Neolib2 Documentation

Release 0.1.1

Joshua Gilman

December 17, 2014

Base:

CHAPTER 1

NeolibBase

HTTP:

CHAPTER 2

Page

CHAPTER 3

HTMLForm

Inventory:

CHAPTER 4

Inventory

CHAPTER 5

MSInventory

CHAPTER 6

USBackInventory

CHAPTER 7

UserInventory

CHAPTER 8

USFrontInventory

Item:

CHAPTER 9

InventoryItem

CHAPTER 10

InventoryItemList

CHAPTER 11

Item

CHAPTER 12

ItemList

CHAPTER 13

MSItem

CHAPTER 14

MSItemList

CHAPTER 15

USBackItem

CHAPTER 16

USBackItemList

CHAPTER 17

USFrontItem

CHAPTER 18

USFrontItemList

CHAPTER 19

WizardItem

CHAPTER 20

WizardItemList

Registration:

CHAPTER 21

RegisterUser

Shop:

CHAPTER 22

History

CHAPTER 23

MainShop

CHAPTER 24

Transaction

class `neolib.shop.Transaction.`**`Transaction`**

Represents a transaction that occurred in a user's sales history

Attributes: date: The date the transactions occurred **item**: The item that was purchased **buyer**: The user who bought the item **price**: The price the item was sold at (integer)

CHAPTER 25

UserBackShop

CHAPTER 26

UserFrontShop

CHAPTER 27

Wizard

User:

CHAPTER 28

Bank

Neopet

class `neolib.user.Neopet.`**`Neopet`**

Represents a Neopet

Attributes:

name: The neopet's name
gender: The neopet's gender
species: The neopet's species
age: The neopet's age
level: The neopet's level

health: The neopet's current health
mood: The neopet's current mood
hunger: The neopet's current hunger level

CHAPTER 30

Profile

CHAPTER 31

User

Hooks:

CHAPTER 32

Hook

CHAPTER 33

UserDetails

CHAPTER 34

For developers

34.1 Neolib 2 Development Primer

This primer is intended on giving an introductory overview on how to assist development of Neolib 2. This primer will cover the basic principles being used in Neolib 2 as well as general recommendations when making additions and/or modifications.

34.1.1 1. The NeolibBase class

The `NeolibBase` class is the base class that any class doing work on a user's account should inherit. To better explain: Any class that will be using a `User` instance to access Neopets and/or perform an action on Neopets on a user's behalf.

The base class provides developers access to a group of functions that solve common problems which arise when automating Neopets. It also sets a common pattern from which the rest of the library expands upon. There are key things to note:

1. All xpath queries should be stored in the *_paths* attribute. Each path should have an easily identified name attached to it in a dictionary format. It is possible to have several layers of nested dictionaries in this attribute, however this should only be done if it improves readability.

Example:

```
_paths = {
    'inventory': {
        'item': 'query'
    }
}
```

2. When querying a page with an xpath query, the internal *_xpath()* function should be used. This function takes the name or path to an xpath string query and applies it to the *HTMLElement* or *Page* object passed to it. It returns the results just as the standard *HTMLElement.xpath()* function would.

Example:

```
result = self._xpath('inventory/item', pg)
```

3. Note #1 applies to regular expression queries as well. They should be stored in the *_regex* attribute and can be formatted just like xpaths.

4. Note #2 applies to querying with regular expressions as well. The internal *_search()* function should be used for querying a string, *HTMLElement*, or *Page*. The results are returned in a list as if *re.findall()* had been used.

5. Note #1 applies to urls as well. All urls should be stored in the *_urls* attribute and can be formatted just like xpaths.

6. Note #2 applies to requesting urls as well. The internal *_get_page* method should be used for requesting configured urls with the configured `User` instance.

7. The internal `User` instance supplied in the initialization of the base class and copied to the *_usr* attribute should be used for all transactions that require a `User` instance.

8. All child classes of the base class should override the *_log_name* attribute to something unique and meaningful to that particular class. This attribute is appended to log entries and helps in identifying where the entry originated from.

34.1.2 2. Parsing Tips

Neolib 2 uses a combination of xpath and regular expressions to parse content from Neopets HTML pages. The xpath and regular expression queries are stored and use as described in section 1 of this primer. When writing a class that needs to parse content take the following guidelines into consideration:

1. As far as Neolib 2 is concerned, xpath is preferred over regular expressions. If all content can be parsed with xpath, this would be the preferred method rather than combining the two.

2. It is not uncommon to come across such poorly formatted HTML on Neopets that using xpath purely is simply not ideal. In this case xpath should be used to reach as far into the document as feasible and the remaining information should be extracted from the resulting *HTMLElement* using regular expressions.

3. Numbers are not expected to be completely uniform across the library, however the following should be taken into consideration when determining if a value should be stored as an integer or string: If the label of the attribute is overwhelmingly clear that it will contain an integer (I.E *neopoints*) then store it as an integer, otherwise defer to a string.

34.1.3 3. Accessibility

Neolib 2 strives to have a consistent interface for an end user trying to create an automation script. The library prefers a natural and uniform way for accessing underlying API's. The main method Neolib 2 does this is by objectifying everything and creating intelligent methods and attributes that a person would intuitively expect to be in place (for instance, accessing a user's inventory by simply typing *usr.inventory*). That being said, when making new additions to the library, take the following into account:

1. When available, always use a base class for inheriting common methods and attributes. For instance, if you're creating an interface for a type of inventory, ensure it inherits the `Inventory` class properly.

2. If the interface you're creating is directly related to a user, then ensure to create an accessible attribute inside the `User` class for accessing your interface. For instance, it would be ideal to place an attribute to access a Bank interface inside of the `User` class because every user has a bank.

3. Prefer properties over attributes if the value being accessed needs to be loaded into memory before becoming useful. For instance, when first initializing a `User` instance all of the attributes default to none. This can be confusing to the end-user if they don't know to call individual load() functions to propogate the attributes with data. Therefore, it has been determined the best way to approach this scenario is to pre-load the data in a property definition if it hasn't already been loaded.

4. Always make an attempt to document your code prior to making a commit to the master branch. Remember that document strings are for the end user and hashtag comments inside of the code are for helping developers understand your logic.

34.1.4 4. Conclusion

The most important thing to do before contributing to Neolib 2 is to thoroughly review the existing code. While doing so it's important to mentally note the structure being used and the common approaches being taken to retrieving and storing data. If you have questions or concerns please feel free to open up an issue and an appropriate developer will assist in answering it. Now go forth and conquer Neopets!

CHAPTER 35

Indices and tables

- *genindex*
- *modindex*
- *search*

Index