Python快速入门教程

原作者：张路

修改人：谢作如

注：用鼠标选中In[ ] 的代码，点击菜单中的“运行”。运行成功后，括号中将出现数字，表示执行次序。

为什么要学Python?

第一，简单，跟写字说话差不多
第二，好玩，跟玩捉迷藏差不多
第三，有用，差不多什么都能做

计算机能干啥?

看似无所不能，控制各类机器、装置，加工处理各类信息。实则大计算器是也，擅长处理各种数据类型，进行各种运算，除加减乘除等简单计算外，还能三角函数、数据统计等。

Python能干啥?

别想太多了，**Python至少能当个不错的计算器用**!

In [ ]:

```
# 比如，我们来算1+1
1+1
```

In [ ]:

```
1+2
```

In [ ]:

```
# 再算一个复杂一点的
1+2+3+4+5+6+7+8+9+10
```

不过，虽然计算机有把子傻力气，给多少数都能很快计算出来，但我们输入这些数据的时间就变长了，所以算得再快也没用，就好像飞机虽然跑得快，但是从市里到飞机场花的时间很长一样。这让我想起《测量世界》中的一个情节，高斯的数学老师为了偷闲吃个苹果，让孩子们一个数一个数地从1加到100，刚开始为了看孩子们的反应，还是一个数一个数地写，后来干脆就说就这样一直加到100。那么我们有办法像高斯的老师那样，告诉计算机你就从1加到100就行了？答案是有的，但是光靠列算式是解决不了的，这回我们要下一堆命令了。

首先我们要让计算机明白计算的范围是1到100，在英语里范围叫range，所以我们要给计算机出个**完形填空题** `range()`，但这里的数是要我们来填的，1到100的写法就是"1,100"，合起来1到100这句话的说法就是range(1,100)。

In [ ]:

```
list(range(1,101))
```

这种下命令的形式就叫函数，因为无法用数学符号表示，所以就要用词语和填空的形式，其中词语叫做命令或函数名，需填空的部分叫做参数。

知道范围怎么表达了，然后就要让计算机知道，我们是想把这个范围里的数字一个一个去加，那么如何表达一个一个这件事呢，这就用到了另一个句式，叫for什么什么in什么，也就是对于数字范围内的每个数，怎么着怎么着。

In [ ]:

```
# 先写要做多少次
for num in range(1,101):
    # 然后写每一次我们做什么事情
    pass
```

这里我们用到了另一个词num，它是number的缩写，因为我们要把取出来的数字先记下来，然后才能去进行运算，所以这里用了一个叫num的卡片，把取出来的数先下上去，这样计算的时候方便我们查看，取另外一个数的时候，我们再把原来的数擦掉，然后再写上新的数字，这在计算机语言里叫做存储变量或变量。很明显，我们还缺少一个表示结果的卡片sum，所谓的加法不过就是先把sum设为零，然后每取一个num，就把它和sum相加，用得到的结果再去更新sum，如此反复，这和我们手工计算的方式是一样的，只不过计算机算的比较快而已。

In [ ]:

```
%%latex
\begin{align}&1\\+&2\\\hline&3\\+&3\\\hline&6\\...\end{align}
```

In [ ]:

```
sum = 0
for num in range(1,101):
    temp = sum + num
    sum = temp
print(sum)
```

这里我们浪费了一个存储变量temp，因为它只起到保存中间结果的作用，其实把中间结果直接保存到sum里就可以了，只是样子可起来有点令人费解。

In [ ]:

```
sum = 0
for num in range(1,101):
    sum = sum + num

print(sum)
```

习惯就好了。还有更偷懒的办法呢。

In [ ]:

```
sum = 0
for num in range(1,101):
    sum += num

print(sum)
```

记住，图省事是程序员的天性，尤其是男性程序员。程序员最怕敲入重复的东西，所以想尽一切办法来避免这种现象或问题的出现，除了采用改进语言的办法外，后面我们还会看到各种变态的招数，他们一般管这个叫做代码复用，其实就是嫌做重复的事情啰嗦、乏味，所以当你写代码时出现大量的拷贝、粘贴或出现重复的词句时就要警惕了，是不是代码写的有点啰嗦、难看了？

现在，尽管看上去我们也能计算出结果，但如果数字比较少的话，每回都要敲这么写东西去计算貌似也挺费劲的，而且因为那些英文单词记错或句子写错还容易计算出错，最好我们只写一次，让计算机记住，然后我们每次只要告诉一个词就行了。嗯，确实是个好主意，这在程序里叫做自己编一个命令，让计算机记住，以后只要告诉计算机编的那个命令就可以了，正规的说法这叫**定义**一个函数。

定义在英文里叫做define，图省事我们就把它叫def吧，定义什么呢？有两个要素，一个是命令的名字比如add，一个是命令要处理的对象(numbers)，当然，我们也可以直接定义一个名字就行了，但是填空的括号必须留着，把上面的代码直接粘进来就可以了

In [ ]:

```
def sum():
    sum = 0
    for num in range(1,101):
      sum += num

    print(sum)
```

这样，我们要计算从1到100的和，只要打sum()就行了

In [ ]:

```
sum()
```

算多少遍都可以，但是慢着，如果我们想计算1加到500怎么办？是不是还要从头写那些代码，所以看起来这个sum()看起来不错，但是好像是一锤子买卖，不堪大用，有什么好办法呢？

这时候，上面讲得参数就派上用场了，我们可以为参数设定一个存储变量numbers，这样当用户调用我们编的命令时，只需把要计算的数填到空空里，就等于给numbers赋值了，然后我们在命令代码中再把 `for num in range(1,101)` 中的数据范围替换成numbers就可以了。

In [ ]:

```
def sum(numbers):
    sum = 0
    for num in numbers:
      sum += num

    print(sum)
```

In [ ]:

```
sum(range(1,101))
```

In [ ]:

```
sum(1)
```

我们看到如果调用sum时，我们没有按要求输入一个范围就出现错误了，最好我们能给出一个提示，当然前提是我们能够知道用户所输入的数据类型，python给我们提供了一个指令叫type。

In [ ]:

```
type(range(1,101))
```

In [ ]:

```
type([1,2,3,4,5])
```

进行判断的命令叫做if，需要判断的事情是numbers的类型，也就是type(numbers)，看看它是否等于range或list，等于在python里用两个等于号表示

In [ ]:

```
def sum(numbers):
    if type(numbers) == list or type(numbers) == range:
      sum = 0
      for num in numbers:
        sum += num
      print(sum)
    else:
      print("参数错误，应该输入range或列表")
```

In [ ]:

```
sum(1)
```

In [ ]:

```
sum([1,2,3,4,5])
```

In [ ]:

```
sum(range(1,101))
```

但是如果我们还像保留以前的爽体验，却出现了错误

In [ ]:

```
sum()
```

不要紧，python还提供了另外一件武器，就是可以给参数设默认值!

In [ ]:

```
def sum(numbers = range(1,101)):
    if type(numbers) == list or type(numbers) == range:
      sum = 0
      for num in numbers:
        sum += num
      print(sum)
    else:
      print("参数错误，应该输入range或列表")
```

In [ ]:

```
sum()
```

这样看虽然是解决暂时的问题了，但是感觉用起来还是别扭，比如为什么我们不能用 `sum(1,2,3,4,5)` 来计算少数量的加法，而要多加那两个中括号呢，为什么我们不能用 `sum(form=1,to=100)` 来表示从1加到100，要用一个莫名其妙的命令range呢？很好的问题！ 但是我们的程序或判断逻辑就因此变得复杂起来了，而且这其中还用到可变参数的技巧。

In [ ]:

```
def sum1(*numbers, **nameNumbers):
    pass
```

这里的星号 * 代表是可以有任意多(包括没有)的参数，它们都被存入numbers列表中，两个星号则是表示也可以使用name=value的方式输入参数，比如 `from=1, to=101` ，输入的参数会被存储在字典nameNumbers中，我们需要在命令的逻辑中分别处理这两种情况。

In [ ]:

```
def sum(numbers = range(1,101)):
    sum = 0
    if type(numbers) == list or type(numbers) == range:
      for num in numbers:
        sum += num
    return sum
```

In [ ]:

```
def sum1(*numbers, **nameNumbers):
    sum2 = 0
    if len(nameNumbers) == 2 and 'f' in nameNumbers and 't' in nameNumbers:
        sum2 += sum(range(nameNumbers['f'],nameNumbers['t']))
    if len(numbers) > 0:
        sum2 += sum(list(numbers))

    return sum2
```

In [ ]:

```
sum1(1,2,3)
```

In [ ]:

```
sum1(f=1,t=101)
```

In [ ]:

```
sum1(1,2,3,f=4,t=101)
```

In [ ]:

```
sum1()
```

这个功能看起来已经很完整了，那么我以后另外的程序要用怎么办？很简单，把它们做成模块，然后在需要的地方**导入**import进来，记住程序员很讨厌做拷贝粘贴哟。

比如，你可以把下面的代码，复制到一个文本文件中，命名为：mymodule.py。

def add(*numbers, *nameNumbers): sum2 = 0 if len(nameNumbers) == 2 and 'f' in nameNumbers and 't' in nameNumbers: sum2 += sum(range(nameNumbers['f'],nameNumbers['t'])) if len(numbers) > 0: sum2 += sum(list(numbers))

```
    return sum2
```

In [ ]:

```
from mymodule import add
add(f=1,t=101)
```

实际上已经早有高手感觉出了问题，为我们提供好了这类功能，当然这需要加入高手做好的模块NumPy。

In [ ]:

```
import numpy
numpy.sum(numpy.arange(1,101))
```

很方便，几乎只要一个指令就可以了，而且从性能上说，当计算所谓的大数据时，NumPy的性能优势就会显现出来了

当然为了一个求和做一个包实在有点奢侈，所以NumPy的功能远不止这些，比如它还可以求最大、最小和平均值呢。

In [ ]:

```
numpy.max(numpy.arange(1,101))
```

In [ ]:

```
numpy.min(numpy.arange(1,101))
```

In [ ]:

```
numpy.mean(numpy.arange(1,101))
```

还有更吓人的，可以求1到100的平方和

In [ ]:

```
numpy.sum(numpy.arange(1,101)**2)
```

相对于上面我们自己做的sum命令，这里出现的外部引用命令还是有些明显的变化，就是在命令之前多了一个 `numpy.` ，这其实是为了指明后面指令的出处，因为在我们的程序中出现了两个sum，如果不指明出处的话，计算机就会不知所措。另外，如果我们不小心又定义了一个sum指令，还会把外部引入的指令覆盖掉，从而引起程序出错，总起来说，这种通过外部引用的方式自动隔离同名指令的办法确实很实用，有点像我们把重名的人前面加一个家庭或地域的修饰那样，比如北京的张三、天津的张三等。但这也是有代价的，就是要额外增加一个文件。那么，有没有一个方法在一个文件内部也实现这种方法之间的**归类**和隔离呢？这就引入了另外一个词--**类**，英文名字叫class，顾名思义，就是要把不同的定义(包括变量和函数)按不同的特点和类型进行归类，避免混淆，比如我们可以定义一个叫MyMath的类。

In [ ]:

```
class MyMath:
  def sum(self,numbers = range(1,101)):
    sum = 0
    if type(numbers) == list or type(numbers) == range:
      for num in numbers:
        sum += num
    return sum

  def add(self,*numbers, **nameNumbers):
    sum2 = 0
    if len(nameNumbers) == 2 and 'f' in nameNumbers and 't' in nameNumbers:
        sum2 += sum(range(nameNumbers['f'],nameNumbers['t']))
    if len(numbers) > 0:
        sum2 += sum(list(numbers))

    return sum2
```

In [ ]:

```
mylib = MyMath()
mylib.add(1,2,3)
```

注意，为了使类内部的方法之间能够相互调用，在方法的参数中增加了self这一项，英文意思就是自我，看上去模模糊糊的意思是代表类本身，但是联系到前面关于函数的定义，我们可以知道类本身只是一个定义，如果不被调用的话，它是不会主动生成数据的，再结合类使用的方式 `MyMath()` ，很容易就猜想出，self代表执行MyMath()后所生成的数据对象，只不过我们没有在类的定义里明确return而已。之所以要采用统一的名字，是因为尽管结果已经存在内存里面，但我们不知道最终用户定义的引用名称会是什么，所以暂时先用self代替着，当执行 `mylib = MyMath()` 以后这个对象就变成mylib了，里面存储的内容是两个方法sum和add。

一般把这种归类的编程模式叫做面向对象的编程模式，除了能使程序的结构更清晰和便于管理外，还可以通过继承等机制实现代码功能的复用，避免单纯的函数复用造成的混乱局面，实际上这也是Python编程的常用模式。我们在课程之初，其实遇到了两种命令模式，一种是填空式的函数调用，一种是非填空的语句模式，二者的风格貌似有点不太统一，对于初学者可能觉得有点无章可循，实际我们看那些非填空的指令大多数是一句话说不清楚的指令，我们把这种指令叫做复合指令，如果用函数的方式来表示就会变得非常复杂，但其实程序界也有洁癖的语言叫做Lisp，它就坚持用填空的统一风格，甚至连函数名都一并写进去，比如一个for...in语句写起来大略是这个样子：

```
(for (num in nums) (print num))
```

虽然这种括号复括号的方式有些极端，但风格不统一看起来终究有些不舒服，解决的办法就是面向对象。比如前面我们用numpy表示1到100的平方数，采用的是 `numpy.arange(1,101)**2` 的形式，而没有采用中间变量和循环语句，这样整个风格看起来似乎就比较统一了，里面的机制就是 `numpy.arange(1,101)` 返回的是一个所谓的numpy对象，而不是传统意义上的list或range，而且这个对象中定义了 `**2` 所对应的方法，比如我们也可以用power这个词儿来重写上面的语句 `numpy.arange(1,101).power(2)` ，这样看起来是不是就顺溜多了，用人话来说就是把1到100的数每一项都做一遍乘方运算。为了实现这一点，需要在numpy中设立一个保存数值的属性，比如self.value，然后每次调用方法其实都是在修改这个属性，然后将修改后的numpy再吐出来，等待后续处理。

这给我们的启示是，如果你想要Python的语句风格像Processing那样大致统一，那么最好就用面向对象的方式把所有风格不统一的语句重新实现一遍、封装起来。

虽然如此，现实生活中的应用不仅要计算，更需要选择和判断，比如超市卖东西，除计算累积额度外，还要根据额度进行优惠和打折，这样就需要在基本的计算之上，增加额外的步骤对计算结果进行判断比较，然后对其进行加工修改，这个过程就叫程序，其基础是向计算机发出的一串指令，一般以语句或句子的方式呈现。这和我们日常生活中向孩子发出指令是一样的，如果只是用词和字谁也听不懂，比如“吃“、“买”，要成句才能搞清楚语义，比如你来吃饭、你去打瓶酱油来，对计算机也一样，必须提供的信息完整，它才能理解你到底想干什么。

计算机除了算术好，记忆力强外，其实在智力上基本白痴一个，人事基本不懂，所以写程序就跟教小孩儿一样，要有耐心，要自己的想法和意愿用对方的语言表达出来，对于计算机而言，就是想办法把所有事都转化为计算和记忆这两件事，计算机就能像一个大傻子似地不惜力地去干了。最基本的，我们把构成语言的字母进行编号，就把他们转化为数字处理了。更进一步说，是把一切事物转化为计算，再把计算转化为是非相关的逻辑运算，这样计算机就能处理了。典型的例子是，我们在教电路时对进位的处理方式，计算机其实并不懂进位，但这不妨碍我们采用逻辑和空间扩展规则进行模拟，这其实就是图灵机的基本原理吧。

总之，如果没有选择和逻辑，计算机其实和一个大计算器无异，如果逻辑简单且不需要重复使用和控制，那么计算器加人工也能简单应付，怕的是要处理和判断的事物量巨大且复杂，而且需要对其处理的准确性、精确性和处理效率有较高的要求，那么就需要把人做的事情教会计算机去做，这种教的过程就是编程。