python3 指南

用 *python3* 玩转电脑

万泽[①] | 德山书生[②]

版本: 0.01

---

① 作者:

② 编者：邮箱：a358003542@gmail.com。

前言

xverbatim 环境目前不能进行交互，如果终端交互又不支持 pdf 的结果输出了。决定新加入一个模式，1 显示代码 3 用 geany 打开代码。后面就用编辑器调试吧。这里因为涉及到交互，看了一下 tee 命令，似乎使用“python3 test.py | tee test.py.out”这样的命令应该是可行的，但是可惜要某调用 gnome-terminal 就不能生成文件，要某不调用 texmaker 本身就弹不出对话框。现在只好设置一个简单的 13 模式，所谓 3 实际上非常的简单，就是用 geany 打开那个代码，因为 geany 编辑器调整好倒也方便，只是遇到这种交互程序的生成结果需要手动复制了。

需要提醒的是在本文档中不管是 xverbatim 环境生成的 code 文件还是 cverbatim 环境，最前面都多了一个空行，因为最前面有一个 newlinechar 符号最后转变成换行了，我还不知道如何避免，所以，如果你需要将代码文件以可执行模式执行（以脚本文件模式载入的没有问题），你需要进入文件按一下 Backspace 键将第一行消去即可。

主要参考资料：

1.python 入门教程 Python 教學文件作者：Guido van Rossum Fred L. Drake

2.A Comprehensive Introduction to Python Programming and GUI Design Using Tkinter 作者：Bruno Dufour

3.Python and Tkinter Programming JOHN E. G RAYSON

4.learning python v5 主要 python 语言参考 python 学习手册（第四版）老鼠版（略过：23，24，30，31，异常和工具，高级话题，）

5.programming python v4 蟒蛇版

6.python 官网其他参考资料（遇到问题则 google 之，这部分不会专门学习）

7.tkinter 官方文档

8.pyqt4 tutorial 英文网站 也参考了 jimmykuu 的中文翻译，中文翻译网站

目 录

python3 基础

beginning

python 简介

这是一段测试文字

Python 是个成功的脚本语言。它最初由 Guido van Rossum 开发，在 1991 年第一次发布。Python 由 ABC 和 Haskell 语言所启发。Python 是一个高级的、通用的、跨平台、解释型的语言。一些人更倾向于称之为动态语言。它很易学，Python 是一种简约的语言。它的最明显的一个特征是，不使用分号或括号，Python 使用缩进。现在，Python 由来自世界各地的庞大的志愿者维护。

python 现在主要有两个版本区别，python2 和 python3。作为新学者推荐完全使用 python3 编程，本文档完全基于 python3。

完全没有编程经验的人推荐简单学一下 c 语言和 scheme 语言（就简单学习一下这个语言的基本概念即可）。相信我学习这两门语言不会浪费你任何时间，其中 scheme 语言如果你学得深入的话甚至编译器的基本原理你都能够学到。了解了这两门语言的核心理念，基本上任何语言在你看来都大同小异了。

进入 python 的 REPL 环境

在 ubuntu13.10 下终端中输入 python 即进入 python 语言的 REPL 环境，目前默认的是 python2。你可以运行：

```
python  --version
```

来查看。要进入 python3 在终端中输入 python3 即可。

python3 命令行用法

命令行的一般格式就是：

```
python3  [可选项]  test.py  [可选参数1 可选参数2]
```

同样类似的运行`python3  --help`即可以查看 python3 命令的一些可选项。比如加入 **-i** 选项之后，python 执行完脚本之后会进入 REPL 环境继续等待下一个命令，这个在最后结果一闪而过的时候有用。后面的 -c，-m 选项还看不明白。

python 执行脚本参数的传递

上面的命令行接受多个参数都没有问题的，不会报错，哪怕你在 py 文件并没有用到他们。在 py 文件中要使用他们，首先导入 sys 宏包，然后 sys.argv[0] 是现在这个 py 文件在系统中的文件名，接下来的 sys.argv[1] 就是之前命令行接受的第一个参数，后面的就依次类推了。

geany 的相关配置

geany 的其他配置这里不做过多说明，就自动执行命令默认的应该是 python2，修改成为：

```
python3  -i  %f
```

即可。

代码注释

python 语言的注释符号和 bash 语言（linux 终端的编程语言）一样用的是 # 符号来注释代码。然后 py 文件开头一般如下面代码所示：

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

其中代码第一行表示等下如果 py 文件可执行模式执行那么将用 python3 来编译[①]，第二行的意思是 py 文件编码是 utf-8 编码的，python3 直接支持 utf-8 各个符号，这是很强大的一个更新。

多行注释可以利用编辑器快速每行前面加上 # 符号。

Unicode 码支持

前面谈及 python3 是可以直接支持 Unicode 码的，如果以可执行模式加载，那么第二行需要写上:
`#-*-coding:utf-8-*-`
这么一句。

```
#!/usr/bin/env python3
#-*-coding:utf-8-*-
print('\u2460')
```

上面的数字是具体这个 Unicode 符号的十六进制。

代码多行表示一行

这个技巧防止代码越界所以经常会用到。用反斜线 \ 即可。不过通常更常用的是将表达式用圆括号 ( ) 括起来，这样内部可以直接换行并继续。在 python

① 也就是用 chmod 加上可执行权限那么可以直接执行了。第一行完整的解释是什么通过 *env* 程序来搜索 python 的路径，这样代码更具可移植性。

中任何表达式都可以包围在圆括号中。

一行表示多行

python 中一般不用分号，但是分号的意义大致和 bash 或者 c 语言中的意义类似，表示一行结束的意思。其中 c 语言我们知道是必须使用分号的。

输入和输出

最基本的 input 和 print 命令

input 函数请求用户输入，并将这个值赋值给某个变量。注意赋值之后类型是字符串，但后面你可以用强制类型转换——int 函数（变成整数），float 函数（变成实数），str 函数（变成字符串）——将其转变过来。print 函数就是一般的输出函数。

```
x=input(' 请输入一个实数: ')
string001=' 你输入的这个实数乘以 2 等于: '+ str(float(x)*2)
print(string001)
```

程序中的逻辑

条件判断

python 中的条件语句基本格式如下:

```
if  test:
    条件判断执行区块
```

也就是 if 命令后面跟个条件判断语句，然后记住加个冒号，然后后面缩进的区块都是条件判断为真的时候要执行的语句。

```
if  test:
    do something001
else :
    do something002
```

这里的逻辑是条件判断，如果真，do something001；如果假，do something002。

```
if  test001:
    do something001
```

```
elif test002:
    do something002
```

显然你一看就明白了，elif 是 else 和 if 的结合。

逻辑与或否

and 表示逻辑与，or 表示逻辑或，not 表示逻辑否。

下面编写一个逻辑，判断一个字符串，这个字符串开头必须是 a 或者 b，结尾必须是 s，倒数第二个字符不能是单引号'。在这里就演示一下逻辑。。

code:2-1

```
x='agais'
if ((x[0] == 'a' or x[0] == 'b')
    and x[-1] =='s'
    and (not x[-2] =="'"")):
    print('yes it is..')
```

```
yes it is..
```

上面的显示效果是 xverbatim 环境 input 之后的问题，目前主要的问题就是新的一行前面的空格无法显示，想了一些方法都不行，只好作罢。

稍复杂的条件判断

现在我们了解了 if，elif 和 else 语句，然后还了解了逻辑与或非的组合判断。那么在实际编程中如何处理复杂的条件逻辑呢？

首先能够用逻辑语句与或非组合起来的就将其组合起来，而不要过分使用嵌套。然后让我们看下面几个小代码：

tcbcode-2.1

```
x=-2
if x>0:
    print('x 大于 0')
else:
    print('x 小于 0')
```

tcbcode-2.2

```
x=2
if x>0:
    print('x 大于 0')
elif x<0:
    print('x 小于 0')
else:
    print('x 等于 0')
```

tcbcode-2.3

```
x=2
if x>0:
    print('x 大于 0')
if x<0:
    print('x 小于 0')
else:
    print('x 等于 0')
```

如 tcbcode-2.1所示，如果一个情况分成两部分，那么就用 if...eles... 语句，而如果一个情况分成三部分，那么就用 if...elif...else 语句（如 tcbcode-2.2）。同一深度的这些平行语句对应的是“或”逻辑，或者说类似其他编程语言的 switch 语句。我们再看一看 tcbcode-2.3，这个代码是错误的，两个 if 语句

彼此并不构成逻辑分析关系。[①]

tcbcode-2.4

```
x=-2
if x>0:
    print('x 大于 0')
    if x>2:
        print('x>2')
    elif x<2:
        print('0<x<2')
    else:
        print('x=2')
elif x<0:
    print('x 小于 0')
else:
    print('x 等于 0')
```

然后我们看到上面的最后一个代码 tcbcode-2.4，这个例子演示的是在加深一个深度个条件判断语句它当时处于的逻辑判断情况，这个语句的条件判断逻辑是本语句的判断逻辑再和左边（也就是前面）的深度的判断逻辑的“与”逻辑，或者说成是“交集”。比如说 *print('0<x<2')* 这个语句就是本语句的判断逻辑 x<2 和上一层判断逻辑 x>0 的“交集”，也就是 0<x<2。

整个过程的情况如下图所示：

① 四个甚至更多的平行或逻辑就用更多的 elif，读者请自己实验一下。

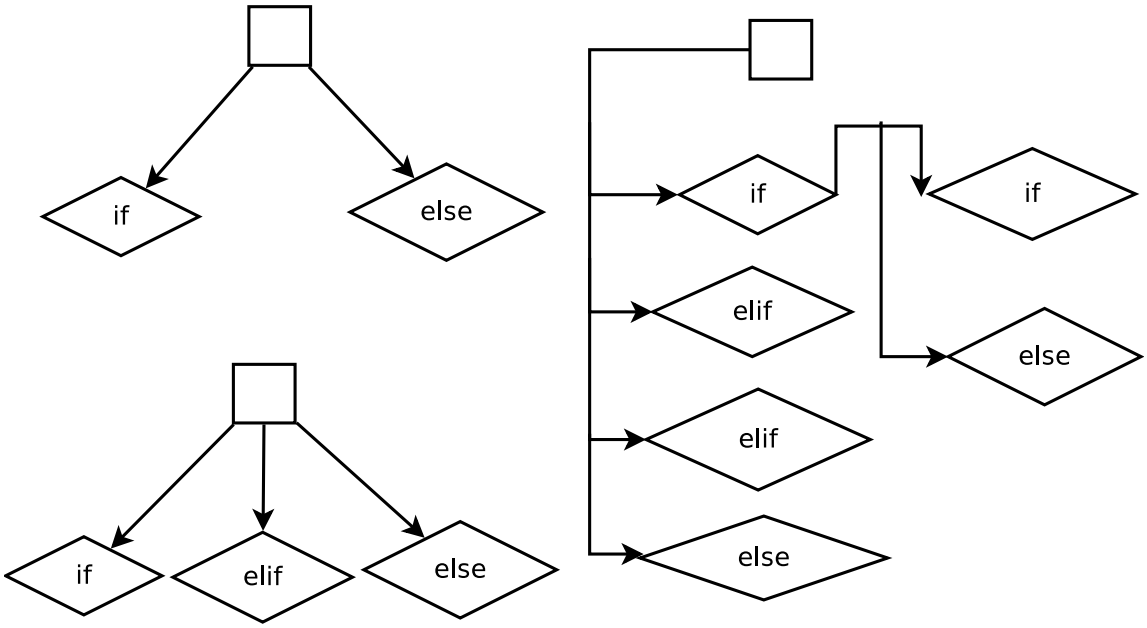


图 2-1: 复杂条件判断

为了在编程的时候对处于何种判断逻辑之下有一个清晰的认识，强烈建议读者好好思考一下。毕竟磨刀不误砍柴功。

其他逻辑小知识

在 python 中，有些关于逻辑真假上的小知识，需要简单了解下。

- 数 0、空对象或者其他特殊对象 None 值都认为是假
- 其他非零的数字或非空的对象都认为是真
- 前面两条用 bool 函数可以进行强制类型转换
- 比较和相等测试会递归作用在数据结构中
- 比较和相等测试会返回 True 或 False（1 和 0 的 custom version（翻译为定制版？））

try 语句捕捉错误

try 语句是编程中用来处理可能出现的错误或者已经出现但并不打算应付的错误最通用的方式。比如一个变量你预先想的是接受一个数值，但是用户却输入了一个字符，这个时候你就可以将这段语句包围在 try 里面；或者有时你在编程的时候就发现了这种情况，只是懒得理会他们，那么简单的把这块出错的语句包围在 try 里面，然后后面跟个 except 语句，打印出一个信息“出错了”，即可。用法如下所示：

tcbcode-2.5

```
while True:
    x=input(' 请输入一个数，将返回它除以 2 之后的数值\n输入"quit" 退出\n')
    if x=='quit':
        break
    try :
        num=float(x)
        print(num/2)
    except:
        print(' 出错了')
```

in 语句

```
>>> 'a' in ['a',1,2]
True
>>> dict
{'a': 1, 'c': 2, 'b': 3, 'd': 4}
>>> 'e' in dict
False
>>> '2' in dict
False
```

从上面例子可以看到，一般的列表判断元素是否存在和我们之前预料的一致，关于字典需要说的就是 in 语句只判断键，不判断值。

迭代

一般有内部重复操作的程序可以先考虑 for 迭代结构实现，实在不行才考虑 while 循环结构，毕竟简单更美更安全。

python 的 for 迭代语句有点类似 lisp 语言的 dolist 和 dotimes 函数，具体例子如下：

code:2-2

```
for x in 'abc':
    print(x)
```

```
a
b
c
```

in 后面跟的是**序列**类型，也就是字符串，列表，数组都是可以的。这个语句可以看作先执行 x='a'，然后执行缩进的区块，后面依次类推。

如果缩进区块中有对原列表进行的操作，为了避免逻辑混乱，之前 in 的列表应该是之前的列表，也就是你需要制造一个之前的列表的复制品，用`list[:]`这样的形式就能达到这个目的。

range 函数

range 函数是为 for 迭代语句准备的，有点类似于 lisp 的 dotimes 函数，但是功能更全更接近 common-lisp 的 loop 宏了。

```
range(1,10,2)
```

range 函数的用法如上，表示从 1 开始到 10，步长为 2，如果用 list 函数将其包裹，将会输出 [1,3,5,7,9]。如果不考虑步长的话，这个 range 函数就有点类似于在序列连着来那一小节3.3.2谈论的区间的情况。所以 range(10) 就可以看作 [0,10)，range(1,10) 就可以看作 [1,10)。但是在这里再加上步长的概念和区间的概念又有所不同了，range 函数产生的是一个什么迭代器对象，目前我只知道这个对象和之前谈论的序列对象是不同的。

code:2-3

```
for x in range(-10,-20,-3):
    print(x)
```

```
-10
-13
-16
-19
```

上面例子还演示了 range 的负数概念，这里如果用区间概念来考察的话，是不能理解的，之所以行得通，是因为它的步长是负数，如果不是负数，那么情况就会和之前讨论的结果类似，将是一个空值。

迭代加上操作

迭代产生信息流并经过某些操作之后生成目标序列：

```
squares=[x**2 for x in [1,2,3,4,5]]
squares
```

循环

while 语句用法和大多数编程语言类似，就是条件控制，循环结构。

```
while test:
    do something
```

break 命令

break 跳出最近的 while 或者 for 循环结构。

continue 命令

continue 命令接下来的循环结构的执行区块将不执行了，跳到条件判断那里看看是不是继续循环。如果是，那么继续循环。

循环结构中的 else 命令

循环结构（for 语句和 while 语句）最后还可以有一个 else 字句，表示循环执行完了（for 是执行完了，while 是条件为假时也相当于执行完了。）将执行这个语句。

如果 break 跳出语句，那么最后的 else 字句将不执行，说明 else 字句是平行于该循环结构的语句。

pass 命令

pass 命令就是什么都不做。pass 命令即可用于循环语句也可用于条件语句，其中有些条件语句有的时候是需要什么都不做。

程序中的操作对象

python 和 c 语言不同，c 是什么`int x = 3`，也就是这个变量是整数啊，字符啊什么的都要明确指定，python 不需要这样做，只需要声明`x = 3` 即可。但是我们知道任何程序语言它到最后必然要明确某一个变量（这里也包括后面的更加复杂的各个结构对象）的内存分配，只是 python 语言帮我们将这些工作做了，所以就让我们省下这份心吧。

```
''' 这是一个多行注释
    你可以在这里写上很多废话
    '''
x = 10
print(x,type(x))
```

python 程序由各个模块（modules）组成，模块就是各个文件。模块由声明（statements）组成，声明由表达式（expressions）组成，表达式负责创造和操作对象（objects）。在 python 中一切皆对象。python 语言内置对象（数值、字符串、列表、数组、字典、文件、集合、其他内置对象。）后面会详细说明之。

赋值

python 中的赋值语法非常的简单，x=1，就是一个赋值语句了。和 c 语言不同，c 是必须先声明 int x 之类，开辟一个内存空间，然后才能给这个 x 赋

值。而 python 的 x=1 语句实际上至少完成了三个工作：一，判断 1 的类型（动态类型语言必须要这步）；二，把这个类型的对象存储在内存里面；三，创建 x 这个名字和这个名字指向这个内存，x 似乎可以称之为对应 c 语言的指针对象。

序列赋值

```
x,y=1,'a'
[z,w]=['b',10]
print(x,y,z,w)
```

我们记得 python 中表达式可以加上圆括号，所以这里x,y 产生的是一个数组(x,y)，然后是对应的数组平行赋值，第二行是列表的平行赋值。这是一个很有用的技巧。

同时赋相同的值

```
x=y='a'
z=w=2
print(x,y,z,w)
```

这种语句形式 c 语言里面也有，不过内部实现机制就非常的不一样了。python 当声明 x=y 的时候，x 和 y 是相同的指针值，然后相同的指针值都指向了'a' 这个字符串对象，也可以说 x 和 y 就是一个东西，只是取的名字不同罢了。

但如果写成这种形式：

```
x=1
y=1
```

那么 x 和 y 还是指向的同一个对象，甚至是同一内存区块吗？如果是的话，那我对 python 内部如何处理的实现了这样的效果很感兴趣了。

增强赋值语句

x=x+y 可以写作 x += y。类似的还有：

+=	&=	»=
-=	|=	«=
*=	^=	**=
/=	%=	//=

数值

python 的数值的内置类型有：int，float，complex 等[①]。python 的基本算术运算操作有加减乘除（+ - * /）。然后‘=’表示赋值，类似数学书上的中缀表达式和优先级和括号法则等，这些都是一般编程语言说到烂的东西了。

```
print((1+2)*(10-5)/2)
print(2**100)
```

二进制八进制十六进制

二进制的数字以 0b（零比）开头，八进制的数字以 0o（零哦）开头，十六进制的数字以 0x（零艾克斯）开头。

```
0b101010, 0o177, 0x9ff
```

① 这些 int、float 等命令都是强制类型转换命令

以二进制格式查看数字使用 bin 命令，以十六进制查看数字使用 hex 命令。

```
>>> bin(42)
'0b101010'
>>> hex(42)
'0x2a'
```

数学幂方运算

x^y，x 的 y 次方如上面第二行所述就是用x**y 这样的形式即可。此外 pow 函数作用是一样的，pow(x,y)。

数值比较

数值比较除了之前提及的 >，<，== 之外，>=，<=，!= 也是有的（大于等于，小于等于，不等于）。此外 python 还支持连续比较，就是数学格式 $a < x < b$，x 在区间 (a, b) 的判断。在 python 中可以直接写成如下形式：a<x<b。这实际实现的过程就是两个比较操作的进一步与操作。

相除取整

就作为正整数相除使用x//y 得到的值意义还是很明显的就是商。带上负号感觉有点怪了，这里先略过。相关的还有取余数，就是x%y，这样就得到 x 除以 y 之后的余数了，同样带上负号情况有变，这里先略过。

复数

python 直接支持复数，复数的写法是类似1+2j 这样的形式，然后如果 z 被赋值了一个复数，这样它就是一个复数类型，那么这个类具有两个属性量，**real**

和 **imag**。也就是使用`z.real` 就给出这个复数的实数部。imag 是 imaginary number 的缩写，虚数，想像出来的数。

abs 函数

大家都知道 abs 函数是绝对值函数，这个 python 自带的，不需要加载什么宏包。作用于复数也是可以的：

```
z=3+4j
print(z.real,z.imag)
print(abs(z))
```

这个和数学中复数绝对值的定义完全一致，也就是复数的模：
$|z| = \sqrt{a^2 + b^2}$

round 函数

简单的理解就是这个函数实现了对数值的四舍五入功能，第二个参数默认是 0，即保留零位小数的意思。

```
>>> round(3.1415926,0)
3.0
>>> round(3.1415926,1)
3.1
>>> round(3.1415926,2)
3.14
>>> round(3.1415926,3)
3.142
>>> round(3.1415926,4)
```

```
3.1416
>>> round(3.1415926,5)
3.14159
```

min，max 和 sum 函数

min，max 函数的用法和 sum 的用法稍微有点差异，简单起见可以认为 min，max，sum 都接受一个元组或者列表（还有其他？），然后返回这个元组或者列表其中的最小值，最大值或者相加总和。此外 min 和 max 还支持 min(1,2,3) 这样的形式，而 sum 不支持。

```
>>> min((1,6,8,3,4))
1
>>> max([1,6,8,3,4])
8
>>> sum([1,6,8,3,4])
22
>>> min(1,6,8,3,4)
1
```

位操作

python 支持位操作的，这里简单说一下：位左移操作 «，位与操作 &，位或操作 |，位异或操作 ^。

```
>>> x=0b0001
>>> bin(x << 2)
'0b100'
>>> bin(x | 0b010)
```

```
'0b11'
>>> bin(x & 0b1)
'0b1'
>>> bin(x ^ 0b101)
'0b100'
```

math 宏包

在`from math import *`之后，可以直接用符号 pi 和 e 来引用圆周率和自然常数。此外 math 宏包还提供了很多数学函数，比如：

sqrt 开平方根函数，sqrt(x)。

sin 正弦函数，类似的还有 cos，tan 等，sin(x)。

degrees 将弧度转化为角度，三角函数默认输入的是弧度值。

radians 将角度转化位弧度，radians(30)。

log 开对数，log(x,y)，即 $\log_y x$，y 默认是 e。

exp 指数函数，exp(x)。

pow 扩展了内置方法，现在支持 float 了。pow(x,y)

这里简单写个例子：

```
>>> from math import *
>>> print(pi)
3.141592653589793
>>> print(sqrt(85))
9.219544457292887
>>> print(round(sin(radians(30)),1))#sin(30°)
0.5
```

更多具体细节请参看官方文档。

random 宏包

random 宏包提供了一些函数来解决随机数问题。

random random 函数产生 0 到 1 之间的随机实数（包括 0）

uniform uniform 函数产生从 a 到 b 之间的随机实数（a，b 的值指定，包括 a。）

randrange randrange 函数产生从 a 到 b 之间的随机整数，步长为 c（a，b，c 的值指定，相当于 choice(range(start,stop,step))，range(0,5) 是从 0 到 4 不包括 5，所以这里是从 a 到 b 不包括 b。）

choice choice 随机从一个列表或者字符串中取出一个元素。

下面是一个简单的例子:

code:3-1

```
from random import *
print(random())
print(uniform(1,10))
print(randrange(1,6))
print(choice('abcdefghij'))
print(choice(['①','②','③']))
```

```
0.1688146260256782
4.810220165697622
4
b
```

①

更多详细细节请参阅官方帮助文档。

numpy 宏包

关于 numpy 宏包的介绍请看文件夹 [玩 *ipython notebook*]。

序列

字符串，列表，元组（tuple，这里最好翻译成元组，因为里面的内容不一定是数值。）都是序列（sequence）的子类，所以序列的一些性质他们都具有，最好在这里一起讲方便理解记忆。

len 函数

len 函数返回序列所含元素的个数：

code:3-2

```
string001='string'
list001=['a','b','c']
tuple001=(1,2,3,4)

for x in [string001,list001,tuple001]:
    print(len(x))
```

```
6
3
4
```

调出某个值

对于序列来说后面跟个方括号，然后加上序号（程序界的老规矩，从 0 开始计数。），那么调出对应位置的那个值。还以上面那个例子来说明。

code:3-3

```
string001='string'
list001=['a','b','c']
tuple001=(1,2,3,4)

for x in [string001,list001,tuple001]:
    print(x[2])
```

```
r
c
3
```

倒着来

倒着来计数 -1 表示倒数第一个，-2 表示倒数第二个。依次类推。

code:3-4

```
string001='string'
list001=['a','b','c']
tuple001=(1,2,3,4)

for x in [string001,list001,tuple001]:
    print(x[-1],x[-2])
```

```
g n

c b

4 3
```

连着来

前面不写表示从头开始，后面不写表示到达尾部。中间加个冒号的形式表示从那里到那里。这里**注意**后面那个元素是不包括进来，看来 python 区间的默认含义都是包头不包尾。这样如果你想要最后一个元素也进去，只有使用默认的不写形式了。

code:3-5

```
string001='string'
list001=['a','b','c']
tuple001=(1,2,3,4)

for x in [string001,list001,tuple001]:
    print(x[1:3],x[-2:-1],x[:-1],x[1:],x[1:-1])
```

```
tr n strin tring trin

['b', 'c'] ['b'] ['a', 'b'] ['b', 'c'] ['b']

(2, 3) (3,) (1, 2, 3) (2, 3, 4) (2, 3)
```

用数学半开半闭区间的定义来理解这里的包含关系还是很便捷的。

1. 首先是数学半开半闭区间，左元素和右元素都是之前叙述的对应的定位点。左元素包含右元素不包含。
2. 其次方向应该是从左到右，如果定义的区间是从右到左，那么将产生空值。

3. 如果区间超过，那么从左到右包含的所有元素就是结果。

4. 最后如果左右元素定位点相同，那么将产生空值，比如：`string001[2:-4]`，其中 2 和 -4 实际上是定位在同一个元素之上的。额外值得一提的列表插入操作，请参看列表的插入操作这一小节。3.5.1

序列的可更改性

字符串不可以直接更改，但可以组合成为新的字符串；列表可以直接更改；元组不可以直接更改。

序列的加法和减法

两个字符串相加就是字符串拼接了。乘法就是加法的重复，所以一个字符串乘以一个数字就是自己和自己拼接了几次。列表还有元组和字符串一样大致情况类似。

code:3-6

```
print('abc'+'def')
print('abc'*3)
print([1,2,3]+[4,5,6])
print((0,'a')*2)
```

```
abcdef
abcabcabc
[1, 2, 3, 4, 5, 6]
(0, 'a', 0, 'a')
```

字符串

python 语言不像 c 语言字符和字符串是不分的，用单引号或者双引号包起来就表示一个字符串了。单引号和双引号的区别是一般用单引号，如果字符串里面有单引号，那么就使用双引号，这样单引号直接作为字符处理而不需要而外的转义处理——所谓转义处理和其他很多编程语言一样用 \ 符号。比如要显示' 就输入\'。

三单引号和三双引号

在单引号或者双引号的情况下，你可以使用\n 来换行，其中 \n 表示换行。此外还可以使用三单引号''' 或者三双引号""" 来包围横跨多行的字符串，其中换行的意义就是换行，不需要似前面那样的处理。

```
print('''\
这是一段测试文字
  this is a test line
      其中空白和      换行都所见所得式的保留。''')
```

find 方法

字符串的 find 方法可用来查找某个子字符串，没有找到返回 -1，找到了返回字符串的偏移量。用法就是：`s.find('d')`。

replace 方法

字符串的 replace 方法进行替换操作，接受两个参数：第一个参数是待匹配的子字符串，第二个参数是要替换成为的样子。

upper 方法

将字符串转换成大写形式。

isdigit 方法

类似的还有 isalpha 方法，测试是不是数字或字母。值得注意的是就算是字母组成的语句，中间有空间也会返回 False。

split 方法

字符串的 split 方法可以将字符串比如有空格或者逗号等分隔符分割而成，可以将其分割成子字符串列表。默认是空格是分隔符。

join 方法

字符串的 join 方法非常有用，严格来说它接受一个迭代器参数，不过最常见的是列表，如下所示：

```
>>> list001=['a','b','c']
>>> "".join(list001)
'abc'
```

rstrip 方法

字符串右边的空格都删除。换行符也会被删除掉。

format 方法

字符串的 format 方法方便对字符串内的一些变量进行替换操作，其中花括号不带数字跟 format 方法里面所有的替换量，带数字 0 表示第一个替换量，后面类推。

```
print('1+1={0}, 2+2={1}'.format(1+1,2+2))
```

转义和不转义

\n　　\t　这是一般常用的转义字符，换行和制表。此外还有\\ 输出 \ 符号。

如果输出字符串不想转义那么使用如下格式:

```
>>> print(r'\t \n \test')
\t \n \test
```

列表

方括号包含几个元素就是列表。

列表的插入操作

字符串和数组都不可以直接更改所以不存在这个问题，列表可以。其中列表还可以以一种定位在相同元素的区间的方法来实现插入操作，这个和之前理解的区间多少有点违和，不过考虑到定位在相同元素的区间本来就概念模糊，所以在这里就看作特例，视作在这个定位点相同元素之前插入吧。

code:3-7

```
list001=['one','two','three']
list001[1:-2]=['four','five']
print(list001)
```

```
['one', 'four', 'five', 'two', 'three']
```

除了序列中的一些继承的操作之外，列表还有很多方法，实际上这还算少的（如果你见识了 lisp 中各种列表操作）。因为列表这个数据结构可以直接修改相当灵活，下面我打算将我学 lisp 语言中接触到的一些列表操作对应过来一一说明之：

extend 方法似乎和列表之间的加法重合了，比如 list001.extend([4,5,6]) 就和 list001=list001+[4,5,6] 是一致的，而且用加法表示还可以自由选择是不是覆盖原定义，这实际上更加自由。所以 extend 方法略过。

insert 方法也就是列表的插入操作，这个前面关于列表的插入实现方法说过一种了，所以 insert 方法也略过。

append 方法

python 的 append 方法和 lisp 中的 append 还是有点差异的，python 的 append 就是在最后面加**一个元素**，如果你 append 一个列表那么这一个列表整体作为一个元素。lisp 的 append 函数和 python 的 extend 方法类似，接受一个列表。

其次 append 是 list 类中的一个方法，也就是 list001.append 这样的形式，也就是永久的改变了某个列表实例的值了。

reverse 方法

reverse 方法不接受任何参数，直接将一个列表永久性地翻转过来。

sort 方法

也就是排序，永久性改变列表。默认是递增排序，可以用 **reverse=True** 来调成递减排序。可以用 **key=function** 来设置排序的函数，这个排序函数是单参数函数。

类似的有 sorted 函数，不同的 sorted 函数返回的是一个新的列表而不是原处修改。

删除某个元素

- 赋空列表值，相当于所有元素都删除了。
- pop 方法：接受一个参数，就是列表元素的定位值，然后那个元素就删除了，方法并返回那个元素的值。如果不接受参数默认是删除最后一个元素。
- remove 方法：移除第一个相同的元素，如果没有返回相同的元素，返回错误。
- del 函数：删除列表中的某个元素。

```
>>> list001=['a','b','c','d','e']
>>> list001.pop(2)
'c'
>>> list001
['a', 'b', 'd', 'e']
>>> list001.pop()
```

```
'e'
>>> list001
['a', 'b', 'd']
>>> list001.remove('a')
>>> list001
['b', 'd']
>>> del list001[1]
>>> list001
['b']
```

index 方法

index 方法返回某个相同元素的偏移值。

```
>>> list001=[1,'a',100]
>>> list001.index('a')
1
```

列表元素的替换

lisp 语言中有 subst 函数，是 substitute 的缩写。作用于整个列表，列表中所有出现的某个元素都要被另一个元素替换掉。

由于我现在对如何修改 python 语言内置类还毫无头绪，只好简单写这么一个函数了。

code:3-8

```
def subst(list001,element001,element002):
    try:
        list001.index(element001)
```

```
    except ValueError:
        return list001
    else:
        n=list001.index(element001)
        del list001[n]
        list001[n:n]=[element002]
        return subst(list001,element001,element002)

print(subst([1,'a',3,[4,5]],[4,5],'b'))
print(subst([1,1,5,4,1,6],1,'replaced'))
```

```
[1, 'a', 3, 'b']
['replaced', 'replaced', 5, 4, 'replaced', 6]
```

这个 subst 函数接受三个参数，表示接受的列表，要替换的元素和替换成为的元素。这里使用的程序结构是 try...except...else... 语句。其中 try 来侦测是不是有错误，其中 index 方法是看那个要替换的元素存不存在，由于不存在这个函数将产生一个 **ValueError** 错误，所以用 except 来接著。既然没有要替换的元素了，那么返回原列表即可，程序中止。

else 语句接著没有错误的时候你要执行的操作，先 index 再删掉这个元素，再在之前插入那个元素，然后使用了递归算法，调用函数自身。

列表解析

lisp 中的 mapcar 函数有这个功用，python 中的 map 函数基本上和它情况类似。

我们先来看 lisp 中的情况：

code:3-9

```
(defun square (n) (* n n))
(format t "~&~s" (mapcar #'square '(1 2 3 4 5)) )
```

```
(1 4 9 16 25)
```

再来看 python 中的情况：

code:3-10

```
def square(n):
    return n*n

print(list(map(square,[1,2,3,4,5])))
print([square(x) for x in [1,2,3,4,5]])
```

```
[1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
```

map 函数将某个应数应用于某个列表的元素中并生成一个 map 对象，需要外面加上 list 函数才能生成列表形式。第二种方式更有 python 风格，是推荐使用的列表解析方法。

count 方法

统计某个元素出现的次数。

```
>>> list001=[1,'a',100,1,1,1]
>>> list001.count(1)
4
```

for 语句的进阶

在 lisp 语言的 loop 宏中，还有很多高级应用，比如

collect 将迭代产生的所有信息收集到列表中。

summing 将迭代产生的所有信息加到一起。

count 跟着一个判断函数，每次迭代运行一次，然后记录得到的 True 即真值的情况的总数。

minimize 将每次迭代的结果进行比较，然后返回最小值。

maximize 同 minimize，返回最大值。

append 将每次迭代产生的列表 append 在一起。

那么在 python 中如何实现以上功能呢？

在这里最基本的是通过迭代语句产生一个列表，然后通过某些函数比如 minimize 对应 min 函数，maximize 对应 max 函数等对这个列表进行一些操作即可。

code:3-11

```
from random import *
def random_list_max(n):
    y=[randint(1,n) for x in range(1000)]
    list_count=[y.count(x) for x in range(1,n+1)]
    return list_count.index(max(list_count))+1

print(random_list_max(40))
```

```
30
```

字典

与列表一样字典是可变的，可以像列表一样引用然后原处修改，del 语句也适用。

创建字典

字典是一种映射，并没有从左到右的顺序，只是简单地将键映射到值。字典的声明格式如下：

```
dict001={'name':'tom','height':'180','color':'red'}
dict001['name']
```

或者创建一个空字典，然后一边赋值一边创建对应的键：

```
dict002={}
dict002['name']='bob'
dict002['height']=195
```

所以对字典内不存在的键赋值是可行的。

字典里面有字典

和列表的不同就在于字典的索引方式是根据“键”来的。

```
dict003={'name':{'first':'bob','second':'smith'}}
dict003['name']['first']
```

字典遍历操作

字典特定顺序的遍历操作的通用做法就是通过字典的 keys 方法收集键的列表，然后用列表的 sort 方法处理之后用 for 语句遍历，如下所示：

```
dict={'a':1,'c':2,'b':3}
dictkeys=list(dict.keys())
dictkeys.sort()
for key in dictkeys:
    print(key,'->',dict[key])
```

如果你对字典遍历的顺序没有要求，那么就可以简单的这样处理：

```
>>> for key in dict:
...     print(key,'->',dict[key])
...
c -> 2
a -> 1
b -> 3
```

sorted 函数

sorted 方法可以对字典直接排序，返回的是该字典键值的列表。

```
dict={'a':1,'c':2,'b':3}
for key in sorted(dict):
    print(key,'->',dict[key])
```

values 方法

和 keys 方法类似，收集的值。

```
>>> dict001.values()
dict_values([3, 1, 2])
>>> list(dict001.values())
[3, 1, 2]
```

items 方法

和 keys 和 values 方法类似，不同的是返回的是 (key,value) 对。

```
>>> dict001.items()
dict_items([('c', 3), ('a', 1), ('b', 2)])
>>> list(dict001.items())
[('c', 3), ('a', 1), ('b', 2)]
```

字典的 in 语句

可以看到 in 语句只针对字典的键，不针对字典的值。

```
>>> dict001={'a':1,'b':2,'c':3}
>>> 2 in dict001
False
>>> 'b' in dict001
True
```

字典对象的 get 方法

get 方法是去找某个键的值，为什么不直接引用呢，get 方法的好处就是某个键不存在也不会出错。

```
>>> dict001={'a':1,'b':2,'c':3}
>>> dict001.get('b')
2
>>> dict001.get('e')
```

update 方法

感觉字典就是一个小型数据库，update 方法将另外一个字典里面的键和值覆盖进之前的字典中去，称之为更新，没有的加上，有的覆盖。

```
>>> dict001={'a':1,'b':2,'c':3}
>>> dict002={'e':4,'a':5}
>>> dict001.update(dict002)
>>> dict001
{'c': 3, 'a': 5, 'e': 4, 'b': 2}
```

pop 方法

pop 方法类似列表的 pop 方法，不同引用的是键，而不是偏移地址，这个就不多说了。

字典解析

这种字典解析方式还是很好理解的。

```
>>> dict001={x:x**2 for x in [1,2,3,4]}
>>> dict001
{1: 1, 2: 4, 3: 9, 4: 16}
```

zip 函数创建字典

可以利用 zip 函数来通过两个列表平成一个字典，这个有时很有用。zip 函数刚开始返回的是什么 zip 对象，然后对其进行字典解析。

```
>>> dict001=zip(['a','b','c'],[1,2,3])
>>> dict001
<zip object at 0xb6b464ac>
>>> dict001={a:b for (a,b) in dict001}
>>> dict001
{'c': 3, 'a': 1, 'b': 2}
```

这种解析方式的理解有一定难度，主要是 zip 函数这里比较抽象了。而下面这段代码是python2的，就比较好理解一点。所以要理解 python3 的那种语法，只好认为 python3 中的 zip 对象只是还未展开的如下所示结构的那种列表。值得一提的在 python3 中dict 强制变换不能使用了。

```
>>> zip(['a','b','c'],[1,2,3])
[('a', 1), ('b', 2), ('c', 3)]
>>> dict001=dict(zip(['a','b','c'],[1,2,3]))
>>> dict001
{'a': 1, 'c': 3, 'b': 2}
```

字典的集合操作

python3 的字典更多的接近集合了，它现在支持很多集合操作，比如 -，&，| 等。字典的集合操作是以键参与集合操作的。

集合

集合可以用过可以通过 set 函数创建一个集合对象，集合的元素是无序的，数学上类似的集合操作这里都有：

```
>>> set001=set('hello')
>>> set001
set(['h', 'e', 'l', 'o'])
>>> set002={'h','a','b'}
>>> set002
set(['a', 'h', 'b'])
>>> print(set001)
set(['h', 'e', 'l', 'o'])
>>> set001 & set002
set(['h'])
>>> set001 | set002
set(['a', 'b', 'e', 'h', 'l', 'o'])
>>> set001 - set002
set(['e', 'l', 'o'])
>>> {x**2 for x in [1,2,3,4,5]}
set([16, 1, 4, 25, 9])
```

其中&　|　-就是交，与，差的意思。

元组

圆括号包含几个元素就是元组 (tuple)。元组和列表的不同在于元组是不可改变。元组也是从属于序列对象的，元组的很多方法之前都讲了。而且元组在使用上和列表极其接近，有很多内容这里也略过了。

值得一提的是如果输入的时候写的是 *x,y* 这样的形式，实际上表达式就加上括号了，也就是一个元组了 *(x,y)*。

文件

写文件

对文件的操作首先需要用 open 函数创建一个文件对象，简单的理解就是把相应的接口搭接好。文件对象的 write 方法进行对某个文件的写操作，最后需要调用 close 方法写的内容才真的写进去了。

```
file001 = open('test.txt','w')
file001.write('hello world1\n')
file001.write('hello world2\n')
file001.close()
```

如果你们了解 C 语言的文件操作，在这里会为 python 语言的简单便捷赞叹不已。就是这样三句话：创建一个文件对象，然后调用这个文件对象的 wirte 方法写入一些内容，然后用 close 方法关闭这个文件即可。

读文件

一般的用法就是用 open 函数创建一个文件对象，然后用 read 方法调用文件的内容。最后记得用 close 关闭文件。

```
file001 = open('test.txt')
filetext=file001.read()
print(filetext)
file001.close()
```

此外还有 readline 方法是一行一行的读取某文件的内容。

open 函数的处理模式

open 函数的处理模式如下:

'r' 默认值，read，读文件。

'w' wirte，写文件，如果文件不存在会创建文件，如果文件已存在，文件原内容会清空。

'a' append，附加内容，也就是后面用 write 方法内容会附加在原文件之后。

'b' 处理模式设置的附加选项，'b' 不能单独存在，要和上面三个基本模式进行组合，比如'rb' 等，意思是二进制数据格式读。

'+' 处理模式设置的附加选项，同样'+' 不能单独存在，要和上面三个基本模式进行组合，比如'r+' 等，+ 是 updating 更新的意思，也就是既可以读也可以写，那么'r+'，'w+'，'a+' 还有什么区别呢？区别就是'r+' 不具有文件创建功能，如果文件不存在会报错，然后'r+' 不会清空文件，如果'r+' 不清空文件用 write 方法情况会有点复杂；而'w+' 具有文件创建功能，然后'w+' 的 write 方法内容都是重新开始的；而'a+' 的 write 方法内容是附加在原文件上的，然后'a+' 也有文件创建功能。

用 with 语句打开文件

类似之前的例子我们可以用 with 语句来打开文件，这样就不用 close 方法来关闭文件了。然后 with 语句来提供了类似 try 语句的功能可以自动应对打开文件时的一些异常情况。

```
with open('test.txt','w') as file001:
    file001.write('hello world1\n')
    file001.write('hello world2\n')

with open('test.txt','r') as file001:
    filetext=file001.read()
    print(filetext)
```

除字符串外其他类型的读取

文本里面存放的都是字符串类型，也就是写入文件需要用 str 函数强行将其他类型转变成字符串类型，而读取进来想要进行一些操作则需要将字符串类型转变回去。比如用 int 或者 float 等，不过列表和字典的转变则需要 eval 函数。

eval 这个函数严格来讲作用倒不是为了进行上面说的类型转换的，它就是一个内置函数，一个字符串类型 python 代码用 eval 函数处理了之后就能转变为可执行代码。

```
>>> eval('1+1')
2
>>> eval('[1,2,3]')
[1, 2, 3]
>>> eval("{'a':1,'b':2,'c':3}")
{'c': 3, 'b': 2, 'a': 1}
```

推荐使用 pickle 宏包来处理其他类型的文件读写问题，相对来说更简单更安全。请参看 pickle 宏包这一小节10。

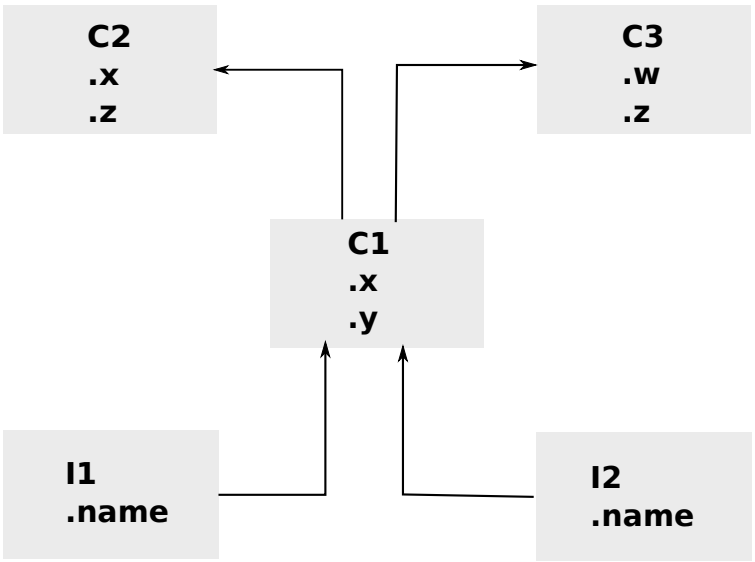
总结

类

类相当于自己创造一个自己的操作对象。一般面向对象 (OOP) 编程的基本概念这里不重复说明了，如有不明请读者自己随便搜索一篇网页阅读下即可。

python 中类的结构

python 中的类就好像树叶，所有的类就构成了一棵树，而 python 中超类，子类，实例的重载或继承关系等就是由一种搜索机制实现的：



python 首先搜索 self 有没有这个属性或者方法，如果没有，就向上搜索。比如说实例 I1 没有，就向上搜索 C1，C1 没有就向上搜索 C2 或 C3 等。

实例继承了创造他的类的属性，创造他的类上面可能还有更上层的超类，类似的概念还有子类，表示这个类在树形层次中比较低。

well，简单来说类的结构和搜索机制就是这样的，很好地模拟了真实世界知识的树形层次结构。

上面那副图实际编写的代码如下：

```
class C2: ...
class C3: ...
class C1(C2,C3): ...
l1=C1()
l2=C1()
```

其中 class 语句是创造类，而 C1 继承自 C2 和 C3，这是多重继承，从左到右是内部的搜索顺序（会影响重载）。l1 和 l2 是根据类 C1 创造的两个实例。

对于初次接触类这个概念的读者并不指望他们马上就弄懂类这个概念，这个概念倒并一定要涉及很多哲学的纯思考的东西，也可以看作一种编程经验或技术的总结。多接触也许对类的学习更重要，而不是纯哲学抽象概念的讨论，毕竟类这个东西创造出来就是为了更好地描述现实世界的。

最后别人编写的很多宏包就是一堆类，你就是要根据这些类来根据自己的情况情况编写自己的子类，为了更好地利用前人的成果，或者你的成果更好地让别人快速使用和上手，那么你需要好好掌握类这个工具。

类的最基础知识

类的创建

```
class MyClass:
    something
```

类的创建语法如上所示，然后你需要想一个好一点的类名。类名规范的写法是首字母大写，这样好和其他变量有所区分。

根据类创建实例

按照如下语句格式就根据 MyClass 类创建了一个实例 myclass001。

```
myclass001=MyClass()
```

类的属性

```
>>> class MyClass:
...  name='myclass'
...
>>> myclass001=MyClass()
>>> myclass001.name
'myclass'
>>> MyClass.name
'myclass'
>>> myclass001.name='myclass001'
>>> myclass001.name
'myclass001'
>>> MyClass.name
'myclass'
```

如上代码所示，我们首先创建了一个类，这个类加上了一个 name 属性，然后创建了一个实例 myclass001，然后这个实例和这个类都有了 name 属性。然后我们通过实例加上点加上 name 的这种格式引用了这个实例的 name 属性，并将其值做了修改。

这个例子简单演示了类的创建，属性添加，实例创建，多态等核心概念。后面类的继承等概念都和这些大同小异了。

类的方法

类的方法就是类似上面类的属性一样加上 def 语句来定义一个函数，只是函数在类里面我们一般称之为方法。这里演示一个例子，读者看一下就明白了。

```
>>> class MyClass:
...  name='myclass'
...  def double(self):
...   self.name=self.name*2
...   print(self.name)
...
>>> myclass001=MyClass()
>>> myclass001.name
'myclass'
>>> myclass001.double()
myclassmyclass
>>> myclass001.name
'myclassmyclass'
```

这里需要说明的是在类的定义结构里面，self 代表着类自身，self.name 代表着对自身 name 属性的引用。然后实例在调用自身的这个方法时用的是 myclass001.double() 这样的结构，这里 double 函数实际上接受的第一个参数就是自身，也就是 myclass001，而不是无参数函数。所以类里面的方法（被外部引用的话）至少有一个参数 self。

类的继承

实例虽然说是根据类创建出来的，但实际上实例和类也是一种继承关系，实例继承自类，而类和类的继承关系也与之类似，只是语法稍有不同。下面我们来看这个例子：

code:4-1

```
class Hero():
    def addlevel(self):
        self.level=self.level+1
        self.hp=self.hp+self.addhp

class Garen(Hero):
    level=1
    hp=455
    addhp=96

garen001=Garen()
for i in range(6):
    print('级别:',garen001.level,'生命值: ' ,garen001.hp)
    garen001.addlevel()
```

```
级别: 1 生命值: 455
级别: 2 生命值: 551
级别: 3 生命值: 647
级别: 4 生命值: 743
级别: 5 生命值: 839
级别: 6 生命值: 935
```

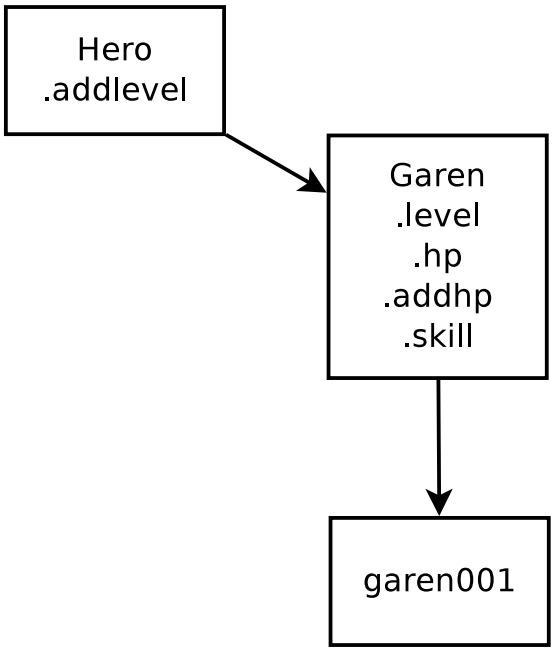


图 4-1: 类的继承示例

这里就简单的两个类，盖伦 Garen 类是继承自 Hero 类的，实例 garen001 是继承自 Garen 类的，这样 garen001 也有了 addlevel 方法，就是将自己的 level 属性加一，同时 hp 生命值也加上一定的值，整个过程还是很直观的。

类的内置方法

如果构建一个类，就使用 pass 语句，什么都不做，python 还是会为这个类自动创建一些属性或者方法。

```
>>> class TestClass:
...  pass
...
>>> dir(TestClass)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__',
```

```
 '__gt__',  '__hash__', '__init__', '__le__', '__lt__',
  '__module__', '__ne__', '__new__', '__reduce__',
  '__reduce_ex__', '__repr__', '__setattr__',
  '__sizeof__', '__str__', '__subclasshook__',
   '__weakref__']
```

这些变量名字前后都加上双下划线是给 python 这个语言的设计者用的，一般应用程序开发者还是不要这么做。

这些内置方法用户同样也是可以重定义他们从来覆盖掉原来的定义，其中特别值得一讲的就是__init__ 方法或者称之为构造函数。

__init__ 方法

__init__ 方法对应的就是该类创建实例的时候的构造函数。比如：

```
>>> class Point:
...  def __init__(self,x,y):
...   self.x=x
...   self.y=y
...
>>> point001=Point(5,4)
>>> point001.x
5
>>> point001.y
4
```

这个例子重载了__init__ 函数，然后让他接受三个参数，self 等下要创建的实例，x，还有 y 通过下面的语句给这个待创建的实例的属性 x 和 y 赋了值。

self 意味着什么

self 在类中是一个很重要的概念，当类的结构层次较简单时还容易看出来，当类的层次结构很复杂之后，你可能会弄糊涂。self 就是指现在引用的这个实例。比如你现在通过调用某个实例的某个方法，这个方法可能是一个远在天边的某个类给出的定义，就算如此，那个定义里面的 self 还是指调用这个方法的那个实例，这一点要牢记于心。

类的操作第二版

现在我们可以写出和之前那个版本相比更加专业的类的使用版本了。

code:4-2

```
class Hero():
    def addlevel(self):
        self.level=self.level+1
        self.hp=self.hp+self.addhp

class Garen(Hero):
    def __init__(self):
        self.level=1
        self.hp=455
        self.addhp=96
        self.skill=['不屈','致命打击','勇气','审判','德玛西亚正义']

garen001=Garen()
for i in range(6):
    print('级别:',garen001.level,'生命值: ' ,garen001.hp)
    garen001.addlevel()
print('盖伦的技能有: ',"".join([x + '  ' for x in garen001.skill]))
```

```
级别: 1 生命值: 455
级别: 2 生命值: 551
级别: 3 生命值: 647
级别: 4 生命值: 743
级别: 5 生命值: 839
级别: 6 生命值: 935
盖伦的技能有: 不屈 致命打击 勇气 审判 德玛西亚正义
```

似乎专业的做法类里面多放点方法，最好不要放属性，不太清楚是什么。但确实这样写给人感觉更干净点，方法是方法，如果没有调用代码就放在那里我们不用管它，后面用了构造函数我们就去查看相关类的构造方法，这样很省精力。

类的操作第三版

code:4-3

```
class Unit():
    def __init__(self,hp,atk,color):
        self.hp=hp
        self.atk=atk
        self.color=color
    def __str__(self):
        return '生命值: {0}, 攻击力: {1}, 颜色: \
        {2}'.format(self.hp,self.atk,self.color)

class Hero(Unit):
    def __init__(self,level,hp,atk,color):
        Unit.__init__(self,hp,atk,color)
```

```
        self.level=level
    def __str__(self):
        return '级别: {0}, 生命值: {1}, 攻击力: {2}, \
        颜色: {3}'.format(self.level,self.hp,self.atk,self.color)

    def addlevel(self):
        self.level=self.level+1
        self.hp=self.hp+self.addhp
        self.atk=self.atk+self.addatk

class Garen(Hero):
    def __init__(self,color='blue'):
        Hero.__init__(self,1,455,56,color)
        self.name='盖伦'
        self.addhp=96
        self.addatk=3.5
        self.skill=['不屈','致命打击','勇气','审判','德玛西亚正义']

if __name__ == '__main__':
    garen001=Garen('red')
    garen002=Garen()
    print(garen001)
    unit001=Unit(1000,1000,'gray')
    print(unit001)
    for i in range(6):
        print(garen001)
        garen001.addlevel()
    print('盖伦的技能有: ',"".join([x + '  ' for x in garen001.skill]))
```

级别: 1, 生命值: 455, 攻击力: 56, 颜色: red

生命值：1000，攻击力：1000，颜色：gray

级别：1, 生命值：455，攻击力：56，颜色：red

级别：2, 生命值：551，攻击力：59.5，颜色：red

级别：3, 生命值：647，攻击力：63.0，颜色：red

级别：4, 生命值：743，攻击力：66.5，颜色：red

级别：5, 生命值：839，攻击力：70.0，颜色：red

级别：6, 生命值：935，攻击力：73.5，颜色：red

盖伦的技能有：不屈 致命打击 勇气 审判 德玛西亚正义

现在就这个例子相对于第二版所作的改动，也就是核心知识点说明之。其中函数参量列表中这样表述`color='blue'` 表示 blue 是 color 变量的备选值，也就是 color 成了可选参量了。

构造函数的继承和重载

上面例子很核心的一个概念就是`__init__` 构造函数的继承和重载。比如我们看到 garen001 实例的创建，其中就引用了 Hero 的构造函数，特别强调的是只有创造实例的时候比如这样的形式 Garen() 才叫做调用了 Garen 类的构造方法，比如这里 `Hero.__init__(self,1,455,56,color)` 就是调用了 Hero 类的构造函数，这个时候需要把 self 写上，因为 self 就是最终创建的实例 garen001，而不是 Hero，而且调用 Hero 类的构造函数就必须按照它的参量列表形式来。这个概念需要弄清楚!

理解了这一点，在类的继承关系中的构造函数的继承和重载就好看了。比如这里 Hero 类的构造函数又是继承自 Unit 类的构造函数，Hero 类额外有一个参量 level 接下来也要开辟存储空间配置好。

__str__ 函数的继承和重载

第二个修改是这里重定义了一些类的__str__ 函数，通过重新定义它可以改变默认 print 某个类对象是的输出。默认只是一段什么什么类并无具体内容信息。具体就是 return 一段你想要的字符串样式即可。

类的其他内置方法

类还有其他的一些内置方法，比如__add__ 就控制这对象面对加号时候的行为。这些我们暂时先略过。

操作或者函数

自定义函数

定义函数用 def 命令，语句基本结构如下：

```
def yourfunctionname(para001,para002...):
    do something001
    do something002
```

参数和默认参数

定义的函数圆括号那里就是接受的参数，如果参数后面跟个等号，来个赋值语句，那个这个赋的值就是这个参数的默认值。比如下面随便写个演示程序：

code:5-1

```
def test(x='hello'):
    print(x)
test()
test('world')
```

```
hello
world
```

递归函式

虽然递归函式能够在某种程度上取代前面的一些循环或者迭代程序结构，不过不推荐这么做。这里谈及递归函式是把某些问题归结为数学函数问题，而这些问题常常用递归算法更加直观（不一定高效）。比如下面的菲波那奇函数：

code:5-2

```
def fib(n):
    if n==0:
        return 1
    if n==1:
        return 1
    else:
        return fib(n-1)+fib(n-2)

for x in range(5):
    print(fib(x))
```

```
1
1
2
3
5
```

我们可以看到，对于这样专门的数学问题来说，用这样的递归算法来表述是非常简洁易懂的。至于其内部细节，我们可以将上面定义的 fib 称之为函式，函

式是一种操作的模式，然后具体操作就是复制出这个函式（函数或者操作都是数据），然后按照这个函式来扩展生成具体的函数或者操作。

下面看通过递归函式来写阶乘函数，非常的简洁，我以为这就是最好最美的方法了。

code:5-3

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)

print(fact(0),fact(10))
```

```
1 3628800
```

其实通过递归函式也可以实现类似 for 的迭代结构，不过我觉得递归函式还是不应该滥用。比如下面通过递归函式生成一种执行某个操作 n 次的结构：

code:5-4

```
def dosomething(n):
    if n==0:
        pass
    elif n==1:
        print('do!')
    else:
        print('do!')
        return dosomething(n-1)

print(dosomething(5))
```

```
do!
do!
do!
do!
do!
None
```

可以看到，如果把上面的 print 语句换成其他的某个操作，比如机器人向前走一步，那么这里 dosomething 换个名字向前走 (5) 就成了向前走 5 步了。

不定参量函数

我们在前面谈到 sum 函数3.2.8只接受一个列表，而不支持这样的形式：sum(1,2,3,4,5)。现在我们设计这样一个可以接受不定任意数目参量的函数。首先让我们看看一种奇怪的赋值方式。

序列解包赋值

```
>>> a,b,*c=1,2,3,4,5,6,7,8,9
>>> print(a,b,c,sep=' | ')
1 | 2 | [3, 4, 5, 6, 7, 8, 9]
>>> a,*b,c=1,2,3,4,5,6,7,8,9
>>> print(a,b,c,sep=' | ')
1 | [2, 3, 4, 5, 6, 7, 8] | 9
>>> *a,b,c=1,2,3,4,5,6,7,8,9
>>> print(a,b,c,sep=' | ')
[1, 2, 3, 4, 5, 6, 7] | 8 | 9
```

带上一个星号 * 的变量变得有点类似通配符的味道了，针对后面的序列[①]（数组，列表，字符串），它都会将遇到的元素收集在一个列表里面，然后说是它的。

函数中的通配符

```
>>> def test(*args):
...  print(args)
...
>>> test(1,2,3,'a')
(1, 2, 3, 'a')
```

我们看到类似上面序列解包赋值中的带星号表通配的概念，在定义函数的时候写上一个带星号的参量（我们可以想象在函数传递参数的时候有一个类似的序列解包赋值过程），在函数定义里面，这个 args 就是接受到的参量组成的数组。

mysum 函数

code:5-5

```
def mysum(*args):
    return sum([arg for arg in args[:]])

print(mysum(1,2,3,4,5,6))
```

21

① 似乎序列赋值内置迭代操作

这样我们定义的可以接受任意参数的 mysum 函数，如上所示。具体过程就是将接受到的 args 进行列表解析，然后用 sum 函数处理了一下。

任意数目的可选参数

在函数定义的写上带上两个星号的变量 **args，那么 args 在函数里面的意思就是接受到的可选参数组成的一个字典值。

```
>>> def test(**args):
...  print(args)
...
>>> test(a=1,b=2)
{'b': 2, 'a': 1}
```

老实说一般参数，可选参数（关键字参数），任意参数，任意关键字参数所有这些概念混在一起非常的让人困惑了，这一块有时间再好好琢磨一下。

lambda 函式

lambda λ 表达式这个在刚开始介绍 lisp 语言的时候已有所说明，简单来说就是函数只是一个映射规则，变量名，函数名都无所谓的。这里就是没有名字的函数的意思。

具体的样子如下面所示：

code:5-6

```
f=lambda x,y,z:x+y+z
print(f(1,2,3))
```

6

lanmbda 函式在有些情况下要用到，比如 pyqt 里面的信号－槽机制用 connect 方法的时候，槽比如是函数名或者无参函数，如果用户想加入参量的话，可以使用 lamba 函式引入，具体这里我还不够清晰。

宏包

现在让我们进入宏包基础知识的学习吧，建立编写自己的宏包，这样不断积累自己的知识，不断变得更强。

实际上之前我们已经接触过很多 python 自身的标准宏包或者其他作者写的第三方宏包，而 import 和 from 语句就是加载宏包用的。这里我们主要讨论如何自己编写自己的宏包。

from 语句和 import 语句内部作用机制很类似，只是在变量名的处理方式上有点差异（from 会把变量名复制过来）。这里重点就 import 的工作方式说明如下：

1. 首先需要找到宏包文件。
2. 然后将宏包文件编译成位码（需要时，根据文件的时间戳。），你会看到新多出来一个`__pycache__` 文件夹。
3. 执行编译出来的位码，创建该 py 文件定义的对象。

这三个步骤是第一次 import 的时候会执行的，第二次 import 的时候会跳过去，而直接引用内存中已加载的对象。

找到宏包文件

python 宏包的搜索路径会搜索几个地方，这些地方最后都会放在 sys.path 这个列表里面，所以在你的 py 文件刚开始修改这个 sys.path，append 上你想

要的地址也是可以的。我在这里选择了这种简单的方法，除此之外还有很多方法这里先不涉及。

比如主文件一般如下：

```
import os,sys
sys.path.append(os.environ['HOME']+'/pymf')
from pyconfig import *
```

这里为什么使用`from pyconfig import *`这样的语句而不是 import 语句呢？因为我决定整个项目的主 py 文件除了这一个 from 语句之外不会再 import 或者 from 其他宏包了，其他所有宏包的引用都放在 pyconfig.py 这个主配置文件里面。

pyconfig.py 任务就是加载最常用最通用的一些宏包，如果你实际编写的另外一个项目通用 pyconfig 文件满足不了你的要求了，那么你可以把那个 pyconfig 文件复制过来，然后放在你的项目文件夹里面，然后继续衍化修改。这个经验是我从 LaTeX 文档的编写中总结出来的，既满足了共性又满足了个性。

那么为什么要用 from 语句，很简单。如果用 import 语句，那么 pyconfig.py 文件里面 import math 宏包，在主 py 文件里面引用就要使用这样的格式`pyconfig.math.pi`，这既不方便而且违背大家平时惯用的那种 math.pi 格式。

现在我们让可以开始编写自己的宏包吧。

编写宏包

well，编写宏包就是一些 py 文件，然后宏包的名字和 py 文件里面的内容编写好就是了。

我现在编写了一个 pyconfig.py 文件，放在主文件夹（ubuntu 系统）的 pymf 文件里面的。里面定义了一个斐波拉契函数，如下所示：

```
# 菲波那奇数列
def fib(n):
    if n==0:
        return 1
    if n==1:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

然后我们的测试小脚本如下:

code:6-1

```
import os,sys
sys.path.append(os.environ['HOME']+'/pymf')
from pyconfig import *

print([fib(n) for n in range(10)])
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

一些你觉得常用的类和函数就直接放在 pyconfig.py 文件里面吧，然后一些不太常用的你可以分类出来放在其他 py 文件里面，然后 pyconfig.py 文件用 import 或者 from 语句来加载那个宏包即可，这里就不多说了。

import 语句

import 语句的一般使用方法之前已有接触，比如 import math，然后要使用 math 宏包里面的函数或者类等需要使用这样的带点的变量名结构: math.pi。

此外 import 语句还有一个常见的缩写名使用技巧，比如 import numpy as np，那么后面就可以这样写了，np.array，而不是 numpy.array。

from 语句

from 语句的使用有以下两种情况：

```
from this import this
from what import *
```

第一种形式是点名只导入某个变量，第二种形式是都导入进来。我想读者肯定知道这点，使用第二种导入形式的时候要小心变量名覆盖问题，这个自己心里有数即可。

reload 函数

reload 函数可以重新载入某个宏包，这个重载和程序重新运行第一次载入宏包又有点区别。

python3 高级篇

类

静态方法

```
class Test:
#    @staticmethod
    def hello():
        print('aaa')

test=Test()
test.hello()
```

在上面的例子中，我们希望创造一个函数，这个函数和 self 或者其他类都没有关系（这里的其他类一般指继承来的）。如上所示，hello 函数只是希望简单打印一小段字符，如上面这样的代码是错误的，如果我们在这个函数上面加上**@staticmethod**，那么上面这段代码就不会出错了，

```
class Test:
    @staticmethod
    def hello():
        print('aaa')
```

```
test=Test()
test.hello()
```

这样在类里面定义出来的函数叫做这个类的静态方法，静态方法同样可以继承等等，而静态方法通常使用最大的特色就是不需要建立实例，即可以直接从类来调用，如下所示：

```
class Test:
    @staticmethod
    def hello():
        print('aaa')

Test.hello()
```

模块

多个宏包 py 文件组成一个多文件夹目录的整体就是一个模块，这个暂时还用不到而且处理起来更加复杂，暂时略过。

pyqt4GUI 设计

常用的宏包

pickle 宏包

pickle 宏包可以将某一个复杂的对象永久存入一个文件中，以后再导入这个文件，这样自动将这个复杂的对象导入进来了。

将对象存入文件

```
import pickle

class Test:
    def __init__(self):
        self.a=0
        self.b=0
        self.c=1
        self.d=1

    def __str__(self):
        return str(self.__dict__)

if __name__ == '__main__':
    test001=Test()
    print(test001)
    testfile=open('data.pkl','wb')
```

```
    pickle.dump(test001,testfile)
    testfile.close()
```

从文件中取出对象

值得一提的是从文件中取出对象，原来的类的定义还是必须存在，也就是声明一次在内存中的，否则会出错。

```
import pickle

class Test:
    def __init__(self):
        self.a=0
        self.b=0
        self.c=1
        self.d=1

    def __str__(self):
        return str(self.__dict__)

if __name__ == '__main__':
    testfile=open('data.pkl','rb')
    test001=pickle.load(testfile)
    print(test001)
    testfile.close()
```

shelve 宏包