本笔记基于python官网3.13.2教程

1.课前甜点

2.使用python解释器

3.python速览

4.更多控制流工具

4.1 if语句

可有零个或多个 `elif` 部分，`else` 部分也是可选的。关键字 '`elif`' 是 'else if' 的缩写，用于避免过多的缩进。`if` ... `elif` ... `elif` ... 序列可以当作其它语言中 `switch` 或 `case` 语句的替代品。

如果是把一个值与多个常量进行比较，或者检查特定类型或属性，`match` 语句更有用。详见 match 语句。

4.2 for语句

在可迭代序列中中迭代

```
# 度量一些字符串:
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

for循环如果想要在迭代的同时修改迭代对象的内容，基本采用下面两种方式

- 迭代多项集的副本
- 创建新的多项集

```
# 创建示例多项集
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# 策略：迭代一个副本
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# 策略：创建一个新多项集
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3 range函数

4.4 break和continnue

4.5 循环的elsse语句

在for或while循环中break语句可以对应一个else子句，如果循环不是通过break的情况结束，else子句会执行

> 当配合循环使用时，`else` 子句更像是 `try` 语句的 `else` 子句而不像 `if` 语句的相应子句：一个 `try` 语句的 `else` 子句会在未发生异常时运行，而一个循环的 `else` 子句会在未发生 `break` 时运行。

4.6 pass语句

pass语句不执行任何动作

4.7 match语句 （*）

match语句接受一个表达式并把其值与多个case块进行比较（类似swich语句）

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

变量名 _ 是通配符，必定匹配成功

...

4.8 定义函数

关键字def、形参、缩进、return

4.9 函数定义详解

函数定义支持可变数量的参数

- 位置参数
- 关键字参数

> **重要警告：** 默认值只计算一次。默认值为列表、字典或类实例等可变对象时，会产生与该规则不同的结果。例如，下面的函数会累积后续调用时传递的参数：
>
> ```
> def f(a, L=[]):
>  L.append(a)
>  return L
>
> print(f(1))
> print(f(2))
> print(f(3))
> ```
>
> 输出结果如下：
>
> ```
> [1]
> [1, 2]
> [1, 2, 3]
> ```
>
> 不想在后续调用之间共享默认值时，应以如下方式编写函数：
>
> ```
> def f(a, L=None):
>  if L is None:
>      L = []
>  L.append(a)
>  return L
> ```

参数类型限制

- / 前面位置参数
- *后面关键字参数

*在变量名前如果在形参和实参下有不同含义

- 形参 *args，接受任意数量位置参数，元组传入
- 形参**kwargs，接受任意数量关键字参数，字典传入
- 实参*args，解包位置参数
- 实参**kwargs，解包关键字参数

lambda表达式

lambda a,b:a+b

5.数据结构（*）

5.1 列表

添加

- list.**append**(*x*)

  在列表末尾添加一项。 类似于 `a[len(a):] = [x]`。

- list.**extend**(*iterable*)

  通过添加来自 iterable 的所有项来扩展列表。 类似于 `a[len(a):] = iterable`。

- list.**insert**(*i*, *x*)

  在指定位置插入元素。第一个参数是插入元素的索引，因此，`a.insert(0, x)` 在列表开头插入元素，`a.insert(len(a), x)` 等同于 `a.append(x)`。

删除

- list.**remove**(*x*)

  从列表中删除第一个值为 *x* 的元素。未找到指定元素时，触发 `ValueError` 异常。

- list.**pop**([*i*])

  移除列表中给定位置上的条目，并返回该条目。 如果未指定索引号，则 `a.pop()` 将移除并返回列表中的最后一个条目。 如果列表为空或索引号在列表索引范围之外则会引发 `IndexError`。

- list.**clear**()

  移除列表中的所有项。 类似于 `del a[:]`。

其他

- list.**index**(*x*[, *start*[, *end*]])

  返回列表中第一个值为 *x* 的元素的零基索引。未找到指定元素时，触发 `ValueError` 异常。可选参数 *start* 和 *end* 是切片符号，用于将搜索限制为列表的特定子序列。返回的索引是相对于整个序列的开始计算的，而不是 *start* 参数。

- list.**count**(*x*)

  返回列表中元素 *x* 出现的次数。

- list.**sort**(***, *key=None*, *reverse=False*)

  就地排序列表中的元素（要了解自定义排序参数，详见 `sorted()`）。

- list.**reverse**()

  翻转列表中的元素。

- list.**copy**()

  返回列表的浅拷贝。 类似于 `a[:]`。

> `insert`, `remove` 或 `sort` 等仅修改列表的方法都不会打印返回值 -- 它们返回默认值 `None`。 [1]
> 这是适用于 Python 中所有可变数据结构的设计原则。

5.1.1 列表实现堆栈

append和pop

5.1.2 列表实现队列

列表作为队列效率低，最好使用collections.deque

```
from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry 到了
>>> queue.append("Graham")          # Graham 到了
>>> queue.popleft()                 # 第一个到的现在走了
'Eric'
>>> queue.popleft()                 # 第二个到的现在走了
'John'
>>> queue                           # 按到达顺序排列的剩余队列
deque(['Michael', 'Terry', 'Graham'])
```

deque定义，append添加，popleft删除

5.2 del语句

del语句用来删除切片位置或清空

5.3 元组和序列

和列表一样有索引和切片操作

元组：有多个逗号隔开的值，不可变

```
不可变对象 ：int，string，float，tuple    -- 可理解为C中，该参数为值传递
可变对象   ：list，dictionary             -- 可理解为C中，该参数为指针传递
```

5.4 集合（*）

集合是由不重复匀速组成的无序容器

基本用法

- 成员检测
- 消除重复元素

创建方法

- set函数
- {}，空集合是不能用，因为{}创建的是空字典

集合支持合集、交集等运算

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # 显示重复项已被移除
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # 快速成员检测
True
>>> 'crabgrass' in basket
False

>>> # 演示针对两个单词中独有的字母进行集合运算
>>>
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # a 中独有的字母
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # 存在于 a 中但不存在于 b 中的字母
{'r', 'd', 'b'}
>>> a | b                              # 存在于 a 或 b 中或两者中皆有的字母
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # 同时存在于 a 和 b 中的字母
{'a', 'c'}
>>> a ^ b                              # 存在于 a 或 b 中但非两者中皆有的字母
```

5.5 字典（*）

与以连续整数为索引的序列不同，字典是以 *键* 来索引的，键可以是任何不可变类型；字符串和数字总是可以作为键。

> 元组在其仅包含字符串、数字或元组时也可以作为键；如果一个元组直接或间接地包含了任何可变对象，则不可以用作键。 你不能使用列表作为键，因为列表可使用索引赋值、切片赋值或 `append()` 和 `extend()` 等方法进行原地修改。

创建字典

- {}
- dict([key,value],...)

> 对字典执行 `list(d)` 操作，返回该字典中所有键的列表，按插入次序排列（如需排序，请使用 `sorted(d)`）。检查字典里是否存在某个键，使用关键字 `in`。

5.6 循环的技巧（*）

字典循环，可以使用items方法同时提取键和值

序列循环，可以使用enumeate函数同时提取位置索引和值

同时循环多个序列，使用zip函数可以讲其中元素一一匹配

```
questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

按指定顺序循环序列，可以用 `sorted()` 函数，在不改动原序列的基础上，返回一个重新的序列

使用 `set()` 去除序列中的重复元素。使用 `sorted()` 加 `set()` 则按排序后的顺序，循环遍历序列中的唯一元素

一般来说，在循环中修改列表的内容时，创建新列表比较简单，且安全

5.7 深入条件控制

布尔运算符 `and` 和 `or` 是所谓的 *短路* 运算符：其参数从左至右求值，一旦可以确定结果，求值就会停止。

短路运算符的返回值通常是最后一个求了值的参数。

```
string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

> 注意，Python 与 C 不同，在表达式内部赋值必须显式使用 海象运算符 `:=`。 这避免了 C 程序中常见的问题：要在表达式中写 `==` 时，却写成了 `=`。

5.8 序列和其他类型的比较

序列对象可以与相同序列类型的其他对象比较。

当比较不同类型的对象时，只要待比较的对象提供了合适的比较方法，就可以使用 `<` 和 `>` 进行比较。例如，混合的数字类型通过数字值进行比较，所以，0 等于 0.0，等等。如果没有提供合适的比较方法，解释器不会随便给出一个比较结果，而是引发 `TypeError` 异常。

6.模块 （*）

模块就是包含python定义和语句的文件。其文件名是模块名加后缀.py。

在模块内部，通过全局变量 `__name__` 可以获取模块名（即字符串）。

6.1 模块详解

模块可以导入其他模块。 根据惯例可以将所有 `import` 语句都放在模块（或者也可以说是脚本）的开头但这并非强制要求。 如果被放置于一个模块的最高层级，则被导入的模块名称会被添加到该模块的全局命名空间。

模块名后使用 `as` 时，直接把 `as` 后的名称与导入模块绑定。

```
import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1.1 以脚本方式执行模块

可以用以下方式运行 Python 模块：

```
python fibo.py <参数>
```

这项操作将执行模块里的代码，和导入模块一样，但会把 `__name__` 赋值为 `"__main__"`。 也就是把下列代码添加到模块末尾：

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

这个文件既能被用作脚本，又能被用作一个可供导入的模块，因为解析命令行参数的那两行代码只有在模块作为“main”文件执行时才会运行：

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

当这个模块被导入到其它模块时，那两行代码不运行：

>>>

```
>>> import fibo
>>>
```

这常用于为模块提供一个便捷的用户接口，或用于测试（把模块作为执行测试套件的脚本运行）。

6.1.2 模块搜索路径

当导入一个名为 `spam` 的模块时，解释器首先会搜索具有该名称的内置模块。 这些模块的名称在 `sys.builtin_module_names` 中列出。 如果未找到，它将在变量 `sys.path` 所给出的目录列表中搜索名为 `spam.py` 的文件。 `sys.path` 是从这些位置初始化的:

- 被命令行直接运行的脚本所在的目录（或未指定文件时的当前目录）。
- `PYTHONPATH` （目录列表，与 shell 变量 `PATH` 的语法一样）。
- 依赖于安装的默认值（按照惯例包括一个 `site-packages` 目录，由 `site` 模块处理）。

初始化后，Python 程序可以更改 `sys.path`。脚本所在的目录先于标准库所在的路径被搜索。这意味着，脚本所在的目录如果有和标准库同名的文件，那么加载的是该目录里的，而不是标准库的。这一般是一个错误，除非这样的替换是你有意为之。详见 标准模块。

6.1.3 “已编译”的python文件

为了快速加载模块，Python 把模块的编译版本缓存在 `__pycache__` 目录中，文件名为 `module.*version*.pyc`，version 对编译文件格式进行编码，一般是 Python 的版本号。例如

Python 对比编译版与源码的修改日期，查看编译版是否已过期，是否要重新编译。此进程完全是自动的。此外，编译模块与平台无关，因此，可在不同架构的系统之间共享相同的库。

Python 在两种情况下不检查缓存。

- 一，从命令行直接载入的模块，每次都会重新编译，且不储存编译结果；
- 二，没有源模块，就不会检查缓存。

为了让一个库能以隐藏源代码的形式分发（通过将所有源代码变为编译后的版本），编译后的模块必须放在源目录而非缓存目录中，并且源目录绝不能包含同名的未编译的源模块。

简而言之：就是**pycache**让模块导入更快

6.2 标准模块

python自带一个标准模块的库

一些模块是内嵌到解释器里面的， 它们给一些虽并非语言核心但却内嵌的操作提供接口，要么是为了效率，要么是给操作系统基础操作例如系统调入提供接口。

> 变量 `sys.path` 是字符串列表，用于确定解释器的模块搜索路径。该变量以环境变量 `PYTHONPATH` 提取的默认路径进行初始化，如未设置 `PYTHONPATH`，则使用内置的默认路径。可以用标准列表操作修改该变量：

6.3 dir函数

内置函数dir用于查找模块定义的名称，返回结果是经过排序的字符串列表

没有参数时，dir列车当前已定义的名称（不包含内置函数和变量-builtins）

6.4 包 （*）

包是通过使用“带点号模块名”来构造 Python 模块命名空间的一种方式。 例如，模块名 `A.B` 表示名为 `A` 的包中名为 `B` 的子模块。 就像使用模块可以让不同模块的作者不必担心彼此的全局变量名一样，使用带点号模块名也可以让 NumPy 或 Pillow 等多模块包的作者也不必担心彼此的模块名冲突。

需要有 `__init__.py` 文件才能让 Python 将包含该文件的目录当作包来处理（除非使用 namespace package，这是一个相对高级的特性）。 这可以防止重名的目录如 `string` 在无意中屏蔽后继出现在模块搜索路径中的有效模块。 在最简单的情况下，`__init__.py` 可以只是一个空文件，但它也可以执行包的初始化代码或设置 `__all__` 变量

> 注意，使用 `from package import item` 时，item 可以是包的子模块（或子包），也可以是包中定义的函数、类或变量等其他名称。`import` 语句首先测试包中是否定义了 item；如果未在包中定义，则假定 item 是模块，并尝试加载。如果找不到 item，则触发 `ImportError` 异常。
>
> 相反，使用 `import item.subitem.subsubitem` 句法时，除最后一项外，每个 item 都必须是包；最后一项可以是模块或包，但不能是上一项中定义的类、函数或变量。

6.4.1 从包中导入 *

使用 `from sound.effects import *` 时会发生什么？你可能希望它会查找并导入包的所有子模块，但事实并非如此。因为这将花费很长的时间，并且可能会产生你不想要的副作用，如果这种副作用被你设计为只有在导入某个特定的子模块时才应该发生。

唯一的解决办法是提供包的显式索引。`import` 语句使用如下惯例：如果包的 `__init__.py` 代码定义了列表 `__all__`，运行 `from package import *` 时，它就是被导入的模块名列表。发布包的新版本时，包的作者应更新此列表。如果包的作者认为没有必要在包中执行导入 * 操作，也可以不提供此列表。例如，`sound/effects/__init__.py` 文件可以包含以下代码：

```
__all__ = ["echo", "surround", "reverse"]
```

这意味着 `from sound.effects import *` 将导入 `sound.effects` 包的三个命名子模块。

请注意子模块可能会受到本地定义名称的影响。 例如，如果你在 `sound/effects/__init__.py` 文件中添加了一个 `reverse` 函数，`from sound.effects import *` 将只导入 `echo` 和 `surround` 这两个子模块，但 **不会** 导入 `reverse` 子模块，因为它被本地定义的 `reverse` 函数所遮挡:

```
__all__ = [
    "echo",      # 指向 'echo.py' 文件
    "surround",  # 指向 'surround.py' 文件
    "reverse",   # !!! 现在指向 'reverse' 函数 !!!
]

def reverse(msg: str):  # <-- 此名称将覆盖 'reverse.py' 子模块
    return msg[::-1]    #     针对 'from sound.effects import *' 的情况
```

6.4.2 相对导入

6.4.3 多目录中的包

7.输入与输出（*）

7.1 更复杂的输出格式 （*）

格式化输出包括

- 使用格式化字符串字面量，f 'xx{变量名}'
- str.forma()方法

7.1.1 格式化字符串字面量

f'{变量名:格式}'

```
import math
print(f'The value of pi is approximately {math.pi:.3f}.')
```

在 `':'` 后传递整数，为该字段设置最小字符宽度，常用于列对齐：

>>>

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

`=` 说明符可被用于将一个表达式扩展为表达式文本、等号再加表达式求值结果的形式。

```
>>>
```

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

7.1.2 字符串format方法

7.2 读写文件（*）

`open()` 返回一个 file object，最常使用的是两个位置参数和一个关键字参数：`open(filename, mode, encoding=None)`

```
f = open('workfile', 'w', encoding="utf-8")
```

第一个实参是文件名字符串。

第二个实参是包含描述文件使用方式字符的字符串。

mode 的值包括

- `'r'`，表示文件只能读取；
- `'w'` 表示只能写入（现有同名文件会被覆盖）；
- `'a'` 表示打开文件并追加内容，任何写入的数据会自动添加到文件末尾。
- `'r+'` 表示打开文件进行读写。

mode 实参是可选的，省略时的默认值为 `'r`

UTF-8 是现代事实上的标准，除非你知道你需要使用一个不同的编码，否则建议使用 `encoding="utf-8"` 。

在模式后面加上一个 `'b'`，可以用 *binary mode* 打开文件。二进制模式的数据是以 `bytes` 对象的形式读写的。在二进制模式下打开文件时，你不能指定 *encoding*

> 在文本模式下读取文件时，默认把平台特定的行结束符（Unix 上为 `\n`，Windows 上为 `\r\n`）转换为 `\n`。在文本模式下写入数据时，默认把 `\n` 转换回平台特定结束符。这种操作方式在后台修改文件数据对文本文件来说没有问题，但会破坏 `JPEG` 或 `EXE` 等二进制文件中的数据。注意，在读写此类文件时，一定要使用二进制模式。

在处理文件对象时，最好使用 `with` 关键字。优点是，子句体结束后，文件会正确关闭，即便触发异常也可以。而且，使用 `with` 相比等效的 `try`-`finally` 代码块要简短得多

如果没有使用 `with` 关键字，则应调用 `f.close()` 关闭文件，即可释放文件占用的系统资源。

> 调用 `f.write()` 时，未使用 `with` 关键字，或未调用 `f.close()`，即使程序正常退出，也**可能**导致 `f.write()` 的参数没有完全写入磁盘。

7.2.1 文件对象的方法

`f.read(size)` 可用于读取文件内容，它会读取一些数据，并返回字符串（文本模式），或字节串对象（在二进制模式下）。 *size* 是可选的数值参数。省略 *size* 或 *size* 为负数时，读取并返回整个文件的内容；

文件大小是内存的两倍时，会出现问题。*size* 取其他值时，读取并返回最多 *size* 个字符（文本模式）或 *size* 个字节（二进制模式）。如已到达文件末尾，`f.read()` 返回空字符串（`''`）。

```
f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` 从文件中读取单行数据；字符串末尾保留换行符（`\n`），只有在文件不以换行符结尾时，文件的最后一行才会省略换行符。这种方式让返回值清晰明确；只要 `f.readline()` 返回空字符串，就表示已经到达了文件末尾，空行使用 `'\n'` 表示，该字符串只包含一个换行符。

```
f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

从文件中读取多行时，可以用循环遍历整个文件对象。这种操作能高效利用内存，快速，且代码简单：

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

`f.write(string)` 把 *string* 的内容写入文件，并返回写入的字符数。

```
>>> f.write('This is a test\n')
15
```

写入其他类型的对象前，要先把它们转化为字符串（文本模式）或字节对象（二进制模式）：

```
>>> value = ('the answer', 42)
>>> s = str(value)  # 将元组转换为字符串
>>> f.write(s)
18
```

`f.tell()` 返回整数，给出文件对象在文件中的当前位置，表示为二进制模式下时从文件开始的字节数，以及文本模式下的意义不明的数字。

`f.seek(offset, whence)` 可以改变文件对象的位置。通过向参考点添加 *offset* 计算位置；参考点由 *whence* 参数指定。 *whence* 值为 0 时，表示从文件开头计算，1 表示使用当前文件位置，2 表示使用文件末尾作为参考点。省略 *whence* 时，其默认值为 0，即使用文件开头作为参考点。

```
f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # 定位到文件中的第 6 个字节
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # 定位到倒数第 3 个字节
13
>>> f.read(1)
b'd'
```

在文本文件（模式字符串未使用 `b` 时打开的文件）中，只允许相对于文件开头搜索（使用 `seek(0, 2)` 搜索到文件末尾是个例外），唯一有效的 *offset* 值是能从 `f.tell()` 中返回的，或 0。其他 *offset* 值都会产生未定义的行为。

7.2.2 使用json保存结构化数据

字符串可以很容易地写入文件或从文件中读取。 数字则更麻烦一些，因为 `read()` 方法只返回字符串

标准库模块 `json` 可以接受带有层级结构的 Python 数据，并将其转换为字符串表示形式；这个过程称为 *serializing*。 根据字符串表示形式重建数据则称为 *deserializing*。 在序列化和反序列化之间，用于代表对象的字符串可以存储在文件或数据库中，或者通过网络连接发送到远端主机。

```
import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

`dumps()` 函数还有一个变体， `dump()` ，它只将对象序列化为 text file 。因此，如果 `f` 是 text file 对象，可以这样做：

```
json.dump(x, f)
```

要再次解码对象，如果 `f` 是已打开、供读取的 binary file 或 text file 对象：

```
x = json.load(f)
```

JSON文件必须以UTF-8编码。当打开JSON文件作为一个 text file 用于读写时，使用 `encoding="utf-8"`

> json.dumps(dict, indent)：将Python对象转换成json字符串
> json.dump(dict, file_pointer)：将Python对象写入json文件

8.错误和异常

错误可以分为两种：语法错误和异常

8.3 异常的处理

可以编写程序处理特定的异常

try语句的工作原理：

- 首先，执行 *try 子句* （ `try` 和 `except` 关键字之间的（多行）语句）。
- 如果没有触发异常，则跳过 *except 子句*， `try` 语句执行完毕。
- 如果在执行 `try` 子句时发生了异常，则跳过该子句中剩下的部分。 如果异常的类型与 `except` 关键字后指定的异常相匹配，则会执行 *except 子句*，然后跳到 try/except 代码块之后继续执行。
- 如果发生的异常与 *except 子句* 中指定的异常不匹配，则它会被传递到外层的 `try` 语句中；如果没有找到处理器，则它是一个 *未处理异常* 且执行将停止并输出一条错误消息。

一个try语句可以有多个except子句来位不同异常处理程序，一个except语句也可以捕获多个异常。

一个except子句中的咧匹配的异常时该类本身的实例或其所派生的类的实例

发生异常时，它可能具有关联值，即异常 *参数* 。是否需要参数，以及参数的类型取决于异常的类型。

except 子句 可能会在异常名称后面指定一个变量。 这个变量将被绑定到异常实例，该实例通常会有一个存储参数的 `args` 属性。 为了方便起见，内置异常类型定义了 `__str__()` 来打印所有参数而不必显式地访问 `.args`。

>>>

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # 异常的类型
...     print(inst.args)     # 参数保存在 .args 中
...     print(inst)          # __str__ 允许 args 被直接打印，
...                          # 但可能在异常子类中被覆盖
...     x, y = inst.args     # 解包 args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

`BaseException` 是所有异常的共同基类。它的一个子类， `Exception` ，是所有非致命异常的基类。不是 `Exception` 的子类的异常通常不被处理，因为它们被用来指示程序应该终止。它们包括由 `sys.exit()` 引发的 `SystemExit` ，以及当用户希望中断程序时引发的 `KeyboardInterrupt` 。

`try` ... `except` 语句具有可选的 *else 子句*，该子句如果存在，它必须放在所有 *except 子句* 之后。 它适用于 *try 子句* 没有引发异常但又必须要执行的代码。

8.4 触发异常

raise语句支持强制触发指定的异常

raise唯一的参数就是要触发的异常。这个参数必须时异常实例或异常类（如果传递的是异常类，讲通过调用没有参数的构造函数来隐式实例化。

如果知识像判断是否触发异常，但不打算处理该异常，可以使用简单的raise语句重新触发异常

8.5 异常链

如果一个未处理的异常发生在except部分内，他将会有背处理的异常附加到他上面，并包括在错误信息中

```
try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
```

```
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    open("database.sqlite")
    ~~~~^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

它还允许使用 `from None` 表达禁用自动异常链:

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError from None
RuntimeError
```

为了表明一个异常是另一个异常的直接后果， `raise` 语句允许一个可选的 `from` 子句:

```
# exc 必须为异常实例或为 None。
raise RuntimeError from exc
```

转换异常时，这种方式很有用。例如:

>>>

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    func()
    ~~~~^^
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
```

```
    raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

8.6 用户自定义异常

程序可以通过创建新的异常类命名自己的异常

异常都是从Exception类派生的

8.7 定义清理操作

try 语句还有一个可选子句，用户定义在所有情况下都必须执行的清理操作

```
try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt
```

存在finally子句，finally子句是try语句结束前执行的最后一项任务。无论是否触发异常

- 如果执行 `try` 子句期间触发了某个异常，则某个 `except` 子句应处理该异常。如果该异常没有 `except` 子句处理，在 `finally` 子句执行后会被重新触发。
- `except` 或 `else` 子句执行期间也会触发异常。 同样，该异常会在 `finally` 子句执行之后被重新触发。
- 如果 `finally` 子句中包含 `break`、`continue` 或 `return` 等语句，异常将不会被重新引发。
- 如果执行 `try` 语句时遇到 `break`、`continue` 或 `return` 语句，则 `finally` 子句在执行 `break`、`continue` 或 `return` 语句之前执行。
- 如果 `finally` 子句中包含 `return` 语句，则返回值来自 `finally` 子句的某个 `return` 语句的返回值，而不是来自 `try` 子句的 `return` 语句的返回值。

下面是例子展示

```
def divide(x, y):
...     try:
...         result = x / y
```

```
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    divide("2", "1")
    ~~~~~~^^^^^^^^^^
  File "<stdin>", line 3, in divide
    result = x / y
             ~~^~~
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

如上述所述，任何情况下都会执行finally子句

实际应用程序中，finally子句对于释放外部资源（例如文件或网络连接）非常有用

8.8 预定义的清理操作

with语句

8.9 引发和处理多个不相关的异常

内置的ExceptionGroup打包了一个异常实例的列表，这样可以触发多个异常，同时它本身就是一个异常，所以可以被捕获

```
def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
>>> f()
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 3, in f
  |     raise ExceptionGroup('there were problems', excs)
  | ExceptionGroup: there were problems (2 sub-exceptions)
```

```
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
    +------------------------------------
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>
```

通过使用 `except*` 代替 `except`，我们可以有选择地只处理组中符合某种类型的异常。在下面的例子中，显示了一个嵌套的异常组，每个 `except*` 子句都从组中提取了某种类型的异常，而让所有其他的异常传播到其他子句，并最终被重新引发。

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 2, in f
  |     raise ExceptionGroup(
  |     ...<12 lines>...
  |     )
  | ExceptionGroup: group1 (1 sub-exception)
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2 (1 sub-exception)
```

```
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
```

注意，嵌套在一个异常组中的异常必须是实例，而不是类型。

8.10 用注释细化异常情况

有时候，在异常捕获的适合添加信息是很有用的。为了实现这个目的。异常有个add_note(note)的方法接受一个字符串，并将其添加到异常的注释列表。

```
try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise TypeError('bad type')
TypeError: bad type
Add some information
Add some more information
>>>
```

9.类（*）

9.1 名称和对象

对象之间相互独立，多个名称可以绑定到同意对象

9.2 Python作用域和命名空间

namespace（命名空间）是从名称到对象的映射

不同命名空间中的名称之间绝对没有关系

点好知乎的名称是**属性**，modname.funcname。modname是模块对象，funcname是模块属性。模块属性和模块中定义的全局名称之间存在直接的映射：他们共享相同的命名空间

命名空间是不同时刻创建的，且拥有不同的生命周期。

- **内置名称的命名空间**是python解释器启动时创建的，永远不会删除。
- 模块的**全局命名空间**在读取模块定义时创建，并且通常会持续到解释器退出。

- 函数的**局部命名空间**在函数被调用时被创建，并在函数返回或抛出未处理的异常时被删除。

作用域的分类有四种：

1. L 局部作用域
2. E 闭包函数外的函数作用域
3. G 全局作用域
4. B 内建作用域

查询顺序是L > E > G > B

注意区别：命名空间和作用域

> ## 命名空间比作“储物柜”：
>
> 想象你有一排储物柜，每个柜子里都存放着一些东西（例如，衣服、书、玩具等）。在编程中，**命名空间** 就是这些“储物柜”，它们用来存放变量、函数和其他对象。每个命名空间都存放着一个名字和对应的对象（比如变量的值）。
>
> - **全局命名空间**：就像一个大的储物柜，里面可以存放整个程序的公共变量或函数。
> - **局部命名空间**：是一个小储物柜，它只存在于函数内部，用来存放该函数内的变量。
>
> ## 作用域比作“访问范围”：
>
> 作用域就像你在一个商店里“能去的地方”。比如，你在商店里有几个区域：某些区域你可以随便进入（比如公共区域），有些区域你只能在某些条件下进入（比如试衣间或仓库）。
>
> - **全局作用域**：就像商店的公共区域，你可以随时进出，在整个程序中都能访问。
> - **局部作用域**：就像试衣间，你只能在特定的范围内使用，只有在函数内部，你才能访问到函数内的“储物柜”。
> - **封闭作用域**：就像商店的某个中间区域，你可以访问，但必须通过其他区域才能进入。
>
> ## 结合这个比喻来理解：
>
> - **命名空间**：储物柜，存放具体的物品（变量、函数）。
> - **作用域**：访问范围，决定你能进入哪个储物柜并获取其中的物品。

Python 有一个特殊规定。如果不存在生效的 `global` 或 `nonlocal` 语句，则对名称的赋值总是会进入最内层作用域。赋值不会复制数据，只是将名称绑定到对象。删除也是如此：语句 `del x` 从局部作用域引用的命名空间中移除对 `x` 的绑定。

9.3 初探类 (*)

9.3.1 类定义语法

最简单的类定义形式

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

当进入类定义时，将创建一个新的命名空间，并将其用作局部作用域 --- 因此，所有对局部变量的赋值都是在这个新命名空间之内。 特别的，函数定义会绑定到这里的新函数名称。

当 (从结尾处) 正常离开类定义时，将创建一个 *类对象*。 这基本上是一个围绕类定义所创建的命名空间的包装器；我们将在下一节中了解有关类对象的更多信息。 原始的 (在进入类定义之前有效的) 作用域将重新生效，类对象将在这里与类定义头所给出的类名称进行绑定 (在这个示例中为 `ClassName`)。

9.3.2 Class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 Python 中所有属性引用所使用的标准语法: `obj.name`。

类的 *实例化* 使用函数表示法。 可以把类对象视为是返回该类的一个新实例的不带参数的函数。

> ### 1. 什么是 Python 类对象?
>
> **类对象**是指**类本身**，即类的定义，它是 Python 中的一个对象，属于元类（metaclass）。类对象本身是由元类（如 `type`）创建的。在 Python 中，类定义时会创建一个类对象，这个类对象可以用来实例化对象。
>
> 换句话说，类对象就是你定义的类的“蓝图”或模板，是创建实例（对象）的基础。
>
> **示例:**
>
> ```
> class MyClass:
>  pass
>
> # MyClass 是一个类对象
> print(MyClass)  # <class '__main__.MyClass'>
> ```
>
> 在上面的例子中，`MyClass` 是一个类对象，它本身也是一个对象。我们通过它来创建实例。

2. 什么是类的实例化?

类的实例化是指**通过类对象创建具体的实例对象**。实例化是通过调用类的构造方法（即 `__init__` 方法）来创建一个对象。

- 类是一个模板或蓝图，而实例是类的实际化。实例拥有类中定义的属性和方法，可以进行操作和修改。
- 类的实例化即创建对象的过程，也就是说，你通过类对象创建了该类的一个具体实例对象。

示例:

```
class MyClass:
    def __init__(self, name):
        self.name = name

# 实例化 MyClass，创建一个对象
obj = MyClass("Python")

print(obj)        # <__main__.MyClass object at 0x7fc944c9f250>
print(obj.name)   # Python
```

在这个例子中，`obj` 是通过 `MyClass` 类实例化出来的对象。类定义了一个构造方法 `__init__`，并且 `obj` 是该类的一个实例，拥有 `name` 属性。

3. 类的实例化和类对象有什么关系?

- **类对象**是类本身，它定义了如何创建实例。
- **类的实例化**是通过类对象来创建具体的实例对象。实例对象拥有类定义的属性和方法。

在实例化的过程中，Python 会根据类对象来创建新的实例（对象），每个实例是类的一个独立副本。通过类对象，你可以创建多个实例，每个实例的状态是独立的，具有自己的属性值。

关系总结:

- **类对象**是类的定义本身，是用来生成实例的蓝图。
- **类的实例化**是创建实例对象的过程，通过类对象来生成实例对象。

类对象是静态的，它定义了类的结构和行为；而**实例化**是动态的，通过类对象生成的实例是实际的对象，存储数据并能调用类的方法。

4. 更直观的比喻:

- **类对象**：就像是制造手机的工厂的设计图，设计图定义了手机的外观和功能。
- **实例化**：是通过这个设计图建造实际的手机。每次建造出来的手机都是根据设计图来做的，但每个手机是独一无二的，具有独立的状态（例如颜色、型号等）。

希望这些解释能够帮助你理解类对象和实例化之间的关系!

当一个类定义了 `__ init __()` 方法时，类的实例化会自动为新创建的类实例唤起 `__init__()`

9.3.3 实例对象

9.3.4 方法对象

方法的特殊之处就在于实例对象会作为函数的第一个参数被传入。 在我们的示例中，调用 `x.f()` 其实就相当于 `MyClass.f(x)`。

> 注意区别方法对象和函数对象
>
> def func():{}　　func()
>
> def func2(self):{}　x.func2()

方法的运作方式如下。 当一个实例的非数据属性被引用时，将搜索该实例所属的类。 如果名称表示一个属于函数对象的有效类属性，则指向实例对象和函数对象的引用将被打包为一个方法对象。 当传入一个参数列表调用该方法对象时，将基于实例对象和参数列表构造一个新的参数列表，并传入这个新参数列表调用相应的函数对象。

9.3.5 类和实例变量

通常，实例变量时用于每个实例的唯一数据，而类变量用于类的所有实例共享的属性和方法

```
class Dog:

    kind = 'canine'         # 类变量被所有实例所共享

    def __init__(self, name):
        self.name = name    # 实例变量为每个实例所独有

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # 被所有的 Dog 实例所共享
'canine'
>>> e.kind                  # 被所有的 Dog 实例所共享
'canine'
>>> d.name                  # 为 d 所独有
'Fido'
>>> e.name                  # 为 e 所独有
'Buddy'
```

值得注意的是，共享数据在涉及可变对象（例如列表）时可能导致出人意料的结果（通常为错误）

```
class Dog:

    tricks = []             # 类变量的错误用法

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
```

```
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # 非预期地被所有的 Dog 实例所共享
['roll over', 'play dead']
```

正确的类设计应该使用实例变量:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # 为每个 Dog 实例新建一个空列表

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 补充说明

如果同样的属性名称同时出现在实例和类中，则属性查找会优先选择实例。

数据属性可以被方法以及一个对象的普通用户（“客户端”）所引用。 换句话说，类不能用于实现纯抽象数据类型。 实际上，在 Python 中没有任何东西能强制隐藏数据 --- 它是完全基于约定的。 （而在另一方面，用 C 语言编写的 Python 实现则可以完全隐藏实现细节，并在必要时控制对象的访问；此特性可以通过用 C 编写 Python 扩展来使用。）

方法的第一个参数常常被命名为 `self`。 这也不过就是一个约定: `self` 这一名称在 Python 中绝对没有特殊含义。

9.5 继承（*）

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

python有两个内置函数可被用于继承机制

- 使用 `isinstance()` 来检查一个实例的类型: `isinstance(obj, int)` 仅会在 `obj.__class__` 为 `int` 或某个派生自 `int` 的类时为 `True`。
- 使用 `issubclass()` 来检查类的继承关系: `issubclass(bool, int)` 为 `True`，因为 `bool` 是 `int` 的子类。 但是，`issubclass(float, int)` 为 `False`，因为 `float` 不是 `int` 的子类。

9.5.1 多重继承

```
class DerivedClassName(Base1, Base2, Base3):
    <语句-1>
    .
    .
    .
    <语句-N>
```

简单：可以认为搜索从父类所继承属性的操作是深度优先、从左到右的。

真实情况：方法解析顺序会动态改变以支持对 `super()` 的协同调用。

9.6 私有变量

python实际上不存在那种通常意义的私有变量（比如c++），但是python可以使用下划线标识该变量是私有变量（这是一种约定）

9.7 杂项说明

python虽然不用声明数据类型，但是如果需要类似c语言的数据类型说明，可以引入dataclasses模块

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

9.8 迭代器

大多数容器对象可以使用for语句

for语句会在容器对象上调用iter()，该函数返回一个定义了 `__next__()` 方法的迭代器对象。该方法讲注意访问容器中的元素。当元素用尽时，`__next__()` 将引发StopIteration异常来通知结束for循环，此外除了调用 `__next__()` 方法来迭代，还可以通过next函数来调用迭代器的 `__next__()` 方法实现

```
s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

综上所述

for循环 s

- 在迭代器上调用iter函数 iter(s)
- 迭代器s 调用 `__iter__()` 方法，返回迭代器对象，通常是自身。即s
- 在迭代器上调用next函数next(s)
- 迭代器s调用 `__next__()` 方法
    - 迭代元素
    - 抛出StopIteration异常被for循环捕获

因此，通过上述描述一个合理的迭代器需要有 `__iter__` 和 `__next__` 两个方法

```
class Reverse:
```

```
    """对一个序列执行反向循环的迭代器。"""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

上述是一个自定义类的反向迭代器

9.9 生成器（*）

生成器是一个用于创建迭代器的工具

用法：在定义的标准函数中的返回值使用yield

结果：每次在生成器中调用next()时，代码就会从上次离开的位置恢复执行

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

生成器是一种迭代器，但它们的实现方式不同，生成器通过 `yield` 自动提供迭代器功能，而不需要显式地实现 `__iter__()` 和 `__next__()` 方法。

> 当你定义一个普通的函数并在其中使用 `yield` 语句时，Python 会自动将这个函数转换成一个 **生成器函数**。生成器函数与普通函数不同，普通函数执行时会一次性返回一个值并结束，而生成器函数在执行时会在遇到 `yield` 语句时暂停，并返回 `yield` 后面的值，同时保存当前的执行状态。
>
> 当你调用生成器函数时，它并不会立即执行，而是返回一个生成器对象。这个生成器对象符合迭代器协议（实现了 `__iter__()` 和 `__next__()` 方法），因此可以像迭代器一样使用。

生成器对象是如何创建的?

当你调用 `reverse('golf')` 时，Python 会返回一个 **生成器对象**，这个对象是一个实现了 `__iter__()` 和 `__next__()` 方法的迭代器。你可以理解为，**生成器对象是 `reverse('golf')` 这个调用的结果。**

换句话说，`reverse('golf')` 返回的生成器对象是一个迭代器对象，它在每次迭代时会调用 `__next__()`，并从 `yield` 语句处恢复执行。

生成器对象与 `data` 的关系

生成器对象与 `data` 的关系是：**生成器对象是由生成器函数创建的，而数据（例如 `'golf'`）是传递给生成器函数的输入。** 生成器对象的内部状态会跟踪迭代的数据元素。

具体而言，生成器对象并不直接等于 `data`，而是它的封装体，用来迭代 `data` 中的元素。在你的例子中，生成器对象是 `reverse('golf')` 返回的对象，它负责逐步生成 `'g'`, `'o'`, `'l'`, `'f'` 等值。

9.10 生成器表达式

生成器表达式类似列表推掉是，但外层不是方括号而是圆括号

生成器表达式相比完整的生成器更紧凑但较不灵活，相比等效的列表推导式则更为节省内存。

```
sum(i*i for i in range(10))                 # 平方和
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # 点乘
260

>>> unique_words = set(word for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

10.标准库简介

10.1 操作系统接口

os模块提供了许多与操作系统交互的函数

os.

- getcwd() 返回当前工作目录
- chdir()　改变当前工作目录
- system('mkdir today') 在系统shell中与运行mkdir命令

一定要使用 `import os` 而不是 `from os import *` 。这将避免内建的 `open()` 函数被 `os.open()` 隐式替换掉，因为它们的使用方式大不相同。

对于日常文件和目录管理任务，shutil模块提供了更易于使用的更高级别的接口

```
import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2 文件通配符

glob模块提供了一个在目录中使用通配符搜索创建文件列表的函数

```
import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3命令行参数

需要导入sys模块

10.4 错误输出重定向和程序终止

sys模块中的属性

- stdin

- stdout
- stderr

10.5 字符串模式匹配

re模块-正则表达式工具

10.6 数学

math模块提供对用于浮点数学运算的下层C库函数的访问

random模块提供了进行随机选择的工具

statistics模块计算数据的基本统计属性

10.7 互联网访问

许多模块可用于访问互联网和处理互联网协议。最简单的两个

- urllib.request 用于从URL检索数据
- smtplib 用于发送邮件

10.8 日期和时间

`datetime` 模块提供了以简单和复杂的方式操作日期和时间的类。

10.9 数据压缩

常见的数据存档和压缩格式由模块直接支持，包括：`zlib`，`gzip`，`bz2`，`lzma`，`zipfile` 和 `tarfile`。：

>>>

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 性能测量

有些用户对同一问题的不同方法的相对性能有兴趣。

python提供了一种可以立刻回答这些问题的测量工具

例如，元组封包和拆包功能相比传统的交换参数可能更具吸引力。`timeit` 模块可以快速演示在运行效率方面一定的优势:

>>>

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

与 `timeit` 的精细粒度级别相反， `profile` 和 `pstats` 模块提供了用于在较大的代码块中识别时间关键部分的工具。

10.11 质量控制

doctest模块提供了工具，用于扫描模块并验证文档字符串中嵌入的测试。

```
def average(values):
    """计算数字列表的算术平均值

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # 自动验证嵌入式测试
```

`unittest` 模块不像 `doctest` 模块那样易于使用，但它允许在一个单独的文件中维护更全面的测试集:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main()  # 从命令行调用时会执行所有测试
```

10.12 自带电池

“Python 自带电池”（Batteries Included）是 Python 语言的一个核心理念，指其**标准库（Standard Library）功能丰富且全面**，无需额外安装第三方库即可完成许多常见任务。这一说法源自 Python 官方对“开箱即用”体验的重视。

11.标准库二

11.1 格式化输出

reprlib模块提供了一个定制化版本的repr()函数

pprint模块提供了更加复杂的打印控制

textwrap模块能够格式化文本段落，以适应给定的屏幕木宽度

locale模块处理与特定地域文化相关的数据结构

11.2 模板

string模板包含一个通用的Template类，具有适用于最终用户的简化语法。它允许用户在不更改应用逻辑的情况下定制自己的应用。

```
from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

格式化操作通过占位符实现，占位符由 `$` 加上合法的 Python 标识符（只能包含字母、数字和下划线）构成。

`$$` 将被转义成单个字符 `$`

如果占位符没有提供值，substitute方法将会抛出异常，可以使用safe_substitute方法可以将没有提供值的占位符保留

```
t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template可以自定义占位符

```
class BatchRename(Template):
...     delimiter = '%'
...
```

11.3 使用二进制数据记录格式

struck模块提供了pack和unpack函数，用于处理必定长度的二进制记录格式

11.4 多线程（*）

多线程是一种对于非顺序依赖的多个任务进行解耦的技术。

多线程可以提高应用的响应速度

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # 等待背景任务结束
print('Main program waited until background was done.')
```

多线程应用面临的主要挑战是，相互协调的多个线程之间需要共享数据或其他资源。为此，threading 模块提供了多个同步操作原语，包括线程锁、事件、条件变量和信号量。

当然！下面是一个示例，展示了不使用集中请求设计和使用集中请求设计的区别。

不使用集中请求设计

在这个示例中，多个线程直接访问共享资源（在本例中是一个计数器）。没有适当的同步机制可能会导致竞态条件。

```
import threading

# 共享资源
counter = 0

# 线程执行的任务
def increment():
    global counter
    for _ in range(100000):
        counter += 1

# 创建多个线程并启动
threads = []
for _ in range(5):
    thread = threading.Thread(target=increment)
    threads.append(thread)
    thread.start()
```

```
# 等待所有线程完成
for thread in threads:
    thread.join()

print(f"Final counter value: {counter}")
```

问题：
在这个代码中，多个线程同时访问 `counter` 这个共享资源。由于没有任何同步机制，`counter += 1` 这个操作不是原子的，它可能会发生竞态条件。最终的计数器值可能不是预期的 `500000`，因为多个线程可能同时读取和修改 `counter` 的值，导致丢失更新。

使用集中请求设计

通过使用 `queue.Queue` 和一个专门的线程来处理所有的资源请求，我们可以避免上面的竞态条件。

```
import threading
import queue

# 共享资源
counter = 0

# 队列，用于线程间通信
request_queue = queue.Queue()

# 专门的线程来处理资源请求
def handle_requests():
    global counter
    while True:
        # 获取任务（如果队列为空则阻塞）
        task = request_queue.get()
        if task == "STOP":
            break  # 结束线程
        counter += 1
        request_queue.task_done()

# 创建并启动处理线程
handler_thread = threading.Thread(target=handle_requests)
handler_thread.start()

# 创建多个线程向队列发送请求
threads = []
for _ in range(5):
    def increment():
        for _ in range(100000):
            request_queue.put("INCREMENT")

    thread = threading.Thread(target=increment)
    threads.append(thread)
    thread.start()

# 等待所有线程完成
for thread in threads:
    thread.join()
```

```
# 发送结束信号，通知处理线程结束
request_queue.put("STOP")
handler_thread.join()

print(f"Final counter value: {counter}")
```

优点：

1. 所有的 `counter` 更新请求都被发送到 `request_queue` 中，由单独的 `handle_requests` 线程来处理。
2. `handle_requests` 线程是唯一一个直接访问 `counter` 资源的线程，因此避免了并发访问冲突。
3. `queue.Queue` 自动处理线程间通信的同步，因此无需手动管理锁。

通过这种方式，程序变得更容易理解和维护，并且避免了竞态条件和并发问题。

> ## 为什么选择集中资源访问的方式？
>
> 1. **减少复杂性和同步错误**： 使用同步机制时，如果不小心，可能会引入死锁、竞态条件等问题，特别是在复杂的应用中。线程锁等同步原语虽然强大，但它们的使用需要非常小心，一些微小的设计错误可能导致程序无法预测地崩溃或死锁。
>
>   集中资源访问的设计通过队列让所有线程的请求都通过一个处理线程，这样就避免了多线程直接访问共享资源的复杂性。这种方法虽然牺牲了一些性能（因为所有请求都必须通过一个线程），但可以极大地降低错误的可能性，特别是在资源访问冲突的情况下。
> 2. **避免过度使用同步原语**： 如果每个线程都需要直接操作共享资源并且需要同步机制来保证数据一致性，整个系统可能需要大量的同步控制代码，这会导致代码变得冗长且难以维护。集中式设计通过队列简化了线程之间的协调逻辑，使得各个线程只负责自己的任务，而不需要涉及资源管理或同步问题。
> 3. **简化调试和维护**： 采用集中的方式使得资源访问集中在一个线程上，这样如果程序出现错误，开发者可以更容易地找到问题的根源，因为访问资源的逻辑都集中在一个地方。使用同步机制时，线程的并发执行会让问题变得更加难以复现和调试。
> 4. **性能权衡**： 采用集中式设计会带来性能上的损失，因为所有资源访问都必须经过队列的传递，但这种性能损失对于大多数应用而言是可以接受的。特别是在一些 IO 密集型的任务中（比如网络请求、文件处理等），与计算密集型任务相比，这种性能损失是可以忽略不计的。
>
> ## 同步机制的选择和应用
>
> 同步机制（如锁、条件变量等）仍然是处理多线程共享资源时的一种常用且有效的方法，尤其在以下场景中：
>
> - **性能要求较高**：如果应用需要最大化性能，集中资源访问可能导致瓶颈，而通过适当的同步机制（如锁）来控制对共享资源的访问，可以在多线程环境中更好地平衡性能和安全性。
> - **粒度控制更细**：如果多个线程对不同资源的访问需要更加灵活和细粒度的控制，使用同步机制可以更好地定制化行为。
>
> ## 总结
>
> 虽然同步机制是解决多线程竞争问题的常见方法，Python选择集中资源访问的方式是为了减少多线程并发带来的复杂性，避免频繁使用同步原语造成的设计难度和调试问题。在某些场景下，这种设计可以显著提高代码的可读性、可维护性和可靠性，尤其在资源访问较少竞争的情况下，集中式设计非常适合。然而，对于性能要求极高的场景，适当使用同步原语可能是更合适的选择。

11.5 日志记录

logging模块提供功能齐全且灵活的日志记录系统

11.6 弱引用

Python 会自动进行内存管理（对大多数对象进行引用计数并使用 garbage collection 来清除循环引用）。 当某个对象的最后一个引用被移除后不久就会释放其所占用的内存。

在 Python 中，**弱引用**（weak reference）是指对对象的引用，但不增加该对象的引用计数。也就是说，弱引用不会阻止垃圾回收器回收对象。这与普通引用不同，普通引用会增加对象的引用计数，直到引用计数变为 0 时对象才会被垃圾回收。

为什么需要弱引用?

通常，Python 中的垃圾回收是基于引用计数的，当一个对象的引用计数为 0 时，它会被回收。然而，有时我们希望在对象没有其他强引用的情况下依然允许它被回收，而又希望能够访问它。这时候，弱引用就派上用场了。

弱引用的使用场景

弱引用通常用于缓存、观察者模式等场景。例如，当你希望缓存某个对象，但不希望缓存阻止对象被回收时，可以使用弱引用。

Python 中的弱引用

在 Python 中，弱引用是通过 `weakref` 模块实现的。 `weakref` 提供了创建弱引用的功能，避免了常规引用对对象生命周期的干扰。

基本使用

创建弱引用

你可以通过 `weakref.ref()` 创建一个弱引用对象，这个弱引用对象在不再有强引用的情况下会被垃圾回收。

```
import weakref

class MyClass:
    def __init__(self, name):
        self.name = name

obj = MyClass("Object1")
weak_obj = weakref.ref(obj)
```

```
print(weak_obj)  # 打印弱引用对象
print(weak_obj())  # 打印引用的对象（可以通过调用弱引用对象来访问原对象）

del obj  # 删除原对象
print(weak_obj())  # 删除原对象后，弱引用返回 None
```

解释：

- `weakref.ref(obj)` 创建了一个 `obj` 的弱引用 `weak_obj`。
- 通过 `weak_obj()` 可以访问原对象，但如果原对象被垃圾回收，`weak_obj()` 将返回 `None`。
- 删除 `obj` 后，原对象的引用计数为 0，对象被垃圾回收，因此 `weak_obj()` 返回 `None`。

使用 `weakref` 与容器

`weakref` 也可以用于创建弱引用的容器。比如，`weakref.WeakValueDictionary` 是一个字典，它允许存储弱引用的对象值，且不会增加这些对象的引用计数。

```
import weakref

class MyClass:
    def __init__(self, name):
        self.name = name

d = weakref.WeakValueDictionary()

obj1 = MyClass("Object1")
obj2 = MyClass("Object2")

d["obj1"] = obj1
d["obj2"] = obj2

print(d["obj1"].name)  # 输出 Object1

del obj1  # 删除 obj1，obj1 会被垃圾回收
print(d.get("obj1"))  # 输出 None，因为 obj1 被回收了
```

解释：

- `WeakValueDictionary` 是一个字典容器，它存储对象的弱引用。如果对象被回收，这些条目会自动被删除。
- 删除 `obj1` 后，`d["obj1"]` 不再有效，返回 `None`。

弱引用的优点

1. **避免内存泄漏**：在某些情况下，我们希望对象在没有强引用时可以被垃圾回收。使用弱引用可以避免强引用导致的内存泄漏。
2. **节省内存**：弱引用不会增加对象的引用计数，可以用于缓存或映射，避免过多的内存占用。

弱引用的局限性

1. **对象生命周期依赖于强引用**：弱引用不会增加对象的引用计数，只有当对象没有任何强引用时，它才会被垃圾回收。如果你需要保持对对象的访问，必须确保至少有一个强引用存在。
2. **无法修改对象**：弱引用对象本身无法像普通引用那样修改对象的状态，它只能访问原对象。

总结

- **弱引用** 是一种不增加对象引用计数的引用方式，可以在对象没有强引用时允许其被垃圾回收。
- Python 中的弱引用由 `weakref` 模块提供，常用于缓存、映射等需要控制对象生命周期的场景。

11.7 用于操作列表的工具

array模块

bisect模块

collection模块中的deque对象

```
from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

heapq模块

```
from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # 将列表重新调整为堆顺序
>>> heappush(data, -5)                 # 添加一个新条目
>>> [heappop(data) for i in range(3)]  # 获取三个最小的条目
[-5, 0, 1]
```

heapq 模块提供了基于常规列表来实现堆的函数。 最小值的条目总是保持在位置零。 这对于需要重复访问最小元素而不希望运行完整列表排序的应用来说非常有用

11.8 十进制浮点运算

decimal模块

12.虚拟环境和包

12.1 概述

这意味着一个Python安装可能无法满足每个应用程序的要求。如果应用程序A需要特定模块的1.0版本但应用程序B需要2.0版本，则需求存在冲突，安装版本1.0或2.0将导致某一个应用程序无法运行。

这个问题的解决方案是创建一个 virtual environment，一个目录树，其中安装有特定Python版本，以及许多其他包。

12.2 创建虚拟环境

用于创建和管理虚拟环境的模块是 `venv`。 `venv` 将安装运行命令所使用的 Python 版本（即 `--version` 选项所报告的版本）。 例如，使用 `python3.12` 执行命令将会安装 3.12 版。

要创建虚拟环境，请确定要放置它的目录，并将 `venv` 模块作为脚本运行目录路径:

```
python -m venv tutorial-env
```

这将创建 `tutorial-env` 目录，如果它不存在的话，并在其中创建包含 Python 解释器副本和各种支持文件的目录。

虚拟环境的常用目录位置是 `.venv`。 这个名称通常会令该目录在你的终端中保持隐藏，从而避免需要对所在目录进行额外解释的一般名称。 它还能防止与某些工具所支持的 `.env` 环境变量定义文件发生冲突。

创建虚拟环境后，您可以激活它。

在Windows上，运行:

```
tutorial-env\Scripts\activate
```

在Unix或MacOS上，运行:

```
source tutorial-env/bin/activate
```

（这个脚本是为bash shell编写的。如果你使用 **csh** 或 **fish** shell，你应该改用 `activate.csh` 或 `activate.fish` 脚本。）

激活虚拟环境将改变你所用终端的提示符，以显示你正在使用的虚拟环境，并修改环境以使 `python` 命令所运行的将是已安装的特定 Python 版本。 例如:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May  6 2016, 10:59:36)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
'~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

要撤销激活一个虚拟环境，请输入:

```
deactivate
```

到终端。