# 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。

**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。

\_\_file\_\_变量：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。

## 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

Day 13

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。

time模块

1. 第三方模块（通常为开源），需要自己安装。
2. 用户自己编写的模块（可以作为其他人的第三方模块）

## 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。

Day 19 14.20

# 包package

## 定义

Day14 am

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

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] 推荐这种方式，这样就可以直接使用方法名或者类名，就好像两个模块在一个文件里一样

# 导入包内的所有子包和模块

from 包名 import \*

from 包名.模块名 import \* 这个与上面第三个类似

#导入最好是从项目根目录开始导入，不管两个模块是不是在一个包里

## 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

一般可以在包的\_\_init\_\_.py文件中来写可以控制包里的哪些成员可以被调用

案例：

my\_ project /

main.py

common/

\_\_init\_\_.py

double\_list\_helper.py

list\_helper.py

sill\_system/

\_\_init\_\_.py

skill\_data.py

skill\_deployer.py

skill\_manager.py

# 异常处理Error

## 异常

Day 14 15.00

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。

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

-- 异常基类Exception。

语法错误是错误，不是异常无法处理，只能改代码

## 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 多个except ，else 语句只会执行一个

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。

## raise 语句

Day 14 17.18

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。。

（看面向对象继承例子代码中使用）

## 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

Day 15

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

Day 15 11.00

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

## 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。

注意 for 遍历的本质

iterator = list01.\_\_iter\_\_()

while True:

try:# 如果获取了全部元素,则执行except

# 2.　获取下一个元素(迭代过程)

item = iterator.\_\_next\_\_() ===》for item in list01:

print(item) print(item)

# 3.停止迭代（StopIteration　错误）

except StopIteration:

break # 跳出循环体

可迭代对象 有\_\_iter\_\_() 方法， 可以返回一个迭代器对象

迭代器对象 有\_\_next\_\_() 方法 ，返回可迭代对象的元素

# 生成器generator

Day 15 15.30 生成器源码

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。（函数内部的循环次数和外部遍历的for循环次数相同，外面遍历一次里面计算一次）
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。

存储的是算法，而不是结果

## 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。

ListHelper 函数

1. 语法

-- 创建：

def 函数名():

…

yield 数据

…

-- 调用：

for 变量名 in 函数名():

语句

方法体中 yield 语句都在for 或者while 循环体中使 yield 语句多次操作。每次执行到yield语句都会将结果返回出去，并暂时退出函数，等到函数外面的for循环再次执行到函数时，就会从上次的yield语句的下一步开始新运转

def fun():

for I in range(5):

yiedl I

===> def fun():  
 list=[]

for I in range(5)

return list

上面一个fun()得到一个生产器对象（是可迭代对象的一种）可以通过for循环遍 历，或者list() tuple() 等容器构造函数转换为其他类型的对象。里面为0,1,2,3,4，

存储的是逻辑方法。内存空间占用很小。

下面的fun()得到的是一个可迭代对象。包含的元素为0,1,2,3,4 。

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。

## 内置生成器

### 枚举函数enumerate

Day 16 10.30

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

enumerate(可迭代对象) 返回一个元素为（索引，元素）的可迭代对象

这里的索引是元素被取出的顺序。可以修改初始值

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。

类似与生产器都是会返回一个可迭代对象

## 生成器表达式

Day 16 am

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] )==》等同于生成器函数

# 函数式编程

Day 16 pm

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。

Def fun():

pass

a=fun

a() 就是执行fun（）可以用变量名加小括号（）来调用函数。意味着任意在函数中

形参（） 的格式可以代表任何函数方法 根据传入的实参 来确定函数执行时的真实操作，类似于，类的继承和多态，

def fun(a):

a()

如果某个fun 函数中表达式只要改变部分代码，如if 判断的条件，就可以实现不同的功能的话，我们就可以通过将这个改变的地方，替换成一个抽象的函数表达式，通过传入不同的a 来实现不同的功能。a 参数就想父类一样被不同的函数替换

## 函数作为参数 回调函数

将（函数）‘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

Day 16 pm 16.00

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

实例： （list , lamada x : 2x ） x为 list里的元素 ，lamada 表达式返回 每个list 元素 x 的2倍值 2x

Def double（list）：

For i in list：

Yield 2i

-- 调用：

变量(实参) 执行结果为 return 方法体

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。

当函数体比较简单且只运用一次时，可以用这种格式替换

### 内置高阶函数

Day 17 ListHelper 里面都有

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(过滤)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

Day 18 am

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（如果修改外部变量就要执行 nonlocal 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

Day 18 am

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 内嵌函数(\*args, \*\*kwargs): 这是为达到要求定义的新函数。将函数

需要添加的新功能 赋值给func名，来达到函数名不变

return func(\*args, \*\*kwargs) 功能增加的结果

return wrapper(内嵌函数名，就用这个单词也行)

@ 函数装饰器名称

def 原函数名称(参数):

函数体

调用：

原函数(参数)

**d**ef print\_func\_name(func):

# 包装新旧功能

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

# 增加的新功能

print(func.\_\_name\_\_)

# 旧功能

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

return wrapper # 返回包装器

@print\_func\_name # say\_hello = print\_func\_name(say\_hello)

def say\_hello(name):

print(name,"hello")

return "哈哈"

@print\_func\_name

def say\_goodbye(name,age):

print(age,name,"goodbye")

#---------以上是定义者--以下是调用者-----------------

print(say\_hello("张无忌"))

say\_goodbye("赵敏",25)

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）

编程步鄹：

1.接到需要修改的函数，和需要增加的功能

2.编写函数装饰器代码

3.在原函数上加上@函数装饰器名。

4.测试

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。

**拓展学习，Python描述符协议，类的装饰器，微软LINQ框架**