Table of Contents › Module 3: Container Types & Advanced Data Structures › Week 7 › Tutorial: Build a Vending Machine using OOP

Tutorial: Build a Vending Machine using OOP

Starter code:

https://classroom.github.com/a/vOWeVjpQ (https://classroom.github.com/a/vOWeVjpQ)

Instructions below are copied from the repo's README.

Vending machine – Take 2: Using OOP

This tutorial revisits a previous example and uses object-oriented programming paradigms to represent the concepts related to a vending machine. We will see how classes and objects will help us better represent the "real world" and help us track state:

Design

Let's begin by designing our vending machine by modelling different objects. First, we have:

- Coins
- Products
- Vending Machine

We can consider these as abstract classes / concepts. In practise, we have specific coins like quarters, loonies & toonies. For products, we have chips, candy and drinks – these could be broken down further into specific products like "355ml Coca-Cola can". These are considered concrete classes which all share common properties to the abstracts coins and products.

We must also model our vending machine and define actions on it:

- `+ insert_coin(coin: Coin)`
- `+ buy_product(product: type) -> Product`
- `+ get_balance() -> int`
- `+ get_change() -> List[Coin]`

Models

Coins

Coin

```
value: int
```

```
label: str
str() -> str
```

The following classes inherit from the parent `Coin` class. Inheritance is achieved by providing parentheses around the class definition and providing the parent class(es) as arguments.

Nickel(Coin)

value	`5`
`str()`	`'5¢'`

Dime(Coin)

value	`10`
`str()`	`'10¢'`

Quarter(Coin)

value	`25`
`str()`	`'25¢'`

Loonie(Coin)

value	`100`
`str()`	`'$1'`

Toonie(Coin)

value	`200`
`str()`	`'$2'`

Products

Product

```
name: str
price: int
str() -> str
```

Chips(Product)

`name`	`'Chips'`
`price`	`225`
`str() -> str`	`Chips: $2.25`

Drink(Product)

`name`	`'Drink'`
`price`	`275`
`str() -> str`	`Drink: $2.75`

Candy(Product)

`name`	`'Candy'`
`price`	`315`
`str()`	`Candy: $3.15`

Vending Machine

VendingMachine

```
coins: List[Coin]
purchases: List[Product]
insert_coin(coin: Coin)
buy_product(product: type) -> Product
get_balance() -> int
get_change() -> List[Coin]
```

.insert_coin(coin: Coin)

- The `coin` parameter will accept any instance of the `Coin` class
- When the function insert_coin() is called, store the inserted coin a list on the object

.buy_product(product: str) -> Product

- The product argument may be one of the following types: Drink, Candy, or Chips, but be flexible enough to accept other Product types. Any other value should raise a ValueError exception. Note: these are the Product classes we've defined, and **not their instances**. This is the same difference as these are the Product classes we've defined, and **not their instances**. This is the same difference as the type int and the instance int()

E.g.

```
machine = VendingMachine()
machine.buy_product(products.Drink) # Good. This is a type
machine.buy_product(products.Drink()) # Bad (for the purpose of this function)
```

- If the vending machine balance is less than the cost of the product, a custom exception called `InsufficientFunds` should be raised.

- Upon successful purchase, an instance of the product should be returned, and, the purchase should be added to a list of purchases on the object.

.get_balance() -> int

- The get_balance function should return the sum of inserted coins minus the sum of the price of purchased products.

- Scenario: when no coins are inserted, and no purchases have been made, the balance should be zero.

- Scenario: given that two toonies are inserted into the machine and a candy bar was purchased, the method should return 75.

.get_change() -> List[Coin]

- If the vending machine balance is greater than zero, return a list of coins (can be any combination of 5, 10, 25, 100, 200) which will sum up to the balance.

- If for whatever reason the balance is not a multiple of 5, then the sum of coins returned should be rounded down to the nearest multiple of 5, and not exceed the balance.

- The coins returned should be the largest first, then the smallest.

- The list of inserted coins & purchased products should be cleared (get_balance should be zero)

- Scenarios:
  - When the balance is 0, no quarters should be returned
  - When the balance is 25, a quarter should be returned
  - When the balance is 400, two toonies should be returned
  - When the balance is 300, a toonie and loonie should be returned
  - When the balance is 265, a toonie, two quarters, a dime and a nickel should be returned
  - When the balance is 7, a nickel should be returned
  - When the balance is negative, nothing should be returned

Solving

- Begin by building the Coin classes, followed by the Product. Starter tests have already been provided for these classes.
- Start writing tests for the VendingMachine methods, in the order defined above.

Solution

https://github.com/sheridan-python/tutorial-oop-vending-machine-solution/
(https://github.com/sheridan-python/tutorial-oop-vending-machine-solution/commits/master)

Download Print ‹ ›

Activity Details

You have viewed this topic

Last Visited Oct 16, 2020 4:21 PM