1. 闭包

1.1掌握闭包的用途和用法

```python
account_amount = 0      # 账户余额
def atm(num, deposit=True):
    global account_amount
    if deposit:
        account_amount += num
        print(f"存款:+{num}，账户余额:{account_amount}")
    else:
        account_amount -= num
        print(f"取款:-{num}，账户余额:{account_amount}")


atm(300)
atm(300)
atm(100, False)
```

```
D:\dev\Python\Python310\
存款:+300，账户余额:300
存款:+300，账户余额:600
取款:-100，账户余额:500
```

通过全局变量account_amount来记录余额

尽管功能实现是ok的，但是仍有问题：

- 代码在命名空间上（变量定义）不够干净、整洁
- 全局变量有被修改的风险

闭包的功能：**解决全局变量被修改的风险**

闭包的定义：

- 在函数嵌套的前提下，内部函数是用来外部函数的变量，并且外部函数返回了内部函数，我们把这个使用外部函数变量的内部函数称为闭包。

```python
account_amount = 0      # 账户余额
def atm(num, deposit=True):
    global account_amount
    if deposit:
        account_amount += num
        print(f"存款:+{num}，账户余额:{account_amount}")
    else:
        account_amount -= num
        print(f"取款:-{num}，账户余额:{account_amount}")


atm(300)
atm(300)
atm(100, False)
```

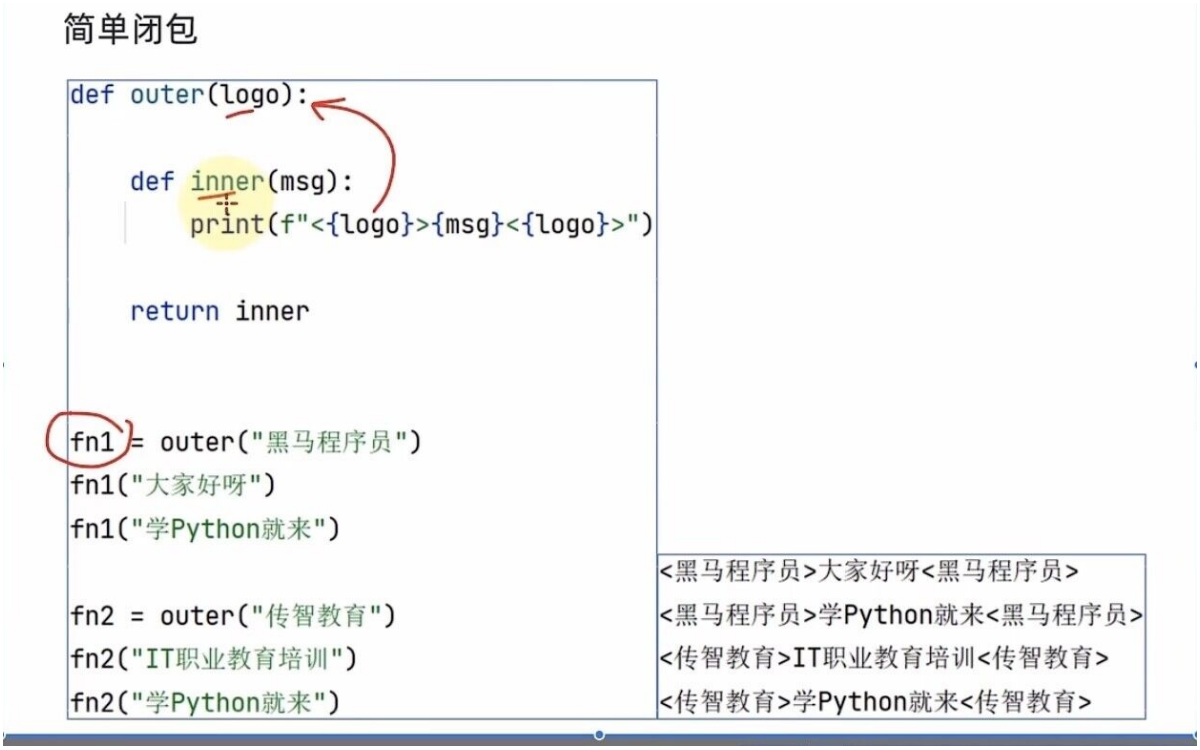


```python
def account_create(initial_amount=0):
    def atm(num, deposit=True):
        nonlocal initial_amount
        if deposit:
            initial_amount += num
            print(f"存款:+{num}，账户余额:{initial_amount}")
        else:
            initial_amount -= num
            print(f"取款:-{num}，账户余额:{initial_amount}")

    return atm


fn = account_create()
fn(300)
fn(200)
fn(300, False)
```

```
D:\dev\Python\Python31
存款:+300，账户余额:300
存款:+200，账户余额:500
取款:-300，账户余额:200
```



- 巧妙点：返回的是inner函数

1.2 掌握nonlocal关键字的作用

- 使用nonlocal关键字修改外部函数的值，定义在内部函数

```
1  def outer(num1):
2
3      def inner(num2):
4          nonlocal num1 #没有这个语句。无法修改闭包外部函数的值,定义在内部函数
5          num1 += num2
6          print(num1)
7
8  fn = outer(10)
9  fn(20)
10 >>>30
```

1.3 闭包的注意事项

优点，使用闭包可以让我们得到：

- 无需定义全局变量即可实现通过函数，持续的访问，修改某个值
- 闭包使用的函数在所用于函数内，难以被错误的**调用修改**

缺点：

- 由于内部函数持续引用外部函数的值，所以会导致这一部分内存空间不被释放，一直占用内存。

1.4 总结

1.什么是闭包

定义双层嵌套函数，**内层函数可以访问外层函数的变量**

将内存函数作为外层函数的返回，此内层函数就是闭包函数

2.nonlocal关键字的作用

在闭包函数中想要修改外部函数的变量值需要用Nonlocal声明这个外部变量

2. 装饰器

1.掌握装饰器的作用和用法

装饰器定义：装饰器其实也是一种闭包，其功能就是在不破坏目标函数原有代码和功能的前提下，为目标函数增加新功能。

装饰器的一般写法(闭包写法)

```
def outer(func):
    def inner():
        print("我要睡觉了")
        func()
        print("我起床了")

    return inner


def sleep():
    import random
    import time
    print("睡眠中.....")
    time.sleep(random.randint(1, 5))


fn = outer(sleep)
fn()
```

定义一个闭包函数，在闭包函数内部：

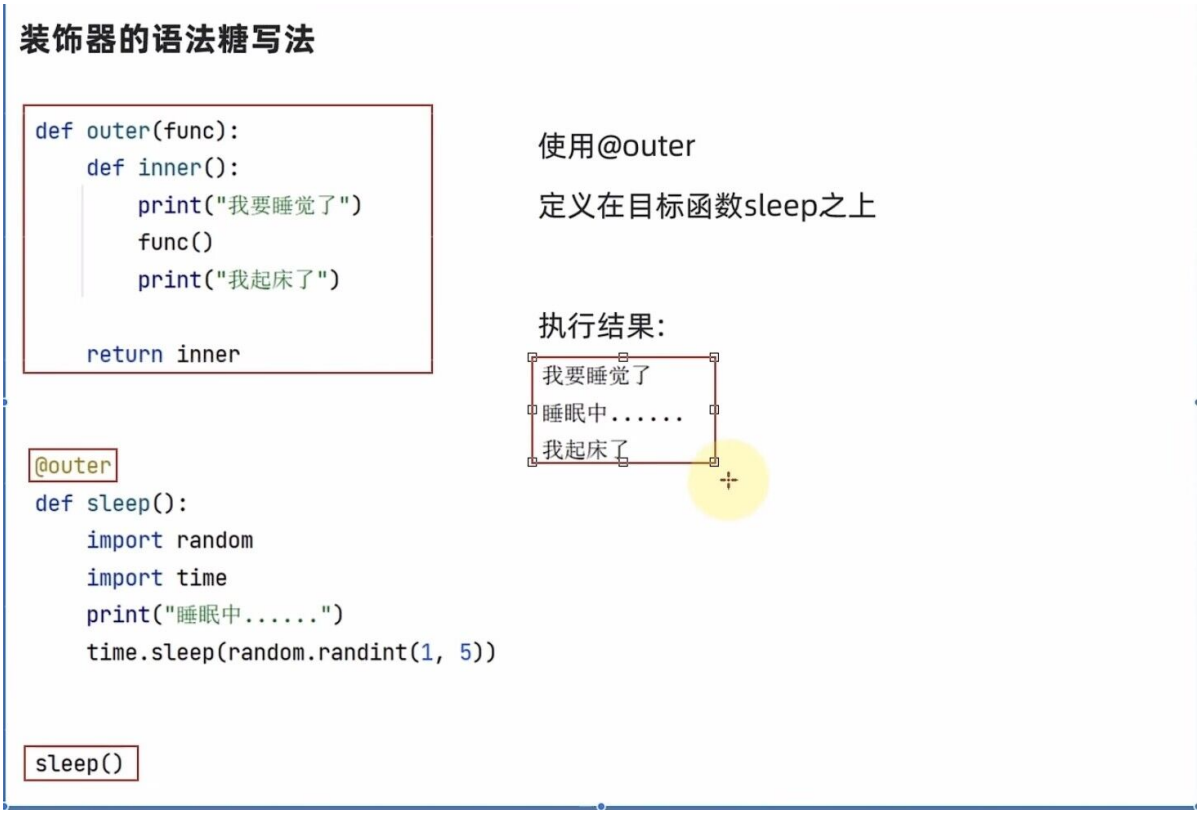
- 执行目标函数
- 并完成功能的添加

执行结果：

```
我要睡觉了
睡眠中.....
我起床了
```

本质上还是使用inner函数

2.装饰器的语法糖写法




```python
def outer(func):
    def inner():                          #定义一个闭包函数，在闭包函数内部：
        print("我要睡觉了")
        func()
        print("我要起床了")

    return inner
@outer
def sleep():
    import random
    import time
    print("...睡眠中")
    time.sleep(random.randint(1,5))

sleep()
```

3.什么是装饰器

装饰器就是使用创建一个闭包函数，在闭包函数内调用目标函数。

可以达到不改动目标函数的同时，增加额外的功能。

3. 设计模式

设计模式是一种编程的套路，可以极大的方便程序的开发。最常见最经典的就是面向对象编程。

1. 单例模式

达到效果：无论创建多少对象，只保存一个实例存在。

单例模式

```
class Tool:
    pass


t1 = Tool()
t2 = Tool()
print(t1)
print(t2)
```

创建类的实例后，就可以得到一个完整的、独立的类对象。

通过print语句可以看出，它们的内存地址是不相同的，即t1和t2是完全独立的两个对象。

```
<__main__.Tool object at 0x0000027246FCCEE0>
<__main__.Tool object at 0x0000027246FCD9F0>
```

某些场景下，我们需要一个类无论获取多少次类对象，都仅仅提供一个具体的实例

用以节省创建类对象的开销和内存开销

比如某些工具类，仅需要1个实例，即可在各处使用

这就是单例模式所要实现的效果。

单例模式（Singleton Pattern）是一种常用的软件设计模式，该模式的主要目的是确保某一个类只有一个实例存在。

在整个系统中，某个类只能出现一个实例时，单例对象就能派上用场。

- 定义：保证一个类只有一个实例,并提供一个访问它的全局访问点
- 适用场景:当一个类只能有一个实例，而客户可以从一个众所周知的访问点访问它时。

单例的实现模式：

```
class StrTools:
    pass
```

```
str_tool = StrTools()
```

在一个文件中定义如上代码

```
from test import str_tool
```

```
s1 = str_tool
s2 = str_tool
print(s1)
print(s2)
```

在另一个文件中导入对象

```
<test.StrTools object at 0x0000013D001DB910>
<test.StrTools object at 0x0000013D001DB910>
```

s1和s2是同一个对象

2. 工厂模式

当需要大量创建一个类的实例的时候，可以使用工厂模式。

即，从原生的使用类的构造去创建对象的形式

迁移到，基于工厂提供的方法去创建对象的形式。

```
class Person:
    pass

class Worker(Person):
    pass
class Student(Person):
    pass
class Teacher(Person):
    pass

worker = Worker()
stu = Student()
teacher = Teacher()
```

```
class Person:
    pass

class Worker(Person):
    pass
class Student(Person):
    pass
class Teacher(Person):
    pass

class Factory:
    def get_person(self, p_type):
        if p_type == 'w':
            return Worker()
        elif p_type == 's':
            return Student()
        else:
            return Teacher()

factory = Factory()
worker = factory.get_person('w')
stu = factory.get_person('s')
teacher = factory.get_person('t')
```

```
class Person:
    pass

class Worker(Person):
    pass
class Student(Person):
    pass
class Teacher(Person):
    pass

class Factory:
    def get_person(self, p_type):
        if p_type == 'w':
            return Worker()
        elif p_type == 's':
            return Student()
        else:
            return Teacher()

factory = Factory()
worker = factory.get_person('w')
stu = factory.get_person('s')
teacher = factory.get_person('t')
```

- 使用工厂类的get_person()方法去创建具体的类对象

优点：

- 大批量创建对象的时候有统一的入口，易于代码维护
- 当发生修改，仅修改工厂类的创建方法即可
- 符合现实世界的模式，即由工厂来制作产品（对象）

4.多线程

1.进程，线程和并行执行

1.1 了解什么是进程，线程

- 进程：一个程序，运行在操作系统之上，那么便称之这个程序为一个运行进程，并分配进程ID方便系统管理。
- 线程：线程是属于进程的，一个进程可以开启多个线程，执行不同的工作，是进程实际工作最小单位。
- 例子：

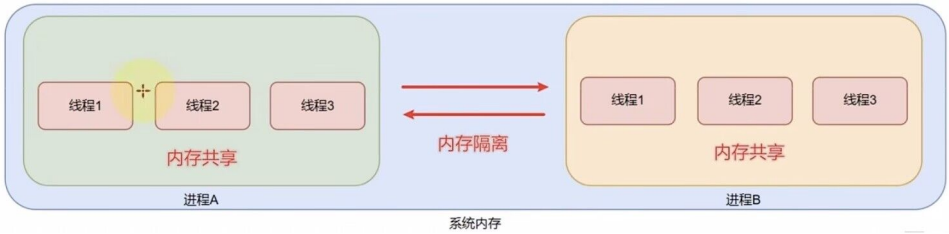
- 进程好比一家公司，是操作系统对程序进行运行管理的单位
- 线程就好比公司的员工，进程可以有多个线程(员工)，是进程实际的工作者
- 操作系统中可以运行多个进程，即多任务运行
- 一个进程中可以运行多个线程，即多线程运行

- **进程、线程**

  注意点：

  进程之间是内存隔离的，即不同的进程拥有各自的内存空间。这就类似于不同的公司拥有不同的办公场所。

  线程之间是内存共享的，线程是属于进程的，一个进程内的多个线程之间是共享这个进程所拥有的内存空间的。这就好比，公司员工之间是共享公司的办公场所。



1.2 了解什么是并行执行

并行执行的意思指的是同一时间做不同的工作。

进程之间就是并行执行的，操作系统可以同时运行好多程序，这些程序都是在并行执行。

除了进程外，线程其实也是可以并行执行的。

也就是比如一个Python程序，其实是完全可以做到：

- 一个线程在输出：你好
- 一个线程在输出：Hello

像这样一个程序在同一时间做两件乃至多件不同的事情，我们就称之为：多线程并行执行

1.3总结

- 什么是进程
  - 程序在操作系统内运行，即成为一个运行进程
- 什么是线程
  - 进程内部可以有多个线程，程序的运行本质上就是由进程内部的线程在实际工作的。
- 什么是并行执行
  - 多个进程在同时运行，即不同的程序同时运行，称之为：**多任务并行执行**
  - 一个进程内的多个线程在同时运行，称之为：**多线程并行执行**

2.多线程编程

2.1 掌握使用threading模块完成多线程编程

threading模块

绝大所数编程语言，都允许多线程编程，Python也不例外。Python的多线程可以通过threading模块来实现。

```
import threading

thread_obj = threading.Thread([group [, target [, name [, args [, kwargs]]]]])
- group: 暂时无用，未来功能的预留参数
- target: 执行的目标任务名
- args: 以元组的方式给执行任务传参
- kwargs: 以字典方式给执行任务传参
- name: 线程名，一般不用设置

# 启动线程，让线程开始工作
thread_obj.start()
```

- args参数通过元组(按参数顺序)的方式传参
- 或使用kwargs参数用字典的形式传参
- ```python
  import threading
  import time

  def sing(msg):
      while True:
          print(msg)
          time.sleep(1)

  def dance(msg):
      while True:
          print(msg)
          time.sleep(1)

  sing_thread = threading.Thread(target=sing,args=("我要唱歌，哈哈哈",))#元组形式传参
  dance_thread = threading.Thread(target=dance,kwargs={"msg":"我在跳舞，啦啦啦啦啦"})#字典形式传参

  sing_thread.start()
  dance_thread.start()
  #输出
  我要唱歌，哈哈哈
  我在跳舞，啦啦啦啦啦啦
  我在跳舞，啦啦啦啦啦啦我要唱歌，哈哈哈

  我在跳舞，啦啦啦啦啦啦我要唱歌，哈哈哈

  我在跳舞，啦啦啦啦啦啦我要唱歌，哈哈哈
  ```
-
  ```

2.2多线程编程程序

```
import threading
import time

'''
让一个PYTHON程序实现启动两个线程
每个线程执行一个函数，执行不同的功能
'''
def sing():
    while True:
        print('我在唱歌，啦啦啦')
        time.sleep(1)

def dance():
    while True:
        print('我在跳舞，哗哗哗')
        time.sleep(1)

sing_thread = threading.Thread(target=sing)#threading模块中的Thread类，实例化了一个类对象
dance_thread = threading.Thread(target=dance)

sing_thread.start()
dance_thread.start()
#多线程输出
我在唱歌，啦啦啦
我在跳舞，哗哗哗
我在跳舞，哗哗哗我在唱歌，啦啦啦

我在唱歌，啦啦啦
我在跳舞，哗哗哗
我在唱歌，啦啦啦
```

单线程与多线程对比

```
import time

'''
让一个PYTHON程序实现启动两个线程
每个线程执行一个函数，执行不同的功能
'''
def sing():
    while True:
        print('我在唱歌，啦啦啦')
        time.sleep(1)

def dance():
    while True:
        print('我在跳舞，哗哗哗')
        time.sleep(1)

sing()
dance()
```

```
#单线程输出（只会执行第一个函数）
我在唱歌，啦啦啦
我在唱歌，啦啦啦
我在唱歌，啦啦啦
我在唱歌，啦啦啦
```

2.3 总结

1.threading模块的使用

`thread_obj = threading.Thread(target=func)`创建线程对象

`thread_obj.start()`启动线程执行

2.如何传参

```
sing_thread = threading.Thread(target=sing,args=("我要唱歌，哈哈哈",))
dance_thread = threading.Thread(target=dance,kwargs={"msg":"我在跳舞，啦啦啦啦啦啦"})

sing_thread.start()
dance_thread.start()
```

5.网络编程

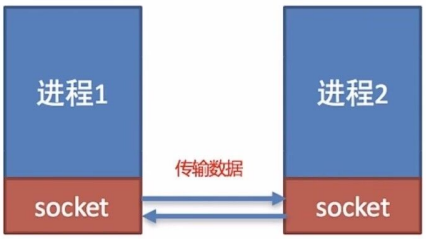
5.1服务端开发

1. 了解什么是Socket网络编程

Socket

socket (简称 套接字) 是进程之间通信一个工具，好比现实生活中的插座，所有的家用电器要想工作都是基于插座进行，进程之间想要进行网络通信需要socket。

Socket负责进程之间的网络数据传输，好比数据的搬运工。



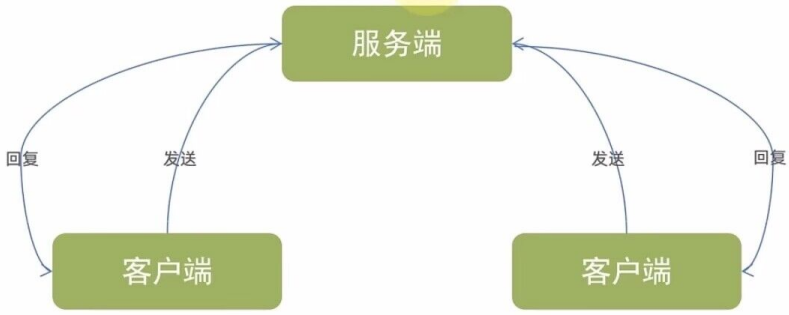
大多数软件都使用到了Socket进行网络通讯

客户端和服务端

2个进程之间通过Socket进行相互通讯，就必须有服务端和客户端

Socket服务端：等待其它进程的连接、可接受发来的消息、可以回复消息

Socket客户端：主动连接服务端、可以发送消息、可以接收回复



2. 基于Socket完成服务端程序开发

socket服务端编程

主要分为如下几个步骤：

1. 创建socket对象

```
import socket
socket_server = socket.socket()
```

2. 绑定socket_server到指定IP和地址

```
socket_server.bind(host, port)
```

3. 服务端开始监听端口

```
socker_server.listen(backlog)
# backlog为int整数，表示允许的连接数量，超出的会等待，可以不填，不填会自动设置一个合理值
```

4. 接收客户端连接，获得连接对象

```
conn, address = socket_server.accept()
print(f"接收到客户端连接，连接来自：{address}")
# accept方法是阻塞方法，如果没有连接，会卡再当前这一行不向下执行代码
# accept返回的是一个二元元组，可以使用上述形式，用两个变量接收二元元组的2个元素
```

5. 客户端连接后，通过recv方法，接收客户端发送的消息

```
while True:
    data = conn.recv(1024).decode("UTF-8")
    # recv方法的返回值是字节数组（Bytes），可以通过decode使用UTF-8解码为字符串
    # recv方法的传参是buffsize，缓冲区大小，一般设置为1024即可
    if data == 'exit':
        break
    print("接收到发送来的数据：", data)
# 可以通过while True无限循环来持续和客户端进行数据交互
# 可以通过判定客户端发来的特殊标记，如exit，来退出无限循环
```

6. 通过conn（客户端当次连接对象），调用send方法可以回复消息

```
while True:
    data = conn.recv(1024).decode("UTF-8")
    if data == 'exit':
        break
    print("接收到发送来的数据：", data)

    conn.send("你好呀哈哈哈".encode("UTF-8"))
```

7. conn（客户端当次连接对象）和socket_server对象调用close方法，关闭连接

```python
import socket

#1.创建一个socket对象
socket_server = socket.socket()
#2.绑定socker_sever的指定IP和端口,绑定服务端的IP地址和端口
socket_server.bind(('localhost',8888))
# 3.监听端口
socket_server.listen(1)
#listen方法内接受一个整数传参数，表示接受的链接数量
#4.等待客户端链接
# result: tuple = socket_server.accept() #rusult接收到的是一个二元元组
# conn = result[0] # 客户端和服务端的链接对象
# adress = result[1] #客户端的地址信息
conn,adress = socket_server.accept()
# accept方法返回的的是二元元组(链接对象,客户端地址信息)
# 可以通过 变量1,变量2 = socket_server.accept()的形式直接接收二元元组内的两个元素
#accept()方法，是阻塞的方法，等待客户端的连接，如果没有链接，就卡在这一行不向下执行了
print(f"接收到了客户端的链接，客户端的连接地址信息是:{adress}")
# 5.接收客户端的信息,要使用客户端和服务端的本次链接对象，而非socker_server对象
while True:
    data: str = conn.recv(1024).decode('utf-8')
    # recv接受的参数是缓冲区大小，一般给1024即可
    # recv方法的返回值是一个字节数组也就是bytes对象，不是字符串，可以通过decode方法通过UTF-8编码，将字节数组转化为字符串对象
    print(f'客服端发来的消息是:{data}')
    # 6.发送回复消息
    msg = input("请输入你要和客户端回复的消息:").encode('utf-8')#encode可以将字符串编码为字节数组对象
    if msg = 'exit':
        break
    conn.send(msg)
# 7.关闭链接
conn.close()
socket_server.close()
```

```
33
```

5.2 客户端开发

1. 基于Socket完成客户端程序开发

Socket客户端编程

主要分为如下几个步骤:

1. 创建socket对象

```
import socket
socket_client = socket.socket()
```

2. 连接到服务端

```
socket_client.connect(("localhost", 8888))
```

3. 发送消息

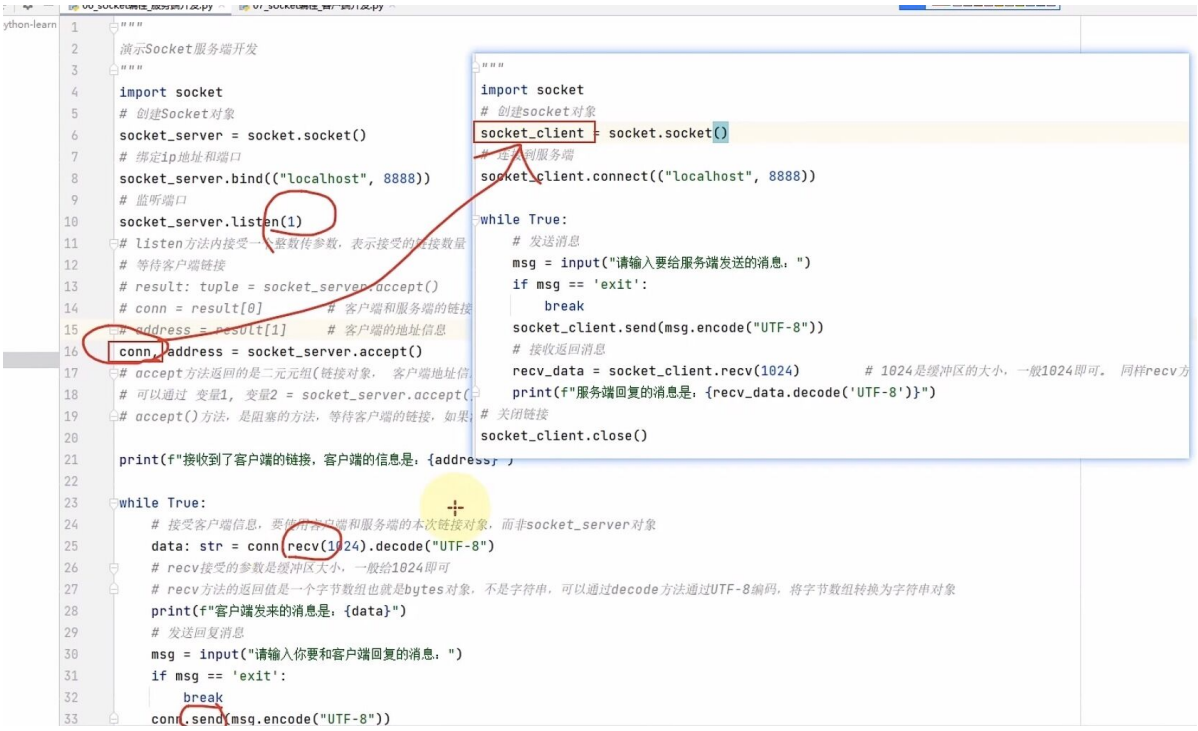
```
while True:     # 可以通过无限循环来确保持续的发送消息给服务端
    send_msg = input("请输入要发送的消息")
    if send_msg == 'exit':
        # 通过特殊标记来确保可以退出无限循环
        break
    socket_client.send(send_msg.encode("UTF-8"))    # 消息需要编码为字节数组（UTF-8编码）
```

```python
import socket
# 1.创建一个socket对象
socket_client = socket.socket()
# 2.链接到服务端
socket_client.connect(('localhost',8888))

while True:
    # 3.发送消息
    send_msg = input("请输入要发送的消息")
    if send_msg == "exit":
        break
    socket_client.send(send_msg.encode('utf-8'))

    # 4.接受返回消息
    recv_data = socket_client.recv(1024) #1024是缓冲区大小，一般1024即可
    #recv方法是阻塞式的，既不接收到返回，就卡在这里
    print('服务端回复的消息为:',recv_data.decode("utf-8"))

# 5. 关闭链接
    socket_client.close()
```



```python
"""
演示Socket服务端开发
"""
import socket
# 创建Socket对象
socket_server = socket.socket()
# 绑定ip地址和端口
socket_server.bind(("localhost", 8888))
# 监听端口
socket_server.listen(1)
# listen方法内接受一个整数传参数，表示接受的连接数量
# 等待客户端链接
# result: tuple = socket_server.accept()
# conn = result[0]        # 客户端和服务端的链接
# address = result[1]     # 客户端的地址信息
conn, address = socket_server.accept()
# accept方法返回的是二元元组(链接对象，客户端地址信
# 可以通过 变量1, 变量2 = socket_server.accept(
# accept()方法，是阻塞的方法，等待客户端的链接，如果

print(f"接收到了客户端的链接，客户端的信息是：{address}")

while True:
    # 接受客户端信息，要使用客户端和服务端的本次链接对象，而非socket_server对象
    data: str = conn.recv(1024).decode("UTF-8")
    # recv接受的参数是缓冲区大小，一般给1024即可
    # recv方法的返回值是一个字节数组也就是bytes对象，不是字符串，可以通过decode方法通过UTF-8编码，将字节数组转换为字符串对象
    print(f"客户端发来的消息是：{data}")
    # 发送回复消息
    msg = input("请输入你要和客户端回复的消息：")
    if msg == 'exit':
        break
    conn.send(msg.encode("UTF-8"))
```

```python
"""
import socket
# 创建socket对象
socket_client = socket.socket()
# 连接到服务端
socket_client.connect(("localhost", 8888))

while True:
    # 发送消息
    msg = input("请输入要给服务端发送的消息：")
    if msg == 'exit':
        break
    socket_client.send(msg.encode("UTF-8"))
    # 接收返回消息
    recv_data = socket_client.recv(1024)        # 1024是缓冲区的大小，一般1024即可。 同样recv方
    print(f"服务端回复的消息是：{recv_data.decode('UTF-8')}")
# 关闭链接
socket_client.close()
```

6.正则表达式

6.1 基础匹配

1. 了解什么是正则表达式

正则表达式，又称规则表达式，是使用单个字符串来描述，匹配某个句法规则的字符串，常被用来检索，替换那些符合某个模式(规则)的文本。

简单来说，正则表达式就是使用：字符串定义规则，并通过规则去验证字符串是否匹配。

正则的三个基础方法

- match匹配

Python正则表达式，使用re模块，并基于re模块中三个基础方法来做正则匹配。

分别是: match、search、findall三个基础方法

- re.match(匹配规则，被匹配字符串)

从被匹配字符串开头进行匹配，匹配成功返回匹配对象（包含匹配的信息），匹配不成功返回空

```python
s = 'python itheima python itheima python itheima'

result = re.match('python', s)
print(result)  # <re.Match object; span=(0, 6), match='python'>
print(result.span())  # (0, 6)
print(result.group())  # python
```

```python
s = '1python itheima python itheima python itheima'

result = re.match('python', s)
print(result) # None
```

```python
import re
s = "python itheima"
#1. match从头匹配,头部不匹配，后面也不理会
result = re.match("python",s)
print(type(result))
print(result.span())

```

- search(匹配规则，被匹配字符串)

搜索整个字符串，找出匹配的。从前向后，找到第一个后，就停止，不会继续向后

```python
s = '1python666itheima666python666'

result = re.search('python', s)
print(result)           # <re.Match object; span=(1, 7), match='python'>
print(result.span())    # (1, 7)
print(result.group())   # python
```

整个字符串都找不到，返回None

```python
s = 'itheima666'

result = re.search('python', s)
print(result)           # None
```

- findall(匹配规则，被匹配字符串)

匹配整个字符串，找出全部匹配项

```python
s = '1python666itheima666python666'

result = re.findall('python', s)
print(result)           # ['python', 'python']
```

找不到返回空list: []

```python
s = '1python666itheima666python666'

result = re.findall('itcast', s)
print(result)           #[]
```

2. 掌握re模块的基础使用

- re.match,从头开始匹配，匹配第一个命中项
- re.search，全局匹配，匹配第一个命中项
- re.findall，全局匹配，匹配全部命中项

6.2 元字符匹配

1. 掌握正则表达式的各类元字符规则

元字符匹配

在刚刚我们只是进行了基础的字符串匹配，正则最强大的功能在于元字符匹配规则。

单字符匹配：

字符	功能
.	匹配任意1个字符（除了\n），\. 匹配点本身
[]	匹配[]中列举的字符
\d	匹配数字，即0-9
\D	匹配非数字
\s	匹配空白，即空格、tab键
\S	匹配非空白
\w	匹配单词字符，即a-z、A-Z、0-9、_
\W	匹配非单词字符

示例：

字符串 s = “itheima1 @@python2 !!666 ##itcast3”

- 找出全部数字：re.findall(r‘\d’, s)

字符串的r标记，表示当前字符串是原始字符串，即内部的转义字符无效而是普通字符

- 找出特殊字符：

- 找出全部英文字母：

re.findall(r’[a-zA-Z]’, s)

[]内可以写：[a-zA-Z0-9] 这三种范围组合或指定单个字符如[aceDFG135]

2. 了解字符串的r标记作用

7.递归