# 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]

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。

### 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。
3. 模块中以下划线(\_)开头的属性，不会被导入，通常称这些成员为隐藏成员。

# 方式1  
# 语法:import 模块  
# 本质:使用变量module01存储该模块地址  
# 优点:不用担心成员冲突问题  
# 使用:需要通过变量访问  
import module01  
  
module01.fun01()  
  
import module01 as m  
  
m.fun01()  
  
# 方式2  
# 本质:将模块指定成员导入到当前模块作用域中  
# from 模块 import 成员  
# 使用:直接使用导入的成员  
# 缺点:可能造成成员冲突  
from module01 import fun01  
from module01 import MyClass01  
  
  
def fun01():  
 print("demo01--fun01")  
  
  
fun01()  
  
c01 = MyClass01()  
c01.fun02()  
  
# 方式3  
# from 模块 import \*  
# 优点:可以一次导入多个成员,避免一个一个导入.  
# 缺点: 代码可读性不高,命名冲突几率更大.  
from module01 import \*  
  
# from module02 import \*  
# from module03 import \*  
fun01()  
  
c01 = MyClass01()  
c01.fun02()

**模块变量**

\_\_all\_\_变量：定义可导出成员，仅对from xx import \*语句有效。

\_\_doc\_\_变量：文档字符串。查看当前模块的文档注释

\_\_file\_\_变量：查看当前模块对应的文件路径名。

\_\_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。

*"""  
 模块相关概念  
"""*from module05 import \*  
  
fun01()  
# 1. 隐藏模块成员  
# \_fun02()# 不能访问隐藏成员  
# fun03()# 不能访问\_\_all\_\_以外的成员  
  
# from module05 import \_fun02  
# \_fun02()# 能访问隐藏成员  
  
# from module05 import fun03  
# fun03() # 能访问\_\_all\_\_以外的成员  
  
# 2. 查看当前模块的文档注释  
print(\_\_doc\_\_)  
  
# 3. 查看当前模块的文件路径  
# /home/tarena/1906/month01/code/day14/demo03.py  
print(\_\_file\_\_)  
  
# 4.查看当前模块名称 --- 判断当前模块是否为主模块  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("是主模块(程序从当前模块开始执行)")

## 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。

只有第一次导入时加载到内存空间里。

## 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）
5. 内置模块time

*"""*

*标准库模块*

*时间* *time*

*练习:exercise01.py*

*练习:exercise02.py*

*"""*

import time

# 1. 获取当前时间戳(从1970年1月1日到现在经过的秒数)

# 1563326340.661173

print(time.time())

# 2. 获取当前时间元组(年,月,日,小时,分钟,秒,一周的第几天,一年的第几天,夏令时)

time\_tuple = time.localtime()

print(time\_tuple)

# 3. 时间戳 -->时间元组

print(time.localtime(15633263))

# 4. 时间元组 -- 时间戳

print(time.mktime(time\_tuple))

# 5. 时间元组 --> 字符串

print(time.strftime("%y/%m/%d %H:%M:%S",time\_tuple))

print(time.strftime("%Y/%m/%d %H:%M:%S",time\_tuple))

# 6. 字符串 --> 时间元组

# "19/07/17 09:36:48"

print(time.strptime("19/07/17 09:36:48","%y/%m/%d %H:%M:%S"))

## 搜索顺序

搜索内建模块(builtins)

sys.path 提供的路径，通常第一个是程序运行时的路径。

在pycharm里设置主目录可以直接运行导包，但在终端里默认第一个执行的程序为主程序，会导致导包失败。可以在sys.path中手动添加主目录路径后在终端成功运行。

# 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。

## 导入

import 包名 [as 包别名] 需要设置\_\_all\_\_

import 包名.模块名 [as 模块新名]

import 包名.子包名.模块名 [as 模块新名]

from 包名 import 模块名 [as 模块新名]

from 包名.子包名 import 模块名 [as 模块新名]

from 包名.子包名.模块名 import 成员名 [as 属性新名]

# 导入包内的所有子包和模块

from 包名 import \*

from 包名.模块名 import \*

## 搜索顺序

sys.path 提供的路径

## \_\_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

### \_\_all\_\_

记录from 包 import \* 语句需要导入的模块

案例：

my\_ project /

main.py

common/

\_\_init\_\_.py

double\_list\_helper.py

list\_helper.py

skill\_system/

\_\_init\_\_.py

skill\_deployer.py

skill\_manager.py

# 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。

-- 为实现异常(NotImplementedError)：尚未实现的方法。

-- 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

*异常处理*

def div\_apple(apple\_count):

*"""*

*分苹果*

*"""*

person\_count = int(input("请输入人数:"))# ValueError

result = apple\_count / person\_count# ZeroDivisionError

print("每个人分到了%d个苹果"%result)

"""

try:

div\_apple(10)

except:

# 错误的处理逻辑

print("出错喽")

"""

"""

try:

div\_apple(10)

except ValueError:

print("输入的人数必须是整数")

except ZeroDivisionError:

print("输入的人数不能是零")

except Exception:

# 错误的处理逻辑

print("出错喽")

else:

print("没有错误执行的代码")

"""

try:

# 如果有错误

div\_apple(10)

finally:

# 不处理错误,但有一件非常重要的事情,必须执行.

print("一定执行的代码")

print("后续逻辑.........")

# 练习:定义函数,在控制台中获取成绩.

# 要求:如果异常,则继续获取成绩,直到正确为止.

1. 作用：将程序由异常状态转为正常流程。程序不会停止运行，会执行后续的语句
2. 说明：

as 子句是用于绑定错误对象的变量，可以省略，接收错误信息

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。

## 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。

## 自定义异常

1. 定义：

class 类名Error(Exception):

def \_\_init\_\_(self,参数):

super().\_\_init\_\_(参数)

self.数据 = 参数

1. 调用：

try:

….

raise 自定义异常类名(参数)

….

except 定义异常类 as 变量名:

变量名.数据

1. 作用：封装错误信息

# 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

能for循环的对象必须拥有\_\_iter\_\_()方法

1. 原理：

迭代器 = 可迭代对象.\_\_iter\_\_()

while True:

try:

print(迭代器.\_\_next\_\_())

except StopIteration:

break

## 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。

# 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”

通过yield将函数分割为多个\_\_next\_\_()方法

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

-- 第一次运行时将yield关键字以前的代码放在next方法中。

后面每一次从程序离开处继续执行

-- 将yield关键字后面的数据作为next方法的返回值。

## 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。

### 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 。

## 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] )

# 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

实际作用： 将lambda表达式作为函数实参传递。

### 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。（返回生成器对象）
3. sorted(可迭代对象，key = 函数,reverse = bool值)：排序，返回值为排序结果。生成新的可迭代对象
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。

外部函数调用完不释放内存，内部函数可以继续使用

### 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。
2. 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。
3. 优点：内部函数可以使用外部变量。
4. 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。
5. 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。
2. 用途：新功能不是原函数永久需求的功能，只是某些时候用 ---🡪用装饰器
3. 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(拦截原函数)

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。