Accenture MLOps Expectations

Python Packaging

Goal: Build Reusable python packages and register them in central package registry by refactoring the current codebase.

Success Criterion:

1. How much good design principles have been followed.
2. unit test coverage and CI step setup

Example: One of the popular Design patterns.

GitHub - theomitsa/Python-factories

Design Pattern Factory

```
"""
Factory Method Design Pattern

Intent: Provides an interface for creating objects in a superclass, but
allows
subclasses to alter the type of objects that will be created.
"""


from __future__ import annotations
from abc import ABC, abstractmethod


class Logistic(ABC):
    """
    The Creator class declares the factory method that is supposed to
return an
    object of a Product class. The Creator's subclasses usually provide
the
    implementation of this method.
    """

    @abstractmethod
    def create_transport(self):
        """
        Note that the Creator may also provide some default
implementation of
        the factory method.
        """
        pass

    def plan_delivery(self) -> str:
        """
        Also note that, despite its name, the Creator's primary
```

```
responsibility
        is not creating products. Usually, it contains some core
business logic
        that relies on Product objects, returned by the factory method.
        Subclasses can indirectly change that business logic by
overriding the
        factory method and returning a different type of product from
it.
        """

        # Call the factory method to create a Product object.
        product = self.create_transport()

        # Now, use the product.
        result = f"Creator: The same creator's code has just worked
with {product.deliver()}"

        return result


"""
Concrete Creators override the factory method in order to change the
resulting
product's type.
"""


class RoadLogistic(Logistic):
    """
    Note that the signature of the method still uses the abstract
product type,
    even though the concrete product is actually returned from the
method. This
    way the Creator can stay independent of concrete product classes.
    """

    def create_transport(self) -> Transport:
        return Truck()


class SeaLogistic(Logistic):
    def create_transport(self) -> Transport:
        return Ship()


class Transport(ABC):
    """
    The Product interface declares the operations that all concrete
products
    must implement.
    """

    @abstractmethod
    def deliver(self) -> str:
        pass


"""
Concrete Products provide various implementations of the Product
interface.
"""


class Truck(Transport):
    def deliver(self) -> str:
        return "{Deliver By Truck}"


class Ship(Transport):
    def deliver(self) -> str:
        return "{Deliver By Ship}"


def client_code(transpot: Transport) -> None:
    """
    The client code works with an instance of a concrete creator,
albeit through
    its base interface. As long as the client keeps working with the
creator via
    the base interface, you can pass it any creator's subclass.
    """

    print(f"Client: I'm not aware of the creator's class, but it still
works.\n"
          f"{transpot.plan_delivery()}", end="")


if __name__ == "__main__":
    print("App: Launched with the RoadLogistic.")
    client_code(RoadLogistic())
    print("\n")

    print("App: Launched with the SeaLogistic.")
    client_code(SeaLogistic())
```

Abstract Factory

```
from __future__ import annotations
from abc import ABC, abstractmethod


class AbstractFactory(ABC):
    """
    The Abstract Factory interface declares a set of methods that return
    different abstract products. These products are called a family and
are
    related by a high-level theme or concept. Products of one family
are usually
    able to collaborate among themselves. A family of products may have
several
    variants, but the products of one variant are incompatible with
products of
    another.
    """
    @abstractmethod
    def create_product_a(self) -> AbstractProductA:
        pass

    @abstractmethod
    def create_product_b(self) -> AbstractProductB:
        pass


class ConcreteFactory1(AbstractFactory):
    """
    Concrete Factories produce a family of products that belong to a
single
    variant. The factory guarantees that resulting products are
compatible. Note
    that signatures of the Concrete Factory's methods return an abstract
    product, while inside the method a concrete product is instantiated.
    """

    def create_product_a(self) -> AbstractProductA:
        return ConcreteProductA1()

    def create_product_b(self) -> AbstractProductB:
        return ConcreteProductB1()


class ConcreteFactory2(AbstractFactory):
    """
    Each Concrete Factory has a corresponding product variant.
    """

    def create_product_a(self) -> AbstractProductA:
        return ConcreteProductA2()
    def create_product_b(self) -> AbstractProductB:
        return ConcreteProductB2()


class AbstractProductA(ABC):
    """
    Each distinct product of a product family should have a base
interface. All
    variants of the product must implement this interface.
    """

    @abstractmethod
    def useful_function_a(self) -> str:
        pass


"""
Concrete Products are created by corresponding Concrete Factories.
"""


class ConcreteProductA1(AbstractProductA):
    def useful_function_a(self) -> str:
        return "The result of the product A1."


class ConcreteProductA2(AbstractProductA):
    def useful_function_a(self) -> str:
        return "The result of the product A2."


class AbstractProductB(ABC):
    """
    Here's the the base interface of another product. All products can
interact
    with each other, but proper interaction is possible only between
products of
    the same concrete variant.
    """
    @abstractmethod
    def useful_function_b(self) -> None:
        """
        Product B is able to do its own thing...
        """
        pass

    @abstractmethod
    def another_useful_function_b(self, collaborator: AbstractProductA)
-> None:
        """
        ...but it also can collaborate with the ProductA.

        The Abstract Factory makes sure that all products it creates
are of the
        same variant and thus, compatible.
        """
        pass


"""
Concrete Products are created by corresponding Concrete Factories.
"""


class ConcreteProductB1(AbstractProductB):
    def useful_function_b(self) -> str:
        return "The result of the product B1."

    """
    The variant, Product B1, is only able to work correctly with the
variant,
    Product A1. Nevertheless, it accepts any instance of
AbstractProductA as an
    argument.
    """

    def another_useful_function_b(self, collaborator: AbstractProductA)
-> str:
        result = collaborator.useful_function_a()
        return f"The result of the B1 collaborating with the ({result})"


class ConcreteProductB2(AbstractProductB):
    def useful_function_b(self) -> str:
        return "The result of the product B2."

    def another_useful_function_b(self, collaborator: AbstractProductA):
        """
        The variant, Product B2, is only able to work correctly with the
        variant, Product A2. Nevertheless, it accepts any instance of
        AbstractProductA as an argument.
        """
        result = collaborator.useful_function_a()
        return f"The result of the B2 collaborating with the ({result})"


def client_code(factory: AbstractFactory) -> None:
    """
    The client code works with factories and products only through
abstract
```

```
    types: AbstractFactory and AbstractProduct. This lets you pass any
factory
    or product subclass to the client code without breaking it.
    """
    product_a = factory.create_product_a()
    product_b = factory.create_product_b()

    print(f"{product_b.useful_function_b()}")
    print(f"{product_b.another_useful_function_b(product_a)}", end="")


if __name__ == "__main__":
    """
    The client code can work with any concrete factory class.
    """
    print("Client: Testing client code with the first factory type:")
    client_code(ConcreteFactory1())

    print("\n")

    print("Client: Testing the same client code with the second factory
type:")
    client_code(ConcreteFactory2())
```

Design Pattern Builder

```
class QueryBuilder:
     def __init__(self):
          self.select_value = ''
          self.from_table_name = ''
          self.where_value = ''
          self.groupby_value = ''

     def select(self, select_arg):
          self.select_value = select_arg
          return self

     def from_table(self, from_arg):
          self.from_table_name = from_arg
          return self

     def where(self, where_arg):
          self.where_value = where_arg
          return self

     def groupby(self, groupby_args):
          self.groupby_value = groupby_args
          return self
```

```
    def build(self):
        if self.where_value:
            where_clause = f'WHERE {self.where_value}'
        if self.groupby_value:
            groupby_clause = f'GROUP BY {self.groupby_value}'

        return f"""
            SELECT {self.select_value}
            FROM {self.from_table_name}
            {where_clause}
            {groupby_clause}
        """

# Simple builder pattern example for building queries
query = QueryBuilder()
query.select('Customer, Region, SUM(SaleValue)') \
    .from_table('Sales') \

    # Optional - add where and groupby clauses.
    # Object construction can easily be either simple or complex
query.where('DATEDIFF(day, TimeStamp, CURRENT_TIMESTAMP) < 180') \
    .groupby('Customer, Region')

# Builder pattern collects the optional arguments and builds the actual
SQL text
query_text = query.build()

"""
query_text value:
SELECT Customer, Region, SUM(SaleValue)
FROM Sales
WHERE DATEDIFF(day, TimeStamp, CURRENT_TIMESTAMP) < 180
GROUP BY Customer, Region
"""
```

Design Pattern Decorator

```
from time import time

def log_time(func):
    """Logs the time it took for func to execute"""
    def wrapper(*args, **kwargs):
        start = time()
        val = func(*args, **kwargs)
        end = time()
        duration = end - start
        print(f'{func.__name__} took {duration} seconds to run')
        return val
    return wrapper

# Example usage
@log_time
def get_data(db, query):
    """Gets data from a SQL-based database"""
    data = db.get(query)
    return data

if __name__ == '__main__':
    # Decorated function will print 'get_data took X seconds to run'
    db = SQLDB()
    query = 'SELECT * FROM foo'
    data = get_data(db, query)
```

Kubeflow/Vertex AI Component Development

Goal: Build reusable vertex ai components with docker file, python script, and kubeflow yaml file

Success Criterion:

1. Each needs to be individually deployed.
2. Reusable in cross projects with error handling in place

Example :

examples/named_entity_recognition at master · kubeflow/examples · GitHub

Logging, Monitoring, and Exception Handling

Goal: Refactor existing code to handle logging monitoring and error handling.
Success Criterion:

1. Error Handling via try catch block customized to the error in question.
2. Logging the steps and each important execution steps.

Example:

A Complete Guide to Logging in Python with Loguru | Better Stack Community

Python Exception Hierarchy

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
  +-- ImportWarning
```

```
     +-- UnicodeWarning
     +-- BytesWarning
```

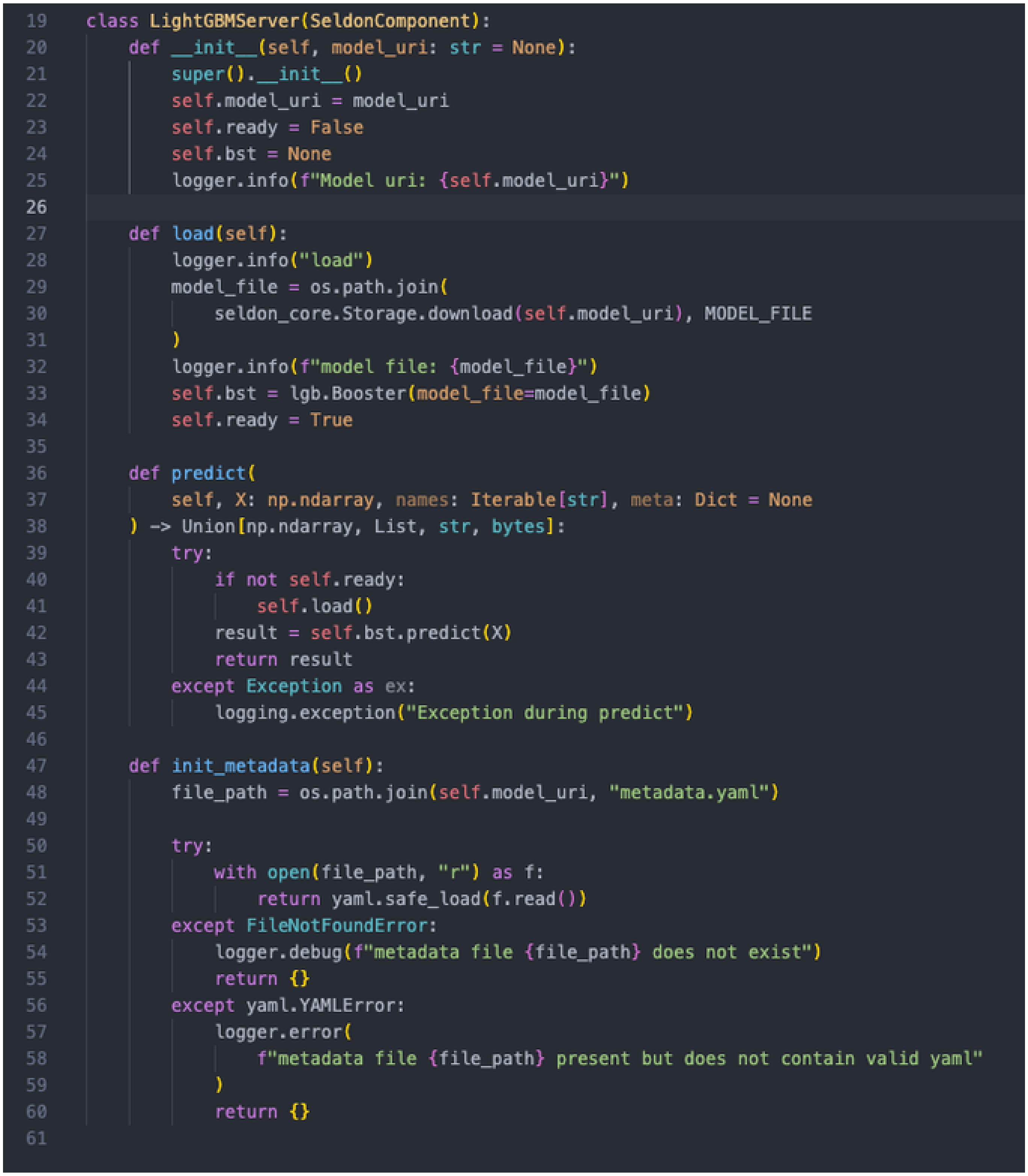
Build custom exceptions like

```
# Raise an instance of the Exception class itself
raise Exception('Ummm... something is wrong')
# Raise an instance of the RuntimeError class
raise RuntimeError('Ummm... something is wrong')
# Raise a custom subclass of Exception that keeps the timestamp the
exception was created
from datetime import datetime
class SuperError(Exception):
    def __init__(self, message):
        Exception.__init__(message)
        self.when = datetime.now()
raise SuperError('Ummm... something is wrong')
```

Pandas : pandas/__init__.py at 00af20a22c6c64ad2d8f48345beaede0e946d630 · pandas-dev/pandas · GitHub

Few code snippets might be useful.

1. I used this pattern for deploying models using Seldon.io



```python
class LightGBMServer(SeldonComponent):
    def __init__(self, model_uri: str = None):
        super().__init__()
        self.model_uri = model_uri
        self.ready = False
        self.bst = None
        logger.info(f"Model uri: {self.model_uri}")

    def load(self):
        logger.info("load")
        model_file = os.path.join(
            seldon_core.Storage.download(self.model_uri), MODEL_FILE
        )
        logger.info(f"model file: {model_file}")
        self.bst = lgb.Booster(model_file=model_file)
        self.ready = True

    def predict(
        self, X: np.ndarray, names: Iterable[str], meta: Dict = None
    ) -> Union[np.ndarray, List, str, bytes]:
        try:
            if not self.ready:
                self.load()
            result = self.bst.predict(X)
            return result
        except Exception as ex:
            logging.exception("Exception during predict")

    def init_metadata(self):
        file_path = os.path.join(self.model_uri, "metadata.yaml")

        try:
            with open(file_path, "r") as f:
                return yaml.safe_load(f.read())
        except FileNotFoundError:
            logger.debug(f"metadata file {file_path} does not exist")
            return {}
        except yaml.YAMLError:
            logger.error(
                f"metadata file {file_path} present but does not contain valid yaml"
            )
            return {}

```