第六周学习报告

2. 用 VS Code 打开项目目录，新建一个 `environment.yml` 文件，指定安装 Python 3.12，然后运行 `conda env create` 命令创建 Conda 环境

```
(base)
李意如@LAPTOP-9J8HOMDD MINGW64 /c/Users/李意如/repo/week06 (main)
$ cp ../week05/environment.yml ./
```

```
(base)
李意如@LAPTOP-9J8HOMDD MINGW64 /c/Users/李意如/repo/week06 (main)
$ conda env create
```

检查自己有哪些环境

conda env list

想删除哪个环境:conda remove -n xxx

想换成哪个环境：conda activate xxx

换成以后 conda list 查看

3. 创建一个 `guessing_game.py` 文件，复制粘贴以下代码，运用 `pdb` 调试器理解其运行流程：

- 在 `(pdb)` 提示符下练习使用 `l` (显示代码)、`n` (执行当前行)、`p` (打印表达式)、`s` (步入调用)、`pp` (美观打印)、`c` (继续执行) 等命令 (参考文档)

```
(week06)
李意如@LAPTOP-9J8HOMDD MINGW64 /c/Users/李意如/repo/week06 (main)
$ python -m pdb guessing_game.py
> c:\users\李意如\repo\week06\guessing_game.py(1)<module>()
-> import random
(Pdb) l
  1  -> import random
  2
  3
  4     def guessing_game():
  5         # 生成 1 到 100 之间的随机整数
  6         secret_number = random.randint(1, 100)
  7         n = 0
  8
  9         print("欢迎来到猜数字游戏！我已经想好了一个 1 到 100 之间的数字，你可
 10
 11         while True:
(Pdb)
```

l 是 list 的缩写 ，作用是列出当前正在调试的代码片段。默认会显示以当前执行行为中心的周围几行代码，方便查看上下文，了解代码执行位置和周边逻辑。例如在你当前调试过程中，多次使用 l 可查看不同位置代码情况。

l . 同样是列出代码的命令。这里的 . 表示当前执行位置，l .会精准列出包含当前执行行的代码片段，也是为了帮助你清晰掌握当前调试所在的代码上下文环境，本质和 l 类似，但定位当前行更明确。

```
(Pdb) import wat
(Pdb) wat / str.strip
value: <method 'strip' of 'str' objects>
type: method_descriptor
signature: def strip(self, chars=None, /)
"""
Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.
"""
```

这是在 Python 的 Pdb 调试环境中使用 `wat` 工具查询 `str.strip` 方法信息 。`wat` 是一个第三方调试增强工具，导入后可用于查询函数、方法等对象的详细信息。这里执行 `wat / str.strip` ，获取并展示了 `str.strip` 方法的类型（`method_descriptor` ）、签名（接收 `self` 、`chars` 参数 ）及功能描述（去除字符串首尾空白字符，指定 `chars` 时则去除其中字符 ）。

4. 创建一个 `flow_controls.py` 文件，让豆包 (或 DeepSeek 等任何大模型) 生成例子，尝试运行，体会理解以下 Python 流程控制语句：

- `for` 迭代循环 (iteration loop)
- `while` 条件循环 (conditional loop)
- `break` 打断跳出循环
- `continue` 跳至下一轮循环
- `for...else` 循环未被打断的处理
- `if` 条件分支
- `if...elif[...elif]` 多重条件分支
- `if...else` 未满足条件的处理
- `try...except[...except...else...finally]` 捕捉异常的处理
- `raise` 主动抛出异常

（1）for

示例 1 遍历列表　其中第三行也可以写为 fruit+=”,ok”

environment.yml U　guessing_game.py U　flow_controls.py U

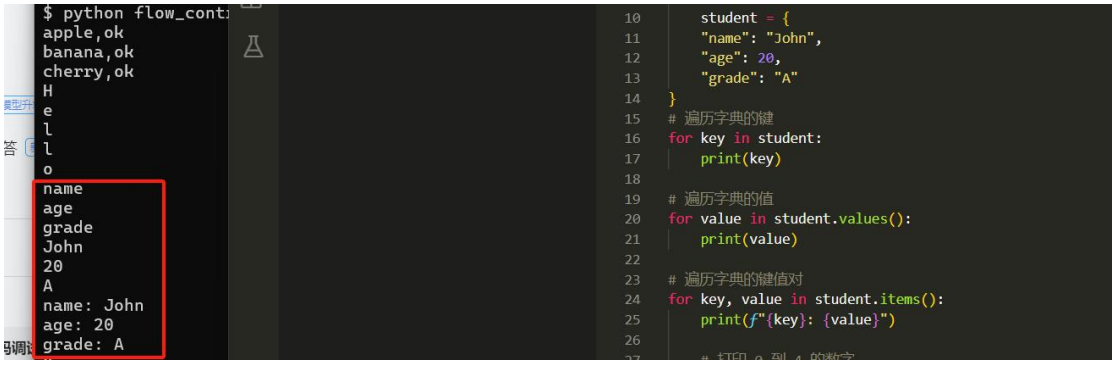
flow_controls.py > ...

```
1  fruits = ["apple", "banana", "cherry"]
2  for fruit in fruits:
3      fruit=fruit+",ok"
4      print(fruit)
```

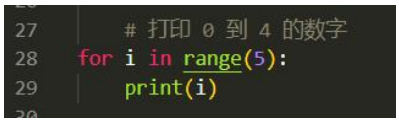
示例 2 遍历字符串

```
6      message = "Hello"
7  for char in message:
8      print(char)
```

示例 3 遍历字典



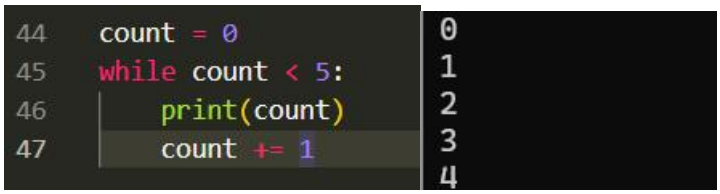
示例 4 使用 range() 函数



示例 5 嵌套 for 循环

（2）while

示例 1 简单计数循环



示例 2 使用 continue 语句

示例 3 循环中的 else 语句

（3）for...else 循环未被打断的处理

示例 1：检查列表中是否有偶数

```python
numbers = [1, 3, 5, 7, 9]
for num in numbers:
    if num % 2 == 0:
        print(f"找到了偶数 {num}")
        break
else:
    print("列表中没有偶数")
```

在这个例子中，for 循环遍历 numbers 列表。if 语句会检查当前元素是否为偶数，若找到偶数就会打印该偶数并使用 break 跳出循环，此时 else 代码块不会执行。要是列表里没有偶数，for 循环会正常结束，进而执行 else 代码块，输出 “列表中没有偶数”。

（4）try...except[...except...else...finally] 捕捉异常的处理

```python
try:
    num = int(input("请输入一个整数: "))
    result = 10 / num
    print(f"结果是: {result}")
except ValueError:
    print("输入的不是有效的整数，请重新输入。")
except ZeroDivisionError:
    print("除数不能为零，请输入一个非零的整数。")
```

```
请输入一个整数： 0
除数不能为零，请输入一个非零的整数。
```

在这个例子中，try 块里尝试将用户输入转换为整数，并进行除法运算。如果用户输入的不是有效的整数，会触发 ValueError；如果输入的是 0，会触发 ZeroDivisionError。不同的异常会被相应的 except 块捕获并处理。

（5）raise 主动抛出异常

```python
# 定义一个函数用于计算两个数的商
def divide_numbers(a, b):
    if b == 0:
        # 当除数为 0 时，主动抛出 ZeroDivisionError 异常
        raise ZeroDivisionError("除数不能为零")
    return a / b

try:
    result = divide_numbers(10, 0)
    print(result)
except ZeroDivisionError as e:
    print(f"捕获到异常: {e}")
```

在这个例子中，`divide_numbers` 函数用于计算两个数的商。当除数 `b` 为 `0` 时，使用 `raise` 语句抛出 `ZeroDivisionError` 异常，并附带错误信息。在 `try` 块中调用该函数，由于除数为 `0`，会触发异常，被 `except` 块捕获并输出错误信息。

（6）python 用 try 和 raise 配合流程控制，举例子

```python
def get_valid_integer():
    while True:
        try:
            user_input = input("请输入一个正整数: ")
            num = int(user_input)
            if num <= 0:
                raise ValueError("输入的数必须是正整数。")
            return num
        except ValueError as e:
            print(f"输入无效: {e}")


valid_num = get_valid_integer()
print(f"你输入的有效正整数是: {valid_num}")

```

```
请输入一个正整数: 2.5
输入无效: invalid literal for int() with base 10: '2.5'
请输入一个正整数: 0
输入无效: 输入的数必须是正整数。
请输入一个正整数: 3
你输入的有效正整数是: 3
```

5. 创建一个 `mylib.py` **模块** (module)，在里面定义以下函数，再创建一个 `myjob.py` **脚本** (script)，从 `mylib.py` 导入函数并尝试调用：

   - 定义函数 `func1`，没有形参，没有返回值
   - 定义函数 `func2`，没有形参，有返回值
   - 定义函数 `func3`，只有一个 **位置形参** (positional parameter)，先尝试传入 **位置实参** (positional argument) 调用，再尝试传入 **命名实参** (named argument) 调用，再尝试不传实参 (会报错)
   - 定义函数 `func4`，只有一个 **命名形参** (named parameter)，先传入 **位置实参** 调用，再传入 **命名实参** 调用，再尝试不传实参 (取默认值)
   - 定义函数 `func5`，接受多个位置形参和命名形参，尝试以位置/命名各种不同方式传入实参，注意位置参数必须排在命名参数之前
   - 定义函数 `func6`，在形参列表中使用 `/` 来限定只接受位置实参的形参
   - 定义函数 `func7`，在形参列表中使用 `*` 来限定只接受命名实参的形参
   - 定义函数 `func8`，在位置形参的最后，在形参名称前使用 `*` 允许传入任意数量的位置实参 (被打包为元组)
   - 定义函数 `func9`，在命名形参的最后，在形参名称前使用 `**` 允许传入任意数量的命名实参 (被打包为字典)
   - 定义函数 `func10`，接受两个位置形参，一个命名形参，尝试在调用时使用 `*` 将可迭代对象 (如元组或列表) 自动解包，按位置实参传入
   - 定义函数 `func11`，接受一个命名形参，两个命名形参，尝试在调用时使用 `**` 将映射对象 (如字典) 自动解包，按命名实参传入
   - 定义函数 `func12`，给函数添加 **内嵌文档** (docstring)，给形参和返回值添加 **类型注解** (type annotation)，提高函数签名的可读性

func1

```python
def func1():
    x=50
    y=x**0.5-7
    print(y)
```

```python
import mylib

breakpoint()
```

调用

```python
import mylib

mylib.func1()
```

```
李意如@LAPTOP-9J8HOMDD MINGW64 /c/Users/李意如/repo/week06 (main)
$  python myjob.py
0.0710678118654755
```

6. 把 `mylib` 模块转变为 **软件包** (package) 安装进当前的 Conda 环境来使用

   - 把 `myjob.py` 脚本移动至 `scripts/myjob.py`，再次尝试运行，会发现 `import mylib` 失败，这是由于 `mylib` 并没有打包成 **软件包** (package) 安装
   - 将 `mylib.py` 模块移动至 `src/mypkg/mylib.py`，创建 `src/mypkg/__init__.py` 文件，准备好软件包的源代码
   - 创建 `pyproject.toml` 配置文件，按照 文档 填写基本的软件包信息
   - 在 `pyproject.toml` 配置文件里，按照 文档 填写软件包的 **构建** (build) 配置
   - 使用 `pip install -e .` 以本地可编辑模式把当前软件包安装进当前 Conda 环境
   - 修改 `environment.yml` 文件，使得 `conda env create` 自动安装本地可编辑软件包