# 模块Module

## 定义

包含一系列代码(数据/函数/类/语句)的文件，通常以.py结尾。

## 作用

让一些相关的代码，有逻辑的组织在一起，使结构更加清晰。

有利于团队开发。

## 导入

### import

1. 语法：import 模块名 [as 别名]
2. 作用：将某个模块整体导入到当前模块
3. 本质：使用变量名(模块名)关联指定模块代码。
4. 使用：模块名.成员

### from import

1. 语法：from 模块名 import 成员 [as 别名]
2. 作用：将模块内的指定的成员导入到当前模块作用域中。
3. 使用：成员

### from import \*

1. 语法：from 模块名 import \*
2. 作用：将模块内的所有成员导入到当前模块作用域中。
3. 注意：

-- 小心重名

-- 不能导入隐藏成员(以一个下划线开头)。

### pycharm 设置项目根目录

文件夹右键-->Mark Directory as --> Sources Root

## 模块变量

1. \_\_name\_\_ 属性：是当前模块名称；

注意：如果程序从当前模块运行，则名称被改为 \_\_main\_\_

作用：在if代码块中定义只能从当前模块开始执行，才调用的代码。

if \_\_name\_\_ == "\_\_main\_\_":

测试代码

1. \_\_file\_\_属性：当前模块的文件路径。

练习：将学生管理系统代码分成多个模块；

模块互相导入，可以正常运行。

ui.py ---> 存放StudentManagerView

bll.py ---> 存放StudentManagerController

models.py ---> 存放StudentModel

main.py ---> 存放调用学生管理器的代码。

导入方式：import 模块名

from 模块名 import 成员

from 模块名 import \*

## 加载过程

在模块导入时，模块的所有语句都会执行。

如果一个模块已经导入，则再次导入时不再重复执行语句。

## 分类

1. 内置模块(builtins):解释器可以直接使用。
2. 标准库模块：安装python时自带的，经过导入后可直接使用。
3. 第三方模块：需要自行下载，再导入使用。
4. 自定义模块：经过导入后可直接使用。

# 包package

## python程序结构

包

模块

语句

函数

类

方法

## 定义

将模块以文件夹的形式进行分组管理。

必须包含\_\_init\_\_.py

## 作用

让一些相关的模块有逻辑的组织在一起。

## 导入

import 包名[as 别名]

import 包名.子包名[as 别名]

import 包名.模块[as 别名]

from 包 import 模块[as 别名]

from 包.子包 import 模块[as 别名]

from 包 import \*

-- 必须在 \_\_init\_\_.py文件中，定义\_\_all\_\_ = [“模块名”]

## 搜索顺序

sys.path 提供的路径

第一个路径是运行模块所在路径

如果希望从项目其他模块运行，那么必须在pycharm中设置项目根目录。

练习:

my\_project/

\_\_init\_\_.py

main.py

common/

\_\_init\_\_.py

double\_list\_helper.py

list\_helper.py

sql\_helper.py

skill\_system/

\_\_init\_\_.py

skill\_data.py

skill\_deployer.py

实现：

1. main 调用 list\_helper/ skill\_deployer
2. skill\_deployer调用skill\_data/ list\_helper/ double\_list\_helper
3. 要求：模块中必须包含类，必须包含后两种导包方式

code18/my\_project

# 异常处理

代码：day18/code03

## 异常

1. 定义：运行时检测到的错误。
2. 现象：当异常发生时，程序不再向下继续执行，而转到函数调用的语句。

直到遇到异常处理语句。

1. 常见异常：

-- IndexError：索引越界

-- ValueError：值错误

-- NameError：变量未定义

-- AttributeError:对象没有相应属性

-- NotImplementedError: 未实现异常

-- KeyError：没有对应的键

## 处理

1. 语法：

try:

可能发生错误的代码

except 异常类型1 [as 变量名]:

处理逻辑1

except 异常类型2 [as 变量名]:

处理逻辑2

else:

未发生异常执行的代码

finally:

无论是否异常，一定执行的代码

1. 作用：将程序由异常状态转换为正常流程
2. 说明：

-- as 字句可有可无，as 后面的变量绑定的是错误对象。

练习：

定义获取成绩的方法。

要求：如果输入错误，重新录入。

1. --- 100

代码：code18/exercise03

练习：将学生管理系统UI层可能出错的逻辑，进行异常操作。

要求：哪里出错，解决哪里，然后按照既定流程向后执行.

## raise

作用：抛出一个错误，让程序进入异常状态。

目的：在程序调用层数较深时，向主程序传递错误信息要层层return比较麻烦，所以建议使用人为抛出异常。

## 自定义异常类

代码：code18/code04

1. 作用：封装需要传递的错误信息。
2. 定义：

class 类名Error(Exception):

def \_\_init\_\_(self,参数):

self.错误信息 = 参数

1. 使用：

-- 在需要传递错误信息的代码，编写：raise 类名Error(错误信息)

-- 在需要接受错误信息的代码，编写：

try:

代码….

except 类名Error as e:

e就是传递的错误信息对象

练习：定义学生类，通过自定义异常，

返回成绩(0--100)错误信息(code,message)。

代码：code18/exercise04

# 迭代

每一次对过程的重复称之为”迭代一次”，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

代码day18/code01

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 可迭代对象类名:

def \_\_iter\_\_(self):

return 迭代器

-- 使用：

for 变量名 in 可迭代对象:

语句

1. 原理：

变量 = 可迭代对象. \_\_iter\_\_()

while True:

try:

元素 = 变量.\_\_next\_\_()

except:

break

## 迭代器iterator

代码day18/code02

1. 定义：具有\_\_next\_\_函数的对象，可以获取下一个元素。
2. 语法：

class 迭代器名称:

def \_\_init\_\_(self,参数):

self.数据 = 参数 # 数据就是聚合对象

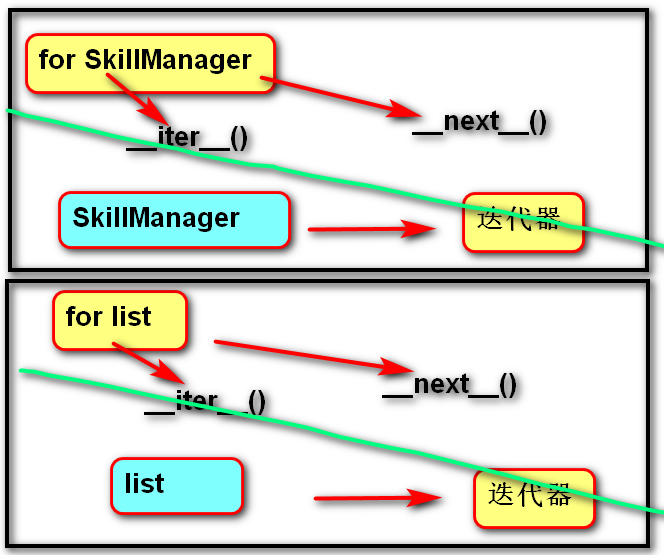
def \_\_next\_\_():

if 没有元素:

raise StopIteration() # 停止迭代

return 数据中的元素 #聚合对象的元素

1. 作用：使用者只需要通过一种方式(\_\_next\_\_)，便可简单明了的获取聚合对象中的各个元素，无需了解其内部结构。



练习：参照下列代码，自定义MyRange类，实现以下效果。

for item in range(5):

print(item)# 0 1 2 3 4

for item in MyRange(5):

print(item)# 0 1 2 3 4

代码：day18/exercise01

# 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. 惰性(延迟)操作：通俗的讲在需要的时候才计算结果。

## 生成器函数

1. 定义：含有yield关键字的函数，返回值是生成器对象。
2. 语法：

-- 创建

def 方法名():

…

yield 数据

…

-- 使用

for 变量名 in 方法名():

语句

1. \*本质：

-- 创建

class 生成器名称:

def \_\_iter\_\_(self):

return 迭代器对象

-- 使用

生成器变量名 = 生成器名称()

迭代器变量名 =变量名.\_\_iter\_\_()

while True:

try:

print(迭代器变量名.\_\_next\_\_())

except:

break

1. 语法说明：

-- 调用生成器函数返回迭代器对象，不执行函数体。

-- 执行过程：

(1) 调用生成器函数，自动创建迭代器对象。

(2) 调用迭代器对象的\_\_next\_\_()方法才执行生成器函数体。

(3) 每次执行到yield关键字时返回数据，暂时离开。

(4) 待下次调用\_\_next\_\_()方法继续执行。

1. 生成迭代器对象原理：
2. 将yield关键字以前的代码放到\_\_next\_\_()方法中。
3. 将yield关键字以后的数据作为\_\_next\_\_()方法返回值。

## 生成器表达式

代码：day18/code06

1. 语法：(表达式 for 变量 in 可迭代对象 [if 条件])
2. 定义：用推导式语法创建的生成器对象。

# 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。

### 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。

## 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：

在一个函数内部的函数,同时内部函数又引用了外部函数的变量。

1. 本质：

闭包是将内部函数和外部函数的执行环境绑定在一起的对象。

1. 优点

-- 内部函数可以使用外部变量。

1. 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。

1. 作用：

实现python装饰器。

### 函数装饰器decorators

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。

1. 语法

def 函数装饰器名称(func):

def 内嵌函数(\*args, \*\*kwargs):

需要添加的新功能

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

return wrapper

@ 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：

原函数名称 = 函数装饰器名称（原函数名称）

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。

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。

# 文件操作

文件：保存在持久化存储设备(硬盘、U盘、光盘..)上的一段数据。

从功能角度分类：文本文件（打开后会自动解码为字符）、二进制文件(视频、音频等)。

从数据角度分类：所有文件都是由字节(Byte)组成。

操作步骤：打开、读写、关闭。

## 对文件进行文本操作

1. 语法：

with open(文件名, "操作方式") as 变量名:

变量名.write("字符")

变量名. writelines (字符串列表)

变量名. readline () # 读取一行

变量名. readlines () # 读取全部行

for 变量 in 变量名: # 迭代读取每行

语句

2. 操作模式：只读r 只写 w 追加 a

## 对文件进行二进制操作

1. 字节串bytes：

-- 定义：由一系列字节组成的不可变序列容器。

-- 字节：8bit(位)组成的整数，十进制在0—255之间。

-- 字面值：b”abc” --> 97 98 99

bytes() --> 空字节串

-- 字符串转换为字节串：字节串 = 字符串.encode(encoding=”utf-8”)

-- 字节串转换为字符串：字符串 = 字节串. decode(encoding=”utf-8”)

2. 语法：

with open(文件名, "操作方式") as 变量名:

变量名.write(字节串)

变量名.read(字节数) # 读取指定字节

变量名.read() # 读取全部字节

变量名. tell () # 获取文件指针位置

变量名. seek(偏移量) # 设置文件指针位置

3. 操作模式：只读rb 只写 wb 读写r+b