Solutions_Exercises+(OOP+with+Python)_v3.md

Solutions for exercises (OOP with Python)

Exercise 1: Creating class, attributes, methods with Online Shopping Cart

Create a class called *Shopping*

1. Each *Shopping* must contain the following attributes:
   - **name** which indicates the name of the user example: Shopping("Rita")
   - **cart** which indicates the list of goods selected..
   - **balance** which indicates the total cost of the goods selected. Default value is 0.0.
   - **price_list** which indicates the list of the prices of the items selected in the same order of item selection

2. Each *Shopping* must contain the following methods:
   - **select** which takes in the arguments: *item* indicating item (default value None) selected and *price* (default value None) indicating price of the good selected. Also, put the option to select items in a list with their prices like a dictionary. Put in an error check such that price is always positive and your code throws an error if user calls the method with a negative price. It should add the price to the price_list and add the price value to **balance** attribute and the item name to the list attribute **cart** . Finally it should return the new **cart** attribute.
   - **check_balance** which returns a print statement as follows: "Your current balance is {self.balance} Eur"
   - **remove_item** which takes the attributes: *item* indicating item(default value: None). It should subtract the price of the selected item from the **balance** attribute and delete the item name from the list attribute **cart** along with its price from the price list. Again put the option so that user can remove several items in a list. Put in an error check such that the name belongs to the list of items in cart and your code throws an error if user wants to remove an item which does not exist in the cart. Finally it should return the new **cart** attribute.

```
class Shopping:
    def __init__(self,name, cart=[],price_list=[], balance=0.0):
        self.cart = cart
        if not self.cart:
            self.cart=[]
        self.name = name
        self.price_list=price_list
        self.balance = balance

    def add_item (self, obj, price):
        self.cart.append(obj)
        self.price_list.append(price)
        self.balance+=price

    def select(self,item=None, price=None, **kwargs):
        if item is not None and price is not None:
            self.add_item(item, price)
        for key,value in kwargs.items():
            if value<0:
                 raise ValueError(" Price cannot be negative!")
            else:
                self.add_item(key, value)
    def check_balance(self):
        return (f'Your current balance is {self.balance} Eur.')

    def del_item(self,obj):
        ind=self.cart.index(obj)
        price=self.price_list[ind]
```

```
            self.price_list.remove(price)
            self.balance -=price
            self.cart.remove(obj)

        def remove_item(self,item=None, *args):
            if item is not None:
                self.del_item(item)
            for obj in args:
                if obj not in self.cart:
                    raise ValueError(f"{obj} does not exist in cart!")
                else:
                    self.del_item(obj)
            return self.cart

buyer=Shopping('Ben')
#kwarg ={"jeans"= 50}

buyer.select('lemons', 5)
kwargs={"Jeans": 50, "oranges": 5}
buyer.select(**kwargs)
print(buyer.cart)
print(buyer.price_list)
buyer.select("jeans2", 50)
print(buyer.cart)
print(buyer.price_list)
print(buyer.balance)

args =['oranges', 'jeans2']
buyer.remove_item(*args)
print(buyer.cart)
print(buyer.price_list)
buyer.check_balance()

buyer.remove_item('lemons')
print(buyer.cart)
print(buyer.price_list)
buyer.check_balance()
```

```
['lemons', 'Jeans', 'oranges']
[5, 50, 5]
['lemons', 'Jeans', 'oranges', 'jeans2']
[5, 50, 5, 50]
110.0
['lemons', 'Jeans']
[5, 50]
['Jeans']
[50]
```

```
'Your current balance is 50.0 Eur.'
```

Exercise 2: Multiple Inheritance and mro with Mendel's Dominant and Recessive Traits Theory

Create two classes **Dominant** and **Recessive** having attributes:

1. The dominant class has following attributes: **seed_shape** (default='round'), **pod_color** (default='green'), **flower_color**(default='purple')

2. The recessive class has following attributes: **seed_shape** (default='wrinkled'), **pod_color** (default='yellow'), **flower_color**(default='white')
3. Create subclass **Pea** which inherits from the above two classes, having same attributes but donot spcify them. Let Python's mro assign them to the "dominant" traits.

```
class Dominant:
    def __init__(self,seed_shape='round', pod_color='green', flower_color='purple'):
        self.seed_shape=seed_shape
        self.pod_color=pod_color
        self.flower_color=flower_color
class Recessive:
    def __init__(self,seed_shape='wrinkled', pod_color='yellow', flower_color='white'):
        self.seed_shape=seed_shape
        self.pod_color=pod_color
        self.flower_color=flower_color
class Pea(Dominant, Recessive):
    def __init__(self):
        super().__init__()

plant1=Pea()
print(plant1.seed_shape)
print(plant1.pod_color)
print(plant1.flower_color)
```

```
round
green
purple
```

Exercise 3: Inheritance, name wrangling, class methods with Online Video Calling App

Create a class **VideoApp**

1. The *VideoApp* must contain the following class attributes:
   - **participants_online** which contains the number of active callers default to 0
   - *action* containing a list of 2 strings: "speaking", "silent"
2. The_VideoApp_ must contain the class method:
   - **show_participants_online** which returns the following message: "{cls.participants_online} are attending meeting."
3. The *VideoApp* must contain the following attributes:

- **user_name** which assigns name of the caller
- *company* which indicates the company represented by the caller

4. The *VideoApp* must contain the following methods:

- **go_online** which increases the attribute _participants_online by 1
- **go_offline** which subtracts 1 from the attribute _participants_online and returns the new _participants_online
- *status* which takes argument *action* and checks if the action argument matches the string in the *action* class attribute, otherwise throws ValueError: "The user must either be "speaking" or "silent"". It also returns the message: "{self.user_name} is *action*"

```
class VideoApp:
    participants_online=0,
    action=['speaking', 'silent']

    @classmethod
    def show_participants_online(cls):
        print(f"{cls.participants_online} are attending meeting")
```

```
    def __init__(self,user_name, company):
        self.user_name=user_name
        self.company = company
    def go_online(self):
        VideoApp.participants_online+=1
    def go_offline(self):
        VideoApp.participants_online-=1
        return VideoApp.participants_online
    def status(self, action):
        if action not in VideoApp.action:
            raise ValueError("The user must either be 'speaking' or 'silent'")
        return f"{self.user_name} is {action}"

user1=VideoApp('Remy', 'InfoRegister')
user1.go_online
user1.status('silent')
```

```
'Remy is silent'
```

Create another class **Message**

1. It has following attributes:
   - **user_name**
   - *message*
2. It has one method:
   - _status_ which returns: "{self.user_name} sent the message: {self.message}". Make this method a __method

```
class Message:
    def __init__(self, user_name, message):
        self.user_name=user_name
        self.message = message
    def __status(self):
        return f"{self.user_name} sent the message: {self.message}"

user2=Message('David', 'Hello! Good day to all.')
user2._Message__status()
```

```
'David sent the message: Hello! Good day to all.'
```

Create another class **ChatApp** which inherits from both parent classes in the following order: **VideoApp** and **Message**

1. Each *ChatApp* takes same attributes as base classes

Now create an instance of this class and explicity call the method __status__ from Message class

```
class ChatApp(VideoApp, Message):
    def __init__(self, user_name, company, message):
        self.user_name=user_name
        self.company=company
        self.message=message

user3=ChatApp('Kim', 'FinnTech', 'Lets start this meeting')
print(user3._Message__status())
```

```
Kim sent the message: Lets start this meeting
```

Exercise 4.1

The following is an excerpt from the **Trainer** class under inference subfolder. The whole file is available under: **scVI_trainer.py** :
[N.B. Do not worry if you still donot understand everything about the code at this point as we will talk about inference and variational autoencoders later on in this course. Also some parts have been omitted for simplicity]

```
class Trainer:
    r"""The abstract Trainer class for training a PyTorch model and monitoring its statistics. It should be
    inherited at least with a .loss() function to be optimized in the training loop.
    Args:
        :model: A model instance from class ``VAE``, ``VAEC``, ``SCANVI``
        :gene_dataset: A gene_dataset instance like ``CortexDataset()``
        :use_cuda: Default: ``True``.
        :metrics_to_monitor: A list of the metrics to monitor. If not specified, will use the
            ``default_metrics_to_monitor`` as specified in each . Default: ``None``.
        :benchmark: if True, prevents statistics computation in the training. Default: ``False``.
        :verbose: If statistics should be displayed along training. Default: ``None``.
        :frequency: The frequency at which to keep track of statistics. Default: ``None``.
        :early_stopping_metric: The statistics on which to perform early stopping. Default: ``None``.
        :save_best_state_metric:  The statistics on which we keep the network weights achieving the best store, and
            restore them at the end of training. Default: ``None``.
        :on: The data_loader name reference for the ``early_stopping_metric`` and ``save_best_state_metric``, that
            should be specified if any of them is. Default: ``None``.
    """
    default_metrics_to_monitor = []

    def __init__(self, model, gene_dataset, use_cuda=True, metrics_to_monitor=None, benchmark=False,
                 verbose=False, frequency=None, weight_decay=1e-6, early_stopping_kwargs=dict(),
                 data_loader_kwargs=dict()):

        self.model = model
        self.gene_dataset = gene_dataset

        self.data_loader_kwargs = {
            "batch_size": 128,
            "pin_memory": use_cuda
        }
        self.data_loader_kwargs.update(data_loader_kwargs)

        self.weight_decay = weight_decay
        self.benchmark = benchmark
        self.epoch = -1  # epoch = self.epoch + 1 in compute metrics
        self.training_time = 0
        self.early_stopping = EarlyStopping(**early_stopping_kwargs)

    def __getattr__(self, name):
        if '_posteriors' in self.__dict__:
            _posteriors = self.__dict__['_posteriors']
            if name.strip('_') in _posteriors:
                return _posteriors[name.strip('_')]
        return object.__getattribute__(self, name)

    def __delattr__(self, name):
        if name.strip('_') in self._posteriors:
            del self._posteriors[name.strip('_')]
        else:
            object.__delattr__(self, name)


    def register_posterior(self, name, value):
        name = name.strip('_')
        self._posteriors[name] = value

    def corrupt_posteriors(self, rate=0.1, corruption="uniform", update_corruption=True):
        if not hasattr(self.gene_dataset, 'corrupted') and update_corruption:
            self.gene_dataset.corrupt(rate=rate, corruption=corruption)
```

```
        for name, posterior in self._posteriors.items():
            self.register_posterior(name, posterior.corrupted())
```

Based on your previous knowledge of OOP and looking at the code above, answer the following questions:

1. What are the arguments with default values stated in the **Trainer** class i.e. name the arguments which are not necessary to include while instantiating the **Trainer** class?
2. Can you name at least 2 model names the user can use as arguments while instantiating the class **Trainer**? e.g: train_object=Trainer(model=?,gene_dataset,...)
3. Can you name two arguments where you can use variable length of the arguments while instantiating the **Trainer** class?
4. List all the methods that you can see under **Trainer** class. Which of these are Dunder methods?
5. We want to be able to call the **corrupt_posteriors** method as an attribute of the **Trainer** class. Modify the above code to enable this.

Solutions:

1. The arguments with default values are use_cuda (default:True), metrics_to_monitor(default:None), benchmark(default:False), verbose(default:False), frequency(default:None), weight_decay(default:1e-6), early_stopping_kwargs(default:dict()),data_loader_kwargs(default:dict())
2. 2 model names which can be used during instatiating class Trainer are: "VAE", "VAEC". Also "SCANVI" can be used.
3. From the definition of __init__ method, we can see early_stopping_kwargs and data_loader_kwargs are the two arguments which take in a dictionary and hence can be variable length
4. The methods used in Trainer class are: __init__, __getattr__, __delattr__, register_posterior, corrupt_posteriors. Of these __init__ , __getattr__, __delattr__ are the dunder methods
5. The corrupt_posteriors function definition is modified as shown below:

```
@property
def corrupt_posteriors(self, rate=0.1, corruption="uniform", update_corruption=True):
        if not hasattr(self.gene_dataset, 'corrupted') and update_corruption:
            self.gene_dataset.corrupt(rate=rate, corruption=corruption)
        for name, posterior in self._posteriors.items():
            self.register_posterior(name, posterior.corrupted())


# When the actual code runs, you would be able to call it like this:
# trainer1=Trainer(VAE, gene_dataset,**early_stopping_kwargs,**data_loader_kwargs)
# trainer1.corrupt_posteriors
```

Exercise 4.2

The following is an excerpt from the **Inference.py** file under inference subfolder. The whole file is available under: **scVI_inference.py**

```
class UnsupervisedTrainer(Trainer):
    r"""The VariationalInference class for the unsupervised training of an autoencoder.
    Args:
        :model: A model instance from class ``VAE``, ``VAEC``, ``SCANVI``
        :gene_dataset: A gene_dataset instance like ``CortexDataset()``
        :train_size: The train size, either a float between 0 and 1 or and integer for the number of training samples
         to use Default: ``0.8``.
        :\*\*kwargs: Other keywords arguments from the general Trainer class.
    Examples:
        >>> gene_dataset = CortexDataset()
        >>> vae = VAE(gene_dataset.nb_genes, n_batch=gene_dataset.n_batches * False,
        ... n_labels=gene_dataset.n_labels)
        >>> infer = VariationalInference(gene_dataset, vae, train_size=0.5)
        >>> infer.train(n_epochs=20, lr=1e-3)
    """
```

```
    default_metrics_to_monitor = ['ll']

    def __init__(self, model, gene_dataset, train_size=0.8, test_size=None, kl=None, **kwargs):
        super().__init__(model, gene_dataset, **kwargs)
        self.kl = kl
        if type(self) is UnsupervisedTrainer:
            self.train_set, self.test_set = self.train_test(model, gene_dataset, train_size, test_size)
            self.train_set.to_monitor = ['ll']
            self.test_set.to_monitor = ['ll']

    @property
    def posteriors_loop(self):
        return ['train_set']


class AdapterTrainer(UnsupervisedTrainer):
    def __init__(self, model, gene_dataset, posterior_test, frequency=5):
        super().__init__(model, gene_dataset, frequency=frequency)
        self.test_set = posterior_test
        self.test_set.to_monitor = ['ll']
        self.params = list(self.model.z_encoder.parameters()) + list(self.model.l_encoder.parameters())

    @property
    def posteriors_loop(self):
        return ['test_set']

    def train(self, n_path=10, n_epochs=50, **kwargs):
        for i in range(n_path):
            # Re-initialize to create new path
            self.model.z_encoder.load_state_dict(self.z_encoder_state)
            self.model.l_encoder.load_state_dict(self.l_encoder_state)
            super().train(n_epochs, params=self.params, **kwargs)

        return min(self.history["ll_test_set"])
```

Answer the following questions:

1. Which class does **UnsupervisedTrainer** inherit from?
2. If we print the AdapterTrainer.__mro__, what order of inheritance would you see?
3. Look at the **posteriors_loop** method for the class AdapterTrainer. Modify the code such that now we donot want this to be the attribute of the class, but rather as a method which takes in argument *set* and returns this value and default it to ['test_set']

Solutions

1. UnsupervisedTrainer inherits from the Trainer class.
2. The AdapterTrainer inherits from UnsupervisedTrainer which inherits from Trainer so the order of mro is: AdapterTrainer > UnsupervisedTrainer>Trainer

```
print(AdapterTrainer.__mro__)
```

```
(<class '__main__.AdapterTrainer'>, <class '__main__.UnsupervisedTrainer'>, <class '__main__.Trainer'>, <class
'object'>)
```

```
class AdapterTrainer(UnsupervisedTrainer):
    def __init__(self, model, gene_dataset, posterior_test, frequency=5):
        super().__init__(model, gene_dataset, frequency=frequency)
        self.test_set = posterior_test
        self.test_set.to_monitor = ['ll']
        self.params = list(self.model.z_encoder.parameters()) + list(self.model.l_encoder.parameters())
```

```
    def posteriors_loop(self, set=['test_set']):
        return set

    def train(self, n_path=10, n_epochs=50, **kwargs):
        for i in range(n_path):
            # Re-initialize to create new path
            self.model.z_encoder.load_state_dict(self.z_encoder_state)
            self.model.l_encoder.load_state_dict(self.l_encoder_state)
            super().train(n_epochs, params=self.params, **kwargs)

        return min(self.history["ll_test_set"])
```