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**：高级语言面向对象编程实验 | **学号**：20151910042 | **上机实践日期**：2017-04-18 |
| **上机实践编号**：No.2 | **组号**： | **上机实践时间**：上午3、4节 |

# 一、实验目的

1. 熟悉Python面向对象编程，为数据结构与算法的学习奠定实验基础

2. 熟悉教材第二章的代码片段

# 二、实验内容

1. 熟悉Python面向对象技术：封装、继承、多态、设计模式、程序代码的组织结构等

2. 调试主讲教材第二章的Python程序

# 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | # 2.3.1 Example\_CreditCard Class  **class** **CreditCard:**  **def** \_\_init\_\_**(**self**,**customer**,**bank**,**acnt**,**limit**):**  self**.**\_customer **=** customer  self**.**\_bank **=** bank  self**.**\_account **=** acnt  self**.**\_limit **=** limit  self**.**\_balance **=** 0  **def** get\_customer**(**self**):**  **return** self**.**\_customer  **def** get\_bank**(**self**):**  **return** self**.**\_bank  **def** get\_account**(**self**):**  **return** self**.**\_account  **def** get\_limit**(**self**):**  **return** self**.**\_limit  **def** get\_balance**(**self**):**  **return** self**.**\_balance  **def** charge**(**self**,**price**):**  **if** price **+** self**.**\_balance **>** self**.**\_limit**:**  **return** **False**  **else:**  self**.**\_balance **+=** price  **return** **True**  **def** make\_payment**(**self**,**amount**):**  self**.**\_balance **-=** amount  **if** \_\_name\_\_ **==** '\_\_main\_\_'**:**  wallet **=** **[]**  wallet**.**append**(**CreditCard**(**'John Bowman'**,**'California Savings'**,**\  '5391 0375 9387 5309'**,**2500**))**  wallet**.**append**(**CreditCard**(**'John Bowman'**,**'California Fedoral'**,**\  '3485 0399 3395 1954'**,**3500**))**  wallet**.**append**(**CreditCard**(**'John Bowman'**,**'California Finance'**,**\  '5391 0375 9387 5309'**,**5000**))**  **for** val **in** range**(**1**,**17**):**  wallet**[**0**].**charge**(**val**)**  wallet**[**1**].**charge**(**2**\***val**)**  wallet**[**2**].**charge**(**3**\***val**)**    **for** c **in** range**(**3**):**  **print(**'Customer ='**,**wallet**[**c**].**get\_customer**())**  **print(**'Bank ='**,**wallet**[**c**].**get\_bank**())**  **print(**'Account ='**,**wallet**[**c**].**get\_account**())**  **print(**'Limit ='**,**wallet**[**c**].**get\_limit**())**  **print(**'Balance ='**,**wallet**[**c**].**get\_balance**())**  **while** wallet**[**c**].**get\_balance**()** **>** 100**:**  wallet**[**c**].**make\_payment**(**100**)**  **print(**'New balance ='**,**wallet**[**c**].**get\_balance**())**  **print()** |

程序代码 1

运行结果：



运行结果 1

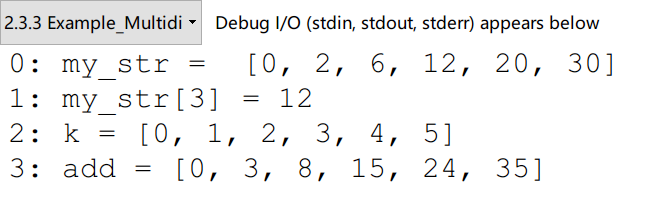
2.

程序代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | # 2.3.3 Example\_Multidimensional Vector Class  **class** **Vector:**  **def** \_\_init\_\_**(**self**,**d**):**  self**.**\_coords **=** **[**0**]** **\***d  **def** \_\_len\_**(**self**):**  **return** len**(**self**.**\_coords**)**  **def** \_\_getitem\_\_**(**self**,**j**):**  **return** self**.**\_coords**[**j**]**  **def** \_\_setitem\_\_**(**self**,**j**,**val**):**  self**.**\_coords**[**j**]** **=** val  **def** \_\_add\_\_**(**self**,**other**):**  **if** len**(**self**.**\_coords**)** **!=** len**(**other**.**\_coords**):**  **raise** ValueError**(**'dimensions must agree'**)**  result **=** Vector**(**len**(**self**.**\_coords**))**  **for** j **in** range**(**len**(**self**.**\_coords**)):**  result**[**j**]** **=** self**[**j**]** **+** other**[**j**]**  **return** result  **def** \_\_eq\_\_**(**self**,**other**):**  **return** self**.**\_coords **==** other**.**\_coords  **def** \_\_ne\_\_**(**self**,**other**):**  **return** **not** self **==** other  **def** \_\_str\_\_**(**self**):**  **return** '<' **+** str**(**self**.**\_coords**)[**1**:-**1**]** **+** '>'  my\_str **=** Vector**(**6**)**  **for** i **in** range**(**6**):**  my\_str**[**i**]** **=** i **\*** **(**i **+** 1**)**  **print(**'0: my\_str = '**,**my\_str**.**\_coords**)**  **print(**'1: my\_str[3] ='**,**my\_str**.**\_\_getitem\_\_**(**3**))**  k **=** Vector**(**6**)**  **for** i **in** range**(**6**):**  k**[**i**]** **+=** i  **print(**'2: k ='**,**k**.**\_coords**)**  k**.**\_coords**.**\_\_add\_\_**(**my\_str**.**\_coords**)**  **print(**'3: add ='**,**k**.**\_\_add\_\_**(**my\_str**).**\_coords**)** |

程序代码 2

运行结果：



运行结果 2

分析：

所谓类的意思，与C语言的结构基本一致，结构中包含的元素是打点引用，而类中的元素，不仅可以打点引用member，还可以引用方法。如是而已。在程序代码 2中，有一个比较明显的错误，导致了函数调用会出现问题，在\_\_add\_\_()方法下，它的参数应该是Vector类的，但是直接调用会出错，因为Vector类并没有len方法，只有Vector.\_coords才有len方法。所以上面的程序进行了修改。

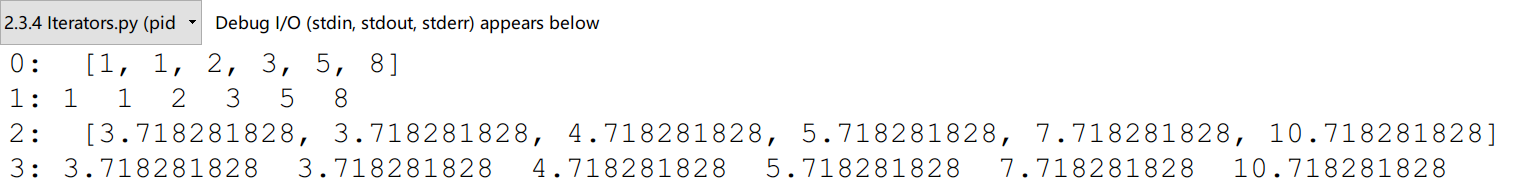
3.

程序代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | # 2.3.4 Iterators  **class** **SequenceIterator:**  **def** \_\_init\_\_**(**self**,**sequence**):**  self**.**\_seq **=** sequence  self**.**\_k **=** **-**1  **def** \_\_next\_\_**(**self**):**  self**.**\_k **+=** 1  **if** self**.**\_k **<** len**(**self**.**\_seq**):**  **return(**self**.**\_seq**[**self**.**\_k**])**  **else:**  **raise** StopIteration**(**'End'**)**  **def** \_\_iter\_\_**(**self**):**  **return** self  seq **=** **[**1**,**1**,**2**,**3**,**5**,**8**]**  **print(**'0: '**,**seq**)**  s **=** SequenceIterator**(**seq**)**  **print(**'1: '**,**end**=**''**)**  **for** i **in** range**(**6**):**  **print(**s**.**\_\_next\_\_**(),**' '**,**end**=**''**)**  **print(**''**)**  s**.**\_k **=** **-**1  **for** i **in** range**(**6**):**  s**.**\_seq**[**i**]** **+=** 2.718281828  **print(**'2: '**,**seq**)**  **print(**'3: '**,**end**=**''**)**  s**.**\_k **=** **-**1  **for** i **in** range**(**6**):**  **print(**s**.**\_\_next\_\_**(),**' '**,**end**=**''**)** |

程序代码 3

运行结果：



运行结果 3

分析：

迭代器生成的对象，是惰性的。我们不知道什么时候结束，也不能跳跃性取值，只能根据需要逐个读取。迭代器给出了一个遍历一个对象的方法。而且在外界看来有通用性，那就是\_\_next\_\_()方法。通过\_\_iter\_\_方法，可以将可迭代对象变成一个迭代器，进行遍历。

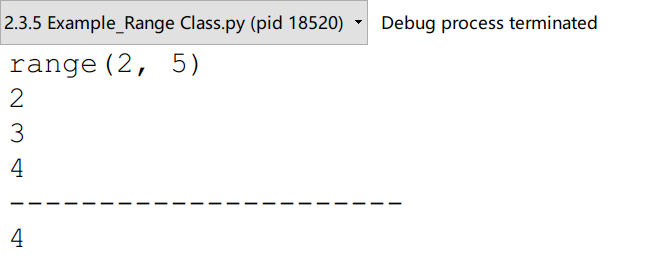
4.

程序代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | # 2.3.5 Example\_Range Class  **class** **Range:**  **def** \_\_init\_\_**(**self**,**start**,**stop**=None,**step**=**1**):**  **if** step **==** 0**:**  **raise** ValueError**(**'step cannot be 0'**)**  **if** stop **is** **None:**  start**,**stop **=** 0**,**start  self**.**\_length **=** max**(**0**,(**stop **-** start **+** step **-** 1**)//**step**)**  self**.**\_start **=** start  self**.**\_step **=** step  **def** \_\_len\_\_**(**self**):**  **return** self**.**\_length  **def** \_\_getitem\_\_**(**self**,**k**):**  **if** k **<** 0**:**  k **+=** len**(**self**)**  **if** **not** 0 **<=** k **<** self**.**\_length**:**  **raise** IndexError**(**'index out of range'**)**  **return** self**.**\_start **+** k **\*** self**.**\_step  **print(**range**(**2**,**5**))**  dd **=** Range**(**2**,**5**,**1**)**  **for** i **in** Range**(**0**,**3**):**  **print(**dd**[**i**])**  **print(**'----------------------'**)**  **print(**dd**.**\_\_getitem\_\_**(**2**))** |

程序代码 4

运行结果：



运行结果 4

分析：

这个例子重点观测参数的调用。这个class对于参数的要求不高，两个或者三个都可以按照设计思维进行解读。

5.

程序代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | # 2.4.1 Extending the CreditCard Class  **class** **CreditCard:**  **def** \_\_init\_\_**(**self**,**customer**,**bank**,**acnt**,**limit**):**  self**.**\_customer **=** customer  self**.**\_bank **=** bank  self**.**\_account **=** acnt  self**.**\_limit **=** limit  self**.**\_balance **=** 0  **def** get\_customer**(**self**):**  **return** self**.**\_customer  **def** get\_bank**(**self**):**  **return** self**.**\_bank  **def** get\_account**(**self**):**  **return** self**.**\_account  **def** get\_limit**(**self**):**  **return** self**.**\_limit  **def** get\_balance**(**self**):**  **return** self**.**\_balance  **def** charge**(**self**,**price**):**  **if** price **+** self**.**\_balance **>** self**.**\_limit**:**  **return** **False**  **else:**  self**.**\_balance **+=** price  **return** **True**  **def** make\_payment**(**self**,**amount**):**  self**.**\_balance **-=** amount  **class** **PredatoryCreditCard(**CreditCard**):**  **def** \_\_init\_\_**(**self**,**customeer**,**bank**,**acnt**,**limit**,**apr**):**  super**().**\_\_init\_\_**(**customer**,**bank**,**acnt**,**limit**)**  self**.**\_apr **=** apr  **def** charge**(**self**,**price**):**  success **=** super**().**charge**(**price**)**  **if** **not** success**:**  self**.**\_balance **+=** 5  **return** success  **def** process\_month**(**self**):**  **if** self**.**\_balance **>** 0**:**  monthly\_factor **=** pow**(**1 **+** self**.**\_apr**,**1**/**12**)**  self**.**\_balance **\*=** monthly\_factor |

程序代码 5

分析：

这是一个继承类的实例。不考虑面向对象的抽象观点，那么重点就是继承的实现。在调用super函数时，会对子类进行初始化，而且是照搬父类的初始化。

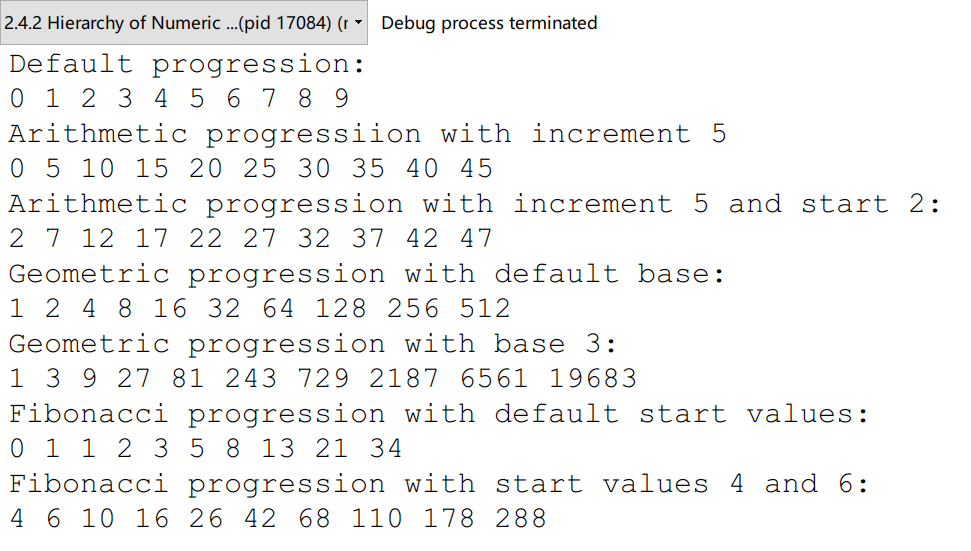
6.

程序代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | # 2.4.2 Hierarchy of Numeric Progressions  **class** **Progression:**  **def** \_\_init\_\_**(**self**,**start**=**0**):**  self**.**\_current **=** start  **def** \_advance**(**self**):**  self**.**\_current **+=** 1  **def** \_\_next\_\_**(**self**):**  **if** self**.**\_current **is** **None:**  **raise** StopIteration**()**  **else:**  answer **=** self**.**\_current  self**.**\_advance**()**  **return** answer  **def** \_\_iter\_\_**(**self**):**  **return** self  **def** print\_progression**(**self**,**n**):**  **print(**' '**.**join**(**str**(**next**(**self**))** **for** j **in** range**(**n**)))**  **class** **ArithmeticProgression(**Progression**):**  **def** \_\_init\_\_**(**self**,**increment**=**1**,**start**=**0**):**  super**().**\_\_init\_\_**(**start**)**  self**.**\_increment **=** increment  **def** \_advance**(**self**):**  self**.**\_current **+=** self**.**\_increment  **class** **GeometricProgression(**Progression**):**  **def** \_\_init\_\_**(**self**,**base**=**2**,**start**=**1**):**  super**().**\_\_init\_\_**(**start**)**  self**.**\_base **=** base  **def** \_advance**(**self**):**  self**.**\_current **\*=** self**.**\_base  **class** **FibonacciProgression(**Progression**):**  **def** \_\_init\_\_**(**self**,**first**=**0**,**second**=**1**):**  super**().**\_\_init\_\_**(**first**)**  self**.**\_prev **=** second **-** first  **def** \_advance**(**self**):**  self**.**\_prev**,**self**.**\_current **=** \  self**.**\_current**,**self**.**\_prev **+** self**.**\_current  **if** \_\_name\_\_ **==** '\_\_main\_\_'**:**  **print(**'Default progression:'**)**  Progression**().**print\_progression**(**10**)**    **print(**'Arithmetic progressiion with increment 5'**)**  ArithmeticProgression**(**5**).**print\_progression**(**10**)**    **print(**'Arithmetic progression with increment 5 and start 2:'**)**  ArithmeticProgression**(**5**,**2**).**print\_progression**(**10**)**  **print(**'Geometric progression with default base:'**)**  GeometricProgression**().**print\_progression**(**10**)**  **print(**'Geometric progression with base 3:'**)**  GeometricProgression**(**3**).**print\_progression**(**10**)**  **print(**'Fibonacci progression with default start values:'**)**  FibonacciProgression**().**print\_progression**(**10**)**  **print(**'Fibonacci progression with start values 4 and 6:'**)**  FibonacciProgression**(**4**,**6**).**print\_progression**(**10**)** |

程序代码 6

运行结果：



运行结果 5

分析：

# 五、实验体会

Translation:

**Chapter 2 Object-Oriented Programming**

＊第二章 面向对象编程

2.1 Goals, Principles, and Patterns

＊2.1节 目标、原则与模式

As the name implies, the main “actors” in the Object-Oriented paradigm are called ***objects***. Each object is an ***instance*** of a ***class***. Each class presents to the outside world a concise and consistent view of the instance of this class, without going into too much unnecessary detail or giving others access to the inner working of the objects. The class definition typically specifies ***instance variables***, also known as ***data members***, that the object contains, as well as the ***methods***, also known as ***member functions***, that the object can execute. This view of computing is intended to fulfill several goals and incorporate several design principles, which we discuss in this chapter.

＊正如题目所提及的那样，面向对象实例中的主角是对象。每个对象都是相应类的实例。在外界看来，这个类的实例在彼此之间都是简洁而一致的，并没有产生太多不必要的细节，也没有让其他人得以访问对象的内部工作空间。类的定义明确指定了实例变量以及方法，其中前者又称为数据成员，后者又称数据函数，数据函数可以被该对象调用，数据成员被对象包含。这种计算方式旨在实现几个目标以及统一若干设计原则，我们将在这一章中详细讨论。

2.1.1 Object-Oriented Design Goals

＊2.1.1节 面向对象设计的目标

Software implementations should achieve robustness, adaptability, and reusability.

＊软件开发应该追求健壮性、适应性以及复用性。

**Robustness**

＊健壮性

Every good programmer wants to develop software that is correct, which means that a program produces the right output for all the anticipated inputs in the program’s application. In addition, we want software to be robust, that is, capable of handling unexpected inputs that are not explicitly defined for its application. For example, if a program is expecting a positive integer (perhaps representing the price of an item) and instead is given a negative integer, then the program should be able recover gracefully from this error. More importantly, in life-critical applications, where a software error can lead to injury or loss life, software that is not robust could be deadly. This point was driven home in the late 1980s in accidents involving Therac-25, a radiation-therapy machine, which severely overdosed six patients between 1985 and 1987, some of whom died from complications resulting from their overdose. All six accidents were traced to software errors.

＊每一个优秀的程序员在开发中都想看到的是，在任何合理的输入情况下，程序都能够运行而且得到正确的输出结果。但是除此之外，我们还希望软件可以变得健壮，所谓的健壮，指的就是程序能处理未遇到过的异常输入。例如，程序需要输入一个正整数，比方说是需要某件物品的单价，但是用户却输入了一个负整数，这时候程序应该能在这个异常输入下进行适度的处理。更重要的是，在一些性命攸关的应用程序中，一个软件上面的错误就有可能导致病患的受伤甚至死亡。在此领域中，缺乏健壮性的软件将会是致命的。在20世纪八十年代的1985 - 1987年之间，型号为Therac-25的放疗机器，因为软件问题，导致了6个人接受了过量的放射，造成了严重的医疗事故。而正是这次事故将软件的健壮性这个概念提高到一个新层面。而这6个人中的一些也在后来的时间里因为接受了过度放疗而死亡。

**Adaptability**

＊适应性

Modern software applications, such as Web browsers and Internet search engines, typically involve large programs that are used for many years. Software, therefore, needs to be able to evolve over time in response to changing conditions in its environment. Thus, another important goal of quality software is that it achieves ***adaptability*** (also called ***evolvability***). Related to this concept is ***portability***, which is the ability of software to run with minimal change on different hardware and operating system platforms. An advantage of writing software in Python is the portability provided by the language itself.

＊像网页浏览器以及互联网搜索引擎这种现代化的软件，基本上都包含那种需要被使用好多年大型程序。因此，为了应对不断变化的软件应用环境，软件需要与时俱进。因此，一个关于高质量软件设计目标被提出，这就是软件的适应性，或叫做进化性。与之相关的一个概念就是可移植性，也就是说软件可以在稍加改动的基础之上，就能在不同的硬件平台与不同的操作系统上运行。而用Python开发的一个优势就是Python自身支持可移植性。

**Reusability**

＊复用性

Going hand in hand with adaptability is the desire that software be reusable, that is, the same code should be usable as a component of different systems in various applications. Developing quality software can be an expensive enterprise, and its cost can be offset somewhat if the software is designed in a way that makes it easily reusable in future applications. Such reuse should be done with care, however, for one of the major sources of software errors in the Therac-25 came from inappropriate reuse of Therac-20 software (which was not object-oriented and not designed for the hardware platform used with the Therac-25).

＊相同的代码可以当作一个组件被用在不同应用场景下的不同系统上，而这个目标，即软件的复用性，与软件的适应性一样被人们狂热地追求。开发高质量的软件会是一个代价花费巨大的事业，但是如果能采取一种设计方式，使得软件能在将来的应用场景中继续得以使用，那么所花费的代价有可能会降低一些。然而，这种复用需要小心进行，正如之前那个Therac-25的例子，它的软件问题主要原因就来自于对Therac-20软件做了不适宜的复用。（Therac-20软件设计并不是面向对象的，而且也不是针对Therac-25的硬件平台所设计的。）

2.1.2 Object-Oriented Design Principles

＊2.1.2节 面向对象设计原则

Object-Oriented Design Principles: Chief among the principles of the object-oriented approach, which are intended to facilitate the goals outlined above, are the following:

* Modularity
* Abstraction
* Encapsulation

＊面向对象设计原则：面向对象设计方法的主要原则如下，他们是针对上面的设计目标而被设定的：

* 模块化
* 抽象化
* 封装

**Modularity**

＊模块化

Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized. Modularity refers to an organizing principle in which different components of a software system are divided into separate units.

＊现代软件系统为了能让整个系统良好运转，通常都要包含多个不同的而且彼此之间可以准确交互的组件。为了让组件之间的交互变得可靠，我们需要好好组织这些组件。模块化指的就是一种把软件系统的不同组件分割成独立的单元，从而实现这种需求的设计原则。

As a real-world analogy, a house or apartment can be viewed as consisting of several interacting units: electrical, heating, and cooling, plumbing, and structural. Rather than viewing these systems as one giant jumble of wires, bents, pipes, and boards, the organized architect designing a house or apartment will view them as separate modules that interact in well-defined ways. In so doing, he or she is using modularity to bring a clarity of thought that provides a natural way of organizing functions into distinct manageable units.

＊房子或公寓作为一种对于自然世界的模仿，可以被视为一种包含了多个交互式单元的集合体，比如电力单元，供热和制冷单元，管道系统，还有房屋框架。一般人可能觉得这些子系统看起来就像是一些线材、框架、管子以及木板等混乱的东西堆在一起，但是从有组织的建筑设计观点来看，它们都是独立的单元，并且彼此之间可以通过事先设定好的方法进行良好交互。在这样做的时候，我们就可以使用模块化的思维，用极其自然的方式将一些功能组织到不同的而易于管理的单元中。

In like manner, using modularity in a software system can also provide a powerful organizing framework that brings clarity to an implementation. In Python, we have already seen that a nodule is a collection of closely related functions and classes that are defined together in a single file of source code. Python’s standard libraries include, for example, the math module, which provides definitions for key mathematical constants and functions, and the os module, which provides support for interacting with the operating system.

＊通过同样的方式，在软件系统中使用模块化思维也能带来强大的组织框架，有了这个框架，构想实现就变得很明确了。在Python语言里面，模块指的就是一个单独的源代码文件，里面写有一些彼此之间相关性很强的函数和类。Python的标准库中有重要的数学函数与常量的math模块，提供与系统交互的os模块。

The use of modularity helps support the goals listed in Section 2.1.1. Robustness is greatly increased because it is easier to test and debug separate components before they are integrated into a lather software system. Furthermore, bugs that persist in a complete system might be traced to a particular component, which can be fixed in relative isolation. The structure imposed by modularity also helps enable software reusability. If software modules are written in a general way, the modules can be reused when related need arises in other contexts. This is particularly relevant in a study of data structures, which can typically be designed with sufficient abstraction and generality to be reused in many applications.

＊模块化设计原则的用处就是能够实现2.1.1节中列举的那些目标。采用了模块化设计原则之后，程序的健壮性就变得更强了，因为在上线之前的测试以及调试对于单独的模块而言十分简单。此外，完整系统中出现的bug可以被定向到确切的模块，这样一来我们就可以将他相对屏蔽掉从而修复问题。模块化设计所带来的结构也能够增加软件的复用性。如果软件的某些模块是通过通用的方式写成的，那么在其他相近的地方，我们就可以重新使用这些模块。这在数据结构的学习中尤为重要，因为数据结构通常就是被设计得具有足够的抽象性以及通用性，从而可以在很多应用中得以复用。

Abstraction

＊抽象

The notion of ***abstraction*** is to distill a complicated system down to its most fundamental parts. Typically, describing the parts of a system involves naming them and explaining their functionality. Applying the abstraction paradigm to the design of data structures gives rise to ***abstract data types*** (ADTs). An ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations. An ADT specifies ***what*** each operation does, but not ***how*** it does it. We will typically refer to the collective set of behaviors supported by an ADT as its public interface.

＊

As a programming language, Python provides a great deal of latitude in regard to the specification of an interface. Python has a tradition of treating abstractions implicitly using a mechanism known as duck typing. As an interpreted and dynamically typed language, there is no “compile time” checking of data types in Python, and no formal requirement for declarations of abstract base classes. Instead programmers assume that an object supports a set of known behaviors, with the interpreter raising a run-time error if those assumptions fail. The description of this as “duck typing”, comes from an adage attributed to poet James Whitcomb Riley, stating that “when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”

＊

More formally, Python supports abstract data types using a mechanism known as an abstract base class (ABC). An abstract base class cannot be instantiated (i.e., you cannot directly create an instance of that class), but it defines one or more common methods that all implementations of the abstraction must have. An ABC is realized by one or more concrete classes that inherit from the abstract base class while providing implementations for those method declared by the ANCs, although we omit such declarations for simplicity. We will make use of several existing abstract base classes coming from Python’s collections module, which includes definitions for several common data structure ADTs, and concrete implementations of some of those abstractions.

＊

Encapsulation

＊封装

Another important principle of object-oriented design is encapsulation. Different components of a software system should not reveal the internal details of their respective implementations. One of the main advantage of encapsulation is that it gives one programmers freedom to implement the details of a component, without concern that other programmers will be writing code that intricately depends on those internal decisions. The only constraint on the programmer of a component is to maintain the public interface for the component, as other programmers will be writing code that depends on that interface. Encapsulation yields robustness and adaptability, for it allows the implementation details of parts of a program to change without adversely affecting other parts, thereby making it easier to fix bugs or add new functionality with relatively local changes to a component.

＊

Throughout this book, we will adhere to the principle of encapsulation, making clear which aspects of a data structure are assumed to be public and which are assumed to be internal details. With that said, Python provides only loose support for the encapsulation. By convention, names of members of a class (both data members and member functions) that start with a single underscore character (e.g., \_secret) are assumed to be nonpublic and should not be relied upon. Those conventions are reinforced by the intentional omission of those members from automatically generated documentation.

＊

2.3.4 Iterators

＊2.3.4节 迭代器

Iteration is an important concept in the design of data structures. We introduced Python’s mechanism for iteration in section 1.8. In short, an ***iterator*** for a collection provides one key behavior: It supports a special method named \_\_next\_\_ that returns the next element of the collection, if any, or raises a StopIteration exception to indicate that there are no further elements.

＊迭代是数据结构设计中的一个很重要的概念。在1.8节中我们介绍了Python下的迭代机制。简而言之，一个迭代器提供了一个重要的行为：它支持一种叫做\_\_next\_\_特殊的方法，这个方法返回序列中的下一个元素，如果不存在下一个了，那么就抛出一个停止迭代的异常，借此表示往下已经没有其余元素了。

Fortunately, it is rare to have to directly implement an iterator class. Our preferred approach is the use of the ***generator*** syntax (also described in section 1.8), which automatically produces an iterator of yielded values.

＊幸运的是我们并不需要自己设计这样的迭代器。我们更加喜爱的方法就是利用生成器语句（早在1.8节就介绍过了）自动化地产生数值的迭代器。

Python also helps by providing an automatic iterator implementation for any class that defines both \_\_len\_\_ and \_\_getitem\_\_. To provide an instructive example of a low-level iterator, Code Fragment 2.5 demonstrates just such an iterator class that works on any collection that supports both \_\_len\_\_ and \_\_getitem\_\_. This class can be instantiated as SequenceIterator(data). It operates by keeping an internal reference to the data sequence, as well as a current index into the sequence. Each time \_\_next\_\_ is called, the index is incremented, until reaching the end of the sequence.

＊Python为每一个定义了\_\_len\_\_与\_\_getitem\_\_这两种方法的类都提供了一种自动化的迭代器。为了提供一个低级的、具有指导性的迭代器例子，如下代码模块2.5演示了一个迭代器类，它可以对任何支持\_\_len\_\_与\_\_getitem\_\_方法的元素集合提供迭代支持。这个类可以被函数性地调用，就像是SequenceIterator(data)这样。它通过记录数据序列的内部引用来进行操作（比如说记录当前的下标索引）。每次调用\_\_next\_\_方法，这个索引都会增长，直到到达序列的尾部。

2.4.3 Abstract Base Class

＊2.4.3节 抽象基类

When defining a group of classes as part of an inheritance hierarchy, one technique for avoiding repetition of code is to design a base class with common functionality that can be inherited by other classes that need it. As an example, the hierarchy from Section 2.4.2 includes a Progression class, which serves as a base class for three distinct subclasses: ArithmeticProgression, GeometricPrograssion, and FibonacciProgression. Although it is possible to create an instance of the Progression base class, there is little value in doing so because its behavior is simply a special case of an ArithmeticProgression with increment 1. The real purpose of the Progression class was to centralize the implementations of behaviors that other progressions needed, thereby streamlining the code that is related to those subclasses.

＊

In classical object-oriented terminology, we say a class is an ***abstract base class*** if its only purpose is to serve as a base class through inheritance. More formally, an abstract base class is one that cannot be directly instantiated, while a concrete class is one that can be instantiated. By this definition, out Progression class is technically concrete, although we essentially designed it as an abstract base class.

＊在经典的面向对象术语中，只有一个类的存在意义仅仅是作为一个被继承的基类时，我们才称这个类是抽象的基类。或者更加正式一点，一个抽象的基类就是一个不能直接被实例化的类，而与之相对的具体类是可以被实例化的。通过这个定义来看，Progression类是一个具体类，尽管我们在设计过程中是把它视为一个抽象基本类来进行的。

# 六、参考文献

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