[笔记][从Python开始学编程]

Python

前言

作者：张腾飞
笔名：Vamei
多线程并发下载

---

1. 用编程改造世界

1.1 从计算机到编程

《模仿游戏》英国数学家阿兰·图灵`Alan Mathison Turing`破解德国密码机，用的是机电式的计算机器
图灵提出了通用计算机的理论概念
计算机学科的最高奖项以图灵命名
第一台计算机：宾夕法尼亚大学的埃尼阿克`ENIAC`，`Electronic Numerical Integrator And Computer`
计算机采用的主要元件：真空管 – 大规模集成电路
大多采用了冯·诺依曼体系：计算机采用二进制运算，包括控制器、运算器、存储器、输入设备和输出设备这五个部分。

- 控制器：管理其他部分的工作，决定执行指令的顺序，控制不同部件之间的数据交流
- 运算器：算术运算加减乘除、和逻辑运算与或非。控制器 + 运算器 = 中央处理器（`CPU`，`Central Processing Unit`）
- 存储器：存储信息的部件
- 输入设备：向计算机输入信息的设备
- 输出设备：计算机向外输出信息的设备

计算机用户大多数不需要直接与硬件打交道，归功于操作系统`Operation System`
操作系统是运行在计算机上的一套软件，负责管理计算机的软硬件资源。

操作系统提供了一套**系统调用** `System Call` ，规定了操作系统支持哪些操作。
系统调用提供的功能非常基础，有时候调用起来很麻烦，操作系统因此定义一些**库函数** `Library Routine` ，将系统调用组合成特定的功能。

1.2 所谓的编程，是做什么的

复用代码的关键是**封装** `Packaging` ，即把执行特殊功能的指令打包成一个程序块，然后给这个程序块起个容易查询的名字。
如果要重复使用这个程序块，则可以简单地通过名字调用。
操作系统就是将一些底层的硬件操作组合封装起来，供上层的应用程序调用。
封装的代价是消耗计算机资源。

封装代码的方式也有很多种。根据不同的方式，程序员写程序时要遵循特定的编程风格，如**面向过程编程**、**面向对象编程**和**函数式编程**。
每种编程风格都是一种**编程范式** `Programming Paradigm` 。
编程的需求总是可以通过多种编程范式来分别实现，区别只在于这个范式的方便程度而已。
由于不同的范式各有利弊，所以现代不少编程语言都支持多种编程范式，以便程序员在使用时取舍。
`Python` 就是一门多范式语言。
一些大学的计算机专业课程，选择了分别讲授代表性的范式语言，比如 `C` 、 `Java` 、 `Lisp` ，以便学生在未来学习其他语言时有一个好的基础。
`Python` 这样的多范式编程语言提供了一个对比学习多种编程范式的机会。
从面向过程、面向对象、函数式三种主流范式除法，在一本书的篇幅内学三遍 `Python` 。
一旦学会了编程，你会发现，软件主要比拼的就是大脑和时间，其他方面的成本都极为低廉。
正如《黑客与画家》一书中所说，程序员是和画家一样的创作者。无穷的创造机会，正是编程的一大魅力所在。

编程是人与机器互动的基本方式。人们通过编程来操纵机器。
对机器的调配和占有能力，将会取代血统和教育，成为未来阶级区分的衡量标准。这也是编程教育变得越来越重要的原因。

1.3 为什么学 `Python`

高级语言的关键是封装，让程序编写变的简单。
`Python` 正是因为在这一点上做的优秀，才成为主流编程语言之一。
`Python` 的作者是**吉多·范·罗苏姆**（ `Guido von Rossum` ）。
罗苏姆是荷兰人。
1982年，他从阿姆斯特丹大学（ `University of Amsterdam` ）获得了数学和计算机硕士学位。
罗苏姆希望有一种通用语言，既能像 `C` 语言那样调用计算机所有的功能接口，又能像 `Shell` 那样轻松地编程。最早让罗苏姆看到希望的是 `ABC` 语言。
`ABC` 语言是由荷兰的数学和计算机研究所（ `Centrum Wiskunde & Informatica` ）开发的。
这家研究所是罗苏姆上班的地方，因此罗苏姆正好能参与 `ABC` 语言的开发。
`1989` 年，为了打发圣诞节假期，罗苏姆开始写 `Python` 语言的编译 / 解释器。

`1991` 年，第一个 `Python` 编译 / 解释器诞生。它使用 `C` 语言实现的，能够调用 `C` 语言生成的动态链接库，即 `.so` 文件。
罗苏姆认为，程序员读代码的时间要远远多于写代码的时间，和 `ABC` 语言一样，用缩进代替花括号，从而保证程序更易读 `readability` 。
与 `ABC` 不一样的是，罗苏姆同样重视实用性 `practicality` 。他认为如果是常识上已经确立的东西，就没有必要过度创新。
`Python` 还特别在意可扩展性 `extensibility` 。
`Python` 的流行与计算机性能的大幅提高密不可分。 `ABC` 语言失败的一个重要原因是硬件的性能限制。

`1991` 年，林纳斯·托瓦兹在 comp.os.minix 新闻组上发布了 `Linux` 内核源代码，吸引了大批程序员加入开发工作，引领了开源运动的潮流。 `Linux` 和 `GNU` 相互合作，最终构成了一个充满活力的开源平台。

罗苏姆本人也是一位开源先锋，他维护了一个邮件列表，并把早期 `Python` 用户都放在里面。
罗苏姆充当了社区的决策者，因此他被称为仁慈的独裁者 `Benevolent Dictator For Life` 。

`Python` 的一个理念是自带电池 `Battery Included` 。它已经有了功能丰富的模块。
所谓模块，就是别人已经编写好的 `Python` 程序，能实现一定的功能。
这些模块既包括 `Python` 自带的标准库，也包括了第三方库。

1.4 最简单的 `Hello world`

运行 `Python` 的方式有两种：

- 命令行 `Command Line` 。会有 `>>>` 的提示符，输入的语句会被 `Python` 解释器 `interpreter` 转化成计算机指令。
- 写一个程序文件 `Program File` 。 `Python` 的程序文件以 `.py` 为后缀。

与命令行相比，程序文件适用于编写和保存量比较大的程序。

程序文件的另一好处是可以加入注释 `comments` 。
注释的内容不会被当做程序执行。
在 `Python` 的程序文件中，每一行从 `#` 开始的文字都是注释。

如果注释的内容较多，在一行里放不下，可以用**多行注释** `multiline comments` 。

```
"""
多行注释
"""
```

多行注释用三个连续的双引号或者单引号，两组引号之间就是多行注释的内容。

如果在 `Python 2` 使用中文，需要加入一行编码信息，说明程序文件中使用了支持中文的 `utf-8` 编码。而 `Python 3` 不需要。

```
```

> `Python 3` 默认都是 `utf-8` 编码

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

> `Hello world!` 之所以流行，是因为它被经典教程教材《 `C` 程序设计语言》用作例子。

附录A `Python` 的安装与运行

1. 官方版本安装

`Mac`
`Mac` 上预装 `Python` ，可以直接使用。
如果想要使用其他版本的 `Python` ，建议用 `Homebrew` 安装。
`Homebrew` 是 `Mac` 下的软件包管理工具，官网： http://brew.sh/

打开终端 `Terminal` ，运行：
`python`

上面输入的 `python` 通常是一个软链接，指向某个版本的 `Python` 命令。
还可以指定版本号 `python 3.5`

退出 `Python` 使用 `exit()`

运行 `Python` 文件使用 `python hello.py`
如果文件不在当前目录，需要指定完整路径名。

把 `Python` 改成可执行脚本，只要在 `hello.py` 第一行加入需要使用的解释器：
`#!/usr/bin/env python`
然后把 `hello.py` 的权限改成可执行：
`chmod 755 hello.py`
最后就可以直接在终端执行文件： `./hello.py`

`Linux`
与 `Mac` 类似，自带 `Python` ，自己安装的话所使用的软件管理器不一样。
`Ubuntu` 上安装 `Python` ：
`sudo apt-get install python`

> 实际上用 `sudo apt install python` 也行

`Windows`
需要到官网下载安装包安装。

www.python.org
安装时记得勾选 `Add python.exe to Path`

---

2. 其他 `Python` 版本

官方的 `Python` 版本主要提供了编译/解释器的功能。
其他非官方的版本还有：

- `Anaconda`
- `Enthought Python Distribution (EPD)`

`Anaconda` 是免费的，而 `EPD` 对学生和科研人员免费。

附录B `virtualenv`

`virtualenv` 可以给每个版本的 `Python` 创建一个虚拟环境

安装 `virtualenv`
`pip install virtualenv`

创建虚拟环境： `/usr/bin/python3.5` 是解释器所在位置， `venv` 是虚拟环境名称
`virtualenv -p /usr/bin/python3.5 venv`

激活虚拟环境
`source venv/bin/activate`

退出虚拟环境
`deactivate`

---

2. 先做键盘侠

2.1 计算机会算术

数值计算
加减乘除，乘方求余。

字符串的加法运算时字符串拼接。
字符串的整数乘法是重复。

逻辑运算
一个假设性的说法被称为命题。

逻辑的任务就是找出命题的真假。

计算机之所以采用二进制，是技术上的原因。
许多组成计算机的原件，都只能表达两个状态。
这样造出的系统也相对稳定。

在 `Python` 中，我们使用 `True` 和 `False` 两个关键字来表示真假，它们被称为布尔值 `Boolean`。

与运算就像是联结的两座桥，必须两座桥都通畅，才能过河。
用 `and` 表示与运算。

或运算就像并行跨过河的两座桥，任意一座通畅，就能让行人过河。
用 `or` 表示或运算。

非运算，对一个命题求反，用 `not` 表示非运算。

判断表达式
判断表达式其实就是用数学形式写出来的命题。
等于 `==`
不等于 `!=`
小于 `<`
小于或等于 `<=`
大于 `>`
大于等于 `>=`

运算优先级
按照先后顺序

- 乘方 `**`
- 乘除 `*` `/`
- 加减 `+` `-`
- 判断 `==` `>` `>=` `<` `<=`
- 逻辑 `!` `and` `or`

括号会打破优先级

2.2 计算机记性好

变量革命
计算机存储器中的每个存储单元都有一个地址，就像是门牌号。
我们可以吧数据存入特定门牌号的隔间，然后通过门牌号来提取之前存储的数据。

但是用内存地址来为存储的地址建索引，其实并不方便：

- 内存地址相当冗长，难以记忆
- 每个地址对应的存储空间大小固定，难以适应类型多变的数据
- 对某个地址进行操作前，并不知道该地址的存储空间是否已经被占用

变量和内存地址类似，也起到了索引数据的功能。
但是，不同之处在于，根据变量的类型，分配的存储空间会有大小变化。
程序员给变量起一个变量名，在程序中作为该变量空间的索引。
数据交给变量，在需要的时候通过变量的名字来提取数据。

赋值 `Assignment` 把数据交给变量保存的过程。
变量名是从内存中找到对应数据的线索。
数学上，用符号代替数值的做法称为**代数**。
变量提供的符号化表达方式，是实现代码复用的第一步。

`Python` 能自由改变变量类型的特征被称为**动态类型** `Dynamic Typing`。
静态类型 `Static Typing` 的语言中，变量有事先说明好的类型。

为了效率和实用性，计算机在内存中必须要分类型存储。
动态类型的语言把区分类型的工作交给解释器。

容器型变量：有一些变量，能像一个容器一样，收纳多个数据。
序列 `Sequence` 是有顺序的数据集合。
序列包含的一个数据被称为序列的一个**元素** `element`。
有两种：

- 元组 `Tuple`（有的翻译成**定值表**）
- 列表 `List`

序列元素的位置索引称为**下标** `Index`。

字典也是容器类型，不具备序列那样的连续有序性，适合存储结构松散的一组数据。大部分情况使用**字符串**作为字典的键值。

2.3 计算机懂选择

2.4 计算机能循环

`continue` **跳过**
`break` **终止**

`Python` 的官方文档中提供了一套代码规范，`PEP8`
`PEP` 是 `Python` 改善建议 `Python Enhancement Proposal` 的简称。
包含了 `Python` 发展历程中的关键文档。

---

3. 过程大于结果

面向过程的其他封装方法：函数和模块。

3.1 懒人炒菜机

在数学上，函数代表了集合之间的对应关系。
看待函数 `Function` 的三种方式：

- 集合的对应关系
- 数据的魔法盒子
- 语句的封装

由于函数定义中的参数是一个形式代表，并非真正数据，所以又称为形参 `Parameter`。

`return` 的作用

- 终止函数
- 制定返回值

如果没有 `return` 或者后面没有值的话，返回 `None`。
在函数调用时出现的参数称为实参 `argument`。

3.2 参数传递

基本传参：位置传参和关键字传参。
包裹 `packing` 传参：包裹位置传参和包裹关键字传参。

基本传参和包裹传参混用：
位置 – 关键字 – 包裹位置 – 包裹关键字

在函数调用时，`*` 和 `**` 代表解包裹。
包裹传参和解包裹并不是相反操作，而是两个相对独立的功能。

> 个人理解：它们出现的位置不同，一个是在函数定义处，一个是在函数调用处。
> 函数定义时，把参数收集到元组或者字典，函数调用时，把元组或者字典展开为参数。

调用函数时，传参方式也可以混合：
位置 – 关键字 – 位置解包裹 – 关键字解包裹

3.3 递归

递归 `Recursion`，在一个函数定义中，调用了这个函数自身。

递归要求程序要有一个能够到达的终止条件 `Base Case`。

递归源自数学归纳法。
数学归纳法 `Mathematical Induction` 是一种数学证明方法，常用于证明命题在自然数范围内成立。
如果我们想要证明某个命题对于自然数 `n` 成立，那么：

- 证明命题对于 `n = 1` 成立
- 假设命题对于 `n` 成立，`n` 为任意自然数，则证明在此假设下，命题对于 `n + 1` 成立。
- 命题得证

这就好像**多米诺骨牌**，我们确定 `n` 的倒下会导致 `n + 1` 的倒下，然后只要推倒第一块骨牌，就能保证任意骨牌的倒下。

函数栈
程序中的递归要用到栈 `Stack` 这一数据结构。
数据结构 是计算机存储数据的组织方式。
栈的每一个元素，称为一个帧 `frame`。
栈只支持两个操作：

- `push` 推入
- `pop` 弹出

程序运行的过程，可以看作是一个先增长栈，后消灭栈的过程。
每次函数调用，都伴随着一个帧入栈。
当函数返回时，相应的帧会出栈。
等到程序的最后，栈清空，程序就完成了。

3.4 引入那把宝剑

一个 `py` 文件就构成一个模块。
对于面向过程语言来说，模块是比函数更高一层的封装。
程序可以以文件为单位实现复用。
把常见的功能编到模块中，方便未来使用，就成为所谓的库 `library`。

查询搜索路径
```
import sys
print(sys.path)
```
`sys.path` 是一个列表，可以动态改变搜索路径。

静态修改搜索路径
在 `Linux` 系统可以更改 `home` 文件夹下的 `.bashrc`：
```
export PYTHONPATH=/home/vamei/mylib:$PYTHONPATH
```
意思是：在原有的 `PYTHONPATH` 基础上，加上 `/home/vamei/mylib`。

补充
模块搜索路径

当模块是第一次导入的时候：

- 第一级：在内置模块中寻找
- 第二级：在 `sys.path` 中寻找
  - 当前目录
  - 环境变量 `PYTHONPATH` 中指定的路径列表
  - `Python` 安装路径
  - `Python` 安装路径下的 `.pth` 文件中的路径
  - `Python` 安装路径的 `lib` 库
  - `lib` 库中的 `.pth` 文件中的路径

当模块是第二次导入的时候：

- 从已经加载的模块中寻找
  - `import sys`
  - `print(sys.modules)` 查看已经加载的模块

3.5 异常处理

程序员眼中的 `bug` ，是指程序缺陷。
这些程序缺陷会引发错误或者意想不到的后果。

> 曾经有一只蛾子飞进一台早期计算机，造成这台计算机出错。从那以后， `bug` 就被用于指代程序缺陷。这只蛾子后来被贴在日志本上，至今还在美国国家历史博物馆展出。

只有在运行时，编译器才会发现的错误被称为**运行时错误** `Runtime Error` 。
`Python` 要比静态语言更容易产生运行时错误。
还有一种错误，称为**语义错误** `Semantic Error` 。编译器认为你的程序没有问题，可以正常运行。但当检查程序时，却发现程序并非你想做的。这种错误最为隐蔽，也最难纠正。

修改程序缺陷的过程称为 `debug` 。

测试驱动开发 `TDD` ， `Test-Driven Development` 。

对于运行时可能产生的错误，我们可以提前在程序中处理，有两个目的：

- 让程序中止前进行更多的操作，比如提供更多的关于错误的信息
- 让程序在犯错后依然能运行下去，提高程序的容错性

`except` 后面没有任何参数，表示所有的异常都交给这段程序处理。
如果无法将异常交给合适的对象，那么异常将继续向上层抛出，直到被捕捉或者造成主程序报错。

---

4. 朝思暮想是对象

4.1 轻松看对象

类和对象与面向过程中的函数和模块一样，提高了程序的可复用性。
它们还加强了程序模拟真实世界的能力。
模拟，正是面向对象编程的核心。

面向对象范式可以追溯到 `Simula` 语言。
克里斯登·奈加特是挪威国防部的数学家，用电脑处理国防中的计算问题。
他发现很难用过程式的编程方式模拟真实世界的情况。
后来他遇见了计算机专家奥利-约翰·达尔。
他们共同实现了面向对象的 `Simula` 语言。

我们可以把面向对象看作是故事和指令之间的桥梁。
程序员用一种故事式的编程语言描述问题，随后编译器会把这些程序翻译成机器指令。
但在计算机发展的早期，这些额外的翻译工作会消耗太多的计算机资源，因此，面向对象的编程范式并不流行。
一些纯粹的面向对象语言，也常因为效率低下而受到诟病。

随着计算机性能的提高，效率问题不再是瓶颈。
人们转而关注程序员的产量，开始发掘面向对象语言的潜力。

比雅尼·斯特劳斯特鲁普在 `C` 语言的基础上增加面向对象的语法结构，创造出 `C++` 语言。

`Python` 也是一门面向对象语言，它比 `Java` 还要历史悠久。
`Python` 的一条哲学理念：一切皆对象。

类的概念和我们日常生活中的类差不多。
我们把相近的东西归为一类，并且给这个类起一个名字。
在计算机语言中，我们把个体称为对象。一个类别下，可以有多个个体。

通过调用类，我们创造出这个类下面的一个对象。

`Python` 定义了一系列的**特殊方法**，又被称为**魔法方法** `Magic Method`。
`__init__()` 方法会在每次创建对象的时候自动调用：

- 可以在初始化对象属性
- 可以加入其他指令

调用类时，类的后面可以跟一个参数列表，这里放入的数据将传给 `__init__()` 的参数。

`self` 的目的：在方法内部引用对象自身。
功能：

- 操作对象属性
- 在一个方法内部调用同一类的其他方法

4.2 继承者们

我们可以通过继承 `Inheritance` 减少程序中的重复信息和重复语句。
类 `object` 是 `Python` 中的一个内置类，它充当了所有类的祖先。

分类往往是人了解世界的第一步。
卡尔·林奈提出一个分类系统，通过父类和子类的隶属关系，为进一步的科学发明铺平了道路。
面向对象语言及继承机制，正式模拟人的有意识分类过程。

覆盖 `override`

4.3 那些年，错过的对象

`dir()` 用来查询一个类或者对象的所有属性
`help()` 查询函数和类的说明文档
`pass` 是 `Python` 的一个特殊关键字，用于说明在该语法结构中 什么都不做。
这个关键字保持了程序结构的完整性。

列表对象的一些方法

- `count`
- `index`
- `append`
- `sort`
- `reverse`
- `pop`
- `remove`
- `insert`
- `clear`

字符串是特殊的元组。
字符串的一些方法

- `str.count(sub)`
- `str.find(sub)` 如果找不到返回 `-1`
- `str.index(sub)` 如果找不到，会报错
- `str.rfind(sub)`
- `str.rindex(sub)`
- `str.isalnum()`
- `str.isalpha()`
- `str.isdigit()`
- `str.istitle()`
- `str.isspace()`
- `str.islower()`
- `str.isupper()`
- `str.split([sep, [max]])`
- `str.rsplit([sep, [max]])`
- `str.join(s)`

- `str.strip([sub])`
- `str.replace(sub, new_sub)`
- `str.capitalize()`
- `str.lower()`
- `str.upper()`
- `str.swapcase()`
- `str.title()`
- `str.center(width)`
- `str.ljust(width)`
- `str.rjust(width)`

4.4 意想不到的对象

补充
关于可迭代对象、迭代器、生成器的理解

> 一句话总结：迭代器属于可迭代对象，生成器是特殊的迭代器。

可迭代对象是可以用于 `for 元素 in 对象` 循环的对象。
可迭代对象至少要实现以下两种魔法方法之一：

- `__iter__` 方法：返回一个迭代器。
- `__getitem__` 方法：返回索引操作 `[]` 对应的值。

`for 元素 in 对象` 循环的工作方式

- 检查对象是否实现了 `__iter__()` 方法，如果有，执行 `iter(对象)` 操作，获取该对象的迭代器，这相当于执行了 `对象.__iter__()` 方法。
- 如果对象没有实现 `__iter__()` 方法，判断是否实现了 `__getitem__()` 方法。如果实现了，则自动创建一个迭代器，然后从 `[0]` 开始进行索引取值。

  > 这也是为什么 序列对象 都是可迭代对象的原因。因为所有的序列对象都遵循 序列协议，它们都实现了 `__getitem__()` 方法和 `__len__()` 方法。

- 如果以上两个方法都没有实现，抛出 `TypeError` 异常。
- 由上面的描述可见：两个方法都实现的情况下会优先调用对象的 `__iter__()` 方法。
- 获取到迭代器之后，不断地对迭代器执行 `next(迭代器)`，直到捕获 `StopIteration` 异常
- `next(迭代器)` 相当于执行了 `迭代器.__next__()` 方法，该方法的返回值会赋值给 `for 元素 in 对象` 语句中的 `元素` 变量。

```
class Message:
    def __init__(self):
        self.val = ['大', '王', '叫', '我', '来', '巡', '山']

    def __getitem__(self, i):
```

```
            print('我是 __getitem__')
            return self.val[i]

        # def __iter__(self):
        #     print('我是 __iter__')
        #     return iter(self.val)


    msg = Message()
    for i in msg:
        print(i)

    """
    我是 __getitem__
    大
    我是 __getitem__
    王
    我是 __getitem__
    叫
    我是 __getitem__
    我
    我是 __getitem__
    来
    我是 __getitem__
    巡
    我是 __getitem__
    山
    我是 __getitem__
    """
```

迭代器

- 实现了 `__iter__()` 方法和 `__next__()` 方法的对象就是迭代器对象。
- 所有的迭代器都是可迭代的。
- 一般要求迭代器的 `__iter__()` 方法返回它本身，即 `return self`。
- 而 `__next__()` 方法需要不断地依次返回元素，直到所有元素耗尽，此后一直抛出迭代终止的异常，即 `raise StopIteration`。

> 另外还有一种少见的写法是：
> 在 `__iter__()` 方法中返回一个实现了 `__next__()` 方法的对象。
> 相当于把迭代器拆成两部分写。

生成器

生成器包括生成器函数和生成器表达式。它们都属于迭代器。

函数对象

任何一个有 `__call__()` 特殊方法的对象都被当做是函数。

模块对象
`import` 相当于导入模块对象的属性和方法。
`模块.方法()` 和 `模块.属性` 就相当于是 `对象.方法()` 或者 `对象.属性`。
可以将功能相似的模块放在同一个文件夹中，构成一个**模块包**。
该文件夹必须包含 `__init__.py` 文件。
每个模块对象都有一个 `__name__` 属性，用于记录模块的名字。
当 `.py` 文件作为主程序运行时，它的 `__name__` 属性为 `'__main__'`。

异常对象
抛出异常时会抛出一个对象
`raise ZeroDivisionError()`

---

5. 对象带你飞

5.1 存储

对于计算机来说，数据的本质就是有序的二进制数序列。
如果以字节为单位，也就是每 `8` 位二进制数序列为单位，那么这个数据序列就称为文本。
因为 `8` 位的二进制数序列正好对应 `ASCII` 编码中的一个字符。

`content = f.read(10)` 读取 `10` 个字节
`content = f.readline()` 读取一行
`content = f.readlines()` 读取所有行，放在列表中

如果想写入一行，需要加入 `\n`(`Unix` 系统) 或者 `\r\n`(`Windows` 系统)。

写入的方法有 `f.write('...')` 和 `f.writelines(一个字符串组成的列表)`。

上下文管理器 `context manager`：用于规定某个对象的使用范围。
一旦进入或者离开该使用范围，则会有特殊操作被调用。

`print(f.closed)` 检查文件是否关闭

使用上下文管理器的语法时，`Python` 会在进入程序块之前调用文件对象的 `__enter__()` 方法，在结束程序块的时候调用文件对象的 `__exit__()` 方法。
在文件对象的 `__exit__()` 方法中，有 `self.close()` 语句。

任何定义了 `__enter__()` 方法和 `__exit__()` 方法的对象都可以用于上下文管理器。

`__enter__()` 返回一个对象。上下文管理器会使用这一对象作为 `as` 所指的变量。
`__exit__()` 有四个参数。`__exit__()` 参数中的 `exc_type`，`exc_val`，`exc_tb` 用于描述异常。
如果正常运行结束，这三个参数都是 `None`。

`pickle` 包

通过 `pickle` 包，我们可以把某个对象保存下来，再存成磁盘里的文件。

对象的存储分两步：

- 将对象在内存中的数据直接抓取出来，转化成一个有序的文本，即所谓的序列化 `Serialization`。
- 将文本存入文件

等到需要的时候，从文件中读出文本，再放入内存，就可以获得原有的对象。

`pickle.dumps()` 方法把对象转换成字节串的形式。
然后再把字节串写入到文件。

```
import pickle

class Bird:
    have_feather = True
    reproduction_method = 'egg'

summer = Bird()
# 第一步：将对象转换成字节串
pickle_string = pickle.dumps(summer)
# 第二步：存储该字节串
with open('summer.pkl', 'wb') as f:
    f.write(pickle_string)
```

其实可以通过 `dump()` 方法，一次完成两步（**序列化对象并存储**）。

```
import pickle


class Bird:
    have_feather = True
    reproduction_method = 'egg'


summer = Bird()
# 使用 dump() 方法：直接序列化对象然后存储
with open('summer.pkl', 'wb') as f:
    pickle.dump(summer, f)  # 序列化并保存对象
```

读取的时候也分两步，先读取字节串，然后 `loads()` 把字节串转换成对象。
或者直接使用 `load()` 一次性完成两步。
当我们从文件中读取对象时，程序中必须已经定义过类。

```
import pickle


class Bird:
    have_feather = True
    reproduction_method = 'egg'


with open('summer.pkl', 'rb') as f:
    summer = pickle.load(f)


print(summer.have_feather)
"""
True
"""
```

5.2 一寸光阴

`time` **包**
在硬件基础上，计算机可以提供**挂钟时间** `Wall Clock Time`。
挂钟时间是从某个固定时间起点到现在的时间间隔。
对于 `Unix` 系统来说，起点时间是 `1970` 年的 `1` 月 `1` 日的 `0` 点 `0` 分 `0` 秒。
其他的日期信息都是从挂钟时间计算得到的。
计算机还可以测量 `CPU` 实际运行的时间，也就是**处理器时间** `Processer Clock Time`，以测量计算机性能。
当 `CPU` 处于闲置状态时，处理器时间会暂停。

`time.time()` 查看挂钟时间

```
import time

# 查看挂钟时间，单位是秒
print(time.time())
"""
1545705930.5438366
"""
```

`time.clock()` 为程序计时
在 `Unix` 系统，返回的是**处理器时间**，我们获得的是 `CPU` 运行时间。
在 `Windows` 系统，返回的则是挂钟时间。

```
import time
```

```
# 给程序计时
start = time.clock()
for i in range(1000):
    print(i ** 2)
end = time.clock()
print(end - start)
```

方法 `sleep()` 可以让程序休眠。
`time.struct_time` 对象，将挂钟时间转换为年、月、日、时、分、秒，存储在对象的各个属性中。
还可以用 `time.mktime()` 把 `struct_time` 对象转换成 `time` 对象。

```
import time

# 将挂钟时间转换为 struct_time 对象

# 返回 struct_time 的 UTC 时间
st1 = time.gmtime()
print(st1)
"""
time.struct_time(tm_year=2018, tm_mon=12, tm_mday=25, tm_hour=2, tm_min=5
4, tm_sec=37, tm_wday=1, tm_yday=359, tm_isdst=0)
"""

# 返回 struct_time 的当地时间，当地时区根据系统环境决定
st2 = time.localtime()
print(st2)
"""
time.struct_time(tm_year=2018, tm_mon=12, tm_mday=25, tm_hour=10, tm_min=
54, tm_sec=37, tm_wday=1, tm_yday=359, tm_isdst=0)
"""

# 将 struct_time 对象转换为 time 对象
s = time.mktime(st1)
"""
1545679651.0
"""
```

> `tm_isdst` 是夏令时的标识，`0` 代表没有实行夏令时
> 通过 `time.gmtime()` 得到的时间，`tm_dst` 一定是 `0`，因为这是格林威治天文时间。

`datetime` 包
`datetime` 包是基于 `time` 包的一个高级包。
可以理解为由 `date` 和 `time` 两个部分组成。
`datetime` 模块下有两个类：

- `datetime.date` 类，相当于日历
- `datetime.time` 类，相当于手表

还可以直接调用 `datetime.datetime` 类，相当于把日历和手表在一起使用。

```
import datetime

t = datetime.datetime(2012, 9, 3, 21, 30)

print(t)
"""
2012-09-03 21:30:00
"""
```

对象 `t` 的属性：

- `hour` 小时
- `minute` 分
- `second` 秒
- `millisecond` 毫秒
- `microsecond` 微秒
- `year` 年
- `month` 月
- `day` 日
- `weekday` 星期几

专门代表时间间隔的类 `timedelta`
`datetime.timedelta` 的参数：

- `weeks` 星期
- `days` 天
- `hours` 小时
- `seconds` 秒
- `milliseconds` 毫秒
- `microseconds` 微秒

`datetime.datetime` 类可以互相比较。

```
import datetime

t = datetime.datetime(2012, 9, 3, 21, 30)
t_next = datetime.datetime(2012, 9, 5, 23, 30)
delta1 = datetime.timedelta(seconds=600)
delta2 = datetime.timedelta(weeks=3)

print(t + delta1)
print(t + delta2)
print(t_next - t)
```

```
"""
2012-09-03 21:40:00
2012-09-24 21:30:00
2 days, 2:00:00
"""

print(t > t_next)
"""
False
"""
```

使用 `strptime` 解析 `parse`，把字符串转换成 `datetime.datetime` 类的对象。

```
from datetime import datetime

string = 'output-1997-12-23-030000.txt'

fmt = 'output-%Y-%m-%d-%H%M%S.txt'

t = datetime.strptime(string, fmt)
print(t)
"""
1997-12-23 03:00:00
"""
```

`datetime` 对象的 `strftime` 方法，可以将 `datetime` 对象转换为特定格式的字符串。

```
from datetime import datetime

fmt = '%Y-%m-%d %H:%M'
t = datetime(2012, 9, 5, 23, 30)

print(t.strftime(fmt))
"""
2012-09-05 23:30
"""
```

> 【注】个人记法
> `strptime` 解析时间字符串
> `strftime` 格式化时间字符串

5.3 看起来像那样的东西

正则表达式 `Regular Expression`
`Python` 中用 `re` 包来处理正则表达式。

> 个人理解：
> `re.search(找什么, 到哪里去找)`

```
import re

m = re.search('[0-9]', 'abcd4ef')
print(m.group(0))

"""
4
"""
```

- `re.search(pattern, string)` 搜索整个字符串，直到发现符合的子字符串
- `re.match(pattern, string)` 从头开始检查字符串是否符合正则表达式，必须从字符串的第一个字符就开始相符。
- `str = re.sub(pattern, replacement, string)` 对搜索到的子字符串进行替换。
- `re.split(pattern, string, maxsplit=0, flags=0)` 按照正则表达式分隔字符串，返回一个列表
- `re.findall(pattern, string, flags=0)` 返回所有符合正则表达式的子字符串组成的列表

正则表达式语法：

- `.`：任意的一个字符
- `a|b`：字符 `a` 或字符 `b`
- `[afg]`：字符 `a` 或者 `f` 或者 `g` 中的一个
- `[0-4]`：`0-4` 范围内的一个字符
- `[a-f]`：`a-f` 范围内的一个字符
- `[^m]`：不是 `m` 的一个字符
- `\s`：一个空格
- `\S`：一个非空格
- `\d`：一个数字，相当于 `[0-9]`
- `\D`：一个非数字，相当于 `[^0-9]`
- `\w`：数字或字母，相当于 `[0-9a-zA-Z]`
- `\W`：非数字或非字母，相当于 `[^0-9a-zA-Z]`

表示重复：

- `*`：重复任意次
- `+`：重复 `1` 次或多次
- `?`：重复 `0` 次或 `1` 次
- `{m}`：重复 `m` 次
  `[1-3]{2}` 相当于 `[1-3][1-3]`
- `{m, n}` 重复 `m` 到 `n` 次

位置相关符号：

- `^` 字符串的起始位置
- `$` 字符串的结尾位置

进一步提取：

- 在正则表达式上给目标加上括号
  用括号 `()` 圈起来的正则表达式的一部分，称为**群** `group`。

  > 【注】多翻译成**组**。

- 使用 `group(number)` 来查询组。 `group(0)` 是整个正则表达式的搜索结果。 `group(1)` 才是第一个组。

```
import re

m = re.search('output_(\d{4})', 'output_1986.txt')
print(m.group(1))

"""
1986
"""
```

使用 `(?P<名字>)` 给子组命名，使用 `.group('名字')` 的形式访问。

```
import re

m = re.search('output_(?P<year>\d{4})', 'output_1986.txt')
print(m.group('year'))
"""
1986
"""
```

5.4 Python 有网瘾

参与通信的个体要遵守特定的协议 `Protocol`
计算机之间的通信就是在不同的计算机间传递信息。
`HTTP` 是最常见的一种网络协议，中文名：超文本传输协议，英文全称 `the Hypertext Transfer Protocol`。

`HTTP` 的工作方式类似于快餐点单：

- 请求 `request`

- 响应 `response`

请求的格式：

- 起始行有三段信息：`GET` 方法，资源路径和协议版本。
- 最常用的还有 `POST` 方法。
- 头信息：类型是 `HOST`，说明了要访问的服务器地址。

相应的格式：

- 起始行有三段信息：协议版本，状态码 `status code`，状态描述
  状态描述是对状态码的文字描述，方便人类阅读， 计算机只关心三位**状态码** `Status Code`。
- 主体包含的资源类型
  - `text/plain`：普通文本
  - `text/html`：`HTML` 文本
  - `image/jpeg`：`jpeg` 图片
  - `image/gif`：`gif` 图片
- `Content-length`：主体的长度部分，以字节为单位
- 主体部分：包含主要的文本数据

常见状态码：

- `200` 一切正常
- `302` 重定向 `Redirect`
- `404` 无法找到 `Not Found`

`http.client` 包

```
import http.client

# 主机地址
conn = http.client.HTTPConnection('www.example.com')
# 请求方法和资源路径
conn.request('GET', '/')
# 获得回复
response = conn.getresponse()
# 回复的状态码和状态描述
print(response.status, response.reason)
# 回复的主体内容
content = response.read()
print(content)  # 字节串形式的内容
```

> 【注】个人理解：建连接 – 发请求 – 得回复

5.5 写一个爬虫

```
import http.client
import re

conn = http.client.HTTPConnection('www.cnblogs.com')
conn.request('GET', '/vamei')
response = conn.getresponse()
content = response.read().decode('utf-8')
content = content.split('\r\n')
pattern = 'posted @ (\d{4}-[0-1]\d-[0-3]\d [0-2]\d:[0-6]\d) Vamei 阅读\
((\d+)\) 评论'
for line in content:
    m = re.search(pattern, line)
    if m:
        print(m.group(1), m.group(2))

"""
2018-08-18 00:47 8741
2018-07-25 20:30 2379
2018-07-19 13:43 7370
2018-07-11 02:55 5071
2018-03-04 22:28 2212
2017-12-30 22:55 3846
2017-04-29 20:41 25283
2017-04-23 17:48 5879
2017-04-20 19:17 5043
2017-04-16 15:45 7801
2017-04-10 15:03 8796
2017-04-04 17:15 2311
2017-04-01 09:24 1856
2017-03-22 22:16 7105
2017-02-22 01:01 3893
2017-02-19 14:08 5075
2017-01-12 20:52 4364
2017-01-05 22:12 2770
2017-01-03 19:35 4597
2016-12-27 23:22 19097
"""
```

---

6. 与对象的深入交往

6.1 一切皆对象

运算符：运算符实际上调用了对象的特殊方法

- `'abc' + 'xyz'` 相当于 `'abc'.__add__('xyz')`
- `1.8 * 2.0` 相当于 `(1.8).__mul__(2.0)`
- `True or False` 相当于 `True.__or__(False)`

元素引用：元素引用实际上也是调用了对象的特殊方法
使用 `dir(对象)` 可以看到对象的所有属性

- `li[3]` 相当于 `li.__getitem__(3)`
- `li[3] = 0` 相当于 `li.__setitem__(3, 0)`
- `del d['a']` 相当于 `d.__delitem__('a')`

内置函数的实现：许多内置函数也是通过调用对象的特殊方法实现的。

- `len([1, 2, 3])` 相当于 `[1, 2, 3].__len__()`
- `abs(-1)` 相当于 `(-1).__abs__()`
- `(2.3).__int__()` 相当于 `int(2.3)`

6.2 属性管理

属性覆盖
对象的属性是**分层管理的**。

> 对象 – 类 – 父类 – `object` 类

对象不需要重复存储其祖先类的属性，所以分层管理的机制可以节省存储空间。
某个属性可能在不同层被重复定义，`Python` 在向下遍历的过程中，会选取先遇到的哪一个。
这正是属性覆盖的原理所在。
子类的属性比父类的同名属性有优先权，这是属性覆盖的关键。
如果是对属性进行赋值，就不会分层查找，而是会新建一个属性。
直接修改某个祖先类的属性。
`Bird.feather = 3` 等同于修改 `Bird` 的 `__dict__` 属性，即 `Bird.__dict__['feather'] = 3`。

特性 `property`
特性是特殊的属性，可以修改某属性的时候，让依赖于该属性的其他属性也同时变化。
`属性名 = property(获取, 设置, 删除, 说明字符串)`

`__getattr__()` 方法
当我们访问对象的属性时，如果在 `__dict__` 机制找不到属性，那么就会调用对象的 `__getattr__(self, name)` 方法。
`__getattribute__()` 用于查询任意属性。
`__setattr__(self, name, value)` 和 `__delattr__(self, name)` 用于设置和删除属性。

即时生成属性的方式

- `property` 函数或者 `@property` 装饰器
- `__getattr__` 魔法方法
- `descriptor` 类

6.3 我是风儿，我是沙

动态类型 `Dynamic Typing`
对象名其实是指向对象的一个引用。
对象是存储在内存中的实体。
对象名是指向这一对象的引用 `reference`。
`id()` 查看引用指向的是哪个对象，这个函数返回对象的编号。
变量名是个随时可以变更指向的引用，那么它的类型自然可以在程序中动态变化。
还可以用 `is` 来判断两个引用是否指向同一个对象。

可变对象 `Mutable Object`
不可变对象 `Immutable Object`

函数的参数传递，本质上传递的是引用。

6.4 内存管理

在 `Python` 中，引用与对象分离。
一个对象可以有多个引用，而每个对象中都存有指向该对象的引用总数。
即**引用计数** `Reference Count`。
`sys.getrefcount()` 查看引用计数。
注意：当使用某个引用作为参数，传递给 `getrefcount()` 时，参数实际上是创建了一个临时的引用。因此 `getrefcount()` 所得到的结果，会比期望的多 `1`。

```
from sys import getrefcount

a = [1, 2, 3]
print(getrefcount(a))
"""
2
"""
```

```
b = a
print(getrefcount(b))
"""
3
"""
```

对象引用对象
数据容器对象，比如列表和字典，可以包含多个对象。
实际上，容器对象中包含的并不是元素对象本身，而是指向各个元素对象的引用。

使用 `globals()` 查看引用关系词典，该词典记录了所有的全局引用。
赋值 `a = 1` 相当于让字典中的一个键为 `'a'` 的元素引用整数对象 `1`。

`objgraph` 绘制引用关系。

```
x = [1, 2, 3]
y = [x, dict(key1=x)]
z = [y, (x, y)]

import objgraph
objgraph.show_refs([z], filename='ref_topo.png')
```

两个对象可以相互引用，从而构成所谓的**引用环** `Reference Cycle`。

```
a = []
b = [a]
a.append(b)
```

单个对象自己引用自己

```
from sys import getrefcount
a = []
a.append(a)
print(a)
print(getrefcount(a))
"""
[[...]]
3
"""
```

引用计数减少的情况

- 使用 `del` 关键字删除某个引用，会减少引用计数。
- 引用被重定向到其他对象

垃圾回收 `Garbage Collection`
当 `Python` 的某个对象的引用计数降为 `0`，即没有任何引用指向该对象时，该对象就成为了要被回收的垃圾了。

垃圾回收时，`Python` 不能进行其他的任务。频繁的垃圾回收将大大降低 `Python` 的工作效率。
当 `Python` 运行时，会记录其中分配对象 `Object Allocation` 和取消分配对象 `Object Deallocation` 的次数。
当两者的差值高于某个阈值的时候，垃圾回收才会启动。
查看该阈值：

```
import gc

print(gc.get_threshold())
"""
(700, 10, 10)
"""
```

`700` 是垃圾回收启动的阈值。
后面两个 `10` 是与分代回收相关的阈值。
可以通过 `gc.set_threshold()` 方法重新设置。
手动启动垃圾回收 `gc.collect()`

基础回收方式之外，还有**分代回收** `Generation` 的策略。
这一策略的基本假设：存活时间越久的对象，越不可能在后面的程序中变成垃圾。
对于“长寿”对象，可以减少在垃圾回收中扫描它们的频率。
`Python` 将所有的对象分为 `0`、`1`、`2` 三代，所有的新建对象都是 `0` 代对象。
当某一代对象经历过垃圾回收，依然存活，那么它就被归入下一代对象。
垃圾回收启东时，一定会扫描所有的 `0` 代对象。
如果 `0` 代经过一定次数的垃圾回收，就会启动对 `0` 代和 `1` 代的扫描清理。
当 `1` 代也经历了一定次数的垃圾回收之后，就会启动对 `0`、`1`、`2` 代的扫描，即对所有对象进行扫描。
`(700, 10, 10)` 的 `10` 的意思是：每 `10` 次 `0` 代垃圾回收，会配合 `1` 次 `1` 代的垃圾回收；每 `10` 次 `1` 代垃圾的回收，才会有 `1` 次 `2` 代的垃圾回收。

```
import gc
gc.set_threshold(700, 10, 5)
```

孤立的引用环

```
a = []
b = [a]
```

```
a.append(b)
del a
del b
```

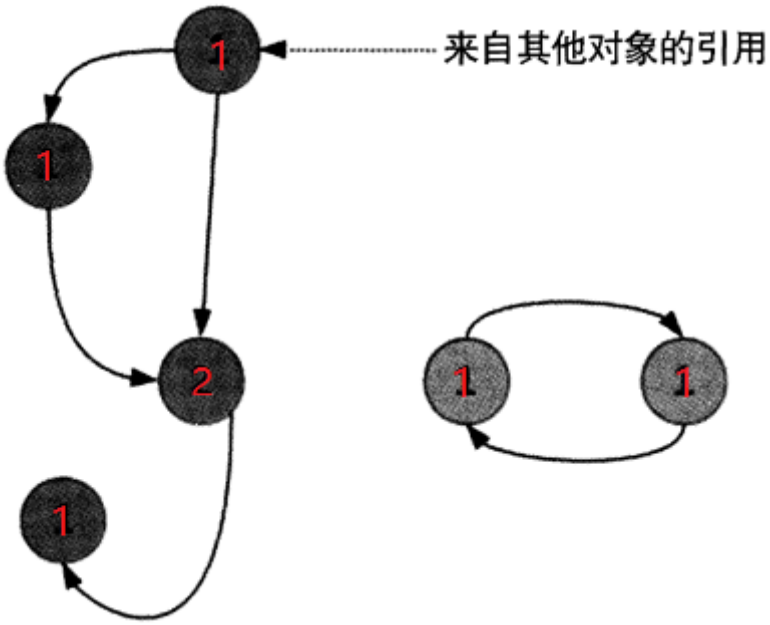
如何回收引用环？

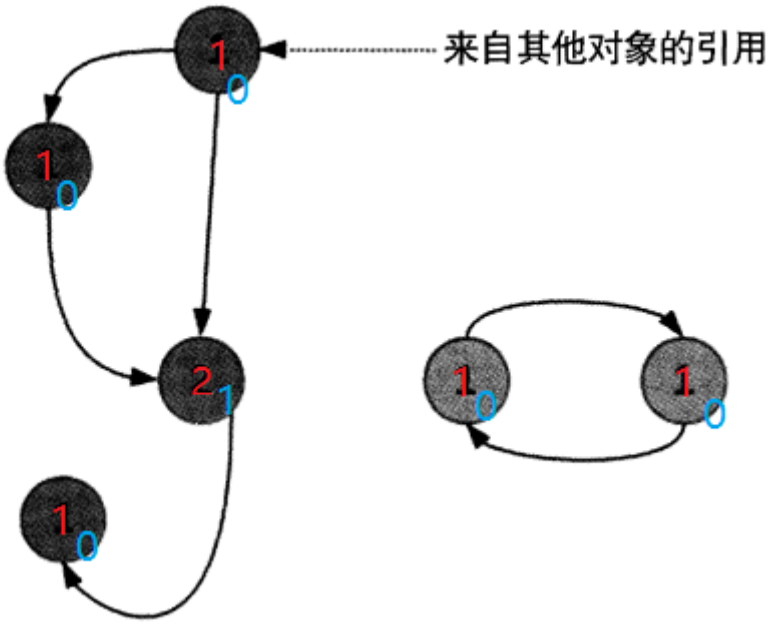
`Python` 先复制每个对象的引用计数，记为 `gc_ref`。

每个对象 `i` 的引用计数为 `gc_ref_i`。

遍历所有对象 `i`，对于每个对象 `i` 所引用的对象 `j`，将相应的 `gc_ref_j` 减 `1`。

在结束遍历后，`gc_ref` 不为 `0` 的对象，和这些对象引用的对象，以及继续更下游引用的对象，需要被保留，其他对象则被垃圾回收。





7. 函数式编程

7.1 又见函数

函数式编程：`Fuctional Programming`
面向过程利用选择和循环结构、函数、模块对指令进行封装。
面向对象实现了另一种形式的封装。
作为第三种编程范式，函数式编程的本质也在于封装。
函数式编程以函数为中心进行代码封装。
函数式编程的一个重点：函数是**第一级对象**，能像普通对象一样使用。

> 【注】`first-class object`

函数式编程强调了**函数的纯粹性** `purity`。
一个纯函数是没有副作用的 `Side Effect`。
即这个函数的运行不会影响其他函数。
纯函数像一个沙盒，把函数带来的效果控制在内部，从而不影响程序的其他部分。
为了达到纯函数的标准，函数式编程要求其变量都是不可变的类型（不能是列表和字典）。
`Python` 并非完全的函数式编程语言。

纯函数的好处：

- 没有副作用，不会影响程序其它部分
- 方便进行并行化运算

竞跑条件 `Race Condition`

> 个人理解：两个或者多个进程同时访问同一资源，由于进程执行顺序的不确定性，从而导致结果的不确定性，这就是竞跑条件。
> 参考：https://blog.csdn.net/u012562273/article/details/56486776/

函数式编程消灭了副作用，在无形中消除了竞跑条件的可能。

早期 `Python` 并没有函数式编程的相关语法。
后来加入了 `lambda` 函数，以及 `map`、`filter`、`reduce` 等高阶函数。
从而加入了函数式编程的特征。
但是 `Python` 没有严格执行语法规范，缺乏相关优化，因此不是完整的函数式编程语言。我们可以把它的函数式编程的特征作为体验的起点。

并行运算：多条指令同时执行。
串行运算：每次执行一条指令。（一般来说，一台单处理器的计算机同一时间内只能执行一条指令）

大规模并行运算通常是在有多个主机组成的**集群** `Cluster` 上进行的。
主机之间可以借助高速的网络设备通信。
我们可以在单机上通过多进程或多线程的方式，模拟多主机的并行处理。
单机的处理器按照“分时复用”的方式，把运算能力分配给多个进程。

多进程编程 `multiprocessing`
`join()` 方法用于在主程序中等待相应进程完成。

进程：一个运行中的程序。
进程有自己的内存空间，用来存储运行状态、数据和相关代码。
一个进程一般不会直接读取其他进程的内存空间。

在一个进程内部，可以有多个称为“线程”的任务。
线程之间可以共享同一个进程的内存空间。

7.2 被解放的函数

在函数式编程中，函数是第一级对象。
“第一级对象”：函数能像普通对象一样使用。

函数可以作为参数和返回值
很多语言都能把函数作为参数使用，比如 `C` 语言。
在图形化界面编程时，这样一个作为参数的函数经常起到回调 `Callback` 的作用。

```
def line_conf():
    b = 15

    def line(x):
        return 2 * x + b

    b = 5
    return line


if __name__ == '__main__':
    my_line = line_conf()
    print(my_line(5))

"""
15
"""
# 之所以返回 15 而不是 25
# 是因为迟绑定机制
# 在调用函数的时候才对用到的变量求值，而不是定义时确定该值。
```

一个函数和它的环境变量合在一起，就构成了一个闭包 `closure`。
在 `Python` 中，所谓的闭包是一个包含有环境变量取值的函数对象。
环境变量取值被赋值到函数对象的 `__closure__` 属性中。

> 【注】所以在上面那段代码中的闭包指的就是 `line` 这个函数

```
def line_conf():
    b = 15

    def line(x):
        return 2 * x + b

    b = 5
    return line


if __name__ == '__main__':
    my_line = line_conf()
    print(my_line.__closure__)
    print(my_line.__closure__[0].cell_contents)

"""
(<cell at 0x000002070C867558: int object at 0x0000000077DD6CA0>,)
5
"""
```

【个人理解】【重要】
闭包的定义
闭包是一个引用了外部变量的函数对象。
闭包的写法 或者叫做 闭包的表现形式

- 外部函数嵌套内部函数
- 内部函数引用外部变量
- 外部函数返回内部函数

`my_line` 的 `__closure__` 属性中包含了一个元组，这个元组中的每个元素都是 `cell` 类型的对象。
第一个 `cell` 包含的就是整数 `5`，也就是我们返回闭包时的环境变量 `b` 的取值。

闭包可以提高代码的可复用性。
比如要定义直线的方程。

```
def line_conf(a, b):
    def line(x):
        return a * x + b
    return line


line1 = line_conf(1, 1)
line2 = line_conf(4, 5)
line3 = line_conf(5, 10)
```

函数 `line()` 与环境变量 `a`、`b` 构成闭包。
闭包实际上 创建了一群形式相似的函数。

> 【注】环境变量这里用外部变量可能好一点。

闭包还可以减少函数的参数（减参）。
闭包的减参作用对于并行运算来说很有意义。
在并行运算的环境下，让每台电脑负责一个函数，上一台电脑的输出和下一台电脑的输入串联。由于每台电脑只能接收一个输入，所以在串联之前必须用闭包之类的办法把参数的个数降为 `1`。

> 【?】这一段不是很理解。

7.3 小女子的梳妆匣

装饰器 `decorator` 可以对一个函数、方法或者类进行加工。
从操作上入手，为函数增加额外的指令。

装饰器 `Python 2.5` 出现，最初用于函数。
`Python 2.6` 可以进一步用于类。

装饰器接收一个可调用对象作为输入参数，并返回一个新的可调用对象。

```
def decorator_demo(old_function):
    def new_function(a, b):
        print('输入了 %d 和 %d' % (a, b))
        return old_function(a, b)
    return new_function

@decorator_demo
def sqare_sum(a, b):
    return a ** 2 + b ** 2

if __name__ == '__main__':
    print(sqare_sum(3, 4))

"""
输入了 3 和 4
25
"""
```

使用 `@decorator_demo` 语句实际上执行的是：
`square_sum = decorator_demo(square_sum)`

`Python` 中的变量名和对象是分离的。
变量名实际是指向一个对象的引用。
从本质上，装饰器起到的作用就是**名称绑定** `name binding`，让同一个变量名指向一个新返回的函数对象，从而达到修改函数对象的目的。
只不过我们很少彻底地更改函数对象。
在使用装饰器时，我们往往会在新函数的内部调用旧的函数，以便保留旧函数的功能。

计时装饰器

```
from time import time

def decorator_time(old_function):
    def new_function(*args, **kwargs):
        t1 = time()
        result = old_function(*args, **kwargs)
        t2 = time()
        print('耗时', t2 - t1)
        return result
    return new_function
```

装饰器可以实现代码的可复用性。
比如为所有处理 `HTTP` 请求的函数加上登录验证的功能，就可以用 `@login_required`。

带参装饰器
上面的 `@decorator_demo`，该装饰器默认它后面的函数是唯一的参数。
装饰器的语法允许我们调用 `decorator` 时，提供其他参数，比如 `@decorator(a)`
带参装饰器 实际上是对原有装饰器的一个函数封装，并返回一个装饰器。
可以把它理解为一个含有环境参量的闭包。

```
# 带有 pre 参数的装饰器
def pre_str(pre=''):
    def decorator(old_function):
        def new_function(a, b):
            print(pre + '输入了', a, b)
            return old_function(a, b)
        return new_function
    return decorator


@pre_str('^_^')
def square_sum(a, b):
    return a ** 2 + b ** 2


if __name__ == '__main__':
    print(square_sum(3, 4))

"""
```

```
^_^输入了 3 4
25
"""
```

使用 `@pre_str('^_^')` 的时候，实际上相当于：
`square_sum = pre_str('^_^')(squaresum)`

装饰类
一个装饰器可以接收一个类，并返回一个类，从而起到加工类的效果。

```
def decorator_class(Someclass):
    class NewClass:
        def __init__(self, age):
            self.total_display = 0
            self.wrapped = Someclass(age)

        def display(self):
            self.total_display += 1
            print('总打印次数:', self.total_display)
            self.wrapped.display()

    return NewClass


@decorator_class
class Bird:
    def __init__(self, age):
        self.age = age

    def display(self):
        print('我的年龄是:', self.age)


if __name__ == '__main__':
    eagle = Bird(5)
    for i in range(3):
        eagle.display()
"""
总打印次数: 1
我的年龄是: 5
总打印次数: 2
我的年龄是: 5
总打印次数: 3
我的年龄是: 5
"""
```

装饰器 `decorator_class` 中，返回了一个新类 `NewClass`。
在类的构造器中，我们用一个属性 `self.wrapped` 记录了原来类生成的对象。

并附加了新属性 `total_display`，用于记录调用 `display()` 的次数。
同时更改了 `display` 方法。

> 【个人理解】
> 调用 `@decorator_class` 相当于
> `Bird = decorator_class(Bird)`

无论是装饰函数，还是装饰类，装饰器的核心作用都是**名称绑定**。

7.4 高阶函数

- 函数可以作为其他函数的参数和返回值（因为函数是一个第一级对象）
- 下面要讲能够接收其他函数作为参数的函数。

高阶函数 `high-order funciton` 是能够接收其他函数作为参数的函数。

装饰器的本质就是一个高阶函数。

高阶函数是函数式编程的一个重要组成部分。
本节讲最具有代表性的高阶函数：

- `map()`
- `filter()`
- `reduce()`

可以用 `lambda` 定义匿名函数，适用于简短函数的定义。
高阶函数：能处理函数的函数。

`map(函数对象, 可迭代对象)`
把可迭代对象的元素取出来作为函数对象的参数，最终返回一个**迭代器**。

> `Python 2.7` 中 `map()` 返回列表。
> 【注】这个之前我没记清楚，`Python 3` 最终返回一个**迭代器**！
> 因为可以对它使用 `next`
> 而且 `iter(map) is map`

通过 `yield` 实现 `map()` 函数

```
def mymap(func, it):
    for i in it:
        yield func(i)


data = [1, 2, 3]
res1 = map(lambda x: x + 3, data)
```

```
res2 = mymap(lambda x: x + 3, data)

for i in res1:
    print(i, end=' ')

print()

for i in res2:
    print(i, end=' ')

"""
4 5 6
4 5 6
"""
```

`map(函数对象, 可迭代对象)` 中的函数对象也可以接收多个参数。
相应的要传多个可迭代对象进去。

```
def square_sum(x, y):
    return x ** 2 + y ** 2


data1 = [1, 2, 3]
data2 = [4, 5, 6]

print(list(map(square_sum, data1, data2)))
"""
[17, 29, 45]
"""
```

一定程度上，`map()` 函数能替代循环的功能。
从另一个角度来看，`map()` 看起来像是对多个目标“各个击破”。
在并行计算中，通过 `Map` 过程，一个大问题可以拆分成很多小问题，从而交给不同的主机处理。
比如图像处理，可以把大图拆成小图给多台主机处理。

`filter(函数对象, 可迭代对象)` 函数
将 `函数对象` 作用于 `可迭代对象`，如果返回结果为 `True`，则将该元素放入迭代器。

```
def larger100(x):
    return x > 100

for i in filter(larger100, [10, 56, 101, 500]):
    print(i)

"""
101
500
```

```
"""
```

> 【注意】注意看 `for i in filter(...):` 的写法！

`functools.reduce(函数对象, 可迭代对象)`
`函数对象` 必须接收两个参数，它把函数对象的累进作用于可迭代对象。
`reduce()` 函数通过某种形式的二元运算，把多个元素收集起来，形成一个单一的结果。
谷歌用于并行运算的软件架构，称为 `MapReduce`。
`Map` 运算的结果分布在多个主机上，然后用 `Reduce` 运算把结果收集起来。

7.5 自上而下

> 【个人理解】
> 越来越方便的写法
> 迭代器、生成器函数、生成器表达式 `Generator Expression`

```
def gen():
    for i in range(4):
        yield i

# 等价于
gen = (x for x in range(4))
```

列表解析 `List Comprehension`

```
x = [1, 3, 5]
y = [9, 12, 13]
l = [x ** 2 for (x, y) in zip(x, y) if y > 10]

print(l)
"""
[9, 25]
"""
```

字典解析

```
>>> d = {k:v for k, v in enumerate('Vamei') if v not in 'Vi'}
>>> d
{1: 'a', 2: 'm', 3: 'e'}
```

懒惰求值
迭代器的元素是实时计算出来的。
在使用该元素之前，元素并不会占据内存空间。
迭代器的工作方式正式函数式编程中的**懒惰求值** `Lazy Evaluation`。

如果说计算最终都不可避免，那么懒惰求值和**即时求值**的运算量并没有什么差别。
但如果不需要穷尽所有的数据元素，那么懒惰求值将节省不少时间。
比如循环到某个条件之后就 `break`，这种情况就不会穷尽所有的数据元素。

懒惰求值还能 节约内存空间，不用存储运算过程的中间结果。
可以在迭代器层面操作，最后一次性完成计算。

`itertools` 里面有很多有用的迭代器。
`count(5, 2)` 从 `5` 开始的整数迭代器，步长为 `2`，无穷迭代器。
`count(7)` 从 `7` 开始，步长为 `1`。
`cycle('abc')` 循环重复序列元素，无穷迭代器。
`repeat(1.2)` 无穷迭代器，重复返回 `1.2`
`repeat(10, 5)` 重复 `10`，一共重复 `5` 次。
`chain([1, 2, 3], [4, 5, 7])` 串联可迭代对象，生成更大的迭代器。
`product('abc', [1, 2])` 多个迭代器的笛卡尔积，相当于**嵌套循环**
笛卡尔积：可以得出集合元素所有可能的组合方式。

```
from itertools import *

for m, n in product('abc', [1, 2]):
    print(m, n)

"""
a 1
a 2
b 1
b 2
c 1
c 2
"""
```

`permutations('abc', 2)` 从 `'abc'` 挑选两个元素，将所有结果排序，返回一个迭代器。
上述结果区分顺序即 `'ab'` 和 `'ba'` 是不同的。

```
for i in permutations('abc', 2):
    print(i)

"""
('a', 'b')
('a', 'c')
('b', 'a')
('b', 'c')
('c', 'a')
```

```
('c', 'b')
"""
```

`combinations('abc', 2)` 同上面的类似，但是不区分顺序。

```
for i in combinations('abc', 2):
    print(i)

"""
('a', 'b')
('a', 'c')
('b', 'c')
"""
```

`combination_with_replacement('abc', 2)`，和上面类似，允许重复

```
for i in combinations_with_replacement('abc', 2):
    print(i)

"""
('a', 'a')
('a', 'b')
('a', 'c')
('b', 'b')
('b', 'c')
('c', 'c')
"""
```

`starmap(pow, [(1, 1), (2, 2), (3, 3)])` 把可迭代对象里面的元素依次取出并解包，传给函数对象调用，把结果作为迭代器返回。

```
for i in starmap(pow, [(1, 1), (2, 2), (3, 3)]):
    print(i)
"""
1
4
27
"""
```

`takewhile(lambda x: x < 5, [1, 3, 6, 7, 1])` 当函数返回 `True` 的时候，收集元素到迭代器。一旦返回 `False`，停止。
`dropwhile(lambda x: x < 5, [1, 3, 6, 7, 1])` 当函数返回 `False` 的时候，跳过元素。一旦返回 `True`，开始收集剩下的所有元素到迭代器。

```
for i in takewhile(lambda x: x < 5, [1, 3, 6, 7, 1]):
```

```
    print(i)
"""
1
3
"""

for i in dropwhile(lambda x: x < 5, [1, 3, 6, 7, 1]):
    print(i)
"""
6
7
1
"""
```

`groupby()` 函数功能类似 `UNIX` 中的 `uniq` 命令。
`groupby(iterable, key=None)` 返回一个迭代器。
这个迭代器返回键和值。如果指定了 `key` 函数，键就是 `key` 函数作用于可迭代对象的每个元素得到的返回值。如果没有指定 `key` 函数，可迭代对象中的元素本身会被用作键。
具有相同键的值，将组成一个迭代器。
返回的形式是：`键，具有相同键的值组成的迭代器`

```
from itertools import groupby

def height_class(h):
    if h > 180:
        return 'tall'
    elif h < 160:
        return 'short'
    else:
        return 'middle'

friends = [191, 158, 159, 165, 170, 177, 181, 182, 190]
friends.sort(key=height_class)

for m, n in groupby(friends, key=height_class):
    print(m)
    print(n)

"""
middle
<itertools._grouper object at 0x000001C9D827ACF8>
short
<itertools._grouper object at 0x000001C9D8285320>
tall
<itertools._grouper object at 0x000001C9D827ACF8>
"""
```

这里在使用 `groupby()` 分组之前，先使用了 `sort` 函数，让具有相同键的元素在位置上靠拢。

完成于 `2018.12.26`