

 Exercises (OOP with Python)_v3.md

Exercises (OOP with Python)

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

Create a class called *Shopping*

- 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
- 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.

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

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

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

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

Create a class **VideoApp**

- 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"
- 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*"

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`

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 explicitly call the method `__status__` from Message class

Exercise 4: OOP with application to scVI

This exercise will help you to identify how OOP has been used in scvi codes to build Deep Generative modeling for Single Cell Transcriptomics. You can find more about the basic usage from [scVI_Basic_Usage](#)

In the following exercise we will work with some extracts of codes from scvi notebooks, and try to answer the questions.

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.__getattr__(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.

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']