Python 知识手册

发布 *1.1*

chengge

2022 年 06 月 25 日

Contents:

CHAPTER 1

第一章：python 基础

1.1 Python 简介

1.1.1 python 是什么？

Python 是一种解释型、面向对象、动态数据类型的高级程序设计语言，它由 Guido van Rossum 在 1989 年发明。Python 官方宣布，2020 年 1 月 1 日，停止 Python 2 的更新。Python 2.7 被确定为最后一个 Python 2.x 版本，目前建议 Python 用户使用 3.X 版本。

1.1.2 Python 做什么？

Python 可以做的事情很多。

- Python 可以与软件一起使用来创建工作流。
- Python 可以连接到数据库系统。它还可以读取和修改文件。
- Python 可用于处理大数据并执行复杂的数学运算。
- Python 可用于快速原型设计，也可用于生产就绪的软件开发。

1.1.3 为何选择 Python ?

Python 适用于不同的平台（Windows、Mac、Linux、Raspberry Pi 等）。Python 有一种类似于英语的简单语法。Python 的语法允许开发人员用比其他编程语言更少的代码行编写程序。Python 在解释器系统上运行，这意味着代码可以在编写后立即执行。这也意味着原型设计可以非常快。Python 可以以程序方式、面向对象的方式或功能方式来处理。

1.1.4 Python 语法与其他编程语言比较

Python 是为可读性设计的，与英语有一些相似之处，并受到数学的影响。Python 使用新行来完成命令，而不像通常使用分号或括号的其他编程语言。Python 依赖缩进，使用空格来定义范围；例如循环、函数和类的范围。其他编程语言通常使用花括号来实现此目的。

1.1.5 Python 特点

- 易于学习：Python 有相对较少的关键字，结构简单，和一个明确定义的语法，学习起来更加简单。
- 易于阅读：Python 代码定义的更清晰。
- 易于维护：Python 的成功在于它的源代码是相当容易维护的。
- 一个广泛的标准库：Python 的最大的优势之一是丰富的库，跨平台的，在 UNIX，Windows 和 Macintosh 兼容很好。
- 互动模式：互动模式的支持，您可以从终端输入执行代码并获得结果的语言，互动的测试和调试代码片断。
- 可移植：基于其开放源代码的特性，Python 已经被移植（也就是使其工作）到许多平台。
- 可扩展：如果你需要一段运行很快的关键代码，或者是想要编写一些不愿开放的算法，你可以使用 C 或 C++ 完成那部分程序，然后从你的 Python 程序中调用。
- 数据库：Python 提供所有主要的商业数据库的接口。
- GUI 编程：Python 支持 GUI 可以创建和移植到许多系统调用。
- 可嵌入: 你可以将 Python 嵌入到 C/C++ 程序，让你的程序的用户获得”脚本化”的能力。

1.2 Python 环境搭建

1.2.1 Python 下载

Python 最新源码，二进制文档，新闻资讯等可以在 Python 的官网查看到：

Python 官网：https://www.python.org/

你可以在以下链接中下载 Python 的文档，你可以下载 HTML、PDF 和 PostScript 等格式的文档。

Python 文档下载地址：https://www.python.org/doc/

1.2.2 Python 安装

Python 已经被移植在许多平台上（经过改动使它能够工作在不同平台上）。

您需要下载适用于您使用平台的二进制代码，然后安装 Python。

如果您平台的二进制代码是不可用的，你需要使用 C 编译器手动编译源代码。

编译的源代码，功能上有更多的选择性，为 python 安装提供了更多的灵活性。

以下是各个平台安装包的下载地址：

1.2.3 集成开发环境: PyCharm

PyCharm 是由 JetBrains 打造的一款 Python IDE，支持 macOS、Windows、Linux 系统。

PyCharm 功能: 调试、语法高亮、Project 管理、代码跳转、智能提示、自动完成、单元测试、版本控制……

PyCharm 下载地址: https://www.jetbrains.com/pycharm/download/

PyCharm 安装地址：http://www.runoob.com/w3cnote/pycharm-windows-install.html

1.2.4 pip 使用方法

说到 pip ，大家都不会陌生。但我相信不少人，只是熟悉几个常用的用法，而对于其他几个低频且实用的用法，却知之甚少，这两天，我把这些用法整理了一下，应该是网络上比较全的介绍

查询软件包

查询当前环境安装的所有软件包

```
$ pip list
```

查询 pypi 上含有某名字的包

```
$ pip search pkg
```

查询当前环境中可升级的包

```
$ pip list --outdated
```

查询一个包的详细内容

```
$ pip show pkg
```

下载软件包

在不安装软件包的情况下下载软件包到本地

```
$ pip download --destination-directory /local/wheels -r requirements.txt
```

下载完，总归是要安装的，可以指定这个目录中安装软件包，而不从 pypi 上安装。

```
$ pip install --no-index --find-links=/local/wheels -r requirements.txt
```

当然你也从你下载的包中，自己构建生成 wheel 文件

```
$ pip install wheel
$ pip wheel --wheel-dir=/local/wheels -r requirements.txt
```

安装软件包

使用 `pip install <pkg>` 可以很方便地从 pypi 上搜索下载并安装 python 包。

如下所示

```
$ pip install requests
```

这是安装包的基本格式，我们也可以为其添加更多参数来实现不同的效果。

1 只从本地安装，而不从 pypi 安装

```
# 前提你得保证你已经下载 pkg 包到 /local/wheels 目录下
$ pip install --no-index --find-links=/local/wheels pkg
```

2 限定版本进行软件包安装

以下三种，对单个 python 包的版本进行了约束

```
# 所安装的包的版本为 2.1.2
$ pip install pkg==2.1.2

# 所安装的包必须大于等于 2.1.2
$ pip install pkg>=2.1.2
```

(下页继续)

(续上页)

```
# 所安装的包必须小于等于 2.1.2
$ pip install pkg<=2.1.2
```

以下命令用于管理/控制整个 python 环境的包版本

```
# 导出依赖包列表
pip freeze >requirements.txt

# 从依赖包列表中安装
pip install -r requirements.txt

# 确保当前环境软件包的版本(并不确保安装)
pip install -c constraints.txt
```

3 限制不使用二进制包安装

由于默认情况下，wheel 包的平台是运行 pip download 命令的平台，所以可能出现平台不适配的情况。

比如在 MacOS 系统下得到的 pymongo-2.8-cp27-none-macosx_10_10_intel.whl 就不能在 linux_x86_64 安装。

使用下面这条命令下载的是 tar.gz 的包，可以直接使用 pip install 安装。

比 wheel 包，这种包在安装时会进行编译，所以花费的时间会长一些。

```
# 下载非二进制的包
$ pip download --no-binary=:all: pkg

#  安装非二进制的包
$ pip install pkg --no-binary
```

4 指定代理服务器安装

当你身处在一个内网环境中时，无法直接连接公网。这时候你使用 `pip install` 安装包，就会失败。

面对这种情况，可以有两种方法：

1. 下载离线包拷贝到内网机器中安装
2. 使用代理服务器转发请求

第一种方法，虽说可行，但有相当多的弊端

- 步骤繁杂，耗时耗力
- 无法处理包的依赖问题

这里重点来介绍，第二种方法：

```
$ pip install --proxy [user:passwd@]http_server_ip:port pkg
```

每次安装包就发输入长长的参数，未免有些麻烦，为此你可以将其写入配置文件中：$HOME/.config/pip/pip.conf

对于这个路径，说明几点

- 不同的操作系统，路径各不相同

```
# Linux/Unix:
/etc/pip.conf
~/.pip/pip.conf
~/.config/pip/pip.conf

# Mac OSX:
~/Library/Application Support/pip/pip.conf
~/.pip/pip.conf
/Library/Application Support/pip/pip.conf

# Windows:
%APPDATA%\pip\pip.ini
%HOME%\pip\pip.ini
C:\Documents and Settings\All Users\Application Data\PyPA\pip\pip.conf (Windows XP)
C:\ProgramData\PyPA\pip\pip.conf (Windows 7及以后)
```

- 若在你的机子上没有此文件，则自行创建即可

如何配置，这边给个样例：

```
[global]
index-url = http://mirrors.aliyun.com/pypi/simple/

# 替换出自己的代理地址，格式为[user:passwd@]proxy.server:port
proxy=http://xxx.xxx.xxx.xxx:8080

[install]
# 信任阿里云的镜像源，否则会有警告
trusted-host=mirrors.aliyun.com
```

5 安装用户私有软件包

很多人可能还不清楚，python 的安装包是可以用户隔离的。

如果你拥有管理员权限，你可以将包安装在全局环境中。在全局环境中的这个包可被该机器上的所有拥有管理员权限的用户使用。

如果一台机器上的使用者不只一样，自私地将在全局环境中安装或者升级某个包，是不负责任且危险的做法。

面对这种情况，我们就想能否安装单独为我所用的包呢?

庆幸的是，还真有。

我能想到的有两种方法:

1. 使用虚拟环境

2. 将包安装在用户的环境中

虚拟环境，之前写过几篇文章，这里不再展开讲。

今天的重点是第二种方法，教你如何安装用户私有的包?

命令也很简单，只要加上 `--user` 参数，pip 就会将其安装在当前用户的 `~/.local/lib/python3.x/site-packages` 下，而其他用户的 python 则不会受影响。

```
pip install --user pkg
```

来举个例子

```
# 在全局环境中未安装 requests
[root@localhost ~]# pip list | grep requests
[root@localhost ~]# su - wangbm
[root@localhost ~]#

# 由于用户环境继承自全局环境，这里也未安装
[wangbm@localhost ~]# pip list | grep requests
[wangbm@localhost ~]# pip install --user requests
[wangbm@localhost ~]# pip list | grep requests
requests (2.22.0)
[wangbm@localhost ~]#

# 从 Location 属性可发现 requests 只安装在当前用户环境中
[wangbm@ws_compute01 ~]$ pip show requests
---
Metadata-Version: 2.1
Name: requests
Version: 2.22.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
Installer: pip
License: Apache 2.0
Location: /home/wangbm/.local/lib/python2.7/site-packages
[wangbm@localhost ~]$ exit
logout
```

(下页继续)

(续上页)

```
# 退出 wangbm 用户，在 root 用户环境中发现 requests 未安装
[root@localhost ~]$ pip list | grep requests
[root@localhost ~]$
```

当你身处个人用户环境中，python 导包时会先检索当前用户环境中是否已安装这个包，已安装则优先使用，未安装则使用全局环境中的包。

验证如下：

```
>>> import sys
>>> from pprint import pprint
>>> pprint(sys.path)
['',
 '/usr/lib64/python27.zip',
 '/usr/lib64/python2.7',
 '/usr/lib64/python2.7/plat-linux2',
 '/usr/lib64/python2.7/lib-tk',
 '/usr/lib64/python2.7/lib-old',
 '/usr/lib64/python2.7/lib-dynload',
 '/home/wangbm/.local/lib/python2.7/site-packages',
 '/usr/lib64/python2.7/site-packages',
 '/usr/lib64/python2.7/site-packages/gtk-2.0',
 '/usr/lib/python2.7/site-packages',
 '/usr/lib/python2.7/site-packages/pip-18.1-py2.7.egg',
 '/usr/lib/python2.7/site-packages/lockfile-0.12.2-py2.7.egg']
>>>
```

6 延长超时时间

若网络情况不是很好，在安装某些包时经常会因为 ReadTimeout 而失败。

对于这种情况，一般重试几次就好了。

但是这样难免有些麻烦，有没有更好的解决方法呢?

有的，可以通过延长超时时间。

```
$ pip install --default-timeout=100 <packages>
```

4. 卸载软件包

就一条命令，不再赘述

```
$ pip uninstall pkg
```

升级软件包

想要对现有的 python 进行升级，其本质上也是先从 pypi 上下载最新版本的包，再对其进行安装。所以升级也是使用 `pip install`，只不过要加一个参数 `--upgrade`。

```
$ pip install --upgrade pkg
```

在升级的时候，其实还有一个不怎么用到的选项 `--upgrade-strategy`，它是用来指定升级策略。

它的可选项只有两个：

- `eager`： 升级全部依赖包
- `only-if-need`：只有当旧版本不能适配新的父依赖包时，才会升级。

在 pip 10.0 版本之后，这个选项的默认值是 `only-if-need`，因此如下两种写法是一互致的。

```
pip install --upgrade pkg1
pip install --upgrade pkg1 --upgrade-strategy only-if-need
```

配置文件

由于在使用 pip 安装一些包时，默认会使用 pip 的官方源，所以经常会报网络超时失败。

常用的解决办法是，在安装包时，使用 `-i` 参数指定一个国内的镜像源。但是每次指定就很麻烦呀，还要打超长的一串字母。

这时候，其实可以将这个源写进 pip 的配置文件里。以后安装的时候，就默认从你配置的这个源里安装了。

那怎么配置呢？文件文件在哪？

使用 `win+r` 输入 `%APPDATA%` 进入用户资料文件夹，查看有没有一个 pip 的文件夹，若没有则创建之。

然后进入这个文件夹，新建一个 `pip.ini` 的文件，内容如下

```
[global]
time-out=60
index-url=https://pypi.tuna.tsinghua.edu.cn/simple/
[install]
trusted-host=tsinghua.edu.cn
```

以上几乎包含了 pip 的所有使用场景，也许有不少用法你还没有用过，不过没关系，你只要收藏本文，等到要用的时候再来查阅即可。

1.3 Python 基础语法

默认情况下，Python 3 源码文件以 UTF-8 编码，所有字符串都是 unicode 字符串。当然你也可以为源码文件指定不同的编码：

1.3.1 标识符

- 第一个字符必须是字母表中字母或下划线 _ 。
- 标识符的其他的部分由字母、数字和下划线组成。
- 标识符对大小写敏感。在 Python 3 中，可以用中文作为变量名，非 ASCII 标识符也是允许的了。

1.3.2 python 保留字

保留字即关键字，我们不能把它们用作任何标识符名称。Python 的标准库提供了一个 keyword 模块，可以输出当前版本的所有关键字：

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def',
↪'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import',
↪ 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try',
↪'while', 'with', 'yield']
```

1.3.3 注释

Python 中单行注释以 # 开头，实例如下：

```
#!/usr/bin/python3

# 第一个注释
print ("Hello, Python!") # 第二个注释
```

多行注释可以用多个 # 号，还有 ‘’’ 和 “”“：

```
#!/usr/bin/python3

# 第一个注释
# 第二个注释

'''
第三注释
```

(下页继续)

(续上页)

```
第四注释
'''

"""
第五注释
第六注释
"""
print ("Hello, Python!")
```

1.3.4 数字 (Number) 类型

python 中数字有四种类型：整数、布尔型、浮点数和复数。

- int (整数), 如 1, 只有一种整数类型 int，表示为长整型，没有 python2 中的 Long。
- bool (布尔), 如 True。
- float (浮点数), 如 1.23、3E-2
- complex (复数), 如 1 + 2j、1.1 + 2.2j

1.3.5 字符串 (String)

- Python 中单引号‘和双引号“使用完全相同。
- 使用三引号 (’’’ 或 “””) 可以指定一个多行字符串。
- 转义符 \。
- 反斜杠可以用来转义，使用 r 可以让反斜杠不发生转义。如 r” this is a line with \n” 则 \n 会显示，并不是换行。
- 按字面意义级联字符串，如 “this ““is ““string” 会被自动转换为 this is string。
- 字符串可以用 + 运算符连接在一起，用 * 运算符重复。
- Python 中的字符串有两种索引方式，从左往右以 0 开始，从右往左以 -1 开始。
- Python 中的字符串不能改变。
- Python 没有单独的字符类型，一个字符就是长度为 1 的字符串。
- 字符串的截取的语法格式如下：变量 [头下标: 尾下标: 步长]

```
word = '字符串'
sentence = "这是一个句子。"
paragraph = """这是一个段落，
可以由多行组成"""
```

实例

```
#!/usr/bin/python3
str='123456789'
print(str)                 # 输出字符串
print(str[0:-1])           # 输出第一个到倒数第二个的所有字符
print(str[0])              # 输出字符串第一个字符
print(str[2:5])            # 输出从第三个开始到第五个的字符
print(str[2:])             # 输出从第三个开始后的所有字符
print(str[1:5:2])          # 输出从第二个开始到第五个且每隔一个的字符（步长为2）
print(str * 2)             # 输出字符串两次
print(str + '你好')         # 连接字符串
print('------------------------------')
print('hello\nrunoob')      # 使用反斜杠(\)+n转义特殊字符
print(r'hello\nrunoob')     # 在字符串前面添加一个 r，表示原始字符串，不会发生转义
```

1.3.6 import 与 from…import

在 python 用 import 或者 from…import 来导入相应的模块。

- 将整个模块 (somemodule) 导入，格式为：import somemodule
- 从某个模块中导入某个函数, 格式为：from somemodule import somefunction
- 从某个模块中导入多个函数, 格式为：from somemodule import firstfunc, secondfunc, thirdfunc
- 将某个模块中的全部函数导入，格式为：from somemodule import *

1.3.7 命令行参数

很多程序可以执行一些操作来查看一些基本信息，Python 可以使用-h 参数查看各参数帮助信息：

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit
```

1.4 Python3 基本数据类型

Python 中的变量不需要声明。每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。

在 Python 中，变量就是变量，它没有类型，我们所说的”类型”是变量所指的内存中对象的类型。等号（=）用来给变量赋值。等号（=）运算符左边是一个变量名，等号（=）运算符右边是存储在变量中的值。例如：

```
#!/usr/bin/python3

counter = 100          # 整型变量
miles   = 1000.0       # 浮点型变量
name    = "runoob"     # 字符串

print (counter)
print (miles)
print (name)
```

执行以上程序会输出如下结果：

```
100
1000.0
runoob
```

1.4.1 多个变量赋值

Python 允许你同时为多个变量赋值。例如：

```
a = b = c = 1
```

您也可以为多个对象指定多个变量。例如：

```
a, b, c = 1, 2, "runoob"
```

1.4.2 标准数据类型

Python3 中有六个标准的数据类型：

- Number（数字）
- String（字符串）
- List（列表）
- Tuple（元组）
- Set（集合）

- Dictionary（字典）

Python3 的六个标准数据类型中：不可变数据（3 个）：Number（数字）、String（字符串）、Tuple（元组）；可变数据（3 个）：List（列表）、Dictionary（字典）、Set（集合）。

Number（数字）

Python3 支持 int、float、bool、complex（复数）。

在 Python 3 里，只有一种整数类型 int，表示为长整型，没有 python2 中的 Long。

像大多数语言一样，数值类型的赋值和计算都是很直观的。

内置的 type() 函数可以用来查询变量所指的对象类型。

```
>>> a, b, c, d = 20, 5.5, True, 4+3j
>>> print(type(a), type(b), type(c), type(d))
<class 'int'> <class 'float'> <class 'bool'> <class 'complex'>
```

String（字符串）

Python 中的字符串用单引号‘或双引号“括起来，同时使用反斜杠 \ 转义特殊字符。

字符串的截取的语法格式如下：

变量 [头下标: 尾下标] 索引值以 0 为开始值，-1 为从末尾的开始位置。

1.5 List（列表）

List（列表）是 Python 中使用最频繁的数据类型。

列表可以完成大多数集合类的数据结构实现。列表中元素的类型可以不相同，它支持数字，字符串甚至可以包含列表（所谓嵌套）。

列表是写在方括号 [] 之间、用逗号分隔开的元素列表。

和字符串一样，列表同样可以被索引和截取，列表被截取后返回一个包含所需元素的新列表。

实例

```
#!/usr/bin/python3

list = [ 'abcd', 786 , 2.23, 'runoob', 70.2 ]
tinylist = [123, 'runoob']

print (list)            # 输出完整列表
print (list[0])         # 输出列表第一个元素
```

(下页继续)

(续上页)

```
print (list[1:3])        # 从第二个开始输出到第三个元素
print (list[2:])         # 输出从第三个元素开始的所有元素
print (tinylist * 2)     # 输出两次列表
print (list + tinylist) # 连接列表
```

以上实例输出结果：

```
['abcd', 786, 2.23, 'runoob', 70.2]
abcd
[786, 2.23]
[2.23, 'runoob', 70.2]
[123, 'runoob', 123, 'runoob']
['abcd', 786, 2.23, 'runoob', 70.2, 123, 'runoob']
```

1.6 Tuple（元组）

元组（tuple）与列表类似，不同之处在于元组的元素不能修改。元组写在小括号 () 里，元素之间用逗号隔开。

元组中的元素类型也可以不相同：

```
#!/usr/bin/python3

tuple = ( 'abcd', 786 , 2.23, 'runoob', 70.2  )
tinytuple = (123, 'runoob')

print (tuple)             # 输出完整元组
print (tuple[0])          # 输出元组的第一个元素
print (tuple[1:3])        # 输出从第二个元素开始到第三个元素
print (tuple[2:])         # 输出从第三个元素开始的所有元素
print (tinytuple * 2)     # 输出两次元组
print (tuple + tinytuple) # 连接元组
```

以上实例输出结果：

```
('abcd', 786, 2.23, 'runoob', 70.2)
abcd
(786, 2.23)
(2.23, 'runoob', 70.2)
(123, 'runoob', 123, 'runoob')
('abcd', 786, 2.23, 'runoob', 70.2, 123, 'runoob')
```

1.7 Set（集合）

集合（set）是由一个或数个形态各异的大小整体组成的，构成集合的事物或对象称作元素或是成员。

基本功能是进行成员关系测试和删除重复元素。

可以使用大括号 { } 或者 set() 函数创建集合，注意：创建一个空集合必须用 set() 而不是 { }，因为 { } 是用来创建一个空字典。

创建格式：

```
#!/usr/bin/python3
sites = {'Google', 'Taobao', 'Runoob', 'Facebook', 'Zhihu', 'Baidu'}
print(sites)   # 输出集合，重复的元素被自动去掉
# 成员测试
if 'Runoob' in sites :
    print('Runoob 在集合中')
else :
    print('Runoob 不在集合中')
# set可以进行集合运算
a = set('abracadabra')
b = set('alacazam')
print(a)
print(a - b)     # a 和 b 的差集
print(a | b)     # a 和 b 的并集
print(a & b)     # a 和 b 的交集
print(a ^ b)     # a 和 b 中不同时存在的元素
```

以上实例输出结果：

```
{'Zhihu', 'Baidu', 'Taobao', 'Runoob', 'Google', 'Facebook'}
Runoob 在集合中
{'b', 'c', 'a', 'r', 'd'}
{'r', 'b', 'd'}
{'b', 'c', 'a', 'z', 'm', 'r', 'l', 'd'}
{'c', 'a'}
{'z', 'b', 'm', 'r', 'l', 'd'}
```

1.8 Dictionary（字典）

字典（dictionary）是 Python 中另一个非常有用的内置数据类型。

列表是有序的对象集合，字典是无序的对象集合。两者之间的区别在于：字典当中的元素是通过键来存取的，而不是通过偏移存取。

字典是一种映射类型，字典用 { } 标识，它是一个无序的键 (key) : 值 (value) 的集合。

键 (key) 必须使用不可变类型。

在同一个字典中，键 (key) 必须是唯一的。

```
#!/usr/bin/python3

dict = {}
dict['one'] = "1 - 菜鸟教程"
dict[2]     = "2 - 菜鸟工具"
tinydict = {'name': 'runoob','code':1, 'site': 'www.runoob.com'}
print (dict['one'])       # 输出键为 'one' 的值
print (dict[2])           # 输出键为 2 的值
print (tinydict)          # 输出完整的字典
print (tinydict.keys())   # 输出所有键
print (tinydict.values()) # 输出所有值
```

以上实例输出结果：

```
1 - 菜鸟教程
2 - 菜鸟工具
{'name': 'runoob', 'code': 1, 'site': 'www.runoob.com'}
dict_keys(['name', 'code', 'site'])
dict_values(['runoob', 1, 'www.runoob.com'])
```

1.9 Python 推导式

Python 推导式是一种独特的数据处理方式，可以从一个数据序列构建另一个新的数据序列的结构体。

Python 支持各种数据结构的推导式：

- 列表 (list) 推导式
- 字典 (dict) 推导式
- 集合 (set) 推导式
- 元组 (tuple) 推导式

1.9.1 列表推导式

列表推导式格式为：

> [表达式 for 变量 in 列表 if 条件] [out_exp_res for out_exp in input_list if condition]

- out_exp_res：列表生成元素表达式，可以是有返回值的函数。
- for out_exp in input_list：迭代 input_list 将 out_exp 传入到 out_exp_res 表达式中。
- if condition：条件语句，可以过滤列表中不符合条件的值，也可以不使用。过滤掉长度小于或等于 3 的字符串列表，并将剩下的转换成大写字母：

```
>>> names = ['Bob','Tom','alice','Jerry','Wendy','Smith']
>>> new_names = [name.upper()for name in names if len(name)>3]
>>> print(new_names)
['ALICE', 'JERRY', 'WENDY', 'SMITH']
```

1.9.2 字典推导式

字典推导基本格式：

> { key_expr: value_expr for value in collection if condition }

使用字符串及其长度创建字典：

```
listdemo = ['Google','Runoob', 'Taobao']
# 将列表中各字符串值为键，各字符串的长度为值，组成键值对
>>> newdict = {key:len(key) for key in listdemo}
>>> newdict
{'Google': 6, 'Runoob': 6, 'Taobao': 6}
```

1.9.3 集合推导式

集合推导式基本格式：

> { expression for item in Sequence if conditional }

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'d', 'r'}
>>> type(a)
<class 'set'>
```

1.9.4 元组推导式

元组推导式可以利用 range 区间、元组、列表、字典和集合等数据类型，快速生成一个满足指定需求的元组。

元组推导式基本格式：

(expression for item in Sequence if conditional)

```
>>> a = (x for x in range(1,10))
>>> a
<generator object <genexpr> at 0x7faf6ee20a50>  # 返回的是生成器对象

>>> tuple(a)        # 使用 tuple() 函数，可以直接将生成器对象转换成元组
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

1.10 Python3 解释器

Linux/Unix 的系统上，一般默认的 python 版本为 2.x，我们可以将 python3.x 安装在 /usr/local/python3 目录中。

安装完成后，我们可以将路径 /usr/local/python3/bin 添加到您的 Linux/Unix 操作系统的环境变量中，这样您就可以通过 shell 终端输入下面的命令来启动 Python3 。

```
$ PATH=$PATH:/usr/local/python3/bin/python3     # 设置环境变量
$ python3 --version
Python 3.4.0
```

在 Window 系统下你可以通过以下命令来设置 Python 的环境变量，假设你的 Python 安装在 C:\Python34 下:

```
set path=%path%;C:\python34
```

1.10.1 交互式编程

我们可以在命令提示符中输入”Python”命令来启动 Python 解释器：

```
$ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在 python 提示符中输入以下语句，然后按回车键查看运行效果：

```
print ("Hello, Python!");
```

1.10.2 脚本式编程

将如下代码拷贝至 hello.py 文件中：

```
print ("Hello, Python!");
```

通过以下命令执行该脚本：

```
python3 hello.py
```

输出结果为：

```
Hello, Python!
```

1.11 Python 注释

确保对模块, 函数, 方法和行内注释使用正确的风格。

Python 中的注释有单行注释和多行注释。

Python 中单行注释以 # 开头，例如：

```
# 这是一个注释
print("Hello, World!")
```

多行注释用三个单引号 ‘’’ 或者三个双引号 “”” 将注释括起来，例如:

1、单引号（’’’）

```
#!/usr/bin/python3
'''
这是多行注释，用三个单引号
这是多行注释，用三个单引号
这是多行注释，用三个单引号
'''
print("Hello, World!")
```

2、双引号（”””）

```
#!/usr/bin/python3
"""
这是多行注释，用三个双引号
这是多行注释，用三个双引号
这是多行注释，用三个双引号
```

(下页继续)

(续上页)

```
"""
print("Hello, World!")
```

1.12 Python 运算符

什么是运算符？举个简单的例子:

4 + 5 = 9 例子中，4 和 5 被称为操作数，+ 称为运算符。

Python 语言支持以下类型的运算符:

- 算术运算符
- 比较（关系）运算符
- 赋值运算符
- 逻辑运算符
- 位运算符
- 成员运算符
- 身份运算符
- 运算符优先级

1.12.1 Python 算术运算符

```
#!/usr/bin/python3

a = 21
b = 10
c = 0

c = a + b
print ("1 - c 的值为: ", c)

c = a - b
print ("2 - c 的值为: ", c)

c = a * b
print ("3 - c 的值为: ", c)

c = a / b
```

(下页继续)

(续上页)

```
print ("4 - c 的值为: ", c)

c = a % b
print ("5 - c 的值为: ", c)

# 修改变量 a 、b 、c
a = 2
b = 3
c = a**b
print ("6 - c 的值为: ", c)

a = 10
b = 5
c = a//b
print ("7 - c 的值为: ", c)
```

以上实例输出结果:

```
1 - c 的值为:  31
2 - c 的值为:  11
3 - c 的值为:  210
4 - c 的值为:  2.1
5 - c 的值为:  1
6 - c 的值为:  8
7 - c 的值为:  2
```

1.12.2 Python 比较运算符

```
#!/usr/bin/python3

a = 21
b = 10
c = 0

if ( a == b ):
   print ("1 - a 等于 b")
else:
   print ("1 - a 不等于 b")

if ( a != b ):
   print ("2 - a 不等于 b")
else:
```

(下页继续)

(续上页)

```
   print ("2 - a 等于 b")

if ( a < b ):
   print ("3 - a 小于 b")
else:
   print ("3 - a 大于等于 b")

if ( a > b ):
   print ("4 - a 大于 b")
else:
   print ("4 - a 小于等于 b")

# 修改变量 a 和 b 的值
a = 5
b = 20
if ( a <= b ):
   print ("5 - a 小于等于 b")
else:
   print ("5 - a 大于  b")

if ( b >= a ):
   print ("6 - b 大于等于 a")
else:
   print ("6 - b 小于 a")
```

以上实例输出结果：

```
1 - a 不等于 b
2 - a 不等于 b
3 - a 大于等于 b
4 - a 大于 b
5 - a 小于等于 b
6 - b 大于等于 a
```

1.12.3 Python 位运算符

```
#!/usr/bin/python3

a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0
```

(下页继续)

(续上页)

```
c = a & b         # 12 = 0000 1100
print ("1 - c 的值为：", c)

c = a | b         # 61 = 0011 1101
print ("2 - c 的值为：", c)

c = a ^ b         # 49 = 0011 0001
print ("3 - c 的值为：", c)

c = ~a            # -61 = 1100 0011
print ("4 - c 的值为：", c)

c = a << 2        # 240 = 1111 0000
print ("5 - c 的值为：", c)

c = a >> 2        # 15 = 0000 1111
print ("6 - c 的值为：", c)
```

以上实例输出结果：

```
1 - c 的值为：  12
2 - c 的值为：  61
3 - c 的值为：  49
4 - c 的值为：  -61
5 - c 的值为：  240
6 - c 的值为：  15
```

1.12.4 Python 逻辑运算符

Python 语言支持逻辑运算符，以下假设变量 a 为 10, b 为 20:

```
#!/usr/bin/python3

a = 10
b = 20

if ( a and b ):
   print ("1 - 变量 a 和 b 都为 true")
else:
   print ("1 - 变量 a 和 b 有一个不为 true")

if ( a or b ):
   print ("2 - 变量 a 和 b 都为 true，或其中一个变量为 true")
```

(下页继续)

(续上页)

```
else:
   print ("2 - 变量 a 和 b 都不为 true")

# 修改变量 a 的值
a = 0
if ( a and b ):
   print ("3 - 变量 a 和 b 都为 true")
else:
   print ("3 - 变量 a 和 b 有一个不为 true")

if ( a or b ):
   print ("4 - 变量 a 和 b 都为 true，或其中一个变量为 true")
else:
   print ("4 - 变量 a 和 b 都不为 true")

if not( a and b ):
   print ("5 - 变量 a 和 b 都为 false，或其中一个变量为 false")
else:
   print ("5 - 变量 a 和 b 都为 true")
```

以上实例输出结果：

```
1 - 变量 a 和 b 都为 true
2 - 变量 a 和 b 都为 true，或其中一个变量为 true
3 - 变量 a 和 b 有一个不为 true
4 - 变量 a 和 b 都为 true，或其中一个变量为 true
5 - 变量 a 和 b 都为 false，或其中一个变量为 false
```

1.12.5 Python 成员运算符

除了以上的一些运算符之外，Python 还支持成员运算符，测试实例中包含了一系列的成员，包括字符串，列表或元组。

```
#!/usr/bin/python3

a = 10
b = 20
list = [1, 2, 3, 4, 5 ]

if ( a in list ):
   print ("1 - 变量 a 在给定的列表中 list 中")
else:
   print ("1 - 变量 a 不在给定的列表中 list 中")
```

(下页继续)

(续上页)

```
if ( b not in list ):
   print ("2 - 变量 b 不在给定的列表中 list 中")
else:
   print ("2 - 变量 b 在给定的列表中 list 中")

# 修改变量 a 的值
a = 2
if ( a in list ):
   print ("3 - 变量 a 在给定的列表中 list 中")
else:
   print ("3 - 变量 a 不在给定的列表中 list 中")
```

以上实例输出结果：

```
1 - 变量 a 不在给定的列表中 list 中
2 - 变量 b 不在给定的列表中 list 中
3 - 变量 a 在给定的列表中 list 中
```

1.12.6 Python 身份运算符

```
#!/usr/bin/python3

a = 20
b = 20

if ( a is b ):
   print ("1 - a 和 b 有相同的标识")
else:
   print ("1 - a 和 b 没有相同的标识")

if ( id(a) == id(b) ):
   print ("2 - a 和 b 有相同的标识")
else:
   print ("2 - a 和 b 没有相同的标识")

# 修改变量 b 的值
b = 30
if ( a is b ):
   print ("3 - a 和 b 有相同的标识")
else:
   print ("3 - a 和 b 没有相同的标识")
```

(下页继续)

(续上页)

```
if ( a is not b ):
   print ("4 - a 和 b 没有相同的标识")
else:
   print ("4 - a 和 b 有相同的标识")
```

以上实例输出结果：

```
1 - a 和 b 有相同的标识
2 - a 和 b 有相同的标识
3 - a 和 b 没有相同的标识
4 - a 和 b 没有相同的标识
```

1.13 Python 条件控制

Python 条件语句是通过一条或多条语句的执行结果（True 或者 False）来决定执行的代码块。

1.13.1 if 语句

Python 中 if 语句的一般形式如下所示：

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

- 如果“condition_1”为 True 将执行“statement_block_1”块语句
- 如果“condition_1”为 False，将判断“condition_2”
- 如果”condition_2”为 True 将执行“statement_block_2”块语句
- 如果“condition_2”为 False，将执行”statement_block_3”块语句

Python 中用 elif 代替了 else if，所以 if 语句的关键字为：if –elif –else。

以下是一个简单的 if 实例：

```
#!/usr/bin/python3

var1 = 100
if var1:
```

(下页继续)

(续上页)

```
    print ("1 - if 表达式条件为 true")
    print (var1)

var2 = 0
if var2:
    print ("2 - if 表达式条件为 true")
    print (var2)
print ("Good bye!")
```

执行以上代码，输出结果为:

```
1 - if 表达式条件为 true
100
Good bye!
```

1.13.2 if 嵌套

在嵌套 if 语句中，可以把 if…elif…else 结构放在另外一个 if…elif…else 结构中。

```
if 表达式1:
    语句
    if 表达式2:
        语句
    elif 表达式3:
        语句
    else:
        语句
elif 表达式4:
    语句
else:
    语句
```

```
# !/usr/bin/python3

num=int(input("输入一个数字： "))
if num%2==0:
    if num%3==0:
        print ("你输入的数字可以整除 2 和 3")
    else:
        print ("你输入的数字可以整除 2，但不能整除 3")
else:
    if num%3==0:
```

(下页继续)

(续上页)

```
        print ("你输入的数字可以整除 3，但不能整除 2")
    else:
        print  ("你输入的数字不能整除 2 和 3")
```

1.14 Python3 循环语句

本章节将为大家介绍 Python 循环语句的使用。

Python 中的循环语句有 for 和 while。

1.14.1 while 循环

Python 中 while 语句的一般形式：

```
while 判断条件(condition)：
    执行语句(statements)……
```

以下实例使用了 while 来计算 1 到 100 的总和：

```
#!/usr/bin/env python3

n = 100

sum = 0
counter = 1
while counter <= n:
    sum = sum + counter
    counter += 1

print("1 到 %d 之和为：%d" % (n,sum))
```

1.14.2 while 循环使用 else 语句

如果 while 后面的条件语句为 false 时，则执行 else 的语句块。

语法格式如下：

```
while <expr>:
    <statement(s)>
else:
    <additional_statement(s)>
```

```
#!/usr/bin/python3

count = 0
while count < 5:
   print (count, " 小于 5")
   count = count + 1
else:
   print (count, " 大于或等于 5")
```

执行以上脚本，输出结果如下：

```
0  小于 5
1  小于 5
2  小于 5
3  小于 5
4  小于 5
5  大于或等于 5
```

1.14.3 for 语句

Python for 循环可以遍历任何可迭代对象，如一个列表或者一个字符串。

for 循环的一般格式如下：

```
for <variable> in <sequence>:
    <statements>
else:
    <statements>
```

Python for 循环实例：

```
>>>languages = ["C", "C++", "Perl", "Python"]
>>> for x in languages:
...     print (x)
...
C
C++
Perl
Python
>>>
```

以下 for 实例中使用了 break 语句，break 语句用于跳出当前循环体：

```
#!/usr/bin/python3

sites = ["Baidu", "Google","Runoob","Taobao"]
for site in sites:
    if site == "Runoob":
        print("菜鸟教程!")
        break
    print("循环数据 " + site)
else:
    print("没有循环数据!")
print("完成循环!")
```

执行脚本后，在循环到“Runoob”时会跳出循环体：

```
循环数据 Baidu
循环数据 Google
菜鸟教程!
完成循环!
```

1.14.4 break 和 continue 语句

break 语句可以跳出 for 和 while 的循环体。如果你从 for 或 while 循环中终止，任何对应的循环 else 块将不执行。

continue 语句被用来告诉 Python 跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

实例:while 中使用 break：

```
n = 5
while n > 0:
    n -= 1
    if n == 2:
        break
    print(n)
print('循环结束。')
```

输出结果为：

```
4
3
循环结束。
```

实例:while 中使用 continue：

```
n = 5
while n > 0:
    n -= 1
    if n == 2:
        continue
    print(n)
print('循环结束。')
```

输出结果为:

```
4
3
1
0
循环结束。
```

1.14.5 pass 语句

Python pass 是空语句，是为了保持程序结构的完整性。

pass 不做任何事情，一般用做占位语句，如下实例

```
>>>while True:
...     pass  # 等待键盘中断 (Ctrl+C)
```

以下实例在字母为 o 时执行 pass 语句块:

```
#!/usr/bin/python3

for letter in 'Runoob':
   if letter == 'o':
      pass
      print ('执行 pass 块')
   print ('当前字母 :', letter)
```

执行以上脚本输出结果为:

```
当前字母 : R
当前字母 : u
当前字母 : n
执行 pass 块
当前字母 : o
执行 pass 块
当前字母 : o
```

(下页继续)

(续上页)

```
当前字母 : b
Good bye!
```

1.15 Python 函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

函数能提高应用的模块性，和代码的重复利用率。你已经知道 Python 提供了许多内建函数，比如 print()。但你也可以自己创建函数，这被叫做用户自定义函数。

1.15.1 定义一个函数

你可以定义一个由自己想要功能的函数，以下是简单的规则：

- 函数代码块以 def 关键词开头，后接函数标识符名称和圆括号 ()。
- 任何传入参数和自变量必须放在圆括号中间，圆括号之间可以用于定义参数。
- 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。
- 函数内容以冒号: 起始，并且缩进。
- return [表达式] 结束函数，选择性地返回一个值给调用方，不带表达式的 return 相当于返回 None。

1.15.2 语法

Python 定义函数使用 def 关键字，一般格式如下：

```
def 函数名（参数列表）:
    函数体
```

默认情况下，参数值和参数名称是按函数声明中定义的顺序匹配起来的。

让我们使用函数来输出”Hello World!”：

```
#!/usr/bin/python3

def hello() :
    print("Hello World!")

hello()
```

更复杂点的应用，函数中带上参数变量:

```
#!/usr/bin/python3
def max(a, b):
    if a > b:
        return a
    else:
        return b
a = 4
b = 5
print(max(a, b))
```

以上实例输出结果：

```
5
```

1.15.3 函数调用

定义一个函数：给了函数一个名称，指定了函数里包含的参数，和代码块结构。

这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从 Python 命令提示符执行。

如下实例调用了 printme() 函数：

```
#!/usr/bin/python3

# 定义函数
def printme( str ):
   # 打印任何传入的字符串
   print (str)
   return

# 调用函数
printme("我要调用用户自定义函数!")
printme("再次调用同一函数")
```

以上实例输出结果：

```
我要调用用户自定义函数!
再次调用同一函数
```

1.15.4 默认参数

调用函数时，如果没有传递参数，则会使用默认参数。以下实例中如果没有传入 age 参数，则使用默认值：

```
#!/usr/bin/python3

#可写函数说明
def printinfo( name, age = 35 ):
   "打印任何传入的字符串"
   print ("名字: ", name)
   print ("年龄: ", age)
   return

#调用printinfo函数
printinfo( age=50, name="runoob" )
print ("------------------------")
printinfo( name="runoob" )
```

以上实例输出结果：

```
名字:  runoob
年龄:  50
------------------------
名字:  runoob
年龄:  35
```

1.15.5 不定长参数

你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述 2 种参数不同，声明时不会命名。基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

加了星号 * 的参数会以元组 (tuple) 的形式导入，存放所有未命名的变量参数。

```
#!/usr/bin/python3

# 可写函数说明
def printinfo( arg1, *vartuple ):
   "打印任何传入的参数"
   print ("输出: ")
```

(下页继续)

(续上页)

```
   print (arg1)
   print (vartuple)

# 调用printinfo 函数
printinfo( 70, 60, 50 )
```

以上实例输出结果：

```
输出：
70
(60, 50)
```

1.15.6 匿名函数

Python 使用 lambda 来创建匿名函数。

所谓匿名，意即不再使用 def 语句这样标准的形式定义一个函数。

- lambda 只是一个表达式，函数体比 def 简单很多。
- lambda 的主体是一个表达式，而不是一个代码块。仅仅能在 lambda 表达式中封装有限的逻辑进去。
- lambda 函数拥有自己的命名空间，且不能访问自己参数列表之外或全局命名空间里的参数。
- 虽然 lambda 函数看起来只能写一行，却不等同于 C 或 C++ 的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

语法

lambda 函数的语法只包含一个语句，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

设置参数 a 加上 10:

```
x = lambda a : a + 10
print(x(5))
```

以上实例输出结果：

```
15
```

1.15.7 return 语句

return [表达式] 语句用于退出函数，选择性地向调用方返回一个表达式。不带参数值的 return 语句返回 None。之前的例子都没有示范如何返回数值，以下实例演示了 return 语句的用法：

```
#!/usr/bin/python3

# 可写函数说明
def sum( arg1, arg2 ):
   # 返回2个参数的和."
   total = arg1 + arg2
   print ("函数内 : ", total)
   return total

# 调用sum函数
total = sum( 10, 20 )
print ("函数外 : ", total)
```

以上实例输出结果：

```
函数内 :  30
函数外 :  30
```

1.16 Python 数据结构

本章节我们主要结合前面所学的知识点来介绍 Python 数据结构。

1.16.1 将列表当做堆栈使用

列表方法使得列表可以很方便的作为一个堆栈来使用，堆栈作为特定的数据结构，最先进入的元素最后一个被释放（后进先出）。用 append() 方法可以把一个元素添加到堆栈顶。用不指定索引的 pop() 方法可以把一个元素从堆栈顶释放出来。例如：

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
```

(下页继续)

(续上页)

```
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

1.16.2 将列表当作队列使用

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

1.16.3 列表推导式

列表推导式提供了从序列创建列表的简单途径。通常应用程序将一些操作应用于某个序列的每个元素，用其获得的结果作为生成新列表的元素，或者根据确定的判定条件创建子序列。

每个列表推导式都在 for 之后跟一个表达式，然后有零到多个 for 或 if 子句。返回结果是一个根据表达从其后的 for 和 if 上下文环境中生成出来的列表。如果希望表达式推导出一个元组，就必须使用括号。

这里我们将列表中每个数值乘三，获得一个新的列表：

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
```

1.16.4 del 语句

使用 del 语句可以从一个列表中根据索引来删除一个元素，而不是值来删除元素。这与使用 pop() 返回一个值不同。可以用 del 语句从列表中删除一个切割，或清空整个列表（我们以前介绍的方法是给该切割赋一个空列表）。例如：

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

1.16.5 集合

集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。

可以用大括号 ({}) 创建集合。注意：如果要创建一个空集合，你必须用 set() 而不是 {}；后者创建一个空的字典，下一节我们会介绍这个数据结构。

以下是一个简单的演示：

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # 删除重复的
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # 检测成员
True
>>> 'crabgrass' in basket
False

>>> # 以下演示了两个集合的操作
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # a 中唯一的字母
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # 在 a 中的字母，但不在 b 中
{'r', 'd', 'b'}
>>> a | b                              # 在 a 或 b 中的字母
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

(下页继续)

(续上页)

```
>>> a & b                           # 在 a 和 b 中都有的字母
{'a', 'c'}
>>> a ^ b                           # 在 a 或 b 中的字母，但不同时在 a 和 b 中
{'r', 'd', 'b', 'm', 'z', 'l'}
```

1.16.6 字典

另一个非常有用的 Python 内建数据类型是字典。

序列是以连续的整数为索引，与此不同的是，字典以关键字为索引，关键字可以是任意不可变类型，通常用字符串或数值。

理解字典的最佳方式是把它看做无序的键 => 值对集合。在同一个字典之内，关键字必须是互不相同。

一对大括号创建一个空的字典：{}。

这是一个字典运用的简单例子：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

1.17 Python 输入和输出

在前面几个章节中，我们其实已经接触了 Python 的输入输出的功能。本章节我们将具体介绍 Python 的输入输出。

1.17.1 输出格式美化

Python 两种输出值的方式: 表达式语句和 print() 函数。

第三种方式是使用文件对象的 write() 方法，标准输出文件可以用 sys.stdout 引用。

如果你希望输出的形式更加多样，可以使用 str.format() 函数来格式化输出值。

如果你希望将输出的值转成字符串，可以使用 repr() 或 str() 函数来实现。

str()：函数返回一个用户易读的表达形式。repr()：产生一个解释器易读的表达形式。

```
>>> s = 'Hello, Runoob'
>>> str(s)
'Hello, Runoob'
>>> repr(s)
"'Hello, Runoob'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'x 的值为： ' + repr(x) + ',  y 的值为： ' + repr(y) + '...'
>>> print(s)
x 的值为： 32.5,  y 的值为： 40000...
>>> #  repr() 函数可以转义字符串中的特殊字符
... hello = 'hello, runoob\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, runoob\n'
>>> # repr() 的参数可以是 Python 的任何对象
... repr((x, y, ('Google', 'Runoob')))
"(32.5, 40000, ('Google', 'Runoob'))"
```

1.17.2 旧式字符串格式化

% 操作符也可以实现字符串格式化。它将左边的参数作为类似 sprintf() 式的格式化字符串, 而将右边的代入, 然后返回格式化后的字符串. 例如:

```
>>> print('常量 PI 的值近似为：%5.3f。' % math.pi)
常量 PI 的值近似为：3.142。
```

1.17.3 读取键盘输入

Python 提供了 input() 内置函数从标准输入读入一行文本，默认的标准输入是键盘。

```
#!/usr/bin/python3

str = input("请输入：");
print ("你输入的内容是: ", str)
```

这会产生如下的对应着输入的结果：

```
请输入：菜鸟教程
你输入的内容是:  菜鸟教程
```

1.17.4 读和写文件

open() 将会返回一个 file 对象，基本语法格式如下:

open(filename, mode)

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "w")

f.write( "Python 是一个非常好的语言。\n是的，的确非常好!!\n" )

# 关闭打开的文件
f.close()
```

1.17.5 pickle 模块

python 的 pickle 模块实现了基本的数据序列和反序列化。

通过 pickle 模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去，永久存储。

通过 pickle 模块的反序列化操作，我们能够从文件中创建上一次程序保存的对象。

基本接口：

pickle.dump(obj, file, [,protocol]) 有了 pickle 这个对象, 就能对 file 以读取的形式打开:

```
x = pickle.load(file)
```

注解：从 file 中读取一个字符串，并将它重构为原来的 python 对象。

file: 类文件对象，有 read() 和 readline() 接口。

```
#!/usr/bin/python3
import pickle

# 使用pickle模块将数据对象保存到文件
data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

1.18 Python 错误和异常

作为 Python 初学者，在刚学习 Python 编程时，经常会看到一些报错信息，在前面我们没有提及，这章节我们会专门介绍。

Python 有两种错误很容易辨认：语法错误和异常。

Python assert（断言）用于判断一个表达式，在表达式条件为 false 的时候触发异常。

1.18.1 语法错误

Python 的语法错误或者称之为解析错，是初学者经常碰到的，如下实例

```
>>> while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

这个例子中，函数 print() 被检查到有错误，是它前面缺少了一个冒号: 。

语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的箭头。

1.18.2 异常

即便 Python 程序的语法是正确的，在运行它的时候，也有可能发生错误。运行期检测到的错误被称为异常。

大多数的异常都不会被程序处理，都以错误信息的形式展现在这里:

```
>>> 10 * (1/0)             # 0 不能作为除数，触发异常
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4 + spam*3              # spam 未定义，触发异常
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2               # int 不能与 str 相加，触发异常
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

1.18.3 异常处理

try/except 异常捕捉可以使用 try/except 语句。

```
while True:
    try:
        x = int(input("请输入一个数字: "))
        break
    except ValueError:
        print("您输入的不是数字，请再次尝试输入！")
```

try/except…else try/except 语句还有一个可选的 else 子句，如果使用这个子句，那么必须放在所有的 except 子句之后。

else 子句将在 try 子句没有发生任何异常的时候执行。

```
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()
```

1.18.4 try-finally 语句

try-finally 语句无论是否发生异常都将执行最后的代码。

```
try:
    runoob()
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print('这句话，无论异常是否发生都会执行。')
```

1.18.5 用户自定义异常

你可以通过创建一个新的异常类来拥有自己的异常。异常类继承自 Exception 类，可以直接继承，或者间接继承，例如:

```
>>> class MyError(Exception):
        def __init__(self, value):
            self.value = value
        def __str__(self):
            return repr(self.value)

>>> try:
        raise MyError(2*2)
    except MyError as e:
        print('My exception occurred, value:', e.value)

My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

1.19 Python 面向对象

python 从设计之初就已经是一门面向对象的语言，正因为如此，在 Python 中创建一个类和对象是很容易的。本章节我们将详细介绍 Python 的面向对象编程。

如果你以前没有接触过面向对象的编程语言，那你可能需要先了解一些面向对象语言的一些基本特征，在头脑里头形成一个基本的面向对象的概念，这样有助于你更容易的学习 Python 的面向对象编程。

接下来我们先来简单的了解下面向对象的一些基本特征。

面向对象技术简介

- 类 (Class): 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
- 方法：类中定义的函数。
- 类变量：类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
- 数据成员：类变量或者实例变量用于处理类及其实例对象的相关的数据。
- 方法重写：如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重写。

- 局部变量：定义在方法中的变量，只作用于当前实例的类。
- 实例变量：在类的声明中，属性是用变量来表示的，这种变量就称为实例变量，实例变量就是一个用 self 修饰的变量。
- 继承：即一个派生类（derived class）继承基类（base class）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个 Dog 类型的对象派生自 Animal 类，这是模拟”是一个（is-a）”关系（例图，Dog 是一个 Animal）。
- 实例化：创建一个类的实例，类的具体对象。
- 对象：通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。和其它编程语言相比，Python 在尽可能不增加新的语法和语义的情况下加入了类机制。

Python 中的类提供了面向对象编程的所有基本功能：类的继承机制允许多个基类，派生类可以覆盖基类中的任何方法，方法中可以调用基类中的同名方法。

对象可以包含任意数量和类型的数据。

1.19.1 类定义

语法格式如下：

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

类实例化后，可以使用其属性，实际上，创建一个类之后，可以通过类名访问其属性。

1.19.2 类对象

类对象支持两种操作：属性引用和实例化。

属性引用使用和 Python 中所有的属性引用一样的标准语法：obj.name。

类对象创建后，类命名空间中所有的命名都是有效属性名。所以如果类定义是这样:

```
#!/usr/bin/python3

class MyClass:
    """一个简单的类实例"""
    i = 12345
    def f(self):
        return 'hello world'
```

(下页继续)

(续上页)

```
# 实例化类
x = MyClass()

# 访问类的属性和方法
print("MyClass 类的属性 i 为：", x.i)
print("MyClass 类的方法 f 输出为：", x.f())
```

以上创建了一个新的类实例并将该对象赋给局部变量 x，x 为空的对象。

执行以上程序输出结果为:

```
MyClass 类的属性 i 为：  12345
MyClass 类的方法 f 输出为：  hello world
```

1.19.3 self 代表类的实例，而非类

类的方法与普通的函数只有一个特别的区别——它们必须有一个额外的第一个参数名称, 按照惯例它的名称是 self。

```
class Test:
    def prt(self):
        print(self)
        print(self.__class__)

t = Test()
t.prt()
```

以上实例执行结果为:

```
<__main__.Test instance at 0x100771878>
__main__.Test
```

从执行结果可以很明显的看出，self 代表的是类的实例，代表当前对象的地址，而 self.class 则指向类。

self 不是 python 关键字，我们把他换成 runoob 也是可以正常执行的:

```
class Test:
    def prt(runoob):
        print(runoob)
        print(runoob.__class__)

t = Test()
t.prt()
```

以上实例执行结果为:

```
<__main__.Test instance at 0x100771878>
__main__.Test
```

1.19.4 类的方法

在类的内部，使用 def 关键字来定义一个方法，与一般函数定义不同，类方法必须包含参数 self, 且为第一个参数，self 代表的是类的实例。

```
实例(Python 3.0+)
#!/usr/bin/python3

#类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性,私有属性在类外部无法直接进行访问
    __weight = 0
    #定义构造方法
    def __init__(self,n,a,w):
        self.name = n
        self.age = a
        self.__weight = w
    def speak(self):
        print("%s 说: 我 %d 岁。" %(self.name,self.age))

# 实例化类
p = people('runoob',10,30)
p.speak()
```

执行以上程序输出结果为:

```
runoob 说: 我 10 岁。
```

1.19.5 继承

Python 同样支持类的继承，如果一种语言不支持继承，类就没有什么意义。派生类的定义如下所示:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

子类（派生类 DerivedClassName）会继承父类（基类 BaseClassName）的属性和方法。

BaseClassName（实例中的基类名）必须与派生类定义在一个作用域内。除了类，还可以用表达式，基类定义在另一个模块中时这一点非常有用:

class DerivedClassName(modname.BaseClassName):

```
#!/usr/bin/python3

#类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性,私有属性在类外部无法直接进行访问
    __weight = 0
    #定义构造方法
    def __init__(self,n,a,w):
        self.name = n
        self.age = a
        self.__weight = w
    def speak(self):
        print("%s 说: 我 %d 岁。" %(self.name,self.age))

#单继承示例
class student(people):
    grade = ''
    def __init__(self,n,a,w,g):
        #调用父类的构函
        people.__init__(self,n,a,w)
        self.grade = g
    #覆写父类的方法
    def speak(self):
        print("%s 说: 我 %d 岁了, 我在读 %d 年级"%(self.name,self.age,self.grade))
```

(下页继续)

(续上页)

```
s = student('ken',10,60,3)
s.speak()
```

执行以上程序输出结果为:

```
ken 说: 我 10 岁了，我在读 3 年级
```

1.20 Python 正则表达式

正则表达式是一个特殊的字符序列，它能帮助你方便的检查一个字符串是否与某种模式匹配。

Python 自 1.5 版本起增加了 re 模块，它提供 Perl 风格的正则表达式模式。

re 模块使 Python 语言拥有全部的正则表达式功能。

compile 函数根据一个模式字符串和可选的标志参数生成一个正则表达式对象。该对象拥有一系列方法用于正则表达式匹配和替换。

re 模块也提供了与这些方法功能完全一致的函数，这些函数使用一个模式字符串做为它们的第一个参数。

本章节主要介绍 Python 中常用的正则表达式处理函数，如果你对正则表达式不了解，可以查看我们的正则表达式 - 教程。

1.20.1 re.match 函数

re.match 尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功的话，match() 就返回 none。

函数语法:

```
re.match(pattern, string, flags=0)
```

函数参数说明:

| 参数 | 描述 | | —| —| | pattern | 匹配的正则表达式 | | string | 要匹配的字符串。| | flags | 标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。

匹配成功 re.match 方法返回一个匹配的对象，否则返回 None。

我们可以使用 group(num) 或 groups() 匹配对象函数来获取匹配表达式。

| 匹配对象方法 | 描述 | | —| —| | group(num=0) | 匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。| | groups() | 返回一个包含所有小组字符串的元组，从 1 到所含的小组号。|

```
import re
print(re.match('www', 'www.runoob.com').span())
print(re.match('com', 'www.runoob.com'))
```

以上实例运行输出结果为:

```
(0, 3)
None
```

实例

```
#!/usr/bin/python3
import re

line = "Cats are smarter than dogs"
# .* 表示任意匹配除换行符（\n、\r）之外的任何单个或多个字符
# (.*?) 表示"非贪婪"模式，只保存第一个匹配到的子串
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print ("matchObj.group() : ", matchObj.group())
   print ("matchObj.group(1) : ", matchObj.group(1))
   print ("matchObj.group(2) : ", matchObj.group(2))
else:
   print ("No match!!")
```

以上实例执行结果如下:

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

1.20.2 re.search 方法

re.search 扫描整个字符串并返回第一个成功的匹配。

函数语法:

```
re.search(pattern, string, flags=0)
```

函数参数说明:

| 参数 | 描述 | | —| —| | pattern | 匹配的正则表达式 | | string | 要匹配的字符串。| | flags | 标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。|

匹配成功 re.search 方法返回一个匹配的对象，否则返回 None。

我们可以使用 group(num) 或 groups() 匹配对象函数来获取匹配表达式。

| 匹配对象方法 | 描述 | | —| —| | group(num=0) | 匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。| | groups() | 返回一个包含所有小组字符串的元组，从 1 到所含的小组号。|

实例

```
import re
print(re.search('www', 'www.runoob.com').span())
print(re.search('com', 'www.runoob.com').span())
```

以上实例运行输出结果为:

```
(0, 3)
(11, 14)
```

实例

```
#!/usr/bin/python3

import re

line = "Cats are smarter than dogs"

searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)

if searchObj:
   print ("searchObj.group() : ", searchObj.group())
   print ("searchObj.group(1) : ", searchObj.group(1))
   print ("searchObj.group(2) : ", searchObj.group(2))
else:
   print ("Nothing found!!")
```

以上实例执行结果如下:

```
searchObj.group() :  Cats are smarter than dogs
searchObj.group(1) :  Cats
searchObj.group(2) :  smarter
```

1.20.3 re.match 与 re.search 的区别

re.match 只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回 None，而 re.search 匹配整个字符串，直到找到一个匹配。

实例

```
#!/usr/bin/python3

import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print ("match --> matchObj.group() : ", matchObj.group())
else:
   print ("No match!!")

matchObj = re.search( r'dogs', line, re.M|re.I)
if matchObj:
   print ("search --> matchObj.group() : ", matchObj.group())
else:
   print ("No match!!")
```

以上实例运行结果如下：

```
No match!!
search --> matchObj.group() :  dogs
```

1.20.4 compile 函数

compile 函数用于编译正则表达式，生成一个正则表达式（Pattern ）对象，供 match() 和 search() 这两个函数使用。

语法格式为：

```
re.compile(pattern[, flags])
```

参数：

- pattern : 一个字符串形式的正则表达式
- flags 可选，表示匹配模式，比如忽略大小写，多行模式等，具体参数为：
- re.I 忽略大小写 * re.L 表示特殊字符集 \w, \W, \b, \B, \s, \S 依赖于当前环境

- re.M 多行模式
- re.S 即为’.‘并且包括换行符在内的任意字符（’.‘不包括换行符）
- re.U 表示特殊字符集 \w, \W, \b, \B, \d, \D, \s, \S 依赖于 Unicode 字符属性数据库
- re.X 为了增加可读性，忽略空格和’#‘后面的注释

实例

```
>>>import re
>>> pattern = re.compile(r'\d+')                    # 用于匹配至少一个数字
>>> m = pattern.match('one12twothree34four')        # 查找头部，没有匹配
>>> print( m )
None
>>> m = pattern.match('one12twothree34four', 2, 10) # 从'e'的位置开始匹配，没有匹配
>>> print( m )
None
>>> m = pattern.match('one12twothree34four', 3, 10) # 从'1'的位置开始匹配，正好匹配
>>> print( m )                                       # 返回一个 Match 对象
<_sre.SRE_Match object at 0x10a42aac0>
>>> m.group(0)   # 可省略 0
'12'
>>> m.start(0)   # 可省略 0
3
>>> m.end(0)     # 可省略 0
5
>>> m.span(0)    # 可省略 0
(3, 5)
```

在上面，当匹配成功时返回一个 Match 对象，其中：

- `group([group1, …])` 方法用于获得一个或多个分组匹配的字符串，当要获得整个匹配的子串时，可直接使用 `group()` 或 `group(0)`；
- `start([group])` 方法用于获取分组匹配的子串在整个字符串中的起始位置（子串第一个字符的索引），参数默认值为 0；
- `end([group])` 方法用于获取分组匹配的子串在整个字符串中的结束位置（子串最后一个字符的索引+1），参数默认值为 0；
- `span([group])` 方法返回 `(start(group), end(group))`。

1.20.5 findall

在字符串中找到正则表达式所匹配的所有子串，并返回一个列表，如果有多个匹配模式，则返回元组列表，如果没有找到匹配的，则返回空列表。

注意： match 和 search 是匹配一次 findall 匹配所有。

语法格式为：

```
re.findall(pattern, string, flags=0)
或
pattern.findall(string[, pos[, endpos]])
```

参数：

- pattern 匹配模式。
- string 待匹配的字符串。
- pos 可选参数，指定字符串的起始位置，默认为 0。
- endpos 可选参数，指定字符串的结束位置，默认为字符串的长度。

查找字符串中的所有数字：

实例

```
import re
result1 = re.findall(r'\d+','runoob 123 google 456')
pattern = re.compile(r'\d+')   # 查找数字
result2 = pattern.findall('runoob 123 google 456')
result3 = pattern.findall('run88oob123google456', 0, 10)
print(result1)
print(result2)
print(result3)
```

输出结果：

```
['123', '456']
['123', '456']
['88', '12']
```

多个匹配模式，返回元组列表：

实例

```
import re

result \= re.findall(r'(\\w+)=(\\d+)', 'set width=20 and height=10')
print(result)
```

```
[('width', '20'), ('height', '10')]
```

1.20.6 re.split

split 方法按照能够匹配的子串将字符串分割后返回列表，它的使用形式如下：

```
re.split(pattern, string[, maxsplit=0, flags=0])
```

参数：

| 参数 | 描述 | | —| —| | pattern | 匹配的正则表达式 | | string | 要匹配的字符串。| | maxsplit | 分割次数，maxsplit=1 分割一次，默认为 0，不限制次数。| | flags | 标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。参见：正则表达式修饰符 - 可选标志 |

实例

```
>>>import re
>>> re.split('\W+', 'runoob, runoob, runoob.')
['runoob', 'runoob', 'runoob', '']
>>> re.split('(\W+)', ' runoob, runoob, runoob.')
['', ' ', 'runoob', ', ', 'runoob', ', ', 'runoob', '.', '']
>>> re.split('\W+', ' runoob, runoob, runoob.', 1)
['', 'runoob, runoob, runoob.']

>>> re.split('a*', 'hello world')   # 对于一个找不到匹配的字符串而言，split␣
↪不会对其作出分割
['hello world']
```

1.21 Python MySQL 数据库连接

1.21.1 什么是 PyMySQL？

PyMySQL 是在 Python3.x 版本中用于连接 MySQL 服务器的一个库，Python2 中则使用 mysqldb。

PyMySQL 遵循 Python 数据库 API v2.0 规范，并包含了 pure-Python MySQL 客户端库。

1.21.2 PyMySQL 安装

在使用 PyMySQL 之前，我们需要确保 PyMySQL 已安装。

PyMySQL 下载地址：https://github.com/PyMySQL/PyMySQL。

如果还未安装，我们可以使用以下命令安装最新版的 PyMySQL：

```
$ pip3 install PyMySQL
```

如果你的系统不支持 pip 命令，可以使用以下方式安装：

1、使用 git 命令下载安装包安装 (你也可以手动下载)：

```
$ git clone https://github.com/PyMySQL/PyMySQL
$ cd PyMySQL/
$ python3 setup.py install
```

2、如果需要制定版本号，可以使用 curl 命令来安装：

```
$ # X.X 为 PyMySQL 的版本号
$ curl -L https://github.com/PyMySQL/PyMySQL/tarball/pymysql-X.X | tar xz
$ cd PyMySQL*
$ python3 setup.py install
$ # 现在你可以删除 PyMySQL* 目录
```

** 注意：** 请确保您有 root 权限来安装上述模块。

> 安装的过程中可能会出现”ImportError: No module named setuptools”的错误提示，意思是你没有安装 setuptools，你可以访问https://pypi.python.org/pypi/setuptools 找到各个系统的安装方法。
>
> Linux 系统安装实例：
>
> ```
> $ wget https://bootstrap.pypa.io/ez_setup.py
> $ python3 ez_setup.py
> ```

1.21.3 数据库连接

连接数据库前，请先确认以下事项：

- 您已经创建了数据库 TESTDB.
- 在 TESTDB 数据库中您已经创建了表 EMPLOYEE
- EMPLOYEE 表字段为 FIRST_NAME, LAST_NAME, AGE, SEX 和 INCOME。
- 连接数据库 TESTDB 使用的用户名为 “testuser”，密码为 “test123”，你可以可以自己设定或者直接使用 root 用户名及其密码，Mysql 数据库用户授权请使用 Grant 命令。

- 在你的机子上已经安装了 Python **pymysql** 模块。

以下实例链接 Mysql 的 TESTDB 数据库：

实例 (Python 3.0+)

```
#!/usr/bin/python3

import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='testuser',
                     password='test123',
                     database='TESTDB')

# 使用 cursor() 方法创建一个游标对象 cursor
cursor = db.cursor()

# 使用 execute()  方法执行 SQL 查询
cursor.execute("SELECT VERSION()")

# 使用 fetchone() 方法获取单条数据.
data = cursor.fetchone()

print ("Database version : %s " % data)

# 关闭数据库连接
db.close()
```

执行以上脚本输出结果如下：

```
Database version : 5.5.20-log
```

1.21.4 创建数据库表

如果数据库连接存在我们可以使用 execute() 方法来为数据库创建表，如下所示创建表 EMPLOYEE：

实例 (Python 3.0+)

```
#!/usr/bin/python3

import pymysql

# 打开数据库连接
```

(下页继续)

(续上页)

```
db = pymysql.connect(host='localhost',
                     user='testuser',
                     password='test123',
                     database='TESTDB')

# 使用 cursor() 方法创建一个游标对象 cursor
cursor = db.cursor()

# 使用 execute() 方法执行 SQL, 如果表存在则删除
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# 使用预处理语句创建表
sql = """CREATE TABLE EMPLOYEE (
         FIRST_NAME  CHAR(20) NOT NULL,
         LAST_NAME  CHAR(20),
         AGE INT,
         SEX CHAR(1),
         INCOME FLOAT )"""

cursor.execute(sql)

# 关闭数据库连接
db.close()
```

1.21.5 数据库插入操作

以下实例使用执行 SQL INSERT 语句向表 EMPLOYEE 插入记录：

```
#!/usr/bin/python3

import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='testuser',
                     password='test123',
                     database='TESTDB')

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 插入语句
```

(下页继续)

(续上页)

```
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
         LAST_NAME, AGE, SEX, INCOME)
         VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
   # 执行sql语句
   cursor.execute(sql)
   # 提交到数据库执行
   db.commit()
except:
   # 如果发生错误则回滚
   db.rollback()

# 关闭数据库连接
db.close()
```

以上例子也可以写成如下形式：

实例 (Python 3.0+)

```
#!/usr/bin/python3

import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='testuser',
                     password='test123',
                     database='TESTDB')

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 插入语句
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
       LAST_NAME, AGE, SEX, INCOME) \
       VALUES ('%s', '%s',  %s,  '%s',  %s)" % \
       ('Mac', 'Mohan', 20, 'M', 2000)
try:
   # 执行sql语句
   cursor.execute(sql)
   # 执行sql语句
   db.commit()
except:
   # 发生错误时回滚
```

(下页继续)

(续上页)

```
   db.rollback()

# 关闭数据库连接
db.close()
```

以下代码使用变量向 SQL 语句中传递参数:

```
.................................
user_id = "test123"
password = "password"

con.execute('insert into Login values( %s,  %s)' % \
            (user_id, password))
.................................
```

1.22 数据库查询操作

Python 查询 Mysql 使用 fetchone() 方法获取单条数据, 使用 fetchall() 方法获取多条数据。

- **fetchone():** 该方法获取下一个查询结果集。结果集是一个对象
- **fetchall():** 接收全部的返回结果行.
- **rowcount:** 这是一个只读属性，并返回执行 execute() 方法后影响的行数。

查询 EMPLOYEE 表中 salary（工资）字段大于 1000 的所有数据:

```
#!/usr/bin/python3

import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='testuser',
                     password='test123',
                     database='TESTDB')

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 查询语句
sql = "SELECT * FROM EMPLOYEE \
       WHERE INCOME > %s" % (1000)
try:
```

(下页继续)

(续上页)

```
   # 执行SQL语句
   cursor.execute(sql)
   # 获取所有记录列表
   results = cursor.fetchall()
   for row in results:
      fname = row[0]
      lname = row[1]
      age = row[2]
      sex = row[3]
      income = row[4]
       # 打印结果
      print ("fname=%s,lname=%s,age=%s,sex=%s,income=%s" % \
             (fname, lname, age, sex, income ))
except:
   print ("Error: unable to fetch data")

# 关闭数据库连接
db.close()
```

以上脚本执行结果如下：

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

1.22.1 数据库更新操作

更新操作用于更新数据表的数据，以下实例将 TESTDB 表中 SEX 为‘M’的 AGE 字段递增 1：

```
#!/usr/bin/python3

import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='testuser',
                     password='test123',
                     database='TESTDB')

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 更新语句
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = '%c'" % ('M')
try:
```

(下页继续)

(续上页)

```
    # 执行SQL语句
    cursor.execute(sql)
    # 提交到数据库执行
    db.commit()
except:
    # 发生错误时回滚
    db.rollback()

# 关闭数据库连接
db.close()
```

1.22.2 删除操作

删除操作用于删除数据表中的数据，以下实例演示了删除数据表 EMPLOYEE 中 AGE 大于 20 的所有数据：

```
#!/usr/bin/python3

import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='testuser',
                     password='test123',
                     database='TESTDB')

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 删除语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > %s" % (20)
try:
    # 执行SQL语句
    cursor.execute(sql)
    # 提交修改
    db.commit()
except:
    # 发生错误时回滚
    db.rollback()

# 关闭连接
db.close()
```

1.22.3 执行事务

事务机制可以确保数据一致性。

事务应该具有 4 个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为 ACID 特性。

- 原子性（atomicity）。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
- 一致性（consistency）。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
- 隔离性（isolation）。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- 持久性（durability）。持续性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

Python DB API 2.0 的事务提供了两个方法 commit 或 rollback。

```
# SQL删除记录语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > %s" % (20)
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 向数据库提交
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()
```

对于支持事务的数据库，在 Python 数据库编程中，当游标建立之时，就自动开始了一个隐形的数据库事务。

commit() 方法游标的所有更新操作，rollback（）方法回滚当前游标的所有操作。每一个方法都开始了一个新的事务。

1.22.4 错误处理

DB API 中定义了一些数据库操作的错误及异常，下表列出了这些错误和异常:

| 异常 | 描述 | | —| —| | Warning | 当有严重警告时触发，例如插入数据是被截断等等。必须是 StandardError 的子类。| | Error | 警告以外所有其他错误类。必须是 StandardError 的子类。| | InterfaceError | 当有数据库接口模块本身的错误（而不是数据库的错误）发生时触发。必须是 Error 的子类。| | DatabaseError | 和数据库有关的错误发生时触发。必须是 Error 的子类。| | DataError | 当有数据处理时的错误发生时触发，例如：除零错误，数据超范围等等。必须是 DatabaseError 的子类。| | OperationalError | 指非用户控制的，而是操作数据库时发生的错误。例如：连接意外断开、数据库名未找到、事务处理失败、内存分配错误等等操作数据库是发生的错误。必须是 DatabaseError 的子类。| | IntegrityError | 完整性相关的错误，例如外键检查失败等。必须是

DatabaseError 子类。| | InternalError | 数据库的内部错误，例如游标（cursor）失效了、事务同步失败等等。必须是 DatabaseError 子类。| | ProgrammingError | 程序错误，例如数据表（table）没找到或已存在、SQL 语句语法错误、参数数量错误等等。必须是 DatabaseError 的子类。| | NotSupportedError | 不支持错误，指使用了数据库不支持的函数或 API 等。例如在连接对象上使用.rollback() 函数，然而数据库并不支持事务或者事务已关闭。必须是 DatabaseError 的子类。|

1.23 Python SMTP 发送邮件

SMTP（Simple Mail Transfer Protocol）即简单邮件传输协议, 它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。

python 的 smtplib 提供了一种很方便的途径发送电子邮件。它对 smtp 协议进行了简单的封装。

1.24 Python3 多线程

多线程类似于同时执行多个不同程序，多线程运行有如下优点：

- 使用线程可以把占据长时间的程序中的任务放到后台去处理。
- 用户界面可以更加吸引人，比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度。
- 程序的运行速度可能加快。
- 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下我们可以释放一些珍贵的资源如内存占用等等。

1.24.1 开始学习 Python 线程

Python 中使用线程有两种方式：函数或者用类来包装线程对象。

函数式：调用 _thread 模块中的 start_new_thread() 函数来产生新线程。语法如下:

_thread.start_new_thread (function, args[, kwargs]) 参数说明:

- function - 线程函数。
- args - 传递给线程函数的参数, 他必须是个 tuple 类型。
- kwargs - 可选参数。

```
#!/usr/bin/python3

import _thread
import time
```

(下页继续)

(续上页)

```
# 为线程定义一个函数
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print ("%s: %s" % ( threadName, time.ctime(time.time()) ))

# 创建两个线程
try:
   _thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   _thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print ("Error: 无法启动线程")

while 1:
   pass
```

执行以上程序输出结果如下：

```
Thread-1: Wed Jan  5 17:38:08 2022
Thread-2: Wed Jan  5 17:38:10 2022
Thread-1: Wed Jan  5 17:38:10 2022
Thread-1: Wed Jan  5 17:38:12 2022
Thread-2: Wed Jan  5 17:38:14 2022
Thread-1: Wed Jan  5 17:38:14 2022
Thread-1: Wed Jan  5 17:38:16 2022
Thread-2: Wed Jan  5 17:38:18 2022
Thread-2: Wed Jan  5 17:38:22 2022
Thread-2: Wed Jan  5 17:38:26 2022
```

1.24.2 线程模块

Python3 通过两个标准库 _thread 和 threading 提供对线程的支持。

_thread 提供了低级别的、原始的线程以及一个简单的锁，它相比于 threading 模块的功能还是比较有限的。

threading 模块除了包含 _thread 模块中的所有方法外，还提供的其他方法：

- threading.currentThread(): 返回当前的线程变量。
- threading.enumerate(): 返回一个包含正在运行的线程的 list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
- threading.activeCount(): 返回正在运行的线程数量，与 len(threading.enumerate()) 有相同的结果。除了使用

方法外，线程模块同样提供了 Thread 类来处理线程，Thread 类提供了以下方法:

- run(): 用以表示线程活动的方法。
- start(): 启动线程活动。
- join([time]): 等待至线程中止。这阻塞调用线程直至线程的 join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
- isAlive(): 返回线程是否活动的。
- getName(): 返回线程名。
- setName(): 设置线程名。

1.24.3 使用 threading 模块创建线程

我们可以通过直接从 threading.Thread 继承创建一个新的子类，并实例化后调用 start() 方法启动新线程，即它调用了线程的 run() 方法:

```
#!/usr/bin/python3

import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, delay):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.delay = delay
    def run(self):
        print ("开始线程：" + self.name)
        print_time(self.name, self.delay, 5)
        print ("退出线程：" + self.name)

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            threadName.exit()
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

# 创建新线程
```

(下页继续)

(续上页)

```
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print ("退出主线程")
```

以上程序执行结果如下;

```
开始线程: Thread-1
开始线程: Thread-2
Thread-1: Wed Jan  5 17:34:54 2022
Thread-2: Wed Jan  5 17:34:55 2022
Thread-1: Wed Jan  5 17:34:55 2022
Thread-1: Wed Jan  5 17:34:56 2022
Thread-2: Wed Jan  5 17:34:57 2022
Thread-1: Wed Jan  5 17:34:57 2022
Thread-1: Wed Jan  5 17:34:58 2022
退出线程: Thread-1
Thread-2: Wed Jan  5 17:34:59 2022
Thread-2: Wed Jan  5 17:35:01 2022
Thread-2: Wed Jan  5 17:35:03 2022
退出线程: Thread-2
退出主线程
```

1.24.4 线程同步

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。

使用 Thread 对象的 Lock 和 Rlock 可以实现简单的线程同步，这两个对象都有 acquire 方法和 release 方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到 acquire 和 release 方法之间。如下:

多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。

考虑这样一种情况：一个列表里所有元素都是 0，线程”set”从后向前把所有元素改成 1，而线程”print”负责从前往后读取列表并打印。

那么，可能线程”set”开始改的时候，线程”print”便来打印列表了，输出就成了一半 0 一半 1，这就是数据的不同步。为了避免这种情况，引入了锁的概念。

锁有两种状态——锁定和未锁定。每当一个线程比如”set”要访问共享数据时，必须先获得锁定；如果已经有别的线程比如”print”获得锁定了，那么就让线程”set”暂停，也就是同步阻塞；等到线程”print”访问完毕，释放锁以后，再让线程”set”继续。

经过这样的处理，打印列表时要么全部输出 0，要么全部输出 1，不会再出现一半 0 一半 1 的尴尬场面。

```
#!/usr/bin/python3

import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, delay):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.delay = delay
    def run(self):
        print ("开启线程：  " + self.name)
        # 获取锁，用于线程同步
        threadLock.acquire()
        print_time(self.name, self.delay, 3)
        # 释放锁，开启下一个线程
        threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

threadLock = threading.Lock()
threads = []

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
thread2.start()

# 添加线程到线程列表
threads.append(thread1)
threads.append(thread2)
```

(下页继续)

(续上页)

```
# 等待所有线程完成
for t in threads:
    t.join()
print ("退出主线程")
```

执行以上程序，输出结果为:

```
开启线程： Thread-1
开启线程： Thread-2
Thread-1: Wed Jan  5 17:36:50 2022
Thread-1: Wed Jan  5 17:36:51 2022
Thread-1: Wed Jan  5 17:36:52 2022
Thread-2: Wed Jan  5 17:36:54 2022
Thread-2: Wed Jan  5 17:36:56 2022
Thread-2: Wed Jan  5 17:36:58 2022
退出主线程
```

1.24.5 线程优先级队列（Queue）

Python 的 Queue 模块中提供了同步的、线程安全的队列类，包括 FIFO（先入先出) 队列 Queue，LIFO（后入先出）队列 LifoQueue，和优先级队列 PriorityQueue。

这些队列都实现了锁原语，能够在多线程中直接使用，可以使用队列来实现线程间的同步。

Queue 模块中的常用方法:

- Queue.qsize() 返回队列的大小
- Queue.empty() 如果队列为空，返回 True, 反之 False
- Queue.full() 如果队列满了，返回 True, 反之 False
- Queue.full 与 maxsize 大小对应
- Queue.get([block[, timeout]]) 获取队列，timeout 等待时间
- Queue.get_nowait() 相当 Queue.get(False)
- Queue.put(item) 写入队列，timeout 等待时间
- Queue.put_nowait(item) 相当 Queue.put(item, False)
- Queue.task_done() 在完成一项工作之后，Queue.task_done() 函数向任务已经完成的队列发送一个信号
- Queue.join() 实际上意味着等到队列为空，再执行别的操作

```
#!/usr/bin/python3

import queue
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print ("开启线程：" + self.name)
        process_data(self.name, self.q)
        print ("退出线程：" + self.name)

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print ("%s processing %s" % (threadName, data))
        else:
            queueLock.release()
        time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = queue.Queue(10)
threads = []
threadID = 1

# 创建新线程
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1
```

(下页继续)

(续上页)

```
# 填充队列
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# 等待队列清空
while not workQueue.empty():
    pass

# 通知线程是时候退出
exitFlag = 1

# 等待所有线程完成
for t in threads:
    t.join()
print ("退出主线程")
```

以上程序执行结果：

```
开启线程： Thread-1
开启线程： Thread-2
开启线程： Thread-3
Thread-3 processing One
Thread-1 processing Two
Thread-2 processing Three
Thread-3 processing Four
Thread-1 processing Five
退出线程： Thread-3
退出线程： Thread-2
退出线程： Thread-1
退出主线程
```

1.25 Python JSON 数据解析

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式。

Python3 中可以使用 json 模块来对 JSON 数据进行编解码，它包含了两个函数：

- json.dumps(): 对数据进行编码。
- json.loads(): 对数据进行解码。

在 json 的编解码过程中，Python 的原始类型与 json 类型会相互转换，具体的转化对照如下：

1.25.1 Python 编码为 JSON 类型转换对应表：

| Python | JSON | | —| —| | dict | object | | list, tuple | array | | str | string | | int, float, int- & float-derived Enums | number | | True | true | | False | false | | None | null |

1.25.2 JSON 解码为 Python 类型转换对应表：

| JSON | Python | | —| —| | object | dict | | array | list | | string | str | | number (int) | int | | number (real) | float | | true | True | | false | False | | null | None |

1.25.3 json.dumps 与 json.loads 实例

以下实例演示了 Python 数据结构转换为 JSON：

```
#!/usr/bin/python3

import json

# Python 字典类型转换为 JSON 对象
data = {
    'no' : 1,
    'name' : 'Runoob',
    'url' : 'http://www.runoob.com'
}

json_str = json.dumps(data)
print ("Python 原始数据： ", repr(data))
print ("JSON 对象： ", json_str)
```

执行以上代码输出结果为：

```
Python 原始数据：  {'url': 'http://www.runoob.com', 'no': 1, 'name': 'Runoob'}
JSON 对象：  {"url": "http://www.runoob.com", "no": 1, "name": "Runoob"}
```

通过输出的结果可以看出，简单类型通过编码后跟其原始的 repr() 输出结果非常相似。

接着以上实例，我们可以将一个 JSON 编码的字符串转换回一个 Python 数据结构：

```
#!/usr/bin/python3

import json

# Python 字典类型转换为 JSON 对象
data1 = {
```

(下页继续)

(续上页)

```
    'no' : 1,
    'name' : 'Runoob',
    'url' : 'http://www.runoob.com'
}

json_str = json.dumps(data1)
print ("Python 原始数据：", repr(data1))
print ("JSON 对象：", json_str)

# 将 JSON 对象转换为 Python 字典
data2 = json.loads(json_str)
print ("data2['name']: ", data2['name'])
print ("data2['url']: ", data2['url'])
```

执行以上代码输出结果为:

```
Python 原始数据：  {'name': 'Runoob', 'no': 1, 'url': 'http://www.runoob.com'}
JSON 对象：  {"name": "Runoob", "no": 1, "url": "http://www.runoob.com"}
data2['name']:  Runoob
data2['url']:  http://www.runoob.com
```

如果你要处理的是文件而不是字符串，你可以使用 **json.dump()** 和 **json.load()** 来编码和解码 JSON 数据。例如:

```
# 写入 JSON 数据
with open('data.json', 'w') as f:
    json.dump(data, f)

# 读取数据
with open('data.json', 'r') as f:
    data = json.load(f)
```

1.26 Python 日期和时间

Python 程序能用很多方式处理日期和时间，转换日期格式是一个常见的功能。

Python 提供了一个 time 和 calendar 模块可以用于格式化日期和时间。

时间间隔是以秒为单位的浮点小数。

每个时间戳都以自从 1970 年 1 月 1 日午夜（历元）经过了多长时间来表示。

Python 的 time 模块下有很多函数可以转换常见日期格式。如函数 time.time() 用于获取当前时间戳, 如下实例:

```
#!/usr/bin/python3

import time  # 引入time模块

ticks = time.time()
print ("当前时间戳为:", ticks)
```

以上实例输出结果：

```
当前时间戳为: 1459996086.7115328
```

时间戳单位最适于做日期运算。但是 1970 年之前的日期就无法以此表示了。太遥远的日期也不行，UNIX 和 Windows 只支持到 2038 年。

1.26.1 什么是时间元组？

很多 Python 函数用一个元组装起来的 9 组数字处理时间:

| 序号 | 字段 | 值 | | —| —| —| | 0 | 4 位数年 | 2008 | | 1 | 月 | 1 到 12 | | 2 | 日 | 1 到 31 | | 3 | 小时 | 0 到 23 | | 4 | 分钟 | 0 到 59 | | 5 | 秒 | 0 到 61 (60 或 61 是闰秒) | | 6 | 一周的第几日 | 0 到 6 (0 是周一) | | 7 | 一年的第几日 | 1 到 366 (儒略历) | | 8 | 夏令时 | -1, 0, 1, -1 是决定是否为夏令时的标识 |

上述也就是 struct_time 元组。这种结构具有如下属性：

| 序号 | 属性 | 值 | | —| —| —| | 0 | tm_year | 2008 | | 1 | tm_mon | 1 到 12 | | 2 | tm_mday | 1 到 31 | | 3 | tm_hour | 0 到 23 | | 4 | tm_min | 0 到 59 | | 5 | tm_sec | 0 到 61 (60 或 61 是闰秒) | | 6 | tm_wday | 0 到 6 (0 是周一) | | 7 | tm_yday | 一年中的第几天，1 到 366 | | 8 | tm_isdst | 是否为夏令时，值有：1(夏令时)、0(不是夏令时)、-1(未知)，默认 -1 |

1.26.2 获取格式化的时间

你可以根据需求选取各种格式，但是最简单的获取可读的时间模式的函数是 asctime():

```
#!/usr/bin/python3

import time

localtime = time.asctime( time.localtime(time.time()) )
print ("本地时间为 :", localtime)
```

以上实例输出结果：

```
本地时间为 : Thu Apr  7 10:29:13 2016
```

1.26.3 格式化日期

我们可以使用 time 模块的 strftime 方法来格式化日期：

```
time.strftime(format[, t])
```

1.26.4 实例

```
#!/usr/bin/python3

import time

\# 格式化成2016-03-20 11:45:39形式
print (time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))

\# 格式化成Sat Mar 28 22:24:24 2016形式
print (time.strftime("%a %b %d %H:%M:%S %Y", time.localtime()))

  \# 将格式字符串转换为时间戳
a \= "Sat Mar 28 22:24:24 2016"
print (time.mktime(time.strptime(a,"%a %b %d %H:%M:%S %Y")))
```

以上实例输出结果：

```
2016-04-07 10:29:46
Thu Apr 07 10:29:46 2016
1459175064.0
```

python 中时间日期格式化符号：

- %y 两位数的年份表示（00-99）
- %Y 四位数的年份表示（000-9999）
- %m 月份（01-12）
- %d 月内中的一天（0-31）
- %H 24 小时制小时数（0-23）
- %I 12 小时制小时数（01-12）
- %M 分钟数（00=59）
- %S 秒（00-59）
- %a 本地简化星期名称
- %A 本地完整星期名称

- %b 本地简化的月份名称
- %B 本地完整的月份名称
- %c 本地相应的日期表示和时间表示
- %j 年内的一天（001-366）
- %p 本地 A.M. 或 P.M. 的等价符
- %U 一年中的星期数（00-53）星期天为星期的开始
- %w 星期（0-6），星期天为星期的开始
- %W 一年中的星期数（00-53）星期一为星期的开始
- %x 本地相应的日期表示
- %X 本地相应的时间表示
- %Z 当前时区的名称
- %% % 号本身

1.26.5 获取某月日历

Calendar 模块有很广泛的方法用来处理年历和月历，例如打印某月的月历：

实例

```
#!/usr/bin/python3

import calendar

cal \= calendar.month(2016, 1)
print ("以下输出2016年1月份的日历:")
print (cal)
```

以上实例输出结果：

```
以下输出2016年1月份的日历:
    January 2016
Mo Tu We Th Fr Sa Su
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

1.26.6 Time 模块

Time 模块包含了以下内置函数，既有时间处理的，也有转换时间格式的：

| 序号 | 函数及描述 | 实例 | | —| —| —| | 1 | time.altzone 返回格林威治西部的夏令时地区的偏移秒数。如果该地区在格林威治东部会返回负值（如西欧，包括英国）。对夏令时启用地区才能使用。| 以下实例展示了 altzone() 函数的使用方法：

```
>>> import time
>>> print ("time.altzone %d " % time.altzone)
time.altzone -28800
```

| 2 | time.asctime([tupletime]) 接受时间元组并返回一个可读的形式为”Tue Dec 11 18:07:14 2008”（2008 年 12 月 11 日周二 18 时 07 分 14 秒）的 24 个字符的字符串。|

以下实例展示了 asctime() 函数的使用方法：

```
>>> import time
>>> t = time.localtime()
>>> print ("time.asctime(t): %s " % time.asctime(t))
time.asctime(t): Thu Apr  7 10:36:20 2016
```

用以浮点数计算的秒数返回当前的 CPU 时间。用来衡量不同程序的耗时，比 time.time() 更有用。

由于该方法依赖操作系统，在 Python 3.3 以后不被推荐，而在 3.8 版本中被移除，需使用下列两个函数替代。

```
time.perf_counter()  # 返回系统运行时间
time.process_time()  # 返回进程运行时间
```

以下实例展示了 ctime() 函数的使用方法：

```
>>> import time
>>> print ("time.ctime() : %s" % time.ctime())
time.ctime() : Thu Apr  7 10:51:58 2016
```

以下实例展示了 gmtime() 函数的使用方法：

```
>>> import time
>>> print ("gmtime :", time.gmtime(1455508609.34375))
gmtime : time.struct_time(tm_year=2016, tm_mon=2, tm_mday=15, tm_hour=3, tm_min=56,␣
↪tm_sec=49, tm_wday=0, tm_yday=46, tm_isdst=0)
```

以下实例展示了 localtime() 函数的使用方法：

```
>>> import time
>>> print ("localtime(): ", time.localtime(1455508609.34375))
localtime():  time.struct_time(tm_year=2016, tm_mon=2, tm_mday=15, tm_hour=11, tm_
↪min=56, tm_sec=49, tm_wday=0, tm_yday=46, tm_isdst=0)
```

(下页继续)

(续上页)

以下实例展示了 sleep() 函数的使用方法：

```
#!/usr/bin/python3
import time

print ("Start : %s" % time.ctime())
time.sleep( 5 )
print ("End : %s" % time.ctime())
```

以下实例展示了 strftime() 函数的使用方法：

```
>>> import time
>>> print (time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))
2016-04-07 11:18:05
```

以下实例展示了 strptime() 函数的使用方法：

```
>>> import time
>>> struct_time = time.strptime("30 Nov 00", "%d %b %y")
>>> print ("返回元组: ", struct_time)
```

以下实例展示了 time() 函数的使用方法：

```
>>> import time
>>> print(time.time())
1459999336.1963577
```

Time 模块包含了以下 2 个非常重要的属性：

| 序号 | 属性及描述 | | —| —| | 1 | **time.timezone** 属性 time.timezone 是当地时区（未启动夏令时）距离格林威治的偏移秒数（>0，美洲;<=0 大部分欧洲，亚洲，非洲）。| | 2 | **time.tzname** 属性 time.tzname 包含一对根据情况的不同而不同的字符串，分别是带夏令时的本地时区名称，和不带的。|

1.26.7 日历（Calendar）模块

此模块的函数都是日历相关的，例如打印某月的字符月历。

星期一是默认的每周第一天，星期天是默认的最后一天。更改设置需调用 calendar.setfirstweekday() 函数。模块包含了以下内置函数：

| 序号 | 函数及描述 | | —| —| | 1 | **calendar.calendar(year,w=2,l=1,c=6)** 返回一个多行字符串格式的 year 年年历，3 个月一行，间隔距离为 c。每日宽度间隔为 w 字符。每行长度为 21* W+18+2* C。l 是每星期行数。| | 2 | **calendar.firstweekday( )** 返回当前每周起始日期的设置。默认情况下，首次载入 calendar 模块时返回 0，即星期一。| | 3 | **calendar.isleap(year)** 是闰年返回 True，否则为 False。

```
>>> import calendar
>>> print(calendar.isleap(2000))
True
>>> print(calendar.isleap(1900))
False
```

返回两个整数。第一个是该月的星期几，第二个是该月有几天。星期几是从 0（星期一）到 6（星期日）。

```
>>> import calendar
>>> calendar.monthrange(2014, 11)
(5, 30)
```

1.27 待补充

CHAPTER 2

第二章：python 进阶

2.1 Python 进阶