Exercise 7)

The following program simulates incomplete functionality of a restaurant app, namely:

* Assigning available in the restaurant tables.
* Seating and vacating of tables.
* Taking orders.
* Reading menu from a excel file.

A Python interpreter 3.10 or newer must be installed with the libraries and support programs of the standard installation from www.python.org.

Start the program "execute.py" in the usual way for the respective operating system, from the interpreter shell or in an IDE, e.g. IDLE.

For further information on the documentation, see Analysis.

1. Analysis:

Within the two files included, all the building blocks have been added.

In specific:

**teamwork.py** – contains 3 classes that that simulate the key objects of the restaurant.

**class Menu:** Using the pandas package, the module automatically reads the excel file with the menu (food.xlsx) from the same directory as the initial file and creates a nested dictionary with the following structure:

*{‘categorie’: {‘type’}: {‘position’: float(price)}}.*

It’s function, ***show\_menu***, allows to print the menu legibly.

The second function ***take\_position*** searches for a specific item in the menu recursively and returns a dictionary with the structure:

*{‘position’: str(), ‘price’: float(), ‘amount’: int()}.*

**class Order:** This class represents an order from one visitor and has 2 attributes: order\_id (unique for every order) and *order\_per\_person*.

*Order\_per\_person* is a list with the order\_id and dictionaries (ordered positions):

*['1', {'position': 'PIZZA-HAWAII', 'price': 10.0, 'amount': 2, 'preferences': None}, }, {'position': 'COLA 0.5', 'price': 3.5, 'amount': 1, ‘preferences’: ‘No Ice’}]*

It supports adding positions to the order, removing them and invoicing. After creating an invoice no further changes can be in the order.

To order food or drinks ***add\_position***runs *take\_position* and appends the returned dictionary to *order\_per\_person*. Optional are preferences that could be added to a dish. If preferences are specified, they must include ‘no’ (no paprika, no garlic etc.) or ‘extra’ (extra cheese, extra tomatoes etc.). All ‘extra’ preferences increase price of the dish by 1 € and all ‘no’ preferences do not influence the price. ***Remove\_position*** simply removes the position’s dictionary from *order\_per\_person*.

The *complete\_order* function creates a order\_id.txt file, where all the ordered items are listed together with their prices and quantities as well as the total.

**class Restaurant:** This class serves to manage tables in general. Each table is assigned with a table\_id (*table\_number*), number of seats (*seats*) and number of customers who are placed at the table (*taken*), which are respectively attributes of the class.

The function ***show\_table*** displays the information above, ***taken\_seat*** updates the number of customers at the table and ***free\_table*** resets to 0 occupied seats. ***Remove\_order*** deletes orders and ***serve*** only combines *table\_id* with each *order\_id* through ‘0’, so waiters could bring food and drinks more easily:

[301, 302, 303, 304] *orders 1, 2, 3, 4 from the table number 3.*

It appends a new extended id and returns an updated *order\_per\_table*.

Additionally there is an external list of tables (tablers), which is predefined and non-changeable within this script.

**execute.py** – console UI that controls performance of functions.

First was importing tablers and the three classes from teamwork.py

The function **execution()** handles the console UI, plain text with choices from 1-4, if anything else is typed in, the menu will appear again.

Option 1-3 correspond with the functions described below, 4 is listed as an exit out of the program.

The function **take\_seats()** asks the user to input how many guests are arriving, shows the available tables fitting for the group and asks the user to assign them a table. If the table is full, no people can be added anymore. If people are seated onto an already partially seated table, the entire table becomes a group, ordering is not an issue here.

The function **ordering()** asks the user to first input a already seated table, then takes an order. People can append and add whatever they desire. After the initial round of orders is done, it asks if any updates want to be made before finalizing the order, inputting that is also put into account. If the order ends up empty at the end, no order will be placed, if there is substance in it, the order will be converted into a .txt and saved as an invoice and an order number will be saved under that table.

The function **pay\_bill()** is the payment method. We chose to separate it as invoices can be managed and stay until payment. It asks the user to input the table and the order number to confirm who will pay. If there are any mistakes, the program asks whether the user put in the wrong thing, if they say they didn’t, they are returned to the UI, if they say they meant another table, they get to choose a table again.

Underneath all is a while loop, making the UI be able to repeat even if certain actions have been finished.

2. Coding

Implemented with Python3.11/3.12

3. Testing

In advance: This is one continuous test

A screenshot of a computer screen

Description automatically generated A screenshot of a computer program

Description automatically generated

A computer screen with white text

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

