CCF 编程能力等级认证
Grade Examination of Software Programming

Python 六级

2026 年 03 月

1 单选题（每题 2 分，共 30 分）

题号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
答案	C	C	D	A	D	B	A	C	D	B	B	B	C	C	A

第 1 题 以下关于 Python 类继承的代码，执行后输出结果是？（）

```python
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        return "动物叫声"

class Dog(Animal):
    def speak(self):
        return f"{self.name}: 汪汪汪"

class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name)  # 调用父类构造方法
        self.color = color
    def speak(self):
        return f"{self.name} ({self.color}) : 喵喵喵"

dog = Dog("旺财")
cat = Cat("咪宝", "橘色")
print(dog.speak(), cat.speak())
```

- A. 动物叫声 动物叫声
- B. 旺财: 汪汪汪 咪宝: 喵喵喵
- C. 旺财: 汪汪汪 咪宝 (橘色) : 喵喵喵
- D. 动物叫声 咪宝 (橘色) : 喵喵喵

第 2 题 下列代码中，`s1.draw()` 和 `s2.draw()` 能正确运行并输出不同结果的主要原因是（ ）。

```python
class Shape:
    def draw(self):
        print("绘制图形")

class Circle(Shape):
    def draw(self):
        print("绘制圆形")

class Rectangle(Shape):
    def draw(self):
        print("绘制矩形")

if __name__ == "__main__":
    s1 = Circle()
    s2 = Rectangle()

    s1.draw()
    s2.draw()
```

- ☐ **A.** `draw()` 是普通成员函数
- ☐ **B.** `Shape` 中的 `draw()` 被声明为虚函数
- ☐ **C.** `Circle` 和 `Rectangle` 中使用了公有继承
- ☐ **D.** 对象变量名不同

第 3 题 下面的代码在主程序 `if __name__ == "__main__":` 中有没有一行会导致运行错误，如果有请找出错误行（）

```python
class Pet:
    def __init__(self, n, a):
        self._name = n
        self._age = a
    def get_name(self):
        return self._name
    def birthday(self):
        self._age += 1

if __name__ == "__main__":
    cat = Pet("奶茶", 2)
    print(cat.get_name())    # ①
    cat.birthday()           # ②
    cat._name = "大橘"       # ③
    print(cat.get_name())    # ④
```

- ☐ **A.** 第 ① 行
- ☐ **B.** 第 ② 行
- ☐ **C.** 第 ③ 行
- ☐ **D.** 无错误行

第 4 题 游乐园的过山车每次限坐 `4` 人，用循环队列管理排队（容量 `MAX=5`，空一格判满）。下面代码执行后，循环队列是否已满？`rear` 的值是多少？

```python
MAX = 5
queue = [None] * MAX
front = 0
rear = 0
def enqueue(x):
    global rear, queue
    queue[rear] = x
    rear = (rear + 1) % MAX

def dequeue():
    global front
    front = (front + 1) % MAX

if __name__ == "__main__":
    enqueue(1)
    enqueue(2)
    enqueue(3)
    enqueue(4)

    dequeue()
    dequeue()

    enqueue(5)
    enqueue(6)

    print("队列数组：", queue)
    print("front下标位置：", front)
    print("rear下标位置：", rear)
```

- ☐ **A.** 已满， `rear = 1`
- ☐ **B.** 未满， `rear = 1`
- ☐ **C.** 已满， `rear = 2`
- ☐ **D.** 未满， `rear = 4`

第 5 题 在以下计算机系统应用场景中，最适合使用循环队列的是（ ）。

- ☐ **A.** 函数调用过程中，保存局部变量和返回地址
- ☐ **B.** 表达式求值中的运算符优先级处理
- ☐ **C.** 操作系统中的进程优先级调度（高优先级先执行）
- ☐ **D.** 生产者和消费者问题中的共享缓冲区

第 6 题 在二叉搜索树（BST）中，若中序遍历的序列为 {1, 2, 3, 4, 5}，且先序遍历的第一个序列元素为 3，则下列说法正确的是（ ）。

- ☐ **A.** 该树一定是一棵完全二叉树。
- ☐ **B.** 元素4和5不可能是兄弟节点。
- ☐ **C.** 元素1所在节点的深度可能大于3（根节点深度为1）。
- ☐ **D.** 元素2一定是元素1的父节点。

第 7 题 某二叉树共有10个结点，记为A~J，已知它的先序遍历序列为：**A B D H I E C F J G**，中序遍历序列为：**H D I B E A F J C G**，则该二叉树的后序遍历序列是（ ）。

- ☐ **A.** H I D E B J F G C A
- ☐ **B.** H I D B E J F G C A
- ☐ **C.** I H D E B J F G C A

- [ ] **D.** H I D E B F J G C A

第 8 题 下列关于树的遍历的说法中，正确的一项是（ ）。

- [ ] **A.** 对任意一棵树进行深度优先遍历，所得序列一定唯一。
- [ ] **B.** 已知一棵二叉树的先序遍历和后序遍历序列，可以唯一确定这棵二叉树。
- [ ] **C.** 若一棵二叉树的先序遍历序列与中序遍历序列相同，则该二叉树一定为只有右子树的链式结构。
- [ ] **D.** 已知一棵二叉树的先序遍历序列，即可唯一地确定该二叉树的结构。

第 9 题 有 `6` 个字符，它们出现的次数分别为：`{2, 3, 3, 4, 6, 8}`，现在用哈夫曼编码为这些字符编码，最小加权路径长度 WPL（每个字符的出现次数×它的编码长度，再把每个字符结果加起来）的值为（ ）。

- [ ] **A.** 58
- [ ] **B.** 60
- [ ] **C.** 62
- [ ] **D.** 64

第 10 题 对 n 个不同符号的符号进行哈夫曼编码。若生成的哈夫曼树共有 115 个结点，则 n 的值是（）。

- [ ] **A.** 60
- [ ] **B.** 58
- [ ] **C.** 57
- [ ] **D.** 56

第 11 题 关于格雷编码（Gray Code），下列说法正确的是（ ）。

- [ ] **A.** 格雷编码中，编码位数越多，相邻编码之间变化的位数也越多
- [ ] **B.** 格雷编码中，相邻两个编码的二进制位恰好有一位不同
- [ ] **C.** 格雷编码就是把普通二进制编码按位取反后得到的结果
- [ ] **D.** 格雷编码不能用于数字电路和状态转换的设计中

第 12 题 给定一棵二叉树，采用广度优先搜索 (BFS) 算法，返回右视图所有节点的值。其中右视图定义为：二叉树的右视图是从树的右侧看过去时可见的节点集合，即右视图中的每个节点都是某一层中最右侧的节点。

```python
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

def rightSideView(root):
    rightmost_value_at_depth = {}
    max_depth = -1

    node_queue = []
    depth_queue = []
    node_queue.append(root)
    depth_queue.append(0)

    while node_queue:
        node = node_queue.pop(0)
        depth = depth_queue.pop(0)

        if node is not None:
            max_depth = max(max_depth, depth)
            rightmost_value_at_depth[depth] = node.val

            node_queue.append(node.left)
            node_queue.append(node.right)

            # (1)
            depth_queue.append(__________)
            depth_queue.append(__________)


    right_view = []
    # (2) 补全
    for depth in range(____________):
        right_view.append(rightmost_value_at_depth[depth])

    return right_view

if __name__ == "__main__":
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.left.right = TreeNode(5)
    root.right.right = TreeNode(4)

    print(rightSideView(root))
```

- ☐ **A.** （1） `depth, depth`；（2） `0, max_depth`
- ☐ **B.** （1） `depth + 1, depth + 1`；（2） `0, max_depth + 1`
- ☐ **C.** （1） `depth + 1, depth + 1`；（2） `1, max_depth + 1`
- ☐ **D.** （1） `depth, depth`；（2） `1, max_depth`

第 13 题 下列关于树的深度优先搜索（DFS）的说法中，正确的是（ ）。

- ☐ **A.** 对树进行 DFS 时，一定是按层从上到下依次访问结点
- ☐ **B.** 对任意一棵树进行 DFS，得到的遍历序列唯一
- ☐ **C.** 对一棵树进行 DFS 时，常借助递归或栈实现
- ☐ **D.** DFS 只能用于二叉树，不能用于普通树

第 14 题 小朋友们去邻里拜年，每个家里有不同数量的糖果。规则是：不能连续进入两个相邻的房子（即不能同时取相邻两家的糖果）。目标是拿到最多糖果。以下是代码实现，请补全横线。

```
def visit(nums):
    if not nums:
        return 0

    size = len(nums)
    if size == 1:
        return nums[0]

    dp = [0] * size
    dp[0] = nums[0]
    dp[1] = max(nums[0], nums[1])

    for i in range(2, size):
        ____________________________

    return dp[size - 1]
if __name__ == "__main__":
    nums1 = [1,2,3,1]
    print(visit(nums1))

    nums2 = [2,7,9,3,1]
    print(visit(nums2))

    nums3 = [5]
    print(visit(nums3))

    nums4 = []
    print(visit(nums4))
```

- ☐ **A.** `dp[i] = dp[i - 1] + nums[i]`
- ☐ **B.** `dp[i] = max(dp[i - 1], dp[i - 2] * nums[i])`
- ☐ **C.** `dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])`
- ☐ **D.** `dp[i] = dp[i - 2] + nums[i]`

第 15 题 元宵节晚上，小朋友沿着一条发光石板路前进，每次可以向前走 1 块或 2 块石板。若 `dp[i] = dp[i - 1] + dp[i - 2]`，下面关于 `dp[i]` 的含义最合适的是（ ）。

- ☐ **A.** 走到第 `i` 块石板的不同走法数量
- ☐ **B.** 走到第 `i` 块石板时，已经走过的石板总数
- ☐ **C.** 从第 `i` 块石板走回起点的最少步数
- ☐ **D.** 从第 `i` 块石板走回起点的最大步数

2　判断题（每题 2 分，共 20 分）

题号	1	2	3	4	5	6	7	8	9	10
答案	×	×	×	×	√	√	×	√	×	√

第 1 题 下面定义了一个表示二维坐标点的类 `Point`，并提供了一个带参数的构造函数，但第 ② 行 `Point b;` 会调用编译器自动生成的默认构造函数，将 `b.x` 和 `b.y` 被初始化为 0.0，程序可以正常编译运行。

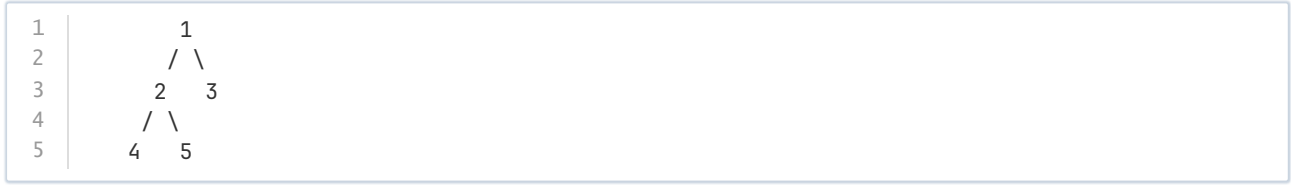
```python
class Point:
    def __init__(self, px, py):
        self.x = px
        self.y = py

    def print(self):
        print(f"({self.x}, {self.y})")

if __name__ == "__main__":
    a = Point(3.0, 4.0)  # ①
    # ②
    b = Point()
    a.print()
```

第 2 题 Python 中的继承支持单继承和多继承，但子类无法直接访问父类的私有成员。

第 3 题 对如下结构的树，执行 `travel` 函数，输出结果是 `1 2 3 4 5`。




```python
class Node:
    def __init__(self, v):
        self.val = v
        self.left = None
        self.right = None

def travel(root):
    if not root:
        return
    stack = []
    stack.append(root)

    while stack: =
        cur = stack.pop()
        print(cur.val, end=" ")
                if cur.right:
            stack.append(cur.right)
        if cur.left:
            stack.append(cur.left)

if __name__ == "__main__":

    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)
    travel(root)
```

第 4 题 若所有字符出现频率相同，则哈夫曼编码一定会得到完全二叉树。

第 5 题 哈夫曼编码是一种变长的前缀编码，在解码时不需要额外的分隔符就能唯一还原，这是因为在哈夫曼树中，任何一个字符的叶子结点都不会成为另一个字符结点的祖先。

第 6 题 在 Python 中使用列表存储按层序遍历的完全二叉树时，若根节点存储在 `tree[0]`，则对于任意非空节点 `tree[i]`，其右孩子（如果存在）必然位于 `tree[2 * i + 2]`。

第 7 题 在 Python 中使用列表模拟栈来非递归地实现二叉树的前序遍历，为了保证遍历顺序正确，在处理完当前结点后，应该先将该结点的左孩子压入栈中，然后再将右孩子压入栈中。（）

第 8 题 设二叉树共有 n 个结点，函数 `preorderTraversal` 的时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

```python
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

def preorder(root, res):
    if root is None:
        return
    res.append(root.val)
    preorder(root.left, res)
    preorder(root.right, res)
def preorderTraversal(root):
    res = []
    preorder(root, res)
    return res


if __name__ == "__main__":

    root = TreeNode(1)
    root.right = TreeNode(2)
    root.right.left = TreeNode(3)
    result = preorderTraversal(root)
    print("前序遍历结果: ", result)
```

第 9 题 以下代码实现了0-1背包问题的一维动态规划解法，内层循环采用经典的逆序遍历方式。若将内层循环改为正序遍历（即 `for j in range(w[i], W + 1):`），仍能得到正确答案。

```python
def knapsack_01():
    W = 5
    w = [2, 3, 4]
    v = [10, 1, 1]
    n = 3
    dp = [0] * (W + 1)

    for i in range(n):
        for j in range(W, w[i] - 1, -1):
            dp[j] = max(dp[j], dp[j - w[i]] + v[i])

    print(dp[W])

knapsack_01()
```

第 10 题 在动态规划问题中，状态空间相同且没有重复计算的情况下，“状态转移方程+递推”与“递归+记忆化搜索”的时间复杂度通常相同。

3 编程题（每题 25 分，共 50 分）

3.1 编程题 1

- **试题名称**：选数
- **时间限制**：1.0 s
- **内存限制**：512.0 MB

3.1.1 题目描述

给定两个包含 n 个整数的数组 $a = [a_1, \ldots, a_n]$ 与 $b = [b_1, \ldots, b_n]$。你需要指定若干下标 $p_1 < \cdots < p_k$ （$1 \le k \le n$）使得以下条件成立：

- $1 \le p_i \le n$（$1 \le i \le k$）；
- $p_{i+1} \ge p_i + b_{p_i}$（$1 \le i < k$）。

你需要在满足以上条件的前提下最大化 $\sum_{i=1}^{k} a_{p_k}$，也即最大化数组 a 对应下标的整数之和。

3.1.2 输入格式

第一行，一个正整数 n，表示数组长度。

第二行，n 个正整数 $a_1, a_2, \ldots, a_n$，表示数组 a。

第三行，n 个正整数 $b_1, b_2, \ldots, b_n$，表示数组 b。

3.1.3 输出格式

一行，一个整数，表示在满足下标条件的前提下，数组 a 对应下标的整数之和的最大值。

3.1.4 样例

3.1.4.1 输入样例 1

```
4
1 2 3 4
3 3 1 1
```

3.1.4.2 输出样例 1

```
7
```

3.1.4.3 输入样例 2

```
6
1 1 4 5 1 4
1 2 3 2 1 0
```

3.1.4.4 输出样例 2

```
11
```

3.1.5 数据范围

对于 40% 的测试点，保证 $2 \leq n \leq 10^3$。

对于所有测试点，保证 $2 \leq n \leq 10^5$，$0 \leq a_i \leq 10^9$，$0 \leq b_i \leq n$。

3.1.6 参考程序

```python
# 接受输入
# n
n = int(input())
# 接受a和b的输入
a = [0] + list(map(int, input().split()))
b = [0] + list(map(int, input().split()))

# 准备动态规划的数组f
# f[i]的含义如题解所述
f = [0] * (n + 5)
# ans表示目前找到的最大的答案
ans = 0

for i in range(1, n + 1):
    # 比较一下，如果将i加入到下标序列中，那么可能得到的最大序列和为f[i]+a[i]，比较这个值和最大的已发现的答案
    ans = max(ans, f[i] + a[i])

    # f[i+b[i]]表示最后一个被选中的下标<=i的情况下能够找到的最大序列和。如果加入a[i]能够让这个值更大，则更新
    if i + b[i] <= n:
        f[i + b[i]] = max(f[i + b[i]], f[i] + a[i])
    # 因为f[i]表示的是i及i以后的下标都能被选中情况下的最大序列和，所以利用f[i]来更新f[i+1]
    f[i + 1] = max(f[i + 1], f[i])

print(ans)
```

3.2 编程题 2

- **试题名称**：完全二叉树
- **时间限制**：1.0 s
- **内存限制**：512.0 MB

3.2.1 题目描述

给定一棵包含 n 个结点的**有根二叉树**，结点依次以 $1, 2, \ldots, n$ 编号，根结点编号为 1。

对于结点 i，其左儿子的编号记为 l_i，右儿子编号记为 r_i。特别地，如果左儿子不存在则 $l_i = 0$，如果右儿子不存在则 $r_i = 0$。

树中每个结点都对应一棵以其为根的子树。请你求出给定有根树的所有 n 棵子树中，有多少棵子树是完全二叉树。

3.2.2 输入格式

第一行，一个正整数 n，表示有根二叉树结点数量。

接下来 n 行，每行两个正整数 l_i, r_i，表示结点 i 的左儿子编号和右儿子编号。

3.2.3 输出格式

输出一行，一个整数，表示所有子树中完全二叉树的数量。

3.2.4 样例

3.2.4.1 输入样例 1

```
4
2 3
4 0
0 0
0 0
```

3.2.4.2 输出样例 1

```
4
```

3.2.4.3 输入样例 2

```
4
2 3
0 0
4 0
0 0
```

3.2.4.4 输出样例 2

```
3
```

3.2.5 数据范围

对于 40% 的测试点，保证 $1 \leq n \leq 500$。

对于所有测试点，保证 $1 \leq n \leq 10^5$。

3.2.6 参考程序

```python
n = int(input())
# 左子节点编号
l = [0] * (n + 1)
# 右子节点编号
r = [0] * (n + 1)
# 节点的“最小叶子节点深度”
mn = [0] * (n + 1)
# 节点的深度
mx = [0] * (n + 1)
# 以节点i为根的子树是否为完全二叉树，1表示是，0表示不是
chk = [0] * (n + 1)

ans = 0

for i in range(1, n + 1):
    li, ri = map(int, input().split())
    l[i] = li
    r[i] = ri

def dfs(u):
    global ans
    # 先假设他是完全二叉树
    chk[u] = 1
    if u == 0:
        return
    # 后序遍历。先遍历左子树，再遍历右子树
    dfs(l[u])
    dfs(r[u])
    # 完全二叉树左右子树必须都是完全二叉树
    chk[u] &= chk[l[u]] & chk[r[u]]
    # 节点u的“最小叶子节点深度”为左右子树“最小叶子节点深度”中较小的+1
    mn[u] = 1 + min(mn[l[u]], mn[r[u]])
    # 节点u的深度为左右子树深度较大的+1
    mx[u] = 1 + max(mx[l[u]], mx[r[u]])

    # 左子树的“最小叶子结点深度”必定大于等于右子树的深度
    chk[u] &= mn[l[u]] >= mx[r[u]]
    # 满深度必定大于等于深度-1
    chk[u] &= mn[u] >= mx[u] - 1

    ans += chk[u]

dfs(1)

print(ans)
```