ngChatCart

Release TBD

Kim Dodds

Oct 22, 2020

CONTENTS:

CHAPTER

ONE

README FILE

1.1 NgChat Cart

A set of python classes and methods that provide cart functionality for chat bots.

1.2 Purpose

We believe that chat and voice based e-commerce through conversational AI has a lot potential. However, when using Rasa to build a bot for ordering food and drink, we found the custom actions too restrictive. Our solution is to provide a generalized cart implementation that be integrated into any chat bot. cart.py includes classes for the cart and menu, as well as classes for items and options. Callable methods cover a wide range of capabilities, including but not limited to: setting a menu, describing menu categories/items, adding/removing items from the cart, caching user carts for multi-user support, etc.

1.3 Implementation

The main class is the cart class, and most methods can be accessed through this class. A menu must be loaded before the cart can be modified or accessed. Build the Sphinx documentation for detailed implementation details.

1.3.1 Building Sphinx documentation:

- Update the api documentation:
  - $ sphinx-apidoc -f -o sphinx/source ngchat_cart
- Run the build:
  - /sphinx$ make {build_type}
- Built files can be found in ngchat_cart/docs

1.4 Integration with Rasa

Custom actions are required to bridge the gap between Rasa and the cart. Use the custom actions to collect information from the tracker, perform checks, and dispatch utterances; call cart.py methods from a custom action to manipulate the cart, menu, or items.

1.5 Lookup table generation

Bots can use lookup tables to ensure that specific strings are consistently identified as entities. The ngChat Cart supports automatic lookup table generation from a menu. To generate lookup tables, first ensure that you have valid menu data for your bot. If it does not already exist, create a `lookup_tables` subdirectory in your bot's `data` directory. Run the `generate_lookup_tables.py` script, passing in the path to your bot's JSON menu and the `lookup_tables` directory where the output will be stored. For example:

```
python ngchat_cart/generate_lookup_tables.py bots/kfc/data/menus/KFC.json bots/kfc/
↪data/lookup_tables
```

The script will generate lookup tables for `options`, `items`, and `query_topics`.

To use these lookup tables, include references in your bot's NLU data. While these can go in any NLU data file used by the bot, we suggest keeping them in an `extra_nlu.md` file, which includes bot-specific NLU data not generated from templates. To include them, you would append the following:

```
## lookup:item
data/lookup_tables/items.txt

## lookup:options
data/lookup_tables/options.txt

## lookup:query_topics
data/lookup_tables/query_topics.txt
```

1.6 Chatette template generation

Chatette templates are used to generate initial training data for bots based on the expansion of templates marked up with entities and intents. The ngChat Cart includes a method to extract menu-specific entities into a chatette template, which can then be included in a bot's Chatette templates to easily add menu-specific entity data to your Chatette templates. These are intended to be used with the `cart.chatette` template file, which includes templates for the intents and entities used by ngChat Cart.

Generate your menu-specific chatette data with the `generate_chatette.py` script, passing in parameters to specify your menu and output filepath. For example:

```
python ngchat_cart/generate_chatette.py bots/kfc/data/menus/KFC.json bots/kfc/data/
↪generated_menu.chatette
```

If the generated file is correct, use it to replace any existing `menu.chatette` file:

```
mv bots/kfc/data/generated_menu.chatette bots/kfc/data/menu.chatette
```

To load this file into a common Chatette file, ensure that you have the `cart.chatette` file in your bot's data directory, and your `common.chatette` file begins with the following lines.

```
|cart.chatette // ngChat Cart templates
|menu.chatette // Menu-specific templates
```

1.7 CART MENU FORMAT

ngChat Cart uses a standardized menu format which is designed to be compatible with the majority of QSR and restaurant products. The menu is composed of different objects: the top level menu object, products, modifiers, modifier groups, and variants.

1.7.1 MENU

At the top level, A menu is composed of 4 attributes: name, currency, products, modifiers.

```
{
    "store_name": Text,
    "currency": Text,
    "products": List[Dict[Text, Any]],
    "modifiers": List[Dict[Text, Any]]
}
```

The store_name is the way that the name will be displayed to users of the bot. In general, the menu file itself should be the same as store_name in lower case. For example: `"store_name": "Starbucks" -> menu file == "starbucks.json"`.

The currency field is the three-letter acronym for the currency used to represent prices in the menu: ex/ `"currency": "USD"`.

The products and modifiers fields are both lists of dictionary objects. The product list contains information about purchasable items and combos (Caffe Americano, 8 Piece Family Meal, Grande Nacho Box Combo, etc). The modifiers list contains information about modifiers that can be attached to these products.

1.7.2 PRODUCTS

Products are purchasable items on the menu. A product entry has the following format:

```
{
    "name": Text,  # The item's name (ex/ Americano, Cheeseburger Meal)
    "id": Text,  # A unique identifier for the product
    "aliases": List[Text],  # A list of synonyms for this item, e.g., ["number 7", "5␣
↪dollar fill up"].
    "categories": List[Text],  # A list of category names relevant to the product (ex/
↪ Hot Drinks)
    "type": {'combo', 'item', 'virtual'},  # product type, as explained above
    "recommendation_score": int,  # A base recommendation score, used for suggesting␣
↪items to users
    "tags": List[Text],  # A list of tags (ex/ hot, sweet, drink, coffee) used for␣
↪item retrieval
    "description": Optional[Text],  # A description of the product
    "ingredients": List[Text],  # A list of ingredients in the product
    "calories": Optional[Union[int, List[int]]],  # calorie information. Can be␣
↪represented as an int, or a range [int, int]
    "price": float,  #  base item price (without modifiers)
```

(continues on next page)

(continued from previous page)

```
    "image": Optional[Text],  # a url to an image
    "modifier_groups": List[Dict[Text, Any]]  # A list of modifier groups␣
↪(customizable options)
}
```

There are three types of products: item, combo, and virtual. item:

Item

A product of type 'item' is a single purchasable item. These items can be customized with modifiers only in their modifier groups. Here's an example of an item from the Starbucks menu:

```
{
    "name": "Caffè Americano",
    "id": 776000,
    "aliases": [
        "cafe americano",
        "coffee americano"
    ]
    "categories": [
        "Hot Coffees"
    ],
    "type": "item",
    "recommendation_score": 538000,
    "tags": [
        "Hot",
        "beverages"
    ],
    "description": "Espresso shots topped with hot water create a light layer of␣
↪crema culminating in this wonderfully rich cup with depth and nuance. \n Pro Tip:␣
↪For an additional boost, ask your barista to try this with an extra shot.",
    "ingredients": [
        "Water",
        "Brewed Espresso"
    ],
    "calories": 15,
    "price": 2.95,
    "image": "https://globalassets.starbucks.com/assets/
↪f12bc8af498d45ed92c5d6f1dac64062.jpg",
    "modifier_groups": [
        {
            "name": "size",
            "category": "Sizes",
            "query_users_for_modifier": true,
            "minimum": 1,
            "maximum": 1,
            "modifiers": [
                {
                    "name": "Short",
                    "id": 663000,
                    "default": false,
                    "price_change": -0.7,
                    "calorie_change": -10
                },
                {
                    "name": "Tall",
                    "id": 663001,
```

(continues on next page)

(continued from previous page)

```
                    "default": false,
                    "price_change": -0.5,
                    "calorie_change": -5
                },
                {
                    "name": "Grande",
                    "id": 663002,
                    "default": true,
                    "price_change": 0.0,
                    "calorie_change": 0
                }
            ]
        }
    ]
}
```

Combo

A combo product is a set of items that come together; for example, a cheesebuger meal that comes with a drink and fries. In contrast to an 'item' type, a combo can have modifiers, as well as other items in it's modifier groups. In the case of a combo, you may be able to choose between several items (do you want a cheeseburger, hambuger, or veggieburger?), and each item may also be customized by its own modifiers. The crucial difference between an item and combo is that a combo will have items in it's modifier groups, and then will check if those items have customizable options. A simple item type will never check if it's modifiers have further customizations. Here's an example of a combo from the TacoBell menu:

```
{
    "name": "Mexican Pizza Combo",
    "categories": [
        "Combos"
    ],
    "recommendation_score": 0,
    "type": "combo",
    "image": "https://www.tacobell.com/images/22604_4._mexican_pizza_combo_640x650.jpg
↪",
    "description": "Served with a large taco, 2 crunchy taco supreme and a Mexican␣
↪pizza.",
    "tags": [
        "Combos"
    ],
    "calories": [
        1330,
        1330
    ],
    "price": 7.19,
    "id": "22604",
    "modifier_groups": [
        {
            "name": "Item 1: Select your drink",
            "minimum": 1,
            "maximum": 1,
            "modifiers": [
                {
                    "name": "Mountain Dew Baja Blast® Freeze - Regular",
                    "id": "1491",
                    "default": false,
```

(continues on next page)

(continued from previous page)

```
                    "price_change": 0.5
                },
                {
                    "name": "Strawberry Skittles® Freeze - Regular",
                    "id": "1615",
                    "default": false,
                    "price_change": 0.4
                }
            ]
        },
        {
            "name": "Item 3: Crunchy Taco Supreme®. Would you like to swap this for
↪something else?",
            "minimum": 0,
            "maximum": 1,
            "modifiers": [
                {
                    "name": "Soft Taco Supreme®",
                    "id": "22111",
                    "default": false,
                    "price_change": 0.0
                },
                {
                    "name": "Crunchy Taco Supreme®",
                    "id": "22101",
                    "default": true,
                    "price_change": 0.0
                },
                {
                    "name": "Nacho Cheese Doritos® Locos Tacos Supreme®",
                    "id": "22173",
                    "default": false,
                    "price_change": 0.6
                }
            ]
        }
    ]
}
```

Virtual

A virtual product is similar to a combo, but always contains an item in its modifier_group whose details and own modifier groups the bot should treat as if they were those of the main item. For example, rather than selecting a Breakfast Burrito, then choosing between bacon and sausage as addons, a user would select the virtual Breakfast Burrito, then select either "Breakfast Burrito - Bacon" or "Breakfast Burrito - Sausage" as a modifier. The bot should then treat this selection as it were the actual cart item when displaying information. Here's an example of a virtual item from the TacoBell menu:

```
{
    "name": "Power Menu Bowl",
    "categories": [
        "Specialties",
        "Power Menu"
    ],
    "recommendation_score": 0,
    "type": "virtual",
```

(continues on next page)

(continued from previous page)

```
    "image": "https://www.tacobell.com/images/22488_power_menu_bowl_640x650.jpg",
    "description": "We start with a bed of seasoned rice and top it with fire grilled␣
↪chicken, black beans, fresh pico de gallo, premium guacamole, sour cream, lettuce,␣
↪shredded cheddar cheese and avocado ranch sauce.",
    "tags": [
        "Specialties",
        "Power Menu"
    ],
    "calories": [
        470
    ],
    "price": 5.49,
    "id": "virtual-22488",
    "modifier_groups": [
        {
            "name": "Which style of Power Menu Bowl would you like?",
            "minimum": 1,
            "maximum": 1,
            "modifiers": [
                {
                    "name": "Power Menu Bowl - Chicken",
                    "id": "22488",
                    "default": true
                },
                {
                    "name": "Power Menu Bowl - Steak",
                    "id": "22489",
                    "default": false
                }
            ]
        }
    ]
}
```

And here's the entry for the 'Power Menu Bowl - Chicken':

```
{
    "name": "Power Menu Bowl - Chicken",
    "categories": [],
    "recommendation_score": 0,
    "type": "item",
    "image": "https://www.tacobell.com/images/22488_power_menu_bowl_640x650.jpg",
    "description": "We start with a bed of seasoned rice and top it with fire grilled␣
↪chicken, black beans, fresh pico de gallo, premium guacamole, sour cream, lettuce,␣
↪shredded cheddar cheese and avocado ranch sauce.",
    "tags": [],
    "calories": [
        45
    ],
    "price": 5.49,
    "id": "22488",
    "modifier_groups": [
        {
            "name": "Would you like to change the protein?",
            "minimum": 0,
            "maximum": 1,
            "modifiers": [
```

(continues on next page)

(continued from previous page)

```
                {
                    "name": "No Chicken",
                    "id": "2271",
                    "default": false,
                    "price_change": 0.0
                },
                {
                    "name": "Extra Chicken",
                    "id": "2273",
                    "default": false,
                    "price_change": 1.3
                },
                {
                    "name": "Shredded Chicken",
                    "id": "22782",
                    "default": false,
                    "price_change": 0.0
                }
            ]
        } ...
    ]
}
```

Note that the two items in its modifier groups in turn have their own distinct modifier groups. The virtual item lets us take advantage of a system already in place for combos (i.e. listing items in modifier groups) in order to set up this kind of conditionality of modifier groups without adding additional complexity. That is, it gives us an easy way to say "if you get it with chicken, here are the modifier groups for that, but if you get it with steak, here are the modifier groups for that."

1.7.3 MODIFIER GROUPS

Each product has a list of modifier groups which contain possible customizations/modifiers for the product. Modifier groups organize these modifiers into meaningful groups which have information about the group type, min and max number of selections, and whether or not the bot should ask the user about the group. Here's the format for a modifier group:

```
"name": Text,  # The name of the group (ex/ size, sauces)
"category": Text,  # A more general category for the group (ex/ toppings, sides)
"query_users_for_modifier": bool,  # whether the bot should ask the user about the␣
↪group
"minimum": int,  # minimum number of modifier selections from the group
"maximum": int,  # maximum number of modifier selections from the group (-1 = no␣
↪limit)
"modifiers": List[Dict[Text, Any]]  # A list of possible modifiers
```

1.7.4 MODIFIERS

The modifiers are the actual customization that can be applied to the product. For example: 'small', 'sour cream', or 'caramel drizzle'. Modifiers exist in two places in the menu: inside a product's modifier group, and inside the modifiers list of the menu. Modifier objects in the modifier list are the absolute representation of the modifier; they contain the concrete information about the modifier that is true no matter what item it is attached to. Here is the format for a full modifier:

```
Text: {  # modifier name as key
    "name": Text,  # modifier name (ex/ caramel drizzle, sour cream)
    "id": Text,  # a unique id for the modifier
    "type": "modifier",  # specify this object is type modifier
    "countable": bool,  # whether this modifier is countable or not (# of shots vs␣
↪size Tall)
    "quantity": Optional[int],  # The default quantity, if it is countable
    "unit_of_measure": Text,  # how the quantity is measured (ex/ shots, pumps,␣
↪packets)
    "price": float,  # price, if the modifier has a standard price
    "calories": Optional[Union[int, List]],  # calories, if the modifier has standard␣
↪calories
    "description": Optional[Text],  # optional description
    "image": Optional[Text],  # url if there is an image available
    "variants": Dict[Text, [Dict[Text, Any]]]  # A list of modifier variants
}
```

Modifiers inside a product's modifier group contain information about how the modifier is when it is attached to a specific item. The information in these modifiers can be thought of as overrides - in general all the information about a modifier can be found in the menu modifiers list, but sometimes the price or calories of a modifier changes depending on what item it is modifying. Here's the format for a modifier inside a modifier group:

```
"name": Text,  # The name of the modifier for easy access
"id": int,  # the if of the modifier (refers back to the modifier object in the menu␣
↪modifier list)
"default": bool,  # if this modifier comes with the item by default or not
"price_change": Optional[float],  # how much this option will change the base price␣
↪of the item
"calorie_change": Optional[Union[int, list[int, int]],  # how much this option will␣
↪change the base calories
"variants": List[Dict[Text, Any]]  # A list of modifier variants
```

1.7.5 VARIANTS

Variants are slightly different versions of a modifier. For example, if there is a modifier 'whipped cream', it might have the variants 'extra whipped cream', 'light whipped cream', and 'no whipped cream'. By definition these variants contradict each other, and therefor only one variant from a single modifier can be added to an item. Similar to how modifiers can override certain aspects of an item, variants can override certain attributes of their parent modifier, such as price and calories. All the possible variants for a modifier can be found in the menu's modifiers list. If a specific item can have a variant as a modifier, that variant will appear in the variants list of the modifier in the modifier group. Here is the format for variants:

```
"name": Text,  # The name of the variant
"price_change": Optional[float],  # how this modification will affect the price of␣
↪the product
"calorie_change": Optional[Union[int, list[int, int]]],  # how this modification will␣
↪affect the calories of the product
```

1.8 JSON MENU FORMAT SCHEMA OVERVIEW:

menu file naming: '/{store_name}_menu.json'

```
{
    "store_name": Text,
    "currency": Text,
    "products": [
        {
            "name": Text,
            "id": Text,
            "categories": [Text],
            "type": {'item', 'combo', 'virtual'}
            "recommendation_score": int,
            "tags": [Text],
            "description": Text,
            "ingredients": [Text],
            "calories": {int, [int, int]},
            "price": float,
            "image": Text,
            "modifier_groups": [
                {
                    "name": Text,
                    "category": Text,
                    "query_user_for_modifier": bool,
                    "minimum": int,
                    "maximum": int,
                    "modifiers": [
                        {
                            "name": Text,
                            "id": Text,
                            "price_change": float,
                            "calorie_change": int,
                            "variants": [
                                {
                                    "name": Text,
                                    "price_change": float,
                                    "calorie_change": int
                                }, ...
                            ]
                        }, ...
                    ]
                }, ...
            ]
        }, ...
    ],
    "modifiers": [
        {
            "name": Text,
            "id": Text,
            "type": "modifier",
            "countable": bool,
            "quantity": int,
            "unit_of_measure": Text,
            "price": float,
            "calories": int,
            "description": Text,
```

(continues on next page)

(continued from previous page)

```
            "image": Text,
            "variants": [
                {
                    "name": Text,
                    "price_change": float,
                    "calorie_change": int
                }, ...
            ]
        }, ...
    ]
}
```

NOTES:

- base price should be based on the default version of the item, like the standard size. Base price is summed with the price of the item's modifiers to get the total cost.

CHAPTER

TWO

NGCHAT_CART

2.1 ngchat_cart package

2.1.1 ngchat_cart.cart module

Shared cart and menu implementation.

Cart implementation including functions to: - Add item to cart - Remove item from cart - Modify item in cart - Read cart contents, total cost, and/or total calories - Empty cart - Checkout

class ngchat_cart.cart.**Calories**(*min_max: Optional[Tuple[int, int]]*)

Bases: object

Class to store, calculate, and format calories.

class ngchat_cart.cart.**Cart**(*cart_items: List[*ngchat_cart.cart.CartItem*] = None*, *active_item:* ngchat_cart.cart.CartItem *= None*, *menu:* ngchat_cart.cart.Menu *= None*)

Bases: object

Class for organizing and manipulating the user's cart.

Contains methods to add items, clear cart, add or modify options, and list the items in the cart, the total calories, or the total price.

Attributes: cart_items (List[CartItem]): A list of CartItem objects that are in the cart active_item (CartItem): currently selected item menu (Menu): The menu object for the current store

add_item(*item:* ngchat_cart.cart.CartItem, *sender_id: str = ''*) → Optional[*ngchat_cart.cart.CartItem*]

Add a single item to the cart.

Item should be a CartItem and cannot be a string-based name.

Args: item (CartItem): A cart item to add sender_id (Optional[Text]): the user's sender id, for logging purposes only

Returns: Optional[CartItem]: CartItem if added successfully, None otherwise

add_option(*item: Union[str,* ngchat_cart.cart.MenuItem*,* ngchat_cart.cart.CartItem*], option: str, mod_group:* ngchat_cart.cart.CartModifierGroup *= None*) → Optional[*ngchat_cart.cart.CartItem*]

Add an option to an existing item in the cart.

If the option already exists in item, increment *option.quantity* by 1.

Args: item (Union[Text, MenuItem, CartItem]): The item to be modified option (Text): The name of the option to be added to the item mod_group (Optional[CartModifierGroup]): The mod group to add the opt to

Return: Optional[CartItem]: modified item for success, None for failure

cart_dump() → Dict[str, Any]
Compile important cart info to dict for use in dialogue policy.

checkout() → None
Checkout all items in the cart.

clear() → None
Remove all items in the cart.

get_item_names() → List[str]
Get a list of the names of items in the cart.

Return: List[Text]: A list of the names of items in the cart

get_matching_cart_item(*item: Union[str, ngchat_cart.cart.MenuItem, ngchat_cart.cart.CartItem]*) → Optional[*ngchat_cart.cart.CartItem*]
Given an item, return the names of the closest match in the cart.

Used to verify that an item is in the cart from actions.py.

Args: item (Union[Text, MenuItem, CartItem]): The item to be matched

Returns: Optional[CartItem]: The closest item match, or None if there is no confident match

get_matching_menu(*store_name: str*) → Optional[*ngchat_cart.cart.Menu*]
Look in the menu directory and return the requested menu object.

Args: store_name (Text): The name of the store/menu

Return: Optional[Menu]: If matching confidence > *CONFIG.FUZZY_MATCH_THRESH*, return the menu object. Otherwise, None

get_matching_menu_item(*item: Union[str, ngchat_cart.cart.MenuItem, ngchat_cart.cart.CartItem]*, *keywords: List[str] = []*) → Optional[*ngchat_cart.cart.MenuItem*]
Given an item, return the name of the closest item on the menu.

Used to verify that an item is in the menu from actions.py.

Args: item (Union[Text, MenuItem, CartItem]): The name of the item to be matched

Returns: Optional[MenuItem]: The name of the closest item match, or None if there is no confident match

get_similar_menu_items(*item: Union[str, ngchat_cart.cart.MenuItem, ngchat_cart.cart.CartItem]*, *keywords: List[str] = []*, *max_results: int = 5*) → Optional[List[str]]
Given the name of an item, return a list of the closest item names on the menu.

Will return a list of item names up to max_results. Will only return results with a confidence higher than CONFIG.FUZZY_SUGGEST_THRESHOLD.

Args: item (Union[Text, MenuItem, CartItem]): The name of the item to find similar items to max_results (int): The number of results to return at most

Return: Optional[List[Text]]: A list of names to return, if there are any above CONFIG.FUZZY_SUGGEST_THRESH

list_contents() → str
Compile a readable list of items in the cart.

Return: Text: Readable list of items in the cart

modify_option(*item: Union[str, ngchat_cart.cart.MenuItem, ngchat_cart.cart.CartItem]*, *option_name: str*, *option_value: Union[ngchat_cart.cart.MenuOption, ngchat_cart.cart.CartOption, ngchat_cart.cart.CartItem, ngchat_cart.cart.MenuItem]*, *mod_group: ngchat_cart.cart.CartModifierGroup = None*, *quantity: int = None*) → Optional[*ngchat_cart.cart.CartItem*]

Modify a single item in the cart.

Args: item (Union[Text, MenuItem, CartItem]): The item to be modified option_name (Text): The name of the specific item detail to be modified option_value (MenuOption): The new desired value, an menu option object quantity (int): an optional param that will modify the quantity of the option [default=None]

Return:

Optional[CartItem]: cart item if successful, None if error, raises ValueError if item not found in cart, raises KeyError if option not found

modify_quantity(*item: Union[str, ngchat_cart.cart.MenuItem, ngchat_cart.cart.CartItem]*, *new_quant: int*) → Optional[*ngchat_cart.cart.CartItem*]

Modify the quantity of a single item in the cart.

Args: item (Union[Text, MenuItem, CartItem]): The name of the item to be modified new_quant (int): The new quantity desired

Return:

Optional[CartItem]: modified item object for success, None for failure, raises ValueError if item not found in cart.

query_item_options(*item_name: Optional[str]*) → Optional[str]

Retrieve options list for the item in the item slot or active item.

Args: item_name (Optional[Text]): The name of the item whose options to list.

Return: Optional[Text]: If found, a string listing available options, otherwise None.

query_item_suggestions(*name: str*, *keywords: List[str] = []*) → List[*ngchat_cart.cart.MenuItem*]

Query ElasticSearch for item suggestions.

Args: name (Text): String to be used to match the item name in the query. keywords (List[Text]): list of str to be used in the keywords field for ElasticSearch multimatch.

Defaults to the empty string.

Return: List[MenuItem]: A list of item suggestions, or empty list if none are found.

query_menu_category(*category_name: Optional[str]*) → List[*ngchat_cart.cart.MenuItem*]

Retrieve a list of items in a menu category.

Args: category_name (Optional[Text]): The name of the menu category to query or None.

Return: Optional[Text]: A list of items in a matching category or an empty list.

query_option_description(*item_name: Optional[str]*, *option_name: Optional[str]*) → Optional[str]

Retrieve option description for an option matching the 'query_topic' slot.

Args: item_name (Optional[Text]): The name of the item whose options to query or None. option_name (Optional[Text]): The option category to query or None.

Return: Optional[Text]: If found, an option description, otherwise None.

remove_item(*item: Union[str,* ngchat_cart.cart.MenuItem, ngchat_cart.cart.CartItem*]*) → Optional[*ngchat_cart.cart.CartItem*]

Remove a single item from the cart.

Args: item (Union[Text, MenuItem, CartItem]): The name of the item to be removed

Return: Optional[CartItem]: item object if successful, None if failure

sum_calories() → *ngchat_cart.cart.Calories*

Get the total number of calories for all the items in the cart.

Return: Calories: The total number of calories in the cart as a Calories object

sum_price() → float

Get the total cost for all the items in the cart.

Return: float: The total cost for all the items in the cart

validate_options(*item: Union[str,* ngchat_cart.cart.MenuItem, ngchat_cart.cart.CartItem*]*, *requested_options: List[str]*) → Tuple[List[str], List[str]]

Given the user's requested item and options, check if they are valid.

Args: item (Union[Text, MenuItem, CartItem]): Requested item requested_options (List[Text]): The name of requested options

Return: Tuple[List[Text], List[Text]: A list of valid options and a list of invalid options

validate_size(*item: Union[*ngchat_cart.cart.MenuItem, ngchat_cart.cart.CartItem, *str]*, *requested_size: str*) → Tuple[Optional[*ngchat_cart.cart.MenuItem*], Optional[str]]

Given the user's requested item, check if it is valid.

Args: item (Union[MenuItem, CartItem, Text]): The user's requested item

Return: Tuple[Optional[MenuItem], Optional[Text]]: (The matching item, match size)

class ngchat_cart.cart.**CartItem**(*menu_type:* ngchat_cart.cart.MenuItem, *quantity: Optional[int] = None*)

Bases: object

Particular instance of an item that will interact w/ the cart.

These objects are stored in cart.cart_items. Each objects contains a MenuItem object which contains it's basic information. Additional options can be specified in the CartItem attributes.

Attributes: menu_item (MenuItem): An MenuItem object which contains all the basic information of the item quantity (int): The number of this exact item desired modifier_groups_added (List[CartModifierGroup]): A modifier group object with only added/default options default (bool): only used if CartItem is part of a combo; whether it is a default modifier or not

property calories

Shortcut to access self.menu_item.calories.

Returns: Calories: A calorie object from the menu_item

describe_option_type(*option_type: str*) → Optional[str]

Return a string that describes all the options for an option type on an item.

Args: option (Text): The option type to be described

Return: Optional[Text]: A string listing the possible options for a specified option type or None if option isn't found

classmethod from_item_name(*name: str*, *menu:* ngchat_cart.cart.Menu) → Optional[*ngchat_cart.cart.CartItem*]

Create a CartItem object by looking up a name from the menu.

Args: name (Text): The name of the item to look up in the menu menu (Menu): The menu, passed in from the cart

Returns: Optional[CartItem]: Returns a class object if successful, or None if the item is not found in the menu

get_added_options(*include_defaults: bool = False*) → List[Union[*ngchat_cart.cart.CartOption*, *ngchat_cart.cart.CartItem*]]
Return a list of all added options from self.modifier_groups_added.

Args: include_defaults (bool): whether or not to include default modifier names

Return: List[Union[CartOption, CartItem]]: A list of modifier objects that have been added to the CartItem

get_best_available_modifier_group(*option: str*) → Optional[*ngchat_cart.cart.CartModifierGroup*]
Given an option name, return the best modifier group to add that option to.

Args: option (Text): The name of an option to add

Return:

CartModifierGroup: The first available modifier group from self.modifier_groups_added to add this option to

get_matching_option_names(*type_name: str*) → List[str]
Given the name of an option type, return the names of all matching options on the menu.

Args: type_name (Text): The name of the option type to be matched

Returns: List[Text]: The values of the closest option type match, or [] if there is no confident match

get_matching_option_type(*type_name: str*) → Optional[str]
Given the name of an option type, return the name of the closest type on the menu.

Used to verify that an option type is in the menu from actions.py.

Args: type_name (Text): The name of the option type to be matched

Returns: Optional[Text]: The name of the closest option type match, or None if there is no confident match

get_option_names(*consider_groups: bool = True*) → List[str]
Get all available option names for this item.

Args:

consider_groups (bool): whether or not to include names from groups that are already satisfied

Return: List[Text]: A list of possible option names

property name
Shortcut to access self.menu_item.name.

Returns: Text: menu item's name

sum_calories() → *ngchat_cart.cart.Calories*
Calculate the total calories of the item, including add-ons.

Return: Calories: Total calories of the item as a range

sum_price() → float
Calculate the cost of the item with its various options.

Return: float: The total price of the item as a float

to_text() → str
Convert a CartItem object to a readable string.

Return: Text: A readable string describing the item

unmet_requirements() → List[*ngchat_cart.cart.CartModifierGroup*]
Check an item for unmet slot requirements.

Return: List[Text]: A list of the names of unsatisfied required fields

class ngchat_cart.cart.**CartModifierGroup**(*menu_modifier_group:* ngchat_cart.cart.MenuModifierGroup)
Bases: *ngchat_cart.cart.MenuModifierGroup*

A class for ModifierGroups that represent modifiers added to a CartItem.

Subclass of MenuModifierGroup. CartModifierGroup is tied to a CartItem and represents all the modifiers that have been added to that item. CartModifierGroup.modifiers can contain CartOption or CartItem objects.

Note: Logic for query_users_for_modifier & minimum. cart.py will count a requirement as unmet if: 1. *query_users_for_modifier == True* regardless of the value of *minimum*, or, 2. *query_users_for_modifier == None* and *minimum > 0*.

When *query_users_for_modifier == False* and *minimum > 0*, it seems contradictory. Usually this is the case that in *modifiers* some items have *default == True* so we do not query users.

Attributes: name (Text): The name of the modifier group category (Text): The category of the modifier group query_users_for_modifier (bool): whether to ask the user about the group minimum/maximum (int): The min/max number of selected modifiers required modifiers (List[Union[CartOption, CartItem]]): A list of selected and/or default modifier objects menu_modifier_group (MenuModifierGroup): A reference to the immutable MenuModifierGroup parent object

add_option(*to_add: str*) → Optional[Union[*ngchat_cart.cart.CartOption*, *ngchat_cart.cart.CartItem*]]
Add an option to the modifier group.

Args: to_add (Text): A validated name of an option that appears in this group

Return: Optional[Union[CartOption, CartItem]]: The successfully added option, or None

initialize_default_modifiers(*modifiers: List[Union[MenuItem, MenuOption]]*) → List[Union[*ngchat_cart.cart.CartItem*, *ngchat_cart.cart.CartOption*]]
Initialize default modifiers for this CartModifierGroup object.

Given the modifiers from the MenuModifierGroups object, keep only the default modifiers and convert them to CartOption/CartItem objects.

Args: modifiers (List[Union[Menuitem, MenuOption]]): A list of modifiers from a MenuModifierGroup object

Return: List[Union[CartItem, CartOption]]: The list of modifiers, converted to Cart{Item/Option} objects

is_full() → bool
Check if the modifier group has more room to add options, or is full

Return: bool: True if the mod group is full, false if more options can be added

is_satisfied() → bool
Check if the modifier group requirements are satisfied

Return: bool: True if requirements are satisfied, else return False

class ngchat_cart.cart.**CartOption**(*menu_option:* ngchat_cart.cart.MenuOption, *quantity: Optional[int] = None*)

Bases: *ngchat_cart.cart.MenuOption*

A class for options that are added to a cart item.

Subclass of MenuOption. A version of a MenuOption that is added to a CartItem and is mutable.

Attributes: name (Text): The name of the options (ex/ whipped cream) id (Text): The unique id for the modifier countable (bool): whether the object is countable or not quantity (int): A quantity for the item unit_of_measure (Optional[Text]): Unit of measure (shots, pumps, etc) price (float): Price increase caused by adding this option to the item, None if unknown calories (Calories): Calories object containing number of calories this option adds to the item description (Optional[Text]): description, if there is one image (Optional[Text]): url to image if there is one variants (List[CartOption]): possible variants with the values 'name', 'price', 'calories' default (bool): Whether or not the option is added to an item by default

initialize_variants(*variants: List[Union[MenuItem, MenuOption]]*) → List[Union[*ngchat_cart.cart.CartItem*, *ngchat_cart.cart.CartOption*]]

Initialize CartOption variants by converting them to CartOption or CartItem objects.

Args:

variants (List[Union[MenuItem, MenuOption]]): The list of variants from the original MenuItem object that this CartItem object was built from

Return: list[Union[CartItem, CartOption]]: The list of variants, converted to Cart{Option/Item} objects

sum_price() → float

Return the total price of the CartOption.

class ngchat_cart.cart.**ESResult**(*es_tuple: Tuple[str, Union[float, int], Dict[str, Any]]*)

Bases: object

Class for results of ES search.

This gives us readable attributes for name and rank results, as well as implementing ordering methods to make sorting cleaner. Note that the @functools.total_ordering class decorator infers all ordering operations from the defined __lt__ and __eq__ methods.

The constructor takes a tuple of 3 elements: (name, rank, source), which comes from a returned ES result *r*: (r['_source']['name'], r['_score'], r['_source']). r['_source'] is stored as *ESResult.source* in case more details are needed from the original retrieval, which is what was injected to the ES index.

class ngchat_cart.cart.**ItemSearch**(*menu_data:* ngchat_cart.cart.Menu, *search_fields: List[str] = None*)

Bases: object

Support for querying the menu and returning the most similar items.

Currently ItemSearch class uses ElasticSearch's 'standard analyzer' to preprocess text fields. According to ES, this analyzer 'works well for most languages'.

generate_docs() → Iterator[Dict]

Generate doc entries to be injected into ES.

inject_index() → None

Inject menu data into an index.

Create a new ElasticSearch index for this store's menu. Loop through each item in the menu and generate a new entry for it. Currently each entry includes: item name, item option names, price, calories

rerank(*retrieved_items: List[*ngchat_cart.cart.ESResult*]*) → List[str]

Rerank the retrieved items based on popularity.

For now, just a simple rerank based on popularity score in ITEM_NAME_2_REC_SCORE. These scores are based on the number of Bing search results for each item name.

Args: retrieved_items (List[ESResult]): A list of ESResult objects w/ item name and rank

Return:

List[Text]: A list of item name suggestions, reranked by popularity then alphabetically to ensure deterministic behavior in case of a tie.

retrieve_items (*item_name: str*, *keywords: List[str]*, *n: int*) → List[*ngchat_cart.cart.ESResult*]
Retrieve top n most relevant items based on keywords.

search (*item_name: str = ''*, *keywords: List[str] = []*, *n: int = 5*, *rerank: bool = False*) → List[str]
Search for an item on the menu given the entities from the user's request.

If n is 1 and *item_name* strictly matches the returned result's name or aliases, return one result without resorting to confidence threshold. If there is a highly confident match, return one result. If there is no exact result, return n similar results if they are within

the suggestion threshold

If rerank is True, rerank similar results by popularity.

Args: item_name (Text): The item entity from the user's utterance keywords (List[Text]): A list of other entities like options, tags, categories, etc n (int): How many results to return if there is no exact match rerank (bool): to rerank similar results by popularity or not

Return: List[Text]: A list of item name results

validate_index_name (*name: str*) → str
Ensure that the name is a valid ElasticSearch index name.

This method will downcase, strip off illegal initial characters, and replace internal illegal characters with '_'. If any characters are replaced, it will log a warning.

If the name consists of the strings '..' or '.', if it exceeds 255 bytes, or if the substitutions result in an empty string, the method will raise an ValueError.

Args: name (Text): The name to be validated.

class ngchat_cart.cart.**Menu** (*name: str*, *currency: str*, *menu_items: Dict[str, ngchat_cart.cart.MenuItem]*, *menu_options: Dict[str, ngchat_cart.cart.MenuOption]*, *categories: Dict[str, List[str]]*)
Bases: object

Class for loading and organizing menus.

Attributes: name (Text): The name of the store/menu currency (Text): Currency, e.g., "USD" menu_items (Dict[Text, MenuItem]): A dictionary of item name, MenuItem object pairs. menu_options (Dict[Text, MenuOption]): A dictionary of option name, MenuOption object pairs categories (Dict[Text, List[Text]]): A dict of category name, list of items in that category

check_typing (*menu: Dict[str, Any]*) → None
Check the typing of the json menu being loaded in and raise errors.

describe_category (*category: str*) → str
Given a particular category, list the possible items.

This function will fuzzy match the *category*.

Args: category (Text): The category that the user is interested in

Return:

Text: A description of the items available for that category. Returns empty string "" if the category is not found.

describe_menu() → str
Describe the menu in general terms, including what categories of items are available.

TODO: in the future, here would be a good place to mention popular or recommended items.

Return: Text: A readable description of the menu

classmethod from_json(*menu_path: str*) → Tuple[str, str, Dict[Any, *ngchat_cart.cart.MenuItem*], Dict[Any, *ngchat_cart.cart.MenuOption*], collections.defaultdict]
Load a menu from a json file into a Menu object.

Args: menu_path (Text): The file path to a json menu

Return: Any: a dictionary of items and a dictionary of categories, or none

generate_chatette(*item_output: Optional[str] = 'item'*, *options_output: Optional[str] = 'options'*, *categories_output: Optional[str] = 'query_topic'*, *indent: int = 4*) → str
Generate *menu.chatette* template with items and options.

menu.chatette is mostly handcrafted, so it is better write this output to *menu-generated.chatette*.

Args: item_output (Optional[Text]): output item entity name. Defaults to "item". options_output (Optional[Text]): output option entity name. Defaults to "options". categories_output (Optional[Text]): output category name. Defaults to "query_topic". indent (int, optional): number of spaces to indent. Defaults to 4.

Returns: Text: a "n" joined string that can directly be written to *menu-generated.chatette*

get_aliases_list() → List[Optional[List[str]]]
Return a list of list for all aliases including the item name itself.

Returns: List[Optional[List[Text]]]: a list of synonym list

get_lookup_tables(*entity_attr_map: Dict[str, List[str]]*) → Dict[str, List[str]]
Generate entity lookup tables for entities whose values are populated from the menu.

Args: entity_attr_map (Dict[Text, List[Text]): Map entity names to menu attributes.

Returns: Dict[Text, List[Text]]: A dict mapping entity names to lookup lists.

We may also want to refine the substitutions done here, and/or allow for some kind of query expansion to include multiple variations.

get_matching_menu_category(*category: str*) → Optional[str]
Given the name of a category, return the name of the closest cat. on the menu.

Used to verify that a category is in the menu from actions.py.

Args: category (Text): The name of the category to be matched

Returns: Optional[Text]: The name of the closest category match, or None if there is no confident match

get_matching_menu_item(*item: Union[str,* ngchat_cart.cart.MenuItem*,* ngchat_cart.cart.CartItem*]*, *keywords: List[str] = []*) → Optional[*ngchat_cart.cart.MenuItem*]
Given the name of an item, return the name of the closest item on the menu.

Used to verify that an item is in the menu from actions.py.

Args: item (Union[Text, MenuItem, CartItem]): The name of the item to be matched

Returns: Optional[MenuItem]: The name of the closest item match, or None if there is no confident match

get_matching_option_type(*type_name: str*, *item: Union[str, ngchat_cart.cart.CartItem, ngchat_cart.cart.MenuItem]*) → Optional[str]

Given the name of an option type, return the name of the closest type on the menu.

Used to verify that an option type is in the menu from actions.py.

Args: type_name (Text): The name of the option type to be matched item (Union[Text, CartItem, MenuItem]): item whose option types to search

Returns: Optional[Text]: The name of the closest option type match, or None if there is no confident match

get_similar_menu_items(*item: Union[str, ngchat_cart.cart.MenuItem, ngchat_cart.cart.CartItem]*, *max_results: int = 5*, *keywords: List[str] = []*) → Optional[List[str]]

Given the name of an item, return a list of the closest item names on the menu.

Will return a list of item names up to max_results. Will only return results with a confidence higher than CONFIG.FUZZY_SUGGEST_THRESHOLD.

Args: item (Union[Text, MenuItem, CartItem]): The name of the item to find similar items to max_results (int): The number of results to return at most

Return: Optional[List[Text]]: A list of names to return, if there are any above CONFIG.FUZZY_SUGGEST_THRESH

classmethod load_menu(*menu_path: str*, *name: str = ''*) → Optional[Menu]

Given a store name and a path to a directory of menus, validate the menu selection.

Args: name (Text): The name of a store/menu menu_path (Text): The path to the directory of shared menus or the menu itself

Raises: ValueError: if parameters are not passed with the proper information

Return: Optional["Menu"]: cls object if successful, None if failure

query_item_options(*item_name: Optional[str]*, *active_menu_item: Optional[ngchat_cart.cart.MenuItem] = None*) → Optional[str]

Retrieve options list for the item in the item slot or active item.

Args: item_name (Optional[Text]): The name of the item whose options to list. active_item (Optional[MenuItem]): The MenuItem corresponding the cart's

current active item, if any.

Return: Optional[Text]: If found, a string listing available options, otherwise None.

query_item_suggestions(*name: str*, *keywords: List[str] = []*) → List[*ngchat_cart.cart.MenuItem*]

Query ElasticSearch for item suggestions.

Args: name (Text): String to be used to match the item name in the query. Keywords (List[Text]): List of str to be used in the keywords field for ItemSearch.

Defaults to the empty string.

Return: List[MenuItem]: A list of item suggestions or empty list if none are found.

query_menu_category(*category_name: Optional[str]*) → List[*ngchat_cart.cart.MenuItem*]

Retrieve a list of items in a menu category.

Args: category_name (Optional[Text]): The name of the menu category to query or None.

Return: List[MenuItem]: The list of items in a menu category.

query_option_description(*item_name: Optional[str]*, *option_name: Optional[str]*, *active_menu_item: Optional[*ngchat_cart.cart.MenuItem*] = None*) → Optional[str]

Retrieve option description for an option matching the 'query_topic' slot.

Args: item_name (Optional[Text]): The name of the item whose options to query or None. option_name (Optional[Text]): The option category to query or None. active_item (Optional[MenuItem]): The MenuItem corresponding the cart's

current active item, if any.

Return: Optional[Text]: If found, an option description, otherwise None.

class ngchat_cart.cart.**MenuItem**(*name: str*, *id: Union[str, int]*, *categories: List[str]*, *rec_score: int*, *tags: List[str]*, *description: Optional[str]*, *ingredients: Optional[List[str]]*, *calories: Optional[Tuple[int, int]]*, *price: Optional[float]*, *image: Optional[str]*, *modifier_groups: List[*ngchat_cart.cart.MenuModifierGroup*]*, *item_type: str*, *quantity: Optional[int]*, *aliases: Set[str]*, **args: Any*, ***kwargs: Any*)

Bases: object

Class for loading and organizing item data.

These class objects are used to store menu data. Additionally, each CartItem object contains an MenuItem object for reference. MenuItem comes with a list of modifier groups, and each group has a list of possible modifiers. A MenuItem object is created based on a json menu with the following format:

"name": "Caffè Americano", "aliases": [# from the KFC menu

"$5 fill ups", "5 dollar fill ups", "five dollar fill ups", "number 15", "number fifteen"

], "id": "776000", "categories": [

"Hot Coffees"

], "type": "item", "recommendation_score": "538000", "tags": [

"Hot", "beverages"

], "description": "Espresso shots topped with hot water.", "ingredients": [

"Water", "Brewed Espresso"

], "calories": [15, 15], "price": 2.95, "image": "https://globalassets.starbucks.com/assets/f12bc8af498d45ed92c5d6f1dac64062.jpg", "modifier_groups": [

{ "name": "size", "category": "Sizes", "query_users_for_modifier": true, "minimum": 1, "maximum": 1, "modifiers": [

{ "name": "Short", "id": 663000, "default": false, "price_change": null, "calorie_change": -10

}, {

"name": "Tall", "id": 663001, "default": false, "price_change": null, "calorie_change": -5

}

]

}, {

“name”: “Creamer”, “category”: “Add-ins”, “query_user_for_modifiers”: false, “minimum”: 0, “maximum”: -1, “modifiers”: [

{ “name”: “Splash of 2% Milk”, “id”: 663004, “default”: false, “price_change”: null, “calorie_change”: null, “variants”: [

{ “name”: “Extra Splash of 2% Milk”, “price_change”: null, “calorie_change”: null

}

]

}

}

}

}

Internally we maintain *option_type2name* (e.g., “MILK CHOICES” -> [“1% Milk”, “2% Milk”])

Attributes: name (Text): The name of the item id (Text): A unique id for the product categories (List[Text]): A list of categories for this item rec_score (int): A popularity/recommendation score for the item tags (List[Text]): A list of tags used for searching relevant items description (Text): A description of the item from the menu ingredients (Text): Ingredients of the item from the menu calories (Calories): Calories object containing the basic calories of the item w/out modifications price (float): The basic cost of the item without modifications/add-ons image (Text): The image URL of the item modifier_groups (List[MenuModifierGroup]): A modifier group object containing possible options for item

apply_item_overrides(*item_details: Dict[str, Any]*, *is_variant: bool = False*) → None

Apply item specific information to a MenuOption object.

Args:

item_details (Dict[Text, Any]): A dict of information about how the option changes when it is applied to a specific item.

is_variant (bool): whether or not the override is a variant (will override name)

check_typing(*name: str*, *data: Dict[str, Any]*) → None

Check typing of all incoming data loaded from the menu.

Args: name (Text): The name of the item data (Dict[Text, Any]): The item data directly from the menu

describe_option_type(*option_type: str*) → Optional[str]

Return a string that describes all the options for an option type on an item.

Args: option (Text): The option type to be described

Return: Optional[Text]: A string listing the possible options for a specified option type or None if option isn’t found

classmethod from_data(*name: str*, *data: Dict[str, Any]*, *options: Dict[str,* ngchat_cart.cart.MenuOption*]*) → Optional[*ngchat_cart.cart.MenuItem*]

Restructure a dict of item info from json as an MenuItem.

Args: name (Text): the name of the item data (Dict[Text, Any]): The data for one item from a menu options (Dict[Text, MenuOption]): menu.menu_options used for creating modifier_group object

Return: MenuItem: A filled MenuItem object, or None

get_matching_item_options(*option: str*, *opt_type: str = None*) → List[Union[*ngchat_cart.cart.MenuItem*, *ngchat_cart.cart.MenuOption*]]

Given the name of an option, return the name of the closest option available.

Used to verify that an option is available for an item from actions.py. e.g., option=1%, opt_type=milk; option=small, opt_type=size. "options" can be either MenuOptions or another MenuItem. Return just the name of the option.

Args: option (Text): The name of the option to be matched opt_type (Optional[Text]): Optional, if you want to specify a particular option type

Returns: List[MenuOption, MenuItem]]: A list of matching option objects

get_matching_item_options_groups(*option: str*, *opt_type: str = None*) → List[Tuple[Union[*ngchat_cart.cart.MenuOption*, *ngchat_cart.cart.MenuItem*], str, Optional[str]]]

Given the name of an option, return a list of tuples containing group and variant info.

Return a list of tuples containing (An item option object from a specific mod group, The name of the modifier group, The parent variant name (if there is one))

Args: option (Text): The name of the option to be matched opt_type (Optional[Text]): Optional, if you want to specify a particular option type

Returns:

List[Tuple(MenuOption, Text, Optional[Text])]: list containing tuples of closest option match object, relevant modifier group name, and variant parent name

get_matching_option_names(*type_name: str*) → List[str]

Given the name of an option type, return the names of all matching options on the menu.

Args: type_name (Text): The name of the option type to be matched

Returns: List[Text]: The values of the closest option type match, or [] if there is no confident match

get_matching_option_type(*type_name: str*) → Optional[str]

Given the name of an option type, return the name of the closest type on the menu.

Used to verify that an option type is in the menu from actions.py.

Args: type_name (Text): The name of the option type to be matched

Returns: Optional[Text]: The name of the closest option type match, or None if there is no confident match

get_option_names(*include_variants: bool = False*) → List[str]

Return a list of all the item's possible options.

"options" can be either MenuOptions or another MenuItem. Return just the name of the option.

Args: include_variants (bool): whether or not to include variant names

Return: List[Text]: A list of all option names in the MenuItem's mod groups

get_option_objects(*option_type: str*) → List[Union[*ngchat_cart.cart.MenuOption*, *ngchat_cart.cart.MenuItem*]]

Given a type of option, return all MenuOption objects of that type.

Args: option_type (Text): The name of the option type (size, topping, etc)

Return:

List[Union[MenuOption, MenuItem]]: A list of MenuOption and/or MenuItem objects, or none if not a valid category

list_option_types() → str
Return a string that describes all the option types available for an item.

Return: Text: A string listing the item's option types.

set_mod_groups(*data: Dict[str, Any]*, *options: Dict[str,* ngchat_cart.cart.MenuOption*]*, *items: Dict[str, MenuItem]*) → None
Finish initialization of modifier_groups.

Because modifier_groups can include references to items, we have to wait until the items list is initialized before going back and calling this method on each instance to set the modifier_groups.

Args: data (Dict[Text, Any]): modifier group data from the menu options (Dict[Text, MenuOption]): Menu.menu_options items (Dict[Text, MenuItem]): Menu.menu_items

sum_calories() → *ngchat_cart.cart.Calories*
Return total calories of this MenuItem.

sum_price() → float
Return the total price of this MenuItem.

class ngchat_cart.cart.**MenuModifierGroup**(*name: str*, *category: Optional[str]*, *query_users_for_modifier: Optional[bool]*, *minimum: int*, *maximum: int*, *modifiers: List[Union[*ngchat_cart.cart.MenuOption*, MenuItem]]*)
Bases: object

Class to store MenuOption and MenuItem modifiers in meaningful groups.
Args: name (Text): Modifier group name such as "Select a Size", "Creamer", "Powders" category (Optional[Text]): Category of this modifier, by default intialized to be the same as *name*. query_users_for_modifier (Optional[bool]): explicitely ask users to choose if set True minimum (int): required minimum maximum (int): required maximum, if -1, no maximum modifiers (List[Dict[Text, Any]]): a list of modifiers

check_typing(*group: Dict[str, Any]*) → None
Check the typing of incoming group data and raise errors if incompatible.

Args: group (Dict[Text, Any]): The group data from the menu

classmethod from_data(*group: Dict[str, Any]*, *options: Dict[str,* ngchat_cart.cart.MenuOption*]*, *items: Dict[str, Any]*) → *ngchat_cart.cart.MenuModifierGroup*
Create modifier group object from menu data.

Args: group (Dict[Text, Any]): Group information from the json menu options (Dict[Text, MenuOption]): Menu.menu_options items (Dict[Text, Any]): Menu.menu_items

get_option_names(*include_variants: bool = False*) → List[str]
Return the names of all possible options in this group.

Args: include_variants (bool): If the result should include variant names or not

Return: List[Text]: A list of option names in the modifier group

is_optional() → bool
Determine whether this modifier group is optional.

This is determined by judging whether *minimum == 0*. Note that if *query_users_for_modifiers == True*, even if this modifier group is optional, we still need to query users for this option. For instance, in KFC menu, "would you like to remove any condiments for a chicken sandwich": 1. No Mayo 2. No Pickle. This is optional but we still ask.

Returns: bool: True or False.

prepend_dummy_modifier(*prevent_duplicates=True*) → None
Prepend a dummy modifier to the front of modifier list.

Why prepend to *front*: so that this option is shown first.

class ngchat_cart.cart.**MenuOption**(*name: str*, *id: Union[str, int]*, *countable: bool*, *quantity: Optional[int]*, *unit_of_measure: Optional[str]*, *price: Optional[float]*, *calories: Optional[Tuple[int, int]]*, *description: Optional[str]*, *image: Optional[str]*, *variants: Dict[str, Any]*, *default: bool = None*)
Bases: object

Class for item options.

Attributes: name (Text): The name of the options (ex/ whipped cream) id (Text): The unique id for the modifier countable (bool): whether the object is countable or not quantity (int): A default quantity for the item unit_of_measure (Optional[Text]): Unit of measure (shots, pumps, etc) price (float): Price increase caused by adding this option to the item calories (Calories): Calories object containing number of calories this option adds to the item description (Optional[Text]): description, if there is one image (Optional[Text]): url to image if there is one variants (List[MenuOption]): possible variants with the values 'name', 'price', 'calories'

apply_item_overrides(*item_details: Dict[str, Any]*, *is_variant: bool = False*) → None
Apply item specific information to a MenuOption object.

Args:

item_details (Dict[Text, Any]): A dict of information about how the option changes when it is applied to a specific item.

is_variant (bool): whether the override is variant or not (this will include overriding the name)

classmethod from_data(*name: str*, *data: Dict[str, Any]*) → *ngchat_cart.cart.MenuOption*
Create an MenuOption class from menu data.

Args: name (Text): The name of the option data (Dict[Text, Text]): A dictionary containing info about the option

Returns: MenuOption: An object containing the passed-in data

sum_calories() → *ngchat_cart.cart.Calories*
Sum and return Calories object.

sum_price() → float
Sum and return price as a float.

ngchat_cart.cart.**format_calories**(*calories: Any*) → Optional[Tuple[int, int]]
Format calories given raw menu data.

Args:

calories (Any): Calorie data directly from the menu. Expecting a single float/int or a list with 1-2 floats/ints

Return:

Optional[Tuple[int, int]]: if the input makes sense, return a tuple containing the calorie min/max. Otherwise None

ngchat_cart.cart.**get_user_cart**(*sender_id: str*) → Cart
Grab a cart object based on sender ID.

Args: sender_id (Text): tracker.sender_id, unique id for user

Returns: Cart: The user's cart object (new if user doesn't have one)

ngchat_cart.cart.**list_to_string**(*to_string: List*, *join: str = 'and'*, *exclude: List[str] = []*) → str
Take an iterable and return a readable list.

Args: to_string (List[Text]): A list of strings join (Text): An optional parameter that determines what the final item should be preceded by exclude (List[Text]): A list of values to exclude from the final string
Return: Text: A readable list as a string

ngchat_cart.cart.**load_config**(*path: str*) → None
Load cart config file.

ngchat_cart.cart.**load_menus**(*path: List[str] = []*) → None
Load all menus in specified in MENU_PATHS to MENU global dict.

ngchat_cart.cart.**price_change_to_text**(*price: float*, *currency: str = 'USD'*) → str
Convert price change to currency-aware text with "+"/"-" to indicate change.
Args: price (float): price number currency (Text, optional): Short code for currency: USD/CAD/EUR/RMB/JPY. Defaults to "USD".
Returns: Text: e.g., "+$3.10", "-$0.30"

ngchat_cart.cart.**price_to_text**(*price: float*, *currency: str = 'USD'*) → str
Convert price to currency-aware text.
Args: price (float): price number currency (Text, optional): Short code for currency: USD/CAD/EUR/RMB/JPY. Defaults to "USD".
Raises: ValueError: if currency is not in USD/CAD/EUR/RMB/JPY.
Returns: Text: e.g., "$3.10", returns "" if price is None

2.1.2 ngchat_cart.generate_chatette module

Use ngchat_cart to generate menu.chatette

ngchat_cart.generate_chatette.**write_menu_chatette**(*menu_fpath: str*, *save_fpath: str*) → None
Generate menu.chatette template
Args: menu_fpath (Text, optional): Menu filepath. save_fpath (Text, optional): Generated .chatatte file path.

2.1.3 ngchat_cart.generate_lookup_tables module

Use ngchat_cart to generate bot entity lookup tables

ngchat_cart.generate_lookup_tables.**write_menu_chatette**(*menu_fpath: str*, *output_dir: str*) → None
Generate entity lookup tables for options, items, and query_topics.
Args: menu_fpath (Text): Menu filepath. output_dir (Text): Template output directory.

2.1.4 Module contents

Docstring.

PYTHON MODULE INDEX

n

INDEX