1 程序结构

1.1 模块 Module

1.1.1 定义

包含一系列数据、函数、类的文件，通常以.py结尾。

1.1.2 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。

1.1.3 导入

1.1.3.1 import

(1) 语法：

import 模块名

import 模块名 as 别名

(2) 作用：将模块整体导入到当前模块中

(3) 使用：模块名.成员

1.1.3.2 from import

(1) 语法：

from 模块名 import 成员名

from 模块名 import 成员名 as 别名

from 模块名 import *

(2) 作用：将模块内的成员导入到当前模块作用域中

(3) 使用：直接使用成员名

```
"""
    module01.py
"""

def func01():
    print("module01 - func01执行喽")


def func02():
    print("module01 - func02执行喽")
```

```
# 导入方式1: import 模块名
# 使用: 模块名.成员
# 原理：创建变量名记录文件地址,使用时通过变量名访问文件中成员
```

```python
# 备注："我过去"
# 适用性：适合面向过程(全局变量、函数)
import module01

module01.func01()

# 导入方式2.1：from 文件名 import 成员
# 使用：直接使用成员
# 原理：将模块的成员加入到当前模块作用域中
# 备注："你过来"
# 注意：命名冲突
# 适用性：适合面向对象(类)

from module01 import func01

def func01():
    print("demo01 - func01")

func01() # 调用的是自己的func01


# 导入方式2.2：from 文件名 import *
from module01 import *

func01()
func02()
```

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。

```python
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

1.1.4 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。

1.1.5 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。

1.1.6 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

1.2 包package

1.2.1 定义

将模块以文件夹的形式进行分组管理。

1.2.2 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) 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

(2) 在main.py中调用skill_manager.py中实例方法。

(3) 在skill_manager.py中调用skill_deployer.py中实例方法。

(4) 在skill_deployer.py中调用list_helper.py中类方法。

1.2.3 导入

1.2.3.1 import

(1) 语法：

import 包

import 包 as 别名

(2) 作用：将包中__init__模块内整体导入到当前模块中

(3) 使用：包.成员

1.2.3.2 from import

(1) 语法：

from 包 import 成员

from 包 import 成员 as 别名

(2) 作用：将包中__init__模块内的成员导入到当前模块作用域中

(3) 使用：直接使用成员名

(4) 演示：

目录结构：

my_project/

main.py

package01/

__init__.py

module01.py

```
"""
    package01/
        module01.py
"""
def func01():
    print("func01执行了")

def func02():
    print("func02执行了")
```

```
"""
    main.py
"""
# 方式1:import 包 as 别名
import package01 as p

p.module01.func01()
p.func02()

# 方式2:from 包 import 成员
from package01 import module01,func02

module01.func01()
func02()
```

```
"""
    package01/
        __init__.py
"""
import package01.module01

from package01.module01 import func01
```

练习：

(1) 根据下列结构，创建包与模块。

my_project02/

main.py

common/

__init__.py

list_helper.py

skill_system/

__init__.py

skill_manager.py

(2) 通过导入包的方式，在main.py中调用skill_manager.py中实例方法。

(3) 通过导入包的方式，在skill_manager.py中调用list_helper.py中类方法。

2 异常处理Error

2.1 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。

2.2 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
```

(2) 作用：将程序由异常状态转为正常流程。

(3) 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)

2.3 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。

```python
class Wife:
    def __init__(self, age):
        self.age = age

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, value):
        if 20 <= value <= 60:
            self.__age = value
        else:
            # 创建异常  -- 抛出 错误信息
            raise Exception("我不要","if 20 <= value <= 60",1001)

# -- 接收 错误信息
while True:
    try:
        age = int(input("请输入你老婆年龄："))
        w01 = Wife(age)
        break
    except Exception as e:
        print(e.args) # ('我不要', 'if 30 <= value <= 60', 1001)
```

3 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。

3.1 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

```python
# 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
```

(3) 原理：

```
迭代器 = 可迭代对象.__iter__()
while True:
    try:
        print(迭代器.__next__())
    except StopIteration:
        break
```

(4) 演示:

```
message = "我是花果山水帘洞孙悟空"
# for item in message:
#     print(item)

# 1. 获取迭代器对象
iterator = message.__iter__()
# 2. 获取下一个元素
while True:
    try:
        item = iterator.__next__()
        print(item)
        # 3. 如果停止迭代则跳出循环
    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.

3.2 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
```

(3) 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。

(5) 演示:

```
class StudentIterator:
    def __init__(self, data):
        self.__data = data
        self.__index = -1
```

```
    def __next__(self):
        if self.__index == len(self.__data) - 1:
            raise StopIteration()
        self.__index += 1
        return self.__data[self.__index]


class StudentController:
    def __init__(self):
        self.__students = []

    def add_student(self, stu):
        self.__students.append(stu)

    def __iter__(self):
        return StudentIterator(self.__students)

controller = StudentController()
controller.add_student("悟空")
controller.add_student("八戒")
controller.add_student("唐僧")

# for item in controller:
#     print(item) #
iterator = controller.__iter__()
while True:
    try:
        item = iterator.__next__()
        print(item)  #
    except StopIteration:
        break
```

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
```

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。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。

4.1 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
```

(3) 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"

(4) 执行过程：

a. 调用生成器函数会自动创建迭代器对象。

b. 调用迭代器对象的**next**()方法时才执行生成器函数。

c. 每次执行到yield语句时返回数据，暂时离开。

d. 待下次调用**next**()方法时继续从离开处继续执行。

(5) 原理：生成迭代器对象的大致规则如下

a. 将yield关键字以前的代码放在next方法中。

b. 将yield关键字后面的数据作为next方法的返回值。

(6) 演示：

```
def my_range(stop):
    number = 0
    while number < stop:
        yield number
        number += 1

for number in my_range(5):
    print(number)  # 0 1 2 3 4
```

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]

4.2 内置生成器

4.2.1 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
```

(2) 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。

(3) 演示：

```
list01 = [43, 43, 54, 56, 76]
# 从头到尾读           -- 读取数据
for item in list01:
    print(item)

# 非从头到尾读         -- 修改数据
for i in range(len(list01)):
    if list01[i] % 2 == 0:
        list01[i] += 1

for i, item in enumerate(list01):  # -- 读写数据
    if item % 2 == 0:
        list01[i] += 1
```

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

4.2.2 zip

(1) 语法：

```
for item in zip(可迭代对象1, 可迭代对象2):
    语句
```

(2) 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。

(3) 演示：

```
list_name = ["悟空", "八戒", "沙僧"]
list_age = [22, 26, 25]

# for 变量 in zip(可迭代对象1,可迭代对象2)
for item in zip(list_name, list_age):
    print(item)
# ('悟空', 22)
# ('八戒', 26)
# ('沙僧', 25)

# 应用:矩阵转置
map = [
    [2, 0, 0, 2],
    [4, 2, 0, 2],
    [2, 4, 2, 4],
    [0, 4, 0, 4]
]
# new_map = []
# for item in zip(map[0],map[1],map[2],map[3]):
#     new_map.append(list(item))
# print(new_map)

# new_map = []
# for item in zip(*map):
#     new_map.append(list(item))

new_map = [list(item) for item in zip(*map)]
print(new_map)
# [[2, 4, 2, 0], [0, 2, 4, 4], [0, 0, 2, 0], [2, 2, 4, 4]]
```

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"]

4.3 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 条件)
```

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.

5 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。

5.1 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。

```
list01 = [342, 4, 54, 56, 6776]

# 定义函数,在列表中查找第一个大于100的数
def get_number_gt_100():
    for number in list01:
        if number > 100:
            return number


# 定义函数,在列表中查找第一个偶数
def get_number_by_even():
    for number in list01:
        if number % 2 == 0:
            return number

# 参数：得到的是列表中的元素
# 返回值：对列表元素判断后的结果(True False)
def condition01(number):
    return number > 100

def condition02(number):
    return number % 2 == 0

# 通用函数
def find_single(condition): # 抽象
    for item in list01:
        # if number > 100:
        # if condition01(item):
        # if condition02(item):
        if condition(item):# 统一
            return item

```

```python
# 变化点函数：查找小于10的数据
def condition03(number):
    return number < 10

print(find_single(condition03))
```

练习1：

需求：

定义函数，在列表中查找第一个奇数

定义函数，在列表中查找第一个能被3或5整除的数字

步骤：

-- 根据需求，写出函数。

-- 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

-- 在当前模块中调用

练习2：

需求：

定义函数，在员工列表中查找所有编号是1003的员工

定义函数，在员工列表中查找所有姓名是孙悟空的员工

步骤：

-- 根据需求，写出函数。

-- 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

-- 在当前模块中调用

```python
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

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),
]
```

5.1.1 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。

(3) 语法

```
# 定义：
变量 = lambda 形参: 方法体

# 调用：
变量(实参)
```

(4) 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。

(5) 演示：

```
from common.iterable_tools import IterableHelper

# 定义函数,在列表中查找所有大于100的数
# def condition01(number):
#     return number > 100

# 定义函数,在列表中查找所有偶数
# def condition02(number):
#     return number % 2 == 0

list01 = [342, 4, 54, 56, 6776]

for item in IterableHelper.find_all(list01,lambda number: number > 100):
    print(item)

for item in IterableHelper.find_all(list01,lambda number: number % 2 == 0):
    print(item)
```

5.1.2 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。

(6) 演示：

```python
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),
]

# 1. map 映射
# 需求:获取所有员工姓名
for item in map(lambda item: item.name, list_employees):
    print(item)

# 2. filter 过滤器
# 需求：查找所有部门是9002的员工
for item in filter(lambda item: item.did == 9002, list_employees):
    print(item.__dict__)

# 3. max min 最值
emp = max(list_employees, key=lambda emp: emp.money)
print(emp.__dict__)

# 4. sorted
# 升序排列
new_list = sorted(list_employees, key=lambda emp: emp.money)
print(new_list)

# 降序排列
new_list = sorted(list_employees, key=lambda emp: emp.money, reverse=True)
print(new_list)

```

练习：

-- 在商品列表，获取所有名称与单价

-- 在商品列表中，获取所有单价小于10000的商品

-- 对商品列表，根据单价进行降序排列

-- 获取元组中长度最大的列表（[1,1],[2,2,2],[3,3,3]）

```python
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),
]
```

5.2 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。

5.2.1 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。

(2) 语法

```python
# 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)
```

(3) 定义：是由函数及其相关的引用环境组合而成的实体。

(4) 优点：内部函数可以使用外部变量。

(5) 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。

(6) 作用：实现python装饰器。

(7) 演示：

```python
def give_gife_money(money):
    print("获得", money, "元压岁钱")
    def child_buy(commodity, price):
        nonlocal money
        money -= price
        print("购买了", commodity, "花了", price, "元,还剩下", money)
    return child_buy

action = give_gife_money(500)
action("变形金刚", 200)
action("芭比娃娃", 300)
```

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

5.2.2 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

```python
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

@ 函数装饰器名称
def 原函数名称(参数):
    函数体

原函数(参数)
```

(3) 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）

```python
def func01():
    print("旧功能")


def new_func(func):
    def wrapper():
        print("新功能")
        func() # 执行旧功能

    return wrapper


# 新功能覆盖了旧功能
# func01 = new_func
```

```
15
16 # 调用一次外部函数(装饰器本质)
17 func01 = new_func(func01)
18 # 调用多次内部函数
19 func01()
20 func01()
```

(3) 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

```
1  def verify_permissions():
2      print("验证权限")
3
4  def insert():
5      print("插入")
6
7  def delete():
8      print("删除")
9
10
11 insert()
12 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

```
1 def sum_data(n):
2     sum_value = 0
3     for number in range(n):
4         sum_value += number
5     return sum_value
6
7 print(sum_data(10))
8 print(sum_data(1000000))
```