Code Analyse

Dateienanalyse

user.py

Zweck: Repräsentiert einen Nutzer des Online-Marktplatzes.
Attribute:

- _id, _password: Login-Daten
- _name_first, _name_family: Vor- und Nachname
- _friends: Set aller Freunde (User-IDs)
- _balance: Budget in Euro (Startwert 500 €)
- _gps_coords, _address: Standortinformationen
- _rating_stars: Liste aller erhaltenen Bewertungen

Kernfunktionen:

- **Geldverwaltung:**
    - decrease_balance(amount) → Geld beim Kauf abziehen
    - increase_balance(amount) → Geld beim Verkauf hinzufügen
- **Freundschaftsverwaltung:**
    - friends_add, friends_add_list, friends_delete
    - is_friend(user_id) → prüft Freundschaft
- **Bewertungssystem:**
    - rate_user(stars) → fügt Bewertung hinzu
    - get_rating_stars_mean() → Durchschnittsbewertung berechnen
- **Darstellung & Zugriff:**
    - pretty_print() → formatiert ID, Name, Wohnort
    - Getter: name(), balance(), id(), address()

Besonderheit:

- Erbt von SetNode → später für Union-Find-Praktikumsgruppen genutzt.

praktikumsgruppen.py

SetNode:

- Repräsentiert einen Knoten im Union-Find (Disjoint-Set)
- _parent → zeigt auf Elternknoten (initial auf sich selbst)

- _weight → Größe des Teilbaums (für Union-by-Weight)

Praktikumsgruppen:

- Dictionary aller Nutzer (Praktikum 1 & 2)
- Später Union-Find-Struktur für Gruppen
- **Union-Find-Methoden (Platzhalter in P1/P2):**
    - find(node) → Wurzelknoten
    - find_byid(user_id) → Gruppe eines Nutzers
    - union(user_id1, user_id2) → Mengen zusammenführen
- Weitere Funktionen (TODO): create_groups, get_groupmembers

Kurzfazit:

- SetNode → Basis für Union-Find (P3)
- Praktikumsgruppen → speichert Nutzer, später Gruppenmanagement

users.py

Zweck: Verwalten aller User im System, erweitert Praktikumsgruppen.
Funktionen:

1. **Konstruktor:**
    - Liest user.csv → Nutzer anlegen
    - Liest friends.csv → Freundschaften eintragen
    - Praktikumsgruppen erzeugen (create_groups)
2. **Nutzerverwaltung:**
    - add(...) → neuen User anlegen
    - num_users() → Anzahl Nutzer
    - get_name_of_user(user_id) → vollständiger Name
    - password_valid(...) → Passwortprüfung
3. **Standort & Distanz:**
    - calc_distance_between_users(...) → Manhattan-Distanz (Platzhalter für späteren Graph)
4. **Darstellung & GUI:**
    - get_user_pretty_print_for_list(...) → User + Praktikumsgruppe
    - get_friends_andgroupmembers_pretty_print(...) → Freunde + Gruppenmitglieder
5. **Freundschaft & Vorschläge:**
    - get_mutual_friends(...) → Freunde von Freunden zählen

    - suggest_friends(...) → Vorschläge (TODO)
6. **Verbindungen prüfen:**
    - are_users_connected(...) → Prüft Verbindungen (TODO)
7. **CSV-Einlesefunktionen:**
    - _read_users_from_csvfile(...) → User anlegen, Gruppen speichern
    - _read_friends_csv(...) → Freundschaften eintragen

Kurzfazit:

- Zentrale Datenverwaltung für User, Freundschaften und Gruppen
- Basis für GUI-Anzeigen und spätere Algorithmen

auction.py

Zweck: Modelliert eine einzelne Auktion mit Item, Verkäufer, Bieter und Käufer.
Attribute:

- _id, _item, _seller_id, _purchaser_id
- _auction_ends → Ablaufzeit
- _recommended2users → interessierte User
- _users_bidding → Min-Heap der Gebote ((-bid, user_id))
- _bids_ordered → chronologische Gebotsliste

Wichtige Methoden:

- bid() → Bieten prüfen und platzieren
- _set_purchaser_id() → Gewinner setzen, Geld verteilen
- get_highest_bid(), get_highest_bidder() → Hilfsmethoden
- get_bid_of_user(uid) → Gebot eines Users
- expired(), sold(), sold_success() → Status prüfen

Besonderheiten:

- Min-Heap speichert negative Werte → größtes Gebot immer oben
- Automatische Entfernung aus Empfehlungen bei Gebot

auctions.py

Zweck: Verwaltung aller Auktionen, Gebote, Transaktionen und Simulation.
Hauptattribute:

- _id_next_auction, _heap, _users, _transactions, _my_simulator, _stop_event

Kernfunktionen:

1. **Auktionen:** add_new_auction, delete

2. **Gebote:** bid_in_auction, _place_random_bids
3. **Abgelaufene Auktionen:** handle_expired_auction, _set_purchaser_id
4. **Abfragen:** get_auctions_offered, get_active_auctions, get_top_auction
5. **Simulation:** _start_simulator, stop_simulation
6. **Hilfsmethoden:** _create_new_auction_id, _read_auctions_from_csvfile, sort_time_left

item.py

Zweck: Repräsentiert ein Produkt auf dem Marktplatz.
Attribute:

- _name, _description, _value_min

Methoden:

- name(), description(), value_min() → Getter

Besonderheit:

- Statische Daten → Besitzer/Käufer über Auction

systemmessages.py

Zweck: Warteschlange für Systemnachrichten in GUI.
Attribute:

- _queue, _lbl_sysmessage, _displaying

Methoden:

- push(message) → Nachricht hinzufügen, ggf. Anzeige starten
- _display_next_message() → private Methode, zeigt Nachrichten sequentiell, 3/15 Sekunden, Label zurücksetzen

Besonderheit:

- Nur eine Nachricht gleichzeitig sichtbar → automatisch ablaufend

trie.py

TrieNode:

- children → Dictionary Buchstabe → Kind
- _is_end_of_word → bool, Wortende

Trie:

- _root → Wurzel TrieNode

Methoden:

- insert(word) → fügt Wort Buchstabe für Buchstabe ein
- search(prefix) → alle Wörter mit Prefix

- _find_words(node, prefix) → rekursiv alle möglichen Wörter ab Node

Besonderheiten:

- Unterstützt schnelle Suche nach Präfixen

tack.py

Zweck: Implementiert den ADT Stack (Stapel) mithilfe einer verketteten Liste bzw. Python-List als Basis.
Klasse: Stack(list)

Kernfunktionen:

- push(item) → Element oben auf den Stapel legen
- pop() → entfernt und liefert das oberste Element (vererbt von list)
- peek() → liefert oberstes Element ohne Entfernen, None wenn leer
- is_empty() → prüft, ob Stapel leer ist
- size() → Anzahl der Elemente im Stapel

Zusatzfunktion (nicht standardmäßig laut Vorlesung):

- update(item, item_new) → ersetzt ein vorhandenes Element durch ein neues

Besonderheiten:

- Stapel ist LIFO (Last-In-First-Out)
- Basis ist Python list, daher schnelle Operationen auf Ende der Liste
- update() ist optional, dient nur für interne Manipulationen