

1. 程序结构

1.1 模块 Module

1.1.1 定义

一个python文件，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 导入模块

1. import

(1) 语法：

```
1  import 模块名
2
3  import 模块名 as 别名
4
5  # 调用
6  模块名.成员
```

(2) 作用：将模块整体导入到当前模块中

2. from import

(1) 语法：

```
1  from 模块名 import 成员名
2
3  from 模块名 import 成员名 as 别名
4
5  from 模块名 import *
6
7  # 调用
8  直接使用成员名(变量，函数，class)
```

(2) 作用：将模块内的成员导入到当前模块作用域中

示例

```

1  """
2      module01.py
3  """
4
5  def func01():
6      print("module01 - func01执行喽")
7
8
9  def func02():
10     print("module01 - func02执行喽")

```

```

1  """
2      demo01.py
3  """
4
5  # 导入方式1: import 模块名
6  # 使用: 模块名.成员
7  # 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
8  # 备注: "我过去"
9  # 适用性: 适合面向过程(全局变量、函数)
10 import module01
11
12 module01.func01()
13
14 # 导入方式2.1: from 文件名 import 成员
15 # 使用: 直接使用成员
16 # 原理: 将模块的成员加入到当前模块作用域中
17 # 备注: "你过来"
18 # 注意: 命名冲突
19 # 适用性: 适合面向对象(类)
20
21 from module01 import func01
22
23 def func01():
24     print("demo01 - func01")
25
26 func01() # 调用的是自己的func01
27
28
29 # 导入方式2.2: from 文件名 import *
30 from module01 import *
31
32 func01()
33 func02()

```

练习1: 创建2个模块module_exercise.py与exercise03.py

将下列代码粘贴到module_exercise模块中,并在exercise中调用。

```

1  '''
2  module_exercise.py

```

```

3     '''
4
5     data = 100
6
7     def func01():
8         print("func01执行喽")
9
10    class MyClass:
11        def func02(self):
12            print("func02执行喽")
13
14        @classmethod
15        def func03(cls):
16            print("func03执行喽")

```

```

1     '''
2     exercise.py
3     '''
4
5     # 1.适合面向过程
6     import module_exercise
7
8     module_exercise.data = 10
9     module_exercise.func01()
10
11    # 调用类方法使用类名
12    module_exercise.MyClass.func03()
13    m = module_exercise.MyClass()
14    # 调用实例方法使用对象
15    m.func02() # func02(m)
16
17
18    # 2.适合面向对象
19    from module_exercise import *
20
21    print(data)
22    func01()
23    MyClass.func03()
24    m = MyClass()
25    m.func02()
26
27    print(__name__)

```

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

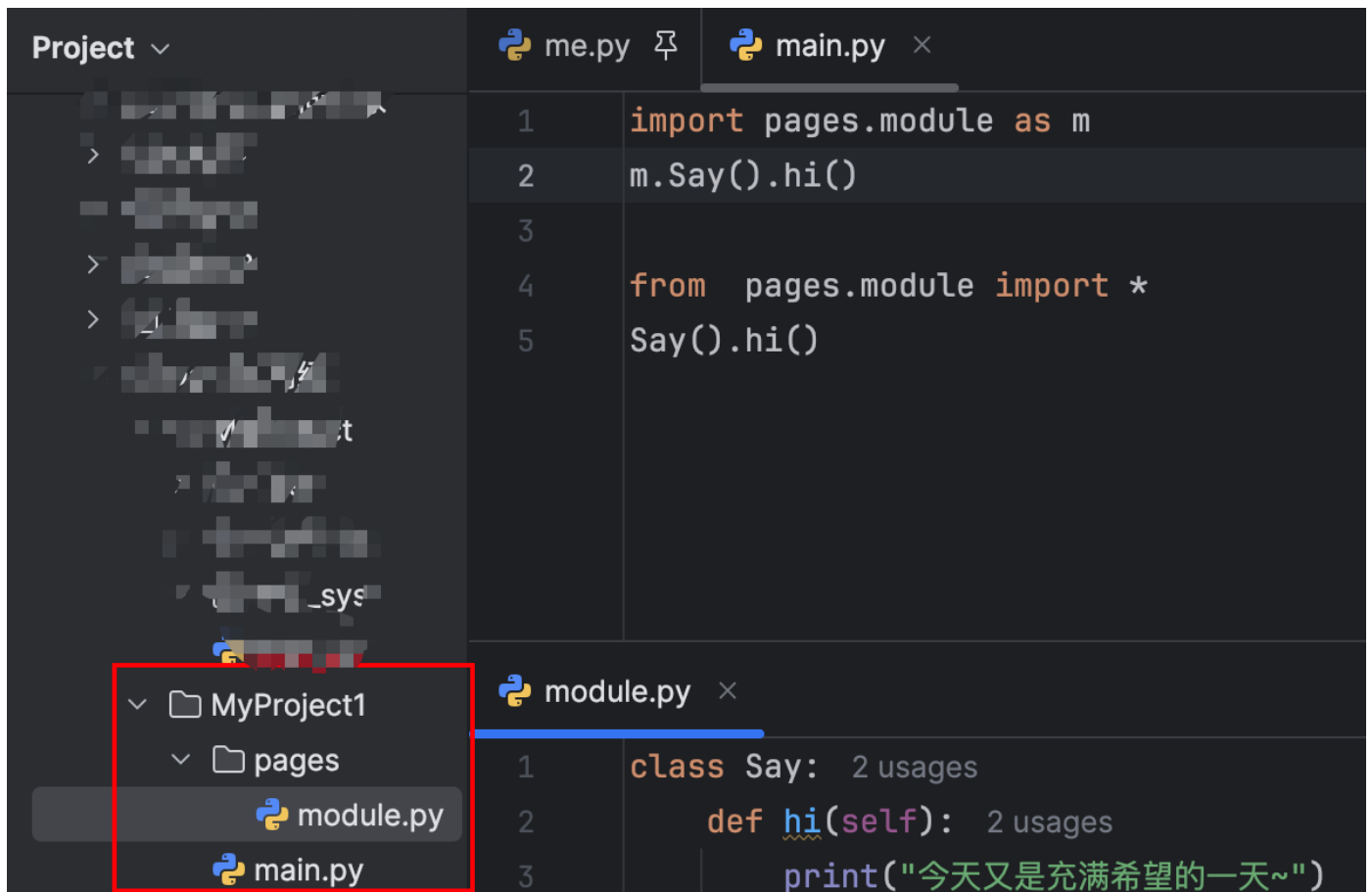
(3) 创建模块 `usl` ,存储XXView

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

模块在包里，导入



1.1.4 模块变量

`__doc__`变量：文档字符串(每个文件最上面的说明)。

`__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 模块分类

- (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）

1.时间模块

- datetime:转换为时间，或者转换成string
- timedelta:操作修改时间

```
1  """
2  时间模块
3  """
4  from datetime import datetime, timedelta
5
6  # 1.表达时间
7  # 现在的时间
8  d1 = datetime.now()
9  # 设定时间 年 月 日 时 分 秒
10 d2 = datetime(2000, 8, 8, 15, 15, 15)
11 print(d1)
12 print(d2)
13
14 # 2.时间的计算
15 delta1 = d1 - d2
16 print(delta1)
17 print(delta1.total_seconds()) # total_seconds 总秒数
18 print(delta1.days()) # days 差多少天
19
20 # 改1个值
21 # 方法1: timedelta 表示18天的时间差 (timedelta对象)
22 print(d2 + timedelta(days=18))
23 # 方法2: 直接计算得到值,有可能越界! 解决方法如: 练习4
24 print(d2.replace(day=d2.day + 10))
25
26
27 # 3.获取
28 print(d2.year)
29 print(d2.month)
30 print(d2.day)
31 print(d2.hour)
32 print(d2.minute)
```

```

33 print(d2.second)
34 print(d2.date())
35 print(d2.weekday()) # 返回的是周几的索引 周一是索引0
36
37 # 4. 格式化
38 print(d2.strftime("%Y年%m月%d日 %H时%M分%S秒")) # display
39 print(d2.strptime("2000年08月08日 15时15分15秒", "%Y年%m月%d日 %H时%M分%S秒")) # 转为时间

```

练习1: 定义函数,根据年月日,计算星期几

```

1 """
2     练习1: 定义函数,根据年月日,计算星期。
3     输入: 2020  9  15
4     输出: 星期二
5 """
6 # 方法1
7 import time
8
9 def get_week_name(year, month, day):
10     # year, month, day --> 字符串
11     str_time = f"{year}-{month}-{day}"
12     # str_time = "%s-%s-%s" % (year, month, day)
13     # 字符串 --> 时间元组
14     tuple_time = time.strptime(str_time, "%Y-%m-%d")
15     # 时间元组 --> 星期数
16     week_index = tuple_time[-3]
17     # 星期数 --> 星期名
18     tuple_week = ("星期一", "星期二", "星期三", "星期四", "星期五", "星期六", "星期日")
19     return tuple_week[week_index]
20
21
22 print(get_week_name(2022, 1, 30))

```

```

1 # 方法2
2 """
3     定义一个函数 输入一个年月日 计算那天是星期几
4     2024, 8, 17  星期六
5 """
6 from datetime import datetime
7
8
9 def calc_weekday(year, month, day):
10     date = datetime(year, month, day)
11     week_index = date.weekday()
12     week_list = ["星期一", "星期二", "星期三", "星期四", "星期五", "星期六", "星期日"]
13     return week_list[week_index]
14
15
16 print(calc_weekday(2024, 8, 17))

```

练习2: 定义函数,根据生日(年月日),计算活了多少天

```
1  """
2  定义一个函数 输入一个年月日  计算生活了多少天
3  2010, 1, 1    5281天
4  """
5
6  from datetime import datetime
7
8
9  def calc_days(year, month, day):
10     date = datetime(year, month, day)
11     times = datetime.now() - date
12     return times.days
13
14  print(calc_days(2010, 1, 1))
15  print(int((datetime.now().timestamp()-datetime(2010,1,1).timestamp())/(24*60*60)))
16
17
18  # 时间对象 <--> 时间戳
19  timestamp = datetime.now().timestamp()
20  date = datetime.fromtimestamp(timestamp)
21  print(timestamp)
22  print(date)
```

练习3: 定义一个函数接受开始日期和总天数作为参数 计算这段时间内所有工作日总时长, 假设一天8小时

```
1  """
2  定义一个函数接受开始日期和总天数作为参数
3  计算这段时间内所有工作日总时长, 假设一天8小时
4  2024 7 1    10    小时 ==>    64
5  """
6  from datetime import datetime, timedelta
7
8
9  def calc_hours(date_str, days):
10     start_date = datetime.strptime(date_str, "%Y-%m-%d")
11     total_hours = 0
12
13     for _ in range(days):
14         if start_date.weekday() < 5:
15             total_hours += 8
16             start_date += timedelta(days=1)
17     return total_hours
18
19
20  print(calc_hours("2024-07-01", 10))
```

练习4: 返回第5个月后的日期

返回第5个天后的日期

```
1  """
2  声明一个函数，接受一个日期和一个数字N作为参数
3  返回第N个月后的日期
4  提示：
5  如果是使用windows的同学需要下载，友情提示，会遇到报错....
6  原因： 系统问题 、网络问题 、权限的问题
7  pip3 install python-dateutil
8  """
9  from datetime import datetime
10 from dateutil.relativedelta import relativedelta
11
12
13 def calc_date(date_str, month):
14     date = datetime.strptime(date_str, "%Y-%m-%d")
15     return date + relativedelta(months=month)
16     # return date + relativedelta(days=day)
17
18 print(calc_date("2024-8-17", 5))
```

练习5: 计算距离下一个周一还有多少天

```
1  """
2  声明一个函数，接受一个日期作为参数， 计算距离下一个周一还有多少天
3  例如： 2024-8-17  2天
4
5  """
6  from datetime import datetime
7
8
9  def calc_next_monday(date_str):
10     date = datetime.strptime(date_str, "%Y-%m-%d")
11     return (7 - date.weekday())
12
13 print(calc_next_monday("2024-08-17"))
14 print(calc_next_monday("2024-08-12"))
15 print(calc_next_monday("2024-08-19"))
16 print(calc_next_monday("2024-08-20"))
```


2.可视化模块

```
1  """
2  pip3 install matplotlib
3  折线图
4  """
5  import matplotlib.pyplot as plt
6
7  x = [1, 2, 3, 4, 5]
8  y = [1, 4, 9, 16, 25]
9
10 # plot折线图, 要求横纵坐标 (可迭代对象, NumPy数组)
11 plt.plot(x,y)
12 plt.title("my plot")
13 plt.xlabel("x")
14 plt.ylabel("y")
15 # 显示图表
16 plt.show()
```

```
1  """
2  饼状图
3  """
4  import matplotlib.pyplot as plt
5  # 1.准备数据
6  # 饼图的面积大小
7  size = [25,58,95,10,30]
8  labels = ["c++", "php", "python", "go", "Harmonyos"]
9  colors = ['gold', 'green', 'pink', 'cyan', 'purple']
10 # 是否突出显示 0.1 扇形和饼图的中心距离一定百分比
11 explode = (0.1,0,0,0,0)
12
13 # 2. 绘制
14 plt.pie(size,explode=explode,labels=labels,colors=colors)
15 plt.axis("equal")
16
17 # 3.显示
18 plt.show()
```

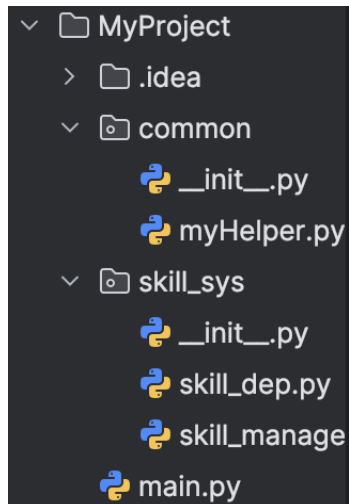
```
1  """
2  条形图
3  """
4  import matplotlib.pyplot as plt
5
6  # 1.准备数据
7  categories = ["java", "PHP", "c++", "Python"]
8  values = [35,59,78,20]
9
10 # 2. 绘制
11 plt.bar(categories,values)
12 plt.title("Bar")
13 plt.xlabel("categories")
```

```
14 plt.ylabel("values")
15
16 # 3.显示
17 plt.show()
```

1.1.7 模块练习

MyProject

(1) 根据下列结构，创建包与模块。



(2) 在main.py中调用skill_manager.py中实例方法。

```
1 # 主入口
2 from skill_sys.skill_manager import SkillManager
3
4 m = SkillManager()
5 m.f2()
```

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

```
1 # 引入SkillDep类
2 from .skill_dep import SkillDep
3
4 class SkillManager:
5     def f2(self):
6         print("我是SkillManager类的f2")
7
8 dep = SkillDep()
9 dep.f1()
```

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

```
1 # 引入myhelp类
2 from common.myHelper import MyHelper
3
4 class SkillDep:
5     def f1(self):
6         print("我是SkillDep的f1")
7
8 help1 = MyHelper()
9 help1.say()
```

(5) myHelper.py

```
1 # 提供给整个项目用的
2 class MyHelper:
3     def say(self):
4         print("我是MyHelper的帮助类")
```

1.2 包package

1.2.1 定义

package: 将模块放入文件夹，进行分组管理。并且有`_init_.py`

标蓝：表示项目根目录

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。

1.2.3 导入包

注意：先mark as sources root,从根目录开始写

1.2.3.1 import

(1) 语法：

```
1 import 包
2 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:

```
1  from 包 import 成员文件
2
3  from 包 import 成员文件 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

练习1: myProject1

目录结构:

```
myProject1 /
  main.py
  package01/
    __init__.py
    module01.py
```

```
1  """
2      package01/
3          module01.py
4  """
5  def func01():
6      print("func01执行了")
7
8  def func02():
9      print("func02执行了")
```

```
1  """
2      main.py
3  """
4  # 方式1:import 包 as 别名
5  import package01 as p
6
7  p.module01.func01()
8  p.func02()
9
10 # 方式2:from 包 import 成员
11 from package01 import module01, func02
12
13 module01.func01()
14 func02()
```

```

1  """
2      package01/
3          __init__.py
4  """
5  import package01.module01
6
7  from package01.module01 import func02

```

练习2： 根据下列结构，创建包与模块。

MyProject02/

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。

```

1  # 根目录/skill_sys/__init__.py
2  # 向外指明需要的类
3  from .skill_manager import SkillManager

```

```

1  # 根目录/main.py
2  # from 包 import 需要的类
3
4  # 无需这么写: from skill_sys.skill_manager import SkillManager
5  from skill_sys import SkillManager
6
7  if __name__ == '__main__':
8      manager=SkillManager()
9      manager.func01()

```

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

1.2.3.3 __init__

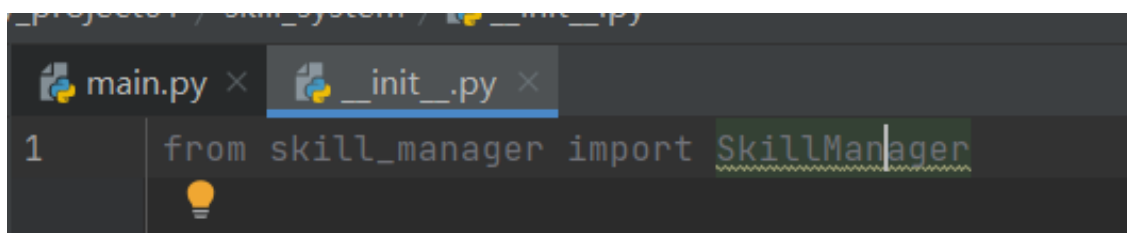
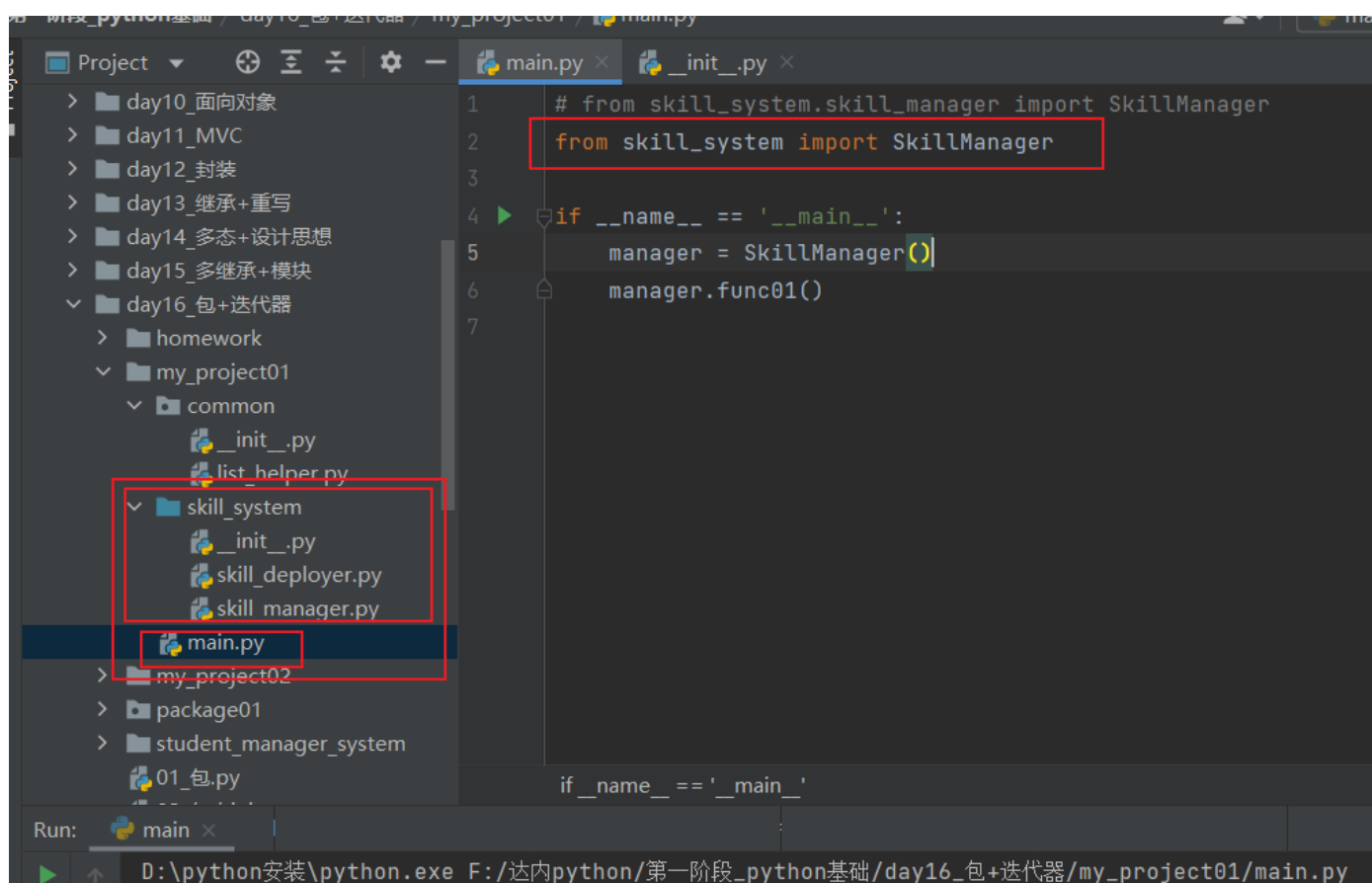
包中init模块的作用： 在包第一次被导入时执行，决定了能导入哪些东西(类/函数)

何时: 自己写，对外提供时，用到init

```
1 '''
2 __init__文件：在包第一次被导入时执行init文件
3 '''
4 from package名.模块文件名 import 类名
5
```

```
1 '''
2 外部使用时
3 '''
4 from package名 import 类名
```

例子：



1.3 区分

python 区分 `from .module02 import *` 和 `from module02 import *`

在Python中，这两种导入方式的区别在于导入的对象是本地（相对）导入还是系统级（绝对）导入。

- `from .module02 import *`：这是一个相对导入，其中的点表示从当前包中导入 `module02` 模块中的所有对象。这种导入方式通常用于包内部模块之间的相互引用。
- `from module02 import *`：这是一个绝对导入，它直接从顶层模块或包中导入。这种导入方式通常用于引用顶层模块或包中的对象，而不是包内部的模块。

1.4 项目：学生信息管理系统

学生信息管理系统V5

可以添加 显示 删除 修改学生信息 姓名 性别 年龄

1. 创建模块 `bll` 存储 `xxController` 业务逻辑层 `business logic layer`
2. 创建模块 `usl` 存储 `xxView` 用户显示层 `user show layer`
3. 创建模块 `dtl` 存储 `xxModel` 数据传输层 `data transfer layer`
4. 创建模块 `main` 调用 `xxView` 的公开方法 启动程序

```
1  """
2  程序启动的入口
3  """
4  from usl import StuView
5
6  # 如果直接运行py文件，__name__就会返回__main__
7  # 如果是被其他文件引入，则__name__返回的是当前模块名
8  # 恒等式
9  # 此行代码定义当前py文件是被运行
10 print(__name__)
11 if __name__ == '__main__':
12     view = StuView()
13     view.main()
```

```
1  """
2  V层
3  """
4  import sys
5  from dtl import StuModel
6  from bll import StuController
7
8
```

```
9 class StuView:
10     def __init__(self):
11         self.__controller = StuController()
12         self.__methods_dict = {
13             "1": self.__input_stu,
14             "2": self.__display_stu,
15             "3": self.__delete_stu,
16             "4": self.__modify_stu,
17             "5": sys.exit,
18         }
19
20     def main(self):
21         while True:
22             self.__display_menu()
23             self.__select_menu()
24
25     def __display_menu(self):
26         print("按1键添加学生信息")
27         print("按2键显示学生信息")
28         print("按3键删除学生信息")
29         print("按4键修改学生信息")
30         print("按5键退出系统")
31
32     def __select_menu(self):
33         number = input("请输入服务数字: ")
34         if number in self.__methods_dict:
35             self.__methods_dict[number]()
36         else:
37             print("暂无此项服务")
38
39     def __input_stu(self):
40         try:
41             age = int(input("请输入学生年龄: ")),
42         except:
43             age = 18
44         model = StuModel(
45             input("请输入学生姓名: "),
46             age,
47             input("请输入学生性别: ")
48         )
49         # 存
50         if self.__controller.add_stu(model):
51             print("添加成功")
52         else:
53             print("添加失败")
54
55     def __display_stu(self):
56         for item in self.__controller.list_stu:
57             print(item)
58
59     def __delete_stu(self):
60         name = input("请输入要修改的姓名: ")
```



```

61         if self.__controller.remove_stu(name):
62             print("success~")
63         else:
64             print("失败! ")
65
66     def __modify_stu(self):
67         pass
68     print(__name__)

```

```

1     """
2     C层
3     """
4     from dtl import StuModel
5
6
7     class StuController:
8         def __init__(self):
9             self.list_stu = []
10
11         def add_stu(self, model):
12             if model: # 和__bool__关联
13                 self.list_stu.append(model)
14                 return True
15             else:
16                 return False
17
18         def remove_stu(self, name):
19             model = StuModel(name)
20             # 更多的想法: 列表推导式??
21             # 3.登堂入室
22             if model in self.list_stu: # 和__eq__关联
23                 self.list_stu.remove(model) # 和__eq__关联
24                 return True
25             return False
26
27             # 2.渐入佳境 重写思想
28             # for item in self.list_stu:
29             #     if item.__eq__(model):
30             #         self.list_stu.remove(item)
31             #         return True
32             # return False
33
34             # 1.原始方案
35             # for item in self.list_stu:
36             #     if item.name == name:
37             #         self.list_stu.remove(item)
38             #         return True
39             # return False

```

```

1     """
2     M层
3     """

```

```
4
5 class StuModel:
6     # 下课休息15分钟回来继续 11: 15
7     def __init__(self, name="", age="18", sex="保密"):
8         self.name = name
9         self.age = age
10        self.sex = sex
11
12    def __str__(self):
13        return f"学生姓名是: {self.name}, 性别为: {self.sex}, 年龄{self.age}"
14
15    def __bool__(self):
16        # 此案例中只考虑姓名不能为空
17        return bool(self.name)
18
19    # 2.渐入佳境 重写思想
20    def __eq__(self, other):
21        # 此案例中只考虑姓名的比较
22        return self.name == other.name
```

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，不断向上返回。

适用性： 针对的是**逻辑错误**,而不是语法错误.

(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。没传参也是这个错误
- 值异常(ValueError)：如果 person = int(input("请输入人数："))不能转换
- 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:

```
1  try:
2      可能触发异常的语句
3  except 错误类型1 [as 变量1]:
4      处理语句1
5  except 错误类型2 [as 变量2]:
6      处理语句2
7  except Exception [as 变量3]:
8      不是以上错误类型的处理语句
9  else:
10     未发生异常的语句
11  finally:
12     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**。运行下去

(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

```
1  def get_score():
2      while True:
3          try:
4              score = int(input("请输入成绩:"))# 如果异常,程序向上返回,不执行return语句
5              return score # 唯一的退出
6          except:
7              print("输入有误")
8
9
10 score = get_score()
11 print("成绩是: %d"%score)
```

2.3 raise 语句

(1) 作用: 抛出一个错误,传递**错误信息**, 让程序进入异常状态。except接收

(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。

```
1  class Wife:
```

```

2     def __init__(self, age):
3         self.age = age
4
5     @property
6     def age(self):
7         return self.__age
8
9     @age.setter
10    def age(self, value):
11        if 20 <= value <= 60:
12            self.__age = value
13        else:
14            # 1.创建异常 -- 抛出 错误信息
15            raise Exception("我不要", "if 20 <= value <= 60", 1001)
16
17    # -- 2.接收 错误信息
18    while True:
19        try:
20            age = int(input("请输入你老婆年龄: "))
21            w01 = Wife(age)
22            break
23        except Exception as e:
24            # 3.读取消息
25            print(e.args) # ('我不要', 'if 30 <= value <= 60', 1001)

```

3. 迭代自己写的对象

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) 可迭代对象

定义：具有__iter__函数的对象，可以返回迭代器。

本质：数据是自定义的Class类对象,外部直接for它拿到数据

(2) 语法

```

1  # 创建:
2  class 可迭代对象名称:
3      def __iter__(self):
4          return 迭代器
5
6  # 实际使用:
7  for 变量名 in 可迭代对象名称():
8      语句

```

(3) 使用时的原理:

```

1  # 1. 获取迭代器
2  迭代器 = 可迭代对象.__iter__()
3  while True:
4      try:
5          # 2. 获取下一个元素
6          print(迭代器.__next__())
7          # 3. 如果迭代完了,则退出循环
8      except StopIteration:
9          break

```

演示: 遍历字符串

```

1  # 可迭代对象是一个可以返回迭代器的对象, 它允许通过遍历得到内部的元素, 但是本身不提供遍历方法(__next__), 但是
  # 它提供了__iter__方法, 此方法返回一个迭代器对象, 然后通过迭代器对象来遍历可迭代对象的元素
2  # 遍历的时候会不断的获取到元素, 如果没有元素可以被获取的时候就会抛出StopIteration异常, 所以需要有异常处理
3
4
5  message = "我是花果山水帘洞孙悟空"
6  # for item in message:
7  #     print(item)
8
9  # 1. 获取迭代器对象
10 iterator = message.__iter__()
11 # 2. 获取下一个元素
12 while True:
13     try:
14         item = iterator.__next__()
15         print(item)
16         # 3. 如果停止迭代则跳出循环
17     except StopIteration:
18         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.

不需要自己实现 `next iter` 方法

```

1  """
2  ① 创建字典，使用迭代思想，打印每个键值对，
3      不需要自己实现 __next__ __iter__ 方法
4  """
5
6  dict1 = {
7      "a": 1,
8      "b": 2,
9      "c": 3,
10 }
11 iterator = dict1.__iter__()
12 while True:
13     try:
14         key = iterator.__next__()
15         print(key, dict1[key])
16     except StopIteration:
17         break

```

需要自己实现next iter方法

```

1  # 迭代器
2  class DictIterator():
3      def __init__(self, data):
4          self.__data = data
5          self.__keys = list(data.keys())
6          self.__index = 0
7
8      def __next__(self):
9          if self.__index < len(self.__keys):
10             key = self.__keys[self.__index]
11             value = self.__data[key]
12             self.__index += 1
13             return key, value
14             raise StopIteration
15
16
17 # 可迭代对象
18 class MyDict:
19     def __init__(self):
20         self.data = {
21             "a": 1,
22             "b": 2,
23             "c": 3,
24         }
25
26     def __iter__(self):
27         return DictIterator(self.data)
28
29
30 d1 = MyDict()
31 iterator = d1.__iter__()
32

```

```

33     while True:
34         try:
35             item = iterator.__next__()
36             print(item) # 元组
37         except StopIteration:
38             break
39
40     # 这样写也行
41     for key, value in d1:
42         print(key, value)

```

面试题：能够参与for循环的条件是什么？

是可迭代对象，具有__iter__函数, 能够获取迭代器对象

3.2 迭代器对象iterator

(1) **迭代器** 定义：可以被next()函数调用并返回下一个值的对象。

(2) 语法

```

1     class 迭代器类名:
2         def __init__(self, 聚合对象):
3             self.聚合对象= 聚合对象
4
5         def __next__(self):
6             if 没有元素:
7                 raise StopIteration
8             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for，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。

1. 遍历容器
2. 懒加载（大数据 无限序列）
3. 链式操作
4. 自定义遍历
5. 和生成器协同

演示：

迭代(for)内部原理

```

1     class StudentIterator:
2         def __init__(self, data):
3             self.__data = data
4             self.__index = -1
5

```

```

6         def __next__(self):
7             if self.__index == len(self.__data) - 1:
8                 raise StopIteration()
9             self.__index += 1
10            return self.__data[self.__index]
11
12
13    class StudentController:
14        def __init__(self):
15            self.__students = []
16
17        def add_student(self, stu):
18            self.__students.append(stu)
19
20        def __iter__(self):
21            return StudentIterator(self.__students)
22
23
24    controller = StudentController() # 数据是: 自定义的Class类对象, 外部直接for它拿到数据
25    controller.add_student("悟空")
26    controller.add_student("八戒")
27    controller.add_student("唐僧")
28
29    # for item in controller:
30    #     print(item)
31
32
33    # 上面for的内部原理
34    iterator = controller.__iter__()
35    while True:
36        try:
37            item = iterator.__next__()
38            print(item)
39        except StopIteration:
40            break

```

练习1: 遍历商品控制器

```

1    class CommodityController:
2        pass
3
4    controller = CommodityController()
5    controller.add_commodity("屠龙刀")
6    controller.add_commodity("倚天剑")
7    controller.add_commodity("芭比娃娃")
8
9    for item in controller:
10        print(item)

```

练习2: 遍历图形控制器


```

1  class GraphicController:
2      pass
3
4  controller = CommodityController()
5  controller.add_graphic("圆形")
6  controller.add_graphic("矩形")
7  controller.add_graphic("三角形")
8
9  for item in controller:
10     print(item)

```

练习3: 创建自定义range类, 实现下列效果.

```

1  class MyRangeIterator:
2      def __init__(self, end):
3          self.end = end
4          self.index = 0
5
6      def __next__(self):
7          if self.index < self.end:
8              res = self.index
9              self.index += 1
10             return res
11             raise StopIteration
12
13  class MyRange:
14      def __init__(self, end):
15          self.end = end
16
17      def __iter__(self):
18          return MyRangeIterator(self.end)
19
20
21  for number in MyRange(5):
22      print(number) # 0 1 2 3 4

```

=>简化: 用生成器

```

1  def my_range(end):
2      index = 0
3      while index < end:
4          yield index
5          index += 1
6
7  # 生成器特点: 每次只存当前, 不存之前
8  for item in my_range(5):
9      print(item)

```

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。

(3)何时：大数据遍历

- 函数有单个结果：
传统思想-使用 `return` 返回容器
- 函数有多个结果(list,tuple):
生成器思想-使用 `yield` 返回元素

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

```
1  # 创建：
2  def 函数名():
3      ...
4      yield 数据
5      ...
6
7  # 调用：
8  for 变量名 in 函数名():
9      语句
```

原理：

```
1  def my_range(stop):
2      number = 0
3      while number < stop:
4          yield number
5          number += 1
6
7  # 循环一次,计算一次,返回一次
8  obj = my_range(5) # 返回生成器对象
9  iterator = obj.__iter__()
10 while True:
11     try:
12         item = iterator.__next__()
13         print(item)
14     except StopIteration:
15         break
16 # =====
17 # 相当于
```

```
18 for num in my_range(5):
19     print(num)
```

(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”

(4) 执行过程:

- 调用生成器函数会自动创建迭代器对象。
- 调用迭代器对象的next()方法时才执行生成器函数。
- 每次执行到yield语句时返回数据，暂时离开。
 - 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。
- 待下次调用next()方法时继续从离开处继续执行。

生成器和迭代器关系:

```
class generator: # 生成器 = 可迭代对象+迭代器
def iter(self): # 1.可迭代对象
    return self
def next(self): # 2.迭代器
    计算数据
    return 数据
```

演示:

```
1 # 生成器思想:几乎不占内存
2 # 问题: 生成器只能用一次, 遍历
3 def my_range(stop):
4     number = 0
5     while number < stop:
6         yield number
7         number += 1
8
9 for number in my_range(5):
10     print(number) # 0 1 2 3 4
11
12 # 解决方案:
13 data = list(my_range(100))
14 for item in data:
15     print(item)
```

练习1: 定义函数,在列表找出所有偶数,一个个给

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

```

1 list1= [43,43,54,56,76,87,98]
2 def find_all_even():
3     for item in list1:
4         if item % 2 == 0:
5             yield item
6 result = find_all_even()# 调函数不执行函数体,但返回生成器对象
7 for number in result:
8     print(number)

```

练习2: 定义函数,在列表找出所有数字

[43,"悟空",True,56,"八戒",87.5,98]

```

1
2
3 alist= [43,"悟空",True,56,"八戒",87.5,98]
4
5
6 def find_all_numbers(lst):
7     for item in lst:
8         if isinstance(item, (int, float, complex)) and not isinstance(item, bool):
9             yield item
10        # try:
11        #     float(item)
12        #     yield item
13        # except ValueError:
14        #     pass
15
16
17 for item in find_all_numbers(alist):
18     print(item)

```

4.2 内置生成器

4.2.1 枚举函数enumerate

(1) 语法:

```

1 for 元组 in enumerate(可迭代对象):
2     语句
3
4 for 索引, 元素in enumerate(可迭代对象):
5     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。又想读又想修改用

演示

```
1 list01 = [54, 0, 65, 65, 7, 0]
2 # 1.快捷键:iter + 回车
3 # 从头到尾读取
4 for item in list01:
5     print(item)
6
7 # 从头到尾修改
8 for i in range(len(list01)): # range也返回生成器
9     if list01[i] == 0:
10         list01[i] = 10
11
12 # 2.快捷键:itere + 回车
13 # 每次返回的是一个元组(第一个元素是索引,第二个元素是元素)
14 for i, item in enumerate(list01):
15     if item == 0:
16         list01[i] = 10
```

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：

```
1 for item in zip(可迭代对象1, 可迭代对象2):
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元组，生成的元组个数由最小的可迭代对象决定。

演示：

```
1 list_name = ["悟空", "八戒", "沙僧"]
2 list_age = [22, 26, 25]
3
4 # 应用1: 几个list -> 一个个元组
5 for item in zip(list_name, list_age):
6     print(item)
7 # ('悟空', 22)
8 # ('八戒', 26)
9 # ('沙僧', 25)
10
11 # 应用2:矩阵转置
12 map = [
13     [2, 0, 0, 2],
14     [4, 2, 0, 2],
15     [2, 4, 2, 4],
```

```

16     [0, 4, 0, 4]
17 ]
18 # new_map = []
19 # for item in zip(map[0],map[1],map[2],map[3]):
20 #     new_map.append(list(item))
21 # print(new_map)
22
23 # new_map = []
24 # for item in zip(*map):  #把2维列表拆开!!!
25 #     new_map.append(list(item))
26
27 # 列表推导式
28 new_map = [list(item) for item in zip(*map)]
29 print(new_map)
30 # [[2, 4, 2, 0], [0, 2, 4, 4], [0, 0, 2, 0], [2, 2, 4, 4]]

```

练习：几个list合并成一个字典

```

1  list1 = ["name", "age", "sex"]
2  list2 = ["xiaoyi", 18, "nv"]
3
4  # 方法1
5  dict_new = {}
6  for key, value in zip(list1, list2):
7      dict_new[key] = value
8  print(dict_new)
9
10 # 方法2
11 dict_new2 = {key: value for key, value in zip(list1, list2)}
12 print(dict_new2)
13
14 # 方法3
15 print(dict(zip(list1, list2)))

```

练习：使用学生列表封装以下三个列表中数据 -> 1个对象数组

list_student_name = ["悟空", "八戒", "白骨精"]

list_student_age = [28, 25, 36]

list_student_sex = ["男", "男", "女"]

```

1 list_student_name = ["悟空", "八戒", "白骨精"]
2 list_student_age = [28, 25, 36]
3 list_student_sex = ["男", "男", "女"]
4
5 class Student():
6     def __init__(self, name, age, sex):
7         self.name = name
8         self.age = age
9         self.sex = sex
10
11 students = []
12 for item in zip(list_student_name, list_student_age, list_student_sex):
13     students.append(Student(item[0], item[1], item[2]))

```

输出json

```

1 import json
2
3 list_student_name = ["悟空", "八戒", "白骨精"]
4 list_student_age = [28, 25, 36]
5 list_student_sex = ["男", "男", "女"]
6
7 students = []
8 for item in zip(list_student_name, list_student_age, list_student_sex):
9     students.append({"name":item[0], "age":item[1], "sex":item[2]})
10
11 # 生成students.json文件
12 with open('students.json', 'w') as f:
13     json.dump(students, f)

```

4.3 生成器表达式(简化)

(1) 定义：用推导式形式创建生成器对象。[] -> ()

何时：得到数据后，要**从头到尾读取一次**用

缺点：操作数据不方便

解决：转换为列表：`list(生成器)`

(2) 语法：

```

1 列表 = [表达式 for 变量 in 可迭代对象 if 条件]
2 生成器 = (表达式 for 变量 in 可迭代对象 if 条件)
3
4 """
5     列表

```

```

6         优点:操作数据方便
7         缺点:占用内存较多
8     生成器
9         优点:占用内存较少
10        缺点:操作数据不方便
11            (不能使用索引/切片,不能删除修改,只能使用一次)
12        适用性/何时:从头到尾读取一次!!!
13
14        解决:转换为列表
15            list(生成器)
16            tuple(生成器)
17            set(生成器)
18    """

```

练习1: 使用生成器表达式在列表中获取所有字符串.

练习2: 在列表中获取所有整数,并计算它的平方.

```

1    """
2    练习1: 使用生成器表达式在列表中获取所有字符串.
3
4    练习2: 在列表中获取所有整数,并计算它的平方.
5    """
6    list01 = [43, "a", 5, True, 6, 7, 89, 9, "b"]
7
8    # 练习1
9    result = (item for item in list01 if type(item) == str)
10   for item in result:
11       print(item)
12
13   # 练习2: 在列表中获取所有整数,并计算它的平方.
14   result = (item ** 2 for item in list01 if type(item) == int)
15   for item in result:
16       print(item)

```

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

(3) 适用性/何时: 多个函数, 主体逻辑相同, 但核心算法不同

(4) 步骤:

1. "封装"[分]:根据需求将多个变化点分别定义到函数中
2. "继承"[隔]:将变化的函数 统一抽象为通用函数的参数,隔离了通用函数与变化函数
3. "多态"[做]:按照通用函数中确定的使用方式,创建变化函数

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。

```
1  list01 = [342, 4, 54, 56, 6776]
2
3  # 定义函数,在列表中查找所有大于100的数
4  def get_number_gt_100():
5      for number in list01:
6          if number > 100:
7              yield number
8
9
10 # 定义函数,在列表中查找所有偶数
11 def get_number_by_even():
12     for number in list01:
13         if number % 2 == 0:
14             yield number
15
16 -----
17 # 参数:得到的是列表中的元素
18 # 返回值:对列表元素判断后的结果(True False)
19 def condition01(number):
20     return number > 100
21
22 def condition02(number):
23     return number % 2 == 0
24
25 def condition03(number):
26     return number < 10
27
28
29 # 通用函数
30 def find_all(condition): # 抽象,个性化函数作为参数传入
31     for item in list01:
32         if condition(item):# 统一
33             yield item
34
35 for item in find_all(condition03):
36     print(item)
```

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：

- 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

```
1 list01 = [45, 56, 567, 78, 89]
2
3 # 参数是:列表中的元素
4 def condition01(item):
5     return item % 2 != 0
6
7 def condition02(item):
8     return item % 3 == 0 or item % 5 == 0
9
10 # 高阶函数:参数是函数的函数
11 def find_single(condition):
12     for item in list01:
13         if condition(item):
14             return item
15
16 print(find_single(condition01))
```

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：

- 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

```

1  class Employee:
2      def __init__(self, eid, did, name, money):
3          self.eid = eid  # 员工编号
4          self.did = did  # 部门编号
5          self.name = name
6          self.money = money
7
8
9  list_employees = [
10     Employee(1001, 9002, "师父", 60000),
11     Employee(1002, 9001, "孙悟空", 50000),
12     Employee(1003, 9002, "猪八戒", 20000),
13     Employee(1004, 9001, "沙僧", 30000),
14     Employee(1005, 9001, "小白龙", 15000),
15 ]
16
17
18 def condition01(emp):
19     return emp.eid == 1003
20
21
22 def condition02(element):
23     return element.name == "孙悟空"
24
25
26 # 高阶函数:参数是函数的函数
27 def find_all(condition):
28     for item in list_employees:
29         if condition(item):
30             yield item
31
32
33 for item in find_all(condition01):
34     print(item.__dict__)

```

pyQt开发可视化软件

书籍: <https://www.bookstack.cn/read/PyQt5-Chinese-tutorial/README.md>

Python PyQT5下载安装: <https://blog.csdn.net/tingguan/article/details/100892128>

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

```
1  # 定义:
2  变量 = lambda 形参: 方法体
3
4  # 调用:
5  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(条件), 且不支持赋值语句。
- lambda不支持赋值语句
- lambda写在实参的地方, 作为个性化条件函数, 作为函数的实参传入

演示:

common包/iterable_tools

```
1  class IterableHelper():
2      @staticmethod
3      def find_all(iterable, condition):
4          for item in iterable:
5              if condition(item):
6                  yield item
```

客户端使用

```
1  from common.iterable_tools import IterableHelper
2
3  # 定义函数, 在列表中查找所有大于100的数
4  # def condition01(number):
5  #     return number > 100
6
7  # 定义函数, 在列表中查找所有偶数
8  # def condition02(number):
9  #     return number % 2 == 0
10
11  list01 = [342, 4, 54, 56, 6776]
12
13  for item in IterableHelper.find_all(list01, lambda number: number > 100):
14      print(item)
15
16  for item in IterableHelper.find_all(list01, lambda number: number % 2 == 0):
17      print(item)
```

练习: 在员工列表查找薪资最高的员工信息
求员工列表中所有员工薪资的总和

```

1  class Emp:
2      def __init__(self, id, did, name, money):
3          self.id = id
4          self.did = did
5          self.name = name
6          self.money = money
7
8      def __str__(self):
9          return f"ID:{self.id}部门的id为{self.did}, 姓名是: {self.name}薪资是: {self.money}"
10
11
12  list1 = [
13      Emp(1001, 130, "小艺", 1000),
14      Emp(1002, 120, "小小宇", 2000),
15      Emp(1003, 110, "芹菜啊", 3000),
16      Emp(1004, 110, "彤彤", 3000),
17      Emp(1005, 110, "琳琳", 4000)
18  ]
19  # 方法1:
20  # maxv = list1[0]
21  # for i in range(1, len(list1)):
22  #     if maxv.money < list1[i].money:
23  #         maxv = list1[i]
24  # print(maxv)
25
26
27  # 方法2: 函数式编程思想
28  def get_max(func):
29
30      maxv = list1[0]
31      for i in range(1, len(list1)):
32          if func(maxv) < func(list1[i]):
33              maxv = list1[i]
34      return maxv
35  print(get_max(lambda item:item.money))
36
37  # 方法3: 内置的max函数

```

5.1.2 内置高阶函数(捷径)

- (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map对象。list()转
- (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) reduce

```
1  from functools import reduce
2
3  res = reduce(lambda x, y: x + y, [1, 2, 3, 4, 5], 10)
4  print(res) # 加上初始值: 25
5  res = reduce(lambda x, y: x if x < y else y, [1, 2, 3, 4, 5])
6  print(res) # 最小值: 1
```

演示:

```
1  class Employee:
2      def __init__(self, eid, did, name, money):
3          self.eid = eid # 员工编号
4          self.did = did # 部门编号
5          self.name = name
6          self.money = money
7
8
9  # 员工列表
10 list_employees = [
11     Employee(1001, 9002, "师父", 60000),
12     Employee(1002, 9001, "孙悟空", 50000),
13     Employee(1003, 9002, "猪八戒", 20000),
14     Employee(1004, 9001, "沙僧", 30000),
15     Employee(1005, 9001, "小白龙", 15000),
16 ]
17
18 # 1. map 映射
19 # 需求: 获取所有员工姓名
20 for item in map(lambda item: item.name, list_employees):
21     print(item)
22 # employees_names_list = list(map(lambda item: item.name, list_employees))
23
24 # 2. filter 过滤器
25 # 需求: 查找所有部门是9002的员工
26 for item in filter(lambda item: item.did == 9002, list_employees):
27     print(item.__dict__)
28
29 # 3. max min 最值
30 emp = max(list_employees, key=lambda emp: emp.money)
31 print(emp.__dict__)
32
33 # 4. sorted
34 # 升序排列
35 # 注意: 返回新列表, 不改变原有列表
36 new_list = sorted(list_employees, key=lambda emp: emp.money)
37 print(new_list)
38
```

```

39 # 降序排列
40 # 注意: 返回新列表, 不改变原有列表
41 new_list = sorted(list_employees, key=lambda emp: emp.money, reverse=True)
42 print(new_list)
43
44 # 注意: 修改原列表
45 list_employees.sort(key=lambda e:e.did )

```

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])

```

1 class Commodity:
2     def __init__(self, cid=0, name="", price=0):
3         self.cid = cid
4         self.name = name
5         self.price = price
6
7 list_commodity_infos = [
8     Commodity(1001, "屠龙刀", 10000),
9     Commodity(1002, "倚天剑", 10000),
10    Commodity(1003, "金箍棒", 52100),
11    Commodity(1004, "口罩", 20),
12    Commodity(1005, "酒精", 30),
13 ]

```

练习:

给定一个数字列表,筛选出所有的正整数,并且求它们的2次幂

`[-3,6,-1,0,3] ==> [6,3] ==> [36,9]`

```

1 list1 = [-3, 6, -1, 0, 3]
2
3 # 方法1
4 res = list(map(lambda x: x ** 2, filter(lambda x: x > 0, list1)))
5 print(res) # [36, 9]
6
7 # 方法2
8 # x是累加器[], y是正在被处理的值
9 from functools import reduce
10 res = reduce(lambda x, y: x + [y ** 2], filter(lambda x: x > 0, list1), [])
11 print(res) # [36, 9]
12 # 如果将filter分离, 要转为list, 再reduce, 防止迭代器消耗

```

5.2 函数作为返回值

逻辑连续，当内部函数被调用时，外函数的变量一直在内存中，类似于类变量(共享)。

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

```
1  # 定义:
2  def func01():
3      a = 10
4      def func02():
5          nonlocal a
6          a -= 1
7          print(a)
8
9      return func02
10
11 # 调用外函数, 接收内函数
12 result = func01()
13 # 调用内函数
14 result()
```

(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：

1. 逻辑连续：从得2000元钱,到不断购买商品的过程连续不中断.
2. 实现python装饰器

演示：

获得.....压岁钱

购买xx商品花了xx元

购买xx商品花了xx元


```

1  def give_gife_money(money):
2      print("获得", money, "元压岁钱")
3
4      def child_buy(commodity, price):
5          nonlocal money
6          money -= price
7          print("购买了", commodity, "花了", price, "元,还剩下", money)
8      return child_buy
9
10 action = give_gife_money(500)
11 action("变形金刚", 200)
12 action("芭比娃娃", 300)

```

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，为原函数添加新功能的函数。

(2) 何时：为原函数添加新功能

(3) 语法：

```

1  def 函数装饰器名称(func):
2      def wrapper(*args, **kwargs):
3          需要添加的新功能
4          return func(*args, **kwargs)
5      return wrapper
6
7  @ 函数装饰器名称
8  def 原函数名称(参数):    # 内部细节也不变
9      函数体
10
11  原函数(参数)    # 调用代码不需要改变

```

(4) 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。

```

1  #def func01():
2  #    print("旧功能")
3  #
4  # def func_new():
5  #    print("新功能")
6  # func01 = new_func    # 1.新功能覆盖了旧功能
7
8
9  # def func01():
10 #    print("旧功能")
11 #
12 # def func_new(func):
13 #    print("新功能")
14 #    func()

```

```

15 # func01 = new_func(func01) # 2.新旧都调用执行了,但并不希望在本行立刻执行新旧功能,想在之后的代码执行
16
17
18
19 def old_func01():
20     print("旧功能")
21
22 def new_func(func):
23     def wrapper():
24         print("新功能")
25         func() # 执行旧功能
26
27     return wrapper
28
29 # 3.调用一次外部函数(装饰器本质)
30 old_func01 = new_func(old_func01)
31 # 调用多次内部函数
32 old_func01()
33 old_func01()

```

(5) 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。

1.标准装饰器

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()

```

解决:

```

1 def verify_permissions(func):
2     def wrapper(*args):
3         print("验证权限") # 2.新东西
4         res = func(*args) # 3.调用旧功能
5         return res
6

```

```

7     return wrapper
8
9     @verify_permissions
10    def insert(data):    # 旧功能
11        print("插入")
12
13    @verify_permissions
14    def delete():        # 旧功能
15        print("删除")
16        return "ok"
17
18    insert("新数据")    # 1.当有人调用旧函数,实际调用内函数
19    delete()

```

练习2: 不知道原函数有几个参数

```

1    """
2        装饰器 - 标准
3        内函数的返回值:旧功能的返回值
4    """
5
6    def func_new(func):
7        def wrapper(*args):# 合成元组
8            print("新功能") # 2.新功能
9            res = func(*args) # 3.再调回去, 用旧功能
10           return res
11        return wrapper
12
13    # func01 = func_new(func01)
14    @func_new
15    def func01(p1):# 3
16        print("旧功能1")
17        return 100
18
19    @func_new
20    def func02(p1,p2):# 3
21        print("旧功能2")
22
23
24    # func01 = func_new(func01)
25    # func02 = func_new(func02)
26
27    # 1:客户端调用, 传数据
28    value = func01(10) # 调用内函数
29    value = func02(10,20) # 调用内函数
30    print(value)

```

练习3: 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

原方法：

```
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))
```

解决：

```
1 import time
2
3 def print_execution_time(func):
4     def wrapper(*args, **kwargs):
5         start = time.time()
6         res = func(*args, **kwargs) # 调用旧功能
7         stop = time.time()
8         print(f"函数执行时间: {stop - start:.2f}s") # 新功能
9         return res
10
11     return wrapper
12
13 @print_execution_time # 调用外函数, 返回内函数给旧功能
14 def sum_data(n):
15     sum_value = 0
16     for number in range(n+1):
17         sum_value += number
18     return sum_value
19
20 print(sum_data(1000000))
```

or

```
1 """
2 sum_data 原本函数执行 for循环 一遍循环一遍加等 返回加等的结果
3 sum_data(100) 100次 0+1+2+...+100
4
5 扩展sum_data功能 增加一个打印函数, 打印执行的时间
6 执行代码前的时间
7 执行代码后的时间
8 执行用时
9 假设:
10 执行前: 00:00:01
11 执行后: 00:00:10
12 消耗了xx
13 """
14 from datetime import datetime
```

```

15
16
17 def print_time(func):
18     def wrapper(num):
19         start = datetime.now()
20         res = func(num)
21         end = datetime.now()
22         print(f"消耗: {(end - start).total_seconds()}s") # 时间差多少秒
23         return res
24
25     return wrapper
26
27 @print_time
28 def sum_data(n):
29     sum = 0
30     for item in range(n + 1):
31         sum += item
32     return sum
33
34
35 print(sum_data(100))

```

添加功能：删除操作检查是否有权限

```

1  """
2  权限验证装饰器
3  查看是否有权运行某个函数(admin权限才可以运行)
4  ["admin","user"]
5  """
6
7  def permission_re(func):
8      def wrapper(str_pre):
9          if str_pre in ["admin", "user"]:
10             if str_pre == "admin":
11                 func()
12             else:
13                 print("没权限")
14          else:
15             print("权限异常")
16
17      return wrapper
18
19
20 @permission_re
21 def f1():
22     print(f"删除成功")
23
24
25 f1("admin") # 删除成功

```

```
26     f1("user") # 没权限
27     f1("gust") # 权限异常
```

2.装饰器参数

写嵌套函数

```
1     def permission_re(permission):
2         def decorator(func):
3             def wrapper():
4                 if permission in ["admin", "user"]:
5                     return func()
6                 else:
7                     print("没权限")
8
9             return wrapper
10
11         return decorator
12
13     @permission_re("admin")
14     # 实际上的执行: f1 = permission_re("acc")(f1)
15     # @permission_re("acc")
16     def f1():
17         print(f"删除")
18
19     f1()
```

3.内置装饰器

就是类的静态方法

```
1     class Stu:
2
3         # 不需要实例化, 直接访问
4         # 不需要约定默认参数self
5         @staticmethod
6         def say():
7             print("青山不改, 绿水长流~")
8
9
10     Stu.say()
```

6.正则表达式

```
1 import re
```

6.1 概述

- 学习动机
 1. 文本数据处理已经成为常见的编程工作之一
 2. 对文本内容的**搜索，定位，提取**是逻辑比较复杂的工作
 3. 为了快速方便的解决上述问题，产生了正则表达式技术
- 定义

即文本的高级匹配模式，其本质是由一系列字符和特殊符号构成的字符串，即正则表达式。

- 原理

通过普通字符和有特定含义的字符，来组成字符串，用以描述一定的字符串规则，比如：重复，位置等，来表达某类特定的字符串，进而匹配。

6.2 元字符使用-findall()

- 普通字符

匹配规则：每个普通字符匹配其对应的字符

```
1 re.findall('ab',"abcdefabcd")
2 # ['ab', 'ab']
```

注意：正则表达式在python中也可以匹配中文

- 或关系

元字符: |

匹配规则: 匹配 | 两侧任意的正则表达式即可 2 个正则表达式

```
1 re.findall('com|cn',"www.baidu.com/www.tmooc.cn")
2 # ['com', 'cn']
```

- 匹配单个字符

元字符: .

匹配规则：匹配除换行外的任意一个字符

```
1 re.findall('张.丰', "张三丰,张四丰,张五丰")
2 # ['张三丰', '张四丰', '张五丰']
```

- 匹配字符集

元字符：[字符集]

匹配规则：匹配字符集中的任意一个字符

表达形式：

[aeiou你我他] 表示 [] 中的任意一个字符

[0-9],[a-z],[A-Z] 表示区间内的任意一个字符

[_#?0-9a-z] 混合书写，一般区间表达写在后面

```
1 re.findall('[aeiou]', "How are you!")
2 # ['o', 'a', 'e', 'o', 'u']
```

- 匹配字符集反集

元字符：[^字符集]

匹配规则：匹配除了字符集以外的任意一个字符

```
1 re.findall('[^0-9]', "Use 007 port")
2 # ['U', 's', 'e', ' ', ' ', 'p', 'o', 'r', 't']
```

- 匹配重复字符

元字符：*

匹配规则：匹配前面的一个字符出现0次或多次

```
1 re.findall('wo*', "wooooo~~w!")
2 # ['wooooo', 'w']
```

元字符：+

匹配规则：匹配前面的字符出现1次或多次

```
1 re.findall('[A-Z][a-z]+', "Hello World")
2 # ['Hello', 'World']
```

元字符: ?

匹配规则: 匹配前面的字符出现0次或1次,可有可无

```
1  # 匹配整数
2  re.findall('-?[0-9]+', "Jame, age:18, -26")
3  # ['18', '-26']
```

元字符: {n}

匹配规则: 匹配前面的字符出现n次

```
1  # 匹配手机号码
2  re.findall('1[0-9]{10}', "Jame:13886495728")
3  # ['13886495728']
```

元字符: {m,n}

匹配规则: 匹配前面的字符出现m-n次

```
1  # 匹配qq号
2  re.findall('[1-9][0-9]{5,10}', "Baron:1259296994")
3  ['1259296994']
```

- 匹配字符串开始位置

元字符: ^

匹配规则: 匹配目标字符串的开头位置

```
1  re.findall('^Jame', "Jame,hello")
2  # ['Jame']
```

- 匹配字符串的结束位置

元字符: \$

匹配规则: 匹配目标字符串的结尾位置

```
1  re.findall('Jame$', "Hi, Jame")
2  # ['Jame']
```

规则技巧: ^ 和 \$ 必然出现在正则表达式的开头和结尾处。

如果两者同时出现, 则中间的部分必须匹配整个目标字符串的全部内容。

- 匹配任意 (非) 数字字符

元字符: `\d` `\D`

匹配规则: `\d` 匹配 任意数字字符, `\D` 匹配 任意非数字字符 **过滤数字**

```
1 # 匹配端口
2 re.findall('\d{1,5}', "Mysql: 3306, http:80")
3 # ['3306', '80']
```

- 匹配任意 (非) 普通字符

元字符: `\w` `\W`

匹配规则: `\w` 匹配普通字符, `\W` 匹配非普通字符

说明: 普通字符指数字, 字母, 下划线, 汉字 **过滤特殊字符**

```
1 re.findall('\w+', "server_port = 8888")
2 # ['server_port', '8888']
```

- 匹配任意 (非) 空字符

元字符: `\s` `\S`

匹配规则: `\s` 匹配空字符, `\S` 匹配非空字符

说明: 空字符指 `\s` 匹配 `\r \n \t \v \f` 字符

```
1 re.findall('\w+\s+\w+', "hello    world")
2 # ['hello    world']
```

- 匹配 (非) 单词的边界位置

元字符: `\b` `\B`

匹配规则: `\b` 表示单词边界, `\B` 表示非单词边界

说明: 单词边界指数字字母(汉字)下划线与其他字符的交界位置。 **找出独立单词!**

```
1 re.findall(r'\bis\b', "This is a test.")
2 # ['is']
```

元字符汇总

类别	元字符符号
匹配字符	<code>.</code> <code>[...]</code> <code>[^...]</code> <code>\d</code> <code>\D</code> <code>\w</code> <code>\W</code> <code>\s</code> <code>\S</code>
匹配重复(几个)	<code>*</code> <code>+</code> <code>?</code> <code>{n}</code> <code>{m,n}</code>
匹配位置	<code>^</code> <code>\$</code> <code>\b</code> <code>\B</code>
其他	<code> </code> <code>()</code> <code>\</code>

注意：当元字符符号与Python字符串中转义字符冲突的情况则需要使用r将正则表达式字符串声明为原始字符串，如果不确定那些是Python字符串的转义字符，则可以在所有正则表达式前加r。

转义无效符

在Python中，在字符串前加 r 来讲当前字符串 的转义字符无效化

```
1 | print(r"My name is \"Neo\"")
```

执行结果如下：

```
>>> My name is \"Neo\"
```

元字符使用示例：

```
1 import re
2
3 # 普通字符
4 # result = re.findall('你好', "abcdef你好cd")
5 # print(result)
6
7 # 或关系
8 # result = re.findall('ab|cd', "abcdefbcab")
9 # print(result)
10
11 # 匹配任意一个字符 除了 \n
12 # result = re.findall('张.丰', "张三丰,张四丰,张五丰")
13 # print(result)
14
15 # [] 匹配字符集中一个字符
16 # result = re.findall('[aeiou]', "How are you!")
17 # result = re.findall('[ ! A-Z]', "How are you! ")
18 # print(result)
19
20 # 字符集取反
```

```
21 # result = re.findall('[^a-z]', "How are you! ")
22 # print(result)
23
24 # 匹配重复0次或多次
25 # result = re.findall('wo*', "wooooo~~w!")
26 # print(result) # ['wooooo', 'w']
27
28 # 匹配重复1次或多次
29 # result = re.findall('wo+', "wooooo~~w!")
30 # print(result) # ['wooooo']
31
32 # 匹配重复0次或1次
33 # result = re.findall('wo?', "wooooo~~w!")
34 # print(result) # ['wo', 'w']
35
36 # 重复指定次数
37 # result = re.findall('wo{3}', "wooooo~~w!")
38 # print(result)
39
40 # 匹配重复 m - n 次
41 # result = re.findall('wo{2,4}', "wooooo~~w!")
42 # print(result)
43
44 # 开头结尾位置
45 # result = re.findall('Jame$', "Hi,Jame")
46 # result = re.findall('^Jame', "Jame,hi")
47 # print(result)
48
49 # 匹配数字字符或者非数字
50 # result = re.findall('\d{1,5}', "1Mysql: 3306, http:80")
51 # print(result) # ['1', '3306', '80']
52
53 # result = re.findall('\D+', "Mysql: 3306, http:80")
54 # print(result) # ['Mysql: ', ', ', 'http:']
55
56 # \w 普通字符 和 \W非普通字符
57 # result = re.findall('\w+', "server_port = 你好")
58 # result = re.findall('\W+', "server_port = 8888")
59 # print(result) # ['server_port', '你好']
60
61 # 空字符 和 非空字符
62 # result = re.findall('\w+\s+\w+', "Hello    world")
63 # result = re.findall('\S+', "Hello    world")
64 # print(result) # ['Hello', 'world']
65
66 # 单词边界
67 # result = re.findall(r'\bis', "This is a test")
68 # print(result) # ['is']
69
70 # 特殊符号匹配
71 # result = re.findall(r'-?\d+\.\d*', "-5.4 3.28 45 76.3 -21")
72 # print(result)
```

```

73 # 匹配特殊符号
74 result = re.findall('\$\\d+', "月薪:$150")
75 print(result)
76
77 # 贪婪
78 # result = re.findall("ab+", "abbbbbbbbbc")
79 # print(result)
80 # 非贪婪 (懒惰模式) + --> +?
81 # result = re.findall("ab+?", "abbbbbbbbbc")
82 # print(result)
83
84 # 子组
85 # result = re.search("(ab)+", "abababab")
86 # result = re.search(r'(?P<xing>王|李)\\w{1,3}', "王者荣耀")
87 # print(result.group()) # 获取匹配内容: 王者荣耀

```

随堂练习:

```

1  import re
2
3  # 匹配出其中大写字母开头的单词
4  result = re.findall('[A-Z][a-z]*', "How are you,Jame!,I am")
5  print(result) # ['How', 'Jame', 'I']
6
7  # 匹配出年月日
8  result = re.findall('[0-9]{1,4}', "今天是: 2021-4-30")
9  print(result) # ['2021', '4', '30']
10
11
12 # 匹配出数字
13 result = re.findall('-?[0-9]+', "-20°的天气, 战士负重15Kg")
14 print(result) # ['-20', '15']
15
16 # 匹配电话号码
17 result = re.findall(r'\b1[3578][0-9]{9}\b', "王总:13838384386,银行卡: 693518345879556790")
18 print(result)
19
20 # 匹配qq号码 6-11
21 result = re.findall('[1-9][0-9]{5,10}', "王总: 4268858")
22 print(result)
23
24 # 验证一个用户注册的用户名是否由6-12位数字字母下划线构成
25 name = input("User:")
26 result = re.findall("^[0-9a-zA-Z]{6,12}$", name)
27 print(result)

```

6.3 匹配规则

1. 特殊字符匹配

- 目的：如果匹配的目标字符串中包含正则表达式特殊字符，则在表达式中元字符就想表示其本身含义时就需要进行 \ 处理。

```
1 特殊字符：. * + ? ^ $ [ ] ( ) { } | \
```

- 操作方法：在正则表达式元字符前加 \ 则元字符就是去其特殊含义，就表示字符本身

```
1 # 匹配特殊字符 . 时使用 \. 表示本身含义
2 # 匹配正负数, 小数
3 re.findall('-?\d+\.?\d*', "123, -123, 1.23, -1.23")
4 # ['123', '-123', '1.23', '-1.23']
```

2. 贪婪模式和非贪婪模式

- 定义

贪婪模式：默认情况下，匹配重复的元字符总是 尽可能多 的向后匹配内容。比如：* + ? {m,n}

非贪婪模式(懒惰模式)：让匹配重复的元字符符合规则情况下 尽可能少/点到即止 的向后匹配内容。?

- 贪婪模式转换为非贪婪模式

在对应的匹配重复的元字符后加 '?' 号即可

```
1 * -> *? 没有也算找到一个了
2 + -> +? 出现1个ok了
3 ? -> ?? 没有也ok了
4 {m,n} -> {m,n}? 就找到m个ok了
```

```
1 re.findall(r'\(.*?\)', "(abcd)efgh(higk)")
2 # ['(abcd)', '(higk)']
3
4 result = re.findall(r'《.*?》', "《我的祖国! 》，《小王子 a s x a》，《流浪地球一，，，》")
5 print(result)
6
7 # 匹配如下书名
8 result = re.findall('《.*?》', "《加油! @奥特曼》，《重生—美少女》，《biobio~ 奥利给》")
9 print(result)
```

3. 正则表达式分组()用search,结果找到 1 个

- 定义: 在正则表达式中, 以()建立正则表达式的内部分组, 子组是正则表达式的一部分, 可以作为内部整体操作对象。
- 作用: 可以被作为整体操作, 改变元字符的操作对象

```
1 result = re.findall('(\w+):(\d+)', "Lily:1999 Tome:2000") #返回()组里的, 当只想要一种数据加()
2 print(result) # [('Lily', '1999'), ('Tome', '2000')]
3
4
5 # 只要李蛋分数
6 result = re.findall('李蛋:(\d+)', "李逵:68, 李蛋:75, 李明:86")
7 print(result) # ['75']
8
9 =====search函数=====
10 # 改变 +号 重复的对象
11 re.search(r'(ab)+', "ababababab").group()
12 # 'ababababab'
13
14 # 改变 |号 操作对象
15 re.search(r'(王|李)\w{1,3}', "王者荣耀").group()
16 # '王者荣耀'
17
18 # 匹配一下IP地址 192.168.4.6 (\d{1,3}\.){3}\d{1,3}
19 result = re.search('(\d{1,3}\.){4}', "IP: 192.168.4.6")
20 print(result.group())
21 # 192.168.4.6
```

- 捕获组(一般用match匹配第一个)

捕获组本质也是一个子组, 只不过 拥有一个名称 用以表达该子组的意义, 这种有名称的子组即为捕获组。


格式: (?Ppattern)

```
1 # 给子组命名为 "pig"
2 re.search(r'(?P<pig>ab)+', "ababababab").group('pig')
3 # 'ab'
```

注意事项:

- 一个正则表达式中可以包含多个子组
- 子组可以嵌套但是不宜结构过于复杂
- 子组序列号一般从外到内, 从左到右计数

`r'((ab)c)d(?P<pig>ef)'`



4. 正则表达式匹配原则

1. 正确性,能够 **正确** 的匹配出目标字符串.
2. 排他性,除了目标字符串之外尽可能少的匹配其他内容. **缩范围**
3. 全面性,尽可能考虑到 **目标字符串的所有情况** ,不遗漏

查正则表达式例子，菜鸟工具

<https://c.runoob.com/front-end/854/>

6.4 Python re模块使用

1. 基础函数使用

```
1 re.findall(pattern,string)
2 功能：根据正则表达式匹配目标字符串内容
3 参数：pattern 正则表达式
4       string 目标字符串
5 返回值：匹配到的内容列表,如果正则表达式有子组则 只能 获取到子组对应的内容
6
```

```
1 re.split(pattern,string,max)
2 功能：使用正则表达式匹配内容,切割目标字符串
3 参数：pattern 正则表达式
4       string 目标字符串
5       max 最多切割几部分
6 返回值：切割后的内容列表!!!!!!!!!!!!
7
```



```
1 re.sub(pattern,replace,string,count)
2 功能：使用一个字符串替换正则表达式匹配到的内容
3 参数：pattern 正则表达式
4         replace 替换的字符串
5         string 目标字符串
6         count 最多替换几处,默认替换全部
7 返回值：替换后的字符串!!!!!!!
8
```

```
1 正则函数使用示例：
2 import re
3
4 string = "Alex:1996,Sunny:1998"
5
6 # 使用## 替换正则表达式匹配到的内容
7 result = re.sub("\d+","##",string,2)
8 print(result)
9
10 # 使用匹配内容分割字符串
11 # result = re.split("\W+",string)
12 # print(result) # ['Alex', '1996', 'Sunny', '1998']
13
14 # 如果正则表达式有子组，那么只返回子组对应内容
15 # result = re.findall("(\\w+):(\\d+)",string)
16 # print(result) # [('Lily', '1999'), ('Tome', '2000')]
```

2. 生成匹配对象

```
1 result = re.finditer(pattern,string)
2 功能：根据正则表达式匹配目标字符串 全部内容
3 参数：pattern 正则表达式
4         string 目标字符串
5 返回值：匹配结果的迭代器 iterator
6
7
8 for item in result:
9     print("匹配内容:",item.group())
10    print("所在位置:",item.span())
```

```
1 result = re.match(pattern,string)
2 功能：从字符串的起始位置开始匹配正则表达式
3 参数：pattern 正则
4         string 目标字符串
5 返回值：匹配内容match object
6
7 结果.group() 得到值
```

```
1 result = re.search(pattern,string)
2 功能：匹配目标字符串 第一个符合的内容，字符串符合也行
3 参数：pattern 正则
4       string 目标字符串
5 返回值：匹配内容match object
6
7 结果.group() 得到值
8 结果.span() 得到切片位置
```

match vs search

```
1 import re
2
3 # 定义正则表达式和目标字符串
4 pattern = r'\d+' # 匹配一个或多个数字
5 text = 'h123 Hello 456'
6
7 # 使用match函数从字符串开头开始匹配
8 match_result = re.match(pattern, text)
9 if match_result:
10     print("Match found:", match_result.group())
11 else:
12     print("No match")
13
14 # 使用search函数在整个字符串中搜索匹配项
15 search_result = re.search(pattern, text)
16 if search_result:
17     print("Search found:", search_result.group())
18 else:
19     print("No search match")
20
21
22 # 结果
23 # No match
24 # Search found: 123
```

3. 匹配对象使用

- span() 获取匹配内容的起止位置
- group(n = 0)

功能：获取match对象匹配内容

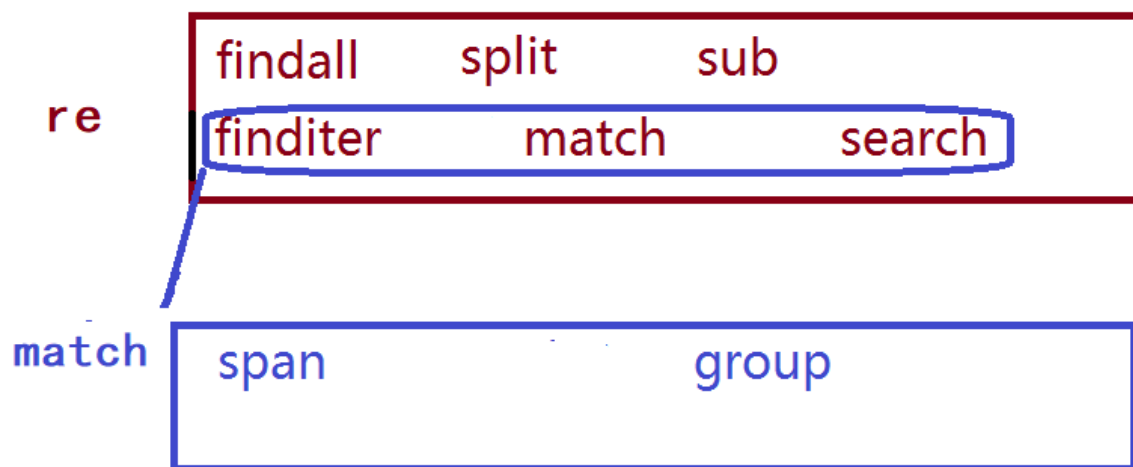
参数：默认为0表示获取整个match对象内容，如果是序列号或者组名则表示获取对应子组内容

返回值：匹配字符串

```

1  # 正则函数使用示例:          常用!!!!!!
2
3  import re
4
5  string = "Alex:1996,Sunny:1998"
6
7  # match匹配开始位置
8  result = re.match("(\\w+):(P<year>\\d+)", string)
9  print(result.group()) # Alex:1996
10 print(result.group(1)) # Alex
11 print(result.group(2)) = print(result.group("year")) # 只获取year组内容
12
13 # search匹配第一处
14 result = re.search("\\d+", string)
15 print(result.group()) # 1996
16
17 # finditer匹配所有, 返回迭代对象
18 result = re.finditer("\\w+", string)
19 # 迭代取值 获取每处匹配内容的match对象
20 for item in result:
21     print("匹配内容: ", item.group())
22     print("所在位置: ", item.span())

```



综合练习：基于 inet.log完成

- 1 编写程序，通过输入一个端口名称（每段首个单词）打印出这个端口描述信息中的address is 的值
- 2
- 3 提示： 段落之间有空行
- 4 端口名称是每段第一个单词

```

5
6 思路： 根据输入的端口名称找到段落
7       在段落中匹配目标
8
9  import re
10
11 # 生成器函数
12 def parg():
13     file = open("inet.log") # 读方式
14     # 每次while循环data获取一段内容
15     while True:
16         data = ""
17         for line in file:
18             if line == "\n":
19                 break
20             data += line
21         if data:
22             yield data # 对外提供一段内容
23         else:
24             return # 到了文件结尾
25
26
27 def main():
28     # 获取端口
29     port = input("端口名称:")
30     for data in parg():
31         # 看一下每段首个单词是否为 port
32         head = re.match("\S+", data).group() # 得到首个单词
33         if head == port:
34             address = re.search("([0-9a-f]{4}\.?)^{3}", data).group()
35             return address
36     return "Not Found"
37
38 # 程序入口
39 print(main())

```

6.5 总结

```

1  re.findall() #得到一个list
2  re.split()  # 得到一个list
3  re.sub()   # 得到一个字符串
4  re.finditer() # 匹配所有 后续遍历 objs
5  re.match() # 从字符串的起始位置开始匹配正则表达式 obj
6  re.search() # 匹配第一处符合正则规则的内容 obj
7
8  info = " 张三  李四      王五  赵柳"
9  result = re.search("\w+", info)
10 print(result.group()) # 获取匹配内容 张三

```

```

11 print(result.span()) # 获取匹配内容对应的位置 (2, 4)
12
13
14
15 1.正常 匹配符合的整体
16 result = re.findall("ab","abcdabcdefa")
17 print(result) # ['ab', 'ab']
18
19 2.[] 匹配里面任意一个符合条件的
20 result = re.findall('[^ !A-Z]', "How are you!")
21 print(result) # ['o', 'w', 'a', 'r', 'e', 'y', 'o', 'u']
22
23 3. * + ? 修饰前面那一个的
24 result = re.findall('wo*', "wooooo~~w!") # ['wooooo', 'w']
25 result = re.findall('wo+', "wooooo~~w!") # ['wooooo']
26 result = re.findall('wo?', "wooooo~~w!") # ['wo', 'w']

```

7. 文件处理

7.1 引入

- 什么是文件

文件是保存在持久化存储设备(硬盘、U盘、光盘..)上的一段数据，一个文本，一个py文件，一张图片，视频音频等这些都是文件。

- 文件分类

- 文本文件：打开后会自动解码为字符，如txt文件，word文件，py程序文件。
- 二进制文件：内部编码为二进制码，无法通过文字编码解析，如压缩包，音频，视频，图片等。

- 字节串类型

- 概念：在python3中引入了字节串的概念，与str不同，字节串以字节序列值表达数据，更方便用来处理二进制数据。
- 字符串与字节串相互转化方法

```

1 - 普通的英文字符字符串常量可以在前面加b转换为字节串，例如：b'hello'
2 - 变量或者包含非英文字符的字符串转换为字节串方法：bytes = str.encode()
3 - 字节串转换为字符串方法：str = bytes.decode()
4
5 注意：python字符串用来表达utf8字符，因为并不是所有二进制内容都可以转化为utf8字符，所以不是所有字节串都能转化为字符串，但是所有字符串都能转化成二进制，所以所有字符串都能转换为字节串。

```

7.2 文件读写操作

使用程序操作文件，无外乎对文件进行读或者写

- 读：即从文件中获取内容
- 写：即修改文件中的内容

对文件实现读写的基本操作步骤为：打开文件，读写文件，关闭文件。

1. 打开文件

```
1  file_object = open(file_name, access_mode='r', buffering=-1, encoding=None)
2  功能：打开一个文件，返回一个文件对象。
3  参数：file_name  文件名；
4          access_mode  打开文件的方式,如果不写默认为‘r’
5          buffering  1表示有行缓冲，默认则表示使用系统默认提供的缓冲机制。
6          encoding='UTF-8'  设置打开文件的编码方式，一般Linux下不需要
7  返回值：成功返回文件操作对象。
```

打开模式	效果
r	以读方式打开，文件必须存在
w	以写方式打开，文件不存在则创建，存在清空原有内容
a	以追加模式打开，文件不存在则创建，存在则继续进行写操作
r+	以读写模式打开 文件必须存在
w+	以读写模式打开文件，不存在则创建，存在清空原有内容
a+	追加并可读模式，文件不存在则创建，存在则继续进行写操作
rb	以二进制读模式打开 同r
wb	以二进制写模式打开 同w
ab	以二进制追加模式打开 同a
rb+	以二进制读写模式打开 同r+
wb+	以二进制读写模式打开 同w+
ab+	以二进制读写模式打开 同a+

注意：

1. 以二进制方式打开文件，读取内容为字节串，写入也需要写入字节串
2. 无论什么文件都可以使用二进制方式打开，但是二进制文件则不能以文本方式打开，否则后续读写会报错。

文件打开代码示例：open_file.py

```

1  # file = open("../day02/2.txt", "r")
2  # file = open("file.txt", "w") # 清除原来内容
3  file = open("file.txt", "a") # 不会清除原来内容
4
5  # 操作文件
6
7  # 关闭
8  file.close()

```

2. 读取文件

- 方法1

```

1  read([size])
2  功能： 来直接读取文件中字符。
3  参数： 如果没有给定size参数（默认值为-1）或者size值为负，文件将被读取直至末尾，给定size最多读取给定数目个字符（字节）。
4  返回值： 返回读取到的内容

```

注意：1. 文件过大时候不建议直接读取到文件结尾，占用内存较多，效率较低

2. 读到文件结尾如果继续进行读操作会返回空字符串。

- 方法2

```

1  readline([size])
2  功能： 用来读取文件中一行
3  参数： 如果没有给定size参数（默认值为-1）或者size值为负，表示读取一行，给定size表示最多读取制定的字符（字节）。
4  返回值： 返回读取到的内容

```

- 方法3

```

1  readlines([sizeint])
2  功能： 读取文件中的每一行作为列表中的一行
3  参数： 如果没有给定size参数（默认值为-1）或者size值为负，文件将被读取直至末尾，给定size表示读取到size字符所在行为止。
4  返回值： 返回读取到的          内容列表!!!!!!!
5
6  例如： ['你好\n', '大爷\n', '挪下车']

```

- 方法4

```

1  # 文件对象本身也是一个可迭代对象，在for循环中可以迭代文件的每一行。
2  for line in file:
3      print(line)

```

```

1  # 打开文件
2  # file = open("file.txt", "r")
3
4  file = open("file.txt", "rb")
5
6  # 1.读取全内容
7  data = file.read()
8  print(data.decode())
9
10 # 2.每次读取一个字符,将文件内容原样打印出来
11 # while True:
12 #     data = file.read(1)
13 #     if not data:
14 #         break
15 #     print(data,end="")
16
17 # 3.按行读取
18 #while True:
19 #     data = file.readline()
20 #     if not data:
21 #         break
22 #     print(data,end="")
23
24 # 读取 所有行 内容
25 # data_list = file.readlines()
26 # print(data_list) # 内容列表
27
28 # 4.迭代每次获取 一行
29 # for line in file:
30 #     print(line)
31
32
33 file.close()

```

随堂练习：基于 dict.txt 完成

```

1  编写一个函数,参数是一个单词,查询这个单词的解释
2  提示 : 单词有可能查不到 None Not Found
3      字符串切片 split()
4  思路 : 使用 word 逐行比对
5
6  def find_word(word):
7      file = open("../day09/dict.txt", "r")
8      # 每次读取一行
9      for line in file:
10         tmp = line.split(' ')[0] # 提取单词
11         # 如果遍历的单词大于word就不用再找了
12         if tmp > word:
13             file.close()

```



```
14         return
15     elif word == tmp:
16         file.close()
17         return line
18
19 print(find_word("abc"))
```

3. 写入文件

- 方法1

```
1 write(data)
2 功能：把文本数据 or 二进制数据块的字符串写入到文件中去
3 参数：要写入的内容
4 返回值：写入的字符个数
```

注意：如果需要换行要手动添加\n

- 方法2:

```
1 writelines(str_list)
2 功能：接受一个字符串列表作为参数，将它们写入文件。
3 参数：要写入的内容列表
```

文件写操作代码示例：

1. 字符串

```
1 # 写打开
2 file = open("file.txt", "w")
3
4 # 1. 写操作
5 file.write("hello, 死鬼\n")
6 file.write("哎呀, 干啥\n")
7
8
9 # 2. 将列表每一项写入到文件
10 data = [
11     "接着奏乐\n",
12     "接着舞\n"
13 ]
14
15 file.writelines(data)
16
17 file.close()
```

```
1 hello,死鬼
2 哎呀,干啥
3 接着奏乐
4 接着舞
```

2. 字节串

```
1 # 写打开
2 file = open("file.txt", "wb")
3
4 # 写操作
5 n = file.write("hello,死鬼\n".encode())
6 file.write("哎呀,干啥\n".encode())
7
8 file.close()
```

3. with操作更好

```
1 # 1. 打开文件
2 with open("xiaoyi.txt", "w", encoding="utf-8") as file:
3     # 2.写
4     file.write("你好\n")
5     file.write("大爷\n")
6     file.write("abcd f asc")
7
8 with open("xiaoyi.txt", "r", encoding="utf-8") as file:
9     print(file.read())
```

写：把代表能够实例化对象的代码写入到文件里

读：读取文件中的内容作为代码去执行

```
1 class StuModel:
2     def __init__(self, name, age, sex):
3         self.name = name
4         self.age = age
5         self.sex = sex
6
7     def __str__(self):
8         return f"姓名: {self.name}, 年龄: {self.age}性别: {self.sex}"
9
10
11 with open("res.txt", "w", encoding="utf-8") as file:
12     file.write('StuModel("张飞", 18, "男")')
13
14 with open("res.txt", "r", encoding="utf-8") as file:
15     # eval 括号中的内容当做代码去执行
16     stu = eval(file.read())
17     print(stu.name)
```

将指定文件复制到这个文件夹下

```
1  随堂练习：编写一个函数，参数传入一个指定的文件，将这个文件
2  复制到当前程序运行目录下，注意不确定文件类型 文本or二进制？
3  思路：边读边写
4
5  def copy(filename):
6      """
7      将指定文件复制到这个文件夹下
8      :param filename: 指定的要复制的文件
9      """
10     new_file = filename.split('/')[-1] # 取出文件名
11     fr = open(filename, "rb") # 原文件
12     fw = open(new_file, "wb") # 新文件
13     # 边读边写
14     while True:
15         data = fr.read(1024)
16         if not data:
17             break
18         fw.write(data)
19     fr.close()
20     fw.close()
21
22
23 copy("/home/tarena/b.jpg")
24
25
26 复制一个地方的文件夹dir->当前文件夹下
27 def copy_directory(dir):
28     dir_name=dir.split('/')[-2]
29     if os.path.exists(dir_name):
30         print(f'文件夹{dir_name}已存在')
31     else:
32         os.mkdir(dir_name) #当前创建一个新文件夹
33         for file in os.listdir(dir):
34             fr = open(dir + file, "rb") # 原文件
35             fw = open(dir_name+"/"+file, "wb") # 新文件
36             fw.write(fr.read()) # 复制一个文件
37             fr.close()
38             fw.close()
39
40 copy_directory(dir = "/Users/liuwei/Desktop/图片/test/")
```

4. 关闭文件

打开一个文件后我们就可以通过文件对象对文件进行操作了，当操作结束后可以关闭文件操作

好处:

1. 可以销毁对象节省资源，（当然如果不关闭程序结束后对象也会被销毁）。
2. 防止后面对这个对象的误操作。

```
1 file_object.close()
```

5. with操作

python中的with语句也可以用于访问文件，在语句块结束后会自动释放资源。

```
1 with open('file', 'r+') as f:  
2     f.read()
```

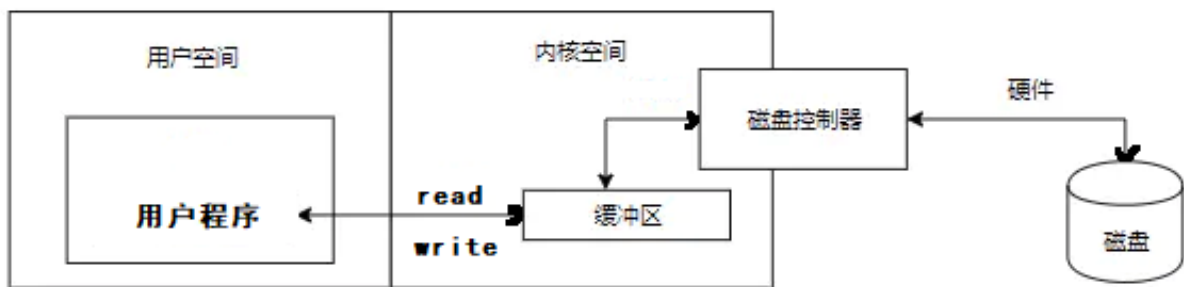
with 使用示例

```
1 # 临时打开文件简单使用 file = open()  
2 with open("file.txt") as file:  
3     data = file.read()  
4     print(data)  
5  
6 # 语句块结束 file 被销毁
```

6. 读写缓冲区

- 定义

系统自动的在**内存**中为每一个正在使用的文件开辟一个空间，在对文件读写时都是先将文件内容加载到缓冲区，再进行读写。



- 作用
 1. 减少和磁盘的交互次数，保护磁盘。
 2. 提高了对文件的读写效率。
- 缓冲区设置

类型	设置方法	注意事项
系统自定义	buffering=-1	
行缓冲	buffering=1	当遇到\n时会刷新缓冲
指定缓冲区大小	buffering>1	必须以二进制方式打开

- 刷新缓冲区条件
 1. 缓冲区被写满
 2. 程序执行结束或者文件对象被关闭 一般情况√
 3. 程序中调用flush()函数

```
1 file_obj.flush()
```

文件读写缓冲示例：buffer.py

```

1  # 1.buffering=1 遇到行刷新保存
2  # file = open("file.txt", "w", buffering=1)
3
4  # 2.buffering > 1 指定缓冲大小, 每10个字节保存一次
5  file = open("file.txt", "wb", buffering=10)
6
7  while True:
8      data = input(">>>")
9      if not data:
10         break
11     file.write(data.encode())
12     # file.flush() # 直接刷新=保存 ctro+s
13
14  file.close()
  
```

7. 文件偏移量

- 定义

打开一个文件进行操作时系统会自动生成一个记录，记录 **每次读写操作时所处的文件位置**，每次文件的读写操作都是从这个位置开始进行的。

注意：

1. r或者w方式打开，文件偏移量在文件开始位置
2. a方式打开，文件偏移量在文件结尾位置

- 文件偏移量控制

```
1 tell()
2 功能：获取文件偏移量大小
3 返回值：文件偏移量
4
```

```
1 seek(offset[, whence])
2 功能：移动文件偏移量位置
3 参数：offset 代表相对于某个位置移动的字节数。负数表示向前移动，正数表示向后移动。
4         whence是基准位置的默认值为 0 代表从文件开头算起，
5                                     1 代表从当前位置算起，
6                                     2 代表从文件末尾算起。
7
```

注意：必须以二进制方式打开文件时，基准位置才能是 1或者2

文件偏移量示例：seek.py

```
1 # 可读可写
2 file = open("file.txt", "wb+")
3
4 # 读
5 file.write("你好".encode())
6 file.flush()
7
8 # 查看偏移量
9 print("文件偏移位置:", file.tell())
10 # 移动偏移量
11 file.seek(-3, 2)
12
13 file.write("向我们".encode()) # 覆盖内容，并且移动偏移量
14
15 # 读后面的，偏移量在写入的下一个
16 data = file.read()
17 print("读取:", data.decode())
```

```
18
19 file.close()
```

随堂练习：

编写一个程序,循环不停的写入日志 (my.log)。每2秒写入一行,要求每写入一行都要显示出来。结束程序后（强行结束）,重新运行要求继续往下写，序号衔接

1. 2020-12-28 18:18:20
2. 2020-12-28 18:18:22
3. 2020-12-28 18:20:25
4. 2020-12-28 18:20:27

提示: time.ctime() 获取当前时间, time.sleep() 间隔

```
1  from time import *
2
3  # 每行及时显示
4  file = open("my.log", 'a+', buffering=1)
5
6  # n = 行数 + 1
7  file.seek(0,0)
8  n = len(file.readlines()) + 1
9
10 while True:
11     tm = "%d. %s\n"%(n,ctime())
12     file.write(tm)
13     n += 1
14     sleep(2)
15
```

7.3 os模块

os模块是Python标准库模块，包含了大量的文件处理函数。

- 获取文件大小

```
1  os.path.getsize("my.log")
2
3  功能： 获取文件大小
4  参数： 指定文件
5  返回值： 文件大小
```

- 查看文件列表(有哪些文件)

```
1 os.listdir(dir)
2
3 功能： 查看文件列表
4 参数： 指定目录
5 返回值： 目录中的文件名列表
```

- 判断文件是否存在

```
1 os.path.exists(file)
2
3 功能： 判断文件是否存在
4 参数： 指定文件
5 返回值： 布尔值
```

- 删除文件

```
1 os.remove(file)
2 功能： 删除文件
3 参数： 指定文件
```

os模块使用示例：file.py

```
1 import os
2
3 print("文件大小:",os.path.getsize("file.txt"))
4 print("文件列表:",os.listdir("."))
5 print("文件是否存在:",os.path.exists("file.txt"))
6
7 os.mkdir(dir_name)    #创建一个文件夹
8 os.makedirs(dir_name) #函数创建层级文件夹
9
10 os.remove("file.txt") #删掉一个文件，需要判断是否存在
11 os.rmdir("test")    #需要空的时候才能删除
```

使用python删除一个文件或文件夹

https://blog.csdn.net/weixin_36670529/article/details/112688471

```
1 # 删掉一个文件夹，尽管里面有东西
2 # 方法1
3 import shutil
4 shutil.rmtree("test")
5
6 # 方法2
7 import os
8 os.removedirs("test")
```

复制几个文件内容 -> 1个大文件union.txt


```

1  # 复制几个文件内容 -> 1个文件
2
3  import os
4
5  dir="../test/"
6
7  fw=open("union.txt","w")
8
9  for file in os.listdir(dir):
10     fr=open(dir+file)      #路径注意!!!
11     fw.write(fr.read())    #复制一个文件
12     fr.close()
13
14  fw.close()

```

7.5 pathlib模块

1. 1个文件的信息

```

1  """
2  文件管理
3  """
4  from pathlib import Path
5
6  # 文件存不存在
7  print(Path("Demo01.py").exists())
8  print(Path("../Day18/Demo01.py").exists())
9
10 #绝对路径
11 print(Path.cwd())
12 print(Path.cwd().joinpath('Demo06.py')) # 拼接路径
13
14 # 显示当前目录下的所有信息 文件和目录
15 res = Path.cwd().iterdir()
16 for item in res:
17     print(item)
18
19 # glob()只找1层
20 # 当前目录找所有含Demo的文件
21 for item in Path.cwd().glob("*Demo*"):
22     print(item)
23
24 print('-----')
25 # parent 父级目录
26 for item in Path.cwd().parent.glob("*/Demo*"):
27     print(item)
28

```

```

29 print('-----')
30
31 # 递归搜索(所有层)
32 for item in Path.cwd().parent.rglob("*.py"):
33     print(item)

```

```

1  from pathlib import Path
2  from datetime import datetime
3
4  # 创建了一个指向Demo01.py的Path对象
5  p = Path('Demo07.py')
6  # 文件名
7  print(p.name)
8  # 是否是文件
9  print(p.is_file())
10 # 文件大小
11 print(item.stat().st_size)
12 # 时间戳 最近修改的时间
13 print(p.stat().st_ctime)
14 # 时间戳 最近看的时间
15 print(p.stat().st_atime)
16
17 # timestamp -> 时间对象
18 print(datetime.fromtimestamp(p.stat().st_ctime)) # 2024-10-08 20:33:17.103020
19
20 a=datetime.fromtimestamp(p.stat().st_ctime)
21 print(datetime.strftime(a, '%Y-%m-%d %H:%M:%S')) # 2024-10-08 20:34:51

```

练习: 搜索文件

```

1  """
2  打印出所有的图片 .png结尾
3  打印出所有文本 文件的ctime 年月日 时分秒
4  """
5  from pathlib import Path
6  from datetime import datetime
7
8  for item in Path.cwd().parent.rglob("*.png"):
9      print(item)
10
11 for item in Path.cwd().parent.rglob("*.txt"):
12     print(datetime.fromtimestamp(item.stat().st_ctime).strftime("%Y年%m月%d日 %H时%M分%S秒"))

```

2. 目录文件操作

```
1  from pathlib import Path
2
3  # 1.增
4  # 文件
5  Path("B.txt").touch()
6  # 目录
7  # Path("test").mkdir() # 创建已经存在的目录会报错
8  Path("test").mkdir(exist_ok=True) # 不会报错
9
10 # 2.删
11 Path("B.txt").unlink()
12 Path("test").rmdir()# 如果不存在就会报错
13 # 解决:
14 # target = Path("test")
15 # if target.exists():
16 #     target.rmdir()
17 #
18 # 3. 改
19 Path("A.txt").rename("AA.txt")
```

练习

```
1  """
2  1. 创建目录aid2407 在这个目录下创建class.txt
3  2. 修改class.txt 为 stu.txt
4  3. 删除目录aid2407
5  """
6  from pathlib import Path
7
8  # Path("aid2407").mkdir()
9  # Path("aid2407/class.txt").touch()
10 # Path("aid2407/class.txt").rename("aid2407/stu.txt")
11
12 # Path("aid2407/stu.txt").unlink()
13 # Path("aid2407").rmdir()
```