云南大学数学与统计学院
上机实践报告

课程名称：数据结构与算法实验	**年级**：2015 级	**上机实践成绩**：
指导教师：陆正福	**姓名**：刘鹏	
上机实践名称：高级语言基本编程实验	**学号**：20151910042	**上机实践日期**：2017-04-18
上机实践编号：No.1	**组号**：	**上机实践时间**：上午三、四节

一、实验目的

1. 熟悉基本的 Python 编程，为数据结构与算法的学习奠定实验基础
2. 熟悉教材第一章的代码片段
3. 与其它程序设计语言（如 C/C++/Java 语言等）作对比。

二、实验内容

1. Python 程序的编辑、编译、运行（建议使用 IDLE）
2. 主讲教材第一章的 Python 程序的调试
3. 其它集成开发环境(IDE)的安装、配置、使用（选做），在熟悉基本操作后，可以转入其它集成开发环境的使用，如可选用 Eclipse。

三、实验平台

Windows 10 Enterprise 中文版；
Python 3.6.0；
Wing IDE Professional 6.0.2-1 集成开发环境。

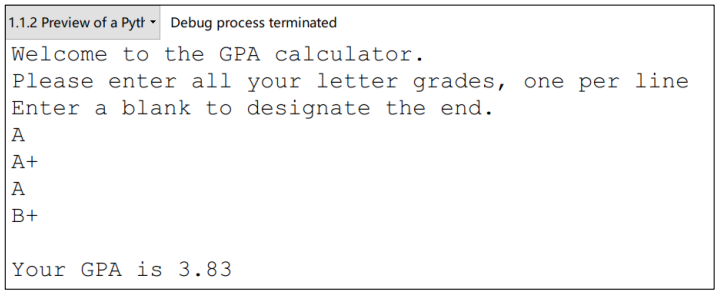
四、实验记录与实验结果分析

1
程序代码：

```
1  # 1.1.2 Preview of a Python Program
2
3  print('Welcome to the GPA calculator.')
4  print('Please enter all your letter grades, one per line')
5  print('Enter a blank to designate the end.')
6  # map from letter grade to point value
7  points = {'A+':4.0,'A':4.0,'A-':3.67,'B+':3.33,'B':3.0,'B-':2.67,\
8           'C+':2.33,'C':2.0,'C-':1.67,'D+':1.33,'D':1.0,'F':0.00}
9  num courses = 0
10 total points = 0
11 done = False
12 while not done:
13     grade = input()
14     if grade == '':
15         done = True
16     elif grade not in points:
17         print("Unknow grade '{0}' being ignored".format(grade))
18     else:
19         num courses += 1
20         total points += points[grade]
21 if num courses > 0:
22     print('Your GPA is {0:.3}'.format(total_points / num_courses))
```

程序代码 1

输出结果



运行结果 1

代码分析：

从代码本身来看，可以看出这一段简单代码中包含了很多 Python 的基本操作，比如基本的循环控制，判断，而且相比较于曾经学过的 C 语言，Python 还多了字典这个类型。如果要用 C 语言实现字典功能，可能就要写一个头文件进行预先定义，从语言的应用角度看，功能越丰富的工具更称手，而且更能使人专注于自己的主要目标。

五、实验体会

Translation:

Chapter 1 Python Primer
＊第一章 Python 语言入门

Python Overview
＊1.1Python 概览

Building data structures and algorithms requires that we communicate detailed instructions to a computer. An excellent way to perform such communications is using a high-level computer language, such as Python. The Python programming language was originally developed by Guido van Rossum in the early 1990s, and has since become a prominently used language in industry and education. The second major version of the language, Python 2, was released in 2000, and the third major version, Python 3, released in 2008. We note that there are significant incompatibilities between Python 2 and Python 3. *This book is based on Python 3 (more specifically, Python 3.1 or later).* The latest version of the language is freely available at **www.python.org**, along with documentation and tutorials.

＊构建数据结构以及算法的时候，我们需要给计算机下达详细的指令。一种绝妙的可以实现这种人机对话的方式就是采用一门高级计算语言，比如说 Python。Python 这门编程语言是由 Guido van Rossum 在上个世纪九十年代早期率先建立，而且自此以后成为了一门被广泛用于工业与教育方面的语言。Python 的第二个主要版本，即 Python 2，已经于 2000 年发布，而它的第三个版本也于 2008 年发布，这也就是 Python 3。我们将会注意到两个版本 Python 有着明显的不兼容。这本书是基于 Python 3 （Python 3.1 及以后的更新的版本）编著的。Python 3 的最新版本可以在 www.python.org 进行免费下载，同时这个网站还提供相应文档与教程。

In this chapter, we provided an overview of the Python programming language and we continue this discussion in the next chapter, focusing on object-oriented principles. We assume that readers of this book have prior programming experience, although not necessarily using Python. This book does not provide a complete description of the Python language (there are numerous language references for that purpose), but it does introduce all aspects of the language that are used in code fragments later in this book.

＊在这一章中，我们将给出一些有关 Python 语言的概览，而且我们将会在接下来的一章中重点介绍面向对象原则。我们假定这本书的读者已经有了前期编程经验，当然这不一定必须是 Python 方面的。这本书没有给出有关 Python 语言的完整描述（有关方面的参考是浩如烟海），但是在这本书中将要用到的那些语言知识，我们都会给出介绍。

1.2.1 Identifiers, Objects, and the Assignment Statement
＊1.2.1 节 标识符、对象、、赋值语句

For readers familiar with other programming languages, the semantics of a Python identifier is most similar to a reference variable in Java or a pointer variable in C++. Each identifier is implicitly associated with the memory address of the object to which it refers. A Python identifier may be assigned to a special object named None, serving a similar purpose to a null reference in Java or C++.

＊对于熟悉其他编程语言的读者而言，Python 的标识符与 Java 语言中的引用以及 C++语言中的指针变量很相似。每一个标识符都标记了它所指向的对象在内存中的地址。一个 Python 的标识符可能被赋值为一个特殊的对象——None，它的作用与 Java 或者 C++中的空引用相同。

Unlike Java and C++, Python is a ***dynamically typed*** language, as there is no advance declaration associating an identifier with a particular data type. An identifier can be associated with any type of object, and it can later be reassigned to another object of the same (or different) type. Although an identifier has no declared type, the object to which it refers has a definite type. In our first example, the characters 98.6 are recognized as a floating-point literal, and thus the identifier temperature is associated with an instance of the float class having that value.

＊Python 不像 Java 或者 C++那样，它是一门动态的语言，因为它前期并没有把给定的标识符与某种固定的数据结构绑定。一个标识符可以和任意类型的对象进行绑定，并且之后，这个标识符可以和其他同类型或者不同类型的对象进行重新绑定。尽管一个标识符没有声明自己的类型，但是它指向的对象却有类型。在我们的第一个例子里，字符 98.6 在字面上被理解为一个浮点型，而 temperature 这个标识符就与拥有着这个数值的一个 float 类的实例联系起来了

A program can establish an ***alias*** by assigning a second identifier to an existing object. Continuing with our earlier example, Figure 1.2 portrays the result of a subsequent assignment, original = temperature.

＊通过标识符之间赋值的这种方式，一个程序可以给一个已经存在的对象建立一个别名。继续沿用我们之前的例子，图 1.2 描述了后置赋值的结果，original = temperature.（言外之意就是，两个标识符都指向了内存中同一块区域。图略。）

Once an alias has been established, either name can be used to access the underlying object. If that object supports behaviors that affect its state, changes enacted through one alias will be apparent when using the

other alias (because they refer to the same object). However, if one of the *names* is reassigned to a new value using a subsequent assignment statement, that does not affect the aliased object, rather it breaks the alias. Continuing with our concrete example, we consider the command:

＊当一个对象的别名建立之后，原来的名字与别名都可以访问这个对象。如果对象包含能够影响他自身状态的那种行为，那么通过引用一个别名而产生的变化自然也会作用在另外的别名上，因为它们指向的都是同一个对象。然而，如果这些别名中的一个被后续赋值语句赋予了新的数值，那么这就不会影响其他的别名对象，而是中断了这个别名机制（而变成了一个新的对象）。还是举我们之前的实例，思考一下这个命令：

temperature = temperature + 5.0

＊temperature = temperature + 5.0

The execution of this command begins with the evaluation of the expression on the right-hand side of the = operator. That expression, temperature + 5.0, is evaluated based on the existing binding of the name temperature, and so the result has value 103.6, that is, 98.6 + 5.0. That result is stored as a new floating-point instance, and only then is the name on the left-hand side of the assignment statement, temperature, (re)assignment to the result. The subsequent configuration is diagrammed in Figure 1.3. Of particular note, this last command had no effect on the value of the existing float instance that identifier original continues to reference.

＊这个语句的执行是从等号的右边开始的。temperature + 5.0 这个语句基于 temperature 所既有的数值，所以运算结果是 103.6，也就是 98.6 + 5.0. 这个结果被储存在了一个新的浮点型实例中，而这之后才是等号左边元素的赋值，即 temperature 被赋予了一个新的浮点型对象。图 1.3 给出了后续的结构。值得注意的是，第二种命令对 original 所指向的浮点型实例并不起作用。

体会：Python 是动态的语言，没有预先声明变量的类型。这一点刚从 C 语言转过来人可能感到很不适应，后来我才知道，Python 的内存管理机制与 C 语言已经完全不一样了，这是因为 C 语言作为静态语言的代表，是将变量固定在某一内存的区域中，任何数值的变化都在这块区域改变。但是 Python 不同，它本身是从 C 语言进行二次开发得到的一门新的解释性语言，它很好地借用了 C 语言中的指针的概念，我猜测在设计 Python 的过程中，C 语言中的 `malloc` 函数被广泛使用，从而使得变量的自增或者赋值基本上都会新开辟另外大小的内存片段，从而直接避开了声明数值类型的麻烦，而且旧的区域在赋值或者自增后，将会被 `free` 函数清空，从而可以被再次调用。

Exceptions are unexpected events that occur during the execution of a program. An exception might result from a logical error or an unanticipated situation. In Python, ***exceptions*** (also known as ***errors***) are objects that are ***raised*** (or ***thrown***) by code that encounters an unexpected circumstance. The Python interpreter can also raise an exception should it encounter an unexcepted condition, like running out of memory. A raised error may be caught by a surrounding context that "handles" the exception in an appropriate fashion. If uncaught, an exception causes the interpreter to stop executing the program to report an appropriate message to the console.

＊我们把程序执行过程中发生的未预料到的事件称为“异常”。能导致异常的原因有很多，比如逻辑错误或者未预料到的情形。在 Python 语言中，异常（即 C 语言中与之相同的 error）是以对象的形式、被发生了不可预知情形的代码抛出的。当遇到像内存溢出一样的情形时，Python 的解释器也能抛出异常。当一个错误被抛出，它可以被周围的上下文捕获并以恰当的方式处理。如果异常未能被捕获，那么这将会导致程序停止运行并且向控制台提交相关信息。

1.7 Exception Handling

＊1.7 节 对于异常的处理

1.7.2 Catching an Exception

＊1.7.2 节 捕获异常

One philosophy for managing exceptional cases is to "***look before you leap.***"

＊一种处理潜在的异常处理哲学就是提前观测。

A second philosophy, often embraced by Python programmers, is that "***it is easier to ask for forgiveness than it is to get permission.***"

＊另一种就是先处理，之后有异常就抛出。

1.8 Iterators and Generators

＊ 1.8 节 迭代器与生成器

In section 1.4.2, we introduced the for-loop syntax beginning as:

＊在 1.4.2 节中，我们介绍了 for 循环的语句：

for *element* **in** *iterable*:

＊**for** *element* **in** *iterable*:

and we noted that there are many types of objects in Python that qualify as being iterable. Basic container types, such as `list`, `tuple`, and `set`, qualify as iterable types. Furthermore, a string can produce an iteration of its characters, a dictionary can produce an iteration of its keys, and a file can produce an iteration of its lines. User-defined types may also support iteration. In Python, the mechanism for iteration is based upon the following conventions:

＊我们注意到，Python 中有许多类型的对象被认为是可迭代的。基本容器类型，如列表，元组和集合，都可以被定义为可迭代类型。此外，字符串可以产生其字符的迭代，字典可以产生其 keys 的迭代，并且文件可以产生其行的迭代。用户定义的类型也可以支持迭代。在 Python 中，迭代的机制基于以下约定

- An ***iterator*** is an object that manages an iteration through a series of values. If variable, i, identifies an iterator object, then each call to the build-in function, next(i), produce a subsequent element from the underlying series, with a StopIteration exception raised to indicate that there are no further elements.

  ＊迭代器是通过一系列值来实现对迭代对象的管理的。如果变量 i 代表一个迭代器对象，那么每次调用内建函数 next(i)都会从底层产生一个后续元素，指导抛出 StopIteration 异常来表示没有其他元素。

- An iterable is an object, obj, that produces an iterator via the syntax iter(obj)

  ＊可迭代的对象 obj，可以通过 iter(obj)语句生成一个迭代器对象。

By these definitions, an instance of a list is an iterable, but not itself an iterator. With data = [1, 2, 4, 8], it is not legal to call next(data). However, an iterator object can be produced with syntax, I = iter(data), and then each subsequent call to next(i) will return an element of that list. The for-loop syntax in Python simply automates this process, creating an iterator for the give iterable, and then repeatedly calling for the next element until catching the StopIteration exception.

＊通过这些约定可以看到，列表是可迭代的，但是列表本身并不是一个迭代器。定义列表 data = [1，2，4，8]，然后调用 next(data)是不合法的。然而，可以使用语法 I = iter(data)生成迭代器对象，然后每个后续调用 next(i)将返回该列表的元素。Python 中的 for 循环已经比较简单地将这个过程自动化了，先是为可迭代对象创建一个迭代器，然后重复调用下一个元素，直到抛出 StopIteration 异常。

More generally, it is possible to create multiple iterators based upon the same iterable object, with each iterator maintaining its own state of progress. However, iterators typically maintain their state with indirect reference back to the original collection of elements. For example, calling iter(data) on a list instance produces an instance of the list_iterator class. That iterator does not store its own copy of the list of elements. Instead, it maintains a current index into the original list, representing the next element to be reported. Therefore, if the contents of the original list are modified after the iterator is constructed, but before the iteration is complete, the iterator will be reporting the updated contents of the list.

＊更一般地，可以基于一个的可迭代对象而创建多个迭代器，并且每个迭代器都可以保持其自身的进度状态。 然而，迭代器通常将间接引用的状态保留在原始的元素集合中。 例如，在列表实例上调用 iter（数据）会生成 list_iterator 类的实例。 该迭代器不存储其自己的元素列表的副本。 相反，它将当前索引植入到原始列表中，借此表示要抛出的下一个元素。 因此，如果在构建迭代器之后、但又是在在迭代过程完成之前修改了原始列表的内容，那么迭代器将报告列表的更新内容。

Python also supports functions and classes that produce an implicit iterable series of values, that is, without constructing a data structure to store all of its values at once. For example, the call range(1000000) does not return a list of numbers; it returns a range object that is iterable. This object generates the million values one at a time, and only as needed. Such a ***lazy evaluation*** technique has great advantage. In the case of range, it allows a loop of the form, **for** j **in** range(1000000):, to execute without setting aside memory for storing one million values. Also, if such a loop were to be interrupted in some fashion, no time will have been spent computing unused values of the range.

＊

We see lazy evaluation used in many of Python's libraries. For example, the dictionary class supports methods keys(), values(), and items(), which respectively produce a "view" of all keys, values, or (key, value) pairs within a dictionary. None of these methods produces an explicit list of results. Instead, the views that are produced are iterable objects based upon the actual contents of the dictionary. An explicit list of values from such an iteration can be immediately constructed by calling the list class constructor with the iteration as a parameter. For example, the syntax list(range(1000)) produces a list with value from 0 to 999, while the syntax list(d.values())produces a list that has elements based upon the current values of dictionary d. We can similarly construct a tuple or set instance based upon a given iterable.

＊

END

六、参考文献

[1] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, *Data Structures and Algorithms in Python*