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

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

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：

```
1 #!/usr/bin/env python3
2 #-*-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。

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

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。

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 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。

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

程序中的逻辑

条件语句

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
```

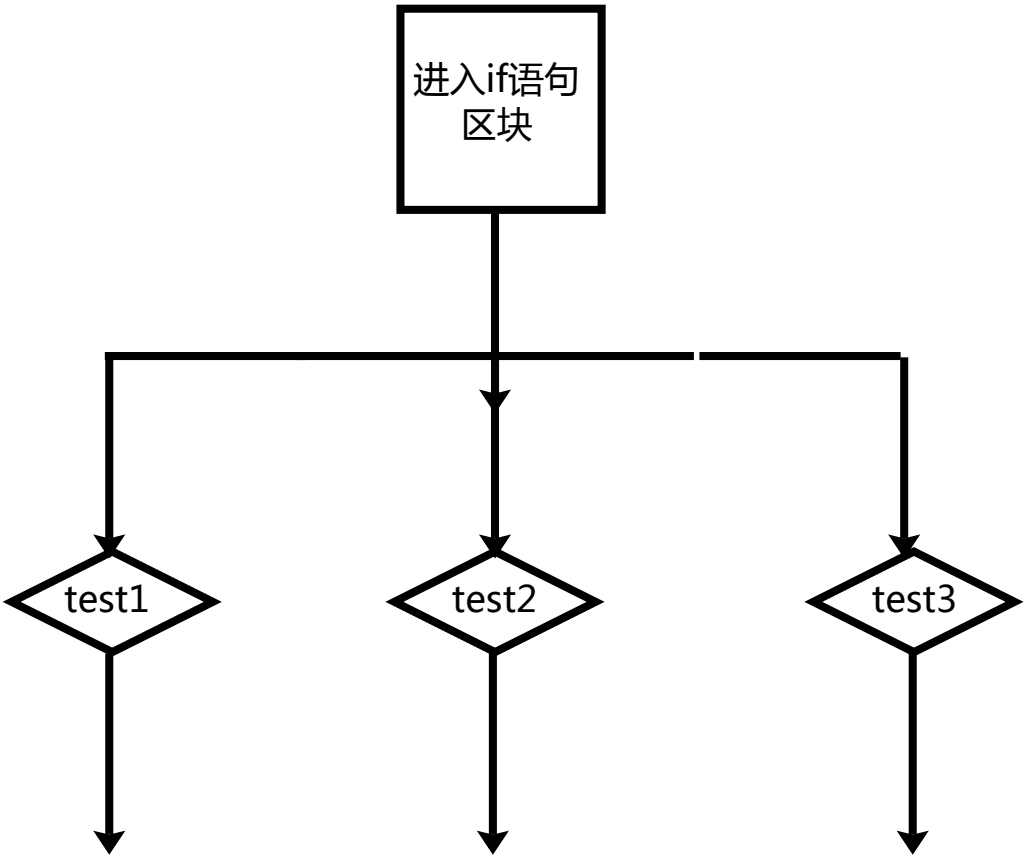
显然你一看就明白了，elif 是 else 和 if 的结合。

code:2-1

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

```
x 等于 0
```

这种结构类似于 switch 语句，你也可以看作类似下面图片描述的横向并排结构。



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:2-2

```
1 x='agais'
2 if ((x[0] == 'a' or x[0] == 'b')
3     and x[-1] =='s'
4     and (not x[-2] ==”’”)):
5     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。

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

```
1 while True:
2     x=input(' 请输入一个数值，我将返回它除以 2 之后的数值 \n 输入"quit" 退出 \n')
3     if x=='quit':
4         break
5     try :
6         num=float(x)
```

```
7     except:
8         print(' 输入有错')
9     else:
10        print(num/2)
11print(' 再见')
```

迭代语句

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:2-3

```
1for x in 'abc':
2    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。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:2-4

```
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。

循环语句

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。

对象同时赋值

下面的例子很好地说明了具体的情况：

平行赋值

```
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)
```

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。

① 这些 int、float 等命令都是强制类型转换命令

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

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。

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，同样带上符号情况有变，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：
$|z| = \sqrt{a^2 + b^2}$

math 宏包

math 宏包提供了以下常数：**math.pi**，**math.e** 等。
还提供了以下函数：
math.sqrt，
math.sin，**math.cos**，**math.tan**，
math.degrees（将弧度转化为角度，这些三角函数默认都是弧度输入。），
math.radians（将角度化为弧度），
math.log10 等。

更多具体细节请参看官方文档。

这里简单写个例子：

```
from math import *
print(pi)
print(sqrt(85))
print(sin(radians(30)))#sin(30°)
```

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。

更多详细细节请参阅官方帮助文档。

下面是一个简单的例子:

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

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:3-1

```
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:3-2

```
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:3-3

```
1 string001='string'
2 list001=['a','b','c']
3 tuple001=(1,2,3,4)
4
5 for x in [string001,list001,tuple001]:
6     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:3-4

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

```
4
5 for x in [string001,list001,tuple001]:
6     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:3-5

```
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

replace 方法

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))
```

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:3-6

```
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。

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。

remove 方法

remove 方法和 lisp 中的 remove 含义大致接近，就是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。

index 方法

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:3-7

```
1 def subst(list001,element001,element002):
2     try:
3         list001.index(element001)
4     except ValueError:
5         return list001
6     else:
7         n=list001.index(element001)
8         del list001[n]
9         list001[n:n]=[element002]
```

```
10          return subst(list001,element001,element002)
11
12 print(subst([1,'a',3,[4,5]],[4,5],'b'))
13 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。

让某个函数依次作用于列表中的元素

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:3-8

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

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

再来看 python 中的情况：

code:3-9

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

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

```
<map object at 0xb70d2e6c>
[1, 4, 9, 16, 25]
```

用 map 函数将会生成一个 map 对象（？），需要外面加上 list 函数才能生成列表形式。第二种形式直接生成列表形式，而且风格更加接近 python，还是推荐使用这种形式把。

count 方法

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:3-10

```
from random import *
def random_list_max(n):
    y=[randint(1,n) for x in range(10000)]
    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
```

find-if remove-if remove-if-not find-if 是将一个判断函数作用于一个列表，然后返回第一个为真的元素。remove-if 是将一个判断函数作用一个列表，真则移去。remove-if-not 这个更常用一些，将一个判断函数作用一个列表，假则移去，留下真的部分。

reduce reduce 是将一个函数作用于一个列表，这个函数需要两个 input 参量，简单来说就是比如说加法，那么这个列表上的数字依此全部加上。大体过程如下图所示：

every every 对后面列表中每一个元素用前面的判断函数来进行判断，如果有一个 nil 那么返回 nil，如果都是 T 那么返回 T。

some

用于列表的 push 和 pop 宏

用牛顿迭代法求平方根数值分析的牛顿迭代法是一种通用方程式求解方法，这里就简单的计算开平方根。

让 n 从 0 到多少记数，然后让另外一个值表示对应的某个函数。比如数学中的级数概念。现在有几何级数，0 对应 1，1 对应 1/2 的一次方，2 对应 1/2 的 2 次方....

元组

圆括号包含几个元素就是元组。

字典

文件

集合

其他内置对象

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。

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:4-1

```
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:4-2

```
def fact(n):
    if n == 0:
```

```
3         return 1
4     else:
5         return n*fact(n-1)
6
7 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:4-3

```
1 def dosomething(n):
2     if n==0:
3         pass
4     elif n==1:
5         print('do!')
6     else:
7         print('do!')
8         return dosomething(n-1)
9
10 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。

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。

lambda 函式在 python 这里主要针对某些简单的函数。比如下面：

code:4-4

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

```
6
```

模块

现在让我们赶快进入模块基础知识的学习，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。

现在让我们建立一个文件，名字就叫做“mypython3.py”。我们知道用 import 命令可以导入某个模块名，但模块内具体的函数等还是要用 mypython3.fib 这样的形式调用，这不太友好。

自建宏包搜索路径管理

在这里我们新建一个 mypython3.py 文件，然后将前面的 fib 函数放进去。接下来要做的就是宏包的搜索路径管理，让 python 编译器能够找到你编写的这个脚本。

这个搜索路径你可以使用`python3 --help` 来查看，是 PYTHONPATH 这个量控制的。现在要做的是让终端每次登录的时候加载上这个变量。

```
1 PYTHONPATH=$HOME/pymf
2 export PYTHONPATH
```

上面的`$HOME` 就是你的主文件目录，然后我新建了一个 pymf 文件夹。这两行内容你写入主文件目录下的**.bashrc** 文件里面，就可以立即生效。

这个网站解释说**.bashrc** 文件是单独每个交互 shell 启动时加载的文件。有些网站谈及修改`.bash_profile` 这个文件，我试了下在 Ubuntu13.10 下并不行。

作为 tex 下的 write18 命令执行 python 脚本似乎并不加载任何 shell 脚本，也没有办法，在这里只好手工加载了。在终端下会自动加载上面的配置，目前还找不到好的解决办法。

导入模块内的变量或者函数

```
form mypython3 import fib
```

这样你就可以直接使用 fib 名字来调用 fib 函数了。

code:5-1

```
1 import os,sys;sys.path.append(os.environ['HOME']+'/pymf')
2 from pyconfig import fib
3 for x in range(5):
4     print(fib(x))
```

```
1
1
2
3
5
```

上面第一行稍作处理至少让其更具通用性吧，多上一行代码也算不上什么事，至少让其更具通用性，目前就这样把。

python 知识高级篇

类

person 数据库

类名一般都大写。

特殊的__init__ 用于这个类具体创建 instance 实例的时候执行的动作。self 表示创建的那个实例，self.name 表示实例的名字，self.name = name 表示接受的 name 将会传递值给 self.name，同时创建的那个实例将会拥有一个自己的 name 属性。其他属性操作类似。__init__ 这里实际上也是进行了函数重载。

和一般函数的做法一样，job=None，表示 job 这个参数是一个可选参数，它有一个默认值 None。

这里新建了一个 Person 类，这个类有三个属性：name，姓名，job，工作，和 pay，薪酬。

新建了 lastname 方法，将会该实例的名字的最后的姓氏。

新建了 giveRaise 方法，还需要一个参数 percent，这样该实例的 pay 属性将会提高这么多百分比。这里的 int 函数是将数值转化为整数。

重新定义__repr__，将会影响 print 函数的行为。

Manager 类

这里定义了一个类 Manager，它还接受一个参数 Person，表示它是 Person 的子类，即一切 Person 类的属性它都将继承。

这里重新定义了 giveraise 方法，用一种巧妙的方式。直接借用 Person 类原有的 giveraise 方法，对参数输入稍作修正。

重载`__init__`方法，提供更加灵活的本地方案。

在 python 中：

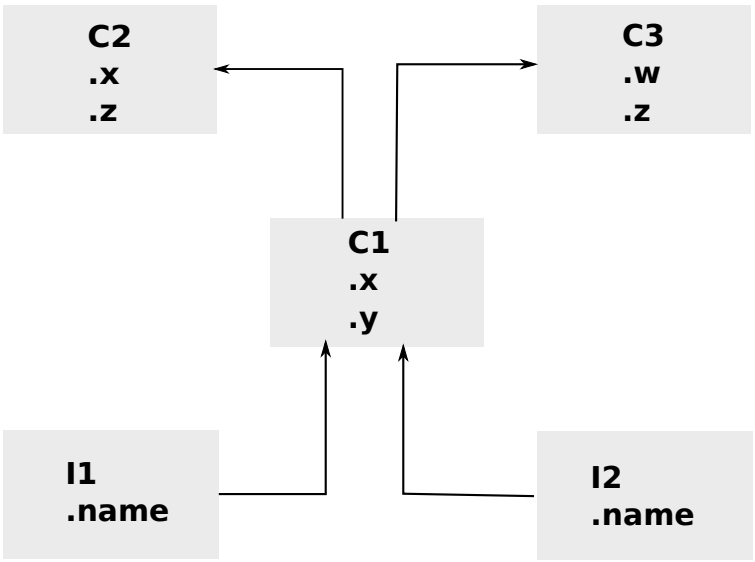
```
instance.method(args...)
```

都会化成这样的形式：

```
class.method(instance, args...)
```

这种转换只针对实例。

python 中超类，子类，实例的重载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。比如说实例没有，就向上搜索那个子类，子类没有就向上搜索那个超类。

如果 py 文件是 import 的形式，那么这段代码将不会执行。只有

以脚本 python3 test.py 之类的形式才执行。

可见继承过来的类将不会有自己默认定义的__init__ 等方法。

tkinter 基础

tkinter 第一个例子

安装 tkinter 宏包

在 ubuntu 下运行下面命令即可：

```
sudo apt-get install python3-tk
```

刚开始

文件 example001.py 在“tkinter 第一个例子”文件夹中：

加上按钮和事件

这是代码文件的开头，其中代码第 5 行使用了星号，这个对于其他宏包并不合适，不过 tkinter 会根据需要来加载类等，所以直接全部引用也没事

这里定义了一些简单的函数，后面会用到的，没什么难理解的。代码第 1 行根据类 Tk 新建了一个实例 mainwindow，这个是惯例，表示母窗体。

这里新建了一个标签，前面谈及的。不同的是采用的匿名方式，直接用 Label 类生成一个匿名 Label 实例然后 pack 进默认母窗体中。后面 pack 函数的 **side** 有选项：**TOP**，**BOTTOM**，**LEFT**，**RIGHT**

表示这个标签的绑定地方，默认是 TOP。

这里根据 Button 类新建了一个按钮实例，各个参数意义很明显，和上面标签声明不同，这里采用的是标准的类的声明使用方式，以后都推荐使用这种方式。

这里代码第 1 行使用了按钮实例 button001 的 bind 方法，这个方法将某一个事件和某一个方法（操作或者函数）绑定起来。这里 **<Button-1>** 是鼠标左键，总的意思是如果按钮 button001 上鼠标左键点击了，那么执行 hello 函数。

后面类似的定义了一个按钮实例 button002，其中 <Double-1> 是鼠标左键双击的意思。

然后调用 mainloop() 函数，整个程序的事件监听循环跑起来。

加入 lambda 函式

```
1 #! /usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 import sys
5 from tkinter import *
6
7
8 mainwindow=Tk()
9
10 def quit(event):
11     print(’ 拜拜.....’)
12     sys.exit()
13
```

```
14
15 Label(text=’ 你好’).pack( side=TOP)
16
17
18 button001=Button(mainwindow, text=’ 点我’)
19 button001.pack(side=LEFT)
20
21
22 button001.bind(’<Button-1>’,lambda event  : print(’ 双击’))
23 button001.bind(’<Double-1>’,quit)# 多个事件
24
25 button002=Button(mainwindow, text=’ 你好’)
26 button002.pack(side=RIGHT)
27 button002.bind(’<Button-1>’,lambda event : print(’ 你好’))
28
29 mainloop()
```

这里和之前的区别就是加入了 lambda 函式，注意第一个参数 event 是必须的。

tkinter 的各个 widget

在 tkinter 中，所有图形部件（widget）（比如 Button，Checkbutton 等）都是类 Widget 的子类，而且他们都是一个平行的继承层次。

Label 类

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

from tkinter import *

tk001=Tk()
labelfont=(' 颜体',20,'bold')
label001=Label(tk001,text=' 你好，万泽。')
label001.config(bg='black',fg='yellow')
label001.config(font=labelfont)
label001.config(height=3,width=20)
label001.pack(expand=YES,fill=BOTH)

mainloop()
```

这里 label001 是根据 Label 类新建的一个标签实例，其依附母体是 tk001，标签文本是“你好，万泽。”。然后 pack 方法使用 **expand=YES,fill=BOTH**，也就是这个标签会填充整个窗口。然后 config 方法进行了这个标签的一些属性设置：

bg 调整背景颜色

fg 调整字体颜色

font 设置其他字体属性，这里有一系列的属性，分别是字族、大小和粗体（此外还有 normal，roman，italic，underline，overstrike[①]。

height 和 width 调整这个标签实例的高和宽。

然后设置了这个标签的初始大小。

字体 **Times** ,**Courier** , **Helvetica**tkinter 保证可以使用，当然其他系统内的自己也可以使用。

Button 类

```
1 #!/usr/bin/env python3
2 #-*-coding:utf-8-*-
3 from tkinter import *
4
5 button001 = Button(text=’ 点我’, padx=10, pady=10)
```

① 删除线

```
6 button001.config(cursor='hand2')
7 button001.config(bd=5, relief=RAISED)
8 button001.config(bg='dark green', fg='white')
9 button001.config(font=('helvetica', 20, 'underline italic'))
10
11 button001.pack(padx=20, pady=20)
12 mainloop()
```

上面新建了一个 Button 类，其中 padx 是按钮的宽，pady 是按钮的高。然后接下来 config 函数配置的有:

cursor 调整鼠标指针移动到按钮上时候的形状，有选项：hand2，gumby，watch，pencil，cross。

bd 调整按钮最外围的边框宽度。

relief 调整按钮的类型，有选项：FLAT，SUNKEN，RAISED，GROOVE，SOLID，RIDGE。

Dialogs

```
1 #!/usr/bin/env python3
2 #-*-coding:utf-8-*-
3
4 from tkinter import *
5 from tkinter.messagebox import *
6
7 def callback():
8     if askyesno(' 确认信息', ' 你真的要退出吗？'):
```

```
9         showwarning(’ 确认退出’, ’ 退出’)
10    else:
11        showinfo(’ 确认不退出’, ’ 不退出’)
12
13 errmsg=’ 没有 spam!’
14 Button(text=’Quit’, command=callback).pack(fill=X)
15 Button(text=’Spam’, command=(lambda: showerror(’Spam’, errmsg))).pack(fill=X)
16 mainloop()
```

Entry 类

```
1 from tkinter import *
2 from quitter import Quitter
3 def fetch():
4     print(’Input => ”%s”’ % ent.get()) # get text
5 root = Tk()
6 ent = Entry(root)
7 ent.insert(0, ’Type words here’)
8 ent.pack(side=TOP, fill=X) # set text
9 # grow horiz
10 ent.focus()
11 ent.bind(’<Return>’, lambda event: fetch())
12 btn = Button(root, text=’Fetch’, command=fetch)
13 btn.pack(side=LEFT)
14 Quitter(root).pack(side=RIGHT)
15 root.mainloop()
```

通过类重用自己的 GUI

tkinter 实例

小型计算器程序

pyqt4 基础

第一个例子

安装 pyqt4

ubuntu 下安装 pyqt4 即安装 python3-pyqt4 即可:

```
sudo apt-get install python3-pyqt4
```

检查 pyqt4 安装情况执行以下脚本即可，显示的是当前安装的 pyqt4 的版本号:

```
1 from PyQt4.QtCore import QT_VERSION_STR
2 print(QT_VERSION_STR)
```

pyqt4 模块简介

QtCore 模块包括了核心的非 GUI 功能，该模块用来对时间、文件、目录、各种数据类型、流、网址、媒体类型、线程或进程进行处理。

QtGui 模块包括图形化窗口部件和及相关类。包括如按钮、窗体、状态栏、滑块、位图、颜色、字体等等。

QtHelp 模块包含了用于创建和查看可查找的文档的类。

QtNetwork 模块包括网络编程的类。这些类可以用来编写 TCP/IP 和 UDP 的客户端和服务器。它们使得网络编程更容易和便捷。

QtOpenGL 模块使用 OpenGL 库来渲染 3D 和 2D 图形。该模块使得 Qt GUI 库和 OpenGL 库无缝集成。

QtScript 模块包含了使 PyQt 应用程序使用 JavaScript 解释器编写脚本的类。

QtSql 模块提供操作数据库的类。

QtSvg 模块提供了显示 SVG 文件内容的类。可缩放矢量图形 (SVG) 是一种用 XML 描述二维图形和图形应用的语言。

QtTest 模块包含了对 PyQt 应用程序进行单元测试的功能。（PyQt 没有实现完全的 Qt 单元测试框架，相反，它假设使用标准的 Python 单元测试框架来实现模拟用户和 GUI 进行交互。）

QtWebKit 模块实现了基于开源浏览器引擎 WebKit 的浏览器引擎。

QtXml 包括处理 XML 文件的类，该模块提供了 SAX 和 DOM API 的接口。

QtXmlPatterns 模块包含的类实现了对 XML 和自定义数据模型的 XQuery 和 XPath 的支持。

phonon 模块包含的类实现了跨平台的多媒体框架，可以在 PyQt 应用程序中使用音频和视频内容。

QtMultimedia 模块提供了低级的多媒体功能，开发人员通常使用 **phonon** 模块。

QtAssistant 模块包含的类允许集成 **Qt Assistant** 到 PyQt 应用程序中，提供在线帮助。

QtDesigner 模块包含的类允许使用 PyQt 扩展 **Qt Designer**。

Qt 模块综合了上面描述的模块中的类到一个单一的模块中。这样做的好处是你不用担心哪个模块包含哪个特定的类，坏处是加载进了整个 Qt 框架，从而增加了应用程序的内存占用。

uic 模块包含的类用来处理.ui 文件，该文件由 Qt Designer 创建，用于描述整个或者部分用户界面。它包含的加载.ui 文件和直接渲染以及从.ui 文件生成 Python 代码为以后执行的类。

第一个例子

文件：第一个例子.py

前面的注释部分就不用说了，然后导入 sys，是为了后面接受 sys.argv 参数。导入 QtGui 是为了后面创建 QWidget 类的实例。

任何程序都需要创建一个 QApplication 类的实例，这里是 app001，后面跟著数字 001 就是为了强调这是一个实例。

然后接下来创建 QWidget 类的实例 widget001，首先是引用类的`__init__`方法，然后 QWidget 类里面有 resize 方法，这个方法调整等下生成的程序窗口的大小。而 setWindowTitle 方法设置等下程序窗口上面的标题。show 方法就是显示这个窗口。

后面我们看到系统要退出是调用的 app001 实例的 exec_ 方法，这一句还不太清楚。

加上图标

现在我在前面第一个程序的基础上稍作修改，来给这个程序加上图标。为了模拟 Texmaker，程序的名字就叫做 Texmaker。

因为自己的 DIY 开始变多了，所以这里新建了一个类，名字就简单叫做 MyQWidget，然后重新定义了这个类的初始函数。首先是继承自 QtGui.QWidget 类，然后延续了该类的初始函数，而 parent 被默认为 None。

然后用 QWidget 的 setGeometry 方法来调整窗口的左上顶点的

坐标和窗口的 X，Y 的大小。这里 0，0 表示从屏幕的最左上点开始显示，同样 800，600 类似前面的 resize 函数的配置。

setWindowTitle 方法前面谈论过了，这里加入图标是通过 setWindowIcon 方法来做到的。这个方法调用了 QtGui.QIcon 方法，不管这么多，后面跟的就是图标的存放路径，使用相对路径。在运行这个例子的时候，请随便弄个图标文件过来。

窗口弹出提示信息

接下来要做的 DIY 就是让这个窗口可以弹出提示信息，就是鼠标停置一会儿会弹出一段小文字。

上面这段代码和前面的代码的不同就在于 MyQWidget 类的初始函数新加入了两条命令。其中 setToolTip 方法设置具体显示的文本内容，而 之间的文字会加粗。然后后面那条命令是设置字体和字号的，我不太清楚这里随便设置系统的字体微软雅黑是不是有效。

退出的时候询问

目前程序点击那个叉叉图标关闭程序的时候将会直接退出，这里新加入一个询问机制。

接下来要做的 DIY 就是让这个窗口可以弹出提示信息，就是鼠标停置一会儿会弹出一段小文字。

这段代码重新了原来的 colseEvent 方法，这里调用的那个方法内部“信息”两个字是弹出的信息框的标题，后面是信息框里面显示的文字。这里具体代码我还不是很懂。

居中显示窗体

接下来要做的 DIY 是让窗体弹出的时候居中显示，前面是设置了窗体的起点坐标的，这里新建了一个 center 方法来确认窗体居中显示。

这里做的改动就是新建了一个 center 方法，接受实例。然后对这个实例也就是窗口的具体位置做一些调整。前面使用了 resize 和 center 两个方法来调整窗口的大小和窗口的位置。

从 center 方法中我们可以看到 move 方法的 X，Y 是从屏幕的坐标原点（0，0）开始计算的。第一个参数 X 表示向右移动了多少宽度，Y 表示向下移动了多少高度。

第二个例子

文件：第二个例子.py

第二个例子是在第一个例子的基础上进行进一步的修改。

QtGui.QMainWindow 类提供应用程序主窗口，可以创建一个经典的拥有状态栏、工具栏和菜单栏的应用程序骨架。（之前使用的是 QWidget 类，现在换成 QMainWindow 类。）

前面第一个例子都是用的 QtGui.QWidget 类创建的一个窗体。关于 QWidget 和 QMainWindow 这两个类的区别参考这个网站得出的结论是：QWdget 类在 Qt 中是所有可画类的基础（这里的意思可能是窗体的基础吧。）任何基于 QWidget 的类都可以作为独立窗体而显示出来而不需要母体（parent）。

QMainWindow 类是针对主窗体一般需求而设计的，它预定义了菜单栏状态栏和其他 widget（窗口小部件）。因为它继承自 QWidget，所以前面谈及的一些属性修改都适用于它。

加上状态栏

在`__init__`方法下加入语句：

```
self.statusBar().showMessage('这是状态栏')
```

这样就显示了状态栏信息。

这里用 QMainWindow 类的 statusBar 方法获得状态栏，然后用状态栏的 showMessage 方法插入状态栏信息。

加上菜单栏

用 QMainWindow 类的 menuBar 方法来获得一个菜单栏。然后用这个菜单栏对象的 addMenu 方法来创建一个新的菜单对象——方法里面的内容是新建菜单显示的名字。

菜单加上动作

某个菜单对象使用 addAction 方法来加上某个动作。

```
file.addAction(exit)
```

加上动作

也就是所谓的 Action 对象，通过 QtGui 类的 QAction 子类创建动作对象。创建过程的三个参数是图标，文本和母体。

```
exit.setStatusTip('退出程序')
```

setStatusTip 方法设置状态栏提示信息。

事件和信号

将事件和信号联系起来用 connect 方法，该方法接受三个参数：第一个是对于谁，第二个是做了什么，第三个下面做什么。

```
1 self.connect(exit, QtCore.SIGNAL('triggered()'),
2  QtCore.SLOT('close()'))
```

这里的意思是对于 self 上面的 exit 对象，接受了信号 triggered，然后执行操作 close()。

第三个例子

加上工具栏