这个单元格给助教，请忽略!

Score:

Comment:

请实现每个 function 内容，确保最终提交的notebook是可以运行的。

每一题除了必须要报告的 输出/图表，可以添加解释（中文即可）。此外可以自定义其他 function / 变量，自由添加单元格，但请确保题目中给出的 function （如第一题的 Print_values）可以正常调用。

Collaboration:

Collaboration on solving the assignment is allowed, after you have thought about the problem sets on your own. It is also OK to get clarification (but not solutions) from online resources, again after you have thought about the problem sets on your own.

There are two requirements for collaboration:

- Cite your collaborators **fully and completely** (*e.g.*, "XXX explained to me what is asked in problem set 3"). Or cite online resources (*e.g.*, "I got inspired by reading XXX") that helped you.
- Write your scripts and report **independently** - the scripts and report must come from you only.

1. Flowchart

Write a function `Print_values` with arguments `a`, `b`, and `c` to reflect the following flowchart. Here the purple parallelogram operator on a list `[x, y, z]` is to compute and print `x+y-10z`. Try your output with some random `a`, `b`, and `c` values. Report your output when `a = 10, b = 5, c = 1`.

```
# 1.Fowchat

# 根据流程图定义排序函数
def print_value(a, b, c):
    # 使用if条件语句实现变量大小比较
    if a > b:
        if b > c:
            # 用变量value_list接收列表
```

```
            value_list = [a,b,c]
        elif a > c:
            # 同上，value_list接收变量
            value_list = [a,c,b]
        else:
            value_list = [c,a,b]
    elif b > c:
        if a > c:
            value_list = [b,a,c]
        else:
            value_list = [b,c,a]
    else:
        value_list = [c,b,a]

    # 最终输出结果公式:x+y-10z
    final_result = value_list[0] + value_list[1] -
10*value_list[2]
    return final_result

final_result = print_value(10, 5,1)
print(f'最终结果是：{final_result}')
```

```
最终结果是：5
```

最终结果

Report your output when `a = 10, b = 5, c = 1`: `5`

第一题解释

1. 首先想到用if条件语句进行解答，然后发现题干中要求利用列表进行计算。
2. 写出初版程序后，发现每判断一次就要打印一遍公式，代码冗余，于是我将计算公式放在if语句之外，整个if语句只输出一个列表。
3. 最后对列表中的值进行计算，得到最终的结果。

2. Continuous ceiling function

Given a list with `N` positive integers. For every element `x` of the list, find the value of continuous ceiling function defined as `F(x) = F(ceil(x/3)) + 2x`, where `F(1) = 1`.

```
# 2. Continuous celing function
```

```python
# 导入math包，需要里面的ceil函数
# ceil() 函数返回数字的上入整数,即向上取整
import math

# 定义连续天花板函数Continuous celing function 递归
def ceil_function(x):
    # 递归终止条件
    if x == 1:
        return 1  # 基础条件 F(1) = 1

    # 递归调用自身
    # ceil() 函数返回数字的上入整数,即向上取整
    return ceil_function(math.ceil(x / 3)) + 2 * x

# 定义找到ceil值，传入变长参数args
def find_ceil(*args):
    # 定义空列表，用于接收变长参数的值
    ceiling_value = []

    # 遍历args，分别用连续天花板函数其中的元素进行计算
    for element in args:
        # 将计算结果添加进列表
        ceiling_value.append(ceil_function(element))

    # 输出最终的ceil值
    print(ceiling_value)


find_ceil(1,2,3,4)
```

```
[1, 5, 7, 13]
```

最终结果

给定一个列表`[1, 2, 3, 4]`

得到的连续天花板函数值为`[1, 5, 7, 13]`

第二题解释

通过查阅资料，得知ceil() 函数用于返回数字的上入整数。

1. 导入math模块
2. 定义递归的连续天花板函数`ceil_function`:在`x != 1`时，函数递归地调用自身，将输入`x`通过`math.ceil(x / 3)`进行处理，即对`x`除以`3`后进行向上

取整，然后加上 `2 * x`。
这个递归调用的过程会一直进行，直到 x 的值递归到 1 为止。

3. 定义`find_ceil函数`
使用变长参数`*args`实现函数接受不定数量的的参数
最后打印输出计算的天花板值列表。

3. Dice rolling

3.1 Given `10` dice each with `6` faces, numbered from `1` to `6`. Write a function `Find_number_of_ways` to find the number of ways to get sum `x`, defined as the sum of values on each face when all the dice are thrown.

```
# 3.Dice rolling
# 3.1
def Find_number_of_ways(num_of_dice):
    # 初始化二维数组 Number_of_ways，大小为 (num_of_dices + 1 x 6 *
num_of_dices + 1)，所有元素初始化为 0
    # 这个数组用于接收所有骰子总和的方案数
    Number_of_ways = []
    for i in range(num_of_dice + 1):
        Number_of_ways_row = []
        for j in range(num_of_dice * 6 + 1):
            Number_of_ways_row.append(0)
        Number_of_ways.append(Number_of_ways_row)

    # 设置边界条件，为递推公式的初始项做准备
    # 边界条件1：没有骰子且总和为0的情况有一种解法（什么都不做）
    Number_of_ways[0][0] = 1

    # 边界条件2：当只有一个骰子时，1到6的总和各有一种组合方式
    for s in range(1, 7):
        Number_of_ways[1][s] = 1

    # 递推公式
    for d in range(2, num_of_dice + 1):
        for s in range(num_of_dice * 6 + 1):
            # # d个骰子和的组合数，可以从d-1骰子的组合数中得到，例如当d = 1
时，和为1-6的组合数都是1；当d=2时，和为
            # Number_of_ways[d][s] = Number_of_ways[d-1][s-1] +
Number_of_ways[d-1][s-2] +
            # Number_of_ways[d-1][s-3] + Number_of_ways[d-1][s-4]
+ Number_of_ways[d-1][s-5] + Number_of_ways[d-1][s-6]
```

```
            for value_dice in range(1, 7):  # 当前骰子可能的结果是 1
到 6
                if s - value_dice >= 0:  # 确保索引合法
                    Number_of_ways[d][s] += Number_of_ways[d-1]
[s-value_dice]

    return Number_of_ways

Number_of_ways_test = Find_number_of_ways(10)

print(f'投10个骰子，总和为20的方案数：{Number_of_ways_test[10][20]}')
```

```
投10个骰子，总和为20的方案数：85228
```

最终结果

投`10`个骰子，总和为`20`的方案数：`85228`

3.2 Count the number of ways for any `x` from `10` to `60`, assign the number of ways to a list called `Number_of_ways`, so which `x` yields the maximum of `Number_of_ways`?

```
# 3.2
# 在方案数二维数组中找到最大方案数的总和及其索引
max_value = max(Number_of_ways_test[10])
max_index = Number_of_ways_test[10].index(max_value)

# 输出最终结果
print(f'投10个骰子时，总和为{max_index}的方案数最大，方案数为
{max_value}')

# 结果:10个骰子时，总和为35的方案数最大，方案数为4395456
```

```
投10个骰子时，总和为35的方案数最大，方案数为4395456
```

最终结果

So which `x` yields the maximum of `Number_of_ways`? `[35]`

第三题解释

3.1

使用动态规划解决问题。

`动态规划`（Dynamic Programming, 简称DP）是一种将复杂问题分解成较小的子问题，并通过记忆子问题的结果来避免重复计算的算法技巧。它的核心思想是`“分而治之 + 记忆化”`。
source：https://zhuanlan.zhihu.com/p/365698607

实现该问题大致分为3步:

1. 初始化二维数组
2. 设置边界条件:
   - 边界条件1：没有骰子且总和为0的情况有一种解法（什么都不做）
     `dp[0][0] = 1`
   - 边界条件2：当只有一个骰子时，1到6的总和各有一种组合方式
3. 递推公式:
   - d个骰子和的组合数，可以从d-1骰子的组合数中得到，递推公式可以表示为：
     `dp[d][s] = dp[d-1][s-1] + dp[d-1][s-2] + dp[d-1][s-3] + dp[d-1][s-4] + dp[d-1][s-5] + dp[d-1][s-6]`
   - 例如当`d=2`时，`dp[2][3] = dp[1][2] + dp[1][1]`
     意思是，想要用2个骰子投出总和为3的数，1个骰子总和为1时，第二个骰子应该投2，即`dp[1][2]`;同理，1个骰子投出2时，第二个应该是1，即`dp[1][1]`。

3.2

在3.1的代码中已经创建了列表`Number_of_ways`

这个列表是一个二维数组，当骰子的数量是10时，数组的大小是`10 ×61`

- 行：骰子的数量；
- 列：骰子总和的方案数
  对于这个问题，使用函数`max()`就可以找到最大方案数。

4. Dynamic programming

4.1 [5 points] Write a function `Random_integer` to fill an array of `N` elements by randomly selecting integers from `0` to `10`.

```python
# 4. Dynamic programming
# 4.1
import random

def Random_integer(N):
    # 生成包含N个元素的数组，元素是0到10之间的随机整数
    return [random.randint(0, 10) for _ in range(N)]

# 生成一个N为10的数组作为示例
N = 10
random_array = Random_integer(N)

random_array
```

```
[1, 5, 4, 5, 0, 9, 2, 9, 8, 3]
```

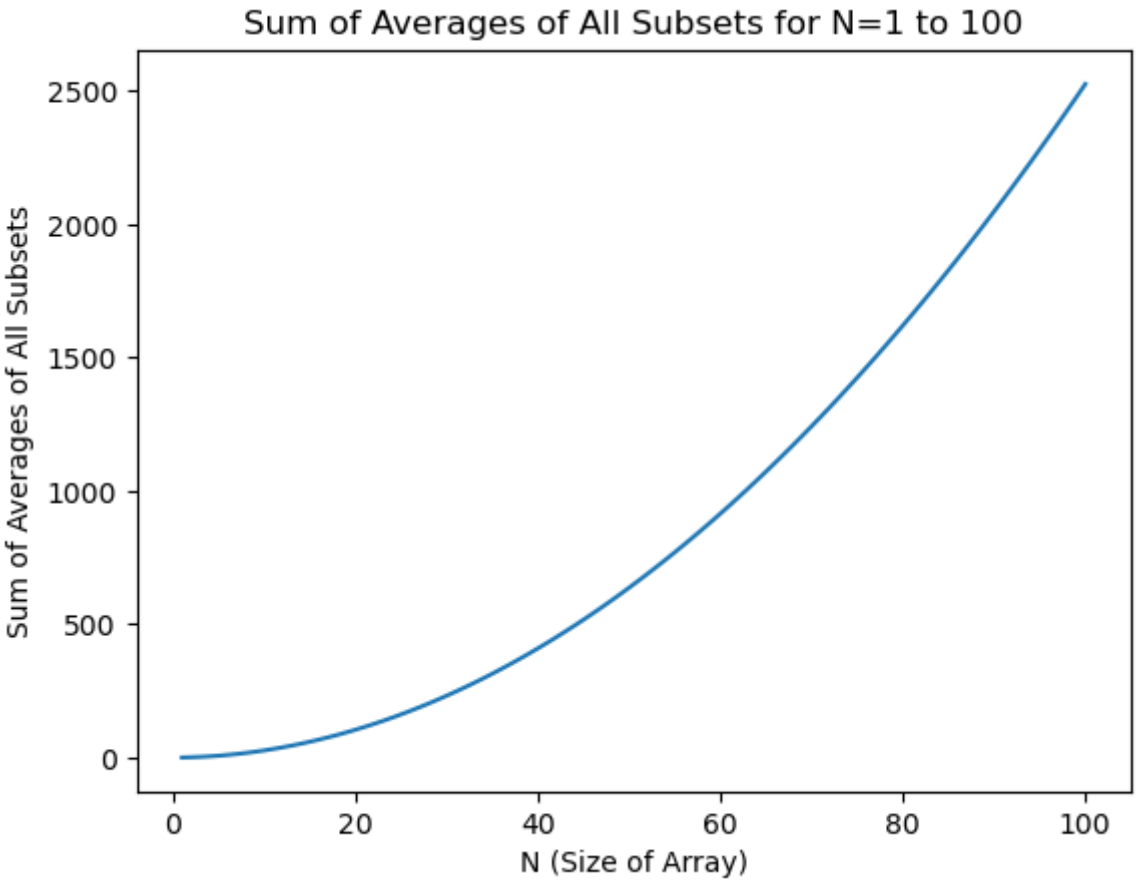
4.2 [15 points] Write a function `Sum_averages` to compute the sum of the average of all subsets of the array. For example, given an array of `[1, 2, 3]`, you `Sum_averages` function should compute the sum of: average of `[1]`, average of `[2]`, average of `[3]`, average of `[1, 2]`, average of `[1, 3]`, average of `[2, 3]`, and average of `[1, 2, 3]`.

```python
# 4.2
# 使用 itertools.combinations 函数生成当前长度 r 的所有子集。
import itertools

def Sum_averages(arr):
    total_sum = 0

    # 生成所有子集
    for r in range(1, len(arr) + 1):
        for subset in itertools.combinations(arr, r):
            total_sum += sum(subset) / len(subset)

    return total_sum

arr = [1, 2, 3]
sum_of_averages = Sum_averages(arr)

print(sum_of_averages)  # 输出结果为 14.0
```

```
14.0
```

4.3 [5 points] Call `Sum_averages` with `N` increasing from `1` to `100`, assign the output to a list called `Total_sum_averages`. Plot `Total_sum_averages`, describe what you see.

```
# 4.3
import matplotlib.pyplot as plt
def Sum_averages_v2(arr):
    N = len(arr)
    total_sum = 0
    subset_count = 2**N - 1  # 所有非空子集的数量

    for i in range(N):
        # 每个元素 arr[i] 在 2^(N-1) 个子集中出现
        total_sum += arr[i] * (2**(N-1))  # 计算每个元素的贡献

    return total_sum / subset_count  # 计算平均值总和

# 生成从1到100的数组，并为每个数组调用优化后的Sum_averages_optimized
Total_sum_averages_v2 = []
for N in range(1, 101):
    arr = list(range(1, N + 1))  # 创建一个从1到N的数组
    result = Sum_averages_v2(arr)
    Total_sum_averages_v2.append(result)

# 绘制优化后的Total_sum_averages_optimized
plt.plot(range(1, 101), Total_sum_averages_v2)

# x y轴名称，图表标题
plt.xlabel('N (Size of Array)')
plt.ylabel('Sum of Averages of All Subsets')
plt.title('Sum of Averages of All Subsets for N=1 to 100')
plt.show()
```



最终结果

Describe what you see. `[随着N增加，所有子集平均和的总和呈指数增长模式]`

第四题解释

4.1

比较简单,使用`random.randint()函数`完成代码。

4.2

针对这个问题，我使用 `itertools.combinations 函数`生成当前长度 `r` 的所有子集。

再结合`sun()函数`和`len()函数`计算子集平均值。

4.3

最开始的思路：

1. 用`for循环`依次创建`1-100`的数组
2. 分别计算这些数组的子集的平均值之和

3. 再使用`ppend()`将结果添加到`Total_sum_averages`的列表中
4. 最后再画图。
   但是，这个方法忽略了一点：生成所有子集并计算它们的平均值对于较大的N是非常耗时的，尤其是当 N 增大到 100 时，子集的组合数量呈指数级增长。(子集的数量：`2^N`)

优化后的思路

我们要计算一个数组中所有子集的平均值的总和。最直接的方法是生成所有可能的子集，计算每个子集的平均值，然后把这些平均值加起来。但是，随着数组的长度（N）增加，子集的数量会成倍增长，比如长度为N的数组有2^N个子集（包括空集）。当N很大时，这个方法会非常慢。

所以我们需要想办法避免直接生成所有子集。

1. 每个元素在子集中的“存在感”

   想象一下，我们有一个数组，比如`[1, 2, 3]`，我们要找到所有子集，然后计算平均值的总和。每个子集的平均值是子集中的所有元素加起来，除以子集的大小。

   现在注意，每个元素在很多不同的子集中会出现。

   每个元素到底在多少个子集中出现，它在这些子集中对平均值的贡献有多大?

2. 每个元素出现的次数

   如果我们仔细观察每个元素会出现在多少个子集中，会发现一个规律：

   假设数组有N个元素，每个元素ai会出现在一半的子集中。为什么呢？因为对于一个元素来说，剩下的N-1个元素要么被选择，要么不被选择。所以每个元素ai出现在2^(N-1)个子集中。

   > 举个例子：如果数组是`[1, 2, 3]`，元素1会出现在`[1]`、`[1, 2]`、`[1, 3]`、`[1, 2, 3]`这四个子集中，而子集总数是`2^3 = 8`，所以1出现在`2^(3-1) = 4`个子集中。

3. 元素对平均值的贡献
   每次元素出现在某个子集中，它对这个子集的平均值的贡献是多少呢？答案是：元素的值除以子集的大小。

   假设我们有子集`[1, 2, 3]`，子集的平均值是`(1 + 2 + 3) / 3`。在这个平均值中，`1`对总结果的贡献是`1/3`，`2`的贡献是`2/3`，`3`的贡献是`3/3`。类似的，对于其他子集，每个元素都会有不同的贡献。

   因此，每个元素会在许多子集中出现，而每次出现，它都会对这些子集的平均值做出贡献。我们需要把这些贡献加起来。

公式总结如下

$$\text{Sum of Averages} = \frac{1}{2^N-1} \sum_{i=1}^{N} 2^{N-i} \cdot a_i$$

最终代码如上所示。

5. Path counting

5.1 [5 points] Create a matrix with `N` rows and `M` columns, fill the right-bottom corner and top-left corner cells with `1`, and randomly fill the rest of matrix with integer `0` or `1`.

```
# 5.1
import numpy as np

def create_matrix(N, M):
    # 创建一个 N x M 的矩阵，元素随机为 0 或 1
    matrix = np.random.randint(0, 2, (N, M))

    # 设置左上角和右下角的元素为 1
    matrix[0, 0] = 1
    matrix[N-1, M-1] = 1

    return matrix

# 尝试生成一个 5x5 的矩阵
N, M = 5, 5
matrix = create_matrix(N, M)
print(matrix)
```

```
[[1 1 0 1 1]
 [0 1 0 1 1]
 [0 1 1 0 1]
 [1 1 1 1 0]
 [0 0 1 0 1]]
```

5.2 [25 points] Consider a cell marked with `0` as a blockage or dead-end, and a cell marked with `1` is good to go. Write a function `Count_path` to count the total number of paths to reach the right-bottom corner cell from the top-left corner cell.

Notice: for a given cell, you are **only allowed** to move either rightward or downward.

```
# 5.2
```

```python
def count_paths(matrix):
    N, M = matrix.shape  # 获取矩阵的行数和列数

    # 如果起点或终点是阻塞的（0），则无路可走
    if matrix[0, 0] == 0 or matrix[N-1, M-1] == 0:
        return 0

    # 创建一个同样大小的矩阵用于存储路径数
    count_of_path = np.zeros((N, M), dtype=int)

    # 初始化起点
    count_of_path[0, 0] = 1

    # 填充count_of_path矩阵
    for i in range(N):
        for j in range(M):
            if matrix[i, j] == 1:
                # 检查是否能从上方到达a
                if i > 0 and matrix[i-1, j] == 1:
                    count_of_path[i, j] += count_of_path[i-1, j]
                # 检查是否能从左侧到达
                if j > 0 and matrix[i, j-1] == 1:
                    count_of_path[i, j] += count_of_path[i, j-1]

    # 返回到达右下角的路径数，如果不可达，则返回 0
    return count_of_path[N-1, M-1]

# 创建一个5x5的矩阵并尝试计算路径数
N, M = 5, 5
matrix_test = np.array([[1, 1, 1, 1, 0],
                        [0, 1, 1, 1, 1],
                        [0, 1, 1, 1, 1],
                        [1, 0, 0, 0, 1],
                        [0, 0, 1, 1, 1]])


total_paths = count_paths(matrix_test)

print(f'矩阵:\n{matrix}')
print(f'总路径数: {total_paths}')
```

```
矩阵:
[[1 1 0 1 1]
 [0 1 0 1 1]
 [0 1 1 0 1]
 [1 1 1 1 0]
 [0 0 1 0 1]]
总路径数: 9
```

5.3 [5 points] Let `N = 10, M = 8`, run `Count_path` for `1000` times, each time the matrix (except the right-bottom corner and top-left corner cells, which remain being `1`) is re-filled with integer `0` or `1` randomly, report the mean of total number of paths from the `1000` runs.

```python
# 5.3创建一个一维数组用于接收路径数量
total_average_paths = []
#
for i in range(1000):
    # 创建10×8的随机数组
    matrix10_8 = create_matrix(10, 8)
    # 使用count_paths函数计算路径
    paths = count_paths(matrix10_8)
    # 将路径数量append数组里面
    total_average_paths.append(paths)
    # print(paths)

# 计算1000个路径数平均值
average_paths = sum(total_average_paths) /
len(total_average_paths)
print(f'总路径平均数是: {average_paths}')
```

```
总路径平均数是: 0.52
```

最终结果

Report the mean of total number of paths from the `1000` runs. `[0.52]`
由于计算的是随机生成的矩阵，因此最终结果是不确定的。

第五题解释

5.1

使用`np.random.randint()函数`创建矩阵

5.2

思路：
对于每一个可以走的格子`(i, j)`：

- 如果能从上面来，就把上面格子的路径数加过来；
- 如果能从左边来，就把左边格子的路径数加过来。
  最终，右下角格子的路径数`dp[N-1][M-1]`就是从左上角到右下角所有可能路径的总数。

5.3

调用前两题的函数，使用`sum(list)/len(list)`计算路径数平均值