Create Environment:

* conda create -p venv python==3.10
* conda activate venv/

Create .gitignore file:

* Files like .venv don’t get uploaded in github

Create folder .github/workflows/main.yaml

* Deployement github actions

Github Setup

* echo "# Credit-Risk-Default-Prediction" >> README.md
* git init
* git add README.md ( git add .)
* git commit -m "first commit"
* git branch -M main
* git remote add origin https://github.com/reevubabbai2003/Credit-Risk-Default-Prediction.git
* git push -u origin main

Setup.py

'''

The setup.py file is an essential part of packaging and

distributing Python projects. It is used by setuptools

(or distutils in older Python versions) to define the configuration

of your project, such as its metadata, dependencies, and more

'''

from setuptools import find\_packages

* find\_packages: scan through every folder and whenever there is \_\_init\_\_.py file consider that folder as a package.

**Why We Create a setup.py File 📦**

Think of a setup.py file as the official instruction manual for your project. It tells packaging tools like pip everything they need to know to handle your code correctly.

Its main purposes are:

1. **To Provide Metadata:** It contains key information about your project, such as its **name**, **version**, **author**, and a brief description. This is what you see when you browse packages on PyPI (the Python Package Index).
2. **To List Dependencies:** The **install\_requires** argument inside setup.py lists all the other Python packages that your project needs to run. When someone installs your project, pip reads this list and automatically installs all of them.
3. **To Find Your Code:** It tells pip which parts of your project are the actual Python packages to be included. The **find\_packages()** function is a common way to do this automatically.
4. **To Make it Installable:** Most importantly, having a setup.py file allows you to bundle your project into standard formats (like a "wheel" file) that can be easily shared and installed by others using a simple pip install command.

Without it, your project is just a collection of scripts that someone would have to manually download and manage.

* **-e** stands for **"editable."**
* **.** is a shortcut for the **current directory** (where your setup.py is).

So, the command means: "Install the project in the current directory in editable mode."

**How It Works: The Shortcut Analogy**

To understand what "editable" means, think of it like creating a shortcut versus making a copy of a file.

* **Normal Install (pip install .):** This is like **making a copy** of your project's files and moving that copy into Python's site-packages directory. If you change your original source code, the installed copy remains unchanged, and you won't see your updates until you reinstall.
* **Editable Install (pip install -e .):** This is like **creating a shortcut**. Instead of copying the files, pip places a link in the site-packages directory that points directly back to your original source code.

**The Connection**

When you run pip install -e ., pip **reads your setup.py file** to understand how to create this "shortcut." It uses the metadata and dependency information from setup.py to link your project into your Python environment.

**Logging -> logger.py**

**What Code Goes Inside logger.py?**

The file typically contains code to configure a logger using Python's built-in logging module. It defines the format of the log messages, the level of detail to capture (e.g., INFO, ERROR), and where the logs should be sent.

**Why We Make This File**

Creating a dedicated logger.py is about moving from messy, temporary debugging to a professional, maintainable system. Think of it like installing a centralized security system in a building instead of placing random webcams in different rooms.

Here are the key reasons:

1. **Centralization and Consistency**: All parts of your application can import the same logger instance. This ensures that every log message across your entire project has the **exact same format** and goes to the **same place**.
2. **Control and Flexibility**: You can change your entire project's logging behavior by editing just this one file. Want to stop logging to a file and send logs to a web service instead? Or change the log level from INFO to DEBUG to get more detail? You only have to make the change in logger.py.
3. **Separates Concerns**: It keeps the logic for *how* to log separate from the business logic of your application. Your main code just needs to say "log this message" (logging.info(...)) without worrying about timestamps, file paths, or formatting.
4. **Different Log Levels**: Logging provides severity levels (DEBUG, INFO, WARNING, ERROR, CRITICAL). This is far more powerful than print(), as you can configure your logger to show only errors in a production environment while showing detailed debug information during development.
5. **Disables print() Statements**: It's a best practice to remove all print() statements used for debugging before deploying code. A logging system replaces them, allowing you to simply change the log level to silence unimportant messages without having to find and remove every print statement.

def \_\_init\_\_(self,error\_message,error\_details:sys):

* error\_details:sys, is a **type hint** in Python.
* **sys**: This is the **expected type**. It tells anyone reading the code that the error\_details parameter is expected to be the Python sys module itself. The sys module provides access to system-specific parameters and functions, and it's commonly used in custom exception handling to get detailed information about an error that just occurred.

**Logging** is the practice of recording events that happen while your program is running. **Exception handling** is the mechanism for managing errors and preventing them from crashing your program.

In short, you use exception handling to catch an error, and logging to write down that the error happened.

**What is Logging? 📝**

Logging is like keeping a detailed journal or a "black box" for your application. Instead of just using print() statements for debugging, logging provides a robust system to record application activity to a file, the console, or another destination.

This is crucial because you can record not just errors, but also normal operational messages, warnings, or detailed debugging information.

**Key Features:**

* **Severity Levels:** You can categorize messages by importance (e.g., DEBUG, INFO, WARNING, ERROR), and configure your logger to only show messages of a certain severity.
* **Timestamps & Context:** Log messages are automatically timestamped and can include context like the file name and line number where the event occurred.
* **Configurability:** You can easily switch logging on or off, or change where logs are sent (e.g., from the console to a file) without changing your application code.

**What is Exception Handling? 🛡️**

Exception handling is your code's safety net. It's how you manage errors gracefully instead of letting them crash your program. You use a try...except block to isolate risky code.

Think of it like airbags in a car. They don't prevent a crash, but they manage the impact to prevent a catastrophic outcome.

**How It Works:**

* **try block:** You place the code that might cause an error inside the try block.
* **except block:** If an error (an "exception") occurs in the try block, the program immediately jumps to the except block. This is your "plan B" code that runs instead of crashing.
* **else block (optional):** This code runs only if the try block completes successfully without any errors.
* **finally block (optional):** This code *always* runs at the end, whether an error occurred or not. It's typically used for cleanup actions, like closing a file.

**How They Work Together: The Perfect Partnership**

The most common pattern is to catch an exception and then log it. This gives you the best of both worlds: your application doesn't crash, and you have a detailed record of what went wrong so you can fix it later.



