# 程序结构

## 模块 Module

### 定义

包含一系列数据、函数、类的文件，通常以.py结尾。

### 作用

让一些相关的数据，函数，类有逻辑的组织在一起，使逻辑结构更加清晰。

有利于多人合作开发。

### 导入

#### import

1. 语法：

import 模块名

import 模块名 as 别名

1. 作用：将某模块整体导入到当前模块中
2. 使用：模块名.成员

#### from import

1. 语法：

from 模块名 import 成员名[ as 别名1]

1. 作用：将模块内的一个或多个成员导入到当前模块的作用域中。
2. 使用：直接使用成员名

##### from import \*

1. 语法：from 模块名 import \*
2. 作用：将某模块的所有成员导入到当前模块。

练习1：

创建2个模块module\_exercise.py与exercise.py

将下列代码粘贴到module\_exercise模块中，并在exercise中调用。

data = 100

def func01():

print("func01执行喽")

class MyClass:

def func02(self):

print("func02执行喽")

@classmethod

def func03(cls):

print("func03执行喽")

练习2：将信息管理系统拆分为4个模块student\_info\_manager\_system.py

（1）创建目录student\_info\_manager\_system

（2）创建模块bll,存储XXController

业务逻辑层 business logic layer

（3）创建模块usl,存储XXView

用户显示层 user show layer

（4）创建模块model,存储XXModel

（5）创建模块main,存储调用XXView的代码

### 模块变量

\_\_doc\_\_变量：文档字符串。

\_\_name\_\_变量：模块自身名字，可以判断是否为主模块。

当此模块作为主模块(第一个运行的模块)运行时，\_\_name\_\_绑定'\_\_main\_\_'，不是主模块，而是被其它模块导入时,存储模块名。

### 加载过程

在模块导入时，模块的所有语句会执行。

如果一个模块已经导入，则再次导入时不会重新执行模块内的语句。

### 分类

1. 内置模块(builtins)，在解析器的内部可以直接使用。
2. 标准库模块，安装Python时已安装且可直接使用。
3. 第三方模块（通常为开源），需要自己安装。
4. 用户自己编写的模块（可以作为其他人的第三方模块）

练习1：定义函数,根据年月日,计算星期。

输入：2020 9 15

输出：星期二

练习2：定义函数,根据生日(年月日),计算活了多天.

输入：2010 1 1

输出：从2010年1月1日到现在总共活了3910天

## 包package

### 定义

将模块以文件夹的形式进行分组管理。

### 作用

让一些相关的模块组织在一起，使逻辑结构更加清晰。

练习1：

1. 根据下列结构，创建包与模块。

my\_project01 /

main.py

common/

\_\_init\_\_.py

list\_helper.py

skill\_system/

\_\_init\_\_.py

skill\_deployer.py

skill\_manager.py

1. 在main.py中调用skill\_manager.py中实例方法。
2. 在skill\_manager.py中调用skill\_deployer.py中实例方法。
3. 在skill\_deployer.py中调用list\_helper.py中类方法。

练习2：

1. 根据下列结构，创建包与模块。

my\_project 02/

main.py

common/

\_\_init\_\_.py

list\_helper.py

skill\_system/

\_\_init\_\_.py

skill\_manager.py

1. 通过导入包的方式，在main.py中调用skill\_manager.py中实例方法。
2. 通过导入包的方式，在skill\_manager.py中调用list\_helper.py中类方法。

### \_\_init\_\_.py 文件

是包内必须存在的文件

会在包加载时被自动调用

#### 作用

记录import 包 语句需要导入的模块

### 搜索顺序

sys.path 提供的路径

# 异常处理Error

## 异常

1. 定义：运行时检测到的错误。
2. 现象：当异常发生时，程序不会再向下执行，而转到函数的调用语句。
3. 常见异常类型：

-- 名称异常(NameError)：变量未定义。

-- 类型异常(TypeError)：不同类型数据进行运算。

-- 索引异常(IndexError)：超出索引范围。

-- 属性异常(AttributeError)：对象没有对应名称的属性。

-- 键异常(KeyError)：没有对应名称的键。

-- 异常基类Exception。

## 处理

1. 语法：

try:

可能触发异常的语句

except 错误类型1 [as 变量1]：

处理语句1

except 错误类型2 [as 变量2]：

处理语句2

except Exception [as 变量3]：

不是以上错误类型的处理语句

else:

未发生异常的语句

finally:

无论是否发生异常的语句

1. 作用：将程序由异常状态转为正常流程。
2. 说明：

as 子句是用于绑定错误对象的变量，可以省略

except子句可以有一个或多个，用来捕获某种类型的错误。

else子句最多只能有一个。

finally子句最多只能有一个，如果没有except子句，必须存在。

如果异常没有被捕获到，会向上层(调用处)继续传递，直到程序终止运行。

练习：创建函数，在终端中录入int类型成绩。如果格式不正确，重新输入。

效果： score = get\_score()

print("成绩是：%d"%score)

## raise 语句

1. 作用：抛出一个错误，让程序进入异常状态。
2. 目的：在程序调用层数较深时，向主调函数传递错误信息要层层return 比较麻烦，所以人为抛出异常，可以直接传递错误信息。。

# 迭代

每一次对过程的重复称为一次“迭代”，而每一次迭代得到的结果会作为下一次迭代的初始值。例如：循环获取容器中的元素。

## 可迭代对象iterable

1. 定义：具有\_\_iter\_\_函数的对象，可以返回迭代器对象。
2. 语法

-- 创建：

class 可迭代对象名称:

  def \_\_iter\_\_(self):

      return 迭代器

-- 使用：

for 变量名 in 可迭代对象:

语句

1. 原理：

迭代器 = 可迭代对象.\_\_iter\_\_()

while True:

try:

print(迭代器.\_\_next\_\_())

except StopIteration:

break

练习1：创建列表,使用迭代思想,打印每个元素.

练习2：创建字典,使用迭代思想,打印每个键值对.

## 迭代器对象iterator

1. 定义：可以被next()函数调用并返回下一个值的对象。
2. 语法

class 迭代器类名:

def \_\_init\_\_(self, 聚合对象):

self.聚合对象= 聚合对象

def \_\_next\_\_(self):

if 没有元素:

raise StopIteration

return 聚合对象元素

1. 说明：

-- 聚合对象通常是容器对象。

4. 作用：使用者只需通过一种方式，便可简洁明了的获取聚合对象中各个元素，而又无需了解其内部结构。

练习1：遍历商品控制器

class CommodityController:

pass

controller = CommodityController()

controller.add\_commodity("屠龙刀")

controller.add\_commodity("倚天剑")

controller.add\_commodity("芭比娃娃")

for item in controller:

print(item)

练习2：遍历图形控制器

class GraphicController:

pass

controller = CommodityController()

controller.add\_graphic("圆形")

controller.add\_graphic("矩形")

controller.add\_graphic ("三角形")

for item in controller:

print(item)

练习3：创建自定义range类，实现下列效果.

class MyRange:

pass

for number in MyRange(5):

print(number)# 0 1 2 3 4

# 生成器generator

1. 定义：能够动态(循环一次计算一次返回一次)提供数据的可迭代对象。
2. 作用：在循环过程中，按照某种算法推算数据，不必创建容器存储完整的结果，从而节省内存空间。数据量越大，优势越明显。
3. 以上作用也称之为延迟操作或惰性操作，通俗的讲就是在需要的时候才计算结果，而不是一次构建出所有结果。

## 生成器函数

1. 定义：含有yield语句的函数，返回值为生成器对象。
2. 语法

-- 创建：

def 函数名():

…

yield 数据

…

-- 调用：

for 变量名 in 函数名():

语句

1. 说明：

-- 调用生成器函数将返回一个生成器对象，不执行函数体。

-- yield翻译为”产生”或”生成”

1. 执行过程：
2. 调用生成器函数会自动创建迭代器对象。
3. 调用迭代器对象的\_\_next\_\_()方法时才执行生成器函数。
4. 每次执行到yield语句时返回数据，暂时离开。
5. 待下次调用\_\_next\_\_()方法时继续从离开处继续执行。
6. 原理：生成迭代器对象的大致规则如下

-- 将yield关键字以前的代码放在next方法中。

-- 将yield关键字后面的数据作为next方法的返回值。

练习1：定义函数,在列表中找出所有偶数

[43,43,54,56,76,87,98]

练习2. 定义函数,在列表中找出所有数字

[43,"悟空",True,56,"八戒",87.5,98]

## 内置生成器

### 枚举函数enumerate

1. 语法：

for 变量 in enumerate(可迭代对象):

语句

for 索引, 元素in enumerate(可迭代对象):

语句

1. 作用：遍历可迭代对象时，可以将索引与元素组合为一个元组。

练习1：将列表中所有奇数设置为None

练习2：将列表中所有偶数自增1

### zip

1. 语法：

for item in zip(可迭代对象1, 可迭代对象2….):

语句

1. 作用：将多个可迭代对象中对应的元素组合成一个个元组，生成的元组个数由最小的可迭代对象决定。

练习：使用学生列表封装以下三个列表中数据

list\_student\_name = ["悟空", "八戒", "白骨精"]

list\_student\_age = [28, 25, 36]

list\_student\_sex = ["男", "男", "女"]

## 生成器表达式

1. 定义：用推导式形式创建生成器对象。
2. 语法：变量 = ( 表达式 for 变量 in 可迭代对象 [if 真值表达式] )

练习1：使用生成器表达式在列表中获取所有字符串.

list01 = [43, "a", 5, True, 6, 7, 89, 9, "b"]

练习2：在列表中获取所有整数,并计算它的平方.

# 函数式编程

1. 定义：用一系列函数解决问题。

-- 函数可以赋值给变量，赋值后变量绑定函数。

-- 允许将函数作为参数传入另一个函数。

-- 允许函数返回一个函数。

2. 高阶函数：将函数作为参数或返回值的函数。

## 函数作为参数

将核心逻辑传入方法体，使该方法的适用性更广，体现了面向对象的开闭原则。

练习1：

需求：

定义函数，在列表中查找奇数

定义函数，在列表中查找能被3或5整除的数字

步骤：

1. 根据需求，写出函数。

2. 因为主体逻辑相同,核心算法不同.

所以使用函数式编程思想(分、隔、做)

创建通用函数find\_all

3. 在当前模块中调用

练习2：

需求：

定义函数，在员工列表中查找编号是1003的员工

定义函数，在员工列表中查找姓名是孙悟空的员工

步骤：

1. 根据需求，写出函数。

2. 因为主体逻辑相同,核心算法不同.

所以使用函数式编程思想(分、隔、做)

创建通用函数find\_single

3. 在当前模块中调用

class Employee:

def \_\_init\_\_(self, eid, did, name, money):

self.eid = eid # 员工编号

self.did = did # 部门编号

self.name = name

self.money = money

# 员工列表

list\_employees = [

Employee(1001, 9002, "师父", 60000),

Employee(1002, 9001, "孙悟空", 50000),

Employee(1003, 9002, "猪八戒", 20000),

Employee(1004, 9001, "沙僧", 30000),

Employee(1005, 9001, "小白龙", 15000),

]

### lambda 表达式

1. 定义：是一种匿名方法。
2. 作用：作为参数传递时语法简洁，优雅，代码可读性强。

随时创建和销毁，减少程序耦合度。

1. 语法

-- 定义：

变量 = lambda 形参: 方法体

-- 调用：

变量(实参)

1. 说明：

-- 形参没有可以不填

-- 方法体只能有一条语句，且不支持赋值语句。

### 内置高阶函数

1. map（函数，可迭代对象）：使用可迭代对象中的每个元素调用函数，将返回值作为新可迭代对象元素；返回值为新可迭代对象。
2. filter(函数，可迭代对象)：根据条件筛选可迭代对象中的元素，返回值为新可迭代对象。
3. sorted(可迭代对象，key = 函数,reverse = bool值)：排序，返回值为排序结果。
4. max(可迭代对象，key = 函数)：根据函数获取可迭代对象的最大值。
5. min(可迭代对象，key = 函数)：根据函数获取可迭代对象的最小值。

练习：

1. 在商品列表，获取所有名称与单价

2. 在商品列表中，获取所有单价小于10000的商品

3. 对商品列表，根据单价进行降序排列

4. ([1,1],[2,2,2],[3,3,3])

获取元组中长度最大的列表

class Commodity:

def \_\_init\_\_(self, cid=0, name="", price=0):

self.cid = cid

self.name = name

self.price = price

list\_commodity\_infos = [

Commodity(1001, "屠龙刀", 10000),

Commodity(1002, "倚天剑", 10000),

Commodity(1003, "金箍棒", 52100),

Commodity(1004, "口罩", 20),

Commodity(1005, "酒精", 30),

]

## 函数作为返回值

逻辑连续，当内部函数被调用时，不脱离当前的逻辑。

### 闭包

1. 三要素：

-- 必须有一个内嵌函数。

-- 内嵌函数必须引用外部函数中变量。

-- 外部函数返回值必须是内嵌函数。

1. 语法

-- 定义：

def 外部函数名(参数):

外部变量

def 内部函数名(参数):

使用外部变量

return 内部函数名

-- 调用：

变量 = 外部函数名(参数)

变量(参数)

1. 定义：是由函数及其相关的引用环境组合而成的实体。
2. 优点：内部函数可以使用外部变量。
3. 缺点：外部变量一直存在于内存中，不会在调用结束后释放，占用内存。
4. 作用：实现python装饰器。

练习：使用闭包模拟以下情景：

在银行开户存入10000

购买xx商品花了xx元

购买xx商品花了xx元

### 函数装饰器decorator

1. 定义：在不改变原函数的调用以及内部代码情况下，为其添加新功能的函数。
2. 语法

def 函数装饰器名称(func):

def wrapper(\*args, \*\*kwargs):

需要添加的新功能

return func(\*args, \*\*kwargs)

return wrapper

原函数 = 内嵌函数

@ 函数装饰器名称

def 原函数名称(参数):

函数体

原函数(参数)

1. 本质：使用“@函数装饰器名称”修饰原函数，等同于创建与原函数名称相同的变量，关联内嵌函数；故调用原函数时执行内嵌函数。

原函数名称 = 函数装饰器名称（原函数名称）

1. 装饰器链：

一个函数可以被多个装饰器修饰，执行顺序为从近到远。

练习1：不改变插入函数与删除函数代码，为其增加验证权限的功能

def verify\_permissions(func):

print("验证权限")

def insert():

print("插入")

def delete():

print("删除")

insert()

delete()

练习2：为sum\_data,增加打印函数执行时间的功能.

函数执行时间公式： 执行后时间 - 执行前时间

def sum\_data(n):

sum\_value = 0

for number in range(n):

sum\_value += number

return sum\_value

print(sum\_data(10))

print(sum\_data(1000000))