Python 期末复习模拟卷笔记

目录

1 单选题

1.1 第 1 题

下面代码输出的结果是 ()

```
def func(num):
    num += 1
a = 10
func(a)
print(a)
```

A. 10

B. 11

C. int

D. 程序执行错误

【笔记】

正确答案：A

Python 中函数是传值调用，不影响原本的值。

1.2 第 2 题

切片操作 `list(range(6))[::-1]` 执行结果为 ()

A. [0, 2, 4, 6]

B. [6, 5, 4, 3, 2, 1]

C. [0, -1]

D. [5, 4, 3, 2, 1, 0]

【笔记】

正确答案：D

`list()` 将对象转换成列表类型，`[::-1]` 将列表反转。

1.3 第 3 题

下列选项中可以获取 Python 整数类型帮助的是 ()

A. help(float)

B. dir(float)

C. help(int)

D. dir(str)

1.4 第 4 题

以下选项中符合 Python 语言变量命名规则的是（）

A. it's

B. 3C

C. pass

D. _AI

【笔记】
正确答案：D

- A 选项中出现了' 符号，在 Python 中的意思是字符串；
- B 选项数字开头是不允许的；
- C 选项是 Python 关键字；
- D 选项符合 Python 变量名要求。

1.5 第 5 题

关于 Python 序列类型的通用操作符和函数，以下选项中描述错误的是（）

A. 如果 x 是 s 的元素，x in s 返回 True

B. 如果 s 是一个序列，s = [1,"kate",True]，s[3] 返回 True

C. 如果 s 是一个序列，s = [1,"kate",True]，s[-1] 返回 True

D. 如果 x 不是 s 的元素，x not in s 返回 True

【笔记】
正确答案：B
列表 s 的长度为 3，可以访问的索引为 0,1,2，如果访问 s[3] 会报错。

1.6 第 6 题

下列选项中不是 Python 保留字的是（）

A. False

B. True

C. do

D. class

【笔记】
正确答案：C
do 是 C 语言关键字，Python 中无此关键字。

1.7 第 7 题

Python 3.x 语言中，以下表达式输出结果为 66 的选项是（）

A. print(”6+6”)

B. print(6+6)

C. print(eval(”6+6”))

D. print(eval(”6” + ”6”))

【笔记】
正确答案：D

- A 选项打印的是字符串”6+6”；
- B 选项打印的是 6+6 的值 12；
- C 选项使用 eval 将两个字符串’6’ 转换成数字 6 后相加，得到 12 并输出；
- D 选项使用 eval 将字符串’66’ 转换成数字 66 并输出。

1.8 第 8 题

关于 eval 函数，以下选项中描述错误的是（）

A. eval 函数的作用是将输入的字符串转为 Python 语句，并执行该语句

B. 如果用户希望输入一个数字，并用程序对这个数字进行计算，可以采用 eval(input()) 组合

C. 执行 eval(”Hello”) 和执行 eval(”’Hello’”) 得到相同的结果

D. 执行 eval(’123’) 输出 123

【笔记】

正确答案：C

eval 函数详解：

1. 基本功能：

- eval() 函数用于执行字符串表达式，并返回表达式的结果
- 语法：eval(expression, globals=None, locals=None)
- expression：要执行的字符串表达式

2. 使用示例：

```
# 数学表达式
eval("2 + 3")       # 返回 5
eval("2 * 3 + 1")   # 返回 7
eval("abs(-5)")     # 返回 5

# 字符串表达式
eval("'Hello'")     # 返回 'Hello'
eval("'3' + '4'")   # 返回 '34'

# 变量表达式
x = 10
eval("x + 5")       # 返回 15

# 列表和字典
eval("[1, 2, 3]")   # 返回 [1, 2, 3]
eval("{'a': 1, 'b': 2}") # 返回 {'a': 1, 'b': 2}
```

3. 错误示例分析：

- eval(”Hello”)：会报错，因为 Hello 被当作变量名，但未定义
- eval(”’Hello’”)：正确，返回字符串’Hello’
- eval(”123”)：返回整数 123
- eval(”’123’”)：返回字符串’123’

4. 常见用法：

```
# 用户输入处理
```

```
user_input = input("请输入表达式: ")
result = eval(user_input) # 注意: 有安全风险

# 配置文件解析
config_str = "{'debug': True, 'port': 8080}"
config = eval(config_str)
```

1.9 第 9 题

下面代码的输出结果是 ()

```
d = {"大海":"蓝色", "天空":"灰色", "大地":"黑色"}
print(d["大地"], d.get("大地","黄色"), d.setdefault('草地','绿色'))
```

A. 黑色黑色 None

B. 黑色黄色绿色

C. 黑色黄色 None

D. 黑色黑色绿色

【笔记】

正确答案：D

- 第一个 d[" 大地"] 在字典 d 中存在键，对应为" 蓝色"；
- 第二个 d.get(" 大地"," 黄色")，其含义为获取键“大地”，如果不存在返回值“黄色”
- 第三个 d.setdefault(' 草地',' 绿色')，key 不存在，将 key 设置为 default 值，并返回 default 这个值。

1.10 第 10 题

以下哪个是 Python 中用于科学计算与可视化的第三方库 ()

A. jieba

B. scipy

C. request

D. random

1.11 第 11 题

以下选项中，不是 Python 对文件的打开模式的是 ()

A. 'r'

B. 'c'

C. 'w'

D. '+'

【笔记】
正确答案：B
Python 文件打开模式说明：

- 'r'：只读模式（默认），文件必须存在
- 'w'：写入模式，会覆盖原文件内容，如果文件不存在则创建
- '+'：可读写模式，必须与其他模式组合使用，如'r+'、'w+'
- 'c'：不是 Python 的文件打开模式，此选项为错误选项

其他常见的文件打开模式：

- 'a'：追加模式，在文件末尾写入，如果文件不存在则创建
- 'x'：独占创建模式，文件必须不存在
- 'b'：二进制模式，与其他模式组合使用，如'rb'、'wb'
- 't'：文本模式（默认），与其他模式组合使用，如'rt'、'wt'

1.12 第 12 题

关于下面代码中的变量 x，以下选项中描述正确的是 ()

```
fo = open(fname, "r")
for x in fo:
    print(x)
fo.close()
```

A. 变量 x 表示文件中的一组字符

B. 变量 x 表示文件中的一行字符

C. 变量 x 表示文件中的一个字符

D. 变量 x 表示文件中的全体字符

【笔记】
正确答案：B
代码分析：

- open(fname, ”r”) 以只读模式打开文件，返回文件对象 fo
- 在 Python 中，当使用 for 循环遍历文件对象时，每次迭代获得的是文件的一行内容
- 变量 x 在每次循环中存储文件的一行字符（包括行尾的换行符）
- print(x) 会输出文件的每一行内容
- fo.close() 关闭文件

知识点：

- 文件对象是可迭代的，for 循环遍历文件对象时逐行读取
- 每行内容包含该行的所有字符以及行尾的换行符
- 这种方式适合处理大文件，因为每次只读取一行到内存中

1.13 第 13 题

以上代码输出结果为 ()

```
for i in "Python":
    print(i, end=", ")
```

A. P*y*t*h*o*n*

B. P,y,t,h,o,n,

C. P y t h o n

D. Python

【笔记】
正确答案：B
代码分析：

- for 循环遍历字符串”Python”，每次取出一个字符赋值给变量 i
- print(i, end=”, ”) 中的 end 参数指定输出结束时不换行，而是输出”, ”
- 因此每个字符后面都会跟着逗号和空格

- 最终输出结果为：P, y, t, h, o, n,

知识点：print 函数的 end 参数用于指定输出结束时的字符，默认为换行符 `n`。

1.14 第 14 题

执行 Python 语句 `nums=set([1,2,2,3,3,3,4])` 和 `print(len(nums))` 的结果是 ()

A. 1

B. 2

C. 4

D. 7

【笔记】
正确答案：C
代码分析：

- 列表 [1,2,2,3,3,3,4] 包含 7 个元素，其中有重复元素
- set() 函数将列表转换为集合，集合具有元素唯一性，会自动去除重复元素
- 转换后的集合 nums 包含元素 {1,2,3,4}
- len(nums) 返回集合中元素的个数，即 4

知识点：

- set 是 Python 的内置数据类型，表示无序且不重复的元素集合
- set 会自动去除重复元素，保持元素的唯一性
- len() 函数返回序列或集合中元素的个数

1.15 第 15 题

如下代码的输出为 ()

```
import re
s = 'a bc abc abbb abbbbbca'
re.findall('ab*', s)
```

A. ['ab', 'ab', 'ab']

B. ['ab', 'abbb', 'abbbb']

C. ['a', 'ab', 'abbb', 'abbbb', 'a']

D. ['a', 'ab', 'abbb', 'abbbb']

【笔记】

正确答案：C

正则表达式详解：

1. 基本概念：

- 正则表达式（Regular Expression）是用于匹配字符串的强大工具
- 'ab*' 表示匹配字母'a' 后跟 0 个或多个字母'b'
- re.findall() 返回字符串中所有匹配模式的子字符串列表

2. 量词详解：

- *：匹配前面字符 0 次或多次（贪婪匹配）
- +：匹配前面字符 1 次或多次
- ?：匹配前面字符 0 次或 1 次
- {n}：恰好匹配 n 次
- {n,}：至少匹配 n 次
- {n,m}：匹配 n 到 m 次

3. 匹配过程分析：

```
字符串：'a bc abc abbb abbbbbca'
模式：'ab*'

匹配过程：
位置0：'a' -> 匹配'a'（0个b）
位置2：'b' -> 不匹配（没有前导'a'）
位置3：'c' -> 不匹配
位置5：'a' -> 开始匹配
位置6：'b' -> 匹配'ab'（1个b）
位置7：'c' -> 结束匹配
位置9：'a' -> 开始匹配
位置10-12：'bbb' -> 匹配'abbb'（3个b）
位置14：'a' -> 开始匹配
位置15-18：'bbbb' -> 匹配'abbbb'（4个b）
位置19：'b' -> 继续匹配但遇到'c'
位置20：'c' -> 结束匹配
```

```
位置21: 'a' -> 匹配'a'（0个b）

结果: ['a', 'ab', 'abbb', 'abbbb', 'a']
```

4. 正则表达式语法详解：

4.1 基本字符类：

```
.     # 匹配任意字符（除换行符）
\d    # 匹配数字[0-9]
\D    # 匹配非数字[^0-9]
\w    # 匹配字母、数字、下划线[a-zA-Z0-9_]
\W    # 匹配非字母数字下划线[^a-zA-Z0-9_]
\s    # 匹配空白字符（空格、制表符、换行符等）
\S    # 匹配非空白字符
```

4.2 自定义字符类：

```
[abc]    # 匹配a、b或c中的任意一个
[a-z]    # 匹配任意小写字母
[A-Z]    # 匹配任意大写字母
[0-9]    # 匹配任意数字（等同于\d）
[a-zA-Z] # 匹配任意字母
[^abc]   # 匹配除a、b、c外的任意字符（取反）
[a-z0-9] # 匹配小写字母或数字
```

4.3 量词（重复次数）：

```
*        # 匹配前面的字符0次或多次
+        # 匹配前面的字符1次或多次
?        # 匹配前面的字符0次或1次
{n}      # 匹配前面的字符恰好n次
{n,}     # 匹配前面的字符至少n次
{n,m}    # 匹配前面的字符n到m次
*?       # 非贪婪匹配0次或多次
+?       # 非贪婪匹配1次或多次
??       # 非贪婪匹配0次或1次
```

4.4 锚点（位置匹配）：

```
^        # 匹配字符串开头
$        # 匹配字符串结尾
\b       # 匹配单词边界
\B       # 匹配非单词边界
\A       # 匹配字符串绝对开头
\Z       # 匹配字符串绝对结尾
```

4.5 分组和选择：

```
()       # 分组，创建捕获组
(?:...)  # 非捕获组
|        # 或者（选择）
\1, \2   # 反向引用第1、2个捕获组
(?P<name>...) # 命名捕获组
(?P=name)    # 引用命名捕获组
```

4.6 特殊字符转义：

```
\.       # 匹配字面意思的点号
\*       # 匹配字面意思的星号
\+       # 匹配字面意思的加号
\?       # 匹配字面意思的问号
\[       # 匹配字面意思的左方括号
\]       # 匹配字面意思的右方括号
\\       # 匹配字面意思的反斜杠
\^       # 匹配字面意思的插入符
\$       # 匹配字面意思的美元符号
```

4.7 实用示例：

```
# 匹配邮箱
r'\w+@\w+\.\w+'

# 匹配手机号（简单版）
r'1[3-9]\d{9}'

# 匹配IP地址
r'\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}'

# 匹配HTML标签
r'<[^>]+>'

# 匹配中文字符
r'[\u4e00-\u9fa5]+'

# 匹配URL
r'https?://[^\s]+'
```

5. re 模块常用函数：

```python
import re

text = 'Python 3.8 is great!'

# 查找所有匹配
re.findall(r'\d+', text)    # ['3', '8']

# 搜索第一个匹配
match = re.search(r'\d+', text) # 返回Match对象或None

# 从字符串开头匹配
match = re.match(r'Python', text) # 匹配成功

# 替换
result = re.sub(r'\d+', 'X', text) # 'Python X.X is great!'

# 分割
parts = re.split(r'\s+', text) # ['Python', '3.8', 'is', 'great!']
```

6. 贪婪匹配 vs 非贪婪匹配：

```python
text = 'abbbbbb'

# 贪婪匹配（默认）
re.findall('ab*', text)  # ['abbbbbb'] - 匹配尽可能多的b

# 非贪婪匹配
re.findall('ab*?', text) # ['a'] - 匹配尽可能少的b
```

知识点总结：

- 理解量词的含义和用法
- 掌握正则表达式的匹配原理
- 熟悉 re 模块的常用函数
- 区分贪婪匹配和非贪婪匹配

1.16 第 16 题

以下不能创建一个元组的语句是 ()

A. tup1 = ()

B. tup2 = 1,

C. tup3 = (1)

D. dict4 = tuple(”123”)

【笔记】
正确答案：C
各选项分析：

- A 选项：tup1 = () 创建一个空元组，正确
- B 选项：tup2 = 1, 创建包含一个元素的元组，逗号是关键，正确
- C 选项：tup3 = (1) 仅仅是给数字 1 加括号，结果是整数 1，不是元组
- D 选项：tuple(”123”) 将字符串转换为元组 (’1’, ’2’, ’3’)，正确

知识点：

- 创建元组时，逗号是关键标识符，不是括号
- 单个元素的元组必须在元素后加逗号：(1,) 或 1,
- 空元组可以用 () 或 tuple() 创建
- tuple() 函数可以将其他可迭代对象转换为元组

1.17 第 17 题

在 PyCharm 中运行整个程序的默认快捷键是 ()

A. Shift+F10

B. Ctrl+F10

C. Shift+Ctrl+F10

D. Shift+Ctrl+Enter

【笔记】
正确答案：A
PyCharm 常用快捷键：

- Shift+F10：运行整个程序（Run）
- Ctrl+F10：运行当前文件
- Shift+Ctrl+F10：运行当前光标所在的配置
- Shift+Ctrl+Enter：完成当前语句

其他常用快捷键：

- Shift+F9：调试程序（Debug）
- Ctrl+Shift+F9：调试当前文件
- F8：单步执行（调试时）
- F9：继续执行（调试时）

1.18 第 18 题

下面的语句中 () 用来把路径 path 设置为默认路径

A. os.chdir(path)

B. os.mkdir(path)

C. os.isdir(path)

D. os.listdir(path)

【笔记】
正确答案：A
os 模块常用函数说明：

- os.chdir(path)：改变当前工作目录到指定路径，相当于 cd 命令
- os.mkdir(path)：创建一个新目录
- os.isdir(path)：判断路径是否为目录，返回 True 或 False
- os.listdir(path)：返回指定目录下的文件和目录列表

其他相关函数：

- os.getcwd()：获取当前工作目录
- os.makedirs(path)：递归创建目录
- os.path.exists(path)：判断路径是否存在
- os.path.join()：连接路径

1.19 第 19 题

在 Python 中，下列说法正确的是 ()

A. 0xad 是合法的十六进制数字表示形式

B. 3+4j 不是合法的 Python 表达式

C. 可以使用 if 作为变量名

D. 0o12f 是合法的八进制数字

【笔记】
正确答案：A
各选项分析：

- A 选项：0xad 是合法的十六进制数字，0x 前缀表示十六进制，ad 是有效的 hex 数字
- B 选项：3+4j 是合法的复数表达式，j 表示虚数单位
- C 选项：if 是 Python 保留字（关键字），不能用作变量名
- D 选项：0o12f 不合法，八进制数字只能包含 0-7，f 不是有效的八进制数字

Python 数字表示形式：

- 十进制：直接写数字，如 123
- 二进制：0b 前缀，如 0b1010
- 八进制：0o 前缀，如 0o777（只能包含 0-7）
- 十六进制：0x 前缀，如 0xff（可包含 0-9 和 a-f）
- 复数：实部 + 虚部 j，如 3+4j

2 判断题

2.1 第 1 题

定义函数时，即便该函数不需要接收任何参数，也必须保留一对空的圆括号来表示这是一个函数。（ ）

【笔记】
正确答案：正确 (T)
解释：

- Python 函数定义语法：`def function_name():`
- 即使函数不需要参数，圆括号 `()` 也是必须的
- 圆括号是函数定义的语法要求，用于区分函数和变量
- 示例：`def hello(): print("Hello")`

2.2 第 2 题

已知 `ls=[2, 4, 6]`，那么执行语句 `ls.append(8)` 之后，ls 的内存地址不变。（ ）

【笔记】
正确答案：正确 (T)
解释：

- `append()` 方法是原地修改操作（in-place operation）
- 列表对象本身不会被重新创建，只是在原有内存空间中添加新元素
- 可以用 `id()` 函数验证：`id(ls)` 在 append 前后相同
- 类似的原地修改方法还有：`extend()`, `insert()`, `remove()` 等

对比：重新赋值操作如 `ls = ls + [8]` 会创建新的列表对象，内存地址会改变。

2.3 第 3 题

continue 语句在一旦在循环结构里被执行，将使得当前循环的整个循环提前结束。（ ）

【笔记】
正确答案：错误 (F)
解释：

- `continue` 语句只是跳过当前循环迭代，继续下一次迭代
- `continue` 不会终止整个循环，循环仍会继续执行
- `break` 语句才会提前终止整个循环

示例对比：

- 使用 `continue`：跳过满足条件的迭代，继续其他迭代
- 使用 `break`：一旦满足条件就立即退出整个循环

2.4 第 4 题

`a,b=b,a` 可以实现 a 和 b 值的交换。（ ）

【笔记】
正确答案：正确 (T)
解释：

- Python 支持多重赋值（multiple assignment）

- 右侧 `b,a` 先被打包成元组 `(b,a)`
- 然后元组被解包赋值给左侧的 `a,b`
- 这种方式简洁高效，无需临时变量

传统交换方式对比：

- 传统方式：`temp=a; a=b; b=temp`（需要临时变量）
- Python 方式：`a,b=b,a`（一行代码完成）

2.5 第 5 题

函数是封装了一些独立的功能，可以直接调用，Python 内置了许多函数，同时可以自建函数来使用。（ ）

【笔记】
正确答案：正确 (T)
解释：

- 函数是代码重用和模块化的基础
- 函数封装特定功能，提高代码可读性和可维护性
- Python 内置函数如：`print()`, `len()`, `max()`, `min()` 等
- 用户可以使用 `def` 关键字自定义函数

函数的优点：

- 代码重用：避免重复编写相同代码
- 模块化：将复杂问题分解为小问题
- 易于测试和调试

2.6 第 6 题

使用内置函数 `open()` 且以'w' 模式打开的文件，文件指针默认指向文件尾。（ ）

【笔记】
正确答案：错误 (F)
解释：

- 'w' 模式（写入模式）文件指针指向文件开头
- 'w' 模式会清空原文件内容，从头开始写入

- 'a' 模式（追加模式）文件指针才指向文件尾

文件打开模式对比：

- 'r'：只读模式，指针在文件开头
- 'w'：写入模式，清空文件，指针在文件开头
- 'a'：追加模式，指针在文件末尾
- 'r+'：读写模式，指针在文件开头

2.7 第 7 题

利用 `print()` 格式化输出，`{2:f}`能够控制浮点数的小数点后保留两位。（ ）

【笔记】

正确答案：错误 (F)

解释：

- `{2:f}`不能控制小数点后保留两位
- 2 表示参数索引（第 3 个参数，从 0 开始计数）
- f 表示浮点数格式，但没有指定精度
- 要保留两位小数需要使用`{2:.2f}`

正确的格式化语法：

- `{2:.2f}`：第 3 个参数保留 2 位小数
- `{:.2f}`：当前位置参数保留 2 位小数
- `{2:f}`：第 3 个参数浮点数格式，但精度由系统默认决定

Python print() 格式化输出详解：

1. 基本格式化语法

- 基本形式：`print("格式字符串".format(参数))`
- f-string 形式：`print(f"格式字符串{变量}")`（Python 3.6+）
- % 格式化：`print("格式字符串" % 参数)`（较老的方式）

2. 格式说明符详解

- `{索引:格式说明符}`
- 索引：0, 1, 2... 或省略（按顺序）

- 格式说明符：[填充字符][对齐][宽度][. 精度][类型]

3. 数字格式化类型

- `d`：十进制整数
- `f`：浮点数（默认 6 位小数）
- `.nf`：浮点数保留 n 位小数
- `e`：科学计数法
- `g`：自动选择 f 或 e 格式
- `o`：八进制，`x`：十六进制

4. 字符串格式化

- `s`：字符串格式
- `{:10s}`：字符串宽度 10，左对齐
- `{:>10s}`：右对齐，`{:^10s}`：居中对齐
- `{:*^10s}`：用 * 填充的居中对齐

5. 格式化示例

- `print("{:.2f}".format(3.14159))` → 3.14
- `print("{:10.2f}".format(3.14))` → 3.14
- `print("{:0>5d}".format(42))` → 00042
- `print(f"{name:^10s}")` → 变量 name 居中显示

2.8 第 8 题

定义类时实现了 `__pow__()` 方法，该类对象即可支持运算符 **。(　)

【笔记】
正确答案：正确 (T)
解释：

- `__pow__()` 是 Python 的魔法方法（magic method）
- 实现该方法后，类对象可以使用 ** 运算符进行幂运算
- 调用 `obj ** n` 等价于调用 `obj.__pow__(n)`

其他常用魔法方法：

- `__add__()`：支持 + 运算符
- `__sub__()`：支持-运算符
- `__mul__()`：支持 * 运算符
- `__str__()`：支持字符串转换

2.9 第 9 题

通过对象和类名都可以调用类方法和静态方法。（ ）

【笔记】

正确答案：正确 (T)

解释：

- 类方法（`@classmethod`）：可通过类名和对象调用
- 静态方法（`@staticmethod`）：可通过类名和对象调用
- 这两种方法不依赖于特定的实例状态

方法调用方式对比：

- 实例方法：只能通过对象调用，需要 `self` 参数
- 类方法：类名. 方法 () 或对象. 方法 ()，接收 `cls` 参数
- 静态方法：类名. 方法 () 或对象. 方法 ()，无特殊参数要求

推荐做法：类方法和静态方法优先使用类名调用，语义更清晰。

类方法与静态方法详细区别：

1. 定义方式

- 类方法：使用 `@classmethod` 装饰器
- 静态方法：使用 `@staticmethod` 装饰器

2. 参数区别

- 类方法：第一个参数必须是 `cls`（代表类本身）
- 静态方法：没有特殊的第一个参数要求
- 实例方法：第一个参数必须是 `self`（代表实例本身）

3. 访问权限

- 类方法：可以访问类属性，不能直接访问实例属性

- 静态方法：不能访问类属性和实例属性
- 实例方法：可以访问类属性和实例属性

4. 使用场景

- 类方法：替代构造函数、操作类属性、工厂方法
- 静态方法：与类相关但不需要访问类/实例数据的工具函数
- 实例方法：操作具体实例的数据和行为

5. 代码示例

```
class MyClass:
    class_var = "类属性"

    def __init__(self, value):
        self.instance_var = value

    @classmethod
    def class_method(cls):
        return f"访问: {cls.class_var}"

    @staticmethod
    def static_method():
        return "静态方法"

    def instance_method(self):
        return f"实例: {self.instance_var}"

# 调用方式
MyClass.class_method() # 类方法
MyClass.static_method() # 静态方法
obj = MyClass("值")
obj.class_method()     # 对象调用类方法
obj.static_method()    # 对象调用静态方法
```

3 填空题

3.1 第 1 题

有如下 Python 程序：

```
def f(a, b):
    if b == 0:
```

```
        print(a)
    else:
        f(b, a % b)
```

则 `print(f(9,6))` 的输出结果是_____、_____

【笔记】

正确答案：3, None

解释：

这是一个经典的辗转相除法求最大公约数（GCD）的递归代码。当 b 为 0 时，a 就等于 gcd(a,b) 的值。由于没有返回值，所以 print(f(a,b)) 输出 None。

3.2 第 2 题

表达式 `sorted(['abc', 'acd', 'ade'], key=lambda x:(x[0],x[2]))` 的值为

_____

【笔记】

正确答案：['abc', 'acd', 'ade']

解释：

sorted 函数使用 key 参数来指定排序的依据。这里的 key 是 lambda x:(x[0],x[2])，意味着对每个字符串取第 0 个字符和第 2 个字符组成元组作为排序键。

分析每个字符串的排序键：

- 'abc': (x[0], x[2]) = ('a', 'c')
- 'acd': (x[0], x[2]) = ('a', 'd')
- 'ade': (x[0], x[2]) = ('a', 'e')

由于第 0 个字符都是'a'，按第 2 个字符排序：'c' < 'd' < 'e'，所以原顺序已经是正确的排序结果。

3.3 第 3 题

表达式`'The first:{1:.3f}, the second is {0:b}'.format(2,3.1415926)`的值为

_____

【笔记】

正确答案：'The first:3.142, the second is 10'

解释：

这是字符串的 format 方法格式化，format(2, 3.1415926) 传入两个参数：

- 参数 0：2

- 参数 1：3.1415926

格式化说明：

- {1:.3f}：取第 1 个参数 3.1415926，格式化为保留 3 位小数的浮点数 → 3.142
- {0:b}：取第 0 个参数 2，格式化为二进制 → 10

因此最终结果为：'The first:3.142, the second is 10'

详细字符串格式化解释在2.7

3.4 第 4 题

假设正则表达式模块 re 已导入，那么表达式 `re.sub('\d+', '1', 'a12345bbbbb67c890d0e')` 的值为

【笔记】

正确答案：'a1bbbbb1c1d1e'

解释：

re.sub(pattern, replacement, string) 函数用于字符串替换：

- pattern: '\d+' 即 \d+，匹配一个或多个连续数字
- replacement: '1'，替换成的字符串
- string: 'a12345bbbbb67c890d0e'，要处理的原字符串

匹配过程：

- 原字符串：'a12345bbbbb67c890d0e'
- \d+ 匹配到的数字序列：12345, 67, 890, 0
- 每个数字序列都被替换为'1'

替换结果：'a1bbbbb1c1d1e'

re 模块的常见函数及用法

- **re.match(pattern, string[, flags])**：从字符串开头开始匹配
  - 只匹配字符串开头，如果开头不符合则返回 None
  - 示例：`re.match(r'\d+', '123abc')` → 匹配到'123'
  - 示例：`re.match(r'\d+', 'abc123')` → 返回 None
- **re.search(pattern, string[, flags])**：在整个字符串中搜索第一个匹配
  - 扫描整个字符串，返回第一个匹配的结果

  - 示例：`re.search(r'\d+', 'abc123def456')` → 匹配到'123'

- **re.findall(pattern, string[, flags])**：查找所有匹配项
  - 返回一个列表，包含所有匹配的字符串
  - 示例：`re.findall(r'\d+', 'a12b34c56')` → ['12', '34', '56']

- **re.finditer(pattern, string[, flags])**：返回匹配对象的迭代器
  - 返回一个迭代器，每个元素是一个 Match 对象
  - 适用于大文本，节省内存
  - 示例：`for m in re.finditer(r'\d+', 'a12b34'): print(m.group())`

- **re.sub(pattern, repl, string[, count=0, flags=0])**：替换匹配项
  - 将匹配的部分替换为指定字符串
  - count 参数指定最多替换次数，0 表示全部替换
  - 示例：`re.sub(r'\d+', 'X', 'a12b34')` → 'aXbX'

- **re.split(pattern, string[, maxsplit=0, flags=0])**：按模式分割字符串
  - 使用正则表达式作为分隔符分割字符串
  - 示例：`re.split(r'[,;]', 'a,b;c,d')` → ['a', 'b', 'c', 'd']

- **re.compile(pattern[, flags])**：编译正则表达式
  - 将正则表达式编译成 Pattern 对象，提高重复使用的效率
  - 示例：`p = re.compile(r'\d+'); p.findall('a12b34')` → ['12', '34']

常用的 flags 参数：

- `re.I` 或 `re.IGNORECASE`：忽略大小写
- `re.M` 或 `re.MULTILINE`：多行模式，^ 和 $ 匹配每行的开头和结尾
- `re.S` 或 `re.DOTALL`：使. 匹配包括换行符在内的所有字符
- `re.X` 或 `re.VERBOSE`：忽略空白和注释，可以写更清晰的正则表达式

3.5 第 5 题

下面代码的功能是，随机生成 20 个介于 [1,50] 之间的整数，然后统计每个整数出现频率。请把缺少的代码补全。

```
import random
x = [random.______(1, 50) for i in range(20)]
d = dict()
```

```
for i in x:
    d[i] = d.get(i, ______) + 1
for k, v in d.items():
    print(k, v)
```

上述横线处分别填写_____、_____

【笔记】

正确答案：randint、0

解释：

第一个空：random.randint(1, 50)

- randint(a, b) 函数生成 [a, b] 范围内的随机整数（包含端点）
- 这里需要生成 [1, 50] 之间的整数，所以用 randint

第二个空：d.get(i, 0)

- dict.get(key, default) 方法：如果 key 存在返回对应值，否则返回 default
- 统计频率时，如果数字第一次出现，频率应该是 0+1=1
- 如果数字已经存在，频率就是原来的值 +1

代码逻辑：生成 20 个随机数 → 遍历每个数 → 统计每个数的出现次数 → 输出结果

4 简答题

4.1 第 1 题

在 Python 的运算符中，| 属于何种类型运算符？在集合 set 中 | 有什么作用？并分析逻辑运算符"or" 的短路求值特性。

【笔记】

答案：

1. | 运算符的类型： | 属于**位运算符**（按位或运算符）。它对整数的二进制位进行按位或运算。

2. 在集合 set 中 | 的作用： 在集合中，| 表示**并集运算**。例如：

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1 | set2 # 结果：{1, 2, 3, 4, 5}
```

3. or 的短路求值特性： **短路求值**是指当第一个操作数能够确定整个表达式的结果时，就不再计算第二个操作数。

对于 or 运算符：

- 如果第一个操作数为 True，则不评估第二个操作数，直接返回第一个操作数的值
- 如果第一个操作数为 False，则返回第二个操作数的值

示例：

```
def func():
    print("函数被调用了")
    return True

result = True or func() # func()不会被调用，不会打印
```

4.2 第 2 题

zip() 函数可以将两个或多个可迭代类型 (元组、列表等) 组合为一个关系对 (元组)，并返回包含这些元素的 zip 对象，该函数非常适合生成键值对。
请根据以下示例数据：

- [(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D')]
- {1: 'A', 2: 'B', 3: 'C', 4: 'D'}(1 分)
- ['A', 'B', 'C', 'D']

4.3 第 3 题

请列举 pip 常用的 5 个命令及其作用。

5 代码阅读题

5.1 第 1 题

阅读下面的代码，并分析假设文件"E:\writeTest.txt" 不存在的情况下两段代码可能发生的问题。
代码 1：

```
fp = open(r'E:\writeTest.txt')
fp.write("python")
fp.close()
```

代码 2：

```
fp = open(r'E:\writeTest.txt', 'a+')
fp.write("python")
fp.close()
```

【笔记】

问题分析：

代码 1 存在的问题：

- `open(r'E:writeTest.txt')` 没有指定打开模式，默认为'r'（只读模式）
- 如果文件不存在，会抛出 `FileNotFoundError` 异常
- 即使文件存在，只读模式下调用 `write()` 方法也会抛出 `io.UnsupportedOperation` 异常
- 这段代码无法正常执行

代码 2 的情况：

- 使用'a+' 模式（追加 + 读写模式）
- 如果文件不存在，会自动创建该文件
- 可以正常写入内容"python"
- 这段代码可以正常执行

文件打开模式详细说明：

基本模式：

- **'r'**：只读模式（默认模式）
  - 文件不存在时抛出 FileNotFoundError
  - 只能读取，不能写入
  - 文件指针位于文件开头
- **'w'**：写入模式
  - 文件不存在时自动创建
  - 文件存在时清空内容（覆盖）
  - 只能写入，不能读取
  - 文件指针位于文件开头
- **'a'**：追加模式
  - 文件不存在时自动创建
  - 文件存在时保留原内容
  - 只能写入，不能读取

- 文件指针位于文件末尾

复合模式（可读可写）：

- **'r+'**：读写模式
  - 文件必须存在，否则报错
  - 可读可写，文件指针位于开头
  - 写入会覆盖对应位置的内容
- **'w+'**：写读模式
  - 文件不存在时创建，存在时清空
  - 可读可写，文件指针位于开头
- **'a+'**：追加读写模式
  - 文件不存在时创建，存在时保留内容
  - 可读可写，写入时指针自动移到末尾
  - 读取时可以移动指针到任意位置

二进制模式（在上述模式后加'b'）：

- 'rb'、'wb'、'ab'、'r+b'、'w+b'、'a+b'
- 用于处理二进制文件（图片、视频、exe 等）
- 返回 bytes 对象而不是字符串

5.2 第 2 题

代码分析

(1) 阅读下面代码，解释其功能并写出执行结果。

```
def demo(*p):
    return sum(p)
print(demo(1, 2, 3, 4, 5))
print(demo(1, 2, 3))
```

(2) 阅读下面代码，解释其功能并写出执行结果。

```
def Join(ls, sep=None):
    return (sep or ',').join(ls)
print(Join(['a', 'b', 'c']))
print(Join(['a', 'b', 'c'], ':'))
```

【笔记】

(1) 代码分析：

功能：定义一个可变参数函数，计算所有传入参数的和。

代码解释：

- `*p` 表示可变参数，接收任意数量的位置参数，组成元组
- `sum(p)` 计算元组中所有数字的和
- 函数返回计算结果

执行结果：

```
15
6
```

(2) 代码分析：

功能：定义一个字符串连接函数，使用指定分隔符或默认逗号连接列表元素。

代码解释：

- `sep=None` 设置默认参数为 None
- `(sep or ',')` 利用短路求值：如果 sep 为 None 或 False，则使用','
- `join(ls)` 用分隔符连接列表中的字符串元素

执行结果：

```
a,b,c
a:b:c
```

知识点总结：

- 可变参数 (`*args`) 的使用
- 默认参数和短路求值的应用
- 字符串 `join()` 方法的使用

5.3 第 3 题

给出如下代码：

```
from random import randint
result = list()
while True:
    result.append(randint(1, 10))
```

```
    if len(result) == 20:
        break
print(result)
```

以上代码中，result 为何种类型变量？程序是否能够正常执行，若不能，请解释原因；若能，请分析其执行结果。

【笔记】

代码分析：

1. result 的类型：

- `result = list()` 创建了一个空列表
- 因此 result 是**列表 (list) 类型**变量

2. 程序执行情况：

- **程序可以正常执行**
- 代码语法正确，逻辑清晰

3. 程序执行过程：

- 创建空列表 result
- 进入无限循环 `while True`
- 每次循环生成一个 [1,10] 范围的随机整数并添加到列表
- 当列表长度达到 20 时，执行 `break` 跳出循环
- 打印包含 20 个随机数的列表

4. 执行结果：

- 输出：包含 20 个随机整数的列表
- 每个整数都在 [1,10] 范围内（包含 1 和 10）
- 示例输出：[3, 7, 1, 9, 2, 5, 8, 4, 10, 6, 1, 3, 9, 7, 2, 8, 5, 4, 10, 6]
- 注意：由于是随机数，每次运行结果都不同

知识点总结：

- 列表的创建方法：`list()` 和 `[]`

- append() 方法向列表末尾添加元素
- len() 函数获取列表长度
- 无限循环 while True 配合 break 的使用

5.4 第 4 题

根据如下代码回答问题：

```
class MyArray:
    def __init__(self, *args):
        if not args:
            self.__value = []
        else:
            for arg in args:
                self.__value = list(args)

    def __sub__(self, n):
        b = MyArray()
        b.__value = [item - n for item in self.__value]
        return b

    def __str__(self):
        return str(self.__value)

# 测试主程序
if __name__ == '__main__':
    m = MyArray(10, 3, 2, 5)
```

请回答以下问题：

(1) __init__ 中参数 *args 具有什么含义？

(2) 请说明 __sub__ 方法的功能。并在测试主程序中调用该方法，使其输出为 [6, -1, -2, 1]，写出调用代码。

【笔记】

代码分析：

(1) *args 的含义：

- *args 是可变参数，表示接收任意数量的位置参数
- 在函数内部，args 是一个元组，包含所有传入的参数
- 允许创建 MyArray 对象时传入 0 个或多个参数
- 例如：MyArray()、MyArray(1)、MyArray(1,2,3) 都是合法的

构造函数逻辑：

- 如果没有传入参数 (`not args`)，创建空列表
- 如果传入参数，将 args 转换为列表赋值给 `self.__value`
- 注意：代码中的 for 循环是多余的，每次都会重新赋值

(2) ___sub___ 方法功能：

- `__sub__` 是 Python 的魔术方法，重载减法运算符'-'
- 功能：将数组中每个元素都减去指定的数 n
- 返回一个新的 MyArray 对象，不修改原对象
- 实现：`[item - n for item in self.__value]` 列表推导式

调用代码分析：

- 当前：`m = MyArray(10, 3, 2, 5)`，即 m.___value = [10, 3, 2, 5]
- 要得到 [6, -1, -2, 1]，需要每个元素减去 4：
  - 10 - 4 = 6
  - 3 - 4 = -1
  - 2 - 4 = -2
  - 5 - 4 = 1

调用代码：

```
result = m - 4
print(result) # 输出：[6, -1, -2, 1]
```

知识点总结：

- 可变参数 `*args` 的使用
- Python 魔术方法 `__sub__` 重载运算符
- 列表推导式的应用
- 私有属性命名约定（双下划线开头）

6 代码编写题

6.1 第 1 题

编程实现，由用户输入一个不多于 5 位的正整数，要求求出它是几位数以及逆序打印出该数。

【笔记】

```
n = int(input())
s = str(n) # 数字转换成字符串
print(len(s)) # 输出长度
print(s[::-1]) # 输出反转后的字符串
```

6.2 第 2 题

循环从用户处获得一组数据，直到用户直接输入回车退出，打印输出所有数据的和。本题不考虑输入异常情况。

【笔记】

```
n = input()
tot = 0

while n: # 空字符串为False
    tot += eval(n) # 转换为int类型并累加
    n = input() # 再输入

print(tot)
```

6.3 第 3 题

假设 os 模块已导入，那么请写出 C:\Windows 文件夹中（无需遍历该文件夹下的子文件夹）所有扩展名为.exe 的文件

【笔记】

方法一：使用 os.listdir() 和字符串方法

```
import os

path = r'C:\Windows'
files = os.listdir(path)

for file in files:
```

```
    if file.endswith('.exe'):
        print(file)
```

方法二：使用列表推导式

```
import os

path = r'C:\Windows'
exe_files = [f for f in os.listdir(path) if f.endswith('.exe')]
for file in exe_files:
    print(file)
```

方法三：使用 os.path.splitext()

```
import os

path = r'C:\Windows'
for file in os.listdir(path):
    name, ext = os.path.splitext(file)
    if ext.lower() == '.exe':
        print(file)
```

代码解释：

- `r'C:\Windows'`：使用原始字符串避免转义问题
- `os.listdir(path)`：列出指定目录下的所有文件和文件夹
- `file.endswith('.exe')`：判断文件名是否以.exe 结尾
- `os.path.splitext()`：分离文件名和扩展名
- `ext.lower()`：转换为小写，增加匹配的容错性

注意事项：

- 此代码只列出当前目录的文件，不会递归遍历子目录
- 需要有访问 C:\Windows 目录的权限
- 实际使用时可能需要异常处理

os 模块的常用函数介绍

1. 目录操作函数：

- **os.getcwd()**：获取当前工作目录

```
import os
current_dir = os.getcwd()
print(current_dir) # 输出当前工作目录的绝对路径
```

- **os.chdir(path)**：改变当前工作目录

```
os.chdir('/home/user/documents') # Linux/Mac
os.chdir(r'C:\Users\Documents') # Windows
```

- **os.listdir(path)**：列出指定目录下的文件和子目录

```
files = os.listdir('.') # 列出当前目录
files = os.listdir('/tmp') # 列出/tmp目录
```

- **os.mkdir(path)**：创建单级目录

```
os.mkdir('new_folder') # 在当前目录创建new_folder
os.mkdir('/tmp/test') # 在/tmp下创建test目录
```

- **os.makedirs(path)**：递归创建多级目录

```
os.makedirs('dir1/dir2/dir3') # 创建多级目录
os.makedirs('dir1/dir2', exist_ok=True) # 如果存在则不报错
```

- **os.rmdir(path)**：删除空目录

```
os.rmdir('empty_folder') # 只能删除空目录
```

- **os.removedirs(path)**：递归删除空目录

```
os.removedirs('dir1/dir2/dir3') # 从内到外删除空目录
```

2. 文件操作函数：

- **os.remove(path)**：删除文件

```
os.remove('test.txt') # 删除test.txt文件
```

- **os.rename(src, dst)**：重命名文件或目录

```
os.rename('old_name.txt', 'new_name.txt') # 重命名文件
os.rename('old_dir', 'new_dir') # 重命名目录
```

- **os.stat(path)**：获取文件或目录的状态信息

```
stat_info = os.stat('file.txt')
print(stat_info.st_size) # 文件大小（字节）
print(stat_info.st_mtime) # 最后修改时间
```

3. os.path 模块（路径操作）：

- **os.path.exists(path)**：判断路径是否存在

```
if os.path.exists('file.txt'):
    print('文件存在')
```

- **os.path.isfile(path)**：判断是否为文件

```
if os.path.isfile('test.txt'):
    print('这是一个文件')
```

- **os.path.isdir(path)**：判断是否为目录

```
if os.path.isdir('/home'):
    print('这是一个目录')
```

- **os.path.join(path1, path2, ...)**：连接路径

```
# 跨平台的路径连接
path = os.path.join('home', 'user', 'file.txt')
# Windows: home\user\file.txt
# Linux/Mac: home/user/file.txt
```

- **os.path.split(path)**：分割路径和文件名

```
dir_name, file_name = os.path.split('/home/user/file.txt')
# dir_name = '/home/user'
# file_name = 'file.txt'
```

- **os.path.splitext(path)**：分离文件名和扩展名

```
name, ext = os.path.splitext('document.pdf')
# name = 'document'
# ext = '.pdf'
```

- **os.path.basename(path)**：获取文件名

```
file_name = os.path.basename('/home/user/file.txt')
# file_name = 'file.txt'
```

- **os.path.dirname(path)**：获取目录名

```
dir_name = os.path.dirname('/home/user/file.txt')
# dir_name = '/home/user'
```

- **os.path.abspath(path)**：获取绝对路径

```
abs_path = os.path.abspath('file.txt')
# 返回file.txt的绝对路径
    \end{lstlisting>

    \item \textbf{os.path.getsize(path)}：获取文件大小
    \begin{lstlisting}
size = os.path.getsize('file.txt') # 返回字节数
```

4. 环境变量操作：

- **os.environ**：环境变量字典

```
# 获取环境变量
home = os.environ.get('HOME') # Linux/Mac
home = os.environ.get('USERPROFILE') # Windows

# 设置环境变量
os.environ['MY_VAR'] = 'value'
```

- **os.getenv(key, default=None)**：获取环境变量

```
path = os.getenv('PATH', '') # 获取PATH，不存在返回空字符串
```

5. 系统相关：

- **os.name**：操作系统名称

```
print(os.name) # 'posix'(Linux/Mac) 或 'nt'(Windows)
```

- **os.system(command)**：执行系统命令

```
os.system('ls -la') # Linux/Mac
os.system('dir')  # Windows
```

- **os.sep**：路径分隔符

```
print(os.sep) # '/' (Linux/Mac) 或 '\' (Windows)
```

6. 遍历目录树 - os.walk()：

```
import os

# 遍历目录树
for root, dirs, files in os.walk('/path/to/directory'):
    print(f'当前目录: {root}')
    print(f'子目录: {dirs}')
    print(f'文件: {files}')

    # 处理每个文件
    for file in files:
        file_path = os.path.join(root, file)
        print(f'文件路径: {file_path}')
```

7. 实用示例：

示例 1：获取目录下所有特定类型的文件

```
import os

def find_files(directory, extension):
    """查找目录下所有指定扩展名的文件"""
    result = []
    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.endswith(extension):
                result.append(os.path.join(root, file))
    return result

# 查找所有.py文件
python_files = find_files('.', '.py')
```

示例 2：创建目录结构

```
import os

def create_project_structure(project_name):
    """创建项目目录结构"""
    dirs = [
        f'{project_name}/src',
        f'{project_name}/tests',
        f'{project_name}/docs',
        f'{project_name}/data'
    ]

    for dir_path in dirs:
        os.makedirs(dir_path, exist_ok=True)
        print(f'创建目录: {dir_path}')
```

示例 3：批量重命名文件

```
import os

def batch_rename(directory, old_ext, new_ext):
    """批量修改文件扩展名"""
    for filename in os.listdir(directory):
        if filename.endswith(old_ext):
            old_path = os.path.join(directory, filename)
            new_name = filename.replace(old_ext, new_ext)
            new_path = os.path.join(directory, new_name)
            os.rename(old_path, new_path)
            print(f'重命名: {filename} -> {new_name}')
```

注意事项：

- 使用 os.path.join() 构建路径，确保跨平台兼容性
- 在删除文件或目录前，先检查是否存在
- 处理文件操作时要考虑异常处理
- 使用原始字符串 (r") 处理 Windows 路径，避免转义问题
- os.makedirs() 的 exist_ok=True 参数可以避免目录已存在的错误

6.4 第 4 题

在游戏应用中，经常会判断鼠标是否点击了某个人物或道具，本题将模拟此场景，编程实现判断某个点是否在某个矩形内，具体要求如下：

(1) 实现 Point 类，类中有私有属性 ___x，___y，代表鼠标的坐标，在类中实现方法 get_x，set_x，get_y，set_y，使其可以分别访问相应属性。(4 分)
(2) 实现 Rectangle 类，类中有私有属性 x，y，代表矩形的左上角坐标，私有属性 width 和 height，代表矩形的宽度和高度。(4 分)
(3) 在 Rectangle 类中实现方法 contain，判断某个点 (Point 类的对象) 是否包含在此矩形中 (4 分)

测试代码如下：

```
p1 = Point(20, 25) # 点p1的坐标为x=20, y=25
# 新建矩形对象rect，其左上角坐标x=10、y=15，矩形的宽度50、高度60
rect = Rectangle(10, 15, 50, 60)
print(rect.contain(p1))

p1.set_x(80) # 设置点p1的x坐标为80
p1.set_y(90) # 设置点p1的y坐标为90
print(rect.contain(p1))
```

输出结果：

```
True
False
```

【笔记】

完整代码实现：

```
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def set_x(self, x):
        self.__x = x

    def set_y(self, y):
        self.__y = y

    def get_x(self):
        return self.__x

    def get_y(self):
        return self.__y

class Rectangle:
```

```
    def __init__(self, x, y, width, height):
        self.__x = x
        self.__y = y
        self.__width = width
        self.__height = height

    def contain(self, point):
        if self.__x <= point.get_x() <= self.__x + self.__width \
        and self.__y <= point.get_y() <= self.__y + self.__height:
            return True
        return False
```

Python 类详解：

1. 类的基本概念：

- 类（Class）是对象的模板，定义了对象的属性和方法
- 对象（Object）是类的实例，具有类定义的属性和行为
- 使用关键字 `class` 定义类，类名通常采用驼峰命名法

2. 构造函数 ___init___：

- `__init__` 是特殊方法，用于初始化对象
- 创建对象时自动调用，相当于其他语言的构造函数
- `self` 参数代表对象本身，必须是第一个参数
- 可以接收参数来初始化对象的属性

3. 属性访问控制：

```
# 公有属性：可以直接访问
self.x = x        # 公有属性

# 私有属性：不能直接从外部访问
self.__x = x      # 私有属性（双下划线开头）

# 访问示例
p = Point(10, 20)
print(p.x)        # 如果x是公有属性，可以直接访问
# print(p.__x)    # 错误！私有属性不能直接访问
print(p.get_x()) # 通过getter方法访问私有属性
```

4. Getter 和 Setter 方法：

- Getter 方法：用于获取私有属性的值
- Setter 方法：用于设置私有属性的值
- 提供了对私有属性的受控访问
- 可以在方法中添加验证逻辑

5. 方法定义和调用：

```
# 方法定义
def method_name(self, parameters):
    # 方法体
    return value

# 方法调用
obj = ClassName()
result = obj.method_name(arguments)
```

6. 代码逻辑分析：

Point 类：

- 私有属性 `__x` 和 `__y` 存储坐标
- 提供 getter 和 setter 方法访问坐标值
- 封装了坐标点的基本操作

Rectangle 类：

- 私有属性 `__x`、`__y`、`__width`、`__height` 存储矩形的位置和尺寸信息
- `contain` 方法判断点是否在矩形内部
- 判断条件：`__x <= point_x <= __x + __width` 且 `__y <= point_y <= __y + __height`

7. 面向对象编程优势：

- **封装**：将数据和操作封装在类中
- **抽象**：隐藏实现细节，只暴露必要接口
- **重用性**：类可以被多次实例化使用
- **维护性**：便于代码的维护和扩展

8. 类装饰器详解：

8.1 @property 装饰器：

- @property 将方法转换为属性，提供更优雅的访问方式
- 可以像访问属性一样调用方法，无需加括号
- 提供了 getter、setter、deleter 的完整支持
- 比传统的 get/set 方法更加 Pythonic

8.2 使用 @property 重写 Point 类：

```
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    @property
    def x(self):
        """获取x坐标"""
        return self.__x

    @x.setter
    def x(self, value):
        """设置x坐标，可以添加验证"""
        if not isinstance(value, (int, float)):
            raise TypeError("坐标必须是数字")
        self.__x = value

    @x.deleter
    def x(self):
        """删除x坐标"""
        print("删除x坐标")
        del self.__x

    @property
    def y(self):
        """获取y坐标"""
        return self.__y

    @y.setter
    def y(self, value):
        """设置y坐标"""
        if not isinstance(value, (int, float)):
            raise TypeError("坐标必须是数字")
        self.__y = value
```

```
    @property
    def distance_from_origin(self):
        """计算到原点的距离（只读属性）"""
        return (self.__x ** 2 + self.__y ** 2) ** 0.5
```

8.3 @property 的使用示例：

```
# 创建点对象
p = Point(3, 4)

# 使用@property装饰的方法，像属性一样访问
print(p.x)                  # 输出：3（调用getter）
print(p.y)                  # 输出：4（调用getter）
print(p.distance_from_origin) # 输出：5.0（只读属性）

# 使用setter设置值
p.x = 10                    # 调用setter
p.y = 20                    # 调用setter

# 验证功能
try:
    p.x = "invalid"         # 触发TypeError异常
except TypeError as e:
    print(e)                # 输出：坐标必须是数字

# 删除属性
del p.x                     # 调用deleter
```

8.4 传统方法 vs @property 对比：

```
# 传统getter/setter方法
point1 = Point(10, 20)
print(point1.get_x())       # 需要调用方法
point1.set_x(30)            # 需要调用方法

# 使用@property装饰器
point2 = Point(10, 20)
print(point2.x)             # 像访问属性一样
point2.x = 30               # 像设置属性一样
```

8.5 @property 的优势：

- **简洁性**：访问语法更加直观，无需 get/set 前缀
- **封装性**：可以在 getter/setter 中添加验证逻辑

- **兼容性**：可以将普通属性升级为 property 而不破坏接口
- **只读属性**：可以创建计算属性（如 distance_from_origin）
- **延迟计算**：属性值可以在访问时动态计算

8.6 其他常用装饰器：

```
class MyClass:
    @staticmethod
    def static_method():
        """静态方法，不需要self参数，不依赖实例"""
        return "这是静态方法"

    @classmethod
    def class_method(cls):
        """类方法，接收cls参数，可以访问类属性"""
        return f"这是{cls.__name__}的类方法"

    @property
    def read_only_prop(self):
        """只读属性"""
        return "只读值"

# 使用示例
obj = MyClass()
print(MyClass.static_method()) # 直接通过类调用
print(MyClass.class_method()) # 通过类调用类方法
print(obj.read_only_prop)     # 访问只读属性
```

8.7 实际应用场景：

- **数据验证**：在 setter 中验证输入数据的有效性
- **计算属性**：根据其他属性动态计算值
- **缓存机制**：在属性中实现计算结果的缓存
- **兼容性**：将直接属性访问升级为方法调用
- **日志记录**：在属性访问时记录日志

9. 扩展知识：

```
# 类属性和实例属性
class MyClass:
    class_var = "类属性" # 类属性，所有实例共享
```

```
    def __init__(self, value):
        self.instance_var = value # 实例属性，每个实例独有

# 特殊方法（魔术方法）
class Point:
    def __str__(self):
        return f"Point({self.__x}, {self.__y})"

    def __repr__(self):
        return f"Point({self.__x}, {self.__y})"

    def __eq__(self, other):
        return self.__x == other.__x and self.__y == other.__y
```