《人工智能》课程系列

Part I: Python 程序设计基础*

武汉纺织大学数学与计算机学院

杜小勤

2018/06/13

Contents

*本系列文档属于讲义性质，仅用于学习目的。Last updated on: September 14, 2019。

1 Python 之父

吉多·范·罗苏姆（Guido van Rossum，1956 年 1 月 31 日—，图1-1），荷兰计算机程序员，为 Python 程序设计语言的最初设计者及主要架构师。在 Python 社区，吉多·范·罗苏姆被人们称为是“仁慈的独裁者”（Benevolent Dictator For Life，BDFL），意思是他仍然关注 Python 的开发进程，并在必要的时刻做出决定。2002 年，在比利时布鲁塞尔举办的自由及开源软件开发者欧洲会议上，吉多·范·罗苏姆获得了由自由软件基金会颁发的 2001 年自由软件进步奖。2003 年五月，吉多获得了荷兰 UNIX 用户小组奖。2006 年，他被美国计算机协会（ACM）认定为著名工程师。

2005 年 12 月，吉多·范·罗苏姆加入 Google。他用 Python 语言为 Google 写了面向网页的代码浏览工具 Mondrian，之后又开发了 Rietveld。在那里他把一半的时间用来维护 Python 的开发。

2012 年 12 月 7 日，云存储公司 Dropbox 宣布吉多·范·罗苏姆加入该公司。

关于 Python 的起源，吉多·范·罗苏姆在 1996 年写到：

图 1-1: Guido van Rossum

六年前，在 1989 年 12 月，我在寻找一门“课余”编程项目来打发圣诞节前后的时间。我的办公室会关门，但我有一台家用电脑，而且没有太多其它东西。我决定为当时我正构思的一个新的脚本语言写一个解释器，它是 ABC 语言的后代，对 UNIX / C 程序员会有吸引力。作为一个略微有些无关想法的人，和一个蒙提·派森的飞行马戏团（Monty Python's Flying Circus）的狂热爱好者，我选择了 Python 作为项目的名称。

在 2000 年他写到：

Python 的前辈，ABC 语言，受到了 SETL 的启发 - 在完成最终设计之前，Lambert Meertens 与纽约大学的 SETL 小组相处了一年的时间。

“Computer Programming for Everybody”是吉多·范·罗苏姆的重要信念。1999 年，吉多·范·罗苏姆向 DARPA 提交了一项名为“Computer Programming for Everybody”的资金申请，并随后说明了他对 Python 的目标：

- 一门简单直观的语言并与主要的竞争者一样强大；
- 开源，以便任何人都可以为它做贡献；
- 代码像纯英语那样容易理解；
- 适用于短期开发的日常任务；

这些想法中的一些已经成为现实。Python 已经成为一门流行的编程语言，尤其是在互联网与人工智能空前发展的大环境下。

实际上，Python 之所以如此成功，与其作者的开放理念密不可分。Python 本身是从 ABC 语言发展起来的，就吉多本人看来，ABC 虽然非常优美和强大，但并没有取得成功，究其原因，吉多认为是非开放造成的。吉多决心在 Python 中避免这一错误，并取得了非常好的效果，完美地结合了 C 和其他一些语言的优点。

另外，吉多·范·罗苏姆还专门为一本名为《Python Programming: An Introduction to Computer Science》的书写了序言，该书的作者是 John Zelle。

2 Python 语言

2.1 Python 的特点

Python，是一种广泛使用的高级编程语言，属于通用型编程语言，第一版发布于 1991 年。可以把它看作是一种改良的 LISP 语言，例如，加入了诸如面向对象等其它一些编程语言所具备的优点。

作为一种解释型语言，Python 的设计哲学强调代码的可读性和简洁的语法，尤其是使用空格缩进的方式来划分代码块，而没有使用传统的大括号或者关键词。相比于 C++ 或 Java，Python 让开发者能够用更少的代码来表达想法。不管是小型还是大型程序，该语言都试图让程序的结构清晰并且简单明了。

与 Scheme、Ruby、Perl、Tcl 等动态类型编程语言一样，Python 拥有动态类型系统和垃圾回收功能，能够自动管理内存使用，并且支持多种编程范式，包括面向对象、命令式、函数式和过程式编程。其本身拥有一个巨大而广泛的标准库。

Python 解释器本身几乎可以在所有的操作系统中运行。Python 的正式解释器 CPython 是用 C 语言编写的、是一个由社区驱动的自由软件，目前由 Python 软件基金会管理。

Python 2.0 于 2000 年 10 月 16 日发布，增加了完整的垃圾回收功能，并且支持 Unicode。同时，整个开发过程更加透明，社区对开发进度的影响逐渐扩大。Python 3.0 于 2008 年 12 月 3 日发布，此版不完全兼容之前的 Python 源代码。不过，很多新特性后来也被移植到了旧的 Python 2.6/2.7 版本中。

Python 是完全面向对象的语言。函数、模块、数字、字符串都是对象。并且完全支持继承、重载、派生、多重继承，有益于增强源代码的复用性。Python 支持重载运算符，因此 Python 也支持泛型设计。相对于 Lisp 这种传统的函数式编

程语言，Python 对函数式设计只提供了有限的支持。有两个标准库（functools, itertools）提供了与 Haskell 和 Standard ML 中类似的函数式程序设计工具。

Python 是一种“高阶动态编程语言”。虽然 Python 可以被粗略地分类为“脚本语言”，但实际上一些大规模软件开发项目例如 Zope、Mnet 及 BitTorrent，Google 也在广泛地使用它。Python 的支持者较喜欢称它为一种“高阶动态编程语言”，原因是简单的“脚本语言”泛指那些仅作简单程序设计任务的语言，如 shell script、VBScript 等，它们只能处理简单的任务，并不能与 Python 相提并论。

Python 也被称为是一种“胶水语言”。Python 本身被设计为可扩充的，并非所有的特性和功能都集成到语言核心。Python 提供了丰富的 API 和工具，以便程序员能够轻松地使用 C、C++、CPython 来编写扩充模块。Python 编译器本身也可以被集成到其它需要脚本语言的程序内。因此，有很多人把 Python 作为一种“胶水语言”使用。使用 Python 将其他语言编写的程序进行集成和封装。在 Google 内部的很多项目，例如，Google 应用服务引擎使用 C++ 编写性能要求极高的部分，然后用 Python 或 Java/Go 调用相应的模块。《Python 技术手册》的作者马特利（Alex Martelli）说，2004 年 Python 已在 Google 内部使用，Google 召募了许多 Python 高手，要求尽量使用 Python，在不得已时改用 C++——在操控硬件的场合使用 C++，而在快速开发时候使用 Python。

2.2 Python 的设计与开发哲学

Python 的设计哲学是“优雅、明确、简单”。Python 的开发哲学是“用一种方法，最好是只有一种方法来做一件事”。如果面临多种选择，Python 开发者一般会拒绝花俏的语法，而选择明确的、没有或者很少有歧义的语法。这些准则被称为“Python 格言”。在 Python 解释器内，运行“import this”可以获得完整的格言列表：

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
```

```
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

2.3 静态编译、动态编译与即时编译

Python 开发人员尽量避开不成熟或者不重要的优化。一些针对非重要部位的加快运行速度的补丁通常不会被合并到 Python 内。再加上因为 Python 属于动态类型语言，动态类型语言是在运行期间检查数据的类型，不得不保持描述变量值的实际类型标记，程序在每次操作变量时，需要执行数据依赖分支，而静态类型语言相对于动态类型语言，在声明变量时已经指定了数据类型和表示方法。根据这一原理，Python 相对于 C、C++ 等静态类型语言来说，运行速度较慢。不过，根据二八定律，大多数程序对速度要求不高。在某些对运行速度要求很高的情况，Python 设计师倾向于使用 JIT 技术，或者使用 C/C++ 语言改写这部分程序。目前可用的 JIT 技术是 PyPy[1]。

[1]即时编译（Just-in-time compilation），动态编译的一种形式，是一种提高程序运行效率的方法。通常，程序有两种运行方式：静态编译与动态解释。静态编译的程序，在执行前全部被翻译为机器码，而解释执行的程序，则是一句一句地边解释边运行。

2.4 Python 的应用场合

Python 程序设计语言可以应用于 Web、GUI、操作系统、人工智能与机器学习等领域的研究与开发工作。

Python 经常被用于 Web 开发。例如，通过 mod_wsgi 模块，Apache 可以运行用 Python 编写的 Web 程序。使用 Python 语言编写的 Gunicorn 作为 Web 服务器，也能够运行 Python 语言编写的 Web 程序。Python 定义了 WSGI 标准应用接口用于协调 Http 服务器与基于 Python 的 Web 程序之间的沟通。一些 Web 框架，如 Django、Pyramid、TurboGears、Tornado、web2py、Zope、Flask 等，可以让程序员轻松地开发和管理复杂的 Web 程序。

Python 对于各种网络协议的支持也很完善，因此经常被用于编写服务器软件和网络爬虫。第三方库 Twisted 支持异步线上编写程序和多数标准的网络协议（包含客户端和服务器），并且提供了多种工具，被广泛用于编写高性能的服务器软件。另外，第三方库 gevent 同样能够支持高性能高并发的网络开发。

YouTube、Google、Yahoo!、NASA 都在内部大量地使用 Python。OLPC 的作业系统 Sugar 项目的大多数软件都是使用 Python 编写。

Python 本身包含的 Tkinter 库能够支持简单的 GUI 开发。但是越来越多的 Python 程序员选择 wxPython 或者 PyQt 等 GUI 包来开发跨平台的桌面软件。使用它们开发的桌面软件运行速度快，与用户的桌面环境相契合。通过 PyInstaller 还能将程序发布为独立的安装程序包。

在很多操作系统里，Python 是标准的系统组件。大多数 Linux 发行版和 Mac OS X 都集成了 Python，可以在终端机下直接运行 Python。有一些 Linux 发行版的安装器使用 Python 语言编写，比如 Ubuntu 的 Ubiquity 安装器、Red Hat Linux 和 Fedora 的 Anaconda 安装器。在 RPM 系列 Linux 发行版中，有一些系统组件就是用 Python 编写的。Gentoo Linux 使用 Python 来编写它的 Portage 软件包管理系统。Python 标准库包含了多个调用作业系统功能的库。通过 pywin32 这

即时编译则混合了二者，一句一句地编译源代码，但是会将翻译过的代码缓存起来以降低性能损耗。相对于静态编译代码，即时编译的代码可以处理延迟绑定并增强安全性。

即时编译器有两种类型，一是字节码翻译，二是动态编译翻译。

微软的.NET Framework，还有绝大多数的 Java 实现，都依赖即时编译以提供高速的代码执行。Mozilla Firefox 使用的 JavaScript 引擎 SpiderMonkey 也用到了 JIT 的技术。Ruby 的第三方实现 Rubinius 和 Python 的第三方实现 PyPy 也都通过 JIT 来明显改善了解释器的性能。

个第三方软件包，Python 能够访问 Windows 的 COM 服务及其它 Windows API。使用 IronPython，Python 程序能够直接调用.Net Framework。

当前，对于人工智能与机器学习领域来讲，Python 被认为是最为热门的程序设计语言。NumPy、SciPy、Matplotlib 可以让 Python 程序员编写科学计算程序，Tensorflow、Keras、PyTorch 等热门深度学习工具包更是使用 Python 编写。很多游戏使用 C++ 编写图形显示等高性能模块，而使用 Python 或者 Lua 编写游戏的逻辑、服务器。相较于 Python，Lua 的功能更简单、体积更小；而 Python 则支持更多的特性和数据类型。很多游戏，如 EVE Online 使用 Python 来处理游戏中繁杂的逻辑。

2.5 标准库与第三方库

Python 拥有一个强大的标准库。Python 语言的核心只包含数字、字符串、列表、字典、文件等常见类型和函数，而由 Python 标准库提供了系统管理、网络通信、文本处理、数据库接口、图形系统、XML 处理等额外的功能。

Python 标准库的主要功能有：

1. 文本处理，包含文本格式化、正则表达式匹配、文本差异计算与合并、Unicode 支持，二进制数据处理等功能；
2. 文件处理，包含文件操作、创建临时文件、文件压缩与归档、操作配置文件等功能；
3. 操作系统功能，包含线程与进程支持、IO 复用、日期与时间处理、调用系统函数、日志（logging）等功能；
4. 网络通信，包含网络套接字，SSL 加密通信、异步网络通信等功能；
5. 网络协议，支持 HTTP，FTP，SMTP，POP，IMAP，NNTP，XMLRPC 等多种网络协议，并提供了编写网络服务器的框架；
6. W3C 格式支持，包含 HTML，SGML，XML 的处理；
7. 其它功能，包括国际化支持、数学运算、HASH、Tkinter 等；

Python 社区提供了大量的第三方模块，使用方式与标准库类似。它们的功能覆盖科学计算、Web 开发、数据库接口、图形系统等多个领域。第三方模块可

以使用 Python 或者 C 语言编写。SWIG、SIP 常用于将 C 语言编写的程序库转化为 Python 模块。Boost C++ Libraries 包含了一组库，Boost.Python，使得以 Python 或 C++ 编写的程序能互相调用。Python 常被用做其他语言与工具之间的“胶水”语言。

比较有名的第三方库，列举如下：

1. Web 框架：

   - Django: 开源 Web 开发框架，它鼓励快速开发, 并遵循 MVC 设计，开发周期短。
   - Flask: 轻量级的 Web 框架。
   - Pyramid: 轻量，同时有可以规模化的 Web 框架，Pylon projects 的一部分。
   - ActiveGrid: 企业级的 Web2.0 解决方案。
   - Karrigell: 简单的 Web 框架，自身包含了 Web 服务，py 脚本引擎和纯 python 的数据库 PyDBLite。
   - Tornado: 一个轻量级的 Web 框架，内置非阻塞式服务器，而且速度相当快
   - webpy: 一个小巧灵活的 Web 框架，虽然简单但是功能强大。
   - CherryPy: 基于 Python 的 Web 应用程序开发框架。
   - Pylons: 基于 Python 的一个极其高效和可靠的 Web 开发框架。
   - Zope: 开源的 Web 应用服务器。
   - TurboGears: 基于 Python 的 MVC 风格的 Web 应用程序框架。
   - Twisted: 流行的网络编程库，大型 Web 框架。
   - Quixote: Web 开发框架。

2. 人工智能与机器学习：

   - PIL: 基于 Python 的图像处理库，功能强大，对图形文件的格式支持广泛。目前已无维护，另一个第三方库 Pillow 实现了对 PIL 库的支持和维护。

- Matplotlib: 用 Python 实现的类 matlab 的第三方库，用以绘制一些高质量的数学二维图形。
- Pandas: 用于数据分析、数据建模、数据可视化的第三方库。
- SciPy: 基于 Python 的 matlab 实现，旨在实现 matlab 的所有功能。
- NumPy: 基于 Python 的科学计算第三方库，提供了矩阵，线性代数，傅立叶变换等等的解决方案。
- scikit-learn: 机器学习第三方库，实现许多知名的机器学习算法。
- TensorFlow: Google 开发维护的开源机器学习库。
- Keras: 基于 TensorFlow，Theano 与 CNTK 的高阶神经网络 API。

3. GUI:

- PyGtk: 基于 Python 的 GUI 程序开发 GTK+ 库。
- PyQt: 用于 Python 的 QT 开发库。
- WxPython: Python 下的 GUI 编程框架，与 MFC 的架构相似。

4. 其它:

- BeautifulSoup 基于 Python 的 HTML/XML 解析器，简单易用。
- gevent: python 的一个高性能并发框架, 使用了 epoll 事件监听、协程等机制将异步调用封装为同步调用。
- PyGame: 基于 Python 的多媒体开发和游戏软件开发模块。
- Py2exe: 将 python 脚本转换为 windows 上可以独立运行的可执行程序。
- Requests: 适合于人类使用的 HTTP 库，封装了许多繁琐的 HTTP 功能，极大地简化了 HTTP 请求所需要的代码量。

3 Python 的基本元素

3.1 标识符

标识符 (Identifier) 是程序代码的重要组成部分，用来表示一个对象的名称。对象可以是模块 (Modules)、模块内的函数 (Function)、变量 (Variable)。

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

图 3-2: Python 关键字

任何语言对标识符都有一个约定。Python 语言中，规定：每个标识符必须以一个字母 (Letter) 或下划线 (Underscore) 开头，后跟任意的字母、数字 (Digit) 或下划线的序列。

在 Python 中，标识符是大小写敏感的 (Case Sensitive)。

3.2 关键字

完整的 Python 关键字 (Keywords) 或保留字 (Reserved Words) 如图3-2所示。关键字组成了 Python 程序的关键结构，用来表示特定的语法结构，有特定的语义。不能使用关键字表示模块、函数或变量。

3.3 内建函数

完整的 Python 内建函数 (Built-in Functions) 如图3-3所示。内建函数执行 Python 程序的某个基本功能，是组成复杂程序或复杂函数的基本要素。

虽然可以使用内建函数的名称作为标识符使用，但是不鼓励这么做，会引起一些不必要的麻烦。

3.4 一个简单的 Python 程序

下面是一个简单的 Python 程序：

```
# convert.py
# A program to convert Celsius temps to Fahrenheit
# by: Susan Computewell
def main():
```

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

图 3-3: Python 内建函数

```
    celsius = eval(input("What is the Celsius temperature? "))
    fahrenheit = 9/5 * celsius + 32
    print("The temperature is", fahrenheit, "degrees Fahrenheit.")

main()
```

请试着找出该程序的变量名、关键字、内建函数、自定义函数名。

3.5 表达式

表达式是一个程序代码片段，用于计算新的数据值。最简单的表达式是字面量 (Literal)。

数值字面量用于表示一个特定的数值 (常量)，例如，在程序 convert.py 中，9、5、32 就是字面量——最简单的表达式。

字符串字面量 (String Literal) 属于文本数据常量，被称为是字符串 (String)，在 Python 中用双引号或单引号表示。注意，双引号或单引号并不是字符串的一部分。

试着在 Python Shell 下输入：

```
>>> 32
```

```
32
>>> "Hello"
'Hello'
>>> "32"
'32'
```

上例中，单引号表示该值是一个文本 (字符串)，而不是一个数值。

同样，一个普通的标识符变量也可以构成表达式：

```
>>> x = 5
>>> x
5
>>> print(x)
5
```

上例中，在 shell 下输入 x，表示对 x 进行 (最简单的) 求值 (Evaluation) 计算。表达式的求值计算就是要计算出该表达式的值。

复杂的表达式可以通过使用运算符将简单的表达式连接起来而构造。对于数值表达式来讲，常用的运算符有：+、-、*、/、**。例如，在 convert.py 中，有一个数值表达式：9/5 * celsius + 32。

Python 也为字符串提供了运算符。例如，可以使用 +:

```
>>> "Bat" + "man"
'Batman'
```

这种操作叫做连接 (Concatenation)，其结果是创建了一个新字符串，它把原来的 2 个字符串连接在了一起，在后面的讲义中，可以看到更多的字符串操作。

在 Python 中，可以使用内建函数 type 查看数据类型：

```
>>> type('abc')
<class 'str'>
>>> type(1)
<class 'int'>
```

3.6 输出语句

最常使用的输出函数是 print，它是 Python 的内建函数。其调用格式如下：

```
print(<expr>, <expr>, ..., <expr>)
print()
```

上述内容表明，print 可以接受 0、1、...、n 个参数，expr 表示表达式。

新建一个 Python 程序，输入下面的语句：

```
print(3+4)
print(3, 4, 3 + 4)
print()
print("The answer is", 3 + 4)
```

将会产生如下输出：

```
7
3 4 7

The answer is 7
```

注意，缺省情况下，print 函数将在各个输出值之间自动加入一个空格，并在末尾自动进行换行。

不过，我们可以使用关键字参数 (Keyword Parameter) 来修改 print 的行为：

```
print("The answer is", end = " ")
print(3 + 4)
```

将会得到如下输出结果：

```
The answer is 7
```

上例中，如果使用end = "\n"，则 print 在末尾将进行换行操作。

3.7 赋值语句

赋值语句是一种非常重要的语句，它将值与变量进行关联。

3.7.1 简单的赋值语句

基本的赋值语句具有如下形式：

```
<variable> = <expr>
```

其中，<variable> 表示一个变量，<expr> 表示一个表达式。该语句的语义是：对右端的表达式求值，并将该值与左端的变量进行关联。例如：

```
x = 3.9 * x * (1 - x)
fahrenheit = 9 / 5 * celsius + 32
x = 5
x = x + 1
```

3.7.2 input 赋值语句

input 属于内建函数，它的作用是将用户输入的数据保存到变量中。具体的 input 语句形式依赖于所要保存的变量类型。

对于文本输入，其语法形式是：

```
<variable> = input(<prompt>)
```

<prompt> 是一个字符串表达式，用于提示用户输入。执行 input 语句时，Python 将 <prompt> 的内容显示在屏幕上，然后等待用户输入一些文本和 <Enter> 键，不管用户输入什么类型的数据，这些数据都将作为字符串存储（除非进行求值转换，后面将介绍）。例如：

```
>>> name = input("Enter your name: ")
Enter your name: John Yaya
>>> name
'John Yaya'
```

对于数值输入，其语法形式稍微复杂：

```
<variable> = eval(input(<prompt>))
```

其中，eval 是另一个 Python 内建函数，执行求值 (evaluate) 功能。例如，在前面的例子中，如果用户输入'32'(字符串类型)，那么 eval 会将它转换成数值类型 32：

```
>>> celsius = eval(input("What is the Celsius temperature? "))
What is the Celsius temperature? 32
>>> celsius
32
```

因此，如果你需要数值，那么在调用 input 函数后还需调用 eval 函数，否则，将得到字符串类型的数据。

另外，input 函数的提示字符串末尾最好添加一个空格，这样可以使得提示字符串与用户输入的数据有一个清晰的界限，便于阅读。

正如前面所述，一个数值数据就是一个简单的表达式，实际上，input 函数也可以接收用户输入表达式：

```
>>> result = eval(input("Enter an expression: "))
Enter an expression: 3 + 4 * 5
>>> result
23
```

3.7.3 同时赋值

同时赋值 (Simultaneous Assignment) 允许同时对多个变量赋值：

```
<var1>, <var2>, ..., <varn> = <expr1>, <expr2>, ..., <exprn>
```

语义上，Python 将对右端所有的表达式求值，然后把这些值赋值给左端相应的变量。例如：

```
sum, diff = x + y, x - y
```

如何交换 2 个变量的值？一种通用的方式是使用一个中间变量：

```
temp = x
x = y
y = temp
```

如果使用同时赋值：

```
x, y = y, x
```

因为赋值是同时进行的，所以避免了其中一个值被覆盖掉的问题。

将同时赋值应用于 input-eval 语句中：

```
x, y = eval(input("Enter two values separated by a comma: "))
```

注意，该语句可以扩展到为多个变量输入数值。另外，此种形式的语句仅限于为多个数值类型变量输入数据，并不适用于其它类型的变量。

3.8 简单循环

循环 for 语句的语法如下：

```
for <var> in <sequence>:
    <body>
```

其中，循环体 <body> 必须使用缩进格式，<var> 是循环索引 (Loop Index)，它的取值来自于 <sequence>。对于 <var> 的每个取值，<body> 将被执行一次。

通常，<sequence> 是一个列表 (List)，列表是 Python 中的一个重要概念，在后面将会介绍。例如:

```
for i in [0, 1, 2, 3, 4, 5]:
    print(i)
```

在 Python 中，有一个内建函数 range，可以动态地生成一个列表：

```
for i in range(10):
    print(i)
```

该程序片段等价于：

```
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print(i)
```

实际上，range(n) 将输出一个列表 [0, 1, 2, ..., n-1](元素不包括 n)，元素的个数是 n。可以使用另外一个内建函数 list 显式地生成一个列表：

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

在一些情况下，我们只关注循环的次数 n，而不用关注循环索引 <var> 的具体值。

再来看另外一个程序：

```
# factorial.py
#    Program to compute the factorial of a number
#    Illustrates for loop with an accumulator

def main():
    n = int(input("Please enter a whole number: "))
    fact = 1
    for factor in range(n,1,-1):
        fact = fact * factor
    print("The factorial of", n, "is", fact)

main()
```

上面的程序中，range(n,1,-1): start = n，end = 1，step = -1，将输出序列 [n,n-1,n-2,...,2]，可以看出，序列中并没有包括 end。另外，range(1,n) 等价于 range(1,n,1)。查看 range 函数的帮助文档，并在终端下试试 range 函数的其它用法。

试着修改程序 factorial.py 中 range 函数的调用参数，使它也能够实现计算阶乘的功能。

注意，factorial.py 可以计算出很大整数的阶乘，例如：

```
Please enter a whole nuber: 100
71
The factorial of 100 is 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

之所以能够计算出这种“巨大”数值，原因是 Python 的整型数据不是采用固定的字节个数来表示的，而是采用任意字节大小（受限于内存空间）[2]。

[2]在其它语言中，例如，C、C++、Java 等，诸如 int 这样的数据类型采用固定的字节个数来表

通常，for 循环作用于一个可迭代对象时，就可以遍历该对象的所有元素。可以使用下面的方法判断一个对象是否可迭代：

```
>>> from collections import Iterable
>>> isinstance('test', Iterable)
True
>>> isinstance([10,20,30], Iterable)
True
```

有时候，程序中需要同时迭代索引和元素，可以使用 enumerate 内建函数：

```
>>> for i, v in enumerate(['1', '2', '3']):
        print(i, v)
0 1
1 2
2 3
```

3.9 程序实例

下面看一个具体的 Python 程序：

```
# File: chaos.py
# A simple program illustrating chaotic behavior.

def main():
    print("This program illustrates a chaotic function")
    x = eval(input("Enter a number between 0 ad 1: ") )
    for i in range(10):
        x = 3.9 * x * (1 - x)
        print(x)

main()
```

示，因而大小是有限制的，其参与各种计算所得的结果也是有限制的。例如，在 Java 中计算 13 的阶乘，所得到的结果就是错误的

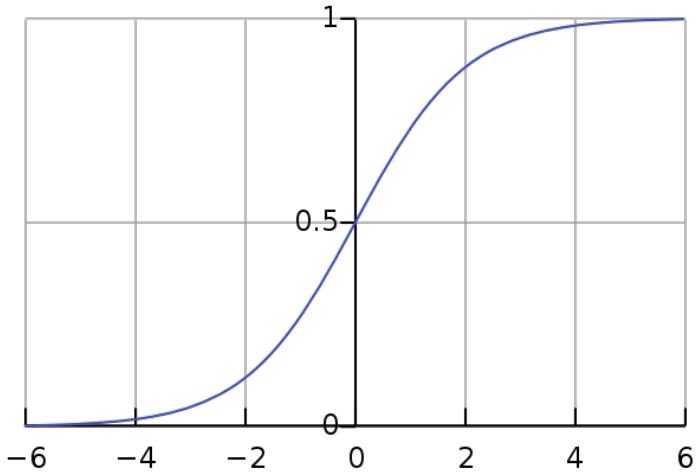

图 3-4: S 形函数的曲线

上面的程序中，x 的计算公式为：$kx(1-x)$，本例中，k = 3.9。从计算的角度来说，表达式 $x = 3.9 * x * (1-x)$ 用于迭代地计算出 x 的值，是一件再平常不过的事情了。但是，计算 x 的表达式与 Logistic 函数 (也被称为 sigmoid 函数或 S 形函数) 有着密切的联系，它在多个领域中发挥着非常重要的作用[3]。

程序 chaos.py 会产生一个有趣的现象，初始值的微小变化将会导致结果有较大的差异，即计算结果对初始值非常敏感，如图3-5所示。

下面再看另一个程序：

```
# futval.py
#    A program to compute the value of an investment
#    carried 10 years into the future

def main():
```

[3]Logistic 函数或 S 形函数的计算公式为：$f(x) = \frac{L}{1+e^{-k(x-x_0)}}$。$x_0$ 表示 S 形的中点，L 表示函数的最大值，k 表示曲线的陡度。标准的 S 形曲线如图3-4所示，其中，$L = 1, k = 1, x_0 = 0$。

该函数由 Pierre Francois Verhulst 在 1844-1845 年间命名，用于研究人口增长的规律。由函数曲线可以看出，它的初始增长曲线与指数函数近似，然后，曲线开始饱和，增长变得缓慢，最后增长停止了。S 形函数广泛应用于人工神经网络、生物学与生态学、经济学等领域。

它也与双曲正切函数 (Hyperbolic Tangent Function)tanh 有着密切的关系：$f(x) = \frac{1}{2} + \frac{1}{2}tanh(\frac{x}{2})$ 或 $tanh(x) = 2f(2x) - 1$，公式表明，S 形函数是 tanh 函数经过缩放和平移获得的。

S 形函数的导数为：$f'(x) = f(x)(1-f(x))$。

```
input   0.25         0.26
--------------------------
      0.731250     0.750360
      0.766441     0.730547
      0.698135     0.767707
      0.821896     0.695499
      0.570894     0.825942
      0.955399     0.560671
      0.166187     0.960644
      0.540418     0.147447
      0.968629     0.490255
      0.118509     0.974630
```

图 3-5: chaos.py 的蝴蝶效应

```
    print("This program calculates the future value")
    print("of a 10-year investment.")

    principal = eval(input("Enter the initial principal: "))
    apr = eval(input("Enter the annual interest rate: "))

    for i in range(10):
        principal = principal * (1 + apr)

    print("The value in 10 years is:", principal)

main()
```

在循环结构中，利用迭代公式计算出 10 年后本金及利息的数额。

3.10 练习

1. 编写一个程序，计算 n 个数的平均值；

2. 修改程序 convert.py，输出 0°-100° 的对应温度值；

4 Python 的数值数据

4.1 数值数据类型

首先来看一个小程序，输入各种零钱的数量并输出美元数值：

```
# change.py
#   A program to calculate the value of some change in dollars

def main():
    print("Change Counter")
    print()
    print("Please enter the count of each coin type.")
    quarters = eval(input("Quarters: "))
    dimes = eval(input("Dimes: "))
    nickels = eval(input("Nickels: "))
    pennies = eval(input("Pennies: "))
    total = quarters * .25 + dimes * .10 + nickels * .05 + pennies * .01
    print()
    print("The total value of your change is", total)

main()
```

运行该程序，得到一个输出示例：

```
Chage Couter

Please enter the count of each coin type.
Quarters: 5
Dimes: 3
Nickels: 4
Pennies: 6

The total value of your change is 1.81
```

上面的例子中，有 2 种数值数据类型：整型 (Integer 或 Whole Number) 和浮点型 (Float)。其中，5、3 等属于整型数据，0.25、0.10、1.81 等属于浮点数据。

在计算机内部或 Python 中，它们的计算与存储方式完全不同，属于完全不同的数值数据类型。数值数据类型决定了它的取值范围、计算方式与计算结果。

Python 提供了内建函数 type，可以查看数据类型：

```
>>> type(3)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> x = 32.0
>>> type(x)
<class 'float'>
```

为什么要设计 2 种不同类型的数值类型？原因如下：

1. 符合人们的习惯。例如，表示数量、用于计数的数据最好是整型数据；
2. 从执行效率的角度考虑。数据的底层表示、存储与运算方式不一样，一般来讲，整型数据的效率要高；
3. 从精度的角度考虑。在计算机实现中，整型数值的表示是精确的，而浮点数值的表示是近似的，不是严格精确的。

注意，如果应用允许的话，请尽量使用整型数据。

图4-6列出了 Python 内建的数值运算操作。

如图4-6所示，虽然 +、-、*、/、**、abs()、% 这些运算操作可以同时作用于整型数据和浮点数据，但是它们的底层实现是不一样的，Python 已经进行了抽象处理，方便程序员使用。

考虑下面的程序片段：

```
>>> 3 + 4
7
>>> 3. 0 + 4. 0
7. 0
```

operator	operation
+	addition
−	subtraction
*	multiplication
/	float division
**	exponentiation
abs()	absolute value
//	integer division
%	remainder

图 4-6: Python 内建的数值运算操作

```
>>> 3 * 4
12
>>> 3 * 4. 0
12. 0
>>> 4 ** 3
64
>>> 4. 0 ** 3
64. 0
>>> 4. 0 ** 3.0
64. 0
>>> abs (5)
5
>>> abs (-3.5)
3.5
```

从上面可以看出，一般而言，整型数据的运算结果是整型，浮点数据的运算结果是浮点型，整型与浮点数据混合运算的结果是浮点型。

然而，Python3.x 提供了 2 种不同的除法：/和//。前者表示浮点除法（不论参与计算的数据是整型还是浮点型，结果均是浮点型），后者表示整除（结果是“整的”，结果的数据类型与参与计算的操作数的最高数据类型相同）：

```
>>> 10 / 3
3.3333333333333335
>>> 10.0 / 3.0
```

```
3.3333333333333335
>>> 10 / 5
2.0
>>> 10 // 3
3
>>> 10.0 // 3.0
3.0
>>> 10 % 3
1
>>> 10.0 % 3.0
1.0
```

% 表示求余计算，结果的数据类型与参与计算的操作数的最高数据类型相同。从数学的角度来看，下面的等式成立：$a = (a//b)(b) + (a\%b)$。

例如，在程序 convert.py 中，有一个表达式：$9/5 * celsius + 32$，其中，9/5 将得到浮点型结果，符合人们的习惯。但是，在其它的程序设计语言中，其结果可能会是一个整型数据（整除计算），需要进行额外的处理，才能得到预期正确的结果。

4.2 类型转换与四舍五入

在混合类型表达式 (Mixed-typed Expression) 中，Python 会将整型数据转换成浮点数据，结果也依然是浮点数据，原因是将整型数据转换成浮点数据是安全的，反之，是不安全的。

```
>>> 3 * 4.0
12.0
```

Python 也支持显式的类型转换，要使用内建函数 int 和 float。

```
>>> int(4.5)
4
>>> int(3.9)
3
>>> float(4)
```

```
4.0
>>> float(4.5)
4.5
>>> float(int(3.3))
3.0
>>> int(float(3.3))
3
>>> int(float(3))
3
```

注意，在将 float 类型数据显示转换到 int 类型数据时，简单地将小数部分截断 (Truncating)，而不是四舍五入 (Rounding)。

如果要四舍五入，可以使用内建函数 round:

```
>>> round(3.14)
3
>>> round(3.5)
4
```

如果要将 float 类型数据转换成另一个 float 类型数据，可以使用带参数的 round 函数：

```
>>> pi = 3.141592653589793
>>> round(pi, 2)
3.14
>>> round(pi,3)
3.142
```

需要注意的是，在 Python 内部，显示出来的数值 3.14 与实际存储的数值是不一样的，这是因为浮点数据本身是近似的而不精确的。Python 在显示时，会自动进行四舍五入处理。

int 和 float 内建函数也可以将字符串转换成数值：

```
>>> int("32")
32
```

```
>>> float("32")
32.0
>>> float("9.8")
9.8
```

注意，在进行类似转换时，int 和 float 函数要比 eval 函数好，因为从某种程度来讲，eval 函数是不安全的，它具有副作用。下面是改进的计算零钱程序：

```
# change2.py
#   A program to calculate the value of some change in dollars
#   This version avoids the eval function

def main():
    print("Change Counter")
    print()
    print("Please enter the count of each coin type.")
    quarters = int(input("Quarters: "))
    dimes = int(input("Dimes: "))
    nickels = int(input("Nickels: "))
    pennies = int(input("Pennies: "))
    total = .25*quarters + .10*dimes + .05*nickels + .01*pennies
    print()
    print("The total value of your change is", total)

main()
```

在程序运行时，如果用户输入了任何非法数据（不能被转换成 int 数据），那么程序将会弹出出错信息并停止运行，因而避免了代码注入攻击 (Code Injection Attack)。

唯一的缺点是 int 和 float 函数不支持同时赋值：

```
>>> #simultaeous input using eval
>>> x,y = eval(input("Enter (x,y): "))
Enter (x,y) : 3,4
```

function	meaning
float(<expr>)	Convert expr to a floating-point value.
int(<expr>)	Convert expr to an integer value.
str(<expr>)	Return a string representation of expr.
eval(<string>)	Evaluate string as an expression.

图 4-7: 数据类型转换函数

```
>>> X
3
>>> y
4
>>> #does not work with float
>>> x,y = float(input("Enter (x,y) : "))
Enter (x,y) : 3,4
Traceback (most recent call last) :
File "<stdin >", line 1, in <module >
ValueError: could not convert string to float: '3,4'
```

在一般情况下，最好使用这种类型转换函数。

在 Python 中，可以使用 str 函数将数值类型数据转换成字符串类型数据：

```
>>> str(500)
'500'
>>> str(3.14)
'3.14'
>>> print("The value is", str(10.0) + '.')
The value is 10.0.
```

图4-7展示了 4 种数据类型转换函数。

4.3 使用 Math 库

下面的程序演示了 Math 库函数 sqrt 的使用：

```
# quadratic.py
#    A program that computes the real roots of a quadratic equation.
```

```
#    Illustrates use of the math library.
#    Note: this program crashes if the equation has no real roots.

import math  # Makes the math library available.

def main():
    print("This program finds the real solutions to a quadratic")
    print()

    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discRoot = math.sqrt(b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)

    print()
    print("The solutions are:", root1, root2 )

main()
```

在上面的程序中，首先导入数学库 math，在计算 discRoot 时，使用了 math.sqrt 函数，用来计算平方根。注意，本程序输入的方程系数要使得方程有实数根，否则，程序将报错退出。

实际上，本程序可以不使用 math.sqrt 函数，可以使用 ** 代替它。其它常见的数学常量与函数如图4-8所示。

4.4 练习

1. 编写一个程序，计算平方根。试着使用下面的三种方法：

   (a) 穷举法；

Python	mathematics	English
`pi`	π	An approximation of pi.
`e`	e	An approximation of e.
`sqrt(x)`	$\sqrt{x}$	The square root of x.
`sin(x)`	$\sin x$	The sine of x.
`cos(x)`	$\cos x$	The cosine of x.
`tan(x)`	$\tan x$	The tangent of x.
`asin(x)`	$\arcsin x$	The inverse of sine x.
`acos(x)`	$\arccos x$	The inverse of cosine x.
`atan(x)`	$\arctan x$	The inverse of tangent x.
`log(x)`	$\ln x$	The natural (base e) logarithm of x.
`log10(x)`	$\log_{10} x$	The common (base 10) logarithm of x.
`exp(x)`	e^x	The exponential of x.
`ceil(x)`	$\lceil x \rceil$	The smallest whole number >= x.
`floor(x)`	$\lfloor x \rfloor$	The largest whole number <= x.

图 4-8: math 库中常见的常量与函数

(b) 二分法；

(c) 牛顿-拉夫逊方法；

将程序的运行结果与 math.sqrt 提供的结果做一个对比；

2. 编写一个程序，输出斐波那契数列 (Fibonacci Sequence)；

3. 编写一个程序，近似计算 $\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$，并将结果与 math.pi 进行比较；

5 图形

本部分主要介绍 Python 的 GUI 编程及基本元素的绘制，使用的底层 GUI 模块是 Python 提供的 Tkinter 库，但是，为方便起见，没有直接使用该库，而是使用 graphics.py，它对 Tkinter 进行了封装，易于初学者使用。

5.1 绘制简单的图元

首先，将 graphics.py 放入当前目录，进入命令行并导入图形库：

```
>>> import graphics
>>> win = graphics.GraphWin()
```

另一种导入方法可以简化输入：

```
>>> from graphics import *
>>> win = GraphWin()
```

这样，无需输入模块名称 graphics 及“.”符号了。实际上，你也可以使用具体的函数和类名称替换“*”，以导入指定的函数和类。此处，使用“*”，表明导入模块 graphics 中的一切函数和类。

上述语句执行后，将会打开一个 GUI 窗口，缺省情况下，窗口的宽度与高度是 200 * 200。

调用以下语句，可以关闭该窗口：

```
>>> win.close()
```

新建一个文件 simple1.py，按如下内容输入：

```
# simple1.py
# Simple Graphics Program
from graphics import *

win = GraphWin()
p1 = Point(50, 60)
print(p1.getX())
print(p1.getY())
p1.draw(win)
p2 = Point(140, 100)
p2.draw(win)

input("Please enter any key to exit")
win.close()
```

上面的程序中，p1 是一个 Point 对象，位于 x=50，y=60 处，p1 的方法 getX 和 getY 分别获取 x 和 y 坐标，方法 draw(win) 表示在 win 窗口对象上绘制 p1 点。运行程序，观察结果。

除了点对象之外，图形库还支持线、圆、矩形、椭圆、多边形及文本，它们的绘制方式与点对象一样。下面是一个稍复杂的绘图程序，结果如图5-9所示：

```
# simple2.py
# Simple Graphics Program
from graphics import *

win = GraphWin()
center = Point(100, 100)
circle = Circle(center, 20)
circle.setOutline('green')
circle.setFill('blue')
circle.draw(win)

label = Text(center, "Circle")
label.draw(win)

rect = Rectangle(Point(30, 50), Point(170, 150))
rect.setOutline('pink')
rect.draw(win)

line = Line(Point(30, 50), Point(170, 150))
line.setOutline('red')
line.draw(win)

oval = Oval(Point(30, 50), Point(170, 150))
oval.draw(win)


input("Please enter any key to exit")
win.close()
```

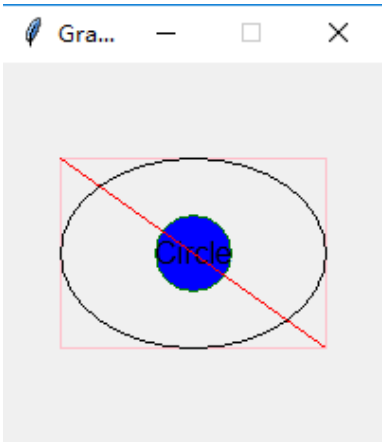

图 5-9: 绘制简单的图元对象

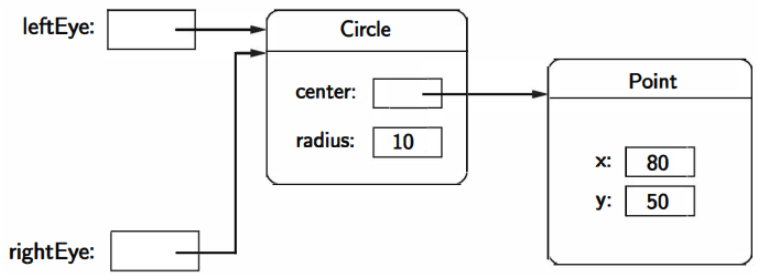

图 5-10: rightEye 是 leftEye 的别名

5.2 注意事项

先来看一个程序片段:

```
## Incorrect way to create two circles .
leftEye = Circle(Point(80 , 50) , 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
rightEye = leftEye
rightEye.move(20 , 0)
```

该程序片段中存在一个问题:rightEye = leftEye，实际上，这 2 个对象是同一个对象，rightEye 只不过是 leftEye 的一个别名对象，如图5-10所示。

针对上述问题，可以使用 clone 方法来解决:

```
## Correct way to create two circles , using clone .
```

```
leftEye = Circle(Point(80 , 50) , 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
rightEye = leftEye.clone( ) # rightEye is a exact copy of the left
rightEye.move(20 , 0)
```

关于别名对象问题，此处稍作扩展介绍一下。Python 的任何东西都是对象（即使它是一个具体的数值，例如 10，也是一个对象）。另一方面，Python 又是动态类型语言。因此，有些情况下，赋值会生成新对象，例如：

```
>>> a = 10
>>> b = a
>>> a = 20
>>> b
10
>>> a
20
```

a 和 b 是 2 个不同的对象。另一些情况下，赋值不会生成新对象，而仅仅只生成别名对象，它们是同一个对象[4]，例如：

```
>>> p = [1,2,3,4]
>>> q = p
>>> q[0] = 5
>>> q
[5, 2, 3, 4]
>>> p
[5, 2, 3, 4]
```

实际上，上述现象涉及到 Python 的对象管理策略、对象的可变性 (mutability) 和不可变性 (immutability) 概念，将在后面做详细介绍（参见7.2）。

[4]技术上，可以统一从 “Python 一切皆对象” 的角度来理解赋值：对象的赋值，只不过是让表达式左边的对象指向右边的对象或将右边对象的 ID 号赋值给左边对象。在这种观点下，所有的赋值表达式中，左边对象就是右边对象的别名对象。详细的解释请参见7.2。

5.3 图形化程序实例

下面将使用学习到的图形化技术来改造前面的 futval.py 程序。由于问题变得稍许复杂一些，为使问题清晰化并最终得以编程实现，可以采取如下步骤：

1. 陈述需求规范，进行总体设计（需求分析 + 总体设计）；
2. 进一步精炼需求规范，将总体设计细化为详细设计，尽可能使用伪代码形式来描述算法（详细设计 + 算法描述）；
3. 使用 Python 设计程序；

首先，提出需求规范并进行总体设计。需要解决的问题是，使用柱状图来图形化显示存款的 (年) 数值：

```
Print a introduction
Get value of principal ad apr from user
Create a GraphWin
Draw scale labels on left side of window
Draw bar at position 0 with height corresponding to principal
For successive years 1 through 10
Calculate principal = principal * (1 + apr)
Draw a bar for this year having a height corresponding to principal
Wait for user to press Enter.
```

然后进一步细化为：

```
Print a introduction
Get value of principal ad apr from user
Create a 320x240 GraphWin titled ''Investment Growth Chart''
Draw label "0.0K" at (20 , 230)
Draw label "2.5K" at (20 , 180)
Draw label "5.0K" at (20 , 130)
Draw label "7.5K" at (20 , 80)
Draw label "10.0K" at (20 , 30)
Draw a rectagle from (40 , 230) to (65 , 230 - principal * 0.02)
```

```
for year running from a value of 1 up through 10 :
    Calculate principal = principal * (1 + apr)
    Calculate xll = 25 * year + 40
    Draw a rectagle from (xll , 230) to (xll+25 , 230 - principal * 0.02)
Wait for user to press Enter
```

最后，依据算法写出 Python 程序：

```
# futval_graph.py

from graphics import *

def main():
    # Introduction
    print("This program plots the growth of a 10-year investment.")

    # Get principal and interest rate
    principal = float(input("Enter the initial principal: "))
    apr = float(input("Enter the annualized interest rate: "))

    # Create a graphics window with labels on left edge
    win = GraphWin("Investment Growth Chart", 320, 240)
    win.setBackground("white")
    Text(Point(20, 230), ' 0.0K').draw(win)
    Text(Point(20, 180), ' 2.5K').draw(win)
    Text(Point(20, 130), ' 5.0K').draw(win)
    Text(Point(20, 80), ' 7.5K').draw(win)
    Text(Point(20, 30), '10.0K').draw(win)

    # Draw bar for initial principal
    height = principal * 0.02
    bar = Rectangle(Point(40, 230), Point(65, 230-height))
    bar.setFill("green")
```

```
    bar.setWidth(2)
    bar.draw(win)

    # Draw bars for successive years
    for year in range(1,11):
        # calculate value for the next year
        principal = principal * (1 + apr)
        # draw bar for this value
        xll = year * 25 + 40
        height = principal * 0.02
        bar = Rectangle(Point(xll, 230), Point(xll+25, 230-height))
        bar.setFill("green")
        bar.setWidth(2)
        bar.draw(win)

    input("Press <Enter> to quit")
    win.close()

main()
```

5.4 坐标系统

在图形系统的设计与实现过程中，一个重要的问题是坐标系统。引入坐标系统的一个主要目的是为了方便程序员处理图形显示问题。例如，在 OpenGL 和 Direct3D 图形引擎中，要将一个 2D/3D 模型显示在屏幕上，需要经过模型坐标系-世界坐标系-相机坐标系-投影坐标系-视口坐标系等一系列坐标变换。这种分层设计的坐标系统带来的一个好处是，可以隔离空间坐标数据处理的复杂度，让程序员在处理这些空间坐标数据时，只需要在世界坐标系处理各种模型的显示与交互即可，剩余的显示工作全部交由图形引擎处理即可，这样，可以大大简化程序员的图形开发任务。

在程序 futval_graph.py 中，问题的解决方案是清晰的，但是解决问题的过程是繁琐的。原因在于，一个柱状图的显示要受到窗口大小、其它柱状图和文本图

元的位置与大小等因素的影响。因而，如果不加以任何分层抽象处理的话，图元的显示工作将是非常耗时的。

为了解决这个问题，可以采取类似于图形引擎那样的“坐标系统”概念来简化图形界面的设计工作。

具体来说，该问题的横坐标 x 表示年份 (year)，其值范围为 $0-10$，纵坐标 y 表示存款总数 (principal)，其值范围为 $\$0-\$10,000$。在原先的程序中，我们将 year 和 principal 这 2 个数据分别转换成了对应的图形坐标数据，即将现实世界中的数据转换到了图形世界中的坐标数据。

实际上，我们可以换一种思路，在这两者之间添加一个坐标系统，该坐标系统的任务是将现实世界中的数据转换成图形世界中的数据，这样程序员在进行图形显示时，直接使用现实世界中的数据，而不用考虑实际的图形坐标数据。

那么，如何实现该坐标系统呢？答案是，使用线性变换在两者之间建立直接的缩放比例关系。对于本例而言，窗口大小为 $320*240$(分别为窗口的 X、Y 方向大小)，问题大小为 $10*10,000$(分别为问题的 year、principal 大小)。它们对应的数据范围分别为 $0-320$ VS. $0-10$、$0-240$ VS. $0-10,000$。可以看出，它们之间实际上是一种线性变换关系。

幸运的是，本图形库提供了坐标系统的转换函数：在创建 GraphWin 对象时，可以使用方法 setCoords 指定坐标系统的变换关系。该方法需要四个参数，分别指定问题域中左下角、右上角的坐标数据。

在给出 futval_graph.py 的修改版之前，我们来看一下该方法的另一个应用场景：

```
# ttt_graph.py
# A graph program displays Tic-Tac-Toe GUI.

from graphics import *
# create a default 200x200 window
win = GraphWin("Tic-Tac-Toe ", 200, 200)
# set coordinates to go from (0 , 0) in the lower left
# to (3 , 3) in the upper right .
win.setCoords(0.0, 0.0, 3.0 , 3.0)
# Draw vertical lines
```

```
Line(Point(1 , 0) , Point(1 , 3)).draw(win)
Line(Point(2 , 0) , Point(2 , 3)).draw(win)
# Draw horizontal lines
Line(Point(0 , 1) , Point(3 , 1)).draw(win)
Line(Point(0 , 2) , Point(3 , 2)).draw(win)

input("Press <Enter> to quit")
```

在上例中，坐标系统的引入，屏蔽了具体的图形窗口与坐标，使得程序员可以专注于问题本身。坐标系统带来的另一个好处是，窗口大小的改变，并不会改变程序员头脑中那个已经得到抽象化了的问题域坐标概念，因此也就不会影响到程序员以往及将来编写的任何代码。

下面是柱状图程序的改进版本：

```
# futval_graph2.py

from graphics import *

def main():
    # Introduction
    print("This program plots the growth of a 10-year investment.")

    # Get principal and interest rate
    principal = float(input("Enter the initial principal: "))
    apr = float(input("Enter the annualized interest rate: "))

    # Create a graphics window with labels on left edge
    win = GraphWin("Investment Growth Chart", 320, 240)
    win.setBackground("white")
    win.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), ' 0.0K').draw(win)
    Text(Point(-1, 2500), ' 2.5K').draw(win)
    Text(Point(-1, 5000), ' 5.0K').draw(win)
```

```
    Text(Point(-1, 7500), ' 7.5k').draw(win)
    Text(Point(-1, 10000), '10.0K').draw(win)

    # Draw bar for initial principal
    bar = Rectangle(Point(0, 0), Point(1, principal))
    bar.setFill("green")
    bar.setWidth(2)
    bar.draw(win)

    # Draw a bar for each subsequent year
    for year in range(1, 11):
        principal = principal * (1 + apr)
        bar = Rectangle(Point(year, 0), Point(year+1, principal))
        bar.setFill("green")
        bar.setWidth(2)
        bar.draw(win)

    input("Press <Enter> to quit.")
    win.close()

main()
```

5.5 交互式图形的设计

交互式图形也被称为是 GUI 图形、图形接口、GUI 图形界面或 GUI 图形接口，既可以用于输入，也可以用于输出。在交互式图形系统中，用户通过鼠标点击 Button、菜单及在对话框中输入数据来与图形系统进行交互。因此，这类应用程序的运行流程是事件驱动的，为这类应用程序编写程序被称为是事件驱动的程序设计。事件驱动的程序设计方法是基于 GUI 的现代程序设计方法的基础与核心。

当用户用鼠标点击按钮、菜单或其它图形元素时，系统将会产生事件对象或消息。该事件对象，封装了具体相关的事件数据，将被发送给有关的应用程序进行处理，从而驱动了该应用程序的运行。

下面将描述交互式图形设计几个方面的问题。

5.5.1 鼠标支持

graphics.py 程序提供了 GraphWin 的成员函数 getMouse，它暂停程序的执行，等待用户在图形窗口中点击鼠标左键，并输出一个 Point 对象，以返回点击点的坐标。下面的程序展示了这一功能：

```
# click.py
from graphics import *

def main():
    win = GraphWin("Click Me!")
    for i in range(10):
        p = win.getMouse()
        print("You clicked at:", p.getX(), p.getY())

main()
```

下面的程序使用鼠标点击实现了三角形的实时交互式绘制功能：

```
# triangle.pyw
from graphics import *

def main():
    win = GraphWin("Draw a Triangle")
    win.setCoords(0.0, 0.0, 10.0, 10.0)
    message = Text(Point(5, 0.5), "Click on three points")
    message.draw(win)

    # Get and draw three vertices of triangle
    p1 = win.getMouse()
    p1.draw(win)
    p2 = win.getMouse()
```

```
    p2.draw(win)
    p3 = win.getMouse()
    p3.draw(win)

    # Use Polygon object to draw the triangle
    triangle = Polygon(p1,p2,p3)
    triangle.setFill("peachpuff")
    triangle.setOutline("cyan")
    triangle.draw(win)

    # Wait for another click to exit
    message.setText("Click anywhere to quit.")
    win.getMouse()

main()
```

5.5.2 键盘支持

GraphWin 类也提供了一个方法 getKey，支持从键盘输入数据。例如，程序：

```
# clickntype.py
# Graphics program illustrating mouse and keypress inputs

from graphics import *

def main():
    win = GraphWin("Click and Type", 400, 400)
    for i in range(10):
        pt = win.getMouse()
        key = win.getKey()
        field = Text(pt, key)
        field.draw(win)
```

```
main()
```

当程序运行到 getKey 语句时，将等待用户输入按键，并返回该按键所代表的字符串。例如，如果用户按下 ENTER 键，将返回"Return"，按下左 CTRL 键，将返回"Control_L"，按下 F1 键，将返回"F1"，按下 a 键，将返回"a" 等。

如果要允许用户输入（一串）文本数据，那么可以使用 Entry 类，下面的程序为 convert.py 增加了 GUI 功能，允许用户在文本框中输入数据：

```
# convert_gui.py
# Program to convert Celsius to Fahrenheit using a simple
#   graphical interface.

from graphics import *

def main():
    win = GraphWin("Celsius Converter", 400, 300)
    win.setCoords(0.0, 0.0, 3.0, 4.0)

    # Draw the interface
    Text(Point(1,3), "   Celsius Temperature:").draw(win)
    Text(Point(1,1), "Fahrenheit Temperature:").draw(win)
    inputText = Entry(Point(2.25,3), 5)
    inputText.setText("0.0")
    inputText.draw(win)
    outputText = Text(Point(2.25,1),"")
    outputText.draw(win)
    button = Text(Point(1.5,2.0),"Convert It")
    button.draw(win)
    Rectangle(Point(1,1.5), Point(2,2.5)).draw(win)

    # wait for a mouse click
    win.getMouse()
```

```
    # convert input
    celsius = float(inputText.getText())
    fahrenheit = 9.0/5.0 * celsius + 32

    # display output and change button
    outputText.setText(round(fahrenheit,2))
    button.setText("Quit")

    # wait for click and then quit
    win.getMouse()
    win.close()

main()
```

5.6 图形模块概览

本部分介绍图形模块中包含的类及其方法，在后续开发图形应用的时候，要经常参考本部分的内容。

5.6.1 GraphWin 类

该类包含的方法如下：

1. GraphWin(title, width, height)

   该类的构造函数，用于实例化一个新的图形窗口对象，它是图元与图形子对象（也被称为 Widget，例如按钮、文本、文本编辑框等）的父容器，为数据的显示及用户交互提供了载体。

   参数是可选的，缺省的 title 是 “Graphics Window”，缺省的 width 和 height 都是 200。

   调用实例：win = GraphWin(“Investment Growth”, 640, 480)

2. plot(x, y, color)

在图形窗口的 (x, y) 处绘制一个像素，color 参数是可选的，缺省的 color 是“black”。

调用实例：win.plot(35, 128, “blue”)

3. plotPixel(x, y, color)

   在图形窗口的原始坐标 (x, y) 处（忽略任何由方法 setCoords 引入的坐标变换）绘制一个像素，color 参数是可选的。调用格式与 plot 相同。

4. setBackground(color)

   设置图形窗口的背景颜色为 color。缺省的背景颜色与系统有关。

   调用实例：win.setBackground(“white”)

5. close()

   关闭图形窗口。

   调用实例：win.close()

6. getMouse()

   等待用户在图形窗口按下鼠标左键，并返回一个 Point 对象，表示鼠标点击位置。

   调用实例：clickPoint = win.getMouse()

7. checkMouse() 功能与 getMouse 类似，但是不会等待用户按下鼠标左键，因此，它会返回鼠标点击点位置（Point 对象）或 None[5]（表示图形窗口没有被鼠标左键点击）。这种检测事件的方式也被称为是异步查询，与之对应的是同步查询或阻塞式查询，后者需要等待事件的发生。

   调用实例：clickPoint = win.checkMouse()，clickPoint 将是 None，如果没有鼠标左键按下事件；否则，将是 Point 对象。

8. getKey()

   等待用户按键，返回一个字符串，表示用户按下的键名称。

   调用实例：keyString = win.getKey()

[5]None 是特殊的 Python 对象，通常用来表示该变量没有任何值。

9. checkKey()

   功能与 getKey 类似，但是不会等待用户按键。其返回值要么是用户按键名称，要么是空字符串“”（如果没有按键）。

   调用实例：keyString = win.checkKey()，keyString 将是空字符串“”，如果没有按键；否则，将是按键名称。

10. setCoords(xll, yll, xur, yur)

    设置图形窗口的坐标系统。窗口的左下角 (lower-left corner) 是 (xll, yll)，右上角 (upper-right corner) 是 (xur, yur)。

    该方法运行后，当前已经绘制的图形对象将被重画，以后绘制的图形对象将会基于新的坐标系统（plotPixel 除外，它始终使用原来的坐标系统绘制一个像素）

    调用实例：win.setCoords(0, 0, 200, 100)

5.6.2 基本图元类

图形模块提供了以下基本图元类：Point、Line、Circle、Oval、Rectangle、Polygon 和 Text。缺省情况下，这些基本图元是黑色边框，内部没有被填充。它们都包含以下方法：

1. setFill(color)

   设置内部填充颜色。调用实例为：someObject.setFill(“red”)。

2. setOutline(color)

   设置边框轮廓线的颜色。调用实例为：someObject.setOutline(“yellow”)。

3. setWidth(pixels)

   设置边框轮廓线的像素宽度（对 Point 对象不起作用）。调用实例：someObject.setWidth(3)。

4. draw(aGraphWin)

   在指定的图形窗口 aGraphWin 进行绘制，返回值是该绘制的对象（实际上就是调用者本身）。调用实例：someObject.draw(someGraphWin)。

5. undraw()

   从图形窗口中撤销绘制，如果对象当前没有被绘制，则该方法不起作用。调用实例：someObject.undraw()。

6. move(dx, dy)

   将对象的水平位置和垂直位置分别移动 dx 和 dy 个单元。如果对象已经显示在图形窗口中，那么移动效果将会立即反映出来。调用实例：someObject.move(10, 15.5)。

7. clone()

   克隆对象，新生成的对象初始时处于未显示状态。调用实例：objectCopy = someObject.clone()。

Point 类

1. Point(x, y)

   构造函数，实例化一个 Point 对象，调用实例：aPoint = Point(3.5, 8)。

2. getX()

   返回对象的 x 坐标值，调用实例：xValue = aPoint.getX()。

3. getY()

   返回对象的 y 坐标值，调用实例：yValue = aPoint.getY()。

Line 类

1. Line(point1, point2)

   构造函数，实例化一个 Line 对象，调用实例：aLine = Line(Point(1, 3), Point(7, 4))。

2. setArrow(endString)

   设置 Line 对象的箭头状态。箭头可以绘制在第 1 个点处，第 2 个点处，两点处或者都没有，其对应的 endString 取值为：“first”、“last”、“both”、“none”。缺省的取值是 “none”。调用实例：aLine.setArrow(“both”)。

3. getCenter()

   返回 Line 对象的中点（克隆），调用实例：midPoint = aLine.getCenter()。

4. getP1(), getP2()

   返回 Line 对象的相应端点（克隆），调用实例：startPoint = aLine.getP1()、endPoint = aLine.getP2()。

Circle 类

1. Circle(centerPoint, radius)

   使用给定的圆心和半径，实例化一个 Circle 对象。调用实例：aCircle = Circle(Point(3, 4), 10.5)。

2. getCenter()

   返回 Circle 对象的圆心 Point 对象（克隆），调用实例：centerPoint = aCircle.getCenter()。

3. getRadius()

   返回 Circle 对象的半径，调用实例：radius = aCircle.getRadius()。

4. getP1(), getP2()

   返回 Circle 对象边界盒的左下角和右上角 Point 对象（克隆），调用实例：llPoint = aCircle.getP1()、urPoint = aCircle.getP2()。

Rectangle 类

1. Rectangle(point1, point2)

   构造函数，实例化一个 Rectangle 对象，point1 对应左下角 Point 对象，point2 对应右上角 Point 对象，调用实例：aRectangle = Rectangle(Point(1, 3), Point(4, 7))。

2. getCenter()

   返回 Rectangle 对象的中点（克隆），调用实例：centerPoint = aRectangle.getCenter()。

3. getP1(), getP2()

   返回 Rectangle 对象的左下角和右上角 Point 对象（克隆），调用实例：llPoint = aRectangle.getP1()、urPoint = aRectangle.getP2()。

Oval 类

1. Oval(point1, point2)

   构造函数，实例化一个 Oval 对象，point1 对应左下角 Point 对象，point2 对应右上角 Point 对象，调用实例：anOval = Oval(Point(1, 3), Point(4, 7))。

2. getCenter()

   返回 Oval 对象的中点（克隆），调用实例：centerPoint = anOval.getCenter()。

3. getP1(), getP2()

   返回 Oval 对象的左下角和右上角 Point 对象（克隆），调用实例：llPoint = anOval.getP1()、urPoint = anOval.getP2()。

Polygon 类

1. Polygon(point1, point2, point3, ...)、Polygon([point1, point2, point3, ...])

   使用给定的 Point 对象作为顶点，实例化一个 Polygon 对象。调用实例：aPolygon = Polygon(Point(1, 2), Point(3, 4), Point(5, 6))、aPolygon = Polygon([Point(1, 2), Point(3, 4), Point(5, 6)])。

2. getPoints()

   返回一个 list 对象，它存放了构成 Polygon 对象的所有顶点对象（克隆），调用实例：pointList = aPolygon.getPoints()

Text 类

1. Text(anchorPoint, textString)

   实例化一个 Text 对象，以 anchorPoint 为中心，水平显示字符串 textString。调用实例：message = Text(Point(3, 4), “Hello!”)。

2. setText(string)

   将 Text 对象的文本信息设置为 string,调用实例:message.setText(“Goodbye!”)。

3. getText()

   返回 Text 对象的文本信息，调用实例：msgString = message.getText()。

4. getAnchor()

   返回 Text 对象的锚点 (anchor point) 对象（克隆），调用实例：centerPoint = message.getAnchor()。

5. setFace(family)

   设置字体名称为 family，可取的值有：“helvetica”、“courier”、“times roman”、“arial”。调用实例：message.setFace(“arial”)。

6. setSize(sizepoint)

   设置字体大小为 sizepoint，取值范围为5~36，调用实例：message.setSize(18)。

7. setStyle(style)

   设置字体风格，可取的值有：“normal”、“bold”、“italic”、“bold italic”。调用实例：message.setStyle(“bold”)。

8. setTextColor(color)

   设置文本的颜色为 color。注意，setFill 方法也具有同样的效果。调用实例：message.setTextColor(“pink”)。

5.6.3 Entry 类

Entry 对象为用户与图形系统进行交互提供了方便，它是一个文本输入框，用户可以输入各种信息。基本图元具有的常见方法（例如 move、draw、undraw、setFill 和 clone 等）也适用于 Entry 对象。

下面是 Entry 特定的方法：

1. Entry(centerPoint, width)

   实例化一个 Entry 对象，它的中心为 centerPoint，宽度为 width，表示显示的字符个数，调用实例：inputBox = Entry(Point(3, 4), 5)。

2. getAnchor()

   返回 Entry 对象的锚点对象（克隆），调用实例：centerPoint = inputBox.getAnchor()。

3. getText()

   返回 Entry 对象的文本，调用实例：inputStr = inputBox.getText()。

4. setText(string)

   设置 Entry 对象的文本，调用实例：inputBox.setText(“32.0”)。

5. setFace(family) 设置字体名称为 family，可取的值有：“helvetica”、“courier”、“times roman”、“arial”。调用实例：inputBox.setFace(“arial”)。

6. setSize(sizepoint)

   设置字体大小为 sizepoint，取值范围为5~36，调用实例：inputBox.setSize(18)。

7. setStyle(style)

   设置字体风格，可取的值有：“normal”、“bold”、“italic”、“bold italic”。调用实例：inputBox.setStyle(“bold”)。

8. setTextColor(color)

   设置文本的颜色为 color。调用实例：inputBox.setTextColor(“pink”)。

5.6.4 Image 类

Image 类提供了图像的显示与基本操作功能。在大多数平台上，最少支持 PPM 和 GIF 图像格式。该类与其它基本图元对象一样，也支持通用方法：move、draw、undraw 和 clone。下面列出了 Image 类特定的方法：

1. Image(anchorPoint, filename)、Image(anchorPoint, width, height)

   构造函数，创建一个 Image 对象，前者从指定图像文件中读取数据并显示，图像中心由 anchorPoint 指定；后者创建一个空白透明的图像。调用实例：flowerImage = Image(Point(100, 100), “flower.gif”)、blankImage = Image(Point(100, 100), 320, 240)。

2. getAnchor()

   返回 Image 对象的中心 Point 对象（克隆），调用实例：centerPoint = flowerImage.getAnchor()。

3. getWidth()

   返回图像的宽度，调用实例：widthInPixels = flowerImage.getWidth()。

4. getHeight()

   返回图像高度，调用实例：heightInPixels = flowerImage.getHeight()。

5. getPixel(x, y)

   返回图像位置 (x, y) 处像素的颜色：一个 list 对象 [red, green, blue]，其中每个分量的取值范围为0~255。如果要将三分量的颜色转换成字符串形式，可以使用函数 color_rgb。

   注意，(x, y) 是图像本身的相对坐标，图像的左上角坐标始终是 (0, 0)。调用实例：red, green, blue = flowerImage.getPixel(32, 18)。

6. setPixel(x, y, color)

   设置指定位置 (x, y) 的像素值。注意，该位置是相对于图像而言的，不是窗口坐标。另外也要注意，本操作较慢。调用实例：flowerImage.setPixel(32, 18, “blue”)。

7. save(filename)

   将图像数据保存到指定文件，支持的图像格式为 GIF 或 PPM。调用实例：flowerImage.save(“mypic.ppm”)。

5.6.5 颜色

颜色可以使用字符串来表示，例如 “red”、“purple”、“green”、“cyan” 等[6]。

图形库提供了一个函数 color_rgb，以颜色的三分量作为参数，它将返回对应的颜色字符串。注意，颜色分量的取值范围是0~255。调用实例：aCircle.setFill(color_rgb(130, 0, 130))。

[6]完全的颜色名字列表，请参考X11 color names.

5.6.6 显示的更新

在一般情况下，只要图形对象的状态发生变化，图形窗口就会自动更新。然而，在某些情况下，例如在交互式环境中使用图形库，必须要强制刷新图形窗口，以反映出图形对象的变化。此时，可以调用函数 update()：它将强制执行所有未执行的图形操作。

在另外一些情况下，出于运行效率与结果的考虑，不需要图形窗口自动刷新显示。例如，在播放动画的时候，希望所有对象的状态都更新完毕之后才刷新图形窗口。此时，可以使用参数 autoflush 来控制图形窗口是否自动刷新：True，自动刷新；False，关闭自动刷新。例如：

```
win = GraphWin("My Animation", 400, 400, autoflush = False)
```

这样，图形系统只在空闲时间或调用 update 函数之后才刷新图形窗口。

update 函数还支持可选参数，它指定每秒最大更新帧率。例如，可以把函数 update(30) 放在循环体的末尾，使得循环每秒最多执行 30 次：

```
win = GraphWin("Update Example ", 320, 200, autoflush=False)
while True:
    key = win.checkKey()
    if key == 'Escape':
        break
    #<drawing commands for the current frame>
    ...
    update(30)
```

5.7 练习

1. 编写一个程序，绘制射箭的标靶，如图5-11所示。注意，每个圆环有相同的宽度，并且该宽度等于黄色圆的半径；

2. 编写一个程序，绘制一张卡通脸；

3. 编写一个程序，绘制冬天雪景（例如，有雪人、圣诞树等）；

图 5-11: 射箭的标靶

4. 扩展程序 futval_graph2.py，允许用户在图形窗口的文本输入框中输入存款与年利率；

5. 根据描述，为应用绘制图形及添加必要的文本信息：绘制一个 2D 坐标系，标上 x、y 轴，然后绘制两条相交的直线，给它们添加相应的函数表达式，并标记出交点及交点坐标；

6. 在上一练习的基础上，添加鼠标功能，支持直线的实时交点计算及信息更新；

7. 使用鼠标实时绘制矩形，实时更新面积和周长信息；

8. 使用鼠标实时绘制三角形，实时更新面积信息。面积计算公式：$area = \sqrt{s(s-a)(s-b)(s-c)}$，其中，$a$、$b$、$c$ 是边长，$s = \frac{a+b+c}{2}$；

6 字符串、列表与文件

6.1 字符串

我们在前面已经接触过字符串数据类型，下面将详细介绍关于字符串的一些操作与方法。

6.1.1 索引与切片

利用下标可以读取字符串中指定的字符 (串)：

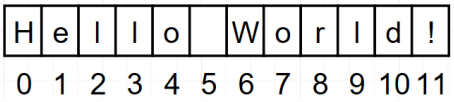

图 6-12: 字符串的下标

```
>>> str = 'Hello World!'
>>> str[0]
'H'
>>> str[6]
'W'
>>> str[0:6]
'Hello '
>>> str[-1]
'!'
>>> str[6:-1]
'World'
```

下标示意图如图6-12所示。

从上面的实例可以看出，单个索引可以读取单个字符：元素的位置既可以从左开始数（索引为正，例如，0 表示第 1 个元素，1 表示第 2 个元素，依此类推），也可以从右开始数（索引为负，例如，-1 表示倒数第 1 个元素，-2 表示倒数第 2 个元素，依此类推）。这种通过单个索引值访问元素的方式被称为索引（Indexing）。

除了索引方式外，还可以通过所谓的切片 (Slicing) 方式读取子字符串：<string>[<start>:<end>]，其中，<start> 和 <end> 都是 int 类型数据，所生成的子字符串包含 <string> 中从 <start> 开始一直到 <end> 结束（不包括 <end>）的元素。

对于切片方式，可以省略 <start> 和/或 <end>：

```
>>> str[:6]
'Hello '
>>> str[6:]
'World!'
>>> str[:]
```

```
'Hello World!'
```

6.1.2 简单连接与重复连接

简单连接 (Concatenation) 操作使用运算符 “+”，它将 2 个字符串连接在一起。重复 (Repetition) 连接操作使用运算符 “*”，它重复连接原字符串多次。

下面的语句片段演示了简单连接与重复连接：

```
>>> str = 'Junk' + ' Mail'
>>> str
'Junk Mail'
>>> str = 'Delete ' + str
>>> str
'Delete Junk Mail'
>>> 3 * 'Egg '
'Egg Egg Egg '
>>> 'Egg '* 2
'Egg Egg '
```

6.1.3 len 与 for

利用内建函数 len 可以得到字符串的字符个数（长度），例如：

```
>>> len(str)
16
```

利用 for 循环可以遍历字符串中的每个元素：

```
for ch in 'Hello World!':
    print(ch, end=' ')
```

图6-13展示了可用于字符串处理的简单操作。

6.1.4 实例

下面的程序，实现了用户名生成功能。例如，输入用户名全称 “John Beeblebrox”，程序将从名字中取第 1 个字符，从姓中取前 7 个子字符串，并将这 2 部分连接在一起，生成一个用户名 “jbeebleb”(假设全部用小写字符)。程序如下：

operator	meaning
+	concatenation
*	repetition
<string>[ ]	indexing
<string>[:]	slicing
len(<string>)	length
for <var> in <string>	iteration through characters

图 6-13: 简单操作概览

```
# username.py
#    Simple string processing program to generate usernames.

def main():
    print("This program generates computer usernames.\n")

    # get user's first and last names
    first = input("Please enter your first name (all lowercase): ")
    last = input("Please enter your last name (all lowercase): ")

    # concatenate first initial with 7 chars of the last name.
    uname = first[0] + last[:7]

    # output the username
    print("Your username is:", uname)

main()
```

再来看另一个程序，输入月份，输出月份的单词缩写：

```
# month.py
#  A program to print the abbreviation of a month, given its number

def main():
    # months is used as a lookup table
```

```
    months = "JanFebMarAprMayJunJulAugSepOctNovDec"

    n = int(input("Enter a month number (1-12): "))

    # compute starting position of month n in months
    pos = (n-1) * 3

    # Grab the appropriate slice from months
    monthAbbrev = months[pos:pos+3]

    # print the result
    print("The month abbreviation is", monthAbbrev + ".")

main()
```

6.2 列表

6.2.1 基本用法

实际上，图6-13列出的运算符和函数，也可以用于其它的顺序数据对象，例如，列表对象 (list)。

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> [1,2] * 3
[1, 2, 1, 2, 1, 2]
>>> list1 = [1,2,3,4]
>>> list1[0]
1
>>> list1[1:4]
[2, 3, 4]
>>> len(list1)
4
```

列表对象非常有用，它的元素可以是任意类型，并且可以进行混合存放：

```
>>> myList = [1 , "Spam" , 4 , "U"]
>>> myList
[1, 'Spam', 4, 'U']
```

在一些应用中，将数据混合存放在一个列表中是很有用的。

下面的程序对 month.py 做出了修改：用列表对象替换了字符串对象。

```
# month2.py
# A program to print the month abbreviation, given its number.

def main():
    # months is a list used as a lookup table
    months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

    n = int(input("Enter a month number (1-12): "))

    print("The month abbreviation is", months[n-1] + ".")


main()
```

可以看出，使用了新的数据类型之后，程序得到了简化。

虽然字符串与列表都是顺序数据对象，但是两者有着重要的区别：字符串对象不能使用下标进行赋值修改，它是不可修改的；而列表对象可以，即列表对象是可以下标存取的或可修改的 (mutable)。请看下面的程序片段：

```
>>> myList = [34 , 26 , 15 , 10]
>>> myList[2]
15
>>> myList[2] = 0
>>> myList
```

```
[34 , 26 , 0 , 10]
>>> myList[0:1] = [1]
>>> myList
[1, 26, 0, 10]
>>> myList[0:2] = [1,2]
>>> myList
[1, 2, 0, 10]
>>> myString = "Hello World"
>>> myString [2]
' 1 '
>>> myString [2] = 'z'
Traceback (most recent call last) :
File "<stdin>" , line 1 , in <module >
TypeError : 'str' object does not support item assignment
```

从上面可以看出，列表对象可以使用索引和切片的方式进行修改，而字符串对象则不行。但是，需要注意的是，字符串对象可以通过其它方式进行（复制）修改，例如，可以调用方法 replace 进行修改：

```
>>> str = 'Hello'
>>> str.replace('H', 't')
'tello'
>>> str
'Hello'
```

但是，replace 方法执行后，原来的字符串对象仍然保持不变，只是新生成了一个（修改后的）字符串对象。其它的方法也是如此，它们不会修改原来的字符串对象[7]。

6.2.2 列表解析式

列表解析式 (List Comprehensions) 是一种简洁地创建列表对象的方式。

生成列表对象的方式有多种，例如，使用 range 函数：

[7]列表对象是可变类型，而字符串对象是不可变类型，详见7.2.2。

```
>>> list(range(11))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

还可以使用 for 循环：

```
>>> import math
>>> l = []
>>> for x in range(11):
        l.append(math.sqrt(x) + 1)
>>> l
[1.0, 2.0, 2.414213562373095, 2.732050807568877, 3.0, 3.23606797749979,
3.449489742783178, 3.6457513110645907, 3.8284271247461903, 4.0,
4.16227766016838]
```

此时，可以使用列表解析式，该方式比较简洁：

```
>>> [math.sqrt(x) + 1 for x in range(11)]
[1.0, 2.0, 2.414213562373095, 2.732050807568877, 3.0, 3.23606797749979,
3.449489742783178, 3.6457513110645907, 3.8284271247461903, 4.0,
4.16227766016838]
```

还可以添加 if 语句：

```
>>> [x * x for x in range(11) if x % 2 == 0]
[0, 4, 16, 36, 64, 100]
```

甚至可以使用多重循环：

```
>>> [m + n for m in [2, 4, 6] for n in [1, 3, 5]]
[3, 5, 7, 5, 7, 9, 7, 9, 11]
```

导出当前目录下所有的文件与子目录：

```
>>> import os
>>> [d for d in os.listdir('.')]
```

6.2.3 列表生成器

列表解析式的主要问题在于，它实际生成一个列表对象，如果该对象占用很大的内存空间，那么空间效率就会很低。为了解决这个问题，可以使用列表生成器 (List Generator)。从数据的生成方式看，可以把列表解析式看作是离线生成列表的方式，而把列表生成器看作是在线生成列表的方式——迭代计算出列表元素。例如：

```
>>> [m + n for m in [2, 4, 6] for n in [1, 3, 5]]
[3, 5, 7, 5, 7, 9, 7, 9, 11]
>>> g = (m + n for m in [2, 4, 6] for n in [1, 3, 5])
>>> g
<generator object <genexpr> at 0x00000128FE45B888>
>>> for x in g:
        print(x, end = ' ')
3 5 7 5 7 9 7 9 11
```

可以看出，在这个例子中，列表解析式与生成器的唯一区别在于，前者使用方括号，后者使用圆括号。g 是一个生成器，也是一个可迭代对象，可以用于 for 循环中。注意，g 是维持内部状态的对象，即如果 g 已经输出完所有的元素，此时再调用 next 内建函数，将会产生 “StopIteration” 异常：

```
>>> next(g)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

实际上，列表解析式可以稍作修改就可以转换成生成器，例如：

```
>>> g = (x for x in range(10) if x % 2 == 0)
>>> for x in g:
        print(x, end = ' ')
0 2 4 6 8
```

如果生成器生成元素的算法比较复杂，还可以使用函数来实现。例如，斐波那契数列的 (函数) 生成器：

```
1  # fibgenerator.py
2
3  def fib(n):
4      a, b = 0, 1
5      for i in range(1, n + 1):
6          yield b
7          a, b = b, a + b
8      return 'done'
9
10 for x in fib(10):
11     print(x, end = ' ')
12 print()
```

输出结果如下：

```
1 1 2 3 5 8 13 21 34 55
```

程序的第 6 行使用了关键字 “yield”，表明该函数是一个生成器，而不是一个普通函数。生成器的执行过程如下：当需要生成器输出一个元素时，会执行该函数，执行后会遇到 yield 语句，此时函数会输出一个元素并返回；下次再次需要生成器输出一个元素时，该函数将从 yield 语句的下一行开始继续执行，最后也会遇到 yield 语句，此时再次输出一个元素。这个过程循环往复，不断输出新元素，直到没有可输出的元素时为止。此时，如果再要求生成器输出元素，那么就会触发生成器的 “StopIteration” 异常，异常生成的对象实际上就是 “done” 字符串对象 (由 return 语句返回)。

6.3 字符编码

字符串属于文本信息，而计算机只能处理数值信息，因而需要一种编码方式，将文本信息转换成数值信息。

早期的计算机只编码了 127 个字符，它们是大小写英文字母、数字和其它一些符号，这个编码表被称为 ASCII 编码。例如，大写字母 “A” 的编码是 65，小写字母 “z” 的编码是 122。

随着计算机的普及，ASCII 编码远远满足不了各种语言文本信息的处理，于是就出现了各种各样的编码方式，仅就中文编码方式而言，就出现了多种，例如 GB2312、GBK、GB18030、BIG5 等。更为严重的是，全世界有上百种语言，例如，日本把日文编到 Shift_JIS 里，韩国把韩文编到 Euc-kr 里，各国有各国的标准，这样就不可避免地出现冲突，造成的结果就是：多语言混合的文本，显示出来就会出现乱码。

为了解决乱码问题，国际标准化组织 ISO 提出了 Unicode 编码标准，它把所有的语言都统一到这套编码里了。Unicode 标准本身也在不断地发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要 4 个字节）。现代操作系统和大多数编程语言都直接支持 Unicode，Python 也不例外。

实际上，虽然乱码问题解决了，但是从存储与网络传输的角度看，Unicode 也还是存在问题的。例如，汉字“中”已经超出了 ASCII 编码的范围，用 Unicode 编码是十进制的 20013，二进制的 0100111000101101。“A” 的 Unicode 编码，需要在前面补 0：0000000001000001。可以看出，如果文本字符都是英文字符，那么 Unicode 文本的存储空间和传输时间将是 ASCII 文本的 2 倍。

因此，为了解决存储与传输问题，就出现了可变长编码的 UTF-8 编码规则[8]。UTF-8 把一个 Unicode 字符根据不同的数字大小编码成 1-6 个字节，常用的英文字母被编码成 1 个字节，汉字通常是 3 个字节，只有很生僻的字符才会被编码成 4-6 个字节。如果你要传输的文本包含大量英文字符，用 UTF-8 编码就能节省空间。UTF-8 的一大优点，就是 ASCII 编码实际上可以被看成是 UTF-8 编码的一部分：大量只支持 ASCII 编码的历史遗留软件可以在 UTF-8 编码下继续工作。

在弄清楚了 ASCII、Unicode 和 UTF-8 的关系之后，我们来看一下字符串的处理流程：在计算机内存中，统一使用 Unicode 数据，当需要保存到硬盘或者需要传输的时候，就转换为 UTF-8 数据。例如，在使用记事本编辑文本文件时，首先从文件中读取 UTF-8 数据，使用 UTF-8 编码规则将文本数据转换成 Unicode 数据并保存到内存中，在编辑完成需要保存时，再使用 UTF-8 编码规则对 Unicode 文本数据进行转换并保存到文件中。

在 Python3 中，Python 支持 Unicode 字符串，例如：

```
>>> print('测试 test')
测试 test
```

[8]注意：Unicode 是一种编码或字符集，而 UTF-8 是一种编码规则或算法。

对于单个字符，Python 提供了内建函数 ord 和 chr（它们分别是 ordinal 和 character 的缩写），允许在字符及其 Unicode 编码之间进行转换：

```
>>> ord('a')
97
>>> ord('A')
65
>>> chr(97)
'a'
>>> chr(960)
>>> ord('中')
20013
>>> chr(20013)
'中'
```

如果知道字符的编码值，还可以用十六进制这么写：

```
>>> '\u4e2d\u6587'
'中文'
```

我们已经知道，在内存中，Python3 统一使用 Unicode 来处理字符串类型数据[9]，而保存在磁盘或在网络上传输的字符串数据可能会采用各种各样的编码规则[10]。它们所对应的数据类型是不一样的，前者对应 str 类型，后者对应 bytes 类型。例如：

```
>>> type('abc')
<class 'str'>
>>> type(b'abc')
<class 'bytes'>
```

注意，bytes 类型的数据使用了前缀 b。

str 与 bytes 之间经常需要进行转换：

[9]Python 的字符串类型用 str 来表示。

[10]文件或网络传输中具体使用的编码规则，例如 GB2312、GBK、BIG5、SHIFT_JIS 等，与语言、应用及历史有关，有些也考虑了存储与传输效率，例如 UTF-8、UTF-16 等。在 Python 中，它们使用 bytes 数据类型来表示。

- str 转换到 bytes

  使用字符串类本身的方法 encode 进行转换，需要指定编码规则：

```
>>> '中文'.encode('utf-8')
b'\xe4\xb8\xad\xe6\x96\x87'
>>> '中文'.encode('utf-16')
b'\xff\xfe-N\x87e'
>>> '中文'.encode('gb2312')
b'\xd6\xd0\xce\xc4'
>>> '中文'.encode('big5')
b'\xa4\xa4\xa4\xe5'
>>> '中文'.encode('gbk')
b'\xd6\xd0\xce\xc4'
>>> '中文'.encode('gb18030')
b'\xd6\xd0\xce\xc4'
>>> 'abc'.encode('ascii')
b'abc'
```

- bytes 转换到 str

  使用 bytes 类本身的方法 decode 进行转换，需要指定编码规则：

```
>>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
'中文'
>>> b'\xff\xfe-N\x87e'.decode('utf-16')
'中文'
>>> b'\xd6\xd0\xce\xc4'.decode('gb2312')
'中文'
>>> b'\xa4\xa4\xa4\xe5'.decode('big5')
'中文'
>>> b'\xd6\xd0\xce\xc4'.decode('gbk')
'中文'
>>> b'\xd6\xd0\xce\xc4'.decode('gb18030')
```

```
'中文'
>>> b'abc'.decode('ascii')
'abc'
```

可以看出，上述 2 个过程是互逆的。另外，需要注意，纯英文的 str 可以用 ASCII 编码为 bytes，含有中文的 str 可以用上面各种编码规则编码为 bytes，但是无法用 ASCII 编码，因为中文编码的范围已经超过了 ASCII 编码的范围，Python 会报错。

在实际应用中，可以利用 encode 和 decode 函数在内存-磁盘/网络之间进行数据转换。例如，将内存中的 str 数据保存到磁盘或发送到网络，需要调用 encode 函数将 str 数据转换成 bytes 数据；反过来，从磁盘或网络上读取到的是 bytes 数据，需要调用 decode 函数将它转换成 str 数据并保存到内存中。

通常情况下，最好使用 UTF-8 编码规则。

Python 源程序也是一个文本文件，为了让中文注释得以顺利显示与识别，可以在文件的开头添加一行：

```
# -*- coding: utf-8 -*-
```

其目的是，告诉 Python 解释器，要按照 UTF-8 编码规则读取源代码；否则，含有中文的源程序可能不会得到正常的解释运行。另外，还需确保源程序编辑器使用的是 UTF-8 编码规则。

6.4 格式化输出

就格式而言，Python 中的格式化输出与 C 中的类似，用% 来表示一个占位符，它对应着一个要替换的内容：

```
>>> 'Hello, %s' % ('World!')
'Hello, World!'
>>> 'Sum = a + b = %d + %d = %d' % (10, 20, 30)
'Sum = a + b = 10 + 20 = 30'
```

常见的占位符有：

- %d：整数；

- %f：浮点数；
- %s：字符串；
- %x：十六进制整数；

其中，整数和浮点数还可以指定是否补 0 及整数与小数的位数：

```
>>> print('%2d-%02d' % (3, 1))
3-01
>>> print('%.2f' % 3.1415926)
3.14
```

另一种格式化字符串的方法是，使用字符串的 format 方法，它会用传入的参数依次替换字符串内的占位符 0、1……：

```
>>> 'Sum = a + b = {0} + {1} = {2:.1f}'.format(10.02, 20.05, 10.02 + 20.05)
'Sum = a + b = 10.02 + 20.05 = 30.1'
>>> '{0:2.3}'.format(3.14159)
'3.14'
>>> '{0:2.3f}'.format(3.14159)
'3.142'
>>> '{0:2.3}'.format(301.14159)
'3.01e+02'
>>> '{0:2.3f}'.format(301.14159)
'301.142'
```

从示例可以看出，加了尾缀 f 的浮点数据（称为固定点浮点数据）格式化，3 表示精确到小数点后 3 位，而没有加尾缀 f 的浮点数据（称为非固定点浮点数据）格式化，3 表示有效数字的个数，在适当的情况下，后者将以科学记数法来表示。

另外，还可以指定文本对齐的方式：

```
>>> '{0:20.3f}'.format(3.14159)
'               3.142'
>>> '{0:<20.3f}'.format(3.14159)
'3.142               '
```

```
>>> '{0:>20.3f}'.format(3.14159)
'               3.142'
>>> '{0:^20.3f}'.format(3.14159)
'       3.142        '
```

<、>、^ 分别表示左对齐、右对齐与居中，缺省情况下，使用右对齐方式。

6.5 split 方法

在常见的文本处理应用中，经常需要对字符串进行分割。例如：

```
>>> "32 24 25 57".split()
['32', '24', '25', '57']
>>> "32, 24, 25, 57".split(",")
['32', '24', '25', '57']
```

在分割时，可以指定分隔符。

下面的实例利用了函数 ord、chr 和 split，实现了一个所谓的文本编码-解码过程。首先，将文本中每个字符转换成相应的 Unicode 编码值：

```
# text2numbers.py
#     A program to convert a textual message into a sequence of
#         numbers, utilizing the underlying Unicode encoding.

def main():
    print("This program converts a textual message into a sequence")
    print("of numbers representing the Unicode encoding of the message.\n")

    # Get the message to encode
    message = input("Please enter the message to encode: ")

    print("\nHere are the Unicode codes:")

    # Loop through the message and print out the Unicode values
```

```
    for ch in message:
        print(ord(ch), end=" ")

    print() # End line of output

main()
```

然后，将上面生成的 Unicode 编码值解码回相应的文本：

```
# numbers2text.py
#     A program to convert a sequence of Unicode numbers into
#         a string of text.

def main():
    print("This program converts a sequence of Unicode numbers into")
    print("the string of text that it represents.\n")

    # Get the message to encode
    inString = input("Please enter the Unicode-encoded message: ")

    # Loop through each substring and build Unicode message
    message = ""
    for numStr in inString.split():
        codeNum = int(numStr)             # convert digits to a number
        message = message + chr(codeNum) # concatentate character to message

    print("\nThe decoded message is:", message)

main()
```

6.6 字符串的其它方法

图6-14列出了其它常用的字符串方法。下面示例了方法的调用：

function	meaning
s.capitalize()	Copy of s with only the first character capitalized.
s.center(width)	Copy of s centered in a field of given width.
s.count(sub)	Count the number of occurrences of sub in s.
s.find(sub)	Find the first position where sub occurs in s.
s.join(list)	Concatenate list into a string, using s as separator.
s.ljust(width)	Like center, but s is left-justified.
s.lower()	Copy of s in all lowercase characters.
s.lstrip()	Copy of s with leading white space removed.
s.replace(oldsub,newsub)	Replace all occurrences of oldsub in s with newsub.
s.rfind(sub)	Like find, but returns the rightmost position.
s.rjust(width)	Like center, but s is right-justified.
s.rstrip()	Copy of s with trailing white space removed.
s.split()	Split s into a list of substrings (see text).
s.title()	Copy of s with first character of each word capitalized.
s.upper()	Copy of s with all characters converted to uppercase.

图 6-14: 常用的字符串方法

```
>>> str = 'this is an apple'
>>> str.capitalize()
'This is an apple'
>>> str.title()
'This Is An Apple'
>>> str.lower()
'this is an apple'
>>> str.upper()
'THIS IS AN APPLE'
>>> str.replace('this', 'that')
'that is an apple'
>>> str.replace('apple','orange')
'this is an orange'
>>> str
'this is an apple'
>>> str.center(30,'*')
'*******this is an apple*******'
>>> '.'.join(['1','2','3','4','5'])
'1.2.3.4.5'
```

注意，由于字符串对象本身是不可修改的，所以调用 str 对象的方法不会修改 str

对象本身，而只会新生成一个字符串对象。

6.7 列表的 append 方法

与字符串对象不同的是，列表对象是可以修改的。有一个重要的方法，append，它将一个元素添加到列表的末端，例如：

```
squares = []
for x in rage(1, 101):
    squares.append(x*x)
```

大家还应该记得字符串解码程序 number2text.py 吧，其中有一条语句：

```
message = message + chr(codeNum)
```

它是字符串连接操作和复制语句的联合使用。可以想象，随着 message 的长度增加，效率会越来越低。为了避免这个问题，可以使用列表对象及其 append 操作来提高效率，因为列表对象是可修改的，并支持尾端添加元素：

```
# numbers2text2.py
#     A program to convert a sequence of Unicode numbers into
#         a string of text. Efficient version using a list accumulator.

def main():
    print("This program converts a sequence of Unicode numbers into")
    print("the string of text that it represents.\n")

    # Get the message to encode
    inString = input("Please enter the Unicode-encoded message: ")

    # Loop through each substring and build Unicode message
    chars = []
    for numStr in inString.split():
        codeNum = int(numStr)               # convert digits to a number
        chars.append(chr(codeNum))          # accumulate new character
```

```
    message = "".join(chars)
    print("\nThe decoded message is:", message)

main()
```

6.8 日期转换

在一些应用中，需要程序提供某种形式的日期格式转换功能。下面的程序利用了字符串方法进行日期格式的转换：

```
# dateconvert.py
#    Converts a date in form "mm/dd/yyyy" to "month day, year"

def main():
    # get the date
    dateStr = input("Enter a date (mm/dd/yyyy): ")

    # split into components
    monthStr, dayStr, yearStr = dateStr.split("/")

    # convert monthStr to the month name
    months = ["January", "February", "March", "April",
              "May", "June", "July", "August",
              "September", "October", "November", "December"]
    monthStr = months[int(monthStr)-1]

    # output result in month day, year format
    print("The converted date is:", monthStr, dayStr+",", yearStr)

main()
```

6.9 货币处理

在前面的找零钱程序中，结果的显示还不是令人很满意。下面的程序进行了改进：

```
# change2.py
#   A program to calculate the value of some change in dollars
#   This version represents the total cash in cents.

def main():
    print("Change Counter\n")

    print("Please enter the count of each coin type.")
    quarters = int(input("Quarters: "))
    dimes = int(input("Dimes: "))
    nickels = int(input("Nickels: "))
    pennies = int(input("Pennies: "))

    total = quarters * 25 + dimes * 10 + nickels * 5 + pennies

    print("The total value of your change is ${0}.{1:0>2}"
          .format(total//100, total%100))

main()
```

在“0>2”中，“0”是填充字符，“>”表示（本字段）靠右显示，“2”表示（本字段）显示位数。该格式化的效果是：以向右对齐的方式显示 2 位整数，不足 2 位的，以“0”作为填充字符。程序的某个执行结果如下：

```
Change Counter

Please enter the count of each coin type.
Quarters: 1
Dimes: 5
```

```
Nickels: 5
Pennies: 1
The total value of your change is $1.01
```

6.10 文件

在 Python 中，使用文本文件是很方便的，因为可以利用内建函数或对象提供的方法在字符串与各种数据类型之间进行转换。

文本文件的每一行都要有行结束标志符。行结束标识符有多种形式，在 Python 中，使用 “\n” 来表示一行结束。例如，多行文本：

```
Hello
World

Goodbye 32
```

被保存到文件中，其具体内容为：

```
Hello\nWorld\n\nGoodbye 32\n
```

文件的处理需要生成一个文件对象，利用下面的语句可以生成一个文件对象：

```
<variable> = open(<name>, <mode>)
```

其中，<name> 是文件名称，<mode> 表示文件打开的模式，可以是 “r” 或 “w”，前者表示读取文件，后者表示写文件。例如：

```
infile = open("numbers.dat", "r")
```

生成了一个（读）文件对象 infile，后面可以利用文件对象提供的方法读取文件内容。有三种方法可以读取文件内容：

- <file>.read()

  读取整个文件内容，将内容放到一个字符串对象中；

- <file>.readline()

  读取文件的一行，将内容放到一个字符串对象中；

- <file>.readlines()

  读取整个文件内容，将内容以列表的形式放到一个列表对象中，列表的每一个元素是文件的每一行；

下面的程序，使用 read 方法读取指定文件并显示：

```
# printfile.py
#     Prints a file to the screen.

def main():
    fname = input("Enter filename: ")
    infile = open(fname,"r")
    data = infile.read()
    print(data)

    infile.close()

main()
```

下面的程序，使用 readline 完成同样的功能：

```
# printfile2.py
#     Prints a file to the screen.

def main():
    fname = input("Enter filename: ")
    infile = open(fname,"r")
    line = infile.readline()
    while line != '':
        print(line[:-1])
        line = infile.readline()

    infile.close()
```

```
main()
```

程序遍历文件的每一行：使用 print 语句输出。注意，line[:-1] 去掉了换行符“\n”，因为缺省情况下，print 会输出一个换行符。你也可以使用语句 print(line, end=“”) 代替语句 print(line[:-1])。

input 函数从用户获取一行（文本）数据，readline 方法从文件读取一行（文本）数据。但是，要注意的是，前者丢掉了换行符，而后者保留了换行符。

下面的程序，使用了方法 readlines，功能完全一样：

```
# printfile3.py
#     Prints a file to the screen.

def main():
    fname = input("Enter filename: ")
    infile = open(fname,"r")
    for line in infile.readlines():
        print(line[:-1])

    infile.close()

main()
```

此程序一次从文件中读取所有的行，将内容放到一个列表对象中。它的缺点是，当文件很大时，将消耗过多的内存。一个解决办法是：

```
# printfile4.py
#     Prints a file to the screen.

def main():
    fname = input("Enter filename: ")
    infile = open(fname,"r")
    for line in infile:
        print(line[:-1])
```

```
    infile.close()

main()
```

该程序将文件对象当作一个文本行的集合（流），使用 for 循环遍历文件的每一行。

写文件时，可以指定打开模式为 “w”，并使用 print 指定要写入的文件对象：

```
# printfile5.py
#     Prints string to a file.

def main():
    fname = input("Enter filename: ")
    outfile = open(fname,"w")

    print("Hello, World!", file = outfile)
    print(outfile.name, file = outfile)

    outfile.close()

main()
```

还记得前面的 username.py 程序吗？它从用户输入的名字中取第 1 个字符，从姓中取前 7 个字符，然后将这 2 部分连接在一起，生成一个新用户名。

下面这个程序以 “批处理” 的方式从输入文件中生成用户名并保存到输出文件中。该输入文件的每一行包含一个用户（由 First Name 和 Last Name 组成，两者之间有空格）。输出文件的每一行是一个新生成的用户名。程序如下：

```
# userfile.py
#    Program to create a file of usernames in batch mode.

def main():
    print("This program creates a file of usernames from a")
```

```
    print("file of names.")

    # get the file names
    infileName = input("What file are the names in? ")
    outfileName = input("What file should the usernames go in? ")

    # open the files
    infile = open(infileName, "r")
    outfile = open(outfileName, "w")

    # process each line of the input file
    for line in infile:
        # get the first and last names from line
        first, last = line.split()
        # create the username
        uname = (first[0]+last[:7]).lower()
        # write it to the output file
        print(uname, file=outfile)

    # close both files
    infile.close()
    outfile.close()

    print("Usernames have been written to", outfileName)

main()
```

在应用程序的运行过程中，有时需要用户选择一个输入文件或输出文件。此时，可以利用 Python 库 tkinter 提供的文件对话框，允许用户选取合适的文件：

```
>>> from tkinter.filedialog import askopenfilename, asksaveasfilename
>>> infilename = askopenfilename()
>>> outfilename = asksaveasfilename()
```

最后 2 条语句将分别打开文件打开对话框和文件保存对话框。如果用户选择了“取消”按钮，则返回值为''，否则，返回值为完整的文件名称。

6.11 练习

1. 编写一个程序，将百分制成绩转换成 A、B、C、D、E 五级制成绩；
2. 编写一个程序，将一个短语中每个单词（空格隔开）的首字母组合成一个缩略语（全部大写）；
3. 编写一个程序，计算字符串中所有 a(A)-z(Z) 所对应的数值之和。例如，“a” 或 “A” 的数值是 1，“b” 或 “B” 的数值是 2，...，“z” 或 “Z” 的数值是 26。如果有一个字符串是'ABC Zelle'，那么它的数值之和是：1+2+3+26+5+12+12+5 = 66；
4. 编写一个程序，输入一个文本文件，使用凯撒加密法[11]进行加密，然后输出到另一个文本文件中，再编写一个解密程序，验证能否正确解密；
5. 编写一个程序，统计一个文本文件中单词的个数，然后输出（单词-个数对）到另一个文本文件中；
6. 编写一个程序，统计一个文本文件中单词的平均长度；
7. 编写一个改进的 chaos.py 程序，允许用户输入 2 个初始值和迭代次数，然后按图6-15所示的方式输出数据；
8. 编写一个改进的 futval.py 程序，提示用户输入投资金额、年利率和投资年数，然后按图6-16所示的方式输出数据；
9. 编写一个程序，按图6-17所示绘制一个学生成绩的水平柱状图。程序应该从输入文件中读取数据。该输入文件的第 1 行包含学生个数，随后的每一行包含学生的姓名及某门课程的分数（0-100）；
10. 编写一个程序，按图6-18所示方式生成一个垂直柱状图。程序应该从输入文件中读取数据。输入文件的每一行包含一个 0-10 的数据；

[11] 它使用所谓的偏移替换法进行字符替换。例如，假设偏移值 (称为 key) 为 2，那么 “A” 将被 “C” 替换，“B” 将被 “D” 替换，...，“Z” 将被 “B” 替换。小写字符的处理，依此类推。

index	0.25	0.26
1	0.731250	0.750360
2	0.766441	0.730547
3	0.698135	0.767707
4	0.821896	0.695499
5	0.570894	0.825942
6	0.955399	0.560671
7	0.166187	0.960644
8	0.540418	0.147447
9	0.968629	0.490255
10	0.118509	0.974630

图 6-15: 改进的 chaos 输出

Year	Value
0	$2000.00
1	$2200.00
2	$2420.00
3	$2662.00
4	$2928.20
5	$3221.02
6	$3542.12
7	$3897.43

图 6-16: 改进的 futval 输出

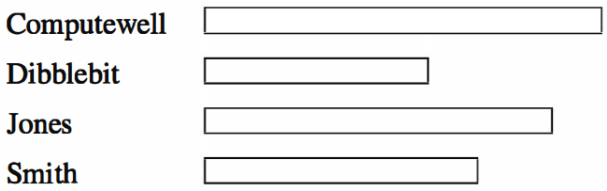

图 6-17: 学生成绩的水平柱状图

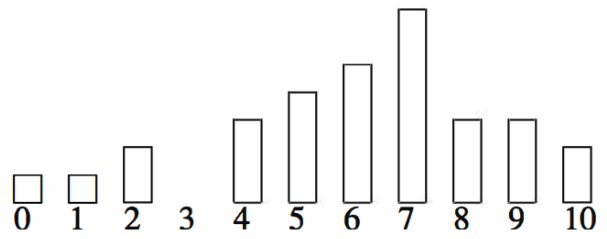

图 6-18: 统计柱状图示例

7 函数与对象

7.1 函数

7.1.1 定义

函数的定义如下：

```
def <name>(<formal-paramters>):
    <body>
    <return <value>>
```

函数名称 <name> 必须是一个合法的标识符，<formal-parameters> 是形式参数序列（0、1、...、n 个元素），return 语句是可选的。

函数的调用格式如下：

```
<name>(<actual-parameters>)
```

<actual-parameters> 是实际参数序列（0、1、...、n 个元素）。

7.1.2 设计原则

下面的程序，定义了几个函数，每个函数的功能都比较明确，整个程序的结构显得很清晰。函数带来的最直接的好处是代码复用：

```
# Program: triangle2.py
import math
from graphics import *

def square(x):
    return x * x

def distance(p1, p2):
    dist = math.sqrt(square(p2.getX() - p1.getX())
                     + square(p2.getY() - p1.getY()))
    return dist
```

```
def main():
    win = GraphWin("Draw a Triangle")
    win.setCoords(0.0, 0.0, 10.0, 10.0)
    message = Text(Point(5, 0.5), "Click on three points")
    message.draw(win)

    # Get and draw three vertices of triangle
    p1 = win.getMouse()
    p1.draw(win)
    p2 = win.getMouse()
    p2.draw(win)
    p3 = win.getMouse()
    p3.draw(win)

    # Use Polygon object to draw the triangle
    triangle = Polygon(p1,p2,p3)
    triangle.setFill("peachpuff")
    triangle.setOutline("cyan")
    triangle.draw(win)

    # Calculate the perimeter of the triangle
    perim = distance(p1,p2) + distance(p2,p3) + distance(p3,p1)
    message.setText("The perimeter is: {0:0.2f}".format(perim))

    # Wait for another click to exit
    win.getMouse()
    win.close()

main()
```

先阅读下面的程序：

```
# happy.py

def happy():
    print("Happy birthday to you!")

def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + ".")
    happy()

def main():
    sing("Fred")
    print()
    sing("Lucy")
    print()
    sing("Elmer")

main()
```

再阅读下面这个程序。将上下两个程序做一个比较，你能得出什么结论？

```
# happy2.py
#     Happy Birthday using value returning functions

def happy():
    return "Happy birthday to you!\n"

def sing(person):
    lyrics = happy()*2 + "Happy birthday, dear " + person + ".\n" + happy()
    return lyrics

def main():
```

```
    for person in ["Fred", "Lucy", "Elmer"]:
        print(sing(person))

main()
```

这 2 个程序完成的功能完全一样，但是第 2 个程序要明显优雅许多，原因何在？再仔细阅读这 2 个程序，你会发现，即便都使用了函数作为基本模块，程序的优雅与优化程度仍然会有所不同。在第 1 个程序中，sing 函数的调用出现了 3 次，happy 函数的调用出现了 3 次，print 函数的调用出现了 4 次。而在第 2 个程序中，它们的调用次数分别为 1 次、2 次、1 次。第 2 个程序使用循环结构替换了（多次重复出现的）顺序结构，巧妙利用了字符串的连接语句替换了多条 print 语句。语句的重复次数减少了，结构也优化了，代码也就自然变得更优雅高效了。

与写作原则类似，好的程序设计原则是：

- 代码尽量要复用——清晰 (Clarity)、简洁 (Simplicity)；
- 避免语句反复出现——效率 (Efficiency)，当然还包括运行与空间效率；
- 多使用优雅的循环结构——优雅 (Elegance)；
- 通用性 (Generality) 与可扩展性 (Scalability)；
- 如果涉及到字符串的打印输出，最好使用 return 返回字符串，而不是在函数中使用 print 函数直接输出字符串。

  这样做的好处是，调用者自己可以决定字符串的处理方式（是输出到屏幕，还是输出到文件等）。这一设计原则同样也适用于其它类似的情形；

前 2 条技巧似乎互相矛盾。实际上，正确的理解应该是，在代码复用的前提下，尽量避免语句反复出现。记住，好的算法与程序就像诗那样，令人赏心悦目，回味无穷。

程序 happy2.py 还有一个优点，即以字符串对象作为统一的数据接口，便于功能的实现。而在程序 happy.py 中，将数据与操作混合在函数 print 中：print 分散在各处，不便于修改。例如，如果需要将结果输出到文件中，那么对程序 happy2.py 来讲，可以很容易地做到这一点，只需要修改 main 函数：

```
def main():
    outf = open("Happy_Birthday.txt", "w")
    for person in ["Fred", "Lucy", "Elmer"]:
        print(verseFor(person), file=outf)
    outf.close()
```

有时候，需要函数返回多个结果，此时可以在 return 语句中使用 “,” 将多个结果分隔开，例如：

```
def sumDiff(x, y):
    sum = x + y
    diff = X - y
    return sum, diff
```

调用函数 sumDiff 时，需要使用 “同时赋值” 来接收返回的多个结果：

```
num1, num2 = input("Please enter two numbers (num1, num2) ").split(",")
s, d = sumDiff(float(num1), float(num2))
print("The sum is", s, "and the difference is", d)
```

技术上讲，所有的 Python 函数都将会返回至少一个值，即使是在定义函数时，程序员没有显式地添加 return <value> 语句——没有 return 语句的函数将缺省返回 None 值。None 是一个特殊的 Python 对象，通常用来表示变量没有任何值。请看下面的程序：

```
def distance(p1, p2):
    dist = math.sqrt(square(p2.getX() - p1.getX()) +
           square(p2.getY() - p1.getY()))
perim = distance(p1, p2) + distance(p2, p3) + distance(p3, p1)
```

在运行时，Python 将会弹出错误提示：

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

原因是，distance 的函数体缺少语句 “return dist”，Python 会缺省地让该函数返回 None 值，因而也就会导致 perim 表达式中出现 “None+None+None” 的错误求值表达式。

递归函数也是一个重要的内容。在函数内部，如果出现直接调用或间接调用本函数的情况，该函数就是递归函数。例如，可以利用递归函数求解阶乘、汉诺塔等问题。理论上，所有的递归函数都可以写成循环形式，但是逻辑不如递归函数清晰。一个值得注意的问题是，要防止递归函数出现“栈溢出”——函数调用是通过栈来实现的，每调用一次，栈深度就加深一次，由于栈的大小不是无限的，所以当递归深度超过极限时，会导致栈溢出[12]。

7.1.3 函数参数

函数的参数类型有比较丰富，包括：位置参数、默认参数、可变参数和关键字参数。

位置参数

所谓的位置参数，无论在函数定义时，还是在函数调用时，都必须遵循严格的格式：定义顺序与调用顺序要严格匹配，或者说，形式参数与实际参数要严格一一对应。这是最常用的函数参数形式，在此不再赘述。

默认参数

默认参数与 C++ 中的定义方式一样，并且也是要放在必选参数 (非默认的位置参数) 之后。例如：

```
1  def PrintInfo(name, age, gender = 'M', major = 'Computer Science'):
2      print('name', name, end=' ')
3      print('age', age, end=' ')
4      print('gender', gender, end=' ')
5      print('major', major, end=' ')
6      print()
7
8  PrintInfo('John', 20)
9  PrintInfo('Zodd', 22, major = 'Physics')
10 PrintInfo('Lili', 21, 'F', 'Mathematics')
```

[12]可以使用“尾递归函数”的优化技术来解决栈溢出问题。尾递归函数在返回时只能调用自身，并且不能与其它的表达式组合在一起。这样，编译器/解释器可以针对尾递归函数执行优化——无论递归调用多少次，都只需占用一个栈，不会出现栈溢出问题。

程序的运行结果：

```
name John age 20 gender M major Computer Science
name Zodd age 22 gender M major Physics
name Lili age 21 gender F major Mathematics
```

可以看出，函数有默认参数时，Python 中的调用形式更加灵活：第 8 行，与 C++ 的调用方式一样；第 9 行，默认参数没有按照指定顺序调用，但是需要提供参数名称。

需要特别注意的是，一般情况下，不要把可变对象作为默认参数，详见7.2.2中的函数my_function 定义。

一个基本原则是，在允许的情况下，编写程序时尽量使用不可变对象：避免修改数据导致的隐形错误；多任务环境下，该类型对象的同时读取不存在任何问题，不需要上锁。

可变参数

在可变参数情况下，传入参数的个数是可变的：0、1、2、...。

假设需要编写一个函数，计算一组数据的和，如果使用列表或元组 (详见10.1) 作为数据结构，程序如下：

```
1 def sum(numbers):
2     result = 0
3     for num in numbers:
4         result = result + num
5     return result
6 
7 l = [1,2,3,4,5]
8 print(sum(l))
9 print(sum((1,2,3,4,5)))
```

注意，在每次调用时，需要传入列表或元组对象，形式上比较繁琐。有一种简化方法——使用可变参数：

```
1 def sum(*numbers):
2     result = 0
```

```
    for num in numbers:
        result = result + num
    return result

l = [1,2,3,4,5]
print(sum(*l))
print(sum(1,2,3,4,5))
```

从函数的定义来看，两者的唯一区别仅仅是，参数前加了一个“*”，其余代码没有任何变化。可变参数的调用也得到了简化：如果要传入列表或元组，可以在对象前面添加“*”；如果是一组数据，直接当作一般参数传入即可[13]。

关键字参数

与可变参数一样，关键字参数可以与其它类型参数一起混用——可变参数转换成元组，而关键字参数转换成字典，请看下例：

```
def PrintInfo(name, age, **kp):
    print('name', name, end = ' ')
    print('age', age, end = ' ')
    for key, value in kp.items():
        print(key, value, end = ' ')
    print()

PrintInfo('Lee', 19)
PrintInfo('John', 20, gender = 'M', major = 'Computer Science')
PrintInfo('Zodde', 22, gender = 'M', major = 'Physics')
PrintInfo('Lili', 21, gender = 'F', major = 'Mathematics')
d = {'Weight':100, 'Height':170, 'Interest':'football'}
PrintInfo('Zhang', 18, **d)
```

运行结果如下：

[13]实际上，在函数调用时，可变参数自动组装成一个元组对象。可变参数可以与其它类型参数一起混用。

```
name Lee age 19
name John age 20 major Computer Science gender M
name Zodde age 22 major Physics gender M
name Lili age 21 major Mathematics gender F
name Zhang age 18 Interest football Height 170 Weight 100
```

程序的第 1 行，kp 是关键字参数，它接收 0、1、2、... 个有参数名的参数。注意，如果已经存在一个字典对象，可以在它的名称前使用 “**”，将它作为关键字参数传入函数，参见第 12-13 行[14]。

如果要限制关键字参数为指定的名称，可以指定关键字的名称 (命名关键字参数):

```
1  def PrintInfo(name, age, *, gender, major):
2      print('name', name, end = ' ')
3      print('age', age, end = ' ')
4      print(gender, major, end = ' ')
5      print()
6
7  #PrintInfo('Lee', 19)
8  PrintInfo('John', 20, gender = 'M', major = 'Computer Science')
9  PrintInfo('Zodde', 22, gender = 'M', major = 'Physics')
10 PrintInfo('Lili', 21, gender = 'F', major = 'Mathematics')
11 #d = {'Weight':100, 'Height':170, 'Interest':'football'}
12 #PrintInfo('Zhang', 18, **d)
```

程序第 1 行，使用 “*” 分割了位置参数与关键字参数，并在后面指定了关键字的名称为 “gender” 和 “major”。(屏蔽的) 第 7 行和第 11-12 行，调用函数时没有传入指定名称的关键字参数，执行时会提示错误。

小结：在定义函数时，可以使用必选 (位置) 参数、默认参数、可变参数、关键字参数和命名关键字参数等 5 种参数，它们可以组合使用。但是，参数必须按如下顺序定义：必选参数、默认参数、可变参数、命名关键字参数和关键字参数。为提高接口定义的可读性，请尽量保持参数组合的简单性与易理解性。

[14] 传入的是该字典对象的克隆。

7.2 对象

从设计伊始，Python 就被定位为一门面向对象的语言，它的重要思想是“一切皆对象”：数字、字符串、元组、列表、字典、函数、方法、类、模块等都是对象，甚至包括你的代码。

什么是对象？不同的编程语言有不同的理解与处理方式。在某些语言中，它意味着所有的对象必须有属性和方法；而在另一些语言中，它意味着所有的对象都可以子类化。

在 Python 中，对象的定义是非常宽泛的——一切皆对象，某些对象既可以没有属性也可以没有方法，而且并不是所有的对象都可以子类化。实际上，“Python 一切皆对象” 和动态语言这 2 个重要特点可以使得 Python 能够以一种统一的方式来处理所有对象——任何对象都可以参与赋值并作为参数进行传递。在讨论可变的 (mutable) 和不可变的 (immutable) 对象之前，先来讨论 Python 中对象的性质及相关函数，这对正确理解它们非常重要。

7.2.1 对象的属性

在 Python 中，所有的对象都有三个属性：

- ID 号/身份号

  每个对象都有一个唯一的身份号来标识自己，任何对象的身份号都可以使用内建函数 id() 来得到，可以简单地认为它就是该对象的内存地址。

```
>>> a=3
>>> id(a)
1681355472
>>> id(3)
1681355472
>>> id(4)
1681355504
>>> b=4
>>> id(b)
1681355504
```

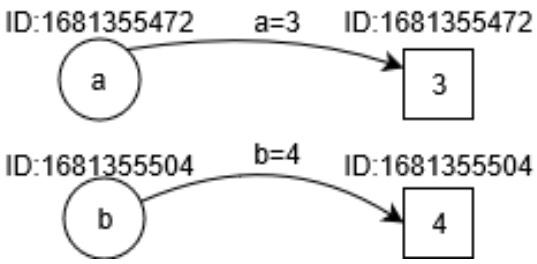

图 7-19: 对象 a 与 b 的内存模型示意图 1

如图7-19所示，3 和 4 都是对象，都有自己的 ID 号，执行 a=3 后，将对象 3 的 ID 号复制给了对象 a，此时，对象 a 就是对象 3 的别名对象[15]。

继续执行下面的语句，结果如图7-20所示。可以看出，变量 a 把自己的 ID 号（即对象 3 的 ID 号）复制给了变量 b，使得 b 也指向了对象 3：

```
>>> b=a
>>> id(b)
1681355472
```

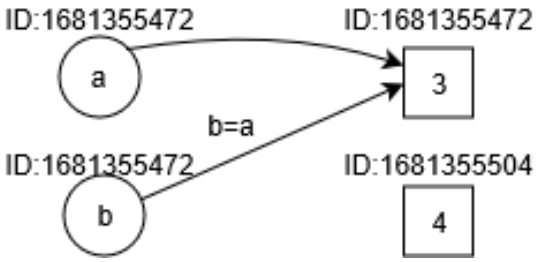

图 7-20: 对象 a 与 b 的内存模型示意图 2

继续执行语句 b=2，此时，对象的内存模型示意图如图7-21所示。

```
>>> b=2
>>> id(b)
1681355440
>>> id(a)
1681355472
```

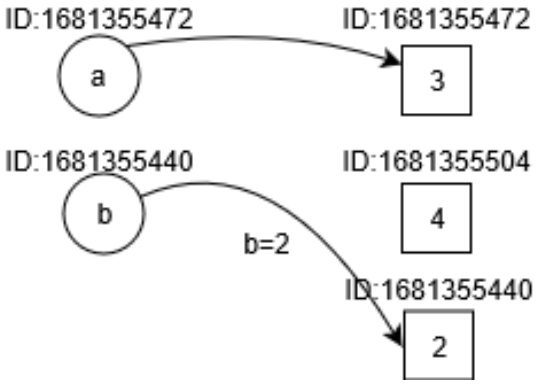

图 7-21: 对象 a 与 b 的内存模型示意图 3

[15]但并不是说，修改了对象 a，就等价于修改了对象 3，或者反过来，修改了对象 3，就等价于修改了对象 a。例如，假如再次执行 a=4，Python 会将对象 4 的 ID 号复制给对象 a，而并不是真正地修改对象 a 原来指向的那个对象（此例为对象 3，而且对象 3 也不是可以修改的类型，它是不可变的 (immutable)）。除非是可变的 (mutable) 对象，否则，在赋值时，Python 只是简单地修改它的 ID 号，而不会真正地去修改它（的内部元素）。因此，这种情况下的“别名对象”不是传统意义上的“别名对象”概念了，它是与“可变/不可变对象”概念相关的。关于“可变与不可变对象”的概念，将在下面详细讨论。

实际上，从 Python 的角度看，上面的对象都是不可变的 (immutable) 对象，即它们是不能被修改的，即使执行诸如 “a=3/a=4、c=a/a=b/b=c” 这样的语句序列，表面上看，它们可以被修改。但是，在 Python 内部，它们的处理方式与其它语言的处理方式截然不同，即它们的对象内存模型有很大的区别。正确理解对象的内存模型以及可变与不可变性，对于编写合格高效的 Python 程序非常重要。

继续执行下面的语句。对象的内存模型示意图如图7-22所示。

```
>>> L1=[1,2,3,4]
>>> id(L1)
2373690431560
>>> L2=[1,2,3,4]
>>> id(L2)
2373690431496
>>> L3=L1
>>> id(L3)
2373690431560
>>> L3[0]=5
>>> L3
[5, 2, 3, 4]
>>> L1
[5, 2, 3, 4]
>>> id(L3)
2373690431560
>>> id(L1)
2373690431560
```

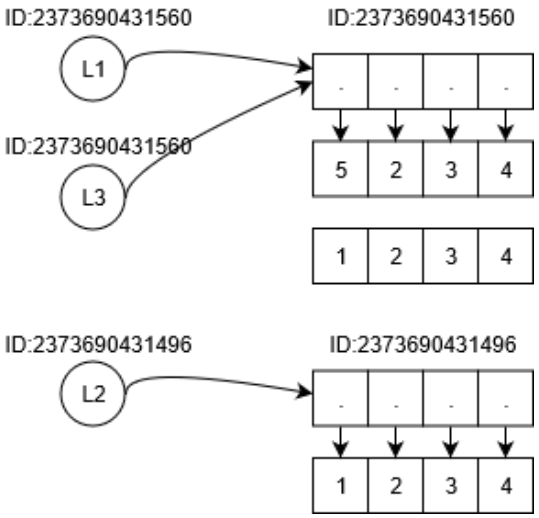

图 7-22: 对象 a 与 b 的内存模型示意图 4

从该图可以看出，在开始阶段，即使 L1 与 L2 的元素个数与内容完全一样，它们也有不同的 ID 号，是不同的对象。L3=L1 语句使得 L3 是 L1 的别名对象[16]，在执行 L3[0]=5 之后，L3 与 L1 一起发生了改变，因为 L1 是列表对象，是可变对象，而 L3 是 L1 的别名对象。

[16]L3 是 L1 真正的别名对象，只要一方发生改变，另一方也会随之改变，因为它们都是可变的对象。

继续执行下面的 Python 语句。可以发现，str1 和 str2 实际上是同一个对象，因为它们的值都一样，且'test'是字符串对象，是不可变对象。

```
>>> str1='test'
>>> id(str1)
2747844700792
>>> str2='test'
>>> id(str2)
2747844700792
```

总之，可变对象与不可变对象的行为有着本质的区别，它们对程序的效率、函数的参数传递方式有着很大的影响，详见下文。

- 类型

  类型决定了对象的值的类型、属性以及方法。可以使用内建函数 type 来查看对象的类型：

```
>>> a=3
>>> type(a)
<class 'int'>
>>> b='test'
>>> type(b)
<class 'str'>
>>> L1=[1,2,3,4]
>>> type(L1)
<class 'list'>
```

- 值

  值，即对象所表示的数据。

上述三个属性，随着对象的创建而产生。如果对象支持更新操作，那么它的值是可变的 (mutable)，否则为只读或不可变的 (immutable，例如，数字、字符串、元组等都属于不可变类型)。

Python 的“一切皆对象”是有技术保证的。从技术上来讲，每一个对象都有两个标准的头部信息：一个是类型标识符，用于标识对象的类型；另一个是引用计数器，用于确定对象是否需要回收[17]。另外，Python 也缓存了某些不变的对象，以便复用它们，而不需要每次创建新的对象（从上述对象的 id 操作可以看出）。

7.2.2 可变性与不可变性

在 Python 中，有些对象是可变的 (mutable)，有些对象是不可变的 (immutable)。可变的对象，意味着可以对它进行修改；而不可变的对象，是不能被修改的，当试图改变它们时，Python 将会返回一个新对象。显然，这 2 类对象的行为截然不同。可变与不可变性会对程序的运行效率、函数参数的行为方式都将产生重要的影响。

表7-1和表7-2分别列出了不可变对象和可变对象。

表 7-1: 不可变的对象

int	float	decimal
complex	bool	string
tuple	range	frozenset
bytes		

表 7-2: 可变的对象

list	dict	set
bytearray	自定义类 (除非改变)	

一般来讲，容器类和用户自定义类是可变的，而标量类型几乎总是不可变的。当然，有一些例外，例如，tuple 和 frozenset 是不可变的，string 也是不可变的（如果想要字符串具备 in-place 的可变功能，可以使用 bytearray 对象）。

了解对象的可变性与不可变性有什么作用？下面请看一个字符串连接程序：

```
string_build = ""
for data in container:
    string_build += str(data)
```

该程序的问题在于：在 for 循环体内，迭代执行字符串的连接操作，而string_build是不可变的对象——每次迭代时，Python 将 2 个 string 对象进行连接并生成一个

[17]可以通过 sys 模块中的 getrefcount() 函数查询引用计数器的值，例如，import sys、sys.getrefcount(1)。

新的 string 对象。因此，如果 for 循环迭代许多次，将会生成一个很大的 string 对象，更为严重的是，在迭代过程中将会生成很多个临时 string 对象，代价较大。

下面的三段代码用不同的方式避免了这个问题：

```
builder_list = []
for data in container:
    builder_list.append(str(data))
"".join(builder_list)

### Another way is to use a list comprehension
"".join([str(data) for data in container])

### or use the map function
"".join(map(str, container))
```

这些代码充分利用了 list 对象的可变性，在迭代过程中，将不会生成大量的临时对象。

当然，我们也要注意对象可变性带来的隐形问题，请看下面的程序：

```
def my_function(param=[]):
    param.append("thing")
    return param

my_function() # returns ["thing"]
my_function() # returns ["thing", "thing"]
```

程序的运行结果可能并不如你所愿，原因在于：

- Python 只定义函数一次；
- Python 把缺省的参数当作函数定义的一部分；
- Python 在所有的函数调用中会共享可变对象；

因此，不要把可变对象当作函数参数的缺省值（例如，上例中的 [ ]，它是一个列表，可变对象），正确的做法是使用不可变对象作为缺省值，改进后的程序如下：

```
def my_function2(param=None):
    if param is None:
        param = []
    param.append("thing")
    return param
```

另外，还需注意一个问题，一个对象是不可变的，并不意味着它的元素也是不可变的，例如：

```
>>> t = ('holberton', [1, 2, 3])
>>> t[0]
'holberton'
>>> t[1]
[1, 2, 3]
>>> t[1].append(4)
>>> t[1]
[1, 2, 3, 4]
>>> t
('holberton', [1, 2, 3, 4])
```

可以看出，t 对象的第 2 个元素是一个 list 对象，它本身是一个可变对象，因而可以对它进行修改。

综上所述，Python 以不同的方式来处理可变对象与不可变对象，下面是几个选用原则：

1. 在只读应用中，不可变对象的访问速度要快于可变对象；
2. 在读写数据的情况下，使用可变对象；在只读的情况下，使用不可变对象；
3. “更新”不可变对象会导致新对象的创建，代价大；更新可变对象不会创建新对象 (称为 in-place 或原地修改方式)，代价小；
4. 在处理函数的缺省值时，不要使用可变对象，而要使用不可变对象；
5. 将可变对象与不可变对象当作实际参数进行传递时，产生的效果是不一样的，详见7.3；

7.3 参数传递

我们先来看一个程序：

```
# addinterest1.py
def addInterest(balance, rate):
    print('addInterest:balance', id(balance))
    newBalance = balance * (1+rate)
    balance = newBalance

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
    print('test:amount', id(amount))

test()
```

在终端下运行该程序，将获得类似的结果：

```
> python addinterest1.py
addInterest:balance 1965929877392
1000
test:amount 1965929877392
```

从结果可以看出，函数 test 和函数 addInterest 中的 amount 和 balance 对象拥有一样的 ID 号，而且函数 addInterest 计算的结果并没有返回到函数 test 中。如果学过其它的程序设计语言，例如 C++，结果并不令人感到意外。但是，原因并不一样：

1. 在 Python 中，参数的传递方式只有一种，即传值方式。而在 C++ 这样的语言中，除了传值方式外，还有传递引用的方式；

2. amount 和 balance 对象是 int 对象，属于不可变对象，意味着，在调用函数 addInterest（实参形参相结合）时，它们都指向了对象 1000，执行语句

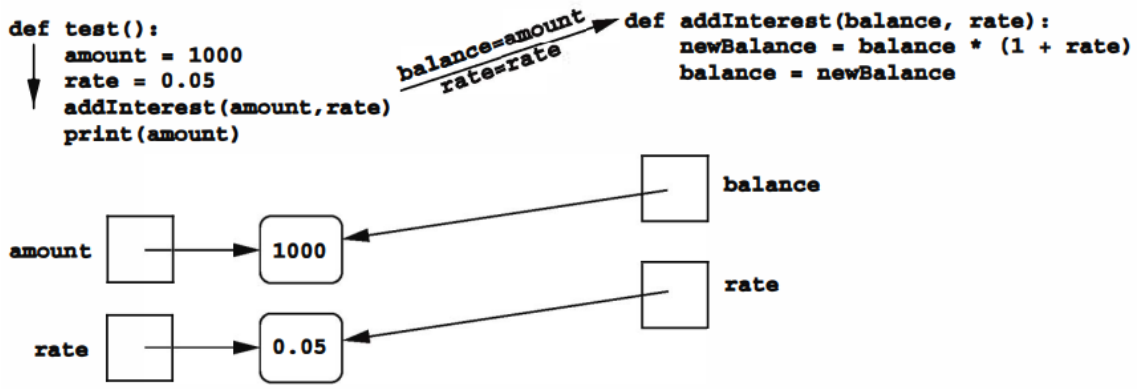

图 7-23: addInterest 的内存模型示意图 1

balance=newBalance（试图更新不可变对象）将导致 Python 创建一个新对象 1050（或理解为 balance 指向了另一个对象 1050）。此时，amount 和 balance 这 2 个对象已经毫无联系了。它们的内存模型示意图如图7-23和7-24所示；

既然 Python 只支持参数的传值方式，那么有没有办法将函数的计算结果返回？有 2 种办法：

1. 利用 return 语句直接返回结果；
2. 利用函数的参数“带出”结果——要利用可变对象；

请看下面的改进程序：

```
# addinterest2.py
#   t is tuple object
def addInterest(t):
    t[1][0] = t[1][0] * (1 + t[1][1])

def test():
    t = ('balance', [1000, 0.05])
    addInterest(t)
    print(t[1][0])

test()
```

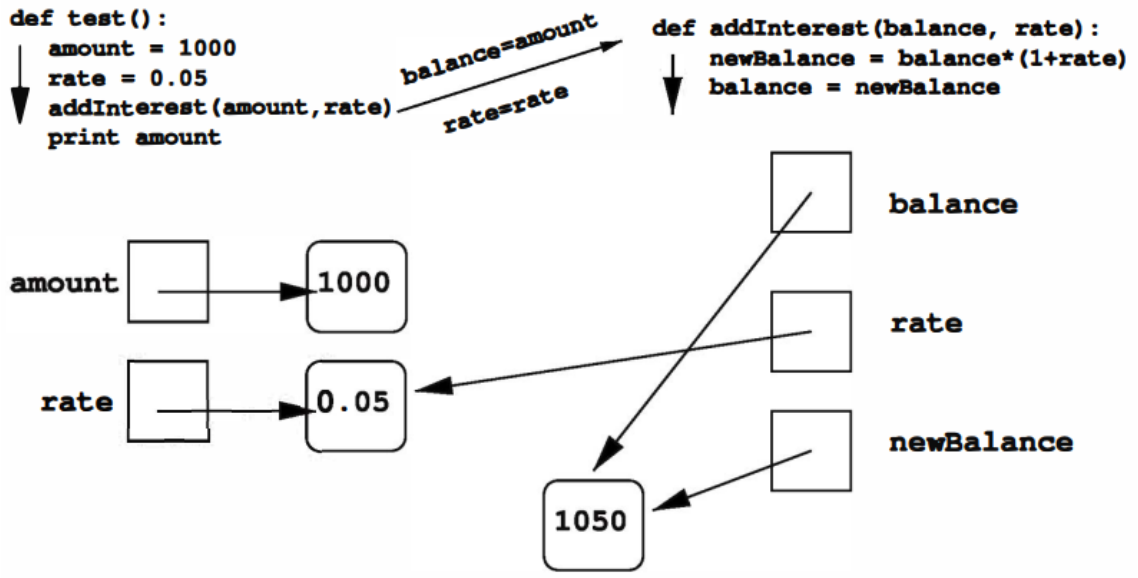

图 7-24: addInterest 的内存模型示意图 2

注意，虽然 t 是一个 tuple 对象（不可变对象），但是它的第 2 个元素是一个 list 对象（可变对象），因此，该程序能够实现指定的功能。内存操作示意图如图7-25所示。

可以进一步改进程序，支持多用户账户的计算：

```
# addinterest3.py

def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)

def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, rate)
    print(amounts)

test()
```

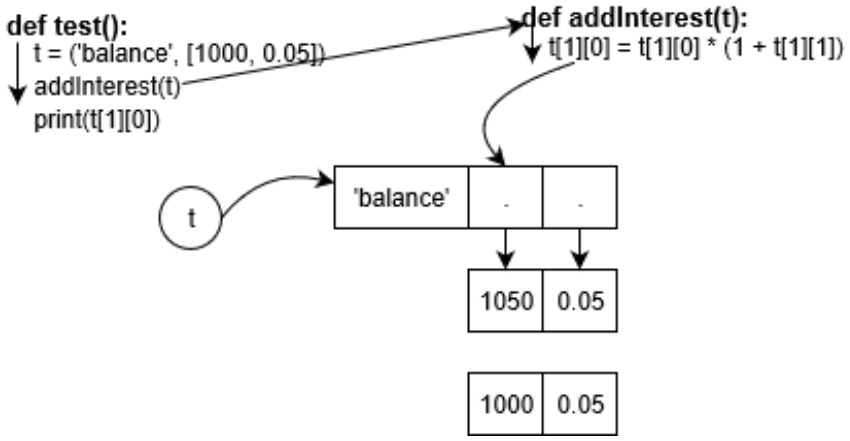

图 7-25: addInterest 的内存模型示意图 3

本程序的内存模型示意图如图7-26所示。

综上所述，Python 采用统一的方式支持函数的参数传递（传值方式——传递对象的 ID 号），将对象分成可变与不可变对象，有效地解决了对象的带值返回问题。

8 选择结构与循环结构

8.1 选择结构

if 语句的格式如下：

```
if <condition>:
    <body>
```

语义非常明确：<condition> 为 True，执行 <body>；否则，跳过 <body>，直接执行后续语句。

其中，<condition> 是条件表达式，可以使用关系表达式和布尔 (逻辑) 表达式，其结果为 True 或 False。简单的条件表达式形式为：<expr> <relop> <expr>，其中，<relop> 表示关系运算符 (Relational Operator)。Python 有 6 种关系运算符，如图8-27所示。

一般来说，我们有 2 种方式可以使用 Python 程序：作为脚本直接运行，例如，假设程序名称为 test.py，那么在 CMD 命令行下输入 “python test.py”，以运行该程序；或者，输入 “import test”，将其作为库导入。

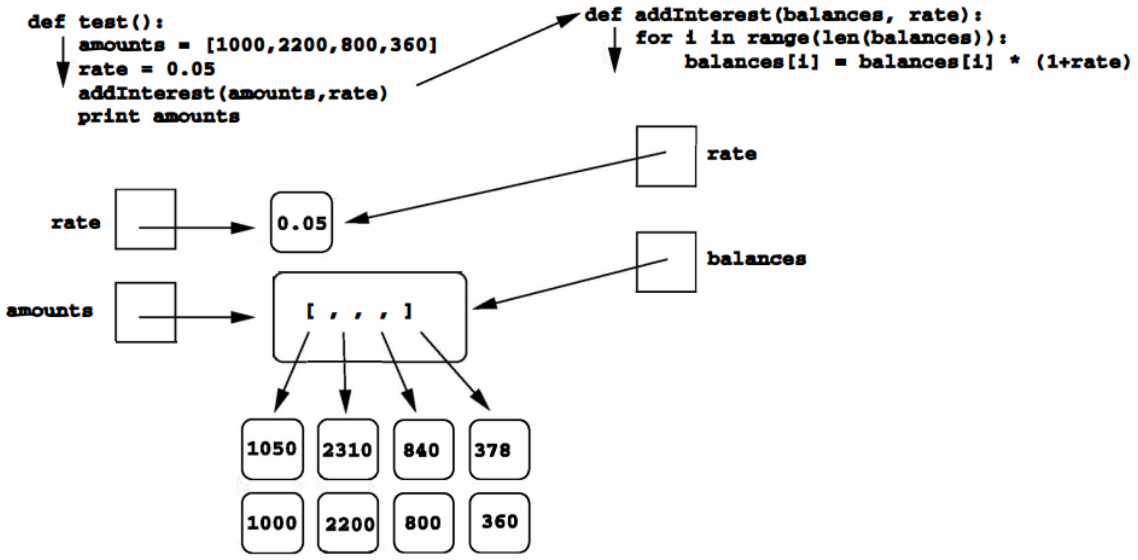

图 7-26: addInterest 的内存模型示意图 4

Python	mathematics	meaning
<	<	less than
<=	≤	less than or equal to
==	=	equal to
>=	≥	greater than or equal to
>	>	greater than
!=	≠	not equal to

图 8-27: 关系运算符

假设 test.py 的内容如下：

```
# test.py

print('__name__ ==', __name__)

def main():
    print('main function')

if (__name__ == '__main__'):
    main()
```

对于该程序，有时候想把它当作脚本直接运行，有时候又想把它当作库导入。实际上，该程序的末尾 2 行，

```
if __name__ == '__main__':
    main()
```

就能够起到这个作用：如果输入 “python test.py”，那么 “__name__” 的值被设置为 “__main__”，此时条件成立，将会调用 main() 函数；否则，如果输入 “import test”，那么 “__name__” 的值将被设置为 “test”，此时条件不成立，不会调用 main() 函数，而只会导入程序的其它部分。因此，我们可以得到 2 种运行结果：

- 在 CMD 终端下，运行命令 python test.py，将得到如下结果：

  ```
  __name__ == __main__
  main function
  ```

- 在 CMD 终端下，运行命令 python，然后在 python 终端下，输入 import test，将得到如下结果：

  ```
  __name__ == test
  ```

这是一个典型的 if 语句的应用案例——依据不同的条件，执行与导入程序的不同部分，以满足特定的要求。

上面的 if 语句被称为单路 (one-way) 选择结构。Python 中，还有一种 if-else 选择结构，可以执行双路 (two-way) 选择：

```
if <condition>:
    <statements>
else:
    <statements>
```

下面的方程求根程序，使用了 if-else 结构：

```
# quadratic3.py
import math

def main():
    print("This program finds the real solutions to a quadratic\n")
    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discrim = b * b - 4 * a * c
    if discrim < 0:
        print("\nThe equation has no real roots!")
    else:
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2)
main()
```

虽然该程序解决了实根有无的判断问题，但是它的表现并不完美。例如，运行程序并输入：

```
>>> main()
This program finds the real solutions to a quadratic
Enter coefficient a: 1
Enter coefficient b: 2
```

```
Enter coefficient c: 1
The solutions are: -1.0 -1.0
```

可以看出，方程有 2 个相同的实根，程序也照样输出了 2 次。实际上，程序应该针对这个特殊情况，进行专门的处理：可以使用多路 (multi-way) 选择结构。

多路选择结构 (if-elif-else) 的语法如下：

```
if <condition1>:
    <statements>
elif <condition2>:
    <statements>
elif <condition3>:
    <statements>
else:
    <default statements>
```

注意，else 子句是可选的。

改进的方程求根程序如下：

```
# quadratic4.py
import math

def main():
    print("This program finds the real solutions to a quadratic\n")
    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discrim = b * b - 4 * a * c
    if discrim < 0:
        print("\nThe equation has no real roots!")
    elif discrim == 0:
        root = -b / (2 * a)
        print("\nThere is a double root at", root)
```

```
    else:
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )
main()
```

8.2 异常与断言

8.2.1 基本用法

一般情况下，程序对输入数据进行某种加工处理时，需要采取一定的措施，防止产生错误的结果。例如，在计算方程根的程序中，如果用户输入了非数值数据（例如，需要数值的地方，输入了字符串类型），或者，即使输入了数值数据，但是不满足实根条件。无论是哪种情况，程序都应该能够正确地进行处理。通常，可以在如下阶段执行数据的检查验证任务：

- 计算前，检查数据是否符合条件；
- 计算后，检查结果是否符合预期；

我们的系列程序 quadratic*.py 很好地完成了第 2 项工作。

但是，这些程序还存在一个问题：为了能够执行“计算后的数据检查验证”工作，我们在程序中，不得不添加一些辅助性的语句（例如，单路、双路或多路的选择语句），使得程序看起来有些凌乱，且有“喧宾夺主”之嫌。

为了解决这个问题，可以使用异常处理。请看下面的程序：

```
# quadratic5.py
import math

def main():
    print ("This program finds the real solutions to a quadratic\n")

    try:
        a = float(input("Enter coefficient a: "))
```

```
        b = float(input("Enter coefficient b: "))
        c = float(input("Enter coefficient c: "))
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2)
    except ValueError:
        print("\nNo real roots")
main()
```

异常处理的语法如下：

```
try:
    <body>
except <ErrorType>:
    <handler>
```

其处理流程是，执行 <body>，如果没有产生异常，那么继续执行 try...except 下面的语句；否则，如果产生了异常，那么 Python 将在（1 个或多个）except 子句中寻找匹配的异常类型，如果找到，则执行对应的异常处理代码。

执行程序 quadratic5.py，试着按如下数据输入：

```
This program finds the real solutions to a quadratic

Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 3

No real roots
```

再次运行该程序，按如下数据输入：

```
This program finds the real solutions to a quadratic

Enter coefficient a: x
```

```
No real roots
```

实际上，这 2 次引发程序异常的原因是不一样的。第 1 次是因为方程没有实根，第 2 次是因为没有输入程序所需的数值类型数据。为了区分这 2 种异常，程序做了如下改进：

```
# quadratic6.py
import math

def main():
    print("This program finds the real solutions to a quadratic\n")

    try:
        a = float(input("Enter coefficient a: "))
        b = float(input("Enter coefficient b: "))
        c = float(input("Enter coefficient c: "))
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )
    except ValueError as excObj:
        if str(excObj) == "math domain error":
            print("No Real Roots")
        else:
            print("Invalid coefficient given")
    except:
        print("\nSomething went wrong, sorry!")

main()
```

程序中的多条 except 语句形成了多路选择结构（类似于 if-elif-else 结构），用于处理指定的异常。最后一条 except 语句将被触发，如果发生的异常没有被前面

所有的 except 语句捕获。实际上，最后一条 except 语句指定了异常的缺省处理方式。注意，如果产生的异常没有被任何一条 except 语句处理，那么程序将会崩溃，Python 弹出错误提示。

“except ValueError as excObj” 和 str(excObj) 这 2 条语句的作用是，将异常对象赋值给变量 excObj，并将 excObj 转换成字符串对象。

一条 except 子句可以指定多个异常 (使用元组对象表示)，例如：

```
except (RuntimeError, TypeError, NameError):
    <handler>
```

8.2.2 raise 语句

raise 语句允许程序发出指定的异常，语法形式如下：

```
raise <Exception Instance> or <Exception Class>
```

如果给 raise 传递的是 <Exception Class>，那么该类将被隐式地实例化 (没有参数) 为该类实例，例如：

```
raise ValueError
raise ValueError()
```

上述 2 条语句是等价的。使用 raise 语句发出指定的异常：

```
>>> raise NameError('Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: Hello
```

在异常处理程序中，如果只是想判断异常是否发生，而又不想处理它，那么可以使用不带参数的 raise 语句，以重新发出该异常：

```
>>> try:
        raise NameError('Hello')
    except NameError:
        print('NameError exception happend!')
        raise
```

```
NameError exception happend!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: Hello
```

再看一例：

```
import sys

def open_file():
    try:
        f = open('myfile.txt')
        s = f.readline()
        i = int(s.strip())
    except OSError as err:
        print("OS error: {0}".format(err))
    except ValueError:
        print("Could not convert data to an integer.")
    except:
        print("Unexpected error:", sys.exc_info()[0])
        raise

def main():
    open_file()

if __name__ == '__main__':
    main()
```

在上面的程序中，最后一个 except 语句块首先打印了一条错误信息，然后重新抛出异常，以方便 open_file 的调用者 main 函数处理该异常。

8.2.3 else 语句

try...except 语句有一个可选子句 else，它必须出现在所有的 except 语句之后：当所有的异常都没有被触发时，才会执行 else 子句的语句块。例如：

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

8.2.4 finally 语句

try...except 语句还有一个可选子句 finally，用于执行 clean-up 功能，无论代码有没有触发异常，其中的语句块都会被执行，例如：

```
1  def divide(x, y):
2      try:
3          result = x / y
4      except ZeroDivisionError:
5          print("division by zero!")
6      else:
7          print("result is", result)
8      finally:
9          print("executing finally clause")
10
11 divide(4, 2)
12 divide(4, 0)
13 divide('4', '2')
```

执行结果如下：

```
result is 2.0
executing finally clause
division by zero!
executing finally clause
executing finally clause
Traceback (most recent call last):
  File "exception_finally.py", line 13, in <module>
    divide('4', '2')
  File "exception_finally.py", line 3, in divide
    result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

8.2.5 with 语句

有些对象，例如文件对象，定义了 clean-up 方法，可以按如下方式使用 with 语句——在文件读取完毕后，将会自动地关闭该文件：

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

无论该段代码有没有发生异常，打开的文件都能够被正确地关闭。

8.2.6 异常参数

异常被触发时，会生成一个异常对象 (as 语句可以指定一个变量与之关联)，然后可以给该对象附加任何属性，可以被打印输出：

```
1 try:
2     raise Exception('Error 1', 'Error 2')
3 except Exception as inst:
4     print(type(inst.args))
5     inst.args = (inst.args[0], inst.args[1], 'Message Error1')
6     print(inst.args)
7     print(inst)
```

```
8       x, y, z = inst.args
9       print('x =', x)
10      print('y =', y)
11      print('z =', z)
```

输出结果如下:

```
<class 'tuple'>
('Error 1', 'Error 2', 'Message Error1')
('Error 1', 'Error 2', 'Message Error1')
x = Error 1
y = Error 2
z = Message Error1
```

异常对象定义了方法 ___str___(), 用于输出属性 inst.args 的值。因此, 语句 print(inst.args) 和 print(inst) 功能一样。

8.2.7 自定义异常

程序可以创建自定义的异常类, 该类应该直接或间接地从 Exception 类派生。请看下面的程序:

```
1  class BaseError(Exception):
2      def __init__(self, ErrorName='BaseError'):
3          Exception.__init__(self, ErrorName)
4
5  class DerivedError(BaseError):
6      def __init__(self, ErrorName='DerivedError'):
7          BaseError.__init__(self, ErrorName)
8
9  class DDerivedError(DerivedError):
10     def __init__(self, ErrorName='DDerivedError'):
11         DerivedError.__init__(self, ErrorName)
12
```

```
13 for cls in [BaseError, DerivedError, DDerivedError]:
14     try:
15         raise cls
16     except DDerivedError as dde:
17         print(dde)
18     except DerivedError as de:
19         print(de)
20     except BaseError as be:
21         print(be)
```

输出结果如下：

```
BaseError
DerivedError
DDerivedError
```

8.2.8 断言

我们可以使用断言 assert 语句来断定程序是否按预期执行，assert 有 2 种形式：

```
assert Boolean-Expression
assert Boolean-Expression, argument
```

执行 assert 语句时，先对布尔表达式求值：结果为 True，继续执行下面的语句；否则，触发'AssertionError' 异常。例如：

```
>>> a = 1
>>> b = 2
>>> assert a == b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert a == b, 'a is not equal b'
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
AssertionError: a is not equal b
>>> assert a != b
>>>
```

8.3 循环结构

在3.8中，我们已经学习过 for 循环结构的使用。它的语法结构如下：

```
for <var> in <sequence>:
    <body>
```

其中，循环体 <body> 必须使用缩进格式，<var> 是循环索引 (Loop Index)，它的取值来自于 <sequence>。对于 <var> 的每个取值，<body> 将被执行一次。

下面的程序利用 for 循环计算 n 个数的平均数：

```
# average1.py


def main():
    n = int(input("How many numbers do you have? "))
    total = 0.0
    for i in range(n):
        x = float(input("Enter a number >> "))
        total = total + x
    print("\nThe average of the numbers is", total / n)


main()
```

for 循环被称为确定循环 (Definite Loop)，因为循环次数是已知的。

还有一种循环，它的循环次数是未知的，或者，循环与否是依条件而定的，我们把这样的循环称为无限循环 (Indefinite Loop) 或条件循环 (Conditional Loop)。在 Python 中，while 语句实现了这种循环：

```
while <condition>:
    <body>
```

其中，<condition> 是一个条件表达式，可以使用关系表达式和布尔 (逻辑) 表达式，与 if 语句中的一样。循环体执行的条件是 <condition> 为 True。while 的循环次数是 0，如果 <condition> 的初始值为 False。while 循环也被称为是 pre-test 循环或当型循环[18]。

下面 2 个程序片段实现的功能一样：

```
i = 0
while i <= 10:
    print(i)
    i = i + 1
```

while 循环需要代码显式地初始化和更新变量 i。

```
for i in range(11):
    print(i)
```

while 循环的一个直接应用是实现交互式循环：由用户决定循环是否继续下去。

下面直接给出 average1.py 的改进版本：

```
# average2.py

def main():
    total = 0.0
    count = 0
    moredata = "yes"
    while moredata[0] == "y":
        x = float(input("Enter a number >> "))
        total = total + x
        count = count + 1
        moredata = input("Do you have more numbers (yes or no)? ")
```

[18]还有一种是直到型循环或 post-test 循环，类似于 repeat...until <condition>，语义是：循环执行，直到条件 <condition> 成立为止。在 Python 中，没有显式的语句支持这种结构。但是，可以使用如下结构代替：

```
while True:
    ...
    if <condition>:
        break;
```

```
    print("\nThe average of the numbers is", total / count)

main()
```

上面的程序对用户不是很友好，可以使用“哨兵循环”(Sentinel Loop) 来改善用户体验：循环一直进行下去，直到碰到“哨兵”(特定值)。哨兵值的选择是任意的，只要它能够与用户数据区分开来。下面是哨兵循环的算法：

```
get the first data item
while item is not the sentinel
    process the item
    get the next data item
```

应用哨兵循环后的改进程序如下：

```
# average3.py

def main():
    total = 0.0
    count = 0
    x = float(input("Enter a number (negative to quit) >> "))
    while x >= 0:
        total = total + x
        count = count + 1
        x = float(input("Enter a number (negative to quit) >> "))
    print("\nThe average of the numbers is", total / count)

main()
```

该程序还是有限制：不能计算任意 n 个数值的平均数，因为哨兵值本身也是一个数值类型，无法从用户数据中区分开。可以尝试使用字符串类型的哨兵值：

```
# average4.py

def main():
```

```
    total = 0.0
    count = 0
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = float(xStr)
        total = total + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit) >> ")
    print("\nThe average of the numbers is", total / count)

main()
```

如果要处理的数据比较多，那么交互式程序就不大适合了，此时，可以将数据存放于文件中，利用文件对象来获取这些数据并进行处理（注意，文件中每行只存放一个数据）：

```
# average5.py

def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    total = 0.0
    count = 0
    for line in infile:
        total = total + float(line)
        count = count + 1
    print("\nThe average of the numbers is", total / count)

main()
```

假定文件允许每一行存放多个数据项（以“,”分割），继续改写该程序：

```
# average6.py
```

```
def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    total = 0.0
    count = 0
    for line in infile:
        # update total and count for values in line
        for xStr in line.split(","):
            total = total + float(xStr)
            count = count + 1
    print("\nThe average of the numbers is", total / count)

main()
```

8.4 逻辑 (布尔) 表达式

Python 提供了 3 种逻辑运算符：and、or、not，语法格式如下：

```
<expr> and <expr>
<expr> or <expr>
not <expr>
```

其中，<expr> 表示关系表达式或逻辑表达式，and、or、not 分别与逻辑语义上的与、或、非相对应，它们的优先级从低到高依次为 or、and、not。因此，表达式：

```
a or not b and c
```

等价于：

```
a or ((not b) and c)
```

一般情况下，为增强程序的可读性，最好在适当的地方添加括号。

例如，下面的逻辑表达式用来判断排球比赛是否结束，假设 a 和 b 为两队得分：

```
(a >= 15 or b >= 15) and abs(a - b) >= 2
```

operator	operational definition
x and y	If x is false, return x. Otherwise, return y.
x or y	If x is true, return x. Otherwise, return y.
not x	If x is false, return `True`. Otherwise, return `False`.

图 8-28: 计算逻辑表达式的短路规则

值得注意的是，在处理逻辑结果 (True 和 False) 的问题上，同 C/C++ 等语言一样，Python 的处理方式也非常灵活：任何内建数据类型都可以被解释成逻辑值。例如，对于数值类型数据 (int 和 float)，0 的逻辑值被解释成 False，任何非 0 的逻辑值被解释成 True；对于序列数据类型 (字符串、列表等)，空序列的逻辑值被解释成 False，非空序列的逻辑值被解释成 True：

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool(32)
True
>>> bool("hello")
True
>>> bool("")
False
>>> bool([1, 2, 3])
True
>>> bool([])
False
```

另外，Python 以短路 (short-circuit) 规则计算逻辑表达式。如图8-28所示，只要 and 表达式的第 1 项为 False，或者 or 的第 1 项为 True，表达式就直接返回结果，甚至不会去计算后面子表达式的值。

试分析比较下面几个程序[19]：

[19]注意，input 会自动过滤掉换行符，例如，如果用户直接按下 <Enter> 键，那么 input 将只返回空字符串。

1.
```
ans = input("What flavor do you want [vanilla]: ")
if ans != "":
    flavor = ans
else:
    flavor = "vanilla"
```

2.
```
ans = input("What flavor do you want [vanilla]: ")
if ans:
    flavor = ans
else:
    flavor = "vanilla"
```

3.
```
ans = input("What flavor do you want [vanilla]: ")
flavor = ans or "vanilla"
```

4.
```
flavor = input("What flavor do you want [vanilla]: ") or "vanilla"
```

它们在功能上完全一样，可以根据个人喜好来选择其中的一种。当然，就可读性和简洁性而言，推荐第 2 种方式。最后 2 种形式，还需要额外考虑短路规则。

8.5 实例：简单的消息循环

在实际应用中，有一种改进的直到型 while 循环，形式上较为优雅简洁：

```
while True:
    get next item
    if item == sentinel:
        break
    process the item
```

它的退出语句在循环体的中间位置，被称为 Loop-and-Half 结构。如果当前处理项 item 与预先设置的哨兵值相等时，循环退出。

在 GUI 程序中，可以应用 Loop-and-Half 结构构成所谓的消息循环：

```
draw the GUI
while True:
    get next event
    if event is "quit signal":
        break
    process the event
clean up and exit
```

其中，可以将 “quit signal” 设置成 ESC 键、q/Q 键或 CLOSE/QUIT 菜单项点击事件。

下面的程序，实现了按键消息循环：

```
# event_loop1.py
#      Simple color changing window with keyboard controls

from graphics import *

def main():
    win = GraphWin("Color Window", 500, 500)

    # Event Loop: handle key presses until user presses the "q" key.
    while True:
        key = win.getKey()
        if key == "q":  # loop exit
            break

        #process the key
        if key == "r":
            win.setBackground("pink")
        elif key =="w":
            win.setBackground("white")
        elif key == "g":
            win.setBackground("lightgray")
```

```
        elif key == "b":
            win.setBackground("lightblue")

    # exit program
    win.close()

main()
```

注意，函数 getKey 会一直等待，直到用户按下某个按键。这种输入方式，被称为阻塞式或模式 (Modal，因为程序限制了用户的输入，只能进入某种特定的交互模式) 输入。

但是，在实际的 GUI 消息循环中，一般会允许用户通过按键、鼠标、菜单、按钮等方式与程序进行交互，即程序应该支持非阻塞式、非模式/多模式 (Non-Modal/Multi-Modal) 输入。在这种情况下，getKey 函数就不适用了，取而代之的是 checkKey 函数，它不会等待用户按键。同时，为了支持鼠标非阻塞式输入，需要使用 checkMouse 函数。

下面是改进后的程序：

```
# event_loop2.py --- color changing window
#       reorganized to incorporate mouse inputs

from graphics import *

def handleKey(k, win):
    if k == "r":
        win.setBackground("pink")
    elif k == "w":
        win.setBackground("white")
    elif k == "g":
        win.setBackground("lightgray")
    elif k == "b":
        win.setBackground("lightblue")
```

```
def handleClick(pt, win):
    pass

def main():
    win = GraphWin("Click and Type", 500, 500)

    # Event Loop: handle key presses and mouse clicks until the user
    #    presses the "q" key.
    while True:
        key = win.checkKey()
        if key == "q":  # loop exit
            break

        if key:
            handleKey(key, win)

        pt = win.checkMouse()
        if pt:
            handleClick(pt, win)

    win.close()
main()
```

该程序精确实现了第 1 个程序的功能，并额外增加了鼠标输入事件的处理结构——尽管目前还未实现任何有用的功能。handleClick 函数体中仅有一条语句“pass”，添加该语句只是为了语法上的需要，它不会执行任何功能。

表达式 key 等价于`key != ""`，表达式 pt 等价于`pt != None`，由于 Python 处理逻辑表达式的灵活性，两者实现的功能完全一样，程序中的写法显得更简洁些。当然，对于初学者而言，可读性略差，而对于有经验的程序员来讲，则不会产生任何问题。

下面的程序，将模式与非模式输入结合在一起，实现了相对复杂的功能：

```
# event_loop3.py
#       Color changing window with clicks to enter text

from graphics import *

def handleKey(k, win):
    if k == "r":
        win.setBackground("pink")
    elif k == "w":
        win.setBackground("white")
    elif k == "g":
        win.setBackground("lightgray")
    elif k == "b":
        win.setBackground("lightblue")

def handleClick(pt, win):
    # create an Entry for user to type in
    entry = Entry(pt, 10)
    entry.draw(win)

    # Go modal: wait until user types Return or Escape Key
    while True:
        key = win.getKey()
        if key == "Return":
            break

    # undraw the entry and draw Text
    entry.undraw()
    Text(pt, entry.getText()).draw(win)
```

```
    # clear (ignore) any mouse click that occurred during text entry
    win.checkMouse()

def main():
    win = GraphWin("Click and Type", 500, 500)

    # Event Loop: handle key presses and mouse clicks until the user
    #    presses the "q" key.
    while True:
        key = win.checkKey()
        if key == "q":  # loop exit
            break

        if key:
            handleKey(key, win)

        pt = win.checkMouse()
        if pt:
            handleClick(pt, win)

    win.close()

main()
```

在 handleClick 函数中，（当用户在窗口客户区点击了鼠标后）先创建并显示一个文本输入框 (Entry 对象)，允许用户在该框内输入字符串。在 while True 循环结构中，使用 getKey 函数，形成了一个典型的模式输入结构——只有当用户按下 ENTER 键，才结束循环。接着，将 Entry 对象隐藏起来，并在原处显示用户输入的字符串。最后调用 checkMouse 函数，将此过程中可能产生的所有鼠标点击事件清除掉（否则，产生的鼠标事件将被 main 函数中的 checkMouse 捕获，而使程序产生非预期的行为）。

8.6 练习

1. 编写一个程序，判断某年是否闰年：非整百年数除以 4，无余为闰，有余为平；整百年数（世纪年）除以 400，无余为闰，有余为平。例如，1800 和 1900 是平年，而 1600 和 2000 是闰年；

2. 编写一个程序：在图形窗口中随机绘制一个运动的圆，当圆与窗口边框发生碰撞时，实现碰撞的效果。下面是实现圆运动的程序片段：

```
win = GraphWin("Update Example ", 320, 200, autoflush=False)
while True:
    key = win.checkKey()
    if key == 'Escape':
        break
    #<drawing commands for the current frame>
    ...
    update(30)
```

其中，update(30) 表示图形窗口每秒的刷新次数为 30。"..." 表示需要补充的代码；

3. 编写一个程序，输出 Fibonacci 序列：1, 1, 2, 3, 5, 8, ...，序列个数由用户指定；

9 类

9.1 类的定义

类的定义格式非常简单：

```
class <class-name>(<base>):
    <method-definitions>
```

其中，<method-definitions> 与标准的函数定义格式一样，只是它们的位置是在类 <class-name> 的内部，<base> 是基类名称[20]。

[20]在 Python2 中，一般要指定基类为 object。在 Python3 中，如果没有指定，缺省都是 object。

例如，定义一个骰子类 MSDice：

```
# msdice.py
#    Class definition for an n-sided dice.

from random import randrange

class MSDice:

    def __init__(self, sides):
        self.sides = sides
        self.value = 1

    def roll(self):
        self.value = randrange(1, self.sides+1)

    def getValue(self):
        return self.value

    def setValue(self, value):
        self.value = value
```

在上面的类定义中，每个方法的第 1 个参数总是特定的——表示类的实例化对象，其名称是任意的，一般命名为 “self”[21]。注意，在定义类的方法时，总是要额外增加一个参数 self，但是在调用方法时，该参数是不会出现的，例如：

```
def main():
    dice = MSDice(12)
    dice.setValue(9)
    print(dice.getValue())
```

[21]在 Python 中，一切皆对象。前文已述，对象分为可变对象与不可变对象。除非特别说明，一般情况下，自定义类的对象是可变对象。因此，self 在类内部是共享的。从这个角度讲，self 相当于 C++ 中的 (某个变量或对象的) 引用，或者，它与 C++ 中 this 指针所起的作用相同。

上面的程序片段中，实例化了一个 MSDice 类的对象 dice，通过`dice.setValue(9)`调用该实例对象的方法。

在类 MSDice 的定义中，有一个特殊的方法`__init__`，它是类的构造函数，用于初始化对象的属性。在该方法内部，将类的属性 sides 和 value 绑定到了 self 上[22]。

从程序 msdice.py 可以看出，类 MSDice 将属性与方法封装在一起，对外界而言，它隐藏了类内部的复杂性，起到了屏蔽和抽象的作用。我们把这种机制称为类的封装性。

9.2 访问限制

程序 msdice.py 存在一个问题：虽然利用封装性将类的属性与方法集成在一起，但是外界可以自由地修改对象的属性，例如：

```
>>> dice = msdice.MSDice(12)
>>> dice.sides = 15
>>> dice.value = 10
```

在缺省情况下，属性与方法都是 public 访问类型，这显然不符合我们的预期。但是，Python 并没有提供显式的语法支持属性和方法的私有访问。有一种解决办法，可以在属性名称前加上两个下划线`__`，使该属性变成私有 (private) 变量[23]。改进后的程序如下：

```
# msdice1.py
#    Class definition for an n-sided dice.

from random import randrange

class MSDice():

    def __init__(self, sides):
```

[22] 实际上，可以将各种属性绑定到 self 上。此例中，只绑定了 sides 和 value 这 2 个属性。注意，如果没有指定前缀 “self.”，那么该变量是一个局部变量，而不是类的属性。

[23] 对类中方法也同样适用。

```
        self.__sides = sides
        self.__value = 1

    def roll(self):
        self.__value = randrange(1, self.__sides + 1)

    def getValue(self):
        return self.__value

    def setValue(self, value):
        self.__value = value
```

进入 Python 命令行，输入：

```
>>> import msdice1
>>> dice = msdice1.MSDice(12)
>>> dice.value
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'MSDice' object has no attribute 'value'
>>> dice.__value
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'MSDice' object has no attribute '__value'
```

可以看出，现在已经不能访问 value 属性了[24]。继续验证：

```
>>> dice.value = 2
>>> dice.value
2
>>> dice.getValue()
1
```

[24]但是，有一种 hacking 方法：dice._MSDice__value，依然可以访问属性 value，其中 MSDice 是类名。这种方法常用于程序调试。

```
>>> id(dice.value)
1749512880
>>> dice._MSDice__value
1
>>> id(dice._MSDice__value)
1749512848
```

从上述结果可以看出，尽管仍然可以执行类似dice.value = 2 这样的语句，但是，dice.value 与 dice 对象内部的 value(内部名称已经变为_MSDice__value) 是 2 个不同的属性[25]。因此，这种方法起到了保护属性的作用。

一般情况下，有必要为私有属性提供读写接口。请看下例：

```
# msdice2.py
#    Class definition for an n-sided dice.

from random import randrange

class MSDice():

    def __init__(self, sides):
        self.__sides = sides
        self.__value = 1

    def roll(self):
        self.__value = randrange(1, self.__sides + 1)

    def __getValue(self):
        return self.__value

    def __setValue(self, value):
        self.__value = value
```

[25]Python 是动态类型语言，可以给实例任意绑定属性。dice.value 是该实例特有的属性，而 dice._MSDice__value 是类的所有实例的属性。

```
    @property
    def sides(self):
        return self.__sides

    @sides.setter
    def sides(self, sides):
        self.__sides = sides

    value = property(__getValue, __setValue)
```

有 2 种方法可以给类中属性定义 get（读）和 set（写）方法 (类似于Java/C# 中的 get/set 属性访问器):

- 使用内建函数 property;

  例如，属性 value 分别关联私有方法__getValue 和__setValue，使用方法如下:

  ```
  >>> dice.value
  2
  >>> dice.value = 4
  ```

- 使用 @property 装饰器;

  例如，属性 sides 分别关联方法 sides(self) 和 self(self, sides)，使用方法与 value 一样;

注意，从形式上看，虽然公有属性也可以使用相似的语句进行读写，但是两者有着本质的区别——私有属性的 get/set 方法是接口，根据需要可以使写方法失效，或者，还可以执行较复杂的参数检测任务，避免传入无效的参数。

需要注意的是，名称类似于__xxx__ 的变量，是特殊变量，它是可以直接访问的，不是 private 变量。另外，有时候，在类的内部也可以看到类似于_xxx 这样的变量，它也是可以直接访问的。但是，按照惯例，该属性的访问类型是保护 (protected) 类型。

9.3 继承与多态

先阅读下面的程序：

```
# people.py
#    Class definition for student, employee, employer,...

class People:
    """People: Base class
        Method: ToString"""

    def __init__(self, name, age, gender):
        self.__name = name
        self.__age = age
        self.__gender = gender

    def ToString(self):
        return ','.join([self.__name, str(self.__age), self.__gender])

class Student(People):
    """Student: Derived class from People
        Method: ToString"""

    def ToString(self):
        return ','.join(['Student', super(Student, self).ToString()])

class Employee(People):
    """Employee: Derived class from People
        Method: ToString"""

    def ToString(self):
        return ','.join(['Employee', super(Employee, self).ToString()])
```

```
def PrintInfo(obj):
    print(obj.ToString())

def main():
    p = People('Qin Shao', 50, 'M')
    s = Student('Li Li', 20, 'F')
    e = Employee('Zhang Jie', 30, 'M')
    PrintInfo(p)
    PrintInfo(s)
    PrintInfo(e)

if (__name__ == '__main__'):
    main()
```

在命令行下输入：python people.py，得到如下结果：

```
Qin Shao,50,M
Student,Li Li,20,F
Employee,Zhang Jie,30,M
```

在程序 people.py 中，定义了三个类：People、Student 和 Employee。后面 2 个类分别从基类 People 中派生，它们自动继承了基类的方法 ToString，并且通过 super 可以调用基类方法。

在 main 函数中，分别使用参数 p、s、e 调用函数 PrintInfo。在该函数中，obj.ToString() 会根据 obj 的类型自动调用 obj 自身的 ToString 方法。本例中，它们分别调用 People 类、Student 类、Employee 类的 ToString 方法。这种现象被称为类的多态性[26]。

使用函数 isinstance 可以判断变量是否属于指定类型：

[26]对于静态语言，例如 C++，在本例中，PrintInfo 的参数必须明确指明是 People 类型对象。而对于动态语言，例如 Python，PrintInfo 的参数不一定是 People 类型对象，可以是任意类型的对象，只要该对象有 ToString 方法。这种特性是非常有用的，例如，在 Python 中，许多函数接收的参数是 File-Like 对象——在调用这些函数时，不必传入真正的文件对象，而只要确保传入的对象实现了文件对象的主要方法。

```
>>> import people
>>> isinstance(3,int)
True
>>> p = people.People('Qin Shao', 50, 'M')
>>> s = people.Student('Li Li', 20, 'F')
>>> e = people.Employee('Zhang Jie', 30, 'M')
>>> isinstance(p,people.People)
True
>>> isinstance(s,people.Student)
True
>>> isinstance(s,people.People)
True
>>> isinstance(p,people.Student)
False
```

可以看出，子类对象 s 也是父类实例；但是，反过来，父类对象 p 就不是子类实例了。实际上，这种关系也符合现实世界中的父子概念关系——例如，Student 是 People 中的一种，反过来，People 未必是 Student 中的一种。

使用函数 dir 可以获得指定对象的所有属性和方法：

```
>>> dir(p)
['ToString', '_People__age', '_People__gender', '_People__name',
 '__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__']
>>> dir('test')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
```

```
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

类似__xxx__ 这样的属性和方法都是有特殊用途的，比如__len__ 方法可以返回对象的长度。如果调用 len() 函数试图获取一个对象的长度，实际上，在 len() 函数的内部，它会自动去调用该对象的__len__ 方法，所以，下面的代码是等价的：

```
>>> len('test')
4
>>> 'test'.__len__()
4
```

这意味着，可以在自定义类中添加类似__len__ 这样的方法，以完成特定功能。

使用函数 getattr、setattr、hasattr 可以分别获取属性的值、设置属性的值以及判断属性是否存在：

```
>>> getattr(p,'_People__name')
'Qin Shao'
>>> setattr(p,'_People__name','Li Hao')
>>> getattr(p,'_People__name')
'Li Hao'
>>> hasattr(p,'_People__name')
True
```

注：在 People 类中，name 是私有属性，必须要使用_People__name 才能直接访问。getattr 和 hasattr 同样适用于方法的获取与判断。

有时候，需要显示类的帮助文档，此时可以使用一个特别的字符串属性__doc__，它存放的是程序中的帮助文档 (首尾各使用三个引号的注释，详见程序 people.py)，例如：

```
>>> import people
>>> print(people.People.__doc__)
People: Base class
Method: ToString
>>> print(people.Student.__doc__)
Student: Derived class from People
Method: ToString
>>> import random
>>> print(random.random.__doc__)
random() -> x in the interval [0, 1).
```

另外，还可以使用 help 函数获取更全面的帮助信息，例如，help(people.People)。

9.4 实例

9.4.1 Dice Roller

Dice Roller 的运行界面如图9-29所示。该程序由 2 个 Button、2 个 6 面骰子组成。

Button 类

Button 类具有以下方法：

- __init__

  创建一个 Button 对象，需要指定父窗口、位置、大小和标签；

- activate

  激活 Button 对象；

- deactivate

  使 Button 对象失效；

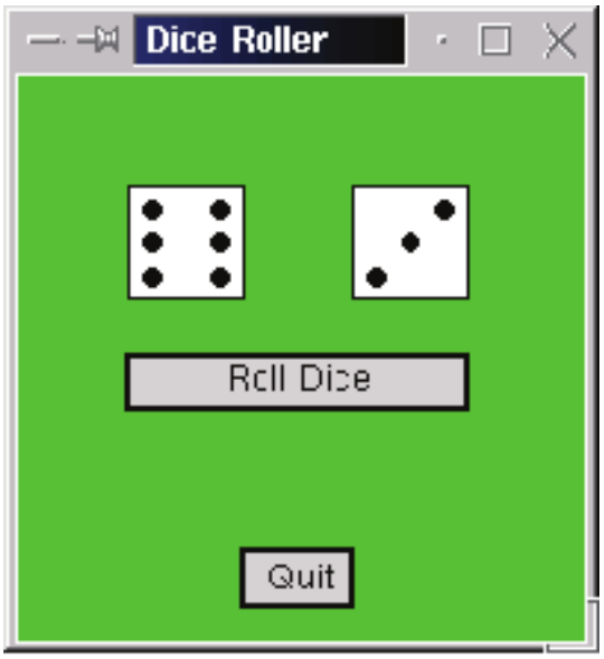

图 9-29: Dice Roller 的运行界面

- clicked

  判断鼠标是否点击了 Button 对象；

- getLabel

  返回 Button 对象的标签；

下面是完整的 Button 类：

```
# button.py
from graphics import *

class Button:

    """A button is a labeled rectangle in a window.
    It is activated or deactivated with the activate()
    and deactivate() methods. The clicked(p) method
    returns true if the button is active and p is inside it."""
```

```
def __init__(self, win, center, width, height, label):
    """ Creates a rectangular button, eg:
    qb = Button(myWin, centerPoint, width, height, 'Quit') """

    w,h = width/2.0, height/2.0
    x,y = center.getX(), center.getY()
    self.xmax, self.xmin = x+w, x-w
    self.ymax, self.ymin = y+h, y-h
    p1 = Point(self.xmin, self.ymin)
    p2 = Point(self.xmax, self.ymax)
    self.rect = Rectangle(p1,p2)
    self.rect.setFill('lightgray')
    self.rect.draw(win)
    self.label = Text(center, label)
    self.label.draw(win)
    self.deactivate()

def clicked(self, p):
    "Returns true if button active and p is inside"
    return (self.active and
            self.xmin <= p.getX() <= self.xmax and
            self.ymin <= p.getY() <= self.ymax)

def getLabel(self):
    "Returns the label string of this button."
    return self.label.getText()

def activate(self):
    "Sets this button to 'active'."
    self.label.setFill('black')
    self.rect.setWidth(2)
```

```
        self.active = True

    def deactivate(self):
        "Sets this button to 'inactive'."
        self.label.setFill('darkgrey')
        self.rect.setWidth(1)
        self.active = False
```

DiceView 类

DiceView 类负责骰子的显示，它具有如下方法：

- __init__

  创建一个 DiceView 类对象，需要指定父窗口、中点、大小；

- setValue

  设置骰子的点数；

下面是完整的 DiceView 类：

```
# dieview.py
from graphics import *
class DiceView:
    """ DiceView is a widget that displays a graphical representation
    of a standard six-sided dice."""

    def __init__(self, win, center, size):
        """Create a view of a dice, e.g.:
           d1 = GDice(myWin, Point(40,50), 20)
        creates a dice centered at (40,50) having sides
        of length 20."""

        # first define some standard values
        self.win = win            # save this for drawing pips later
```

```
        self.background = "white" # color of die face
        self.foreground = "black" # color of the pips
        self.psize = 0.1 * size   # radius of each pip
        hsize = size / 2.0        # half the size of the dice
        offset = 0.6 * hsize      # distance from center to outer pips

        # create a square for the face
        cx, cy = center.getX(), center.getY()
        p1 = Point(cx-hsize, cy-hsize)
        p2 = Point(cx+hsize, cy+hsize)
        rect = Rectangle(p1,p2)
        rect.draw(win)
        rect.setFill(self.background)

        # Create 7 circles for standard pip locations
        self.pip1 = self.__makePip(cx-offset, cy-offset)
        self.pip2 = self.__makePip(cx-offset, cy)
        self.pip3 = self.__makePip(cx-offset, cy+offset)
        self.pip4 = self.__makePip(cx, cy)
        self.pip5 = self.__makePip(cx+offset, cy-offset)
        self.pip6 = self.__makePip(cx+offset, cy)
        self.pip7 = self.__makePip(cx+offset, cy+offset)

        # Draw an initial value
        self.setValue(1)

    def __makePip(self, x, y):
        "Internal helper method to draw a pip at (x,y)"
        pip = Circle(Point(x,y), self.psize)
        pip.setFill(self.background)
        pip.setOutline(self.background)
```

```
        pip.draw(self.win)
        return pip

    def setValue(self, value):
        "Set this dice to display value."
        # turn all pips off
        self.pip1.setFill(self.background)
        self.pip2.setFill(self.background)
        self.pip3.setFill(self.background)
        self.pip4.setFill(self.background)
        self.pip5.setFill(self.background)
        self.pip6.setFill(self.background)
        self.pip7.setFill(self.background)

        # turn correct pips on
        if value == 1:
            self.pip4.setFill(self.foreground)
        elif value == 2:
            self.pip1.setFill(self.foreground)
            self.pip7.setFill(self.foreground)
        elif value == 3:
            self.pip1.setFill(self.foreground)
            self.pip7.setFill(self.foreground)
            self.pip4.setFill(self.foreground)
        elif value == 4:
            self.pip1.setFill(self.foreground)
            self.pip3.setFill(self.foreground)
            self.pip5.setFill(self.foreground)
            self.pip7.setFill(self.foreground)
        elif value == 5:
            self.pip1.setFill(self.foreground)
```

```
            self.pip3.setFill(self.foreground)
            self.pip4.setFill(self.foreground)
            self.pip5.setFill(self.foreground)
            self.pip7.setFill(self.foreground)
        else:
            self.pip1.setFill(self.foreground)
            self.pip2.setFill(self.foreground)
            self.pip3.setFill(self.foreground)
            self.pip5.setFill(self.foreground)
            self.pip6.setFill(self.foreground)
            self.pip7.setFill(self.foreground)
```

main 函数

主函数所在的文件名称为 roller.py，其内容如下：

```
# roller.py
# Graphics program to roll a pair of dice. Uses custom widgets
# Button and DieView.

from random import randrange
from graphics import GraphWin, Point

from button import Button
from diceview import DiceView

def main():

    # create the application window
    win = GraphWin("Dice Roller")
    win.setCoords(0, 0, 10, 10)
    win.setBackground("green2")
```

```
    # Draw the interface widgets
    dice1 = DiceView(win, Point(3,7), 2)
    dice2 = DiceView(win, Point(7,7), 2)
    rollButton = Button(win, Point(5,4.5), 6, 1, "Roll Dice")
    rollButton.activate()
    quitButton = Button(win, Point(5,1), 2, 1, "Quit")

    # Event loop
    pt = win.getMouse()
    while not quitButton.clicked(pt):
        if rollButton.clicked(pt):
            value1 = randrange(1,7)
            dice1.setValue(value1)
            value2 = randrange(1,7)
            dice2.setValue(value2)
            quitButton.activate()
        pt = win.getMouse()

    # close up shop
    win.close()

main()
```

9.4.2 Animated Cannonball

Animated Cannonball 的运行界面如图9-30所示。程序使用动画的方式展示了 Cannonball 发射的全过程。

animation 模块

该模块包含在程序 animation.py 中，由 InputDialog 类、ShotTracker 类以及 main 函数组成。下面是 animation.py 的内容：

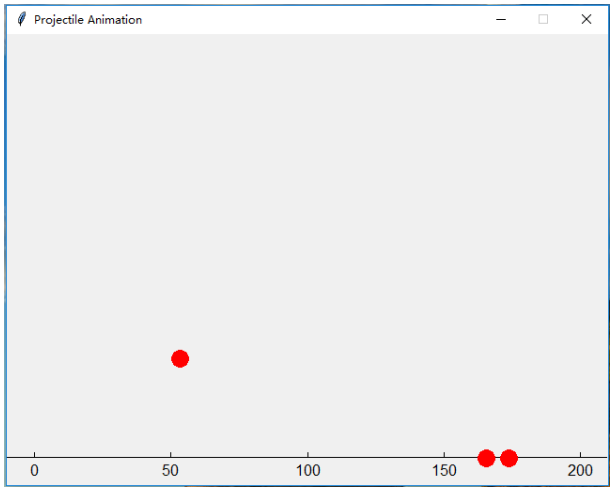

图 9-30: Animated Cannonball 的运行界面

```
# animation.py

# single-shot cannonball animation

from math import sqrt, sin, cos, radians, degrees
from graphics import *
from projectile import Projectile
from button import Button

class InputDialog:

    """ A custom window for getting simulation values (angle, velocity,
    and height) from the user."""

    def __init__(self, angle, vel, height):
        """ Build and display the input window """

        self.win = win = GraphWin("Initial Values", 200, 300)
        win.setCoords(0,4.5,4,.5)
```

```
        Text(Point(1,1), "Angle").draw(win)
        self.angle = Entry(Point(3,1), 5).draw(win)
        self.angle.setText(str(angle))

        Text(Point(1,2), "Velocity").draw(win)
        self.vel = Entry(Point(3,2), 5).draw(win)
        self.vel.setText(str(vel))

        Text(Point(1,3), "Height").draw(win)
        self.height = Entry(Point(3,3), 5).draw(win)
        self.height.setText(str(height))

        self.fire = Button(win, Point(1,4), 1.25, .5, "Fire!")
        self.fire.activate()

        self.quit = Button(win, Point(3,4), 1.25, .5, "Quit")
        self.quit.activate()

    def getValues(self):
        """ return input values """

        a = float(self.angle.getText())
        v = float(self.vel.getText())
        h = float(self.height.getText())
        return a,v,h

    def interact(self):
        """ wait for user to click Quit or Fire button
        Returns a string indicating which button was clicker
        """
```

```
        while True:
            pt = self.win.getMouse()
            if self.quit.clicked(pt):
                return "Quit"
            if self.fire.clicked(pt):
                return "Fire!"

    def close(self):
        """ close the input window """
        self.win.close()

class ShotTracker:

    """ Graphical depiction of a projectile flight using a Circle """

    def __init__(self, win, angle, velocity, height):
        """win is the GraphWin to display the shot, angle, velocity, and
        height are initial projectile parameters.
        """

        self.proj = Projectile(angle, velocity, height)
        self.marker = Circle(Point(0,height), 3)
        self.marker.setFill("red")
        self.marker.setOutline("red")
        self.marker.draw(win)

    def update(self, dt):
        """ Move the shot dt seconds farther along its flight """

        self.proj.update(dt)
        center = self.marker.getCenter()
```

```
        dx = self.proj.getX() - center.getX()
        dy = self.proj.getY() - center.getY()
        self.marker.move(dx,dy)

    def getX(self):
        """ return the current x coordinate of the shot's center """
        return self.proj.getX()

    def getY(self):
        """ return the current y coordinate of the shot's center """
        return self.proj.getY()

    def undraw(self):
        """ undraw the shot """
        self.marker.undraw()

def main():

    # create animation window
    win = GraphWin("Projectile Animation", 640, 480, autoflush=False)
    win.setCoords(-10, -10, 210, 155)
    Line(Point(-10,0), Point(210,0)).draw(win)
    for x in range(0, 210, 50):
        Text(Point(x,-5), str(x)).draw(win)
        Line(Point(x,0), Point(x,2)).draw(win)

    angle, vel, height = 45.0, 40.0, 2.0
    # event loop
    while True:
        # interact with the user
        inputwin = InputDialog(angle, vel, height)
```

```
        choice = inputwin.interact()
        inputwin.close()

        if choice == "Quit": # loop exit
            break

        # otherwise choice is "Fire!"
        # create a shot and track until it hits ground or leaves window
        angle, vel, height = inputwin.getValues()
        shot = ShotTracker(win, angle, vel, height)
        while 0 <= shot.getY() and -10 < shot.getX() <= 210:
            shot.update(1/50)
            update(50)

    win.close()

if __name__ == "__main__":
    main()
```

main 函数 注意，在 main 函数中，实例化对象 win 时指定了参数 autoflush=False，意味着图形对象不会自动刷新，除非调用了 update 函数——当所有相关对象的状态都修改完毕时，将它们一次提交给图形硬件刷新显示。这种方式常用于动画应用中，可以提高程序的图形显示效率与效果。

ShotTracker 类 ShotTracker 类包含 2 个类的实例对象，这 2 个类分别是 Projectile 和 Circle，前者建模 Projectile 对象的物理运动规律，后者负责显示 Projectile 对象。ShotTracker 类的作用是同步它们，并以动画的方式显示。

InputDialog 类 该类负责创建一个参数输入对话框，如图9-31所示。注意，函数 interact 表明，该对话框是一个模式对话框。

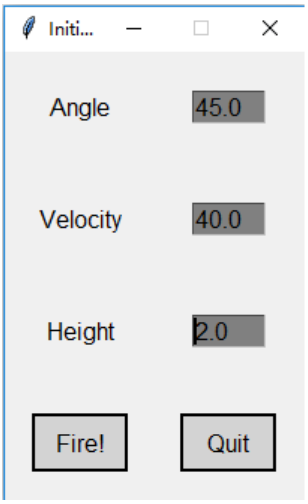

图 9-31: Animated Cannonball 的参数输入对话框

Projectile 类

Projectile 类实现了炮弹飞行的物理运动轨迹的建模，程序如下：

```
# projectile.py

"""
Provides a simple class for modeling the flight of projectiles.
"""

from math import sin, cos, radians

class Projectile:

    """Simulates the flight of simple projectiles near the earth's
    surface, ignoring wind resistance. Tracking is done in two
    dimensions, height (y) and distance (x)."""

    def __init__(self, angle, velocity, height):
        """Create a projectile with given launch angle, initial
        velocity and height."""
```

```
        self.xpos = 0.0
        self.ypos = height
        theta = radians(angle)
        self.xvel = velocity * cos(theta)
        self.yvel = velocity * sin(theta)

    def update(self, time):
        """Update the state of this projectile to move it time seconds
        farther into its flight"""
        self.xpos = self.xpos + time * self.xvel
        yvel1 = self.yvel - 9.8 * time
        self.ypos = self.ypos + time * (self.yvel + yvel1) / 2.0
        self.yvel = yvel1

    def getY(self):
        "Returns the y position (height) of this projectile."
        return self.ypos

    def getX(self):
        "Returns the x position (distance) of this projectile."
        return self.xpos
```

9.5 练习

1. 编写程序，实现 TicTacToe 的图形化界面，并支持鼠标下棋功能；

2. 编写程序，实现图形化的学生信息管理系统，支持数据输入、查询及文件的读取与保存；

10 其它结构化数据类型

到目前为止，我们已经接触到了四种内建 (built-in) 的数据类型：整数、浮点、字符串、列表。

整数和浮点是标量类型，即对象没有可访问的内部结构。字符串和列表是非标量类型或结构化类型，因为可以使用切片方法对它们进行切片或使用索引获取单个元素。此外，还有 2 种结构化类型：元组 (tuple)、字典 (dict)。

10.1 元组

与字符串、列表一样，元组 (tuple) 也是有序序列 (sequence)。字符串和元组都是不可变类型。与字符串不同的是，元组中的元素可以是任意类型且可以混合在一起，这一点与列表一样，但列表是可变类型。简单地说，元组就是不可变的列表。语法上，元组使用圆括号，而列表使用方括号，元素之间都使用逗号分割。与字符串和列表一样，元组可以被组合、索引和切片。如果不涉及到元素的修改，优先使用元组——数据更加安全，效率要优于列表。下面是元组的写法：

```
empty = ()
one = (10,)
jobs = ('engineer', 'soft developer', 'doctor')
student = ('John', 20, 120)
```

注意，只包含一个元素的元组写法[27]。

下面看一下元组的几个操作：

```
>>> t1 = ('John', 10, 'Zelle', 20)
>>> t2 = ('Max', 30)
>>> t1 + t2
('John', 10, 'Zelle', 20, 'Max', 30)
>>> (t1 + t2)[2]
'Zelle'
>>> (t1 + t2)[0:3]
('John', 10, 'Zelle')
```

[27](10) 是整数 10 的另一种繁琐写法，此处的圆括号用来组合表达式。

下面再来看一个有趣的元组操作：

```
>>> t3 = ('1', '2', '3', [1, 2, 3])
>>> t3
('1', '2', '3', [1, 2, 3])
>>> t3[3].append(4)
>>> t3
('1', '2', '3', [1, 2, 3, 4])
```

可以看出，即便元组是不可变的，也可以让元组被“改变”——其实，元组及其元素对象 (ID 号) 本身没有发生变化，只是它的第 4 个元素是一个列表对象，它是可变的。

10.2 字典

前面介绍的字符串、指针与元组，被称为是序列数据，原因在于，可以使用索引访问其中的元素。

在许多应用中，需要一种更加灵活的方式来访问元素，例如，根据用户名获取电话号码、密码或其它信息，根据国家名获取首都名，根据商品名获取价格信息等。将这种数据存取模式抽象出来，就形成了所谓的键-值对 (key-value pair) 数据结构——利用键获取对应的值，这种关系被称为映射 (mapping)。Python 中具有映射关系的数据结构是字典。在其它的程序设计语言中，例如 C++，它们被称为 HASH 表或关联数组。

字典的语法定义如下：

```
<variable> = {key1:value1, key2:value2, ...}
```

注意，键可以是任意的不可变数据类型[28]，值可以是任意类型，包括自定义类型。例如：

```
>>> d = {1:'1',2:'2'}
>>> d[1]
'1'
>>> d[2]
```

[28] 字典使用哈希算法 (Hash Algorithm) 将键转变成值的存储位置，因此要确保键是不可变对象。

```
'2'
>>> p = {'John':12345, 'guido':354590, 'turing':32344}
>>> p['John']
12345
>>> p['guido']
354590
```

可以看出，使用 <dictionary>[<key>] 可以获得与键对应的值。由于字典是可变的，因此，该条语句也可以放在表达式的左边进行赋值：

```
>>> p = {'John':12345, 'guido':354590, 'turing':32344}
>>> p['turing'] = 789
>>> p
{'John': 12345, 'turing': 789, 'guido': 354590}
```

从上例可以看出，与键 “turing” 关联的值已经被修改了；同时，也观察到，字典的元素顺序发生了变化。字典类型在本质上是无序的，Python 以有效的方式存储这些键。

在赋值的时候，如果没有找到指定的键，Python 将自动新增该项：

```
>>> p['Test'] = 567
>>> p
{'Test': 567, 'John': 12345, 'turing': 789, 'guido': 354590}
```

下面的程序片段从文件中读取键-值数据并建立字典：

```
passwd = {}
for line in open('passwords','r'):
    user, pw = line.split()
    passwd[user] = pw
```

图10-32列出了字典的可用方法。下面的例子使用了字典的相关方法：

```
>>> p
{'Test': 567, 'John': 12345, 'turing': 789, 'guido': 354590}
>>> list(p.keys())
```

method	meaning
<key> in <dict>	Returns true if dictionary contains the specified key, false if it doesn't.
<dict>.keys()	Returns a sequence of keys.
<dict>.values()	Returns a sequence of values.
<dict>.items()	Returns a sequence of tuples (key,value) representing the key-value pairs.
<dict>.get(<key>, <default>)	If dictionary has key returns its value; otherwise returns default.
del <dict>[<key>]	Deletes the specified entry.
<dict>.clear()	Deletes all entries.
for <var> in <dict>:	Loops over the keys.

图 10-32: 字典的可用方法

```
['Test', 'John', 'turing', 'guido']
>>> list(p.values())
[567, 12345, 789, 354590]
>>> list(p.items())
[('Test', 567), ('John', 12345), ('turing', 789), ('guido', 354590)]
>>> 'Test' in p
True
>>> 'test' in p
False
>>> p.get('Test', 'unknown')
567
>>> p.get('test', 'unknown')
'unknown'
>>> p.clear()
>>> p
{}
```

在应用中，可以使用 “for key in dict” 结构遍历字典 dict 中的元素。默认情况下，该结构遍历字典中的 key。如果要遍历 value，可以使用 “for value in dict.values()”

结构；如果要同时遍历 key 和 value，可以使用 “for k, v in dict.items()” 结构。

10.3 集合

集合 (set) 和字典类似，也是一组 key 的集合，但是它不存储 value，并且不允许 key 重复。例如：

```
>>> s2 = set(['a', 'b', 'c', 'd', 'a', 'b'])
>>> s2
{'a', 'd', 'b', 'c'}
```

可以看出，重复的元素将被自动过滤掉，并且元素不是有序的。常用的方法如下：

- add(key);
- remove(key);
- &

  计算交集：

  ```
  >>> s1 = set(['a', 'b', 'c'])
  >>> s3 = set(['b', 'c', 'q'])
  >>> s1 & s3
  {'b', 'c'}
  ```

- |

  计算并集：

  ```
  >>> s1 | s3
  {'b', 'a', 'q', 'c'}
  ```

集合和字典的唯一区别在于，前者没有存储对应的 value，而只保留了 key。与字典对 key 的要求一样，集合中也不允许 key 是可变对象，因为无法判断两个可变对象是否相同，也就无法保证集合内部是否存在相同的元素。

10.4 堆

堆 (Heap) 是一种树形数据结构，它的父节点与子节点之间具有某种顺序关系。二叉堆 (Binary Heap) 可以使用列表或数组来表示，第 N 个元素的孩子节点位于 $2*N+1$ 和 $2*N+2$ 中 (按照惯例，索引从 0 开始)。这种表示方法的优点在于，可以使用原址 (in-place) 方式重新对堆进行排序，也不需要预先分配大块内存，可以很方便地添加和删除元素。

堆有最大堆 (Max Heap) 和最小堆 (Min Heap) 两种。在最大堆中，父节点的值大于或等于其子节点的值，而在最小堆中，父节点的值小于或等于其子节点的值。Python 的 heapq 实现了最小堆。

堆的操作如下：

- heappush(list, item)

  向堆中插入一个元素；

- heappop(list)

  删除堆顶元素；

- heapify(list)

  将一个列表转换成堆；

- heapreplace(heap, item)

  删除堆顶元素，然后插入一个新元素；

- nlargest(n, heap)

  返回最大的 n 个数；

- nsmallest(n, heap)

  返回最小的 n 个数；

实例如下：

```
>>> import heapq
>>> import random
>>> heap = []
```

```
>>> d = list(range(10))
>>> random.shuffle(d)
>>> d
[4, 8, 1, 9, 6, 5, 3, 2, 0, 7]
>>> for n in d:
        heapq.heappush(heap, n)
>>> heap
[0, 1, 3, 2, 7, 5, 4, 9, 6, 8]
>>> heapq.heappop(heap)
0
>>> heap
[1, 2, 3, 6, 7, 5, 4, 9, 8]
>>> heapq.nlargest(5,heap)
[9, 8, 7, 6, 5]
>>> heapq.nsmallest(5,heap)
[1, 2, 3, 4, 5]
>>> list=[15, 18, 10, 13, 6, 7, 9, 1, 4, 2]
>>> heapq.heapify(list)
>>> list
[1, 2, 7, 4, 6, 10, 9, 13, 18, 15]
```

10.5 迭代器

首先介绍一下可迭代 (Iterable) 对象与迭代器 (Iterator) 之间的区别。

可迭代对象可以置于 for 循环中，使用 “for x in obj” 结构遍历元素，有 2 种对象属于可迭代对象：

- 结构化数据类型

  例如，字符串、列表、元组、字典、集合等；

- 生成器

  例如，列表生成器及带 yield 的生成器函数；

可以使用内建函数 isinstance 判断一个对象是否可迭代：

```
>>> from collections import Iterable
>>> isinstance([1,2,3], Iterable)
True
>>> isinstance({'1':1,'2':2,'3':3}, Iterable)
True
>>> isinstance('abc', Iterable)
True
>>> isinstance((x for x in range(10)), Iterable)
True
>>> isinstance([x for x in range(10)], Iterable)
True
>>> isinstance(100, Iterable)
False
```

迭代器不但可以置于 for 循环中，也可以被内建函数 next 不断地调用而产生新元素 (直到无法产生新元素而抛出 StopIteration 异常为止)，也可以使用 isinstance 判断对象是否迭代器：

```
>>> from collections import Iterator
>>> isinstance((x for x in range(10)), Iterator)
True
>>> isinstance([1,2,3], Iterator)
False
>>> isinstance({'1':1,'2':2,'3':3}, Iterator)
False
>>> isinstance('abc', Iterator)
False
```

可以看出，字符串、列表、字典等结构化对象是可迭代对象，却不是迭代器；生成器是迭代器[29]。但是，可以使用内建函数 iter 将可迭代对象转换成迭代器：

[29] 一般而言，迭代器可以用来表示元素长度未知的数据流，而可迭代对象的长度是已知的。

```
>>> isinstance(iter([1,2,3]), Iterator)
True
>>> isinstance(iter('abc'), Iterator)
True
```

10.6 实例

10.6.1 统计分析程序

在统计应用中，经常需要计算一组数据的平均值 (Mean)、标准差 (Standard Deviation) 及中值 (Median)。

平均值 $\bar{x}$、标准差 s 的定义如下：

$$\begin{aligned} \bar{x} &= \frac{1}{n}\sum_{i=1}^{n} x_i \\ s &= \sqrt{\frac{\sum_{i=1}^{n}(\bar{x}-x_i)^2}{n-1}} \end{aligned} \tag{1}$$

中值将一组数据分成项数相等的 2 部分，例如，数据集 $[1,2,4,40,50,60]$ 的中值是 $\frac{4+40}{2}=22$，而数据集 $[1,2,40,50,60]$ 的中值是 40。此例，可以使用列表作为数据结构，用来存放一组数据。

下面是完整的程序：

```
1  # stats.py
2  from math import sqrt
3
4  def getNumbers():
5      nums = []      # start with an empty list
6
7      # sentinel loop to get numbers
8      xStr = input("Enter a number (<Enter> to quit) >> ")
9      while xStr != "":
10         x = float(xStr)
11         nums.append(x)   # add this value to the list
12         xStr = input("Enter a number (<Enter> to quit) >> ")
```

```
13      return nums
14
15  def mean(nums):
16      total = 0.0
17      for num in nums:
18          total = total + num
19      return total / len(nums)
20
21  def stdDev(nums, xbar):
22      sumDevSq = 0.0
23      for num in nums:
24          dev = num - xbar
25          sumDevSq = sumDevSq + dev * dev
26      return sqrt(sumDevSq/(len(nums)-1))
27
28  def median(nums):
29      nums.sort()
30      size = len(nums)
31      midPos = size // 2
32      if size % 2 == 0:
33          med = (nums[midPos] + nums[midPos-1]) / 2.0
34      else:
35          med = nums[midPos]
36      return med
37
38  def main():
39      print("This program computes mean, median and standard deviation.")
40
41      data = getNumbers()
42      xbar = mean(data)
43      std = stdDev(data, xbar)
```

```
44      med = median(data)
45
46      print("\nThe mean is", xbar)
47      print("The standard deviation is", std)
48      print("The median is", med)
49
50  if __name__ == '__main__': main()
```

10.6.2 词频分析程序

文本词频分析程序在网页搜索引擎、深度学习与自然语言处理等领域有着广泛的应用。它的主要功能是统计文本中不同词的个数与出现频率，为文本相似度分析、语料数据集的建立提供辅助信息。

从实现的角度看，这是一个多累加器问题：统计每一个单词在文件中出现的次数。算法的步骤是，遍历文件中的每一个单词，并修改相应的计数值。此例可以使用字典作为数据结构——单词作为键，次数作为值。

下面是完整的程序：

```
1   # wordfreq.py
2
3   def byFreq(pair):
4       return pair[1]
5
6   def main():
7       print("This program analyzes word frequency in a file")
8       print("and prints a report on the n most frequent words.\n")
9
10      # get the sequence of words from the file
11      fname = input("File to analyze: ")
12      text = open(fname,'r').read()
13      text = text.lower()
14      for ch in '!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~':
15          text = text.replace(ch, ' ')
```

```
16      words = text.split()
17
18      # construct a dictionary of word counts
19      counts = {}
20      for w in words:
21          counts[w] = counts.get(w,0) + 1
22
23      # output analysis of n most frequent words.
24      n = int(input("Output analysis of how many words? "))
25      items = list(counts.items())
26      items.sort()
27      items.sort(key=byFreq, reverse=True)
28      for i in range(n):
29          word, count = items[i]
30          print("{0:<15}{1:>5}".format(word, count))
31
32  if __name__ == '__main__':  main()
```

函数 byFreq 用于列表的排序方法 sort：列表元素是一个元组，指定元组的第 2 个元素作为排序关键字[30]。第 21 行的 get 方法，隐含了一个选择结构：如果在字典 counts 中没有找到指定键，就返回值 0，否则返回该键对应的 (计数) 值。第 26 行，sort 方法将按缺省方式 (元组的第 1 个元素作为关键字) 排序，即以单词作为排序依据。第 27 行，按元组的第 2 个元素排序，即以次数作为排序依据。

10.6.3 学生成绩分析程序

本程序从数据文件中读入每个学生的信息：name、credit hours、quality points(每一行存放一条学生信息),需要计算每个学生的 GPA(Grade Point Average)——(quality points)/(credit hours)，最后将学生信息按 GPA 从低到高进行排序并保存到文件中。

下面是文件 gpa.py 和 gpasort.py 的内容：

[30]sort 方法的调用格式为：<list>.sort(key = <key_function>)。key_function 是一个函数，它的参数必须是 <list> 的元素，返回值是元素中的某项 (关键字)。

```
# gpa.py
#    Program to find student with highest GPA

class Student:

    def __init__(self, name, hours, qpoints):
        self.name = name
        self.hours = float(hours)
        self.qpoints = float(qpoints)

    def getName(self):
        return self.name

    def getHours(self):
        return self.hours

    def getQPoints(self):
        return self.qpoints

    def gpa(self):
        return self.qpoints/self.hours

def makeStudent(infoStr):
    # infoStr is a tab-separated line: name hours qpoints
    # returns a corresponding Student object
    name, hours, qpoints = infoStr.split("\t")
    return Student(name, hours, qpoints)

def main():
    # open the input file for reading
    filename = input("Enter name the grade file: ")
```

```
32      infile = open(filename, 'r')
33
34      # set best to the record for the first student in the file
35      best = makeStudent(infile.readline())
36
37      # process subsequent lines of the file
38      for line in infile:
39          # turn the line into a student record
40          s = makeStudent(line)
41          # if this student is best so far, remember it.
42          if s.gpa() > best.gpa():
43              best = s
44
45      infile.close()
46
47      # print information about the best student
48      print("The best student is:", best.getName())
49      print("hours:", best.getHours())
50      print("GPA:", best.gpa())
51
52  if __name__ == '__main__':
53      main()
```

```
1   # gpasort.py
2   #    A program to sort student information into GPA order.
3
4   from gpa import Student, makeStudent
5
6   def readStudents(filename):
7       infile = open(filename, 'r')
8       students = []
9       for line in infile:
```

```
10          students.append(makeStudent(line))
11      infile.close()
12      return students
13
14
15  def writeStudents(students, filename):
16      outfile = open(filename, 'w')
17      for s in students:
18          print("{0}\t{1}\t{2}".format(s.getName(), s.getHours(), s.getQPoints()),
19                file=outfile)
20      outfile.close()
21
22
23  def main():
24      print("This program sorts student grade information by GPA")
25      filename = input("Enter the name of the data file: ")
26      data = readStudents(filename)
27      data.sort(key=Student.gpa)
28      filename = input("Enter a name for the output file: ")
29      writeStudents(data, filename)
30      print("The data has been written to", filename)
31
32  if __name__ == '__main__':
33      main()
```

程序 gpasort.py 的第 27 行使用 Student 类的 gpa 方法作为参数传入 sort，表示使用 gpa 作为排序关键字对列表对象 data 进行排序 (该列表的元素是 Student 对象)。

10.6.4 计算器

计算器程序有一些 Button 对象，分别用来实现数字 (0-9)、小数点、运算符 (+-*/)、C(清除显示)、<-(后退删除)、=(计算) 等功能。运行界面如图10-33所示。

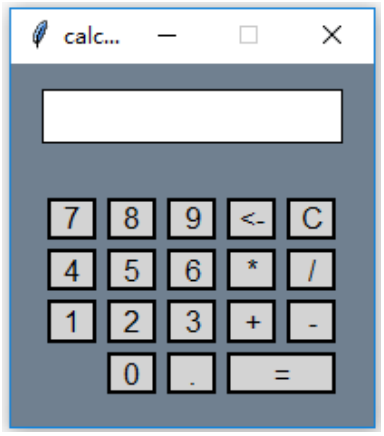

图 10-33: 计算器

完整的计算器程序如下:

```
# calc.pyw -- A four function calculator using Python arithmetic.
#     Illustrates use of objects and lists to build a simple GUI.

from graphics import *
from button import Button

class Calculator:
    # This class implements a simple calculator GUI

    def __init__(self):
        # create the window for the calculator
        win = GraphWin("calculator")
        win.setCoords(0,0,6,7)
        win.setBackground("slategray")
        self.win = win
        # Now create the widgets
        self.__createButtons()
        self.__createDisplay()

```

```
20      def __createButtons(self):
21          # create list of buttons
22          # start with all the standard sized buttons
23          # bSpecs gives center coords and label of buttons
24          bSpecs = [(2,1,'0'), (3,1,'.'),
25                    (1,2,'1'), (2,2,'2'), (3,2,'3'), (4,2,'+'), (5,2,'-'),
26                    (1,3,'4'), (2,3,'5'), (3,3,'6'), (4,3,'*'), (5,3,'/'),
27                    (1,4,'7'), (2,4,'8'), (3,4,'9'), (4,4,'<-'),(5,4,'C')]
28          self.buttons = []
29          for (cx,cy,label) in bSpecs:
30              self.buttons.append(Button(self.win,Point(cx,cy),.75,.75,label))
31          # create the larger = button
32          self.buttons.append(Button(self.win, Point(4.5,1), 1.75, .75, "="))
33          # activate all buttons
34          for b in self.buttons:
35              b.activate()
36
37      def __createDisplay(self):
38          bg = Rectangle(Point(.5,5.5), Point(5.5,6.5))
39          bg.setFill('white')
40          bg.draw(self.win)
41          text = Text(Point(3,6), "")
42          text.draw(self.win)
43          text.setFace("courier")
44          text.setStyle("bold")
45          text.setSize(16)
46          self.display = text
47
48      def getButton(self):
49          # Waits for a button to be clicked and returns the label of
50          #    the button that was clicked.
```

```
51          while True:
52              p = self.win.getMouse()
53              for b in self.buttons:
54                  if b.clicked(p):
55                      return b.getLabel() # method exit
56
57      def processButton(self, key):
58          # Updates the display of the calculator for press of this key
59          text = self.display.getText()
60          if key == 'C':
61              self.display.setText("")
62          elif key == '<-':
63              # Backspace, slice off the last character.
64              self.display.setText(text[:-1])
65          elif key == '=':
66              # Evaluate the expresssion and display the result.
67              # the try...except mechanism "catches" errors in the
68              # formula being evaluated.
69              try:
70                  result = eval(text)
71              except:
72                  result = 'ERROR'
73              self.display.setText(str(result))
74          else:
75              # Normal key press, append it to the end of the display
76              self.display.setText(text+key)
77
78      def run(self):
79          # Infinite 'event loop' to process button clicks.
80          while True:
81              key = self.getButton()
```

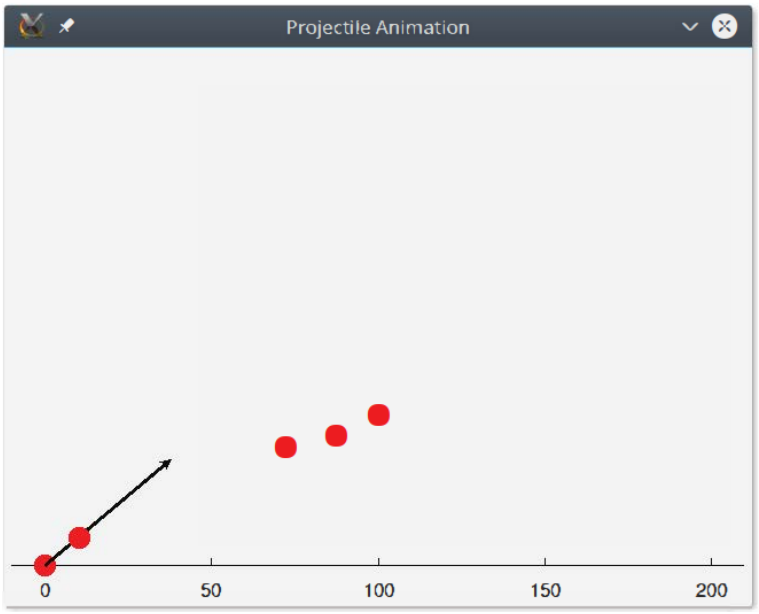

图 10-34: 改进的 Animated Cannonball

```
82                self.processButton(key)
83
84    # This runs the program.
85    if __name__ == '__main__':
86        # First create a calculator object
87        theCalc = Calculator()
88        # Now call the calculator's run method.
89        theCalc.run()
```

10.6.5 改进的 Animated Cannonball

本节对9.4.2给出的程序进行了改进，利用列表跟踪多个 Projectile 对象，可以同时支持这些对象的动画显示，如图10-34所示。

下面是完整的程序：

```
1   # animation2.py
2
3   # multiple-shot cannonball animation
4
```

```
5  from math import sqrt, sin, cos, radians, degrees
6  from graphics import *
7  from projectile import Projectile
8  from button import Button
9
10 class Launcher:
11
12     def __init__(self, win):
13         """Create inital launcher with angle 45 degrees and velocity 40
14         win is the GraphWin to draw the launcher in.
15         """
16
17         # draw the base shot of the launcher
18         base = Circle(Point(0,0), 3)
19         base.setFill("red")
20         base.setOutline("red")
21         base.draw(win)
22
23         # save the window and create initial angle and velocity
24         self.win = win
25         self.angle = radians(45.0)
26         self.vel = 40.0
27
28         # create inital "dummy" arrow
29         self.arrow = Line(Point(0,0), Point(0,0)).draw(win)
30         # replace it with the correct arrow
31         self.redraw()
32
33
34     def redraw(self):
35         """undraw the arrow and draw a new one for the
```

```
36          current values of angle and velocity.
37          """
38
39          self.arrow.undraw()
40          pt2 = Point(self.vel*cos(self.angle), self.vel*sin(self.angle))
41          self.arrow = Line(Point(0,0), pt2).draw(self.win)
42          self.arrow.setArrow("last")
43          self.arrow.setWidth(3)
44
45
46      def adjAngle(self, amt):
47          """ change angle by amt degrees """
48
49          self.angle = self.angle+radians(amt)
50          self.redraw()
51
52
53      def adjVel(self, amt):
54          """ change velocity by amt"""
55
56          self.vel = self.vel + amt
57          self.redraw()
58
59      def fire(self):
60          return ShotTracker(self.win, degrees(self.angle), self.vel, 0.0)
61
62
63  class ShotTracker:
64
65      """ Graphical depiction of a projectile flight using a Circle """
66
```

```
67      def __init__(self, win, angle, velocity, height):
68          """win is the GraphWin to display the shot, angle, velocity, and
69          height are initial projectile parameters.
70          """
71
72          self.proj = Projectile(angle, velocity, height)
73          self.marker = Circle(Point(0,height), 3)
74          self.marker.setFill("red")
75          self.marker.setOutline("red")
76          self.marker.draw(win)
77
78
79      def update(self, dt):
80          """ Move the shot dt seconds farther along its flight """
81
82          self.proj.update(dt)
83          center = self.marker.getCenter()
84          dx = self.proj.getX() - center.getX()
85          dy = self.proj.getY() - center.getY()
86          self.marker.move(dx,dy)
87
88
89      def getX(self):
90          """ return the current x coordinate of the shot's center """
91          return self.proj.getX()
92
93      def getY(self):
94          """ return the current y coordinate of the shot's center """
95          return self.proj.getY()
96
97      def undraw(self):
```

```
98          """ undraw the shot """
99          self.marker.undraw()
100
101
102 class ProjectileApp:
103
104     def __init__(self):
105         self.win = GraphWin("Projectile Animation", 640, 480)
106         self.win.setCoords(-10, -10, 210, 155)
107         Line(Point(-10,0), Point(210,0)).draw(self.win)
108         for x in range(0, 210, 50):
109             Text(Point(x,-7), str(x)).draw(self.win)
110             Line(Point(x,0), Point(x,2)).draw(self.win)
111
112         self.launcher = Launcher(self.win)
113         self.shots = []
114
115     def updateShots(self, dt):
116         alive = []
117         for shot in self.shots:
118             shot.update(dt)
119             if shot.getY() >= 0 and shot.getX() < 210:
120                 alive.append(shot)
121             else:
122                 shot.undraw()
123         self.shots = alive
124
125     def run(self):
126
127         # main event/animation lopp
128         while True:
```

```
129             self.updateShots(1/30)
130
131             key = self.win.checkKey()
132             if key in ["q", "Q"]:
133                 break
134
135             if key == "Up":
136                 self.launcher.adjAngle(5)
137             elif key == "Down":
138                 self.launcher.adjAngle(-5)
139             elif key == "Right":
140                 self.launcher.adjVel(5)
141             elif key == "Left":
142                 self.launcher.adjVel(-5)
143             elif key == "f":
144                 self.shots.append(self.launcher.fire())
145
146             update(30)
147
148         self.win.close()
149
150
151 if __name__ == "__main__":
152     ProjectileApp().run()
```

10.7 练习

1. 给程序 stats.py 添加图形界面；

2. 给程序 wordfreq.py 添加图形界面；

3. 给程序 gpasort.py 添加图形界面；

4. 在使用列表的 sort 方法时，如果给它指定自定义遍历元素的函数 (例如 key=Student.gpa)，程序的执行效率将会降低。此时，有一种解决办法，例如，在 gpasort.py 中，首先可以构造一个列表：[(gpa0, Student0), (gpa1, Student1), ...]，然后使用缺省的 sort 方法排序 (即不指定 key 参数，缺省以元组的第 1 个元素作为关键字排序)，最后再遍历排序后的列表，取出其中的 Student 对象重新生成一个新列表对象。使用该方法重新编写程序 gpasort.py；

5. 创建一个类 StatSet 并测试，该类提供简单的统计计算服务，方法如下：

   (a) `__init__(self)`

   创建一个 StatSet 对象，无初始化数据；

   (b) addNumber(self, x)

   添加数据 x；

   (c) mean(self)

   返回均值；

   (d) median(self)

   返回中值；

   (e) stdDev(self)

   返回标准差；

   (f) count(self)

   返回数据个数；

   (g) min(self)

   返回最小值；

   (h) max(self)

   返回最大值；

6. 继续改进程序 animation2.py，使之更像一个游戏，更具可玩性。例如，添加一个或多个可击中的目标，记录炮弹击中目标的次数并显示，可以考虑让目标移动，还可以考虑增加其它有趣的功能；

11 函数式编程初步

11.1 概述

到目前为止，我们已经接触过 2 种常见的程序设计方法：面向过程的程序设计 (过程式程序设计/编程) 与面向对象的程序设计 (对象式程序设计/编程)。实际上，还有一种函数式程序设计方法 (函数式程序设计/编程，Functional Programming)。

过程式编程的基本单元是函数，它是各类程序设计语言直接支持的基本模块。在系统开发过程中，首先依据需求分析进行系统的总体设计，将复杂的任务分解成简单的子任务，子任务一直分解到易于实现为止，这个过程被称为是自顶向下的分析与设计方法。在实际开发过程中，需要将自顶向下的设计与自底向上的实现方法结合在一起，不断地迭代，最终实现一个满足需求的应用系统。

对象式编程的基本单元是类与对象，它也是面向对象程序设计语言直接支持的基本模块。类与对象是数据和操作的封装体，与系统中的客观实体有着直接的对应关系，具有直观性与一致性特点。此外，类具有如下几个性质：抽象性——类定义了一组具有相似性质的对象、封装性——数据与操作的集成、继承性——子类可以继承父类的数据与操作、多态性——抽象性的具体表现，父类与子类具有相似的操作，但具体的操作又不一样。虽然对象式编程具有很多优点，但是在实际开发过程中，仍然需要将自顶向下的设计与自底向上的实现这 2 个过程结合在一起；否则，直接采用对象式开发方法，同样会造成系统结构不合理不协调等现象。因此，过程式与对象式方法是相互依存、互不替代的关系。

虽然可以将函数式编程归结为过程式方法，但是，它是一种抽象程度更高的编程范式。函数式编程的思想源自于数学理论，它是一种面向数学的编程范式，将计算描述为一种表达式求值，它似乎更适用于数学计算的应用场景。函数式编程中的“函数”概念与程序设计语言中“函数”概念的含义不一样，前者更接近于数学中的“函数”概念——即自变量到因变量的映射关系，而后者指的是一个相对独立的、一般具有名称的代码模块或计算过程 (Subroutine)。

以 Python 为例，在过程式编程中，最常用的基本语句有：

- 函数定义 (def)；
- 条件控制 (if、if-else、if-elif-else)；
- 循环控制 (for、break、continue、while)；

Python 也支持函数式编程[31]，最常用的基本函数和算子有：

- 基本函数 (map、reduce、filter)；
- 基本算子 (lambda)；

实际上，利用上述基本函数和算子就可以实现任意的 Python 程序[32]。

本节将介绍函数式编程的简单内容：函数式编程的基本函数与算子、将过程式编程中的条件与循环控制结构转换到函数式结构。欲掌握更多的函数式编程知识或纯函数式编程范式，请阅读专门介绍函数式编程范式的书籍及学习纯函数式编程语言。

11.2 lambda 算子

lambda 算子或 lambda 函数用于创建微型的、一次性使用的匿名函数，它的基本语法如下：

```
lambda arguments : expression
```

该算子可以有任意个参数，但只能有一个表达式。实例：

```
>>> t = lambda x,y,z:x+y+z
>>> t(1,2,3)
6
>>> t = lambda x,y,z:min(x+y,z)
>>> t(1,3,2)
2
>>> type(t)
<class 'function'>
```

实际上，lambda 算子返回的是函数对象。一般的用法是，将 lambda 算子用作高阶函数的参数，因为高阶函数需要函数对象作为参数。在上述例子中，直接将函数对象返回给变量，这种用法比较少见。

[31]Python 是一种多编程范式语言，同时支持过程式编程、对象式编程与函数式编程，但 Python 不是纯函数式编程语言。

[32]Python 不是纯函数式程序设计语言，也没有必要一定要将程序写成函数式风格。然而，在程序的局部使用这些基本函数与算子却能够大大简化代码。因此，掌握函数式编程的基本方法还是非常有意义的。

11.3 高阶函数

在程序设计语言中，高阶函数 (Higher-order function) 是指将函数作为参数的函数。

在 Python 中，一切皆对象，数据与函数也不例外。例如：

```
>>> max
<built-in function max>
>>> min
<built-in function min>
>>> id(max)
2297893586480
>>> id(min)
2297893586552
```

max 和 min 本身是内建函数[33]，它们也有自己的 ID 号。函数与数据一样，可以赋值给变量，也可以当作参数传入函数。

下面实现一个简单的高阶函数：

```
def higher_fun(x, y, f):
    return f(x, int) and f(y, int)


print(higher_fun(10, 20, isinstance))
print(higher_fun(10, '1', isinstance))
```

当然，该程序非常简单，实现的功能也不复杂，只是为了展示高阶函数的基本形态。

Python 定义了几个高阶函数，它们分别是：map、reduce、filter、sorted，下面将一一介绍。

11.3.1 map

map 的语法形式如下：

```
map(function, iterable, ...)
```

[33]内建函数被定义在模块 builtins 中，缺省已被 Python 导入 (import builtins)。

它接收 2 种类型的参数：函数和可迭代对象。map 函数的作用是，将 function 作用于可迭代对象的每一项——如果传入多个可迭代对象，则 function 将并行作用于它们中的每一个元素；如果多个可迭代对象的长度不一，map 函数简单地忽略掉多余的元素[34]，map 的执行结果作为迭代器 (Iterator) 返回。例如：

```
>>> from collections import Iterator
>>> isinstance(map(max,[1,2,3],[2,3,4]),Iterator)
True
>>> list(map(max,[1,2,3],[2,3,4]))
[2, 3, 4]
```

对于迭代器对象，可以使用 list 函数显式地生成一个列表对象。实现相同功能的一个简单程序如下：

```
def multimax(x, y):
    l = []
    for i in range(len(x)):
        l.append(max(x[i], y[i]))
    return l


print(multimax([1,2,3], [2,3,4]))
```

将两种写法进行比较，可以看出，map 函数写法简洁，熟悉 Python 的程序员能够做到“一眼知其指意”[35]。

11.3.2 reduce

reduce 的语法形式如下：

```
reduce(function, iterable[, initializer])
```

[34] 不同版本之间的处理方式存在差异，先前版本的处理方式是“不足的元素用 None 替代”。

[35] 虽然在底层实现上，map 高阶函数也是采用相似或类似的方式实现的，但是对于程序员来讲，语句的易用性、抽象性更高的数据与函数处理方式一直是他们孜孜以求的目标——这些特性使得程序员可以站在更高的抽象层次上思考与求解问题，不至于陷入大量的细节上。从程序设计语言的历史、发展过程及趋势可以看出这一点。

它按如下方式将 function 作用于可迭代对象，假设 f 表示 function，可迭代对象为一个序列 (例如列表和元组)：$x_1, x_2, ..., x_n$：

$$f(f(f(f(x_1, x_2), x_3), x_4), ...x_n), \tag{2}$$

函数 f 接收 2 个参数。注意，在 Python3 中，reduce 不是内建函数，它已经被移入了 functools 包中。实例如下：

```
>>> from functools import reduce
>>> reduce(lambda x,y:x*y,[1,2,3])
6
>>> reduce(lambda x,y:x+y,[1,2,3,4,5])
15
>>> reduce(lambda x,y:x*y,(),3)
3
>>> reduce(lambda x,y:x*y,[1,2,3],3)
18
>>> reduce(lambda x,y:x*10+y,[1,2,3,4,5])
12345
>>> reduce(lambda x,y:x*10+y,[1,2,3,4,5],6)
612345
```

参数 *initializer*(如果提供) 充当一个缺省元素放在所有序列元素的前面并参与计算。

11.3.3 filter

filter 的语法形式如下：

```
filter(function, iterable)
```

从可迭代对象中过滤出满足条件的元素 (函数 function 返回 True 的那些元素) 序列，并作为迭代器对象返回。它类似于执行下面的语句[36]：

[36]两者并不完全等价。因为 filter 返回迭代器 (Iterator) 对象，而示例返回的是列表对象 (Iterable)，两者并不相同，参见10.5。

```
[item for item in iterable if function(item)]
```

如果 function 不是 None 参数；否则，

```
[item for item in iterable if item]
```

例如：

```
>>> list(filter(None,[0,1,2]))
[1, 2]
>>> list(filter(lambda x:x%2==1, [1, 2, 4, 5, 6, 9, 10, 15]))
[1, 5, 9, 15]
```

11.3.4 sorted

Python 有一个内建函数 sorted，用于对象排序。其语法形式如下：

```
sorted(iterable[, key[, reverse]])
```

关键字参数 key 指定用于数据预处理的函数，它作用于可迭代对象的每一个元素上，并根据返回结果进行排序。关键字参数 reverse = False，表示按从小到大顺序排序 (缺省情况)；否则，reverse = True，表示反向排序。

在一些情况下，可以直接使用该函数对原始数据进行排序：

```
>>> sorted([2,1,4,3])
[1, 2, 3, 4]
```

在另一些情况下，排序前需要对数据进行预处理。例如：

```
>>> sorted([2,1,-4,3],key=abs)
[1, 2, 3, -4]
```

可以看出，该数据序列是按绝对值大小排序的。再看一个字典 (转换为元组列表[37]) 排序的例子：

```
>>> sorted({2:'b',1:'c',-4:'a',3:'d'}.items())
[(-4, 'a'), (1, 'c'), (2, 'b'), (3, 'd')]
```

[37] 因为字典对象是无序的。

```
>>> sorted({2:'b',1:'c',-4:'a',3:'d'}.items(),key=lambda x:x[0])
[(-4, 'a'), (1, 'c'), (2, 'b'), (3, 'd')]
>>> sorted({2:'b',1:'c',-4:'a',3:'d'}.items(),key=lambda x:x[1])
[(-4, 'a'), (2, 'b'), (1, 'c'), (3, 'd')]
```

缺省情况下，按元组的第 1 个元素大小排序，除非使用 key 参数额外指定。

11.4 闭包

函数是过程式编程的基本模块，类与对象是对象式编程的基本模块。使用函数、类与对象的目的在于，简化系统的分析、设计与实现——将自顶向下和自底向上两种方式结合起来，以某种逻辑方式 (结构化和/或面向对象) 将基本模块有机地组织在一起。类与对象的概念，引入了抽象性、层次性、继承性与多态性，也有利于提高代码的复用性 (reusability)。闭包 (Closure) 函数也是一种组织代码的语法结构，它是函数式编程中的重要模块。闭包能够将函数和环境变量结合在一起，也能够有效地提高代码的复用性。

在详细介绍闭包之前，先来看一个程序：

```
1  from functools import reduce
2  def mysum(*num1):
3      lv = 11
4      def mymax(*num2):
5          num = reduce(lambda x,y:x+y, num1)
6          return max(lv, num, max(num2))
7
8      return mymax
9
10 f1 = mysum(1,2,3,4,5)
11 f2 = mysum(1,1,2,2,3)
12 del mysum
13 print(f1(2,4,6,8,10))
14 print(f2(2,4,6,8,10))
15 print(f1.__closure__[0].cell_contents)
```

```
16 print(f1.__closure__[1].cell_contents)
17 print(f2.__closure__[0].cell_contents)
18 print(f2.__closure__[1].cell_contents)
```

它的运行结果如下：

```
15
11
11
(1, 2, 3, 4, 5)
11
(1, 1, 2, 2, 3)
```

程序的第 2-8 行定义了函数 mysum，第 4-6 行定义了它的嵌套函数 mymax。该嵌套函数可以访问外层函数 mysum 的局部变量 (lv) 和参数 num1，这些变量相对于 mymax 的作用域而言，属于外部变量。但是，在 Python 中，这些外部变量属于嵌套函数的环境变量——嵌套函数与环境变量一起就形成了闭包[38]。关于访问环境变量的方法，请参考程序的第 15-18 行。

创建闭包，要注意以下几个方面：

- 必须定义嵌套函数；
- 如果嵌套函数必须引用外层函数的值[39]，最好不要引用外层函数中会发生变化的变量；否则，将会产生非预期的结果；
- 外层函数必须返回嵌套函数；

闭包的优点在于，避免了全局变量的使用，提供了某种形式的数据隐藏功能。从这个角度来说，它也是一种面向对象的解决方案。一般而言，如果需要的属性与方法较少 (大多数情况下，只需要一个方法)，那么，与类相比，闭包提供了一种更优雅的解决方法。但是，如果所需属性与方法较多，还是定义类比较合适。

下面再来看一个程序：

[38]此例中，环境变量的值被保存在函数对象 f1 和 f2 的 ___closure___ 属性中，即使删除了外层函数 mysum，f1 和 f2 也不会受到任何影响。

[39]一般情况下，Python 只允许读取变量的值。

```
def line_equ(a, b):
    def line(x):
        return a*x + b
    return line

line1 = line_equ(1, 1)
line2 = line_equ(4, 5)
print(line1(5))
print(line2(5))
```

程序定义了表示直线方程的闭包line_equ，line1 和 line2 分别是 2 条直线的闭包函数。从本例可以看出，只需要改变闭包的参数，就可以获得表示不同直线的闭包函数。因此，闭包能够提高代码的复用性，并且闭包函数也能够减少参数的传递次数。

11.5 装饰器

本质上，装饰器是一个裹着“语法糖衣 (Syntactic Sugar)”的闭包，它可以作用于函数或类，从而允许它们在不需要做任何修改的情况下，给它们增加一些额外的功能——例如，插入运行日志，进行性能测试与事务处理，设置缓存与权限校验等。装饰器是处理这类问题的绝佳角色。在一些设计过程中，可以将大量与核心功能无关的对象剥离出来，做成装饰器。随后，这些装饰器可以作用于一些相似的场景，完成特定的功能，从而进一步提高了代码的复用性。

先看一个程序实例：

```
def Log(func):
    def Log_Wrapper(*args, **kp):
        print(func.__name__, 'is running')
        func(*args, **kp)
        print(func.__name__, 'is stopping')

    return Log_Wrapper

```

```
9  def DoJob(name = 'Cleaning'):
10     print('Do', name)
11
12 DoJob = Log(DoJob)
13 DoJob()
14 DoJob(name = 'Shopping')
15 DoJob(name = 'Swimming')
```

该程序还未添加“语法糖衣”，可以看出，只是定义了一个闭包 Log，一个函数 DoJob。假设 DoJob 是执行核心业务的函数，我们现在想给该函数添加一些日志或提示信息而又不想修改该函数的定义，于是将该函数对象作为参数传递给闭包 Log。在该闭包内部，先显示运行前信息，然后执行核心函数，最后再显示运行后信息。执行结果如下：

```
DoJob is running
Do Cleaning
DoJob is stopping
DoJob is running
Do Shopping
DoJob is stopping
DoJob is running
Do Swimming
DoJob is stopping
```

实际上，引入装饰器可以进一步增加程序的可读性，并且装饰器是可以嵌套使用的[40]。继续修改上面的程序：

```
1  def Log(func):
2      def Log_Wrapper(*args, **kp):
3          print(func.__name__, 'is running')
4          func(*args, **kp)
5          print(func.__name__, 'is stopping')
```

[40]详见下文。装饰器嵌套相当于闭包的嵌套调用，例如，Log2(Log1(func))，装饰器 Log1 紧贴函数 func，装饰器 Log2 紧贴 Log1，依此类推。

```
6
7       return Log_Wrapper
8
9   @Log
10  def DoJob(name = 'Cleaning'):
11      print('Do', name)
12
13  DoJob()
14  DoJob(name = 'Shopping')
15  DoJob(name = 'Swimming')
```

程序的第 9 行，添加了一个装饰器 @Log(将闭包作用于函数 DoJob)，省去了上例程序的第 12 行，其余代码没有任何变化，程序结果也完全一样。第 2 行，参数的形式为 “*args” 和 “**kp”，表明闭包函数能够接收任意参数 (前者为可变参数，后者为关键字参数)。

如果需要将额外的参数传递给闭包，先尝试不使用装饰器的写法：

```
1   def Log(func, level):
2       def Log_Wrapper(*args, **kp):
3           if level == 1:
4               print('Level Blue')
5           elif level == 2:
6               print('Level Green')
7           elif level == 3:
8               print('Level Red')
9           else:
10              print('Level Default')
11          print(func.__name__, 'is running')
12          func(*args, **kp)
13          print(func.__name__, 'is stopping')
14
15      return Log_Wrapper
16
```

```
def DoJob(name = 'Cleaning'):
    print('Do', name)

DoJob1=Log(DoJob,level=1)
DoJob1()

DoJob2=Log(DoJob,level=2)
DoJob2('Shopping')

DoJob3=Log(DoJob,level=3)
DoJob3('Swimming')
```

使用装饰器的写法如下:

```
import functools
def Log(level):
    def Log_Wrapper1(func):
        @functools.wraps(func)
        def Log_Wrapper2(*args, **kp):
            if level == 1:
                print('Level Blue')
            elif level == 2:
                print('Level Green')
            elif level == 3:
                print('Level Red')
            else:
                print('Level Default')
            print(func.__name__, 'is running')
            func(*args, **kp)
            print(func.__name__, 'is stopping')

        return Log_Wrapper2
    return Log_Wrapper1
```

```
20
21 @Log(level=1)
22 def DoJob(name = 'Cleaning'):
23     print('Do', name)
24
25 DoJob()
26 DoJob(name = 'Shopping')
27 DoJob(name = 'Swimming')
```

在使用装饰器的情况下，为传入额外的参数，在闭包 Log 内需要增加一层嵌套函数。另外，为防止内层嵌套函数的名称__name__ 等属性被覆盖[41]，需要使用 functools 的 wraps 函数保留住传入的函数名称等属性。可见，在这种情况下，装饰器的写法要复杂一些。

装饰器不仅可以作用于函数，而且也能够作用于类 (中的方法)——类装饰器需要定义__call__ 方法。实例如下：

```
1  class Log():
2      def __init__(self, func):
3          self.__func = func
4
5      def __call__(self, level, *args, **kp):
6          if level == 1:
7              print('Level Blue')
8          elif level == 2:
9              print('Level Green')
10         elif level == 3:
11             print('Level Red')
12         else:
13             print('Level Default')
14         print(self.__func.__name__, 'is running')
15         self.__func(*args, **kp)
```

[41]避免有些依赖函数签名的程序出现运行错误。

```
16          print(self.__func.__name__, 'is stopping')
17
18  @Log
19  def DoJob(name = 'Cleaning'):
20      print('Do', name)
21
22  DoJob(level=1)
23  DoJob(level=2, name = 'Shopping')
24  DoJob(level=3, name = 'Swimming')
```

下面再来看一个装饰器嵌套的例子：

```
1   def p_decorate(func):
2      def func_wrapper(name):
3          return "<p>{0}</p>".format(func(name))
4      return func_wrapper
5
6   def strong_decorate(func):
7       def func_wrapper(name):
8           return "<strong>{0}</strong>".format(func(name))
9       return func_wrapper
10
11  def div_decorate(func):
12      def func_wrapper(name):
13          return "<div>{0}</div>".format(func(name))
14      return func_wrapper
15
16  #get_text = div_decorate(p_decorate(strong_decorate(get_text)))
17  @div_decorate
18  @p_decorate
19  @strong_decorate
20  def get_text(name):
21     return "Hello {0}".format(name)
```

```
22
23 print(get_text("John"))
```

运行结果如下：

```
<div><p><strong>Hello John</strong></p></div>
```

程序的第 16 行，表示装饰器嵌套的等价写法。利用闭包参数还可以进一步优化上面的程序：

```
1  def tags(tag_name):
2      def tags_decorator(func):
3          def func_wrapper(name):
4              return "<{0}>{1}</{0}>".format(tag_name, func(name))
5          return func_wrapper
6      return tags_decorator
7
8  @tags("div")
9  @tags("p")
10 @tags("strong")
11 def get_text(name):
12     return "Hello "+name
13
14 print(get_text("John"))
```

11.6 部分函数

粗略地讲，部分函数 (Partial Function[42]) 能够预先将函数对象的某些参数设置为默认值，并生成一个与旧函数不一样的新函数 (对象)，起到简化函数调用的作用，而且也能够解决特定场合下旧函数不能被调用的情况 (下文将使用程序示例进行说明)。例如，如果旧函数有 2 个参数，而调用者只能调用 1 个参数的函数，那么，在此情况下，我们可以将该函数改造成一个部分函数。

先看一个简单的示例：

[42] 有些文献将它翻译为偏函数。实际上，它与数学上的偏函数 (请参考Partial Function Wikipedia) 概念不一样。因此，为避免混淆，将它翻译为部分函数或不完整函数。

```
1  from functools import partial
2  pmin = partial(min, 10)
3  pmax = partial(max, 100)
4
5  print(pmin(20, 30, 40))
6  print(pmax(10, 20, 30))
```

程序运行结果:

```
10
100
```

程序的第 2-3 行，定义了 2 个部分函数，实际上，它们分别为内建函数 min 和 max 设置了 2 个缺省参数 10 和 100(缺省参数将作为第 1 个元素，放置在其它参数的前面)。

下面看一个稍微复杂一点的例子:

```
1  from functools import partial
2  from random import randint
3  from math import sqrt
4
5  # create some data
6  fnx = lambda: randint(0, 10)
7  data = [(fnx(), fnx()) for c in range(10)]
8  target = (2, 4)
9
10 def euclid_dist(v1, v2):
11     x1, y1 = v1
12     x2, y2 = v2
13     return sqrt((x2 - x1)**2 + (y2 - y1)**2)
14
15 peuclid_dist = partial(euclid_dist, target)
16 data.sort(key = peuclid_dist)
17
```

```
18 # verify that it works:
19 for p in data:
20     print(round(peuclid_dist(p), 3))
```

程序的第 15 行定义了部分函数peuclid_dist，它给函数euclid_dist 指定了一个缺省参数 target(用作euclid_dist 的第 1 个参数)。第 16 行调用列表的 sort 方法对列表进行排序——函数peuclid_dist 计算每个数据点到 target 的距离并按从小到大的顺序进行排序。因为 sort 方法中的 key 只接收带 1 个参数的函数，而euclid_dist 并不能满足要求，所以需要将它改造成部分函数peuclid_dist。

12 参考文献

1. John Zelle, Python Programming: An Introduction to Computer Science, 3rd Ed., Franklin Beedle, 2016.
2. Guido van Rossum.
3. Python Wikipedia.
4. 即时编译 JIT 技术.
5. Logistic Function.
6. John V.Guttag，梁杰译。编程导论，Introduction to Computation and Programming Using Python. 人民邮电出版社，2015。
7. 廖雪峰，Python 教程。
8. Python 一切皆对象。
9. Python Objects: Mutable vs. Immutable.
10. Mutable vs Immutable Objects in Python.
11. Functional programming in Python.
12. Python 的函数式编程，从入门到“放弃”.
13. Python 闭包.

14. Python Closures.

15. Python 装饰器.

16. A guide to Python's function decorators.

17. How does the functools partial work in Python?

18. Doug Hellmann. The Python 3 Standard Library by Example. Addison-Wesley, 2017.

19. The Python Tutorial.