
| 浮点数(有理数, Floating Point, float) | 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 |
| 字符串(string, str) | 字符串是以**单引号’或双引号”括起来的任意文本**；可以用转义字符\来标识；可在字符串前加r(raw)表不转义 |
| 布尔值(bool) | True, False；bool()转换时，除了0、空集合、空字符串、None，皆为T |
| 空值 | None |

**3.1.2变量**

（1）可以赋值成任一数据类型；变量名必须是大小写英文、数字和\_的组合，且不能用数字开头。

（2）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。

（3）赋值语句x=x+2，应先计算右侧x+2，再赋值给x。

（4）赋值变量的过程：当我们写：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'。

**3.1.3常量**

在Python中，通常用全部大写的变量名表示常量：比如常用的数学常数π→PI就是一个常量。

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。

**3.2 字符串和编码**

**3.2.1 字符编码**

**1、字符编码的历史**

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。

最早的英文字母+数字+符号→**ASCII编码**；中文→**GB2312**；日文→**Shift\_JIS**；韩文→**Euc-kr**，最后导致了乱码。→为了解决乱码问题，**Unicode**把所有语言都统一到一套编码里，最常用的是用两个字节表示一个字符。但如此，全英文文档浪费空间。→为了节约空间，UTF-8(可变长编码)应运而生，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。如此UTF-8中实际上包含了ASCII，故其支持ASCII编码。

**2、三个字符编码的差别**

**ASCII编码**：最早；1个字节

**Unicode编码**：解决乱码；2个字节(ASCII转换过来只需要在前面补0即可)

**UTF-8编码**：节约空间；一般，英文1个字节，汉字3个字节；可以支持ASCII编码(将Unicode中英文字母去除前面补的0)

eg:

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

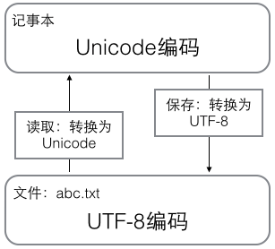
**3、字符编码的工作方式**

如此，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**：

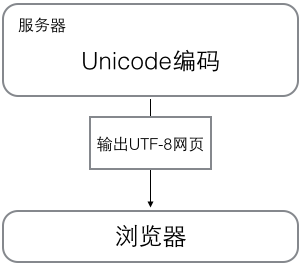
（1）在计算机内存中，统一使用**Unicode编码**；

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**。

（2）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：

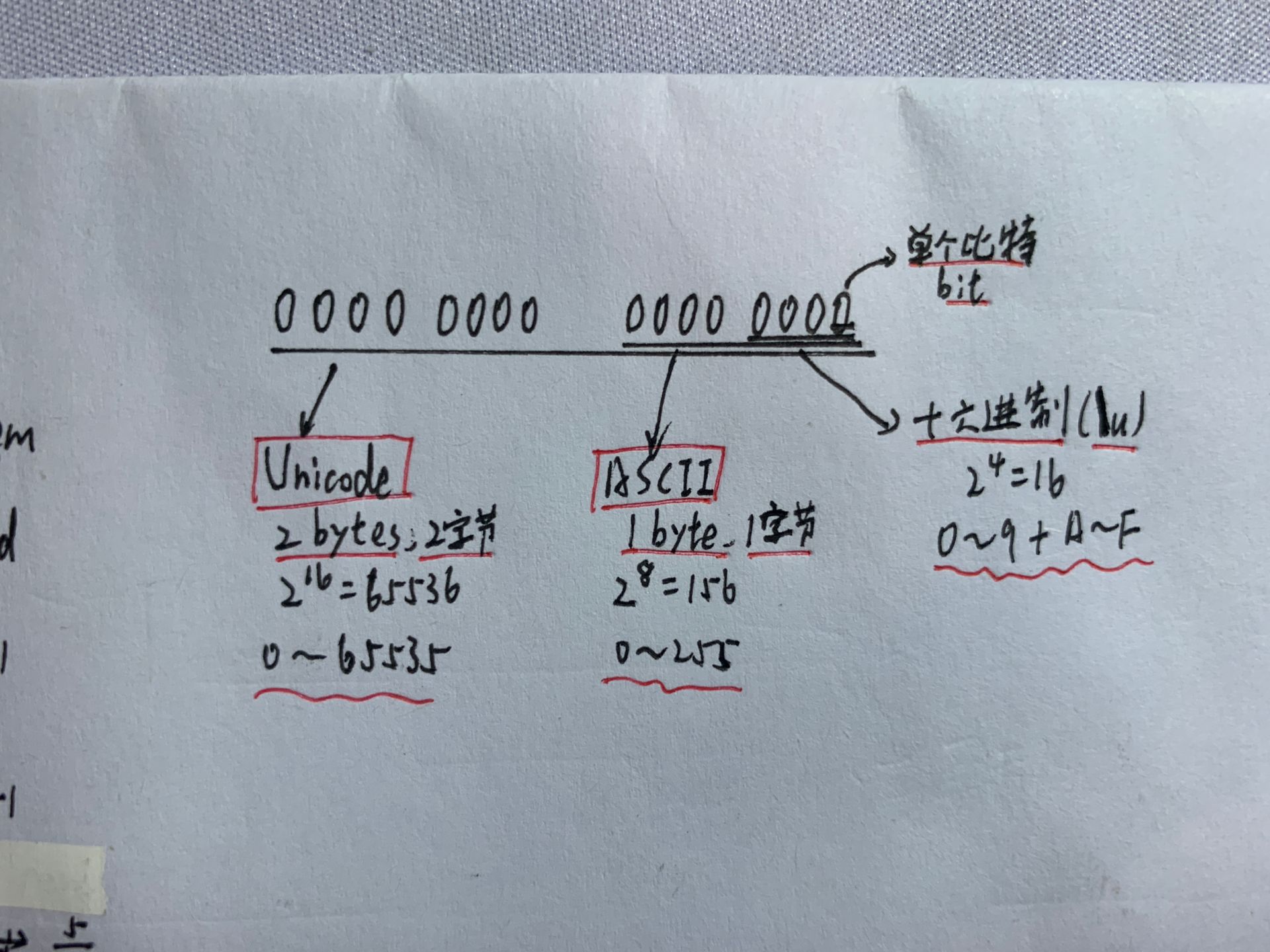
（3）浏览网页的时候，服务器会把动态生成的**Unicode**内容转换为**UTF-8**再传输到浏览器(本地)；

所以你看到很多网页的源码上会有类似<meta charset="UTF-8" />的信息，表示该网页正是用的**UTF-8编码**。

**3.2.2 字符串的编码解码**

在Python 3版本中，字符串是以**Unicode编码**的，也就是说，Python的字符串支持多语言。

**1、字符的整数编码**



对**单个字符的编码**，Python提供了**ord()**函数获取字符的整数表示()，**chr()**函数把编码转换为对应的字符：

>>> ord('A')

65

>>> ord('中')

20013

>>> chr(66)

'B'

>>> chr(25991)

'文'

如果知道字符的整数编码，还可以用**十六进制**(\u)这么写str：

>>> '\u4e2d\u6587'

'中文'

**2、python的bytes类型**

Python对**bytes类型**的数据用带b前缀的单引号或双引号表示：

x = b'ABC'

|  |  |  |
| --- | --- | --- |
|  | ‘ABC’ | b’ABC’ |
| Type | Str | Bytes |
| 内存中编码 | Unicode | Ascii |
| 内存中大小 | 每字符2字节 | 每字符1字节 |
| 保存到硬盘or传输 | Utf-8 | Utf-8 |

**3、编码解码(encode和decode)**

**str.encode**(‘#编码’)→**byte**；**encode**是将**str**编码成电脑看得懂的**byte**

**byte.decode**(‘#编码’)→**str**；**decode**是将**byte**解码成我们看得懂的**str**

**（1）将str给encode成byte时**：①符合ASCII者，可直接用ASCII/UTF-8编码；②所有str，可用UTF-8编码；③由于python3用的Unicode，不能在用其编码

**（2）将byte给decode成str时**：①ASCII者，可直接用ASCII/UTF-8解码；②所有str，可用UTF-8编码；③由于python3用的Unicode，不能在用其解码④如果我们从网络上读取了字节流，应根据其注释选取相应的编码解码。

在**bytes类型中**，无法显示为ASCII字符的字节，用\x##显示。

>>> 'ABC'.encode('ascii')

b'ABC'

>>> '中文'.encode('utf-8')

b'\xe4\xb8\xad\xe6\x96\x87'

>>> b'ABC'.decode('ascii')

'ABC'

>>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')

'中文'

如果bytes中包含无法解码的字节，decode()方法会报错。如果bytes中只有一小部分无效的字节，可以传入errors='ignore'忽略错误的字节：

>>> b'\xe4\xb8**\xe4\xb8\xad**\xad '.decode('utf-8', errors='ignore')

'中'

**len()**函数计算的是str的字符数，如果换成bytes，就计算字节数：

>>> len(b'ABC')

3

>>> len(b'\xe4\xb8\xad\xe6\x96\x87')

6

>>> len('中文'.encode('utf-8'))

6

可见，1个中文字符经过UTF-8编码后通常会占用3个字节，而1个英文字符只占用1个字节。

**4、python源码开头两行注释**

当Python**解释器读取源代码**时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行：

*#!/usr/bin/env python3  
# -\*- coding: utf-8 -\*-*

第一行注释是为了**告诉Linux/OS X系统**，这是一个Python可执行程序，**Windows系统会忽略**这个注释；

第二行注释是为了**告诉Python解释器**，按照UTF-8编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

**3.2.3 字符串的格式化**

**1、用占位符%格式化**

|  |  |
| --- | --- |
| 占位符 | 替换内容 |
| %d | 整数 |
| %f | 浮点数 |
| %s | 字符串 |
| %x | 十六进制整数 |

几个%跟几个变量；+表带符号；用%转义；

格式化整数d时，0表补0，+表带符号，d前面一位数表位数(本身、±、补0都占位数)；

格式化浮点数时，+表带符号，f前一位表保留几位小数

print(**'%+03d-%2d+%%'** % (-3, 1))→-03- 1+%%  
print(**'%.3f'** % 3.1415926)→3.142

**2、用format()格式化**

（1）str中用{}，也用{}转义，可加0123，也可不加，也可指定参数名

print(**'{} {}'**.format(**'hello'**,**'world'**))→hello world  
print(**'{1} {0} {1} {{转义！}}'**.format(**'it\'s 0'**, **'it\'s 1'**))→it's 1 it's 0 it's 1 **{转义！}**  
print(**"网站名：{name}, 地址 {url}"**.format(name=**"菜鸟教程"**, url=**"www.runoob.com"**))→网站名：菜鸟教程, 地址 www.runoob.com

（2）**list的format()**，**\***表参数以list的形式输入site\_list = [**'菜鸟教程'**, **'www.runoob.com'**]  
print(**'网站名：{0[0]}, 地址：{0[1]}'**.format(site\_list)) →网站名：菜鸟教程, 地址 www.runoob.com

print(**'网站名：{0}, 地址：{1}'**.format(\*site\_list)) →网站名：菜鸟教程, 地址 www.runoob.com

（3）**dic的format()**,**\*\***表参数以dic的形式输入site\_dic = {**'name'**:**'菜鸟教程'**, **'url'**:**'www.runoob.com'**}  
print(**'网站名：{0[name]}, 地址:{0[url]}'**.format(site\_dic)) →网站名：菜鸟教程, 地址 www.runoob.com  
print(**"网站名：{name}, 地址 {url}"**.format(\*\*site\_dic)) →网站名：菜鸟教程, 地址 www.runoob.com

**3.3 list和tuple**

**3.3.1 list的使用**

list用[]表示，访问list中的list**可以[1][2]**，末尾增加元素用**append()，**指定位置插入用**insert()**，也可直接**[位置]=**来赋值，删除末尾元素或指定位置元素用**pop()**，

list1 = [**''**, 1, [0, 1]]  
list1.append(**'2'**)  
list1.insert(1, [2325])  
list1.pop()  
list1.pop(1)  
list1[2][1] = 10  
list1[0] = 0

**3.3.2 tuple的使用**

tuple用()来表示。tuple和list一样是有序列表，但其指向不能变动(指向的list中元素可变)。其余访问等操作基本类似。

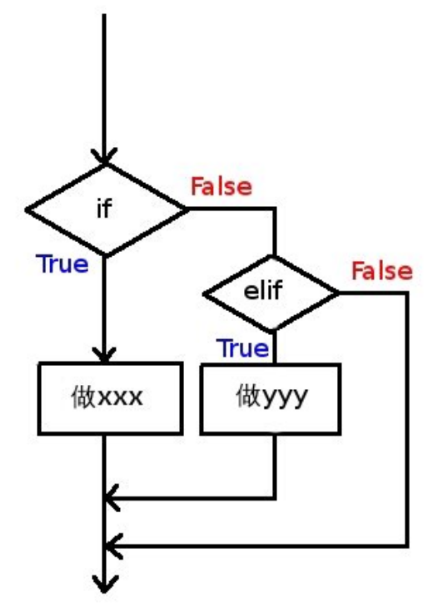
1、当tuple**只有一个元素时**，()会与计算中的()冲突，此处以计算者优先，tuple者需要加,消除歧义。

tuple1 = (1)→1

tuple1 = (1,)→(1,)

2、tuple指向不能变动，但指向的list中元素可变

tuple1 = (1, [1, 2])；tuple1[1][0] = **'01'**→(1, ['01', 2])

**3.4 条件判断**

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>

各处条件判断进行T/F判断。

**3.5 循环**

**3.5.1 for…in**

可以依次把list或tuple中的每个元素迭代出来

常用**range(start, stop[, step)**→[start, stop)，step表步长间隔(默认1)。

**3.5.2 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。

**3.5.3 break、continue、pass**

**break**语句退出此循环条件，**continue**语句直接进行下一次循环，pass语句无作用，仅做占位用。

注意break和continue会使代码执行逻辑分叉过多，容易出错，尽量改变逻辑、条件以少用这两个语句

n = 0  
**while** n < 100:  
 n += 1  
 **if** n >= 20:  
 **break  
 if** n % 2 ==0:  
 **continue** print(n)

→打印出20以下的奇数

**3.6 dic(t)和set**

**3.6.1 dic(t)的使用**

**1、dic(t)的特性**

字典dic(t)全称dictionary，在其他语言中也称为map，使用**键-值**（**key-value**）存储，有极快的查找速度。

**dic和list、tuple不同**，没有顺序，只依靠key来找value

|  |  |  |
| --- | --- | --- |
|  | Dic | List |
| 顺序 | 无顺序，依靠key找value | 有顺序可index |
| 速度 | 查找和插入的速度极快，不会随着key的增加而变慢 | 查找和插入的时间随着元素的增加而增加 |
| 占用空间 | 需要占用大量的内存，内存浪费多 | 占用空间小，浪费内存很少 |

dic的key必须是不可变对象：这是因为dic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dic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）**。key和value之间连着一个hash值，输入key得到hash值，再能找到value，而建立一个hash(0-999)需要1000的整数编码的空间，故dic元素较少时也会占用大量空间。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。

**2、dic的使用**

（1）dic用**{}表示**，增加和改变元素用**d[key]=value**，**删除dic元素只能删除key**(value也会删除掉)用pop(key)

d1 = {**'1'** : 1, 2 : **'2'**}  
d1[**'0'**] = 0  
d1[**'1'**] = **'1'**d1.pop(**'1'**)  
print(d1, d1[**'0'**])→{2: '2', '0': 0} 0

（2）要**避免key不存在的错误**，有两种办法，①是**通过in判断key**是否存在②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

print(1 **in** d1, d1.get(1), d1.get(1, -1))→False None -1

**3.6.2 set的使用**

1、set和dict类似，也是一组key的集合，但不存储value。**在set中，**没有重复的key，key是无序的。

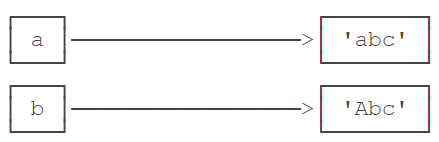
2、set用{}表示，要创建一个set，需要**set([list])**，添加元素用**add(key)**，重复的元素会被自动过滤；删除元素用**remove(key)**

s1 = set([1, 2, 3, 2, 3])  
s1.add(**'a'**)  
s1.add(**'b'**)  
s1.remove(3)  
print(s1)→{'b', 1, 2, 'a'}

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：

s1 = set([1, 2, 3])  
s2 = set([2, 3, 4])  
print(s1 & s2)→{2, 3}  
print(s1 | s2)→{1, 2, 3, 4}

**3.6.3不可变对象**

str是不可变对象，list是可变对象

a1 = **'abc'**a1.replace(**'a'**, **'A'**)  
print(a1)→abc  
b = a1.replace(**'a'**, **'A'**)  
print(b)→Abc

**当调用replace()时**，replace作用在str的’abc’上，创造了个新的str‘Abc’并返回，以此对变量b赋值，变量b便指向新的str‘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。

**4 函数**

**4.1 调用函数**

eg：

print(abs(-3))→3  
print(max(1, 3, -10, 9))→9

print(str(123), int(**'123'**), float(12), bool(0), bool(1))→123 123 12.0 False True

**4.2 定义函数**

**1、函数的组成**

def 函数名functionname(参数名 parameters ):

"函数说明：函数\_文档字符串"

function\_suite

return [表达式expression]

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。**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。

**2、在命令行中调用函数**

可以在该文件的当前目录下启动Python解释器，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）：

┌────────────────────────────────────────────────────────┐

│Command Prompt - python - □ x │

├────────────────────────────────────────────────────────┤

│>>> from abstest import my\_abs │

│>>> my\_abs(-9) │

│9 │

│>>> \_ │

└────────────────────────────────────────────────────────┘

**3、空函数**

如果一个函数还没想好具体怎么写，可用pass语句来先让代码跑起来。

**4、参数检查**

**def** my\_abs(x):  
 **if not isinstance(x, (int, float)):  
 raise TypeError('bad operand type')** **if** x >= 0:  
 **return** x  
 **else**:  
 **return** -x  
→在输入非整数和浮点数时，返回TypeError: bad operand type

**5、返回多个值**

**return** x, y →返回的是一个tuple：(x, y)；命名变量既可以r=(x, y)，也可以x1, y1 = x, y

**4.3 函数的参数**

**4.3.1 位置/必选参数**

未赋予默认值的、**调用时**必须输入的、按照位置顺序依次赋值的参数。

**def** power(x, n):  
 s = 1  
 **while** n > 0:  
 n = n - 1  
 s = s \* x  
 **return** s  
print(power(4, 3))

**4.3.2 默认参数**

**1、默认参数的性质与使用**

赋予默认值的参数；同时**调用时**也可赋值、也按照位置顺序依次赋值的参数。

设置默认参数时，有几点要注意：①必选参数在前，默认参数在后(否则赋值时不好弄)②**多个参数时**，变化小的放后面，可以作为默认参数(能降低调用函数的难度)。③**多个默认参数时**，按顺序赋予或指定赋予

**2、可变对象作默认参数时**

L指向[]，每调用一次，指向的[]都被改变，即可变参数作默认参数的结果(若想不变，可设计if加条件)。

**def** add\_end(L=[]):  
 L.append(**'END'**)  
 **return** L  
print(add\_end() \* 3)→['END', 'END', 'END']

print([1] \* 2)→[1, 1]  
print(add\_end())→['END', 'END']

**4.3.3 可变参数**

**可变参数**就是允许传入0-任意个参数。

def functionname([formal\_args,] **\*var\_args\_tuple**):

"函数\_文档字符串"

function\_suite

return [expression]

可变参数在前加\*表示，调用时，可直接输入参数，也可输入\*list/\*tuple，参数会直接组装成一个新的tuple。

**4.3.4 关键字参数**

**关键字参数**允许传入0-任意个参数，

关键字参数在前加\*\*表示，调用时，可直接输入参数key=value，也可以输入\*\*dic，参数会直接组装成一个新的dic。

**def** person(name, age, \*\*kw):

**4.3.5 命名关键字参数**

**命名关键字参数**用\*,表示/前有可变参数者可不加，可设默认值，未设者必须输入参数key=value

**def** person(name, age, \*zip, city)

**def** person (name, age, \*, zip, city=**'shanghai'**)

**4.3.6 参数组合**

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。

**def** f1(a, b, c=0, \*args, d=7, \*\*kw):  
 print(**'a ='**, a, **'b ='**, b, **'c ='**, c, **'args ='**, args, **'d='**, d,**'kw='**, kw)  
f1(1, 2, 3, 4, 5, 6, d = 7, e = 8, f = 9)→a = 1 b = 2 c = 3 args = (4, 5, 6) d= 7 kw= {'e': 8, 'f': 9}  
通过**一个tuple和dict**，你也可以调用上述函数。**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**

t1 = (1, 2, 3, 4, 5, 6)  
d1 = {**'d'**: 70, **'e'**: 7, **'f'**: 8}  
f1(t1, d1)→a = (1, 2, 3, 4, 5, 6) b = {'d': 70, 'e': 7, 'f': 8} c = 0 args = () d= 7 kw= {}

**4.4 递归函数**

**def** fact(n):  
 **if** n==1:  
 **return** 1  
 **return** n \* fact(n - 1)

→阶乘n!

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。

**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。

**5 高级特性**

**5.1 切片**

适用于list、tuple、string，用[a:b]表示→[a,b)，可[0]，[-1]，[:]，[0:]等

**5.2 迭代：Iteration**

我们可以通过for循环来遍历list或tuple，这种遍历我们称为**迭代（Iteration）**。

其他语言需要下标才能迭代，但python无论有无下标，只要是可迭代对象，均可迭代。

可迭代对象：str，list，tuple，dic

关于dic默认**迭代key**，如果要**迭代value**，可以用for value in d.values()，如果要同时**迭代key和value**，可以用for k, v in d.items()。

如果要对list实现类似Java那样的**下标循环**可用Python内置的enumerate函数把一个list变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：

**for** i, value **in** enumerate([**'A'**, **'B'**, **'C'**]):  
 print(i, value, end=**' ;'**)

→0 A ;1 B ;2 C;

**for** i, j **in** [(1, **'a'**), (2, **'b'**), (1, **'c'**)]:  
 print(i, j, end=**' ;'**)

→1 a ;2 b ;1 c ;

**5.3列表生成器/列表综合：List Comprehensions**

**列表生成式**即**List Comprehensions**，是Python内置的非常简单却强大的可以用来创建list的生成式。

[元素生成表达式 for 参数 in 元素序列 if等筛选表达式]

print([x \* x **for** x **in** range(1, 11) **if** x % 2 == 0])→[4, 16, 36, 64, 100]

print([m + n **for** m **in 'ABC' for** n **in 'XYZ'**])→['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']

d = {**'x'**: **'A'**, **'y'**: **'B'**, **'z'**: **'C'** }  
print([k + **'='** + v **for** k, v **in** d.items()])→['x=A', 'y=B', 'z=C']

**5.4 生成器：generator**

创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为**生成器：generator**。

generator用()表示，单个生成可以用**g.\_\_next\_\_()**，知道出现StopIteration的错误；一般用for…in来遍历。

g1 = (i **for** i **in** range(1, 11))

如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator。**函数**是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成**generator**的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行，同样一般用for…in来执行。

斐波那契数列：

**def** fib(max):  
 n,a,b = 0,0,1  
 **while** n<max:  
 **yield** b  
 n += 1  
 a,b = b, a+b  
 **return** print(**'done'**)  
**for** b **in** fib(6):  
 print(b, end=' ')→1 1 2 3 5 8 done

但是用for循环调用generator时，拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

g = fib(6)  
**while True**:  
 **try**:  
 x = next(g)  
 print(**'g:'**, x, end=**'。'**)  
 **except** StopIteration **as** e:  
 print(**'Generator return value:'**, e.value)  
 **break**

→g: 1。g: 1。g: 2。g: 3。g: 5。g: 8。done

Generator return value: None

**5.5 迭代器**

**1、可迭代对象：Iterable**

可以直接作用于***for循环***的对象统称为**可迭代对象：Iterable**。

一类是**集合数据类型**，如list、tuple、dict、set、str等；

一类是**generator**，包括生成器和带yield的generator function。

**2、迭代器：Iterator**

可以被***next()函数***调用并不断返回下一个值的对象称为**迭代器：Iterator**。如：generator。

把list、dict、str等Iterable变成Iterator可以使用**iter()函数**

Python的**Iterator对象**表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

**6 函数式编程**

**函数**是Python内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之**为面向过程的程序设计**。**函数**就是面向过程的程序设计的**基本单元**。

在**计算机**的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。

而计算则指**数学意义上的计算**，越是抽象的计算，离计算机硬件越远。

对应到**编程语言**，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Lisp语言。

**函数式编程**就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种**纯函数**我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

**函数式编程的**一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

**6.1 高阶函数**

高阶函数英文叫Higher-order function。

**1、变量可以指向函数**

x = abs  
print(x)→<built-in function abs>  
print(x(-1))→1

**2、函数名也是变量名**

**函数名**其实就是指向函数的变量。因此一般不用与函数名重合的变量，否则会破坏函数的调用。

**3、高阶函数**

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为**高阶函数**。

**6.1.1 map/reduce**

map(function, iterable, ...)

**map()**输入一个函数(任意参数)和参数个Iterable→输出Iterator，由于Iterator是惰性的，要用list()把它都算出来并返回一个list。

r = list(map(abs, [1,2,-5,3,-7,-8]))  
print(r)→[1, 2, 5, 3, 7, 8]

reduce(function, iterable[, initializer])

**reduce()**输入1个函数(2个参数)和1个Iterable

reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)

**6.1.2 filter**

filter(function, iterable)

**filter()函数**用于过滤序列。和map()类似，filter()也接收一个函数和一个序列。和map()不同的是，filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

**6.1.3 sorted**

排序算法

sorted(iterable, key=None, reverse=False)

sorted()和sort()区别在于一个接受Iterable，一个接受list；sorted()是一个**高阶函数**，它可以**接收一个key函数**来实现自定义的排序，key指定的函数将作用于list的每一个元素上，并根据key函数返回的结果进行排序。**reverse**=True降序，**reverse**=False升序（默认）。

默认情况下，对字符串排序，是按照ASCII的大小比较的，在ASCII中数字<大写<小写；此时可以用key=str.lower/upper对str进行整理再排序。

**6.2 返回函数**

**1、函数作为返回值返回**

**def** lazy\_sum(\*args):  
 **def** sum():  
 ax = 0  
 **for** n **in** args:  
 ax = ax + n  
 **return** ax  
 **return** sum  
f1 = lazy\_sum(1, 3, 5, 7, 9)  
f2 = lazy\_sum(1, 3, 5, 7, 9)  
print(f1 == f2)

在这个例子中，我们在函数lazy\_sum中又定义了函数sum，并且，内部函数sum可以引用外部函数lazy\_sum的参数和局部变量，当lazy\_sum返回函数sum时，相关参数和变量都保存在返回的函数中，这种称为“**闭包（Closure）**”的程序结构拥有极大的威力。

当我们调用lazy\_sum()时，每次调用都会返回一个新的函数：f1()和f2()的调用结果互不影响。

**2、闭包**

闭包概念：在一个**内部函数中**，对外部作用域的变量进行引用，(并且一般外部函数的返回值为内部函数)，那么内部函数就被认为是闭包。

**返回闭包时**牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

也可以**外函数**临时变量为可变类型(如list)，内函数与此变量绑定，然后直接改变此变量。

**6.3 匿名函数(lambda)**

lambda [arg1 [,arg2,.....argn]]:expression

lambda后跟1-任意参数，冒号后跟表达式，表示一个匿名函数。

print(list(map(**lambda** x: x \* x, [1, 2, 3])))→[1, 4, 9]

**6.4 装饰器（Decorator）**

本质上，decorator就是一个**返回函数的高阶函数**。

**@wraps(f)**能把原函数的信息复制到装饰器中；**@原函数**→原函数名=装饰器名(原函数名)，即在原函数上加了个装饰器。注意装饰器中返回内函数不加()

**from** functools **import** wraps  
**def** decorator\_name(f):  
 @wraps(f)  
 **def** decorated(\*args, \*\*kwargs):  
 **if not** can\_run:  
 **return "Function will not run"  
 return** f(\*args, \*\*kwargs)  
 **return** decorated  
  
@decorator\_name  
**def** func():  
 **return** (**"Function is running"**)  
  
can\_run = **True**print(func())  
*# Output: Function is running*can\_run = **False**print(func())  
*# Output: Function will not run*

**6.5 偏函数**

**偏函数（Partial function）**可以通过设定参数的默认值，降低函数调用的难度。

eg:

class int(x, base=10)

int()函数包含参数base表示str符合N进制，默认十进制。

print(int(**'af'**, 16))→175  
print(int(**'1001'**,2))→9

此时可以用functools来定义偏函数。

**import** functools

**对int()**  
int2 = functools.partial(int, base=2)  
print(int2(**'1010'**))→10  
相当于：  
kw = { **'base'**: 2 }  
int(**'10010'**, \*\*kw)  
**对max()**  
max2 = functools.partial(max, 10)  
print(max2(5, 6, 7))→10  
相当于：  
args = (10, 5, 6, 7)  
max(\*args)

**python3教程 – 廖雪峰 - 目录**

**1 Python解释器**

**1.1 CPython**

**1.2 IPython**

**1.3 PyPy**

**1.4 Jython**

**1.5 IronPython**

**1.6 小结**

**2 第一个Python程序**

**2.1 命令行模式和Python交互模式**

**2.1.1 命令行模式**

**2.1.2 Python交互模式**

**2.2 建立文件**

**2.3 输入和输出Input/Output(IO)**

**2.3.1 输出 print()**

**3 Python基础**

**3.1 数据类型和变量**

**3.1.1 数据类型**

**3.1.2变量**

**3.1.3常量**

**3.2 字符串和编码**

**3.2.1 字符编码**

**1、字符编码的历史**

**2、三个字符编码的差别**

**3、字符编码的工作方式**

**3.2.2 字符串的编码解码**

**1、字符的整数编码**

**2、python的bytes类型**

**3、编码解码(encode和decode)**

**4、python源码开头两行注释**

**3.2.3 字符串的格式化**

**1、用占位符%格式化**

**2、用format()格式化**

**3.3 list和tuple**

**3.3.1 list的使用**

**3.3.2 tuple的使用**

**3.4 条件判断**

**3.5 循环**

**3.5.1 for…in**

**3.5.2 while**

**3.5.3 break、continue、pass**

**3.6 dic(t)和set**

**3.6.1 dic(t)的使用**

**1、dic(t)的特性**

**2、dic的使用**

**3.6.2 set的使用**

**3.6.3不可变对象**

**4 函数**

**4.1 调用函数**

**4.2 定义函数**

**1、函数的组成**

**2、在命令行中调用函数**

**3、空函数**

**4、参数检查**

**5、返回多个值**

**4.3 函数的参数**

**4.3.1 位置/必选参数**

**4.3.2 默认参数**

**1、默认参数的性质与使用**

**2、可变对象作默认参数时**

**4.3.3 可变参数**

**4.3.4 关键字参数**

**4.3.5 命名关键字参数**

**4.3.6 参数组合**

**5 高级特性**

**5.1 切片**

**5.2 迭代：Iteration**

**5.3列表生成器/列表综合：List Comprehensions**

**5.4 生成器：generator**

**5.5 迭代器**

**1、可迭代对象：Iterable**

**2、迭代器：Iterator**

**6 函数式编程**

**6.1 高阶函数**

**1、变量可以指向函数**

**2、函数名也是变量名**

**3、高阶函数**

**6.1.1 map/reduce**

**6.1.2 filter**

**6.1.3 sorted**

**6.2 返回函数**

**1、函数作为返回值返回**

**2、闭包**

**6.3 匿名函数(lambda)**

**6.4 装饰器（Decorator）**

**6.5 偏函数**