得分：98
评语：作业完成认真，代码正确

python第2-6章作业

第二章

题目1：

AAB
B

题目2：

A. _boy - 合法。以字母下划线开头，只包含字母和下划线。

B. :4studen - 不合法。以数字开头，违反了变量名必须以字母或下划线开头的规则。

C. :class - 合法。虽然 class 是Python的关键字，但是关键字也可以作为变量名，只是通常不推荐这样做，因为它可能会导致混淆。

D. first number - 不合法。变量名中包含了空格，违反了变量名只能包含字母、数字和下划线的规则。在Python中，变量名不能包含空格。

题目3：

文件的核心内容是关于Python中三种基本控制流语句的执行过程的简述：

1. **for循环**：
    - `for`循环通常用于遍历序列（如列表、元组、字典、集合、字符串）或其他可迭代对象的元素。
    - 在`for`循环开始时，会初始化迭代过程。
    - 循环体内，每次迭代都会自动获取迭代对象的下一个元素，并将其赋值给指定的循环变量。
    - 循环继续执行，直到迭代完所有元素。
    - 循环结束后，循环变量可能会保留其最终值，但循环本身不再执行。
2. **while循环**：
    - `while`循环基于给定的条件执行。
    - 在每次迭代开始前，会检查`while`后面指定的条件表达式的值。
    - 如果条件为真（True），则执行循环体内的代码块。
    - 循环体执行结束后，再次检查条件，如果条件仍然为真，则重复执行循环体。
    - 这个过程会一直持续，直到条件变为假（False），此时循环终止。
3. **if分支语句**：
    - `if`语句用于根据条件执行不同的代码块。
    - 首先评估`if`后面指定的条件表达式。
    - 如果条件为真，则执行`if`代码块内的语句。
    - 如果条件为假，则跳过`if`代码块，继续执行后续的`elif`（如果存在）或`else`代码块（如果存在）。
    - `elif`和`else`代码块是可选的，可以根据需要有零个、一个或多个。

- `if`语句可以嵌套，即一个`if`语句内部可以包含其他的`if`、`elif`或`else`语句。

题目4：

1.

```
    #提示用户输入第一个数字
    num1 = float(input("请输入第一个数字："))

    #提示用户输入第二个数字
    num2 = float(input("请输入第二个数字："))

    #计算两个数字的乘积
    product = num1 * num2

    #输出乘积
    print("两个数字的乘积是：", product)
```

2.

```
# 黄金分割比
phi = 0.6180339887

# 保留前三位的浮点数输出
formatted_phi = format(phi, ".3f")
print("保留前三位的浮点数输出：", formatted_phi)

# 保留一位小数的百分比输出
formatted_phi_percentage = format(phi * 100, ".1f%")
print("保留一位小数的百分比输出：", formatted_phi_percentage)
```

3.

```
for i in [0, 1, 2, 3, 4, 5]:
    # 使用format方法将"X"重复(2*i+1)次，并格式化到字符串中
    print("[{0}]".format("X" * (2 * i + 1)))
```

题目5：

```
x *= 10
y = (3 - 1) / 8
t = (1, 2, 3, 4, 5)
x = 0.10
```

```
if m > 3:
    print("m大于3")
```

第三章

填空题

1.

6400

2.

2.5 2
(2, 3)

3.

3.14 3

操作题

4.

```
s = "python is a good language"

# 使用正向索引获取 "python"
# 首先找到 "python" 的起始和结束位置
start_index = s.index("python")
end_index = start_index + len("python")

# 提取子字符串
python_forward = s[start_index:end_index]
print("使用正向索引获取的 'python':", python_forward)

# 使用负向索引获取 "python"
# 负向索引从字符串末尾开始计数
python_backward = s[-end_index:-start_index]
print("使用负向索引获取的 'python':", python_backward)
```

5.

```
s = "python is a good language"

# 将首字母大写
s_capitalized = s.capitalize()
```

```
print(s_capitalized)
```

6.

```
s = "python is a good language"

# 统计字母 'o' 出现的次数
count_o = s.count('o')

print("字母 'o' 出现了", count_o, "次")
```

7.

```
s_replaced = s.replace("python", "C++")

print(s_replaced)
```

8.

```
s = "python is a good language"

# 使用 split 方法分割字符串，然后使用 join 方法用空格连接
s_spaced = ' '.join(s.split())

print(s_spaced)
```

9.

```
s = "python is a good language"

# 使用 split 方法将字符串分割成单词列表
words = s.split()

# 使用列表推导式将每个单词用 '*' 包围
starred_words = ['*' + word + '*' for word in words]

# 使用 join 方法将处理后的单词列表连接成新的字符串
s_starred = ' '.join(starred_words)

print(s_starred)
```

10.

```
# 十进制数
decimal = 255

# 转化为二进制
binary = bin(decimal)
print(f"255的二进制表示为：{binary}")

# 转化为八进制
octal = oct(decimal)
print(f"255的八进制表示为：{octal}")

# 转化为十六进制
hexadecimal = hex(decimal)
print(f"255的十六进制表示为：{hexadecimal}")

# 将二进制转化回十进制
decimal_from_binary = int(binary[2:], 2)
print(f"二进制{binary}转回十进制为：{decimal_from_binary}")

# 将八进制转化回十进制
decimal_from_octal = int(octal[2:], 8)
print(f"八进制{octal}转回十进制为：{decimal_from_octal}")

# 将十六进制转化回十进制
decimal_from_hexadecimal = int(hexadecimal[2:], 16)
print(f"十六进制{hexadecimal}转回十进制为：{decimal_from_hexadecimal}")
```

第四章

题目一

1.

```
even_numbers = [number for number in range(100) if number % 2 == 0]

print(even_numbers)
```

2.

```
# 定义列表 Is
Is = ["a", "b", "c", "d", "e"]

# 使用正向索引获得元素“c”
c_forward = Is[2]
print("使用正向索引获得的 'c':", c_forward)
```

```
# 使用负向索引获得元素“c”
c_reverse = Is[-3]
print("使用负向索引获得的 'c':", c_reverse)

# 使用正向切片获得[“b”,“d”]
slice_forward = Is[1:4]
print("使用正向切片获得的 ['b', 'd']:", slice_forward)

# 使用负向切片实现 Is 的反向列表
reversed_Is = Is[::-1]
print("使用负向切片反转的列表:", reversed_Is)
```

3.

```
# 定义列表 cars
cars = ["BYD", "GEELY"]

# 在列表结尾增加元素“TOYOTA”
cars.append("TOYOTA")
print("增加'TOYOTA'后:", cars)

# 在“BYD”和“GEELY”之间增加元素“CHERY”
cars.insert(1, "CHERY")
print("增加'CHERY'后:", cars)

# 在列表结尾增加["BMW","BENZ"]中的元素
cars.extend(["BMW", "BENZ"])
print("增加'BMW'和'BENZ'后:", cars)

# 删除列表最后一个元素和第四个元素
cars.pop()  # 删除最后一个元素
cars.pop(3)  # 删除第四个元素（索引从0开始）
print("删除最后一个元素和第四个元素后:", cars)

# 删除元素“GEELY”
cars.remove("GEELY")
print("删除'GEELY'后:", cars)

# 查找元素“BMW”在列表中的位置索引
index_bmw = cars.index("BMW")
print("元素'BMW'的位置索引:", index_bmw)

# 将元素“CHERY”修改为“QQ”
if "CHERY" in cars:
    cars[cars.index("CHERY")] = "QQ"
print("将'CHERY'修改为'QQ'后:", cars)

# 通过两种方法对列表进行复制
cars_copy1 = cars.copy()  # 方法1: 使用 copy() 方法
cars_copy2 = list(cars)   # 方法2: 使用 list() 函数
```

```
print("复制后的列表:", cars_copy1)
print("复制后的列表:", cars_copy2)

# 对列表进行永久排序
cars.sort()
print("永久排序后的列表:", cars)

# 对列表进行临时排序
sorted_cars = sorted(cars)
print("临时排序后的列表:", sorted_cars)

# 对列表进行翻转
cars.reverse()
print("翻转后的列表:", cars)

# 对列表进行遍历，并按格式输出
for i, car in enumerate(cars, start=1):
    print(f"My {i}st car is {car}。")
```

题目二

4.

数据的不可变性：元组一旦被创建，其中的元素就不能被修改、添加或删除。这意味着元组的元素是固定的，这与列表（List）不同，列表中的元素是可以被修改的。

索引方式访问：与列表一样，元组也支持通过索引来访问其元素。但是，由于元组的不可变性，你不能对其中的元素进行赋值操作。

哈希性：由于元组的不可变性，它们可以被用作字典的键，因为只有不可变类型才能作为字典的键。列表由于是可变的，不能直接作为字典的键。

性能：元组的不可变性使得它们在某些情况下比列表有更优的性能表现。因为Python可以优化元组在内存中的存储，知道它们不会被改变。

数据安全：元组的不可变性提供了一定程度的数据安全保障。当你不希望数据被改变时，使用元组可以确保数据的完整性。

5.

```
# 定义食物和价格的列表
foods = ["bread", "fish", "potato"]
prices = [2.4, 9.8, 0.9]

# 使用 zip 函数将两个列表组合，并遍历输出
for food, price in zip(foods, prices):
    print(f"The price of {food} is {price}。")
```

题目三

6.

A. `{(1,2): 3}`
合法。键是一个元组 `(1,2)`，元组是不可变类型，所以可以作为字典的键。
B. `{[1,2]: 3}`
不合法。因为键是一个列表 `[1,2]`，列表是可变类型，不能作为字典的键。
C. `{"price": 2.4}`
合法。键是一个字符串 `"price"`，字符串是不可变类型，可以作为字典的键。
D. {1: "first"}
合法。键是一个整数 1，整数是不可变类型，可以作为字典的键。

7.

```
# 定义字典 favorite_fruits
favorite_fruits = {"Judy": "watermelon", "Jen": "banana", "Sarah": "orange"}

# 获得字典 favorite_fruits 的长度
length = len(favorite_fruits)
print("字典的长度:", length)

# 获得“Jen”最爱的水果名
jen_fruit = favorite_fruits["Jen"]
print("Jen's favorite fruit is", jen_fruit)

# 增加“Tom”最爱的水果“peach”，“Bob”最爱的水果“Tomato”
favorite_fruits["Tom"] = "peach"
favorite_fruits["Bob"] = "Tomato"
print("增加后字典:", favorite_fruits)

# 删除“Judy”及其最爱的水果名
del favorite_fruits["Judy"]
print("删除Judy后字典:", favorite_fruits)

# 随机删除一个键值对，并捕获被删除的值
key_to_delete = list(favorite_fruits.keys())[0]  # 获取一个键（这里取第一个）
deleted_value = favorite_fruits.pop(key_to_delete)  # 删除并获取被删除的值
print(f"随机删除的键值对是: {key_to_delete}: {deleted_value}")
print("随机删除后字典:", favorite_fruits)

# 将“Sarah”的最爱修改为“watermelon”
favorite_fruits["Sarah"] = "watermelon"
print("修改Sarah的最爱后的字典:", favorite_fruits)

# 遍历字典 favorite_fruits，并按下列格式进行输出
for name, fruit in favorite_fruits.items():
    print(f"{name}'s favorite fruit is {fruit}")
```

8.

```
from collections import Counter

tongue_twister = """
八百标兵奔北坡，
北坡八百炮兵炮。
标兵怕碰炮兵炮，
炮兵怕把标兵碰。
"""

# 移除换行符并统计字符频次
char_count = Counter(tongue_twister.replace('\n', ''))

# 打印字符频次
for char, count in char_count.items():
    print(f"字符 '{char}' 出现了 {count} 次")
```

9.

```
# 定义集合 vegetables 和 fruits
vegetables = {"tomato", "cabbage", "cucumber", "meat"}
fruits = {"banana", "orange", "tomato", "cucumber"}

# 将集合 vegetables 中的 “meat” 删除
vegetables.discard("meat")
print("删除 'meat' 后的 vegetables 集合:", vegetables)

# 将 “eggplant” 加入到集合 vegetables 中
vegetables.add("eggplant")
print("加入 'eggplant' 后的 vegetables 集合:", vegetables)

# 两个集合中，即属于蔬菜又属于水果的元素
common_items = vegetables.intersection(fruits)
print("即属于蔬菜又属于水果的元素:", common_items)

# 两个集合中，所有的蔬菜和水果
all_items = vegetables.union(fruits)
print("所有的蔬菜和水果:", all_items)

# 两个集合中，只属于蔬菜或者只属于水果的元素
exclusive_items = (vegetables - fruits).union(fruits - vegetables)
print("只属于蔬菜或者只属于水果的元素:", exclusive_items)

# 两个集合中，只属于蔬菜的元素
only_vegetables = vegetables - fruits
print("只属于蔬菜的元素:", only_vegetables)
```

第五章

1.

```
空的
```

```
True
```

```
True
```

2.

```
ball_color = "blue"  # 可以改为其他颜色，如 "black", "white", "pink", "purple", "colorful"

# 定义一个字典来映射球的颜色和对应的奖金
prizes = {
    "black": 0,
    "white": 10,
    "blue": 20,
    "pink": 30,
    "purple": 50,
    "colorful": 100
}

# 根据球的颜色分配奖金
prize = prizes.get(ball_color, "无效的球颜色")

# 输出结果
if isinstance(prize, int):
    print(f"摸到的是{ball_color}球，获得奖金{prize}元。")
else:
    print(prize)
```

3.

```
for i in range(1, 10):
    for j in range(1, i + 1):
        # 打印乘法表达式，格式化为两位数，中间用乘号连接
        print(f"{j}*{i}={i*j}", end="\t")
    # 每打印完一行后换行
    print()
```

4.

```
# 初始化存储学生信息的字典
students_of_grade3_class2 = {}

# 使用 while 循环动态输入学生信息
while True:
    # 输入学号
    student_id = input("请输入学生学号（输入'Q'或'q'结束程序）：")
    # 检查是否结束程序
    if student_id.lower() == 'q':
        break
    # 输入姓名
    student_name = input("请输入学生姓名：")
    # 输入性别
    student_gender = input("请输入学生性别：")

    # 将学生信息存储在字典中，以学号为键
    students_of_grade3_class2[student_id] = {"姓名": student_name, "性别":
student_gender}

# 遍历并输出学生信息
print("\n高三二班学生信息：")
for student_id, info in students_of_grade3_class2.items():
    print(f"学号：{student_id}，姓名：{info['姓名']}，性别：{info['性别']}")
```

5.

不能。变量 i 在循环中没有被更新。这意味着 i 的值将始终保持为0，导致循环条件永远为真，从而形成一个无限循环。程序将不断打印数字0，而不会正常结束。

6.

嵌套层数过多通常指的是代码中条件语句（如`if`）、循环（如`for`、`while`）或者函数调用的嵌套使用过多。以下是为什么我们通常不希望代码中有过多嵌套的原因：

1. **降低可读性**：嵌套层数越多，代码的阅读和理解难度越大。对于复杂的嵌套结构，新接触代码的人可能需要花费更多时间来理解代码的执行流程。
2. **增加维护难度**：代码维护时，需要考虑到所有嵌套层的影响，这使得修改、调试和扩展代码变得更加困难。
3. **提高错误风险**：每增加一层嵌套，就增加了出错的可能性。在多层嵌套中，一个小错误可能导致整个代码块的行为异常。
4. **影响性能**：在某些情况下，嵌套结构可能影响代码的性能，尤其是在需要优化执行速度的场景中。
5. **增加复杂度**：嵌套使得代码的逻辑更加复杂，这可能导致逻辑错误和难以追踪的bug。
6. **减少代码复用性**：嵌套代码通常难以独立重用，因为它们与特定的上下文紧密耦合。

7.

当Python中条件测试变得过于复杂时，可以采取以下几种策略来处理：

1. **提前结束函数**：在函数内使用`return`或`raise`等语句提前在分支内结束函数，避免不必要的嵌套分支，使代码更直接也更易读。
2. **封装复杂逻辑**：如果条件分支里的表达式过于复杂，出现了太多的`not/and/or`，可以将这些逻辑封装成独立的函数或方法，以提高代码的可读性。
3. **简化条件判断**：通过消除冗余的检查和合并相似的条件来实现。例如，如果多个条件检查相同变量，可以将其重构为一个单一的检查。
4. **利用缓存机制**：在条件判断中，如果存在重复且计算成本高的子表达式，可以使用缓存来避免重复计算。这种策略称为记忆化（memoization），它特别适合递归函数中的重复计算。
5. **使用逻辑运算符**：使用逻辑运算符（如`and`、`or`）可以有效简化和优化`if`语句。合理地组合条件可以减少代码的复杂度，并提高可读性和执行效率。
6. **优化条件顺序**：优化条件顺序是提高`if`语句性能的一个重要方法。通过将最有可能为真的条件放在前面，可以减少代码执行的时间。
7. **使用字典或集合**：使用字典映射来简化条件判断逻辑，可以显著提高代码的可读性和维护性。
8. **提取函数**：当一个函数过于庞大时，可以将其分解为多个小函数。这有助于减少函数的复杂性，并提高代码的可读性和可维护性。
9. **简化复杂逻辑**：提取函数可以帮助简化复杂的逻辑，使代码更易于理解。
10. **消除重复代码**：将重复的代码提取到一个独立的函数中，可以提高代码的可维护性和可扩展性。
11. **使用生成器表达式**：在需要遍历集合进行条件判断时，使用生成器表达式可以提高效率并减少内存使用。
12. **将条件简化为return语句**：在函数需要返回布尔值时，直接返回条件表达式的计算结果，可以简化代码并提高可读性。

第六章

1.

正确
错误：关键字参数可以在位置参数之后，也可以在位置参数之前。关键字参数的顺序并不影响其与参数名的关联。
不会报错，输出如下

```
a
('b', 'c', 'd')
{'name': 'Sarah', 'age': 18}
```

2.

```python
import random

# 函数用于获取有效输入，确保概率在(0,1)区间内
def get_inputs():
    while True:
        try:
            probability = float(input("请输入单球获胜概率（0到1之间）："))
            if 0 < probability < 1:
                return probability
            else:
                print("输入错误，概率必须在0和1之间，请重新输入。")
        except ValueError:
            print("输入错误，请输入一个有效的数字。")

# 函数用于模拟一局比赛
def simulate_game(home_prob, away_prob):
    home_score, away_score = 0, 0
    while True:
        home_score += random.random() < home_prob
        away_score += random.random() < away_prob
        if (home_score >= 21 and home_score - away_score >= 2) or (away_score >=
21 and away_score - home_score >= 2):
            break
    return home_score, away_score

# 函数用于模拟一场比赛（三局两胜制）
def simulate_match(home_prob, away_prob):
    home_wins, away_wins = 0, 0
    while home_wins < 2 and away_wins < 2:
        home_score, away_score = simulate_game(home_prob, away_prob)
        if home_score > away_score:
            home_wins += 1
        else:
            away_wins += 1
    return home_wins > away_wins

# 模拟1万场比赛的结果
def simulate_matches(num_matches, home_prob, away_prob):
    home_wins = 0
    for _ in range(num_matches):
        if simulate_match(home_prob, away_prob):
            home_wins += 1
    return home_wins / num_matches

# 主程序
if __name__ == "__main__":
    home_prob = get_inputs()  # 主队获胜概率
    away_prob = get_inputs()  # 客队获胜概率
    num_matches = 10000  # 模拟的比赛场数
    win_rate = simulate_matches(num_matches, home_prob, away_prob)
    print(f"在模拟的{num_matches}场比赛中，主队胜率为：{win_rate:.4f}")
```

3.

```
from collections import Counter

# 定义绕口令
tongue_twister = """
八百标兵奔北坡，
北坡八百炮兵炮。
标兵怕碰炮兵炮，
炮兵怕把标兵碰。
"""

# 移除换行符并统计字符频次
char_count = Counter(tongue_twister.replace('\n', ''))

# 按频次降序排列字符
sorted_char_count = char_count.most_common()

# 打印结果
for char, count in sorted_char_count:
    print(f"字符 '{char}' 出现了 {count} 次")
```