Profice

AI ACADEMY

Applicare l’Intelligenza Artificiale nello sviluppo software

AI ACADEMY

Python clean coding, GitHub, Colab
16/06/2025

INTRODUZIONE DELL'ISTRUTTORE

Tamas Szakacs

Formazione

- Laureato come programmatore matematico
- MBA in management

Principali esperienze di lavoro

- Amministratore di sistemi UNIX
- Oracle DBA
- Sviluppatore di Java, Python e di Oracle PL/SQL
- Architetto (solution, enterprise, security, data)
- Ricercatore tecnologico e interdisciplinare di IA

Dedicato alla formazione continua

- Teorie, modelli, framework IA
- Ricerche IA
- Strategie aziendali
- Trasformazione digitale
- Formazione professionale

email: tamas.szakacs@proficegroup.it

MOTIVI E RIASSUNTO DEL CORSO

L'**Intelligenza Artificiale (AI)** è oggi il motore dell'innovazione in ogni settore, grazie alla sua capacità di analizzare dati, automatizzare processi e generare nuove soluzioni. Questo corso offre una panoramica completa e pratica sullo sviluppo di applicazioni AI moderne, guidando i partecipanti dall'ideazione al rilascio in produzione.

Attraverso una **combinazione di teoria chiara ed esercitazioni pratiche**, saranno affrontate le tecniche e gli strumenti più attuali: **machine learning, deep learning, reti neurali, Large Language Models (LLM), Transformers, Retrieval Augmented Generation (RAG)** e progettazione di agenti AI.
Le competenze acquisite saranno applicate in progetti concreti, dallo sviluppo di chatbot all'integrazione di modelli generativi, fino al deploy di soluzioni AI in ambienti reali e collaborativi.

Il percorso è pensato per chi vuole imparare a progettare, valutare e integrare sistemi AI di nuova generazione, con particolare attenzione alle best practice di programmazione, collaborazione in team, sicurezza, valutazione delle performance ed etica dell'AI.

DURATA: 17 GIORNI

OBIETTIVI

Il percorso formativo è progettato per **giovani consulenti junior**, con una conoscenza base di programmazione, che stanno iniziando un percorso professionale nel settore AI.

L’obiettivo centrale è fornire una panoramica pratica, completa e operativa sull’intelligenza artificiale moderna, guidando ogni partecipante attraverso tutte le fasi fondamentali.

OBIETTIVI

- Allineare conoscenze AI, ML, DL di tutti i partecipanti
- Saper usare e orchestrare modelli LLM (closed e open-weight)
- Costruire pipeline RAG complete (retrieval-augmented generation)
- Progettare agenti AI semplici con strumenti moderni (LangChain, tool calling)
- Capire principi di valutazione, robustezza e sicurezza dei sistemi GenA
- Migliorare la produttività come sviluppatori usando tool GenAI-driven
- Padroneggiare best practice di sviluppo, versioning e deploy AI
- Introdurre i fondamenti di Graph Data Science e Knowledge Graph
- Ottenere capacità di valutazione dei modelli e metriche
- Comprensione dell'etica e dei bias nei modelli di intelligenza artificiale
- Approfondire le normative di riferimento: AI Act, compliance e governance AI

Il corso è **estremamente pratico** (circa il 40% del tempo in esercitazioni hands-on, notebook, challenge e hackathon), con l'utilizzo di Google Colab, GitHub, e tutti gli strumenti necessari per lavorare su progetti reali e simulati.

STRUTTURA DELLE GIORNATE – PROGRAMMA BREVE

Tutte le giornate sono di 8 ore (9:00-17:00), con 1 ora di pausa suddivisa (mezz'ora pranzo, due pause da 15 min durante la mattina e il pomeriggio).

La progettazione sintetica delle giornate:

Giorno	Tema	Breve descrizione
1	Git & Python clean-code	Collaborazione su progetti reali, versionamento, codice pulito e testato
2	Machine Learning Supervised	Modelli supervisionati per predizione e classificazione
3	Machine Learning Unsupervised	Clustering, riduzione dimensionale, scoperta di pattern
4	Prompt Engineering avanzato	Scrivere e valutare prompt efficaci per modelli generativi
5	LLM via API (multi-vendor)	Uso pratico di modelli LLM via API, autenticazione, deployment
6	Come costruire un RAG	Pipeline end-to-end per Retrieval-Augmented Generation
7	Tool-calling & Agent design	Progettare agenti AI che usano strumenti esterni
8	Hackathon: Agentic RAG	Challenge pratica: chatbot agentico RAG in team

STRUTTURA DELLE GIORNATE – PROGRAMMA BREVE

Tutte le giornate sono di 8 ore (9:00-17:00), con 1 ora di pausa suddivisa (mezz'ora pranzo, due pause da 15 min durante la mattina e il pomeriggio).

La progettazione sintetica delle giornate:

Giorno	Tema	Breve descrizione
9	Hackathon: Rapid Prototyping	Da prototipo a web-app con Streamlit e GitHub
10	AI Productivity Tools	Workflow con IDE AI-powered, automazione e refactoring assistito
11	Docker & HF Spaces Deploy	Deployment di app GenAI containerizzate o su HuggingFace Spaces
12	AI Act & ISO 42001 Compliance	Fondamenti di compliance e governance AI
13	Knowledge Base & Graph Data Science	Introduzione a Knowledge Graph e query con Neo4j
14	Model evaluation & osservabilità	Metriche avanzate, explainability, strumenti di valutazione
15	AI bias, fairness ed etica applicata	Analisi dei rischi, metriche e mitigazione dei bias
16-17	Project Work & Challenge finale	Lavoro a gruppi, POC/POD, presentazione e votazione progetti

METODOLOGIA DEL CORSO

1. Approccio introduttivo ma avanzato

Il corso è introduttivo nei concetti base dell'AI applicata allo sviluppo, ma affronta anche tecnologie, modelli e soluzioni avanzate per garantire un apprendimento completo.

2. Linguaggio adattato

Il linguaggio utilizzato è chiaro e adattato agli studenti, con spiegazioni dettagliate dei termini tecnici per favorirne la comprensione e l'apprendimento graduale.

3. Esercizi pratici

Gli esercizi pratici sono interamente svolti online tramite piattaforme come Google Colab o notebook Python, eliminando la necessità di installare software sul proprio computer.

4. Supporto interattivo

È possibile porre domande in qualsiasi momento durante le lezioni o successivamente via email per garantire una piena comprensione del materiale trattato.

NOTA

Il corso segue un **approccio laboratoriale**: ogni giornata combina sessioni teoriche chiare e concrete con molte attività pratiche supervisionate, per sviluppare *competenze reali* immediatamente applicabili.

I partecipanti lavoreranno spesso in gruppo, useranno notebook in Colab e versioneranno codice su GitHub, vivendo una vera simulazione del lavoro in azienda AI.

Nessun prerequisito avanzato richiesto: si partirà dagli strumenti e flussi fondamentali, con una crescita graduale verso le tecniche più attuali e richieste dal mercato.

ORARIO TIPICO DELLE GIORNATE

Orario	Attività	Dettaglio
09:00 – 09:30	Teoria introduttiva	Concetti chiave, schema della giornata
09:30 – 10:30	Live coding + esercizio guidato	Esempio pratico, notebook Colab
10:30 – 10:45	*Pausa breve*	
10:45 – 11:30	Approfondimento teorico	Tecniche, best practice
11:30 – 12:30	Esercizio hands-on individuale	Sviluppo o completamento di codice
12:30 – 13:00	Discussione soluzioni + Q&A	Condivisione e correzione
13:00 – 13:30	*Pausa pranzo*	
13:30 – 14:15	Teoria avanzata / nuovi tools	Nuovi strumenti, pattern, demo
14:15 – 15:30	Esercizio a gruppi / challenge	Lavoro di squadra su task reale
15:30 – 15:45	*Pausa breve*	
15:45 – 16:30	Sommario teorico e pratico	
16:30 – 17:00	Discussioni, feedback	Riepilogo, best practice, domande aperte

PANORAMICA DELLA GIORNATA E MODALITÀ DI LAVORO

Obiettivi della giornata:

- imparare a **collaborare su progetti Python** tramite Git e GitHub
- scrivere codice leggibile, testabile e facilmente migliorabile
- acquisire le basi di formattazione, refactoring e testing automatico
- sperimentare la revisione del codice in team

Struttura delle attività:

- alternanza di **sessioni teoriche** (slide, spiegazioni, domande) e **sessioni pratiche** (esercizi su Colab, lavoro in gruppo)
- uso di una **code-base "sporca"** che verrà progressivamente migliorata
- lavoro continuo con strumenti reali: **Git, GitHub, Colab, Black, isort, pytest**

PANORAMICA DELLA GIORNATA E MODALITÀ DI LAVORO

Modalità di lavoro:

- ogni concetto sarà seguito da **esercitazioni pratiche supervisionate**
- collaborazione in team tramite branch e pull request su GitHub
- discussione aperta di errori, conflitti e soluzioni in aula
- domande e difficoltà affrontate insieme, con esempi reali

Output atteso a fine giornata:

- repository condiviso con codice pulito, test coperti, docstring inserite
- maggiore sicurezza nel gestire progetti Python in team
- comprensione pratica delle best practice di sviluppo moderno

Risorse utilizzate:

- esercizi guidati dai docenti
- cheat sheet di Git e PEP-8
- tutorial “flight rules” per risolvere problemi reali

DOMANDE?

Cominciamo!

OBIETTIVI DELLA GIORNATA E AGENDA

Obiettivi della giornata

- comprendere i principi base di **Git** e **GitHub**
- imparare a creare e gestire un repository online
- acquisire le procedure fondamentali per collaborare in team su progetti di codice
- sperimentare il flusso di lavoro pratico: commit, branch, merge, pull request
- preparare l'ambiente di lavoro con **Colab**, **Drive** e **GitHub**
- **avere programmi e dati pronti nella propria repository**

OBIETTIVI DELLA GIORNATA E AGENDA

Agenda della giornata

1. introduzione a git e github
- cos'è il versionamento del codice
- differenze tra git, github e drive

2. creazione account e primo repository
- registrazione su github
- creazione di un nuovo repository
- creazione e upload di file

3. comandi e flussi base
- clonazione del repository in colab
- comandi fondamentali: add, commit, push, pull

4. collaborazione e gestione dei conflitti
- uso dei branch
- pull request e code review
- gestione dei conflitti

5. integrazione con colab e drive
- collegare drive a colab
- lavorare su notebook condivisi
- flusso github–colab–drive

6. esercitazione guidata
- esercizi pratici su repository di esercitazione
- domande, dubbi, confronto

COS'È IL CLEAN CODE E PERCHÉ SERVE

Definizione:
Il clean code è codice **chiaro, leggibile e facilmente modificabile** da chiunque, oggi o in futuro.

Perché è importante:
- rende lo sviluppo di gruppo più veloce e meno soggetto a errori
- aiuta a trovare e correggere bug rapidamente
- permette di aggiungere nuove funzionalità senza rompere il codice esistente
- facilita la revisione del codice (pair-review e pull request)

Principi base del clean code:
- **nomi chiari** per variabili, funzioni e classi (es. calcola_totale() invece di ct())
- **funzioni brevi** e che fanno una sola cosa
- **evita duplicazioni**: se un'operazione si ripete, va isolata in una funzione
- **commenti solo dove servono**: il codice deve "spiegarsi da solo"
- **layout ordinato**: spaziature, indentazione, nessun codice "appeso"

COS'È IL CLEAN CODE E PERCHÉ SERVE

Esempio pratico – codice "sporco" vs "pulito":

```python
# Codice sporco
def f(a, b): return a+b

# Clean code
def somma_due_numeri(primo_numero, secondo_numero):
    """Restituisce la somma di due numeri."""
    return primo_numero + secondo_numero
```

Impatto diretto:
Il clean code riduce i tempi di sviluppo e facilita la collaborazione su Git e nelle code review.
Diventa indispensabile quando il progetto cresce e il team si allarga.

In sintesi:
Il clean code non è "codice perfetto", ma codice che chiunque può **capire, usare, modificare e mantenere** senza fatica.

INTRODUZIONE A GIT

Cos’è Git:

- Sistema di controllo versione distribuito, creato per gestire lo sviluppo di progetti software anche molto grandi.
- Permette di tenere traccia di tutte le modifiche, facilitare il lavoro in team e ripristinare versioni precedenti in caso di errori.

Perché si usa:

- Ogni sviluppatore lavora in una copia locale e può sincronizzare i cambiamenti con il repository centrale.
- Essenziale per collaborazione, trasparenza e sicurezza del codice.

Concetti chiave:

- repository
- commit
- branch
- merge
- remote
- clone

GIT FLOW: BRANCH, FORK, MERGE

Branch:
Un branch è una “linea di sviluppo” separata: permette di lavorare su nuove funzionalità senza toccare il codice principale.
Best practice: creare un branch per ogni funzionalità o bugfix.

Fork:
Copia completa del repository, tipica nei progetti open source per proporre modifiche senza accedere direttamente al repo principale.

Merge:
Unisce i cambiamenti di un branch/fork al branch principale, dopo revisione e test.

Diagramma di flusso:

1. Fork → 2. Clone → 3. Crea branch → 4. Sviluppa → 5. Commit → 6. Push → 7. Pull request → 8. Merge

CLONARE UN REPOSITORY: COMANDO E PRATICA

Come clonare un repository remoto:

- Comando di base:

```
git clone https://github.com/nomeutente/nomerepo.git
```

Cosa succede:

- Viene creata una copia locale completa del progetto.
- Si può iniziare subito a lavorare sui file, creare branch ed effettuare modifiche in locale.

Attenzione agli errori comuni:

- Mancata autenticazione
- Repository privato senza permessi
- URL errato

Consiglio:

- Testa subito il clone per verificare la corretta configurazione.

COMMIT E PUSH: BEST PRACTICE

Commit:

- Ogni commit rappresenta una modifica logica e autonoma del codice.
- Scrivere messaggi chiari e descrittivi (“fix: corregge bug login”).
- Evitare commit troppo grandi e generici (“varie modifiche”).

Push:

- Il push invia i commit dal repository locale al repository remoto (GitHub, GitLab, ecc).

- Comando:

```
git push origin nome-branch
```

Buone pratiche:

- Effettua push frequenti per evitare conflitti.
- Sincronizza spesso con il branch remoto (git pull).

COLLABORAZIONE E GESTIONE PROGETTI IN TEAM

Branch naming convention:

- Usa nomi chiari e condivisi (es: feature/login, bugfix/form).

Pull request (PR):

- Meccanismo per proporre cambiamenti e richiedere revisione prima di unire il codice nel branch principale.
- Ogni PR deve essere revisionata almeno da un collega (“pair-review”).

Ruoli nel team:

- Autore del codice, revisore, responsabile del merge.
- Ogni modifica deve essere tracciabile e giustificata nei commenti della PR.

Schema di collaborazione:

Crea branch → sviluppa → commit → push → pull request → revisione → merge

DOMANDE?

Facciamo gli esercizi!

ESERCITAZIONE – GIT: CLONAZIONE, BRANCH, COMMIT

Obiettivo:
Applicare subito i concetti di Git per lavorare su un repository condiviso, creando branch e registrando modifiche con commit ben descritti.

Attività:

1. **Clona il repository di esercitazione fornito dal docente**
   - Comando da eseguire nel terminale o in Colab:

```
git clone https://github.com/tszakacs-ai/ai-academy.git
```

   - Verifica di essere nella cartella corretta (`cd nome-repo-esercizio`)
2. **Crea un nuovo branch personale**
   - Scegli un nome branch chiaro (es: feature/nome-cognome)
   - Comando:

```
cd ai-academy
git checkout -b feature/nome-cognome
```

3. **M**
   - Scrivi il tuo nome e aggiungi una breve descrizione (max 2 righe).

ESERCITAZIONE – GIT: CLONAZIONE, BRANCH, COMMIT

4. **Registra le modifiche con un commit descrittivo**
   - Comando:

```
git add .
git commit -m "feat: aggiunge descrizione personale al README"
```

5. **Esegui il push del branch su GitHub**
   - Comando:

```
git push origin feature/nome-cognome
```

6. **(Facoltativo) Visualizza il branch e verifica sul repository remoto**
   - Controlla su GitHub che il branch sia stato creato correttamente.

Suggerimenti:

- Usa nomi branch sempre comprensibili e uniformi.
- Scrivi messaggi commit chiari, seguendo lo stile “semantic commit”.
- Se incontri errori, chiedi ai docenti o consulta la sezione “flight rules” nelle slide.

Al termine:

Preparati a creare una pull request nella prossima attività.
Se hai tempo, esplora i file del repository e annota dubbi o difficoltà da discutere insieme.

RISOLVERE I CONFLITTI DI MERGE

Cos'è un conflitto di merge:

- Succede quando due persone modificano le stesse righe di un file in branch diversi, poi tentano di unirli.
- Git segnala il conflitto e richiede un intervento manuale.

Come riconoscerlo:

- Durante `git merge` o `git pull`, compare un messaggio:

```
CONFLICT (content): Merge conflict in nomefile.py
```

Come risolverlo:

1. Apri il file segnalato: troverai i marcatori <<<<<<<, =======, >>>>>>>.
2. Decidi quali cambiamenti mantenere (o unisci le parti).
3. Elimina i marcatori e salva il file pulito.
4. Aggiungi il file risolto:

```
git add nomefile.py
```

5. Completa il merge con:

```
git commit
```

Best practice:

- Comunica sempre col team su modifiche parallele.
- Fai pull e merge frequenti, così i conflitti sono più facili da gestire.

ESERCIZIO – CONFLITTO REALE (CASE STUDY BREVE)

Obiettivo:
Simulare e risolvere un conflitto di merge in modo controllato, sperimentando la procedura reale.

Attività guidata:
1. Il docente fornirà un file da modificare (es: `saluta.py`).
2. Ogni partecipante dovrà, sul proprio branch, modificare la stessa riga di quel file (es. aggiungendo una nota personale).
3. Effettua un commit e prova a fare il merge sul branch principale (main/master).
4. Git genererà un conflitto.
5. Risolvilo seguendo le istruzioni della slide precedente.
6. Completa il merge, effettua push e comunica la risoluzione al docente.

Suggerimenti:
- Non avere paura di sbagliare: questa è una situazione normale nello sviluppo reale.
- Lavora in coppia, confronta la tua soluzione con il vicino.
- Annotati la procedura: tornerà utile nelle attività future.

PULL REQUEST E CODE REVIEW

Cos'è una Pull Request (PR):

- Una PR è una richiesta formale per integrare le modifiche fatte in un branch nel branch principale del progetto.
- Consente di avviare una revisione del codice da parte dei colleghi (code review) prima di fare il merge.

Processo di Pull Request:

1. Dopo aver pushato il proprio branch su GitHub, clicca su "Compare & pull request".
2. Descrivi sinteticamente le modifiche fatte e il motivo.
3. Assegna la PR a un revisore (collega o docente).
4. Il revisore può commentare, richiedere modifiche, o approvare.
5. Una volta approvata, si può procedere al merge.

Best practice:

- PR piccole e ben descritte sono più facili da revisionare.
- Commenta le scelte non ovvie.
- Non fare merge diretto senza almeno una revisione (pair-review).

Schema collaborazione:

Sviluppo → commit → push → PR → review → merge

GIT WORKFLOW – CHEATSHEET

FASE	COMANDO	COSA FA
1. Clona il progetto	git clone <url-repo>	Crea copia locale della repo GitHub
2. Cambia cartella	cd nome-repo	Vai nella cartella del progetto
3. Aggiorna dal server	git pull origin main	Scarica novità dal branch remoto principale
4. Crea nuovo branch	git checkout -b feature/nome	Nuova linea di lavoro (opzionale, consigliato)
5. Lavora sui file	(Modifica/aggiungi file…)	
6. Aggiungi cambiamenti	git add nomefile o git add .	Scegli quali file vuoi salvare
7. Salva cambiamenti	git commit -m "Messaggio chiaro"	Crea “snapshot” locale del lavoro
8. Aggiorna di nuovo	git pull origin main	(Facoltativo, consigliato prima di push)
9. Invia su GitHub	git push origin <nome-branch>	Carica il tuo lavoro sul server
10. (Facoltativo) PR	(Su GitHub: “New Pull Request”)	Chiedi revisione/merge del tuo branch

DOMANDE?

PAUSA

SEMANTIC COMMIT: COSA SONO E PERCHÉ USARLI

Definizione:
Un semantic commit è un messaggio di commit strutturato che segue uno standard e rende la cronologia del progetto leggibile e automatizzabile.

Struttura tipica:

```
<tipo>: descrizione breve
```

Esempi di tipi:

```
feat, fix, docs, refactor, test, chore
```

Perché sono utili:
Permettono di capire subito cosa è cambiato e perché
Facilitano l'automatizzazione di changelog e release
Agevolano la collaborazione e la code review

Esempi:

```
feat: aggiunge autenticazione Google
fix: corregge bug su login
docs: aggiorna README
```

CONVENTIONAL COMMITS: SINTASSI E STANDARD

Cosa sono:

Una convenzione più completa e diffusa per scrivere semantic commit, usata da molte community e tool di CI/CD.

Sintassi base:

```
<tipo>(opzionale-scope): descrizione breve
```

Esempio con scope:

```
feat(login): aggiunge captcha
refactor(api): semplifica endpoint utenti
test(utils): aggiunge test edge cases
```

Regole pratiche:

- Scrivere descrizione in forma imperativa e breve
- Evitare maiuscole e punti finali nella descrizione
- Usare scope solo se necessario (modulo, funzione, feature)

Errore comune:

- Commit vaghi o troppo generici:
  update stuff → ❌
  fix: risolve crash su ricerca → ✅

ESEMPI PRATICI DI SEMANTIC COMMIT

Tipi principali da usare:

- feat: aggiunta di nuove funzionalità
- fix: correzioni di bug
- docs: modifiche alla documentazione
- style: formattazione (spazi, punti e virgola…)
- refactor: riorganizzazione del codice senza aggiungere feature
- test: aggiunta o modifica di test
- chore: modifiche a build, strumenti, librerie

```
git add .
git commit -m "feat(data):
aggiunto preprocessing dati"
```

Esempi reali:

```
feat: implementa filtro ricerca avanzata
fix: sistema bug su validazione email
docs: completa guida setup progetto
style: applica black a tutto il codice
refactor(login): separa funzioni di autenticazione
test: aggiunge test per edge cases login
chore: aggiorna dipendenze requirements.txt
```

Quando usarli:

- Ogni commit dovrebbe seguire uno di questi tipi, per una storia chiara e un workflow efficace su GitHub e nelle code review.

GITHUB ACTIONS: AUTOMAZIONE SU PUSH

Cos'è GitHub Actions:

- Sistema integrato in GitHub per **automatizzare task** legati al ciclo di vita del codice: test, lint, build, deploy, ecc.
- Le azioni vengono definite tramite file .yml all'interno della cartella .github/workflows/.

Automazioni tipiche su push:

- Eseguire il linting del codice (controllo stile, errori base)
- Lanciare test automatici (pytest)
- Bloccare merge se test o lint falliscono

Vantaggi:

- Garantisce che il codice sia sempre controllato prima di essere integrato
- Rende il processo di sviluppo più sicuro, rapido e ripetibile

Esempio di trigger:

- "on: push" → l'azione parte automaticamente a ogni push su un branch

WORKFLOW: LINT + PYTEST SU ACTIONS

Esempio di file workflow (ci.yml):

```yaml
name: CI

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'
      - name: Install dependencies
        run: pip install -r requirements.txt
      - name: Lint with black
        run: black --check .
      - name: Test with pytest
        run: pytest
```

Spiegazione step principali:

- **checkout**: scarica il codice del repo
- **setup-python**: imposta la versione di Python
- **install dependencies**: installa pacchetti necessari
- **lint with black**: controlla formattazione
- **test with pytest**: esegue i test automatici

Risultato atteso:

- Se il lint o i test falliscono, la pull request viene bloccata finché non viene sistemato il codice.

Best practice:

- Automatizza sempre il controllo del codice prima del merge
- Mantieni il workflow aggiornato con le librerie del progetto

DOMANDE?

Facciamo gli esercizi!

ESERCITAZIONE – PULL REQUEST E AUTOMAZIONE

Obiettivo:

Mettere in pratica la creazione di una pull request (PR), attivare l'automazione tramite GitHub Actions, e sperimentare la revisione del codice tra pari.

Attività:

1. **Crea un nuovo branch dal branch principale:**
   - `git checkout -b esercizio-pr`
   - Aggiorna o crea un file Python secondo le istruzioni fornite (ad esempio, aggiungi una funzione con docstring NumPy-style).
   - Esegui git add, git commit (usa semantic commit), e git push sul tuo branch.
     - `git add .`
     - `git commit -m "modifica di esempio per PR"`
     - `git push origin esercizio-pr`
2. **Crea una pull request su GitHub → Pull Requests → New Pull Request**
   - Accedi al repository remoto su GitHub.
   - Clicca su "Compare & pull request".
   - Compila il titolo e la descrizione in modo chiaro.
   - Assegna la PR a un compagno o docente come revisore.

ESERCITAZIONE – PULL REQUEST E AUTOMAZIONE

3. **Verifica che il workflow automatico si attivi (opzionale)**
   - Controlla nella sezione “Actions” di GitHub che la pipeline parta (lint e pytest).
   - Se la pipeline fallisce, leggi i log, correggi gli errori, aggiorna il branch e ripeti il push.
4. **Attendi la revisione (opzionale)**
   - Leggi eventuali commenti, apporta modifiche se richiesto, discuti con il revisore.
   - Una volta approvato, procedi al merge (o chiedi che venga eseguito).

Suggerimenti:

- Scrivi titoli PR precisi e descrizioni complete: facilita la revisione.
- Usa sempre i messaggi semantic commit.
- Se la pipeline automatica fallisce, chiedi aiuto: l’obiettivo è imparare dai problemi reali!

Al termine:

- Segna a parte le difficoltà riscontrate e discuti in gruppo le soluzioni.
- Assicurati che la PR sia stata integrata correttamente e che i test siano verdi prima di considerare completata l’esercitazione.

ANATOMIA DI UN MODULO PYTHON

Cos’è un modulo Python:

- Un modulo è un file `.py` che contiene funzioni, classi e variabili utilizzabili in altri file tramite import.
- Ogni progetto Python è composto da uno o più moduli, spesso organizzati in pacchetti.

Elementi fondamentali di un modulo:

- Definizioni di funzioni e classi
- Costanti e variabili
- Import di altri moduli
- Blocchi speciali per esecuzione diretta

Vantaggi dell’organizzazione modulare:

- Favorisce il riutilizzo del codice
- Rende il progetto scalabile e mantenibile
- Permette di testare e aggiornare parti specifiche senza impatti globali

INIT E MAIN IN PRATICA

init.py:
File speciale presente in ogni cartella di un pacchetto Python.
Permette di trattare la cartella come un pacchetto importabile.
Può essere vuoto o contenere codice di inizializzazione.

main:
Blocca di codice che permette di distinguere l'esecuzione diretta di un file dall'import in altri moduli.
Struttura tipica:

```
if __name__ == "__main__":
    funzione_principale()
```

Permette di scrivere moduli riutilizzabili e script eseguibili.

Esempio pratico:

```
def saluta():
    print("Ciao!")

if __name__ == "__main__":
    saluta()
```

DUNDER METHODS: ALTRI CASI COMUNI

Cos'è un "dunder method":
"Dunder" = double underscore, es: `__init__, __str__`

Sono metodi speciali che Python chiama automaticamente per specifici comportamenti.

Esempi più usati:
- `__init__`: inizializza un oggetto (costruttore)
- `__str__`: rappresentazione leggibile a stampa (str(oggetto))
- `__repr__`: rappresentazione tecnica per debug
- `__len__`: restituisce la lunghezza (come con len())
- `__getitem__`: accesso a elementi con l'indicizzazione (obj[i])
- `__main__`: identifica il punto di ingresso principale di uno script

Perché usarli:
- Permettono di personalizzare il comportamento di oggetti, moduli e script Python.
- Rendono il codice più potente, leggibile e Pythonic.

DISCUSSIONE APERTA – DOMANDE E RISPOSTE

Obiettivo della sessione:
Favorire il confronto su quanto visto finora, chiarire dubbi, condividere difficoltà pratiche e raccogliere feedback diretto.

Spunti per la discussione:

- Hai riscontrato **difficoltà pratiche** nell'uso di Git o durante i primi esercizi?
- Ci sono passaggi della **collaborazione su GitHub** che ti sono sembrati poco chiari?
- **Quali differenze** hai notato tra il lavoro individuale e quello in team con branch e PR?
- Hai suggerimenti su come migliorare la **scrittura di commit o messaggi di PR**?
- Pensi che la struttura "modulare" del codice Python sia facile da applicare ai tuoi progetti reali?
- Quale parte del flusso (branch, merge, conflitti, PR) ti mette più in difficoltà?
- Vorresti approfondire qualche aspetto visto questa mattina?

DISCUSSIONE APERTA – DOMANDE E RISPOSTE

Domande utili :

- C’è qualcosa che ripeteresti da solo per esercitarti?
- Ti sono chiari i benefici del clean code nella collaborazione?
- Ti è chiaro quando usare branch e quando lavorare direttamente su main?
- Hai domande sui dunder methods o su come strutturare i moduli?
- Vuoi esempi aggiuntivi su semantic commit o code review?

Invito attivo:

- Nessuna domanda è “banale”: spesso un dubbio condiviso aiuta tutta la classe.
- Prenditi un minuto per rileggere i tuoi appunti e chiedi anche sulle piccole incertezze.

DOMANDE?

PAUSA PRANZO

FORMATTAZIONE DEL CODICE

Cos’è Black:

- Un formattatore automatico di codice Python.
- Applica regole di stile uniformi e non negoziabili, facilitando la lettura e la collaborazione nei progetti di team.

Perché usarlo:

- Elimina discussioni sullo stile del codice: il formato viene sempre “normalizzato”.
- Garantisce che il codice sia sempre leggibile, senza errori dovuti a spazi o indentazione.

Comandi base:

- Formatta un file: `black nomefile.py`
- Controlla il codice senza modificarlo: `black --check nomefile.py`

Quando usarlo:

- Prima di ogni commit (consigliato)
- Nei workflow automatici (GitHub Actions)

ISORT: ORDINARE GLI IMPORT IN AUTOMATICO

Cos’è isort:

- Un tool che ordina automaticamente le istruzioni import in un file Python secondo regole predefinite.

Perché usarlo:

- Mantiene il codice ordinato e previene errori da import duplicati o in ordine sbagliato.
- Semplifica la revisione del codice e la risoluzione dei conflitti in team.

Comandi base:

- Ordina gli import in un file: `isort nomefile.py`
- Ordina tutti i file in una cartella: `isort .`

Best practice:

- Integrare `isort` nei workflow di sviluppo e nelle pipeline di CI.

DOCSTRING NUMPY-STYLE: TEMPLATE ED ESEMPI

Cos’è una docstring NumPy-style:
Una docstring formattata secondo lo standard NumPy, chiara e strutturata, ideale per generare documentazione automatica con strumenti come Sphinx.

Template base:

```
def somma(a, b):
    """
    Calcola la somma di due numeri.

    Parameters
    ----------
    a : int or float
        Primo numero da sommare.
    b : int or float
        Secondo numero da sommare.

    Returns
    -------
    int or float
        La somma di a e b.
    """
    return a + b
```

Vantaggi:

- La documentazione è chiara, completa e facilmente utilizzabile da tutto il team.
- Agevola il supporto di strumenti di analisi e generazione doc.

DOCSTRING PER SPHINX: QUICKSTART

Cos'è Sphinx:
Uno strumento che permette di generare documentazione leggibile a partire dalle docstring presenti nel codice Python.
Passaggi per usarlo:

1. Installa Sphinx: `pip install sphinx`

2. Crea la struttura della documentazione:

```
sphinx-quickstart
```

3. Configura l'estensione autodoc nel file di configurazione.
4. Genera la documentazione HTML:

```
make html
```

5. Visualizza la documentazione generata nel browser (build/html/index.html).

Best practice:

- Usa sempre docstring NumPy-style nei tuoi moduli.
- Rigenera la documentazione ogni volta che aggiorni funzioni o classi.

ESERCITAZIONE – FORMATTAZIONE AUTOMATICA

Obiettivo:
Applicare la formattazione automatica del codice Python con **Black** e **isort**, aggiungere docstring in stile NumPy alle funzioni.

Attività step-by-step:

1. **Apri un file Python a tua scelta** dal repository di esercitazione
   (oppure usa il file utils.py fornito dai docenti).
2. **Applica isort per ordinare gli import**
   - Comando: `isort utils.py`
   - Controlla che tutti gli import siano ordinati (standard, di terze parti, locali).
3. **Applica Black per formattare il codice**
   - Comando: `black utils.py`
   - Il file verrà riformattato secondo lo standard Black.

ESERCITAZIONE – FORMATTAZIONE AUTOMATICA

4. **Individua almeno 2 funzioni nel file**
   - Se non ci sono, crea due semplici funzioni (esempi sotto).
5. **Aggiungi una docstring NumPy-style a ciascuna funzione**
   - Segui il template fornito nell'esempio.
6. **Verifica che il codice sia leggibile e la docstring corretta**
   - Chiedi al docente o a un compagno un parere sulla chiarezza della documentazione.

Suggerimenti:

- Usa sempre il template NumPy-style per tutte le funzioni (anche se sono semplici).
- Dopo la formattazione, confronta il file prima e dopo per vedere le differenze.
- Se hai dubbi sulla docstring, confrontala con l'esempio proiettato in aula.

Al termine:

Condividi il file aggiornato con il docente via pull request o in team, pronto per la revisione.

REFACTORING: CODE SMELLS COMUNI

Cos’è un code smell:

- Un segnale che indica potenziali problemi nel codice, senza essere per forza un bug.
- Rende il codice difficile da capire, modificare e mantenere.

Esempi di code smells frequenti:

- Funzioni troppo lunghe
- Duplicazione di codice
- Variabili dai nomi poco chiari
- Codice “morto” (mai usato)
- Strutture di controllo troppo annidate

Impatto sul progetto:

- Aumenta il rischio di errori
- Rallenta lo sviluppo
- Complica la collaborazione in team

REFACTORING: FUNZIONI TROPPO LUNGHE

Problema:

- Una funzione con troppe responsabilità è difficile da testare, leggere e riutilizzare.

Come riconoscerla:

- Più di 15–20 righe
- Fa più di una cosa (ad esempio: legge dati, li trasforma e stampa a schermo)

Come migliorare:

- Spezzare la funzione in sotto-funzioni con compiti precisi
- Assegnare nomi chiari alle nuove funzioni

Esempio:

```
# Funzione troppo lunga
def processa_ordini(lista):
    # ... 40 righe di codice ...

# Refactoring
def leggi_ordini():
    # ...
def calcola_totale():
    # ...
def stampa_report():
    # ...
```

CODICE DUPLICATO: PERCHÉ È UN PROBLEMA

Problema:

- Stesso blocco di codice ripetuto in più punti del progetto.

Conseguenze:

- Se serve correggere un bug, bisogna modificarlo ovunque (alto rischio di errori)
- Maggiore difficoltà nel fare manutenzione

Come agire:

- Raccogliere il codice duplicato in una funzione o classe comune
- Usare moduli condivisi

Esempio:

```
# Duplicazione
if user.is_admin:
    print("Accesso admin")
# ... altrove ...
if user.is_admin:
    print("Accesso admin")

# Soluzione
def mostra_accesso_admin():
    print("Accesso admin")
```

ALTRI CODE SMELL TIPICI

Esempi da riconoscere subito:

- Variabili o funzioni dai nomi generici (data, temp, foo)
- Parametri inutilizzati nelle funzioni
- Strutture di controllo annidate oltre 2 livelli (if dentro if dentro if)
- Codice commentato (“codice morto”) lasciato nel file
- Funzioni che modificano variabili globali senza motivo

Per ognuno:

- Chiediti se c’è un modo più chiaro o semplice
- Chiedi un confronto al team per trovare soluzioni condivise

ESERCIZIO – CODE-BASE “SPORCA” DA RIPULIRE

Obiettivo:
Applicare i principi di clean code per identificare e correggere code smell in un programma Python reale.
Individuare code smell reali in un file fornito (“code-base sporca”), proporre e implementare refactoring.

Attività:

1. **Scarica o copia il codice “sporco” fornito dal docente** (vedi esempio precedente).
2. **Analizza il codice e identifica tutti i problemi**:
   - Funzioni troppo lunghe
   - Duplicazione
   - Nomi generici
   - Codice morto/commentato
   - Variabili e parametri inutilizzati
   - Variabili globali
   - Strutture di controllo troppo annidate
   - Stampa diretta nelle funzioni “core”
3. **Pianifica il refactoring**:
   - Scrivi una breve lista delle modifiche che intendi apportare (puoi lavorare in team).
4. **Riscrivi il programma, suddividendo la logica in funzioni chiare e brevi, usando nomi esplicativi e docstring NumPy-style**.
5. **Confronta il tuo risultato con la versione pulita proposta** (fornita dal docente).
6. **Effettua commit semantic per ogni modifica importante** e documenta brevemente ogni refactoring.
7. **Estendi il programma con main().**

SOLUZIONE – CODE-BASE “SPORCA” DA RIPULIRE

Suggerimenti:

- Lavora in modo iterativo: correggi, poi rileggi il codice insieme.
- Lavora in gruppo: confronta le scelte di refactoring.
- Confronta la versione “prima” e “dopo” il refactoring.
- Spiega il perché di ogni cambiamento nella descrizione del commit.
- Se non sei sicuro su una soluzione, chiedi chiarimenti o cerca esempi online.

Al termine:

- Carica la versione refactorata sul branch del tuo team o sulla tua cartella.
- Prepara una breve presentazione delle scelte fatte.

SOLUZIONE – CODE-BASE “SPORCA” DA RIPULIRE

Cosa deve essere fatto e come è stato migliorato:

- **Ogni funzione ha un nome descrittivo e svolge UNA SOLA responsabilità** (SRP: Single Responsibility Principle).
- **Nessuna variabile globale:** tutto viene passato come parametro.
- **Nessuna duplicazione:** il calcolo principale è in una funzione riutilizzabile, la stampa e l’analisi sono separate.
- **Codice morto eliminato.**
- **Nessuna stampa diretta in funzioni “core” che dovrebbero solo restituire valori (esclusa l’analisi/report).**
- **Tutte le funzioni hanno docstring NumPy-style.**
- **Funzioni brevi, senza annidamenti inutili.**
- **Le attività di input/output sono sempre separate dalla logica di calcolo.**
- **Sezione main per l’esecuzione controllata (best practice Python).**

DOMANDE?

PAUSA

TESTING UNITARIO: PRINCIPI BASE

Cos'è il testing unitario:

- È la pratica di scrivere piccoli test automatici che verificano che singole funzioni o metodi ("unità di codice") si comportino come previsto.
- Ogni test copre un caso specifico, ad esempio un input atteso o un comportamento da verificare.

Perché è fondamentale:

- Previene regressioni (vecchi bug che ritornano)
- Facilita il refactoring: se il codice è testato, puoi modificarlo senza paura
- Rende il codice più affidabile e documentato

Esempio semplice:

```
def somma(a, b):
    return a + b

def test_somma():
    assert somma(2, 3) == 5
```

PYTEST: STRUTTURA DI BASE

Cos'è pytest:

- È uno dei framework più diffusi e potenti per il testing in Python.
- Permette di scrivere test semplici come funzioni e di automatizzarne l'esecuzione.

Come si scrive un test con pytest:

- Crea un file che inizia per test_ (es: test_utils.py)
- Scrivi funzioni di test che iniziano per test_
- Usa assert per verificare i risultati attesi

Esempio:

```
def moltiplica(a, b):
    return a * b

def test_moltiplica():
    assert moltiplica(2, 3) == 6
    assert moltiplica(0, 5) == 0
```

Per eseguire i test:

```
pytest
```

FIXTURE PARAMETRICHE: COME E PERCHÉ

Cos’è una fixture:

- Una funzione speciale di pytest che prepara dati o contesto per i test (setup automatico).

Perché usarle:

- Permettono di riutilizzare dati e logica tra più test
- Rendono i test più puliti e meno ripetitivi

Cos’è una fixture parametrica:

- Permette di eseguire lo stesso test con tanti valori diversi automaticamente

Esempio:

```python
import pytest

@pytest.mark.parametrize(
    "a, b, expected", [
        (2, 3, 5),
        (0, 0, 0),
        (-1, 1, 0)
    ]
)
def test_somma(a, b, expected):
    assert somma(a, b) == expected
```

TEST COVERAGE E REPORT

Cos'è la copertura dei test ("coverage"):

- Misura quale percentuale del codice è stata effettivamente eseguita dai test.
- Un coverage alto significa che il codice è ben controllato dai test, ma attenzione: la qualità dei test conta più della sola quantità.

Come si misura:

- Usa il tool coverage.py:

```
coverage run -m pytest
coverage report
coverage html  # genera un report visivo navigabile
```

Best practice:

- Punta a coprire le funzioni critiche e i casi limite
- Aggiorna i test ogni volta che modifichi la logica di una funzione
- Usa i report per identificare parti di codice senza test

ESERCITAZIONE – TEST UNITARI E PAIR-REVIEW

Obiettivo:

Scrivere test automatici con **pytest** per le funzioni del file utils.py (o altro file usato in G1E3), collaborando in team per la revisione reciproca.

Attività:

1.Crea (o apri) il file di test

- Nome consigliato: `test_utils.py`
- Salva il file nella stessa cartella di `utils.py`

2.Scegli almeno 2 funzioni da testare

- Prendi le funzioni (es. moltiplica, dividi)
- Puoi aggiungere funzioni extra se vuoi approfondire

3.Scrivi una funzione di test per ciascuna

- Nome funzione: deve iniziare con `test_`
- Usa diversi valori di input, compresi casi limite (es. divisione per zero)

4.(Facoltativo) Usa pytest.mark.parametrize

- Per provare la stessa funzione con vari dati in automatico

5.Esegui i test

- Da terminale o da Colab, lancia: `pytest`

ESERCITAZIONE – TEST UNITARI E PAIR-REVIEW

Profice

6. **Correggi eventuali errori**
- Se un test fallisce, analizza l'errore e correggi la funzione/documentazione se necessario
7. **Pair-review in classe**
- Scambia il file con un compagno: ognuno legge, commenta e suggerisce miglioramenti sui test dell'altro
- Valuta chiarezza, copertura casi, presenza di docstring
8. **Commit e push finale**
- Effettua commit semantic (test: aggiunti test su moltiplica/dividi)
- Push su branch personale o di gruppo

```python
import pytest
from utils import moltiplica, dividi

def test_moltiplica():
    assert moltiplica(2, 3) == 6
    assert moltiplica(0, 5) == 0
    assert moltiplica(-2, 4) == -8

@pytest.mark.parametrize(
    "a, b, expected", [
        (10, 2, 5),
        (9, 3, 3),
        (0, 1, 0),
    ]
)
def test_dividi(a, b, expected):
    assert dividi(a, b) == expected

def test_dividi_zero():
    with pytest.raises(ValueError):
        dividi(5, 0)
```

ESERCITAZIONE – TEST UNITARI E PAIR-REVIEW

Supporto per gli studenti:

- Se hai dubbi sulla struttura, usa l'esempio qui sopra come punto di partenza.
- Puoi confrontarti con i docenti o chiedere feedback sui tuoi test prima di committare.
- Non preoccuparti di sbagliare: l'obiettivo è imparare a scrivere test robusti e a revisionare quelli degli altri.

```python
import pytest
from utils import moltiplica, dividi

def test_moltiplica():
    """Test della funzione moltiplica con
casi base e negativi."""
    assert moltiplica(2, 3) == 6
    assert moltiplica(0, 5) == 0
    assert moltiplica(-2, 4) == -8

@pytest.mark.parametrize(
    "a, b, expected", [
        (10, 2, 5),
        (9, 3, 3),
        (0, 1, 0),
        (-4, 2, -2)
    ]
)
```

ESERCITAZIONE – TEST UNITARI E PAIR-REVIEW

Al termine:

- Tutti i test devono essere verdi (PASSED).
- Discuti col team eventuali scelte o difficoltà incontrate.
- Condividi osservazioni e miglioramenti emersi dalla pair-review con la classe.

```python
def test_dividi(a, b, expected):
    """Test parametrici per la funzione dividi con vari valori."""
    assert dividi(a, b) == expected

def test_dividi_zero():
    """Test della gestione dell'eccezione divisione per zero."""
    with pytest.raises(ValueError):
        dividi(5, 0)
```

Spiegazione della soluzione:

- **Test delle funzioni** con valori positivi, zero, negativi.
- **Uso di pytest.mark.parametrize** per automatizzare test multipli con diversi input/output.
- **Test dell'eccezione** per la divisione per zero (ValueError).
- **Docstring chiara** per ogni funzione di test, per spiegare lo scopo.
- **Tutti i test possono essere eseguiti da terminale con** bash

PAIR-REVIEW: OBIETTIVI E MODALITÀ

Cos’è il pair-review:

- È il processo in cui due sviluppatori si scambiano e revisionano reciprocamente il codice, prima di integrarlo nel repository principale.

Obiettivi principali:

- Migliorare la qualità e la leggibilità del codice
- Individuare errori o potenziali miglioramenti prima del merge
- Favorire la condivisione di conoscenze e buone pratiche all’interno del team

Modalità di svolgimento:

- Ognuno legge e commenta il codice del collega, suggerendo modifiche o segnalando dubbi
- Le osservazioni vengono discusse e concordate
- Solo dopo il confronto e la correzione si procede con l’approvazione della pull request

Best practice:

- Essere costruttivi e specifici nei feedback
- Motivare sempre i suggerimenti (no a “non mi piace”, sì a “questo nome è poco chiaro”)
- Usare checklist di revisione condivise dal team

COMMENTI E APPROVAZIONE FINALE

Il ciclo dei commenti in pull request:

- Ogni osservazione va inserita direttamente sulla riga interessata nel diff del codice
- Il codice deve essere migliorato sulla base dei commenti ricevuti

Quando approvare:

- Solo se tutti i punti aperti sono risolti
- Se il codice rispetta le convenzioni, è chiaro e testato

Ruoli:

- Chi scrive la PR risponde ai commenti e applica i suggerimenti
- Il revisore controlla che le modifiche siano effettivamente implementate

Approvazione:

- Una PR può essere fusa (merge) solo dopo almeno un’approvazione formale
- L’approvazione va documentata nel sistema (GitHub, GitLab, ecc.)

LAVORO ATTESO: REPOSITORY PULITO

Cosa deve esserci nel repository a fine giornata:

- Tutte le funzioni e i file ripuliti secondo le regole del clean code
- Commit semantic e ordinati
- Docstring NumPy-style presenti in ogni funzione
- Tutti i test unitari devono passare (“verdi”)
- Pull request riviste e approvate in team
- Nessun codice duplicato o morto
- Formattazione automatica applicata (Black, isort)
- Checklist finale compilata

Come verificarlo:

- Usa il comando pytest per verificare che i test siano tutti ok
- Naviga il repository: deve essere facile capire cosa fa ogni file/funzione
- Rivedi la storia dei commit: deve essere leggibile e ordinata

GIT FLIGHT RULES: COSA SONO E COME USARLE

Definizione:

- Raccolta di regole e “soluzioni pronte” per i problemi tipici che si incontrano usando Git (errori di merge, commit sbagliati, rebase, recupero file cancellati…)

Dove trovarle:

- GitHub – Git Flight Rules
- Documentazione consultabile in tempo reale durante lo sviluppo

Quando usarle:

- Ogni volta che incontri un errore non previsto o una situazione che non sai risolvere
- Come risorsa di auto-apprendimento continua

PEP8 CHEAT SHEET: REGOLE DI STILE PYTHON

Cos’è PEP8:

- Lo standard ufficiale di stile per il codice Python
- Definisce regole su nomi, indentazione, lunghezza righe, spazi, commenti, ecc.

Perché usarlo:

- Rende il codice uniforme tra progetti e team diversi
- Facilita la lettura e la collaborazione
- Strumenti come Black e isort applicano automaticamente le regole PEP8

Cheat sheet (estratto):

- Indenta con 4 spazi
- Limite di 79 caratteri per riga
- Spazi attorno agli operatori
- Una riga vuota tra funzioni
- Nomi chiari per variabili e funzioni

Risorsa utile:

- PEP8 – Guida breve ufficiale

RIASSUNTO DELLA GIORNATA

Cosa abbiamo imparato oggi:

- Fondamenti di clean code e collaborazione su progetti Python reali
- Utilizzo pratico di Git: branch, commit, PR, merge, risoluzione conflitti
- Automatizzazione delle revisioni con GitHub Actions
- Scrittura e refactoring di codice leggibile, documentato e testato
- Pair-review e approvazione condivisa
- Formattazione automatica e docstring NumPy-style
- Esercizi reali su code-base “sporca” e testing automatico

PROSSIMI PASSI E MOTIVAZIONE

Cosa portiamo a casa:

- Le basi per scrivere codice Python di qualità, in team
- La sicurezza di poter affrontare un progetto collaborativo con strumenti e regole moderne
- Una metodologia concreta per crescere come sviluppatore e collega

Come continuare:

- **Controlla i programmi e dati pronti nella propria repository**
- Ripassa i materiali e prova a ripetere a casa gli esercizi
- Usa GitHub, Black, pytest nei tuoi progetti
- Consulta flight rules e PEP8 ogni volta che serve
- Non smettere di confrontarti: il codice migliore si scrive sempre insieme

Domande finali? Feedback?

- Lo spazio è aperto per le ultime domande e per suggerire come migliorare le prossime giornate!

GRAZIE PER L'ATTENZIONE