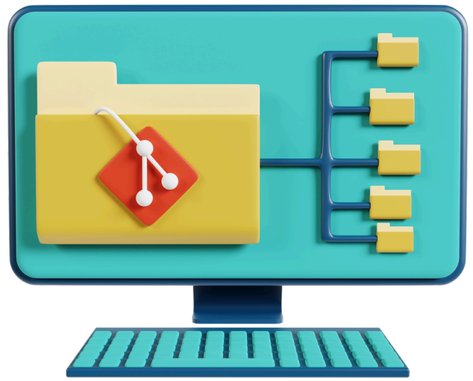
Git e Il versionamento

Un sistema di versionamento è un sistema che registra le modifiche a un file o a un insieme di file nel tempo, in modo che tu possa richiamare versioni specifiche più avanti.

I sistemi di versionamento sono particolarmente utili per gestire il codice sorgente di programmi e documentazione, consentendo a più individui di lavorare insieme su un progetto e mantenendo uno storico delle modifiche effettuate.



Git e Il versionamento

La teoria dietro i sistemi di versionamento si basa sull'idea di migliorare la collaborazione e l'organizzazione nel ciclo di sviluppo del software. Questi sistemi consentono agli sviluppatori di:

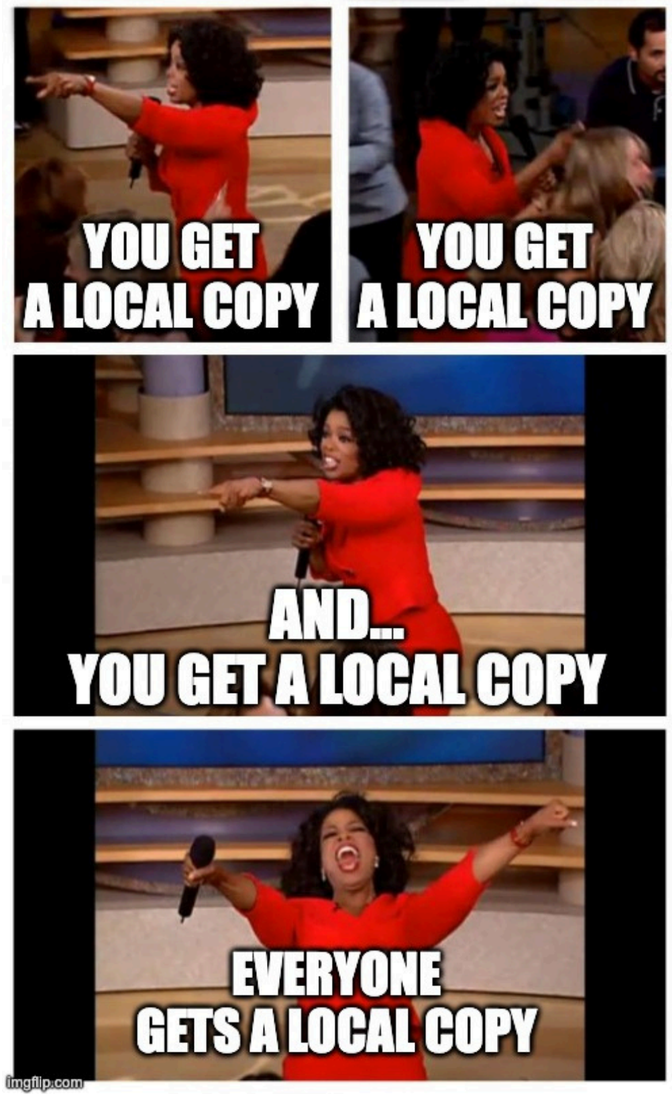
- Conservare la storia: Ogni modifica ai file è tracciata insieme a dettagli come l'autore della modifica e la data.
- Branching e merging: Creare rami di sviluppo indipendenti (branches) per lavorare su nuove funzionalità o correzioni senza disturbare il flusso principale del progetto (main o master). Successivamente, i cambiamenti possono essere integrati (merged) nel ramo principale.
- Gestire conflitti: Risolvere automaticamente i cambiamenti conflittuali tra più contributi e aiutare gli sviluppatori a risolvere i conflitti manualmente quando necessario.

Storia dei sistemi di versionamento

I sistemi di versionamento sono evoluti significativamente nel corso degli anni:

1. Sistemi di versionamento locali: Il primo tipo di sistema di versionamento, come RCS (Revision Control System), memorizzava le modifiche in un database locale. Questo permetteva a un singolo sviluppatore di tenere traccia delle modifiche in modo semplice ma non supportava bene la collaborazione tra più utenti.
2. Sistemi di versionamento centralizzati (CVCS): Sistemi come CVS (Concurrent Versions System) e successivamente Subversion (SVN) introducevano un modello in cui tutte le modifiche venivano memorizzate in un server centrale. Questo permetteva a più sviluppatori di lavorare insieme, ma presentava problemi di single point of failure e accessibilità limitata in assenza di connettività al server.
3. Sistemi di versionamento distribuiti (DVCS): Con l'introduzione di Git (creato da Linus Torvalds per lo sviluppo del kernel Linux) e Mercurial, i sistemi di versionamento hanno adottato un approccio distribuito. Ogni sviluppatore ha una copia completa del repository, inclusa la storia completa delle modifiche, rendendo possibile lavorare in modo completamente autonomo e sincronizzare le modifiche con altri repository quando necessario.

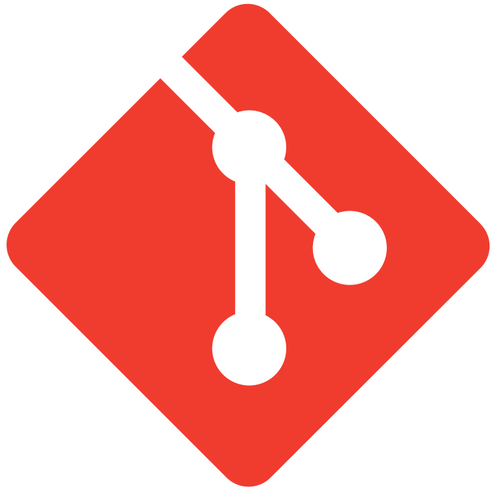
Git and meme


YOU GET
A LOCAL COPY
YOU GET
A LOCAL COPY
AND...
YOU GET A LOCAL COPY
EVERYONE
GETS A LOCAL COPY
imgflip.com

Git e Il versionamento

Git è un sistema di controllo di versione distribuito (DVCS) molto diffuso, progettato per gestire progetti di ogni dimensione con velocità ed efficienza.

Creato da Linus Torvalds nel 2005, Git è diventato lo standard per il versionamento nel mondo dello sviluppo software, specialmente per la gestione del codice sorgente.



Git e Il versionamento

Git è basato su una teoria di snapshot, piuttosto che su differenze tra le versioni. Ogni volta che si salva lo stato del progetto (tramite il comando commit), Git prende uno snapshot di tutti i file che sono stati modificati dall'ultimo commit.

Se un file non è cambiato, Git non salva una nuova copia, ma un link al file identico precedente.

Una delle fondamenta teoriche di Git è il modello di dati che usa per gestire e immagazzinare la storia del progetto. Git considera i dati come un insieme di snapshot di un mini filesystem, con riferimenti che sono organizzati in una struttura ad albero.

Git e Il versionamento

caratteristiche di Git

- Distribuito: Ogni sviluppatore lavora con una copia locale dell'intero repository, non solo con l'ultima versione dei file. Questo significa che ogni clone di un repository Git funziona come un backup completo con tutte le funzionalità.
- Integrità dei dati: Git usa una combinazione di checksum SHA-1 per identificare e gestire i suoi oggetti (commits, file, directory, ecc.), il che garantisce l'integrità del codice sorgente e previene corruzione, manipolazione accidentale o intenzionale dei dati.
- Branching e merging veloci: Git gestisce in modo molto efficiente le operazioni di branch e merge. I branch sono leggeri (si tratta essenzialmente di riferimenti a un commit specifico) e il processo di cambio da un branch all'altro è molto rapido. Il merging è altrettanto veloce, rendendo possibile il flusso di lavoro con molteplici linee di sviluppo simultaneamente.

Git e Il versionamento

caratteristiche di Git

- Stashing e stage area: Git offre un'area di "staging" dove i file possono essere formati per il prossimo commit, oltre alla possibilità di mettere da parte (stash) modifiche non ancora pronte per essere committate, mantenendo così il workspace pulito.
- Tagging: Git permette di taggare specifici commit con etichette stabili, utili per il rilascio di versioni del software (ad esempio, v1.0, v2.0 e così via).
- Velocità: Le operazioni in Git sono estremamente rapide, poiché la maggior parte delle operazioni ha bisogno solo di risorse locali, non di rete. Anche la gestione di grandi progetti è incredibilmente efficiente in termini di performance.

Git e Il versionamento

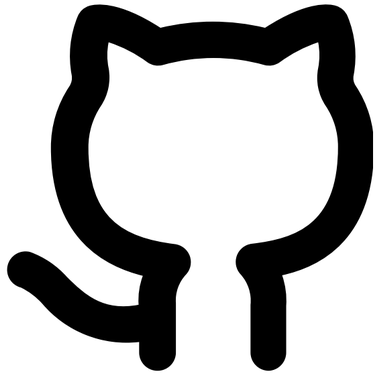
caratteristiche di Git

- Stashing e stage area: Git offre un'area di "staging" dove i file possono essere formati per il prossimo commit, oltre alla possibilità di mettere da parte (stash) modifiche non ancora pronte per essere committate, mantenendo così il workspace pulito.
- Tagging: Git permette di taggare specifici commit con etichette stabili, utili per il rilascio di versioni del software (ad esempio, v1.0, v2.0 e così via).
- Velocità: Le operazioni in Git sono estremamente rapide, poiché la maggior parte delle operazioni ha bisogno solo di risorse locali, non di rete. Anche la gestione di grandi progetti è incredibilmente efficiente in termini di performance.

Git hub

GitHub è una piattaforma di hosting per il controllo di versione e la collaborazione, che permette agli sviluppatori di lavorare più efficacemente su progetti insieme.

Utilizza Git, il sistema di controllo di versione distribuito, per gestire i repository dei progetti. Fondata nel 2008 da Tom Preston-Werner, Chris Wanstrath, e PJ Hyett, GitHub è diventata una delle piattaforme più popolari e ampiamente utilizzate per la collaborazione nel software development.



Git and meme


GIT
git
STOP INTERFERING
WITH MY COMPUTER CLASS
WHERE I TELL PUPILS GIT
IS JUST MICROSOFT GITHUB
IS NOT GITHUB

Git e Il versionamento

Principali Funzionalità di GitHub

Repository Hosting: GitHub permette agli utenti di caricare o clonare repository di codice sorgente. Questi repository sono accessibili agli sviluppatori di tutto il mondo e possono essere sia pubblici che privati.

Gestione di Branch e Merge: Attraverso l'interfaccia di GitHub, gli utenti possono facilmente creare branch e gestire i merge, facilitando i flussi di lavoro di sviluppo parallelo e la revisione del codice.

Pull Requests e Code Review: Una delle funzionalità più potenti di GitHub è la capacità di gestire "pull requests", che sono proposte o richieste di unione di un branch in un altro. Questo facilita le discussioni su specifiche modifiche con i colleghi prima di integrarle nel branch principale.

Git e Il versionamento

Principali Funzionalità di GitHub

Issue Tracking: GitHub offre un sistema di tracciamento degli issue integrato che permette ai team di tenere traccia dei bug, delle richieste di funzionalità e di altri compiti all'interno del progetto.

GitHub Actions: È un ambiente di CI/CD (Continuous Integration/Continuous Delivery) integrato che permette agli utenti di automatizzare test, build e deploy direttamente dalla piattaforma GitHub.

GitHub Pages: Permette agli utenti di ospitare gratuitamente siti web direttamente dai loro repository GitHub. È comunemente usato per ospitare la documentazione del progetto o siti web personali.

Git e Il versionamento

Ma quindi come si usa github?

Branch

Un branch in Git è essenzialmente un puntatore mobile su uno degli snapshot (commits) nel tuo repository. Quando si lavora su una funzionalità, bug fix, o esperimento, creare un nuovo branch permette di lavorare isolatamente senza disturbare il flusso principale del progetto, comunemente rappresentato dal branch "main" o "master".

Fork

Un fork è una copia di un repository intero che viene ospitato nel tuo account GitHub. I fork sono utilizzati per copiare e modificare progetti senza influenzare il repository originale. Sono particolarmente utili in progetti open source, dove gli utenti possono contribuire al progetto originale attraverso i fork.

Merging

Dopo aver completato il lavoro nel branch, è possibile unire le modifiche al branch principale (ad esempio, main). Prima di fare il merge, è spesso necessario eseguire un git pull per aggiornare il tuo repository locale con le ultime modifiche del repository remoto, e risolvere eventuali conflitti.

Git e Il versionamento

Prima di andare ad esercitarci ora keyword!

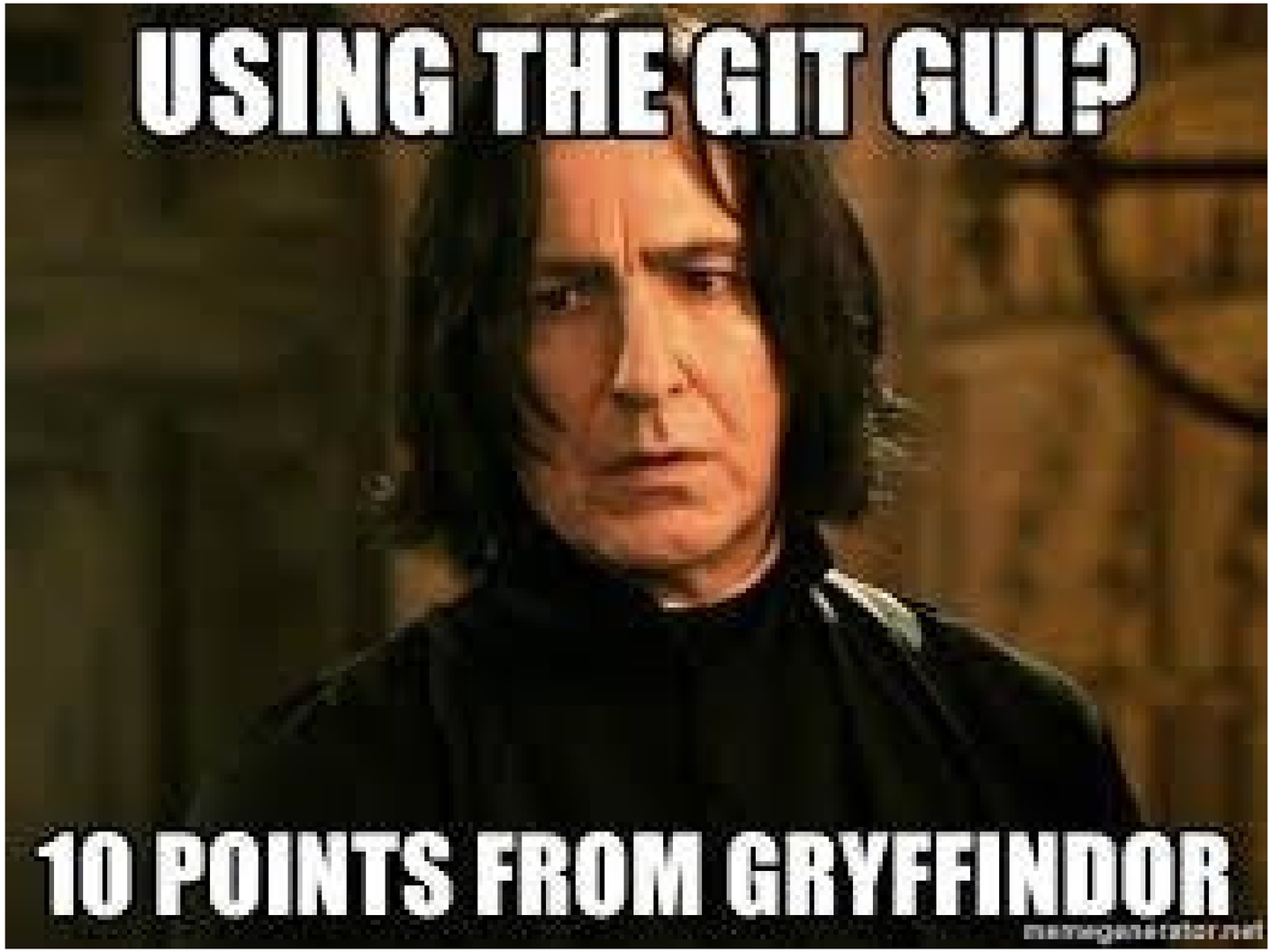
- ***Repository (Repo): Un archivio digitale dove vengono conservati e tracciati i file e la storia dei loro cambiamenti. È l'elemento base per tutto il lavoro in Git e GitHub.***
- ***Commit: Un "salvataggio" o snapshot del tuo progetto in Git. Ogni commit registra le modifiche ai file e contiene un messaggio di commit che descrive cosa è stato cambiato.***
- ***Branch: Una linea indipendente di sviluppo nel repository, utilizzata per lavorare su nuove funzionalità o bug fix senza disturbare il codice principale (main branch).***
- ***Merge: L'azione di unire i cambiamenti da un branch a un altro, tipicamente per integrare le modifiche di una feature branch nella main branch dopo il completamento dello sviluppo.***
- ***Fork: Una copia personale di un altro repository, ospitata nel tuo account GitHub. Viene utilizzato per proporre modifiche al progetto originale o per usarlo come punto di partenza per un nuovo progetto.***
- ***Conflict: Si verifica quando più modifiche allo stesso pezzo di codice sono fatte in modi diversi su branch diversi. Questi devono essere risolti manualmente prima di poter completare un merge.***

Git e Il versionamento

Prima di andare ad esercitarci ora keyword!

- ***Pull Request (PR): Una richiesta inviata ai gestori del repository originale per "tirare" le modifiche dal tuo fork o branch. È usata per discutere le modifiche proposte prima che vengano integrate nel progetto principale.***
- ***Push: Il comando usato per caricare il contenuto del repository locale al repository remoto su GitHub. Questo permette di condividere le modifiche che hai fatto localmente con altri sviluppatori.***
- ***Fetch: Un comando che ti permette di scaricare i cambiamenti da un repository remoto, ma senza integrarli nel tuo lavoro corrente. È utile per vedere cosa stanno facendo gli altri senza influenzare il tuo lavoro locale.***
- ***Pull: Simile a fetch, ma anche integra (merge) i cambiamenti nel tuo branch attuale, aggiornando il tuo lavoro con le ultime modifiche dal repository remoto.***
- ***Stash: Una funzione di Git che ti permette di temporaneamente mettere da parte le modifiche che hai fatto per poter lavorare su qualcos'altro. Puoi poi riapplicare quelle modifiche più tardi.***

Git and meme



GitHub Desktop

GitHub Desktop è un'applicazione grafica che facilita la gestione di repository Git senza dover utilizzare la riga di comando.

È sviluppato da GitHub e offre un'interfaccia utente semplice e intuitiva, ideale per chi è alle prime armi con Git o preferisce un'interfaccia più visuale rispetto alla linea di comando.
GitHub Desktop è disponibile per Windows e macOS.

GitHub Desktop

Come usare GitHub Desktop, per iniziare con GitHub Desktop, segui questi passaggi:

- ***Download e Installazione: Scarica GitHub Desktop dal sito ufficiale di GitHub e installalo sul tuo sistema operativo.***

- ***Configurazione dell'Account: Collega l'applicazione al tuo account GitHub (o GitHub Enterprise) seguendo le istruzioni fornite dall'applicazione.***

- ***Clonazione di un Repository: Clona un repository esistente da GitHub o crea un nuovo repository locale attraverso l'interfaccia.***

- ***Gestione del Codice: Usa l'applicazione per fare commit delle tue modifiche, gestire i branch, e sincronizzare le modifiche con il repository remoto.***

Esercizio con git

Obiettivo

L'obiettivo di questo esercizio è sviluppare un'applicazione Java che simuli il processo di produzione in una fabbrica di cioccolato. L'esercizio richiede di implementare diverse funzionalità, distribuite in vari branch, per poi unirle nel branch principale.

Descrizione - CLASSE BASE --> Fabbrica

La fabbrica di cioccolato deve essere in grado di produrre cioccolatini, tavolette di cioccolato e cioccolato caldo. Ogni prodotto avrà caratteristiche specifiche come tipo di cioccolato (fondente, al latte, bianco) e aggiunte (nocciole, mandorle, biscotti).
La fabbrica potrà produrre solo 100 cioccolato al giorno produrre Un cioccolatino costerà 2 / Una tavoletta da 4 a 24 in base alla sua grandezza / Una cioccolata calda minimo 20, finito 100 di cioccolato sarà chiuso day e andrà stampato un resoconto e una richiesta di continuazione.

Requisiti

Classe Base Cioccolato:

- ***Proprietà comuni come tipo di cioccolato e percentuale di cacao e MAX 100 day.***
- ***Metodo produce() che stampa il tipo di cioccolato e la percentuale di cacao.***

Classe Cioccolatino (estende Cioccolato):

- ***Proprietà aggiuntive come forma e ripieno.***
- ***Override del metodo produce() per includere un calcolo di quanti cioccolato.***

Classe Tavoletta (estende Cioccolato):

- ***Proprietà aggiuntive come peso e se contiene o meno aggiunte (nocciole, ecc.).***
- ***Override del metodo produce() per includere da quanti cioccalatini e composta.***

Classe CioccolatoCaldo (estende Cioccolato):

- ***Proprietà aggiuntive come temperatura e densità.***
- ***Override del metodo produce() per includere da quanto ciccolato e stato fatto***

Istruzioni per il Branching

Branch main:

- ***Contiene solo la classe base Cioccolato.***
- ***Al termine di ogni fase fai il merge del branch nel main.***

Branch feature-cioccolatini:

- ***Crea questo branch dal main.***
- ***Implementa e testa la classe Cioccolatino.***

Branch feature-tavolette:

- ***Crea questo branch dal main.***
- ***Implementa e testa la classe Tavoletta.***

Branch feature-cioccolato-caldo:

- ***Crea questo branch dal main.***
- ***Implementa e testa la classe CioccolatoCaldo.***

Esercizio con SQl e git

Realizzare un piccolo gestionale per la prenotazione di servizi (ad esempio sale conferenze, aule studio o attrezzature).

Il progetto deve impiegare un database SQL per la memorizzazione e l'aggiornamento dei dati e un'applicazione Python per la gestione delle operazioni di prenotazione, visualizzazione e cancellazione.
Organizzare il lavoro in un repository Git, lavorando in un gruppo di 2-3 persone, utilizzando branch separati per gestire le diverse parti del codice (creazione e gestione del database, interfaccia Python, logica di prenotazione, ecc.).

La struttura del database dovrà contenere almeno le tabelle necessarie per:

1. ***Utenti (informazioni di base, ad esempio nome, email).***
2. ***Risorse (sale o oggetti prenotabili, con relativi attributi).***
3. ***Prenotazioni (collega utenti e risorse, registrando data/ora e stato).***

L'applicazione Python dovrà permettere di:

- ***Inserire nuovi utenti.***
- ***Aggiungere nuove risorse.***
- ***Effettuare, visualizzare e cancellare prenotazioni.***
- ***Mostrare un report sintetico delle prenotazioni esistenti***

Esercizio con analisi e git

Si vuole realizzare un'analisi del famoso dataset Iris utilizzando Pandas e NumPy. Il lavoro deve essere suddiviso in Tre branch consecutivi, ognuno dedicato a una parte specifica del progetto.

- ***Branch 1 – Esplorazione iniziale: Caricamento del dataset, verifica delle informazioni di base (dimensioni, tipi di dato, eventuali valori mancanti), statistica descrittiva iniziale (media, deviazione standard, minimo, massimo) e semplice visualizzazione di base (ad esempio conteggio di quante specie ci sono).***

- ***Branch 2 – Analisi approfondita: Utilizzo delle funzionalità di Pandas per analisi più dettagliate (correlazioni tra le variabili, raggruppamenti per specie, confronto dei valori medi e massimi delle diverse caratteristiche)***

- ***Branch 3 - Aggiungi almeno due grafici a scelta (boxplot, scatter plot) per comprendere meglio la distribuzione dei dati e descrivi con un print cosa dimsotrano questi grafici.***

Simulatore “Noleggio Smart Mobility”

Requisiti funzionali (minimi)

1. ***L’app deve permettere di:***
   - ***Registrare un parco mezzi (almeno 6 oggetti di tipi diversi).***
   - ***Elencare solo i mezzi disponibili.***
   - ***Prenotare un mezzo per una durata in minuti.***
   - ***Chiudere una corsa e calcolare il costo totale (polimorfico).***
2. ***Il costo dipende dal tipo del mezzo (tariffe diverse e/o sovrapprezzi).***
3. ***Se il mezzo è elettrico, la batteria scende durante la corsa; se scende sotto una soglia, applicare un fee di “ricarica emergenza”.***
4. ***Se il mezzo è pieghevole, applicare uno sconto su tratte brevi.***
5. ***Validazioni essenziali (es. non prenotare mezzi già in uso, durata > 0, batteria 0–100, ecc.).***

Dominio (vincoli di design)

1) Classe astratta

- ***abstract class Veicolo***
  - ***Campi privati + proprietà (ID, Modello, TariffaBaseAlMinuto, IsDisponibile, ecc.).***
  - ***Costruttori coerenti.***
  - ***Metodi:***
    - ***public abstract decimal CalcolaCosto(int minuti); (polimorfismo obbligatorio)***
    - ***public virtual void AvviaCorsa() / public virtual void TerminaCorsa() con comportamento base (modifica disponibilità).***
  - ***ToString() sovrascritto per stampa leggibile.***

2) Sottoclassi concrete (almeno 3)

- ***Esempi: Monopattino, Bici, Scooter, Auto, E-Bike.***
- ***Ognuna ridefinisce CalcolaCosto con logiche diverse (sovrapprezzi/sconti).***

3) Interfacce (almeno 2, usate da più classi)

- ***interface IElettrico***
  - ***Proprietà: int Batteria { get; }***
  - ***Metodi: void Ricarica(int percento);***
  - ***Contratto: durante TerminaCorsa() un mezzo elettrico riduce la batteria in base ai minuti.***
- ***interface IPieghevole***
  - ***Proprietà: bool ÈPiegato { get; }***
  - ***Metodi: void Piega(); void Apri();***
  - ***Contratto: eventuale sconto su tratte brevi quando piegato.***
- ***(Facoltativa) interface IPrenotabile { bool Prenota(); bool Rilascia(); }***

Nota: Le interfacce devono essere indipendenti dalla gerarchia (le implementano solo i tipi pertinenti).

Incapsulamento (obbligatorio)

- ***Tutti i dati sensibili privati.***
- ***Proprietà con validazione (es. TariffaBaseAlMinuto > 0, Batteria tra 0 e 100).***
- ***Metodi pubblici chiari e coesi; niente setter pubblici non necessari.***

Polimorfismo (obbligatorio)

- ***Usare Veicolo come tipo di riferimento in collezioni/metodi.***
- ***Esempio: List<Veicolo> parco; e operare senza conoscere il tipo concreto.***
- ***Il calcolo del costo avviene chiamando CalcolaCosto(minuti) su Veicolo.***

Comportamenti suggeriti (esempi da adattare)

- ***Monopattino (elettrico): costo = base/min + sovrapprezzo se batteria < 15%.***
- ***Bici (pieghevole): costo = base/min – sconto sui primi 10 minuti se piegata a inizio corsa.***
- ***Scooter (elettrico): costo = base/min + fee fisso per avvio; consumo batteria maggiore.***

Git e Il versionamento

Ecco una lista dei comandi Git più comuni con spiegazioni su come utilizzarli, che funge da guida testuale rapida per iniziare con Git:

Configurazione Iniziale

- git config --global user.name "nome" Imposta il nome che verrà allegato ai tuoi commit e tag.
- git config --global user.email "email@example.com" Imposta l'email che verrà allegata ai tuoi commit e tag.

Creazione e Clonazione di Repository

- git init Inizializza un nuovo repository Git locale.
- git clone <url> Crea una copia locale di un repository Git remoto.

Gestione dei Branch

- git branch Elenco di tutti i branch locali nel repository corrente.
- git branch <nome-branch> Crea un nuovo branch.
- git checkout <nome-branch> Passa a un branch esistente.
- git checkout -b <nome-branch> Crea un nuovo branch e passa ad esso.

Fare Cambiamenti

- git status Mostra lo stato corrente del repository con file modificati e non tracciati.
- git add <file> Aggiunge un file alla staging area in preparazione per il commit.
- git commit -m "messaggio di commit" Registra le modifiche fatte ai file nella staging area con un messaggio di commit.
- git diff Mostra le differenze tra file o branch.

Storico e Log

- git log Mostra una cronologia dei commit per il branch corrente.
- git log --follow <file> Mostra la cronologia dei cambiamenti specifica di un file.

Annullare Cambiamenti

- git checkout -- <file> Annulla le modifiche non committate di un file.
- git reset --hard Annulla tutti i cambiamenti locali nel repository.
- git revert <commit> Crea un nuovo commit che annulla le modifiche di un commit precedente.

Sincronizzazione con Remote

- git push <remote> <branch> Invia i cambiamenti al repository remoto.
- git pull <remote> Riceve cambiamenti e aggiorna il repository locale per essere sincronizzato con il remoto.
- git fetch <remote> Scarica oggetti e riferimenti da un altro repository.

Gestione dei Conflitti

- Quando git pull o git merge riscontrano conflitti, Git ti chiederà di risolverli manualmente. Dopo aver risolto i conflitti in un file:
- git add <file> Marca il file come risolto.
- git commit Completa il merge registrando un nuovo commit.

Utilità

- git stash Salva temporaneamente i file tracciati e non committati, permettendoti di cambiare branch con un workspace pulito.
- git stash pop Riapplica i cambiamenti salvati con git stash.

Esercizio di gruppo

Requisiti Tecnici

- ***Il codice deve essere organizzato utilizzando l'ereditarietà.***
- ***Non è consentito l'uso di librerie esterne come NumPy; la logica delle matrici deve essere gestita tramite liste di liste.***
- ***Potete far riempire la matrice tramite AI***

1. La Superclasse: GestoreMatrice Dovete creare una classe base che abbia:

- ***Attributo: dati (la matrice, passata come lista di liste al costruttore).***
- ***Metodo: stampa_matrice(): stampa la matrice in formato tabellare leggibile (riga per riga).***
- ***Metodo: valida_matrice(): verifica che tutte le righe abbiano la stessa lunghezza (restituisce True o False).***

2. Le Sottoclassi (Specializzazioni)

- ***Ogni membro del gruppo dovrà implementare una delle seguenti classi che ereditano da GestoreMatrice:***

Classe ElaboratoreMatematico(GestoreMatrice) - Deve ereditare costruttore e metodi dalla madre.

- ***Nuovo Metodo: trasponi(): restituisce una nuova matrice in cui le righe diventano colonne e viceversa.***
- ***Nuovo Metodo: moltiplica_per_scalare(k): moltiplica ogni elemento della matrice per il numero k e restituisce la matrice risultante.***

Classe ElaboratoreStatistico(GestoreMatrice) - Deve ereditare costruttore e metodi dalla madre.

- ***Nuovo Metodo: trova_massimo(): restituisce il valore più alto presente nell'intera matrice.***
- ***Nuovo Metodo: media_riga(indice_riga): calcola e restituisce la media aritmetica dei valori di una specifica riga.***

(Se il gruppo è di 3 persone, aggiungere:) - 3. Classe FiltroMatrice(GestoreMatrice)

- ***Nuovo Metodo: azzera_negativi(): sostituisce tutti i numeri negativi nella matrice con 0.***
- ***Nuovo Metodo: appiattisci(): restituisce una singola lista contenente tutti i valori della matrice in sequenza.***

3. Requisiti di Versionamento (Git & GitHub)

Setup Iniziale :

- ***Creare un Repository pubblico su GitHub chiamato gruppoX-matrixtool. (dove X è ciò che preferite)***
- ***Creare il file main.py contenente solo la classe base GestoreMatrice.***
- ***Invitare gli altri membri come Collaborators.***

Sviluppo Parallelo (Tutti i membri):

- ***Ogni membro deve clonare il repository.***
- ***OBBLIGO: Ogni membro deve creare un branch locale nominato con il proprio cognome o la feature (es: feature/statistica o dev-rossi). È vietato lavorare direttamente sul branch main.***
- ***Ogni membro implementa la propria sottoclasse nel file (o in moduli separati, se preferite) lavorando solo sul proprio branch.***

Esercizio di gruppo - Ereditarietà + Incapsulamento – NO astrazione, NO polimorfismo

Obiettivo

In piccoli gruppi da 2 o 3 persone, progettate un semplice gestionale OOP per una gelateria, utilizzando esclusivamente:

- *Incapsulamento (attributi privati + getter/setter)*
- *Ereditarietà tra le classi*
- *Collaborazione tramite GitHub (repository, branch, pull request)*

L'obiettivo è creare un piccolo sistema che gestisca gusti di gelato, gelati speciali e un menu della gelateria.

Requisiti GitHub (OBBLIGATORI)

Ogni gruppo deve:

- *Creare una nuova repository*
- *Aggiungere tutti i membri del gruppo come Collaborators.*
- *Ogni partecipante deve creare un branch personale:*

Ogni persona deve completare TUTTE queste azioni:

- *lavorare sul proprio branch*
- *fare almeno 1 Pull Request*

La repository deve contenere un file README.md con:

- *nomi dei componenti*
- *breve descrizione della suddivisione del lavoro*

1. Classe base: Gusto

Rappresenta un gusto semplice di gelato.
Attributi privati:

- **__nome (stringa)**
- **__prezzo_base (float, prezzo della singola pallina)**
- **__allergeni (lista di stringhe)**

Metodi obbligatori:

- **costruttore __init__(...)**
- **getter e setter per tutti gli attributi**
- **descrizione(): restituisce nome, prezzo e allergeni**

Regola: obbligo di usare incapsulamento → attributi privati con getter/setter.

2. Sottoclassi (Ereditarietà)

A. Classe GustoPremium (deriva da Gusto)
Aggiunge:

- **ingredienti_speciali (lista di stringhe)**
- **sovrapprezzo (float)**

Metodo:

- **descrizione_premium(): restituisce descrizione base + ingredienti speciali + sovrapprezzo**

B. Classe GustoVegano (deriva da Gusto)
Aggiunge:

- **base_vegetale (stringa, es. "soia", "mandorla", "cocco")**

Metodo:

- **descrizione_vegano(): descrizione base + tipo di base vegetale**

3. Classe MenuGelateria

Gestisce l'elenco totale dei gusti disponibili.
Attributi:

- **gusti (lista di gusti, semplici o derivati)**

Metodi:

- **aggiungi_gusto(gusto)**
- **rimuovi_gusto(nome_gusto)**
- **lista_gusti() → stampa elenco formattato**

4. Programma Principale

Il file principale dovrà:

- **creare un oggetto MenuGelateria**
- **creare almeno:**
  - **3 gusti base**
  - **2 gusti premium**
  - **1 gusto vegano**
- **aggiungerli al menu**
- **stampare il menu**
- **rimuovere un gusto**
- **stampare il menu aggiornato**
- **naturalmente tramite un menu**