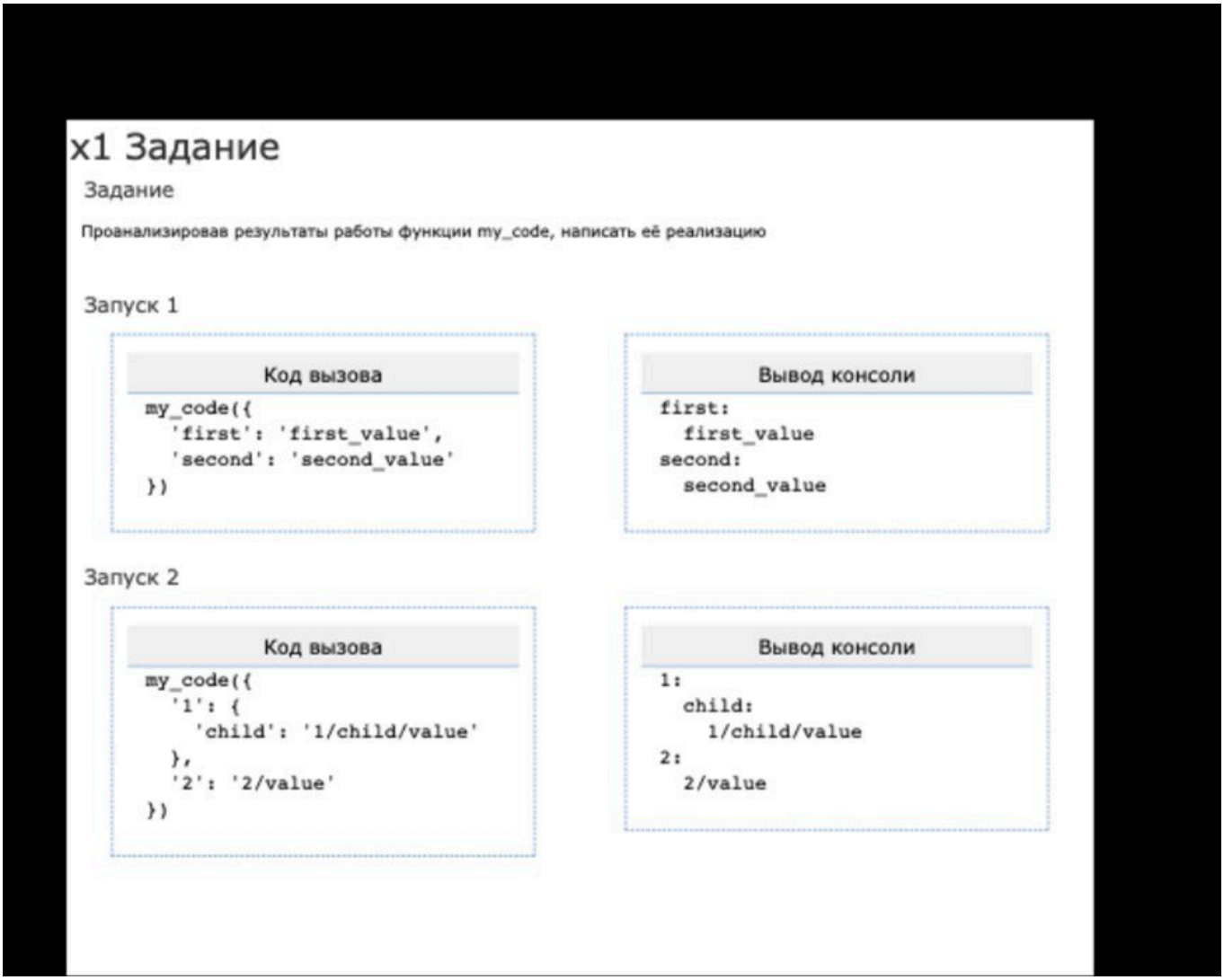
Тестовое задание для INT-13.

Теоретическое задание.

Кейсы для тестирования.
1 - кейс. Ввожу "52". Система выдаёт ответ "Это четное число"
Зачем - Проверяем обработку простого четного положительного числа.
2 - кейс. Ввожу "1337". Система выдаёт "Это не четное число"
Зачем - Проверяем обработку нечетного числа максимально-возможной длины.
3 - кейс. Ввожу "0". Система выдаёт "Это четное число"
Зачем - Проверяем обработку нулевого значения.
4 - кейс. Ввожу "-999". Система выдаёт "Это не четное число"
Зачем - Проверяем обработку минимального отрицательного числа
5-кейс. Ввожу "-44". Система выдаёт "Это четное число"
Зачем - Проверяем обработку четного отрицательного числа. Плюс мы все еще не проверяли вывод для числа, содержащего в вводе три знака. Проверили и это здесь.
6-кейс. Ввожу "1.5". Система выдаёт "Введите целое число"
Зачем - Проверяем обработку дробного числа. Приложение обрабатывает этот случай и просит ввести целое число.
7-кейс. Ввожу "1a2b". Система выдаёт "Введите целое число"
Зачем - Проверяем обработку "нечислового" значения. Приложение обрабатывает этот случай и просит ввести целое число.
8-кейс. Ввожу " 023". Система выдаёт "Это не четное число"
Зачем - Проверяем обработку числа с ведущими нулями и начинающегося с пробела. Приложение обрабатывает этот случай и выдаёт правильный результат.
9-кейс. Ввожу " ". Система выдаёт "Введите не пустое значение"
Зачем - Проверяем обработку пустого ввода. Приложение обрабатывает этот случай и просит ввести значение.
10-кейс. Ввожу "7.00. Система выдаёт "Это нечетное число"
Зачем - Проверяем обработку дробного числа, которое эквивалентно целому числу. Однако происходит вывод "нечетное число", что значит, что он не обрабатывает этот случай корректно. Хотя по условию задания не совсем понятно, но обычно желательна явная обработка типов. С учетом 6го кейса, получается, что разные дробные числа обрабатываются по разному. Зафиксируем то, что программа может принимать и дробные числа, если они эквивалентны целому.

Практическое задание 1.



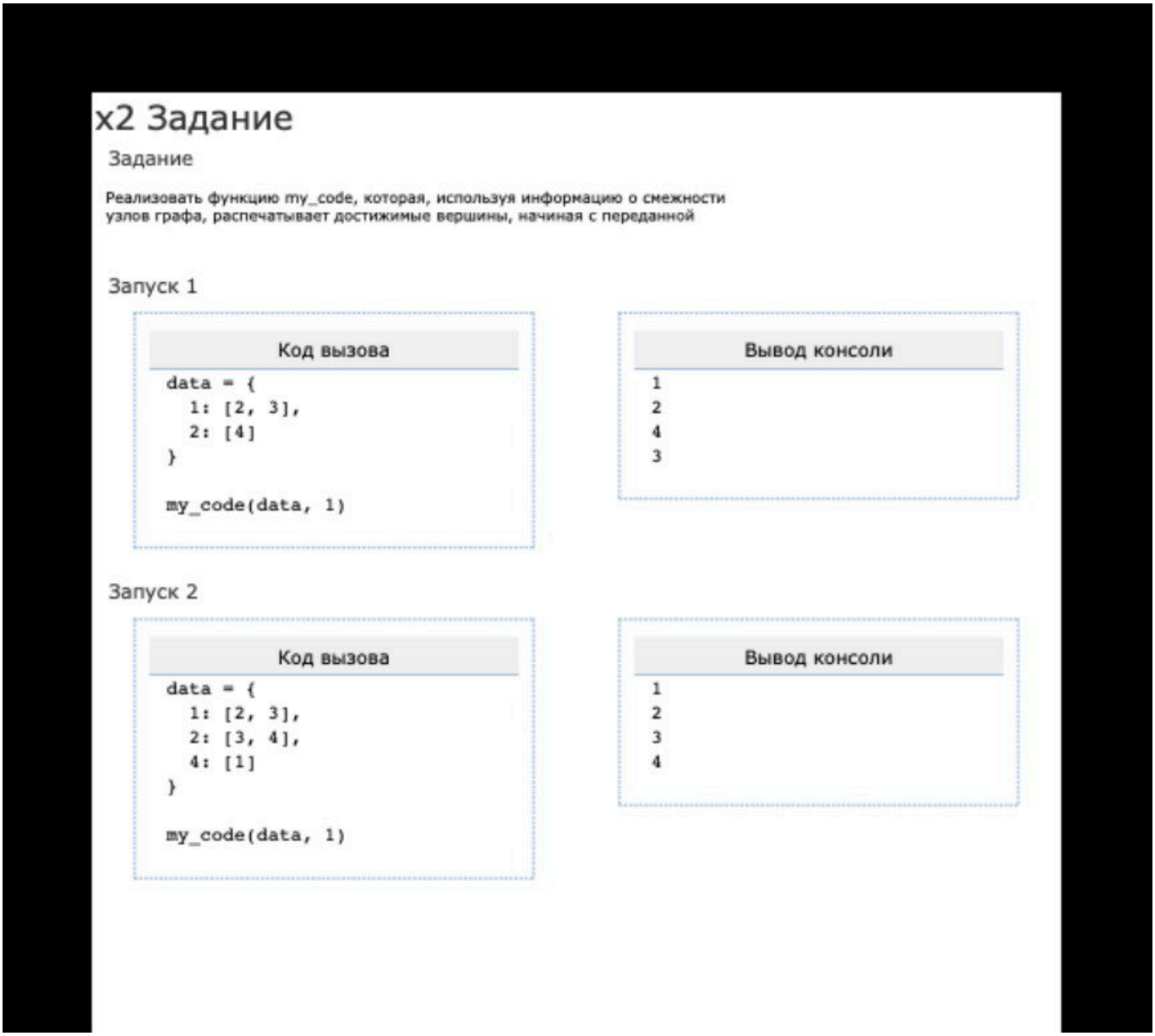
У нас тут простая функция, которая принимает на вход словарь, и выводит данные из него в виде списка, исходя из вложенности.
У меня есть определенная неуверенность, касательно того, что именно в выводе отвечает за отступы - пробелы или табуляция. Я сделал выбор в пользу табуляции, но если там пробелы, то это можно легко получить.
Как можем заметить, количество уровней "вложенности" словарей не ограничено тз, по этой причине мы будем использовать рекурсивный алгоритм. Идея будет такая: принять словарь, разбить по парам "дата-ключ", "ключ" вывести с двоеточием и необходимым отступом, проверить, является ли "значение" словарём, если является - снова рекурсивно вызвать функцию, но уже для значения, если нет - просто вывести значение.
Приступим.

```
def my_code(data, i=0):
    for key, value in data.items():
        print('\t' * i + str(key) + ':')
        if isinstance(value, dict):
            my_code(value, i + 1)
```

```
        else:
            print('\t'* (i + 1) + str(value))
```

Понадобилось ввести дополнительный параметр i, для отслеживания текущего количества отступов. При рекурсивном вызове мы будем увеличивать этот параметр на 1. Изначально зададим его на 0. В остальном всё как я и сказал в идее. isinstance используем для проверки типа, является ли значение типом "dict", то есть словарём.

Практическое задание 2.



Тут от нас требуется реализовать функцию my_code, которая, используя информацию о смежности узлов графа, распечатывает достижимые вершины, начиная с первой.
Какая будет идея? Можно использовать обход графа в глубину или в ширину, но в глубину реализовать легче, так что сделаем его, dfs.
Входные данные представлены в виде списка смежности графа, который представляет из

себя словарь, где ключ - вершина, значение - список допустимых вершин. Так как глубина графа не задаётся, будем использовать рекурсию. В целом никаких изысков тут нет, создаем множество для посещенных вершин(чтобы избежать повторений), реализуем алгоритм dfs, в котором обходим все вершины в глубину, основываюсь на том, есть ли у вершины смежные "соседи", обрабатываем все до тех пор, пока все достижимые не будут обработаны.

```
def my_code(data, start):
        if start not in data:
        return
    visited = set()
    def dfs(node):
        print(node, end='\n')
        visited.add(node)
        for neighbor in data.get(node, []):
            if neighbor not in visited:
                dfs(neighbor)
    dfs(start)
```

Здесь как раз все это реализовано. В 6-й строке использовали "get", чтобы избежать ошибки при возвращении пустого списка(например в первом примере, когда мы доходим до вершины 4, она отсутствует в переданном списке, поэтому вернётся пустой список). Не забудем внести проверку на то, что компонента, с которой мы начинаем, присутствует в списке смежности.

Практическое задание 3.

```
Даны две таблицы в виде списков:
events: list[list] = [[id, timedate, name, …, asset_id], …]
Id – PK, asset_id – FK (Null)
assets: list[list] = [[id, name, …], …]
Id – PK

Необходимо вывести результат работы запроса:  #реализовать функцию query
SELECT event.id, event.name, asset.id, asset.name
FROM events as event LEFT JOIN assets as asset ON event.asset_id = asset.id
ORDER BY event.id
LIMIT 100
"""

def query(events:  list, assets: list) -> list:
    pass

# Пример использования
events = [
    [4, '2024-03-28', 'Event 4', 1],
    [1, '2024-03-26', 'Event 1', 1],
    [6, '2024-03-29', 'Event 6', 3],
    [3, '2024-03-28', 'Event 3', 2],
    [5, '2024-03-29', 'Event 5', None],
    [2, '2024-03-27', 'Event 2', None],
    # Дополнительные события...
]
assets = [
    [4, 'Asset 4'],
    [1, 'Asset 1'],
    [3, 'Asset 3'],
    [2, 'Asset 2'],
    # Дополнительные активы...
]
result = query(events, assets)
for row in result:
    print(row)
# Вывод
# [1, 'Event 1', 1, 'Asset 1']
# [2, 'Event 2', None, None]
# [3, 'Event 3', 2, 'Asset 2']
# [4, 'Event 4', 1, 'Asset 1']
# [5, 'Event 5', None, None]
# [6, 'Event 6', 3, 'Asset 3']
```

Нам нужно реализовать функцию query, которая будет выполнять действия, эквивалентные выполнению SQL-запроса(я так понял ТЗ). На вход подаётся 2 двумерных массива - events и assets. В выводе мы должны вывести двумерный массив, содержащий id ивента, название ивента, id ассета, название ассета и отсортировать по id ивента по возрастанию. Для удобства, так как в списке ассетов у нас есть только id и name, представим их в виде словаря.

После чего с помощью цикла пройдёмся по двумерному массиву с ивентами и необходимые оттуда данные вставим в готовый массив. Далее с помощью sort выполняем сортировку по event id. Не забудем и то, что нам нужно вывести только первые 100, поэтому оставшиеся обрежем. Вот что получится

```
def query(events: list, assets: list) -> list:
    asset_d = {asset[0]: asset[1] for asset in assets}
    result = []
    for event in events:
        event_id, event_date, event_name, asset_id = event
        asset_name = asset_d.get(asset_id, None)
        result.append([event_id, event_name, asset_id, asset_name])
    result.sort(key=lambda x: x[0])
    return result[:100]
```

Добавим тестиков

Задание 1.

Думаю целесообразно будет добавить тестовых запусков. Начнем с первого задания.
Тест 1.

```
data_1 = {
    'A': {'B': {'C': 'A/B/C Value'}},
    'X': 'X Value'
}
```

Здесь сделаем вложенный словарь с многоуровневой вложенностью. В выводе имеем.

```
A:
        B:
                C:
                        A/B/C Value
X:
        X Value
```

Тест 2.

```
data_2 = {}
```

Для второго теста используем пустой словарь. В выводе имеем.

Тест 3.

```
data_3 = {
    'level1': {
        'level2_a': {
            'level3': 'deep_value'
        },
        'level2_b': 42
    },
    'another_key': {
        'another_dict': {
            'new_key': [1, 2, 3]
        }
    }
}
```

Здесь сделаем словарь с несколькими уровнями вложенности и разными типами данных. В выводе имеем.

```
level1:
        level2_a:
                level3:
                        deep_value
        level2_b:
                42
another_key:
        another_dict:
                new_key:
                        [1, 2, 3]
```

Тест 4.

```
data_4 = {
    'string': 'PT',
    'integer': 123,
    'list': [1, 2, 3],
    'dict': {
        'key': 'value'
```

```
    }
}
```

Здесь сделаем упор на разные типы данных. В выводе имеем.

```
string:
        PT
integer:
        123
list:
        [1, 2, 3]
dict:
        key:
                value
```

Тест 5.

```
data_5 = {
    'key1': 'value1',
    'key2': 'value2',
    'key3': 'value3'
}
```

Тут делаем упор на несколько ключей на одном уровне. В выводе имеем.

```
key1:
        value1
key2:
        value2
key3:
        value3
```

Для запуска программы со своими тестами, можете заменить тест data_0, который является тестом из ТЗ.

Задание 2.

Сделаем несколько дополнительных тестовых запусков для второго задания.
Тест 1.

```
data_1 = {
    1: [2, 3],
    2: [3, 4, 5],
    3: [6],
    4: [1, 7],
    7: [8]
}
```

В первом тесте рассмотрим большой граф с циклом и несколькими ветвлениями. В выводе имеем.

```
1
2
3
6
4
7
8
5
```

Тест 2.

```
data_2 = {
    1: [2],
    2: [3],
    3: [4],
    4: [5]
}
```

Для второго теста рассмотрим простой линейный граф. В выводе имеем.

```
1
2
3
4
5
```

Тест 3.

```
data_3 = {
    1: [2, 3],
    2: [4, 5],
    3: [6, 7],
}
```

В третьем тесте возьмем небольшое древо. В выводе имеем.

```
1
2
4
5
3
6
7
```

Тест 4.

```
data_4 = {
    1: [2],
    2: [3],
    4: [5]
}
```

В 4-м тесте используем два независимых подграфа. Выполним запуск с обоих компонент, чтобы проверить, что оба подграфа достижимы.

```
Запуск с первой компоненты(для первого подграфа)
1
2
3
Запуск со второй(с 4)(для второго подграфа)
4
5
```

Тест 5.

```
data5 = {}
```

Для 5-го теста просто попробуем внести пустой граф. Вывод, ожидаемо, будет пустым.

```

```

Для внесения своих тестов также можете просто внести свои данные в data_0, или в любой другой тест.

Задание 3.

Ну и несколько запусков для задания 3.
Тест 1.

```
events1 = [
    [1, '2024-03-26', 'Event 1', 1],
    [2, '2024-03-27', 'Event 2', 2],
    [3, '2024-03-28', 'Event 3', 3],
    [4, '2024-03-29', 'Event 4', 4]
]

assets1 = [
    [1, 'Asset 1'],
    [2, 'Asset 2'],
    [3, 'Asset 3'],
    [4, 'Asset 4']
]

result_1 = query(events1, assets1)
for row in result_1:
    print(row)
```

В качестве первого теста используем простой случай - все events_id совпадают с assets_id. В выводе имеем.

```
[1, 'Event 1', 1, 'Asset 1']
[2, 'Event 2', 2, 'Asset 2']
[3, 'Event 3', 3, 'Asset 3']
[4, 'Event 4', 4, 'Asset 4']
```

Тест 2.

```
events2 = [
    [1, '2024-03-26', 'Event 1', 1],
    [2, '2024-03-27', 'Event 2', None],
    [3, '2024-03-28', 'Event 3', 3],
    [4, '2024-03-29', 'Event 4', 5]
]

assets2 = [
    [1, 'Asset 1'],
    [3, 'Asset 3']
]

result_2 = query(events2, assets2)
for row in result_2:
    print(row)
```

В качестве второго теста используем случай, когда в assets отсутствует часть id, на которые ссылается events. В выводе имеем.

```
[1, 'Event 1', 1, 'Asset 1']
[2, 'Event 2', None, None]
[3, 'Event 3', 3, 'Asset 3']
[4, 'Event 4', 5, None]
```

Тест 3.

```
events3 = [
    [4, '2024-03-29', 'Event 4', 4],
    [1, '2024-03-26', 'Event 1', 1],
    [3, '2024-03-28', 'Event 3', 3],
    [2, '2024-03-27', 'Event 2', 2]
]

assets3 = [
    [1, 'Asset 1'],
    [2, 'Asset 2'],
    [3, 'Asset 3'],
    [4, 'Asset 4']
```

```
]

result_3 = query(events3, assets3)
for row in result_3:
    print(row)
```

В качестве 3-го теста рассмотрим случай, когда в events нет сортировки по event_id. В выводе получим.

```
[1, 'Event 1', 1, 'Asset 1']
[2, 'Event 2', 2, 'Asset 2']
[3, 'Event 3', 3, 'Asset 3']
[4, 'Event 4', 4, 'Asset 4']
```

Тест 4.

```
events4 = []
assets4 = [
    [1, 'Asset 1'],
    [2, 'Asset 2']
]

result_4 = query(events4, assets4)
print(result_4)
```

В качестве 4го теста рассмотрим ситуацию, когда список events пустой, а список assets не пустой. В выводе имеем пустой список.

```
[]
```

Тест 5.

```
events5 = [
    [1, '2024-03-26', 'Event 1', 1],
    [2, '2024-03-27', 'Event 2', None]
]

assets5 = []
```

```
result_5 = query(events5, assets5)
for row in result_5:
    print(row)
```

В качестве 5-го теста рассмотрим наоборот ситуацию, когда список events не пустой, а список assets пустой. В выводе имеем.

```
[1, 'Event 1', 1, None]
[2, 'Event 2', None, None]
```

Ваши тесты для практик

Для того чтобы вы могли провести свои тесты для практик, я добавил в репозиторий практики в директории Test0. Там присутствуют только одни тестовые данные, используемые в ТЗ. Так что вы можете внести свои данные вместо них и выполнить тестирование. Каждые данные, которые я подаю, я заключаю в переменные, так что для запуска необходимо просто поменять их значения и запустить приложение.