1.模块与包

为什么要学习“模块和包”?

- 因为随着代码量的增加，我们需要更有逻辑地组织代码，以及代码文件之间的关系。
- 让代码更容易被定位、归类和引用。
- 让我们可以重复使用自己的和他人的代码，可以多人协作更大型的项目。

1.1 模块 （Module）

1.1.1 模块的概念

Python 模块(Module)，是一个 Python 文件，以 .py 结尾，包含了 Python 对象定义和Python语句。

对于Python中包、模块、函数或者类的理解，可以参照下表：

概念	状态
包	文件夹
模块	文件
函数或者类	文件中的内容

1.1.2 模块的导入

模块导入，相当于把别人的代码拿到自己的文件中使用。

语法：*import 模块名*

（1）导入系统的内置模块

例如:math是python语言中内置的模块，提供了很多数学中的常量与操作。常量math.pi表示圆周率Π;
math.pow(2,3)表示2的3次方。

要引用模块 math，就可以在文件最开始的地方用 **import math** 来引入:

```
import math
print(math.pi)          #输出：3.141592653589793
print(type(math))       #输出：<class 'module'>
```

> 在模块名上 按住“Ctrl + 鼠标左键”，可以打开并跳转到引入的模块源文件中。

（2）导入自定义模块

定义一个模块，名为myModule1.py，其中代码为：

```
def add(a,b):
    print("mymodule1:add....")
    return f"{a}+{b}={a+b}"

def divid(a,b):
    print("mymodule1:divid....")
    return f"{a}/{b}={a/b}"

def multiply(a,b):
    print("mymodule1:multiply....")
    return f"{a}*{b}={a*b}"
```

引用自定义的myModule1模块，并调用其中的函数。

调用模块中的函数，语法：*模块名.函数名([实际参数])*

```
import myModule1

print(myModule1.add(3,2))       #输出：mymodule1:add....
                                #      3+2=5
print(myModule1.divid(3,2))     #输出：mymodule1:divid....
                                #      3/2=1.5
```

1.1.3 模块的运行方式

定义一个模块，名为myModule2.py，其中代码为：

```
#可执行语句
print("mymodule2...")
m = 10
print(f"mymodule2:m={m}")
```

定义模块test2.py ，导入自定义模块myModule2时，被导入模块的”可执行语句“会立即执行。

```
import myModule2        #输出：mymodule2...
                        #输出：mymodule2:m=10
import myModule2
```

注意：执行多次import，一个模块只会被导入一次。上面的打印语句只会输出1次。

> **不提倡上面这种直接在模块中编写功能性语句的做法**
>
> 因为模块被外界引入时，这些代码会默认执行。会对调用当前模块的外部程序造成影响。

既不想影响外部代码，又想要保持本模块内相关功能代码的运行。我们该怎么做呢？

我们可以通过模块中的默认属性__name__来判断。

我们在module2.py 中录入下面代码并执行

```
print(__name__,type(__name__))      #输出：__main__ <class 'str'>
```

我们再执行test2.py，我们会看到上面语句的输出结果为：

```
print(__name__,type(__name__))      #输出：myModule2 <class 'str'>
```

Python中对于模块的调用有两种模式，

- **脚本模式**：自身模块开发时，作为独立程序由解释器直接运行
  __name__ 的内容为字符串：__main__
- **模块模式**：被其他模块导入，为其他模块提供资源（变量、函数、类的定义）
  __name__ 的内容为字符串：myModule2（模块的名字）

在myModule2中编写：

```
if __name__ == '__main__':
    m = 10
    print(f"mymodule2:m={m}")
    print(add(3,2))
```

更常见的一种方式：将函数放在主函数外部上方。

- 以保证被别人调用函数的时候，不会调用自身的内部函数。
- 还可以保证通过这种脚本方式的判断更加安全的实现函数的测试。

```
def main():
    m = 10
    print(f"mymodule2:m={m}")
    print(add(3,2))

#快捷方式：输入m，根据代码提示回车，生成下面代码
#以脚本程序运行，以保证被其他模块调用的时候自动运行自身的可执行代码
if __name__ == '__main__':
    # m = 10
    # print(f"mymodule2:m={m}")
    # print(add(3,2))

    #更常见的写法：
    main()
```

注意：自定义模块的执行只会有一次。

当再次导入自定义模块时，将不会有作用。

```
import myModule2        #导入自定义模块时，被导入模块的”可执行语句“会立即执行
import myModule2        #你执行多次import，一个模块只会被导入一次。
```

1.1.4 模块搜索路径

整个搜索过程顺序：

（1）内置模块（例如：math）

首先搜索系统内部的模块。

（2）当前模块所在的目录

如果在内置模块中没有找到，那就搜索当前所在目录。

如果没有找到，模块名下会有红线提示。运行结果会有如下报错：

```
ModuleNotFoundError:No module named 'myModule3'
```

新建myModule3后，再次运行成功调用自定义myModule3：

```
import myModule3     #输出：myModule3...
```

（3）环境变量PYTHONPATH（默认包含python的安装路径）

将原myModule3.py通过refactor重命名为myModule3_.py，重新将此文件放在别的路径下，并复制路径到环境变量下。（打开我的电脑 - 属性 - 高级 - 环境变量 - 系统变量 - 新建）

变量名为“PYTHONPATH”，变量值为刚才复制路径+“ \”，选择确定。

运行下方代码后依然报错：

```
import myModule3
```

测试是否环境变量配置错误：

右键点击windows按钮 - 选择“运行” - 输入“cmd” - 界面中输入“python”。

然后输入“import myModule3”，界面返回结果“myModule3...”，说明myModule3是可以正常被访问并调用。

由于Pycharm在启动时会将环境变量加载到缓存中并使用，所以在使用中修改环境变量导致修改的环境变量不能及时被反应出来，所以需要重启Pycharm。

重启后正常运行。

（4）Python安装路径下的Lib文件夹

Python安装路径下的Lib，代表库，是Python默认安装的库。

将myModule3_.py文件放在当下Lib文件夹路径下，运行代码依然可以成功访问它。

（5）lib文件夹下的site-packages文件夹（第三方模块）

该文件夹下保存着安装的第三方的工具，将文件myModule3_.py粘贴到该文件夹下，运行代码仍然可以访问到。

（6）sys.path.append()追加的目录

运行以下代码，输出结果为一系列路径。这些路径为系统在进行查找时搜索的路径。

```
import sys
print(sys.path)
```

尝试在该路径中追加一些路径：

先将之前第三方文件路径下的myModule3_.py删除，在该项目下选择“New Directory” - 命名"lib".

将文件复制到该“lib”文件夹下，运行下方代码进行追加路径：

```
import myModule3

import sys
# print(sys.path)

sys.path.append("D:\XX..")      # ""内为新建lib所在路径
print(sys.path)                 # 查看结果是否添加了新的路径，结果中已经打印
                                # 出“myModule3...”
```

提问：**为什么在Lib文件夹下找不到time、math和sys模块?**

因为这些为系统内置模块，想要查看它们，可通过运行下列代码：

```
import sys
print(sys.builtin_module_names)
```

【扩展】 指定搜索路径

以后代码需要放到其他电脑上运行，如何保证仍然能够找到这个文件夹呢?

绝对路径：从盘符出发的路径
相对路径：不是从盘父出发的路径

```
import os

print(__file__)     #输出当前文件的全路径：
D:/itsishu/python_workspace/demo4/demo1.py

print(os.path.dirname(__file__))    #获取指定文件所在的目录（文件夹）
                                    #输出：D:/itsishu/python_workspace/demo4
```

注意：获取文件路径不能使用字符串截取，因为不同操作系统的路径表示不同

```
import sys
import os

print(os.path.dirname(__file__) + r"/lib")      #获取当前项目路径下的lib文件夹
sys.path.append(os.path.dirname(__file__) + r"/lib")
print(sys.path)          #查看当前系统中的路径
```

输出结果：

> D:/itsishu/python_workspace/demo4/lib

```
['D:\itsishu\python_workspace\demo4', 'D:\itsishu\python_workspace\demo4',
'D:\itsishu\python_workspace\resource', 'D:\Program Files\JetBrains\PyCharm
2020.2.3\plugins\python\helpers\pycharm_display', 'D:\soft\python3.8.6\python38.zip',
'D:\soft\python3.8.6\DLLs', 'D:\soft\python3.8.6\lib', 'D:\soft\python3.8.6',
'D:\soft\python3.8.6\lib\site-packages', 'D:\Program Files\JetBrains\PyCharm
2020.2.3\plugins\python\helpers\pycharm_matplotlib_backend',
'D:/itsishu/python_workspace/demo4/lib']
```

1.1.5 其他方式导入模块

（1）从模块中引入指定的函数　（from ... import ...）

```
from myModule1 import add,divid        #引入模块中多个函数，用逗号分隔

print(add(3,2))                  #直接使用 函数名（） 进行调用
print(divid(3,2))
print(myModule1.add(3,2))      #没有被引入的函数，不能被调用
```

（2）一次性引入模块中全部函数　（from ... import *）

```
from myModule1 import *
print(add(3,2))           #引入myModule1中的全部函数
print(divid(3,2))
print(multiply(3,2))
```

（3）import 和 from ... import ...的区别

```
import myModule1
print(myModule1.add(3,2))        #需要使用模块名来调用函数

from myModule1 import *
def add(x,y):
    return f"10*{x}+10{y}={10*(x+y)}"

print(add(3,2))                  #可以直接调用函数，默认优先调用自身定义的函数
```

（4）使用别名，解决变量或函数重名问题

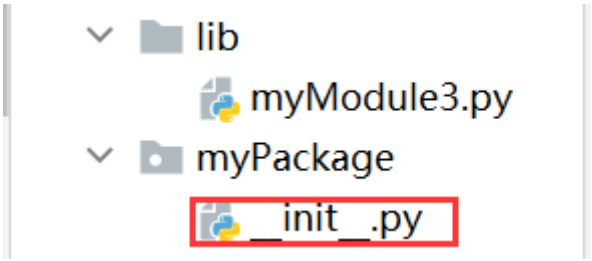
```
#给函数起别名，避免命名冲突
from myModule1 import add as m1add
def add(x,y):
    return f"10*{x}+10{y}={10*(x+y)}"

print(add(3,2))
print(m1add(3,2))

#给模块起别名，简化书写
import myModule1 as m
print(m.add(3,2))
```

1.2 包 （Package）

1.2.1 包的概念

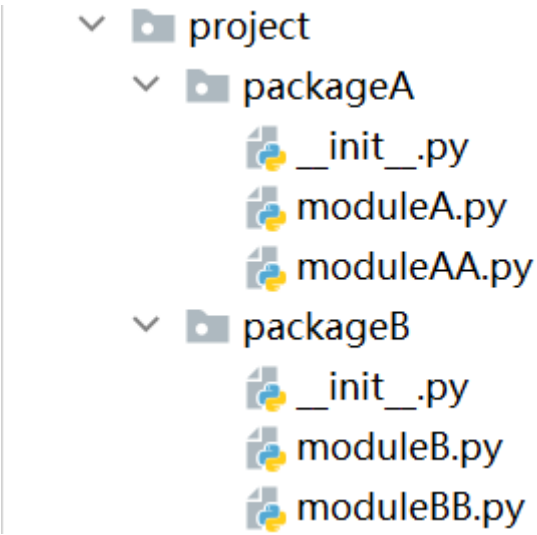
包的本质就是一个文件夹，有__init__.py文件作为标识，不能删除。该标识可以自己创建。

包下可以建立不同模块，方便大型项目管理文件。



在demo4项目文件夹下新建project文件夹，并在project下新建两个包packageA和packageB。

在packageA下新建moduleA.py和moduleAA，



moduleA.py中的内容:

```
info = "moduleA..."
print(info)

def testA():
   print("moduleA:testA()....")
```

moduleAA.py中的内容:

```
info = "moduleAA..."
print(info)

def testAA():
   print("moduleA:testAA()....")
```

在demo4下新建python文件demo2.py，

（1） 调用moduleA中“info”:

```
import project.packageA

print(project.packageA.moduleA.info)  #不能直接调用moduleA.info,需要使用全名
project.packageA.moduleA.testA()      # 运行结果：moduleA...
                                      #          moduleA...
                                      #          moduleA:testA()....
                                      # 出现两次打印moduleA，是因为第一次打印是在
                                      # moduleA里面的print(info),第二次打印是调用
                                      # moduleA时打印的。
```

（2）更简单的调用方式：

```
from project.packageA import moduleA
print(moduleA.info)
moduleA.testA()
```

（3）模块内的变量或函数的直接导入：

```
from project.packageA.moduleA import testA,info
print(info)
testA()
```

输出结果：

> moduleA...
> moduleA...
> moduleA:testA()...

运行结果依然不受影响

（4）包间的模块导入

在packageB中新建Python文件moduleB.py

```
from project.packageA.moduleA import testA,info
print(info)
testA()
```

运行结果一样：

> moduleA...
> moduleA...
> moduleA:testA()....

1.2.2 包和模块的导入

本质：引入了__init__.py文件

示例（1）

在packageA中的__init__.py编辑：

```
print("packageA__init__被调用了...")
```

在demo2.py中调用：

```
import project.packageA.moduleA
```

运行结果表明__init__.py模块被执行了：

> packageA**init**被调用了...
> moduleA...

示例（2）：将包内模块一次性先导入，最后直接引用包来引用所有模块

在package2的__init__.py文件中：

```
import project.packageA.moduleA
print("packageA__init__被调用了...")
```

在demo2.py中：

```
import project.packageA
project.packageA.moduleA.testA()
```

运行结果：

> moduleA...
> packageA**init**被调用了...
> moduleA:testA()....

但是在demo2.py中调用moduleAA失败

在__init__.py中引用moduleAA：

```
import project.packageA.moduleAA
```

运行成功。

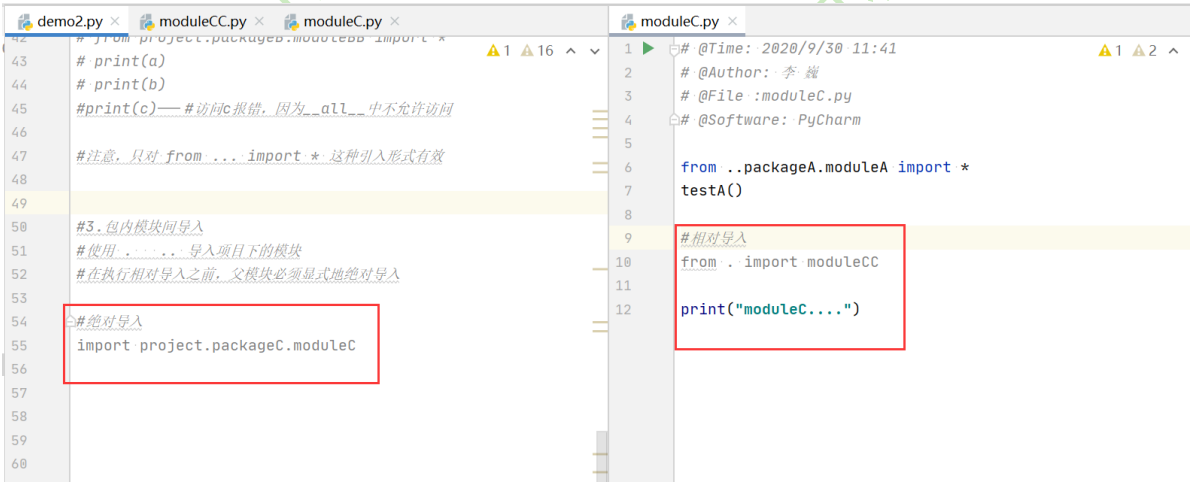
在demo2.py中

```
import project.packageB.moduleBB
```

注意：只对 from ... import *这种引入形式有效

1.2.3 包内模块间导入

绝对导入和相对导入：



使用 . 和 ..导入项目下的模块

在执行相对导入之前，父模块必须显式地绝对导入

2.面向对象

不同的编程思想：

- 面向过程：是以过程为中心的编程思想。
- 面向对象：是对象及其之间的交互为中心的编程思想。

为什么要学习面向对象技术?

计算机编程技术是逐步发展过来的。面向对象技术的发展，也是伴随着计算机计算能力的提升，以及计算机在其他领域的应用，而发展起来的。

面向过程开发过程中的特点，很适合“科学运算领域”。目的明确，变化较少，注重拆解的过程，关注运算效率。

但是随着计算技术在“**现实世界的模拟**”等领域的应用，面临着新的问题与需求：

- 事物的抽象是开放的，未来会根据需要进行调整 （**适应变化的需要**）
- 业务流程不仅仅关注执行的过程，还关注事物间的关系 （**适应抽象的需要**）
- 业务需求庞大的时候，项目周期的瓶颈在于开发时间，而不局限在运行效率上 （**适应效率的需要**）

类比：

- 面向过程的函数间调用

- 面向对象的分类处理：

思维方式的转变：

> 面向对象是相对于面向过程的，比如你要充话费，你会想，可以下个支付宝，然后绑定银行卡，然后在淘宝上买卡，自己冲，这种就是**面向过程**。但是对于你女朋友就不一样了，她是面向“对象”的，她会想，谁会充话费呢？当然是你了，她就给你电话，然后你把之前的做了一遍，然后她收到到帐的短信，说了句，亲爱的。这就是**面向对象**！女生的思维大部分是面向“对象”的！她不关心处理的细节，只关心谁可以，和结果！

比较维度	面向过程	面向对象
背景	科学计算为目标	软件系统开发
编程方法	自顶向下	自底向上
代码结构	程序=数据+算法（函数/过程）	程序=对象+交互
数据操作主体	函数/过程中	对象的方法中
模拟方法	通过函数操纵表现世界的数据和状态	把世界描绘成具有主动性的对象间的交互
编程思维	处理数据的步骤	面向对象分析
运行效率	较高	稍低
大规模开发效率	较低	较高

程序设计思想：

- 面向过程：关注步骤，怎么一步一步完成。
- 面向对象：关注关系，谁能做什么。

  并非对立，而是互为补充。

类比：

面向过程就像“编年体”， 面向对象就像“纪传体”。

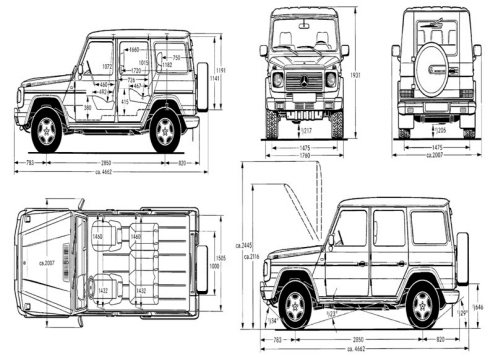
编年体：例如《资治通鉴》的《秦纪》、《汉纪》等，写法类似于：魏明帝太和五年...

纪传体：例如《史记》的《项羽本纪》、《孔子世家》、《廉颇蔺相如列传》等，

写法类似于：孔子去曹适宋，与弟子习礼大树下。

2.1类和对象

2.1.1 类和对象的概念



- **类**（Class）：具有相同属性和方法的一类事物。

  类是抽象的，就像一个模子或图纸。规定了具体对象的特征和行为。

概念描述的差异:

面向过程	面向对象
变量（状态）	属性（特征）
函数（操作）	方法（行为）

- **对象**（Object）：也被称为“实例”。将类中定义的特征具体化（赋值），就是一个对象或实例。

2.1.2 类和对象的定义与调用

```
class Car:                          #定义类:    class 类名:

   energy = "电动"                  #属性,表示"特征"

   def move(self):                  #方法,表示"行为"
      print("在移动...")

c = Car()                           #对象实例化 :  对象名称 = 类名()
print("能源类型:",c.energy)          #访问属性:     对象名称.属性
c.move()                            #调用方法:     对象名称.方法
```

小结:

在属性的表述上，可以将 *c.energy* 理解为 “车辆实例c的能源类型”，符号" . " 可以表述为 “的”

在方法的调用上，可以将 *c.move()* 理解为 “车辆实例c在移动”， 符号" . " 可以表述为 ”在做..."

通常，我们将属性命名为“**名词**”；方法命名为“**动宾结构**”。

类名一般开头字母大写。

属性和方法的命名同样遵守变量的命名规则，建议使用**驼峰命名法**（开头字母小写，后面每个单词开头字母大写）

2.1.3 对象的实例化

2.1.3.1 对象实例化

也称为对象初始化。表示根据类的定义，生成一个具体的对象的过程。

下面是“Dog”狗类的定义，与一直具体的的狗的实例化过程。

```
class Dog:

    def bark(self):          #实例方法   |   成员方法
        print("self:",id(self))
        print("汪汪！")

c = Dog()                    #定义名字为c的一个对象
c.bark()                     #相当于bark（c）将对象c传入了函数bark中
print("c:",id(c))
```

输出结果：

```
self:1575961703952
汪汪！
c:1575961703952
```

我们可以观察到上面的c和self的地址是相同的，说明在调用bark方法时，c将自身的地址传递给了参数self。

2.3.1.2 对象的属性

对象属性，也成为成员属性。可以在类的内部或外部进行定义。

每个对象（或称为“实例”）都有自己的属性空间，彼此互不影响。

（1）在类的外部，添加对象属性

```
class Dog:

  def bark(self):            #实例方法   |   成员方法
     print(f"{self.name}：","汪汪！")    #可以在对象的方法中，通过self来访问对象属性

c = Dog()                    #初始化一个对象，就会分配一个新的内存空间
c.name = "咯咯"              #两只狗的信息彼此独立
d = Dog()
d.name = "豆豆"
#print(f"{c.name}：",end="")
c.bark()
#print(f"{d.name}：",end="")
d.bark()
```

小结:

一个类可以有多个对象，多个对象都对应同一个类。

类是抽象的，对象是具体的。

每个对象是彼此独立的，实例的成员属性互不干扰。

（2）在类的内部，添加对象属性

魔术方法：前后用两个下划线包裹的，具有特殊功能的方法。

__init__ 方法的作用是: 为每个具体的实例初始化 "属于实例本身的属性"。__init__ 方法会在初始化一个对象的时候自动被调用。

```
class Dog:

 def __init__(self):           #初始化对象的时候，默认被调用
    print("__init__:",id(self))
    print("一只小狗诞生了")
                                #__init__方法执行完毕，会默认返回自身对象self

 def bark(self):
    print(id(self),"bark....")

c = Dog()           #默认调用__init__方法，来为对象初始化属性值
print(id(c))
c.bark()            #成员方法调用时，默认会传递自身对象
```

输出结果：

> **init**: 2355044562448
> 一只小狗诞生了
> 2355044562448
> 2355044562448 bark....

练习：让每只小狗出生时必须有名字?

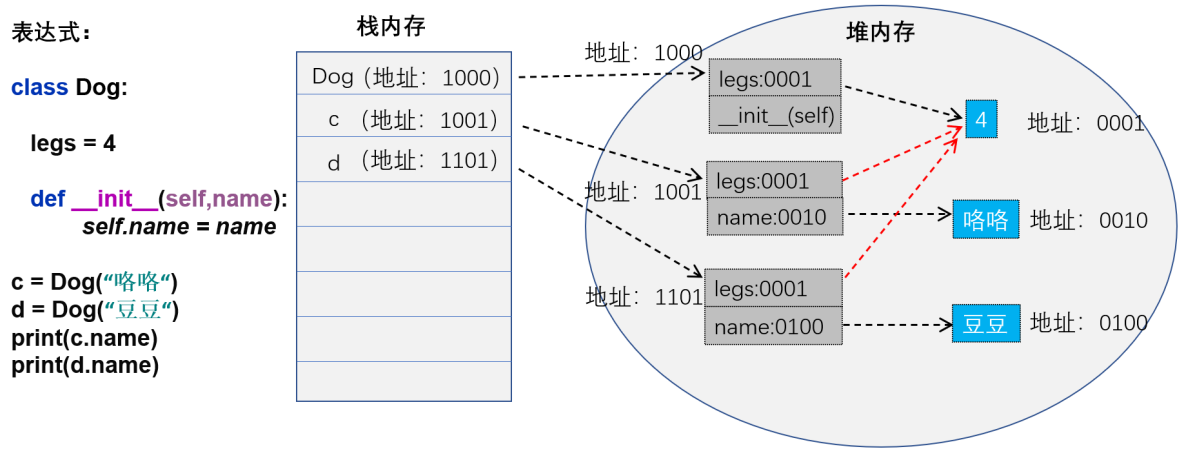
对于每个对象都有，但是内容（信息）彼此不同的属性，我们定义为实例属性，或者成员属性，可以通过__init__方法，在类初始化的时候进行赋值。

```
class Dog:

   def __init__(self,name):          #初始化对象的时候，默认被调用
      self.name = name               #成员属性，每个对象特有的属性；也称做”实例属性“
      print(f"一只名叫{name}的小狗诞生了")

   def bark(self):
      print(f"{self.name}:汪汪！ ")

c = Dog("咯咯")
# print(c.name)
c.bark()
d = Dog("豆豆")
```

```
d.bark()
```

扩展：

类和对象的底层逻辑



课堂练习：

结合上面的图形，请尝试画出下面程序的底层逻辑图：

```
class Dog:

    def __init__(self,name):
        self.name = name
        self.blood = 10

    def bite(self,target):
        target.blood -= 2
        if target.blood <=4 :
            print(f"{target.name}快被{self.name}咬死了")
        else:
            print(f"{self.name}咬了{target.name},{target.name}掉了2滴血")

c = Dog("咯咯")
d = Dog("豆豆")
d.bite(c)                   #输出：豆豆咬了咯咯,咯咯掉了2滴血
d.bite(c)                   #输出：豆豆咬了咯咯,咯咯掉了2滴血
d.bite(c)                   #输出：咯咯快被豆豆咬死了
```

2.1.4 类属性和成员属性

概念:

- 类属性，每个对象共有的属性。不会因为对象个体的改变而改变。
- 成员属性，每个对象独有的属性。对象间互不影响。

访问方式:

- 类属性可以通过：“类名.类属性”的方式访问；也可以通过“对象.类属性”的方式访问
- 成员属性只能通过“对象.类属性”的方式访问

```
class Dog:

   legs = 4                        #类属性，每个对象共有的属性

   def __init__(self,name):
      self.name = name
      #self.legs = 4
      print(f"一只名叫{self.name}的小狗出生了")

c = Dog("咯咯")
d = Dog("豆豆")
print("狗狗c的腿数：",c.legs)     #访问类属性：对象.类属性
print("狗狗d的腿数：",d.legs)

print(Dog.legs)                   #访问类属性：  类名.类属性
```

输出结果:

```
一只名叫咯咯的小狗出生了
一只名叫豆豆的小狗出生了
狗狗c的腿数： 4
狗狗d的腿数： 4
4
```

2.1.5 对象间的互动

- 对象可以作为参数，传递到其他对象的方法中。
- 对象可以通过自身的变量名，访问自身属性和方法；即使在其他对象的内部，也是可以这么做的。
- 一个对象将另一个对象作为自己的属性，表示“拥有、持有”的状态

下面的例子综合表现了上面的概念:

> 剧情:
>
> 狗（豆豆）咬了狗（咯咯）
>
> 人（胡子哥）驱赶狗（豆豆）
>
> 狗（豆豆）咬了人（胡子哥）
>
> 狗（咯咯）咬了狗（豆豆）
>
> 人（胡子哥）收养了狗（咯咯）

```
class Person:

      def __init__(self,name):
         self.name = name
         self.pet = None                    #可以保存另一个对象，作为自身的属性；表示“拥
有”的概念

      def turf_out(self,dog):
         print(f"{self.name}驱赶了{dog.name}")

      def take(self,dog):
         self.pet = dog                     #可以在实例方法中，随时添加/修改对象的属性值
         print(f"{self.name}收留了{dog.name}")

      def getInfo(self):
         if self.pet:                      #可以逐层访问对象及其属性值
            print(f"{self.name}有一只宠物，它的名字是：{self.pet.name}")
         else:
            print(f"{self.name}没有宠物")


   luoluo = Dog("咯咯")
   doudou = Dog("豆豆")
   huzige = Person("胡子哥")

   #1.狗（豆豆）咬了狗（咯咯）
   doudou.bite(luoluo)                      #可以将一个对象作为参数，传递到另一个对象的方法中

   #2.人（胡子哥）驱赶狗（豆豆）
   huzige.turf_out(doudou)

   #3.狗（豆豆）咬了人（胡子哥）
   doudou.bite(huzige)

   #4.狗（咯咯）咬了狗（豆豆)
   luoluo.bite(doudou)

   #5.人（胡子哥）收养了狗（咯咯）
   huzige.take(luoluo)

   huzige.getInfo()
```

输出结果:

> 豆豆咬了咯咯
> 胡子哥驱赶了豆豆
> 豆豆咬了胡子哥
> 咯咯咬了豆豆
> 胡子哥收留了咯咯
> 胡子哥有一只宠物，它的名字是：咯咯

2.2 面向对象三大特征

面向对象的三大基本特征：

- 封装
- 继承
- 多态

2.2.1 封装

概念：

封装就是隐藏对象的属性和实现细节。

封装的思想保证了类内部数据结构的完整性，使用户无法轻易直接操作类的内部数据，这样降低了对内部数据的影响，提高了程序的安全性和可维护性。

好处：

- 只能通过规定方法访问数据
- 隐藏类的实现细节
- 方便修改实现
- 方便加入控制语句

封装的三个层次：

- 类的封装：外部可以任意访问/修改类中的属性和方法
- 私有属性：外部不可以访问/修改类的属性或方法
- 私有属性 + 公有方法：外部有条件限制的访问/修改属性，调用方法

实现方式：

【类的封装】
类的定义：将某些特定属性和方法进行 “隔离”
每个学生有自己的年龄，外部可以通过对象任意读取或修改

```
class Student:

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

s1 = Student("李白",18)
s2 = Student("杜甫",20)
print(s1.name,s1.age)       #输出：李白，18
print(s2.name,s2.age)       #输出：杜甫，20
```

【私有属性】

属性私有：两个下划线开头的属性就是私有属性。私有属性只能在类的内部使用，外部不能使用。

练习：不让外部读取/修改学生的年龄。

```
class Student:

    __secret = True

    def __init__(self,name,age):
        self.name = name
        self.__age = age            #使用2个下划线开头的属性，定义为私有属性
        # print(self.__age)         #类的内部是可以访问私有属性的

s1 = Student("李白",18)
# print(s1.__age)                   #外界无法直接通过属性名称来访问
# print(s1._Student__age)           #虽然语法可以在外部通过_Student__age访问到私有属性，但不建议这样做
```

可以使用__dict__属性，来查看对象中的属性值。

```
print(s1.__dict__)              #可以查看s1对象的属性值
                                #输出：{'name': '李白', '_Student__age': 18}

print(Student.__dict__)         #可以查看Student类中的属性值
#print(Student.__secret)        #不可以直接通过类名访问类的私有属性
```

输出结果：

> {'name': '李白', *'Student__age': 18}*
> *{'__module__': '__main__',* 'Student__secret': True, '__init__': <function Student.__init__ at 0x000001AFAC103EE0>, '__dict__': <attribute '__dict__' of 'Student' objects>, '__weakref__': <attribute '__weakref__' of 'Student' objects>, '__doc__': None}

【 私有属性 + 公有方法】

私有属性 + 公有方法：可以实现“有限制条件的开放给外部”，例如：可以读取年龄，但是不能随意修改年龄。

练习：设定年龄必须在【1，125】之间

```
class Student:

    def __init__(self,name,age):
        self.name = name
        self.__age = age        #使用2个下划线开头的属性，定义为私有属性

    def getAge(self):
        return self.__age

    def setAge(self,age):
        if 0 < age <=125:
            self.__age = age
        else:
            print("不能随意修改年龄")
```

```
s1 = Student("李白",18)
print(s1.getAge())
s1.setAge(200)                #设定年龄之超过限定范围
print(s1.getAge())
```

输出结果：

```
18
不能随意修改年龄
18
```

封装的常用（简化）写法

装饰器的概念：以@开头，调用另一个函数（或方法），扩展了对“所修饰方法”的功能。

```
class Dog:

    # property装饰器：把一个方法伪装成一个属性
    @property
    def bark(self):
        print("bark...")

d = Dog()
# d.bark()            #没有使用property装饰器修饰的时候
d.bark                #使用property装饰器修饰后，在调用这个方法的时候不需要加()就可以直接得到返回值
```

输出结果：

```
bark...
```

还可以对property装饰器修饰的属性用来返回属性值；还可以继续进行setter的设置，用来设置属性值。

```
class Student:

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

    @property
    def age(self):          #读取属性值的方法，除了self没有其他参数
        return self.__myage

    @age.setter             #property修饰的方法名.setter    ，当外给age赋值的时候，会默认调用
    def age(self,age):      #设置属性值的方法，需要传入设定的内容
        if 0 < age <= 125:
            self.__myage = age
        else:
            print("不能随意修改年龄")
```

```
s1 = Student("李白",18)
s1.age = 200
print(s1.name,s1.age)

s1.age = 20
print(s1.name,s1.age)
```

输出结果：

> 不能随意修改年龄
> 李白 18
> 李白 20

小结:

使用 @property 装饰器时，方法名不必与属性名相同.
可以更好地防止外部通过猜测私有属性名称来访问
凡是赋值语句，就会触发set方法。获取属性值，会触发get方法

- 另一种写法（了解）

不使用装饰器，在定义好对外开放的方法后，可以直接在类内定义开放给外部的属性名，后面property对应着获取、设置和删除操作的函数名。

同样可以实现将方法“伪装”成为属性的调用方式。

```
class Student:

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

   def getAge(self):         #读取属性值的方法，除了self没有其他参数
      return self.__myage

   def setAge(self,age):      #设置属性值的方法，需要传入设定的内容
      if 0 < age <= 125:
         self.__myage = age
      else:
         print("不能随意修改年龄")

   def delAge(self):
      del self.__myage

   # 按照顺序传参 :  获取 =>  设置 => 删除
   age = property(getAge,setAge,delAge)

s1 = Student("李白",18)
s1.age = 200
print(s1.name,s1.age)

s1.age = 20
```

```
print(s1.name,s1.age)

del s1.age    #调用delAge方法，实现删除s1的__myage属性值的效果
print(s1.age)  #删除后在访问，会报错：
            #AttributeError: 'Student' object has no attribute '_Student__myage'
```

私有方法，可以在类的内部使用；或可以有条件的开放给外部

```
class Student:

    def talk(self,identity):
        if identity == "死党":
            self.__tellSecrect()
        else:
            print("闲聊...")

    def __tellSecrect(self):
        print("我有一个秘密...")

s = Student()
s.talk("死党")
```

2.2.2 继承

面临的问题:

相似类型的定义过程中，大量重复性的代码

```
class Dog:

    def __init__(self,name):
        self.name = name
        print(f"一个名叫{self.name}的宠物出生了")

    def eat(self):
        print(f"{self.name}在吃东西...")

    def sleep(self):
        print(f"{self.name}在睡觉...")

class Cat:

    def __init__(self,name):
        self.name = name
        print(f"一个名叫{self.name}的宠物出生了")

    def eat(self):
        print(f"{self.name}在吃东西...")

    def sleep(self):
        print(f"{self.name}在睡觉...")
```

```
d = Dog("咯咯")
d.eat()
c = Cat("布丁")
c.sleep()
```

2.2.2.1 继承的概念:

在类名后面添加小括号，其中填写继承自哪个类型。

对相同的属性或方法进行抽象，形成类别的提炼或分拆。

好处:

让类的定义具有更好的扩展性。它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

实现方式:

通过继承创建的新类称为“子类”或“派生类”。

被继承的类称为“基类”、“父类”或“超类”。例如:

Dog 和 Cat 继承了 Pet 类
Dog 和 Cat 称为 Pet 的子类 / 派生类
Pet 称为 Dog 和 Cat 的父类 / 超类 /基类

```
class Pet:

    master = True

    def __init__(self,name):
        self.name = name
        print(f"一个名叫{self.name}的宠物出生了")

    def eat(self):
        print(f"{self.name}在吃东西...")

    def sleep(self):
        print(f"{self.name}在睡觉...")

class Dog(Pet):          # 语法:    class 类名（父类名）:
    pass

class Cat(Pet):
    pass
```

2.2.2.2 子类调用方法的顺序:

子类默认拥有父类公有的属性和方法的定义

- 子类中有，调用子类的
- 子类中没有，调用父类的
- 一旦在子类中找到，就不在父类中查找了

继续上面的代码：

```
#每个对象修改类属性的值，仅在对象内部有效
c.master = False
print(d.master)     #输出：True
print(c.master)     #输出：False
print(Pet.master)   #输出：True

#如果类属性做了修改，所有对象的值都会跟着改变
Pet.master = False
print(d.master)     #输出：False
print(c.master)     #输出：False
print(Pet.master)   #输出：False
```

> 建议：参考<<类和对象的底层逻辑>>图，理解类属性和对象属性修改的原因

子类不能继承父类的私有属性或方法

```
class Father:

    __secrect = "XXXX"
    story = "从前有座山..."

    def tellAStory(self):
        print("我的故事：",self.story)

    def __tellASecret(self):
        print("我的秘密：",Father.__secrect)

class Son(Father):

    def tell(self):
        # Father._Father__tellASecret(self)      #虽然语法支持，但不建议这么做
        self.tellAStory()

s = Son()
s.tell()
```

输出结果：

> 我的故事： 从前有座山...

2.2.2.3 覆盖/重写

概念：子类和父类有相同名称的方法。可以理解为子类对于父类行为的扩展或补充

```
class Pet:

    def __init__(self,name):
        self.name = name
        print(f"一个名叫{self.name}的宠物出生了")

    def eat(self):
        print(f"{self.name}在吃东西...")

class Dog(Pet):

    def lookAfter(self):
        print(f"{self.name}在看门...")

    def eat(self):                          #重写父类的eat方法,重新定义父类的方法
        print(f"{self.name}在啃骨头...")

d = Dog("咯咯")
d.eat()                                     #当子类和父类方法相同时，只会调用子类方法
d.lookAfter()
```

输出结果：

> 一个名叫咯咯的宠物出生了
> 咯咯在啃骨头...
> 咯咯在看门...

2.2.2.4 super()

子类可以在类定义时，可以使用super()调用父类的方法。

```
class Cat(Pet):

    # 应用2：对象初始化时，简化重复属性的赋值
    def __init__(self,name,age,sex):
        # self.name = name
        # print(f"一个名叫{self.name}的宠物出生了")
        # super(Cat, self).__init__(name)
        super().__init__(name)
        self.age = age
        self.sex = sex


    # 应用1：在重写方法时，通过super()调用父类方法后，扩展功能
    def eat(self):
        print(f"{self.name}伸个懒腰")
        super().eat()
        #self.eat()          # 不可行，默认是死循环
        print(f"{self.name}吃完东西后，用唾液洗洗脸..")

c = Cat("布丁",2,"雌性")
c.eat()
```

```
print(c.name,c.age,c.sex)
```

输出结果：

> 一个名叫布丁的宠物出生了
> 布丁伸个懒腰
> 布丁在吃东西...
> 布丁吃完东西后，用唾液洗洗脸..
> 布丁 2 雌性

2.2.3 多态

想要解决的问题：

- 让程序能够有更强的灵活性
- 让程序能够有更好的适应性

概念:

一个类的多种形态。

同一个行为具有多个不同表现形式或形态的能力。是指一个类实例（对象）的相同方法在不同情形有不同表现形式。

好处：

- 1. 消除类型之间的耦合关系
- 2. 可替换性
- 3. 可扩充性
- 4. 接口性
- 5. 灵活性
- 6. 简化性

实现方式：

方式一：通过继承来实现

```
class Animal:
   def eating(self):
      print("动物在吃东西...")

class Pet(Animal):
   def eating(self):
      print("宠物在吃东西...")

class Dog(Pet):
   def eating(self):
      print("狗在吃啃骨头...")

class Cat(Pet):
   def eating(self):
      print("猫在吃鱼...")

class Zoo:
```

```
    def animallEating(self,animal):
        if isinstance(animal,Animal):
            print("这是展示动物吃东西的地方：")
        else:
            print("这是非动物吃饭的展示")
        animal.eating()

z = Zoo()
a = Animal()
d = Dog()
c = Cat()


# Dog 和 Cat 作为Animal的不同形态
# 都可以直接调用 animal所具有的属性或方法
z.animallEating(a)
z.animallEating(c)
z.animallEating(d)
```

输出结果：

> 这是展示动物吃东西的地方：
> 动物在吃东西...
> 这是展示动物吃东西的地方：
> 猫在吃鱼...
> 这是展示动物吃东西的地方：
> 狗在吃啃骨头...

方式二:【Python特有】

没有继承关系，但是具备相同特征/方法，也可以直接使用。

继续在上面的代码后面追加定义：

```
class venusFlytrap:     #捕蝇草
    def eating(self):
        print("捕蝇草在吃小虫子...")

v = venusFlytrap()

z.animallEating(v)      #输出：这是非动物吃饭的展示
                        #      捕蝇草在吃小虫子...
```

关于isinstance 和 type的差别：

```
#判断类型时，只看直接类型
print(type(c) is Cat)           #True
print(type(c) is Animal)        #False
print(type(c) is Pet)           #False

#isinstance 判断对象是否为一个类型的实例
# 判断实例类型时，涵盖父类的类型
print("*"*30)
print(isinstance(c,Cat))        #True
print(isinstance(c,Pet))        #True
print(isinstance(c,Animal))     #True
print(isinstance(v,Animal))     #False
```

2.3 类方法和静态方法

2.3.1 实例方法

实例方法：默认有个self参数，且只能被对象调用。

```
class Dog:

    legs = 4
    teeth = 42

    def printInfo(self):
        print(f"狗有{self.legs}条腿，{self.teeth}颗牙齿")

d = Dog()
d.printInfo()
```

2.3.2 类方法

类方法：默认有个 cls 参数，可以被类和对象调用，使用 @classmethod 装饰器 进行标注的方法。

```
class Dog:

    legs = 4
    teeth = 42

    @classmethod
    def printInfo(cls):
        print(f"狗有{cls.legs}条腿，{cls.teeth}颗牙齿")

d = Dog()
d.printInfo()           #输出：狗有4条腿，42颗牙齿
Dog.printInfo()         #输出：狗有4条腿，42颗牙齿
```

使用类名进行调用时，默认会补充（传入）参数cls 表示当前类

2.3.3 静态方法

静态方法：不带 self 参数和cls参数的方法叫做静态方法，使用 @staticmethod 装饰器 进行标注的方法。

```
pyclass Dog(object):

    @staticmethod
    def bark():
        print("wangwang!!")

d = Dog()
d.bark()        #输出：wangwang!!
Dog.bark()      #输出：wangwang!!
```

静态方法，类和对象均可以调用。

小结:

	需要实例属性	需要类属性	和类相关，不需要类属性
实例方法（self）	是		
类方法（cls）		是	
静态方法（*args）			是

补充:

在有些地方，也会有类似下面的分类和叫法：

- 绑定方法
  - 绑定到对象：方法默认传递 调用对象 作为参数
  - 绑定到类：方法默认传递 类本身 作为参数
- 非绑定方法：

  就是将普通方法放到了类的内部，不会传递实例和类本身作为参数

2.4 魔术方法

在Python中，所有以“__”双下划线包起来的方法，都统称为“Magic Method”（魔术方法），官方文档也称为“特殊方法”。这样的方法往往都有特殊用途，被Python解释器自动调用，所以也不建议个人定义方法名的时候用双下划线包裹，以免产生混淆。

一般用在对 数据运算 / 类型转换 / 容器操作 进行重新定义的时候，就会调用对象中对应名称的魔术方法。

2.4.1 __init__方法

__init__方法，主要作用是为初始化的对象设置初始值。

```
class  Pig:

    def __init__(self):
        print("一只小猪出生了")

p = Pig()
```

2.4.2 __new__方法

Python在初始化对象时：

- 先执行 __new__方法:分配内存空间，并返回构建好的对象(的地址)
- 再执行__init__方法 :为构建好的对象赋予初始值

```
class  Pig(object):

    def __new__(cls):
        print("分配内存空间...")
        print("cls:",cls)
        obj = object.__new__(cls)    #这条语句不能少，通过object类的__new__方法来分配内存空间
        print("obj: ",obj)                #
        return obj

    # __new__返回的对象（地址）就是__init__方法中的self所指向的对象（地址）
    def __init__(self):
        print("self:",self)
        print("一只小猪出生了")

p = Pig()
```

输出结果：

> 分配内存空间...
> cls: <class '**main**.Pig'>
> obj: <**main**.Pig object at 0x0000029CB2269CD0>
> self: <**main**.Pig object at 0x0000029CB2269CD0>
> 一只小猪出生了

可以观察到：obj的地址和self的地址相同。**__new__方法返回的对象（地址）就是__init__方法中的self所指向的对象（地址）。**

如果__new__方法返回为None（空对象），即使已经分配了内存空间，意味着没有将对象传递给__init__方法。**__init__方法不会执行。**

```
class  Pig(object):

    def __new__(cls):
        print("分配内存空间...")
        obj = object.__new__(cls)
        print("obj:",obj)
        return None             #如果返回为None,__init__方法将不会被调用

    def __init__(self):
        print("self:",self)
        print("一只小猪出生了")

p = Pig()
```

输出结果：

> 分配内存空间...
> obj: <**main**.Pig object at 0x00000220AB229070>

为了保证对象初始化的所有参数都能传递给__init__方法，__new__的参数可以使用可变参数来定义，参数值会自动传递到__init__方法中。

```
class  Pig:

    # __new__方法中的*args，**kwargs会将接收到的函数自动传递到__init__方法中
    def __new__(cls, *args, **kwargs):
        print(f"args:{args},kwargs:{kwargs}")
        return object.__new__(cls)

    def __init__(self,name,brother):
        print(f"一只名叫{name}小猪出生了")

p = Pig("佩奇",brother="乔治")
```

输出结果：

> args:('佩奇',),kwargs:{'brother': '乔治'}
> 一只名叫佩奇小猪出生了

2.4.3 __del__方法

从内存中清除对象的时候，对象会默认执行的方法。具体执行的时间节点：

- 程序执行完毕后，释放内存时
- 执行del指令时

(1) 当模块执行完毕后，解释器会自动释放内存。对象被从内存移除时，对象中的__del__方法会被python解释器自动调用。

```
class Pig:

    def __init__(self):
        print("++对象被初始化了++")

    def __del__(self):
        print("--对象被删除了--")

p = Pig()
print("[程序执行完毕了]")
```

输出结果：

```
++对象被初始化了++
[程序执行完毕了]
--对象被删除了--
```

（2）使用del指令删除对象时，也会触发__del__方法的执行

```
class Pig:

    def __init__(self):
        print("++对象被初始化了++")

    def __del__(self):
        print("--对象被删除了--")

p = Pig()
del p                #手动删除对象，释放内存
print("[程序执行完毕了]")
```

输出结果：

```
++对象被初始化了++
--对象被删除了--
[程序执行完毕了]
```

仔细观察可以发现，这一次的 “--对象被删除了--” 出现在 “[程序执行完毕了]” 语句之前。

思考：下面的代码中，删除了p1 为什么没有释放内存呢？

```
class Pig:

    def __init__(self):
        print("++对象被初始化了++")

    def __del__(self):
        print("--对象被删除了--")

p1 = Pig()
p2 = p1
del p1                #手动删除对象，释放内存
#print(p1)             #删除p1后，报错：NameError: name 'p1' is not defined；说明p1确实被删除了
print("[程序执行完毕了]")
```

输出结果：

```
++对象被初始化了++
[程序执行完毕了]
--对象被删除了--
```

结论：只有当指向对象的所有变量都被删除的时候，对象空间才会被释放

```
class Pig:

    def __init__(self):
        print("++对象被初始化了++")

    def __del__(self):
        print("--对象被删除了--")

p1 = Pig()
p2 = p1
del p1
del p2
print("[程序执行完毕了]")
```

输出结果：

```
++对象被初始化了++
--对象被删除了--
[程序执行完毕了]
```

2.4.4 __call__方法

__call__方法，将对象当作函数执行的时候会被默认调用。

概念：

```
class Flight:

    def __call__(self, *args, **kwargs):
        print("__call__方法被调用了")

f = Flight()         #f是一个对象
f()                  #输出：__call__方法被调用了
```

在对象名后面直接加小括号，像函数那样调用，就会触发__call__方法的执行。

应用：描述一架飞机起飞前的准备过程

```
class Flight:

    def __init__(self,number):
        self.number = number
        print(f"--{number}号航班--")

    # 办理登机手续
    def checkIn(self):
        print("办理登机手续")

    # 安全检查
    def securityCheck(self):
        print("安全检查")

    # 登机,起飞
    def bording(self):
        print("登机,起飞")


    def __call__(self, *args, **kwargs):     #将对象的行为进行内部协调与封装
        print("__call__方法被调用了")
        self.checkIn()
        self.securityCheck()
        self.bording()


f = Flight("CA1426")
#写法一：     外部可以依次调用对象中的每个具体方法
# f.checkIn()
# f.securityCheck()
# f.bording()
#写法二：    从逻辑上来说，飞机自身的过程应该有自己进行协调，外部只需要关注对象的行为就可以了
f()
```

输出结果：

> --CA1426号航班--
> 办理登机手续
> 安全检查
> 登机,起飞

如果对象像函数调用过程中，需要传递参数，可以参考：

```
class Flight:

    def __init__(self,number):
        self.number = number
        print(f"--{number}号航班--")

    # 办理登机手续
    def checkIn(self):
        print("办理登机手续")

    # 安全检查
    def securityCheck(self):
        print("安全检查")

    # 登机,起飞
    def bording(self):
        print("登机,起飞")

    def __call__(self, state):
        print("飞机当前状态：",state)
        self.checkIn()
        self.securityCheck()
        self.bording()


f = Flight("CA1426")
f("正常")
```

输出结果:

```
--CA1426号航班--
飞机当前状态： 正常
办理登机手续
安全检查
登机,起飞
```

2.4.5__str__方法

__str__方法主要作用是在对象被打印时，定义输出的字符串内容。

调用时机：

- 打印一个对象的时候，默认会被调用
- 使用str() 对对象进行强制转化后，输出结果时会被调用。

（1）打印对象时，我们可以自定义输出的字符串内容。

```
class Dog:

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

    def __str__(self):
        print(super().__str__())    #还可以通过调用父类对象的__str__方法，打印原有结果
        return f"这是一只名字叫{self.name}的狗"

d = Dog("luoluo")
print(d)
```

输出结果：

> <**main**.Dog object at 0x0000016561619070>
> 这是一只名字叫luoluo的狗

原来我们的打印一个对象时，默认显示的字符串 “<**main**.Dog object at 0x0000016561619070>”，其实是object父类中 __str__方法定义的功能，所以上面我们调用super().__str__()可以还原系统默认的输出效果。

(2)在使用str()函数将对象进行强制类型转化时，也会调用__str__方法。

```
class Dog:

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

    def __str__(self):
        print(super().__str__())    #还可以通过调用父类对象的__str__方法，打印原有结果
        return f"这是一只名字叫{self.name}的狗"

d = Dog("luoluo")

res = str(d)            #输出：<__main__.Dog object at 0x0000018B107F9070>
print(res)              #输出：这是一只名字叫luoluo的狗
```

注意：强制类型转化后的返回值，需要打印才能显示出来。

2.4.6 __repr__方法

__repr__方法作用和__str__方法的作用一样，都是输出对象打印的字符串格式。但最大的不同点在于：

在可变容器中，对象打印默认会调用__repr__方法。

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

    def __str__(self):
        print("__str__")
        return f"这是一只名字叫{self.name}的狗"
```

```
    def __repr__(self):
        print("__repr__")
        return f"dog:{self.name}"

d = Dog("luoluo")
print(d)                #输出：__str__
                        #      这是一只名字叫luoluo的狗

x = [d]
print(x)                #输出：__repr__
                        #输出：[dog:luoluo]
```

更多魔术方法和魔术属性：

魔法方法	含义
基本的魔法方法	
__new__(cls[, ...])	__new__ 是在一个对象实例化的时候所调用的第一个方法
__init__(self[, ...])	构造器，当一个实例被创建的时候调用的初始化方法
__del__(self)	析构器，当一个实例被销毁的时候调用的方法
__call__(self[, args...])	允许一个类的实例像函数一样被调用：x(a, b) 调用 x.__call__(a, b)
__len__(self)	定义当被 len() 调用时的行为
__repr__(self)	定义当被 repr() 调用时的行为
__str__(self)	定义当被 str() 调用时的行为
__bytes__(self)	定义当被 bytes() 调用时的行为
__hash__(self)	定义当被 hash() 调用时的行为
__bool__(self)	定义当被 bool() 调用时的行为，应该返回 True 或 False
__format__(self, format_spec)	定义当被 format() 调用时的行为
有关属性	
__getattr__(self, name)	定义当用户试图获取一个不存在的属性时的行为
__getattribute__(self, name)	定义当该类的属性被访问时的行为
__setattr__(self, name, value)	定义当一个属性被设置时的行为
__delattr__(self, name)	定义当一个属性被删除时的行为
__dir__(self)	定义当 dir() 被调用时的行为
__get__(self, instance, owner)	定义当描述符的值被取得时的行为
__set__(self, instance, value)	定义当描述符的值被改变时的行为
__delete__(self, instance)	定义当描述符的值被删除时的行为
比较操作符	
__lt__(self, other)	定义小于号的行为：x < y 调用 x.__lt__(y)
__le__(self, other)	定义小于等于号的行为：x <= y 调用 x.__le__(y)
__eq__(self, other)	定义等于号的行为：x == y 调用 x.__eq__(y)

魔法方法	含义
__ne__(self, other)	定义不等号的行为：x != y 调用 x.__ne__(y)
__gt__(self, other)	定义大于号的行为：x > y 调用 x.__gt__(y)
__ge__(self, other)	定义大于等于号的行为：x >= y 调用 x.__ge__(y)
算数运算符	
__add__(self, other)	定义加法的行为：+
__sub__(self, other)	定义减法的行为：-
__mul__(self, other)	定义乘法的行为：*
__truediv__(self, other)	定义真除法的行为：/
__floordiv__(self, other)	定义整数除法的行为：//
__mod__(self, other)	定义取模算法的行为：%
__divmod__(self, other)	定义当被 divmod() 调用时的行为
__pow__(self, other[, modulo])	定义当被 power() 调用或 ** 运算时的行为
__lshift__(self, other)	定义按位左移位的行为：<<
__rshift__(self, other)	定义按位右移位的行为：>>
__and__(self, other)	定义按位与操作的行为：&
__xor__(self, other)	定义按位异或操作的行为：^
__or__(self, other)	定义按位或操作的行为：|
反运算	
__radd__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rsub__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rmul__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rtruediv__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rfloordiv__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rmod__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rdivmod__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rpow__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）

魔法方法	含义
__rlshift__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rrshift__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rand__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rxor__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__ror__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
增量赋值运算	
__iadd__(self, other)	定义赋值加法的行为：+=
__isub__(self, other)	定义赋值减法的行为：-=
__imul__(self, other)	定义赋值乘法的行为：*=
__itruediv__(self, other)	定义赋值真除法的行为：/=
__ifloordiv__(self, other)	定义赋值整数除法的行为：//=
__imod__(self, other)	定义赋值取模算法的行为：%=
__ipow__(self, other[, modulo])	定义赋值幂运算的行为：**=
__ilshift__(self, other)	定义赋值按位左移位的行为：<<=
__irshift__(self, other)	定义赋值按位右移位的行为：>>=
__iand__(self, other)	定义赋值按位与操作的行为：&=
__ixor__(self, other)	定义赋值按位异或操作的行为：^=
__ior__(self, other)	定义赋值按位或操作的行为：|=
一元操作符	
__pos__(self)	定义正号的行为：+x
__neg__(self)	定义负号的行为：-x
__abs__(self)	定义当被 abs() 调用时的行为
__invert__(self)	定义按位求反的行为：~x
类型转换	
__complex__(self)	定义当被 complex() 调用时的行为（需要返回恰当的值）
__int__(self)	定义当被 int() 调用时的行为（需要返回恰当的值）

魔法方法	含义
__float__(self)	定义当被 float() 调用时的行为（需要返回恰当的值）
__round__(self[, n])	定义当被 round() 调用时的行为（需要返回恰当的值）
__index__(self)	1. 当对象是被应用在切片表达式中时，实现整形强制转换 2. 如果你定义了一个可能在切片时用到的定制的数值型,你应该定义 __index__ 3. 如果 __index__ 被定义，则 __int__ 也需要被定义，且返回相同的值
	上下文管理（with 语句）
__enter__(self)	1. 定义当使用 with 语句时的初始化行为 2. __enter__ 的返回值被 with 语句的目标或者 as 后的名字绑定
__exit__(self, exc_type, exc_value, traceback)	1. 定义当一个代码块被执行或者终止后上下文管理器应该做什么 2. 一般被用来处理异常，清除工作或者做一些代码块执行完毕之后的日常工作
容器类型	
__len__(self)	定义当被 len() 调用时的行为（返回容器中元素的个数）
__getitem__(self, key)	定义获取容器中指定元素的行为，相当于 self[key]
__setitem__(self, key, value)	定义设置容器中指定元素的行为，相当于 self[key] = value
__delitem__(self, key)	定义删除容器中指定元素的行为，相当于 del self[key]
__iter__(self)	定义当迭代容器中的元素的行为
__reversed__(self)	定义当被 reversed() 调用时的行为
__contains__(self, item)	定义当使用成员测试运算符（in 或 not in）时的行为

2.5 迭代器、推导式和生成器

2.5.1 迭代器

为什么学习迭代器?
作用：对于大数据量的访问，可以节省内存空间。
原理：迭代器不会一次性得到所有数据，而是循环一次计算/读取一次，相当于内存中始终只有一份数据
应用：我们学过的for循环，本质上就是将“可迭代对象”转化为“迭代器”，逐一访问元素的。

2.5.1.1 可迭代对象

概念：__iter__ 如果这个对象含有__iter__ 方法 我们就说它是可迭代对象

内置函数dir( )可以获取当前数据对象内置的方法和属性，例如：

```
t1 = [1,2,3,"a","b","b"]
print(dir(t1))
```

输出结果：

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

控制台如果不能显示多行结果，可以在控制台的左侧，点击左侧“Soft-wrap”(自动换行)按钮，显示全部内容。

我们可以观察到t1对象中是含有__iter__ 方法的，可以使用下面的语句更直接的得出相应结论：

```
print('__iter__' in dir(t1))        #输出：True
```

Python中的可迭代对象有：**列表、元组、字典、集合、字符串、文件。**

可迭代对象，可以遍历数据。 （遍历：逐一访问每个元素）

```
f = open("a.txt","r")            #a.txt文件中有abcd四个字母，每个字母占1行。
print("__iter__" in dir(f))      #输出：true

for i in f:
    print(i,end="")              #输出：abcd
```

2.5.1.2 迭代器

概念：如果同时含有__iter__ 和 __next__ 这两个方法,就说它是迭代器。

所以：
可迭代对象不一定是迭代器 （“可迭代对象”只含有__iter__方法）
迭代器一定是可迭代对象 （“迭代器”同时含有__iter__和__next__ 方法）

作用：可以通过next()控制元素访问的过程

for运行的底层原理：

> for 循环在遍历可迭代对象的时候：
> 第一件事是调用__iter__（）获得一个迭代器
> 第二件事是循环调用__next__ （）

（1）使用iter（）将可迭代对象变为迭代器

```
t1 = [1,2,3,"a","b","c"]
it = iter(t1)
# print(type(it))     #<class 'list_iterator'>
methods = dir(it)
print("__iter__" in methods and "__next__" in methods)      #输出：True
```

说明iter()函数可以将可迭代对象转变为迭代器。

类似的还可以有：

```
t2 =  (1,2,3)
it2 = iter(t2)
print(type(it2))         #输出：<class 'tuple_iterator'>

t3 =  {1,2,3}
it3 = iter(t3)
print(type(it3))         #输出：<class 'set_iterator'>

t4 =  {1:"a",2:"b",3:"c"}
it4 = iter(t4)
print(type(it4))         #输出：<class 'dict_keyiterator'>
```

（2）使用next()方法可以遍历迭代器

```
t1 = [1,2,3,"a","b","c"]
it = iter(t1)
res = next(it)       #每执行一次next(),就按照顺序访问下一个元素
print(res)           #输出：1
res = next(it)
print(res)           #输出：2
```

继续补充下列代码，直至报错：

```
res = next(it)
print(res)           #输出：3
res = next(it)
print(res)           #输出：a
res = next(it)
print(res)           #输出：b
res = next(it)
print(res)           #输出：c
res = next(it)       #访问到最后一个元素时，继续访问会报错
print(res)           #错误类型：StopIteration
```

为了防止访问过多元素从而产生错误，进而导致程序中断，我们可以使用异常处理，来辅助遍历：

```
t1 = [1,2,3,"a","b","c"]
it = iter(t1)
print("遍历开始")
while True:
    try:
        print(next(it))
    except StopIteration:
        print("遍历结束")
        break
```

遍历一遍后，想要重新遍历，直接再写一个循环，会发现没有输出任何元素内容。

```
print("遍历开始")
while True:
    try:
        print(next(it))
    except StopIteration:
        print("遍历结束")
        break
```

输出结果：

> 遍历开始
> 遍历结束

注意：迭代器的遍历是单向的

怎么可以从头访问呢？**重置迭代器。**

```
it = iter(t1)            #在前一次循环后，重新获取迭代器对象。就可以重新遍历元素了。
print("遍历开始")
while True:
    try:
        print(next(it))
    except StopIteration:
        print("遍历结束")
        break
```

其他遍历的方式：

```
t1 = [1,2,3,"a","b","c"]
#写法一：
it = iter(t1)
for i in range(len(t1)):          #使用for循环来遍历，可以不用异常处理，更方便。
    print(next(it))

#写法二：
it = iter(t1)
for i in range(len(t1)):
    print(next(it))
    # print(it.__next__())        #这个写法语法上支持，但不推荐
```

应用案例:

定义一个反转字符串的迭代器。输入：abcd　　输出：dcba

```
class Reverse:                      #因为同时拥有__iter__和 __next__，所以是迭代器

    def __init__(self,data):        #通过对象初始化传入参数，每个迭代器对象都是独立的，参数互不影响。
        self.data = data
        self.index = len(data)

    def __iter__(self):
        self.index = len(self.data) #每次重新获取迭代器对象，需要回复数据的索引，才能从头开始访问
        return self                 #返回自身对象

    def __next__(self):
        if self.index == 0:         #每次遍历访问下一个元素时，通过下标检查是否是第一个元素。
            raise StopIteration
        self.index = self.index - 1 #返回最后一个元素的下标
        return self.data[self.index]

res = Reverse("abcd")
methods =  dir(res)
# print("__iter__" in methods and "__next__" in methods)
for char in res:
    print(char,end="")
print()
print("-"*30)

for char in res:
    print(char,end="")
```

有其他方法，判断一个对象是否为可迭代对象/迭代器吗?

- Iterator 表示迭代器类型
- Iterable 表示可迭代对象

```
from collections.abc import Iterator,Iterable

t1 = [1,2,3,"a","b","c"]
res = isinstance(t1,Iterable)               #列表是一个可迭代对象
print(res)                                  #输出：True
res = isinstance(t1,Iterator)               #列表不是一个迭代器
print(res)                                  #输出：False
```

2.5.2 推导式

又称解析式，是Python的一种独有特性。
概念：推导式是可以从一个数据序列构建另一个新的数据序列的方式。
作用：简化代码
分类：

- 列表推导式
- 字典推导式
- 集合推导式

2.5.2.1 列表推导式

2.5.2.1.1 循环模式

通过for循环来配合表达式或函数，生成列表（序列）。

语法：*[变量表达式 或 有返回值的函数 for 变量 in iterable]*

练习：生成一个列表，包含1至10的平方

- 传统写法：

```
res = []
for i in range(1,11):
    res.append(i * i)
print(res)                  #输出：[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- 推导式写法1：应用“变量表达式”

```
res = [i * i for i in range(1,11) ]         # i * i 是变量表达式，i的值依次从后面的循环中取出
print(res)                  #输出：[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- 推导式写法2：应用“内置函数”

```
res = [pow(i,2) for i in range(1,11) ]      #pow(x,y)是内置函数， 表示x的y次幂
print(res)                  #输出：[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- 推导式写法3：应用“自定义函数（需要有返回值）”

```
def getSquare(x):
    return f"{x}的平方是{x**2}"     #自定义函数必须有返回值，返回值会成为列表中的元素

res = [getSquare(i) for i in range(1,11) ]
print(res)                  #输出：['1的平方是1', '2的平方是4', '3的平方是9', '4的平方是16', '5的平方是25', '6的平方是36', '7的平方是49', '8的平方是64', '9的平方是81', '10的平方是100']
```

> 上面这个题目中还有个细节：推导式中不仅仅能包含数字，也可以是字符串。确切的说，**列表推导式的结果就是列表**，表达式结果或函数返回值只要是列表能够包含的元素类型即可。

课堂练习：

构建一个列表，生成扑克牌：[2,3,4,5,6,7,8,9,"J","Q","K","A"]

参考答案：

```
poker = [i for i in range(2,10)] + list("JQKA")
print(poker)
```

2.5.2.1.2 筛选模式

在循环生成列表元素的过程中包含条件判断语句。

语法：*[变量表达式 或 有返回值的函数 for 变量 in iterable if 条件表达式]*

练习：30以内被3整除的数字

- 传统写法：

```
res = []
for i in range(0,31):
    if i % 3 == 0:
        res.append(i)
print(res)              #输出：[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

- 推导式写法：

```
res = [i for i in range(0,31) if i % 3 == 0]
print(res)              #输出：[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

> 建议：对照传统写法来理解推导式中每一部分的功能

2.5.2.1.3 多重循环

应用场景：会用在矩阵计算、结果筛选与匹配等场合

练习：打印九九表

- 传统写法

```
for i in range(1,10):
    for j in range(1,i+1):
        print(f"{i} * {j} = {i*j}",end="\t")
    print()
```

输出的结果为：

```
1 * 1 = 1
2 * 1 = 2 2 * 2 = 4
3 * 1 = 3 3 * 2 = 6 3 * 3 = 9
4 * 1 = 4 4 * 2 = 8 4 * 3 = 12  4 * 4 = 16
5 * 1 = 5 5 * 2 = 10  5 * 3 = 15  5 * 4 = 20  5 * 5 = 25
6 * 1 = 6 6 * 2 = 12  6 * 3 = 18  6 * 4 = 24  6 * 5 = 30  6 * 6 = 36
7 * 1 = 7 7 * 2 = 14  7 * 3 = 21  7 * 4 = 28  7 * 5 = 35  7 * 6 = 42  7 * 7 = 49
```

```
8 * 1 = 8 8 * 2 = 16  8 * 3 = 24  8 * 4 = 32  8 * 5 = 40  8 * 6 = 48  8 * 7 = 56  8 * 8 = 64
9 * 1 = 9 9 * 2 = 18  9 * 3 = 27  9 * 4 = 36  9 * 5 = 45  9 * 6 = 54  9 * 7 = 63  9 * 8 = 72  9 * 9 = 81
```

- 推导式写法1：

```
res = [f"{i} * {j} = {i*j}" for i in range(1,10) for j in range(1,i+1)]
print(res)
```

> 相对于传统的双层for循环，推导式仅仅是将原有的for循环嵌套写法，改为了一行代码连续书写。

输出的结果为：

```
['1 * 1 = 1', '2 * 1 = 2', '2 * 2 = 4', '3 * 1 = 3', '3 * 2 = 6', '3 * 3 = 9', '4 * 1 = 4', '4 * 2 = 8', '4 * 3 = 12', '4 * 4 = 16', '5 * 1 = 5', '5 * 2 = 10', '5 * 3 = 15', '5 * 4 = 20', '5 * 5 = 25', '6 * 1 = 6', '6 * 2 = 12', '6 * 3 = 18', '6 * 4 = 24', '6 * 5 = 30', '6 * 6 = 36', '7 * 1 = 7', '7 * 2 = 14', '7 * 3 = 21', '7 * 4 = 28', '7 * 5 = 35', '7 * 6 = 42', '7 * 7 = 49', '8 * 1 = 8', '8 * 2 = 16', '8 * 3 = 24', '8 * 4 = 32', '8 * 5 = 40', '8 * 6 = 48', '8 * 7 = 56', '8 * 8 = 64', '9 * 1 = 9', '9 * 2 = 18', '9 * 3 = 27', '9 * 4 = 36', '9 * 5 = 45', '9 * 6 = 54', '9 * 7 = 63', '9 * 8 = 72', '9 * 9 = 81']
```

相对于传统写法，虽然代码简单了，但是形式有缺失。主要原因在于：输出的仅仅是列表的元素（字符串）内容。

- 推导式写法2：

  可以通过自定义函数，将打印“格式”的语句包含到元素输出的过程中。

```
def get99(i,j):
    print(f"{i} * {j} = {i * j}", end="\t")
    if i == j:
        print()

res = [get99(i,j) for i in range(1,10) for j in range(1,i+1)]
```

输出的结果为：

```
1 * 1 = 1
2 * 1 = 2 2 * 2 = 4
3 * 1 = 3 3 * 2 = 6 3 * 3 = 9
4 * 1 = 4 4 * 2 = 8 4 * 3 = 12  4 * 4 = 16
5 * 1 = 5 5 * 2 = 10  5 * 3 = 15  5 * 4 = 20  5 * 5 = 25
6 * 1 = 6 6 * 2 = 12  6 * 3 = 18  6 * 4 = 24  6 * 5 = 30  6 * 6 = 36
7 * 1 = 7 7 * 2 = 14  7 * 3 = 21  7 * 4 = 28  7 * 5 = 35  7 * 6 = 42  7 * 7 = 49
8 * 1 = 8 8 * 2 = 16  8 * 3 = 24  8 * 4 = 32  8 * 5 = 40  8 * 6 = 48  8 * 7 = 56  8 * 8 = 64
9 * 1 = 9 9 * 2 = 18  9 * 3 = 27  9 * 4 = 36  9 * 5 = 45  9 * 6 = 54  9 * 7 = 63  9 * 8 = 72  9 * 9 = 81
```

2.5.2.2 字典推导式

字典推导式，使用方法和列表推导式基本相同。唯一的区别在于形成的序列是字典，元素的格式是“键值对”。

练习：将下面两个列表中的每个元素一一对应，形成键值对，保存到字典中。

```
city = ["北京","上海","广东","四川"]
abbr = ["京","沪","粤","川"]

dic = {city[i] : abbr[i] for i in range(len(city))}
print(dic)      #输出：{'北京': '京', '上海': '沪', '广东': '粤', '四川': '川'}
```

2.5.2.3 集合推导式

集合推导式，使用方法和列表推导式基本相同。唯一的区别在于形成的序列是集合。

练习：使用集合保存100以内所有的偶数。

```
print({i for i in range(2,101,2)})
```

输出的结果：

> {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100}

小结：

推导式的使用场景：

- 构建有规律的数据序列时
- 不建议三层及以上循环时使用

缺点：不方便调试

2.5.3 生成器

迭代器是Python解释器内部定义的数据结构。

生成器 （generator）本质就是迭代器，只是可以自定义数据结构和计算过程。

定义生成器有2种方式：

- 生成器函数
- 生成器表达式

2.5.3.1 生成器函数

概念：包含yield关键字的函数，就是生成器函数

```
def getNum():
    print("返回1")
    # return 1                #一般的函数
    yield 1                   #生成器函数

res = getNum()                #一般函数返回数字1；生成器函数返回了生成器对象 ： generator
print(res)                    #<generator object getNum at 0x000001F31DCA12E0>
```

可以验证生成器对象就是迭代器对象：

```
from collections.abc import Iterator
print(isinstance(res,Iterator))              #输出：True
```

既然是迭代器对象，就可以使用next方法来访问其中的元素：

```
print("运行了next:",next(res))               #输出：返回1              #说明getNum函数执行了
                                             #输出：运行了next: 1     #yield返回的结果输出了
#print("运行了next:",next(res))
```

如果再次调用上面的打印语句，会报错：StopIteration。

说明迭代器元素遍历到最后，没有可以返回的内容了。由此可以推断：**next函数调用依次，yield就执行一次**

2.5.3.1.1 yield

中文翻译为：产出 ，相当于返回每次迭代的结果值。

```
def getNum():
    print("返回1")
    yield 1

    print("返回2")
    yield 2

    print("返回3")
    yield 3

res = getNum()        #返回了生成器对象 ： generator
print("运行了next:",next(res))
print("运行了next:",next(res))
```

输出结果：

> 返回1
> 运行了next: 1
> 返回2
> 运行了next: 2

结论：**每执行一次next函数，yield执行一次，返回了一个迭代器元素，并记录了当前执行的yield的代码位置。**

既然res是迭代器对象，就可以直接使用for循环来遍历其中的全部元素，同时防止出现StopIteration。

```
for i in res:
    print("运行了next:", i)
```

输出结果：

```
返回1
运行了next: 1
返回2
运行了next: 2
返回3
运行了next: 3
```

有了上面的概念，就可以通过定义自己的生成器函数，实现自定义 元素生成过程 的迭代器了。

例题：生成指定数量的随机数。所有的随机数范围都在1-10之间。

```
import random

def getNum(length):                #定义一个生成器，length表示生成随机数的数量
    for i in range(length):
        yield f"当前的数值为：{random.randint(1,10)}"

num = getNum(10)          #通过设置生成器函数的参数，指定随机数个数
for i in range(10):
    print(next(num))      #每执行一次next，就生成一个[1,10]区间的随机数
print("-"*30)
num = getNum(5)           #生成5个随机数，但是内存中始终只有1个数值
for i in range(5):
    print(next(num))      #访问的时候，才产生这个随机数，能够节省内存空间
```

输出结果为：

```
当前的数值为：8
当前的数值为：8
当前的数值为：4
当前的数值为：1
当前的数值为：10
当前的数值为：7
当前的数值为：2
当前的数值为：9
当前的数值为：10
当前的数值为：1
```

```
当前的数值为：9
当前的数值为：9
当前的数值为：1
当前的数值为：3
当前的数值为：9
```

2.5.3.1.2 send

send可以在外界获取下一个生成器元素时，传入指定参数来影响生成器的元素产生过程。

```
def getResult(x):

    print("接收到x:",x)
    y = yield x * x          #在第一次执行完yield返回结果后，代码会停留在这里
                             #直到send函数从外界传入一个数值，相当于执行给y赋值的功能，代码
继续执行
    print("接收到y:", y)
    z = yield y ** 3

    print("接收到z:", z)
    z = yield z + 1000

res = getResult(2)
print("第一次访问：",next(res))        #第一次访问时，需要使用next函数来获取第一个元素
print("第二次访问：",res.send(2))
```

输出结果：

```
接收到x: 2
第一次访问： 4
接收到y: 2
第二次访问： 8
```

如果第一获取元素的时候使用send方法，必须参数是None；否则会报TypeError。

```
def getResult(x):

    print("接收到x:",x)
    y = yield x * x

    print("接收到y:", y)
    z = yield y ** 3

    print("接收到z:", z)
    z = yield z + 1000

res = getResult(2)
# print("第一次访问：",res.send(2))       #报错：TypeError: can't send non-None value
to a just-started generator
print("第一次访问：",res.send(None))     #第一次发送的时候必须参数是None
print("第二次访问：",res.send(10))       #可以向生成器中传入数值，默认再调用next得到下一个
元素
```

```
print("第三次访问：",res.send(50))
```

输出结果：

```
接收到x: 2
第一次访问： 4
接收到y: 10
第二次访问： 1000
接收到z: 50
第三次访问： 1050
```

练习：得到指定数字的平方

```
def getSquare(num):                #自定义生成器函数
    x = num                        #
    while True:
        x = yield x ** 2

gs = getSquare(1)
# print(next(gs))                  #写法一
print(gs.send(None))               #写法二
print(gs.send(10))
print(gs.send(40))
```

输出结果:

```
1
100
1600
```

2.5.3.1.3 yield from

在外界访问生成器时，我们想要将生成器函数中的列表元素，逐一返回给外界。但下面的结果不是我们想要的：

```
def getNum():
    t = [1,2,3,4]
    yield t                #直接返回的是列表对象

res = getNum()
print(next(res))           #输出：[1, 2, 3, 4]
```

这个时候我们需要使用yield from ， 从可迭代对象中产生数据：逐一返回每个元素

```
def getNum():
    t = [1,2,3,4]
    yield from t

res = getNum()
for i in res:
    print(i,end=" ")        #输出：1 2 3 4
```

2.5.3.2 生成器表达式

与推导式的写法基本相同，也有循环模式、筛选模式、多重循环构建。
只有一点不同：不是 [ ]，不是{ } ,而是 **( )**

相对于看起来像“元组推导式”（不存在这个概念）的意思，但它所生成的结果不是序列对象，而是**生成器对象**。

```
g = (i * 2 for i in range(5))
# print(g)          #输出的不是元组，而是：<generator object <genexpr> at
0x00000143948C12E0>
for i in g:
    print(i,end=" ")        #遍历生成器对象，依次输出0 2 4 6 8
```