Some Utils

Decoratori

Perché i decoratori sono importanti

- Separano **logica principale** e **logica trasversale**.
- Evitano duplicazione di codice.
- Rendono il codice più leggibile, modulare e riutilizzabile.
- Collegano la programmazione funzionale e la OOP.

Decoratori

I decoratori sono uno strumento potente ed espressivo in Python che consente ai programmatori di modificare il comportamento di una funzione o di una classe. Sono utilizzati per racchiudere un'altra funzione al fine di estenderne il comportamento, senza modificarla in modo permanente.

Giacomo Di Salvo

Decoratori

Un decoratore in Python è essenzialmente una funzione che accetta un'altra funzione come argomento e ne estende il comportamento senza modificarla esplicitamente. Sono rappresentati dal simbolo @ e vengono posizionati sopra la definizione della funzione che si desidera decorare.

Giacomo Di Salvo

Decoratori

In Python le funzioni:

- Sono **oggetti**.
- Possono essere passate come argomenti.
- Possono essere restituite come valori.

```
•def esterna():
•    def interna():
•        return "Sono interna"
•    return interna
•x = esterna()
•print(x())  # "Sono interna"
```

Decoratori

Higher-Order Functions (HOF) (teoria)ù

- Definizione: funzioni che prendono altre funzioni come input o le restituiscono come output.
- Questo concetto è la **base dei decoratori**.

•def esegui(funzione):

- print(funzione())

```
def prova():
    return "Funzione passata come argomento!"

esegui(prova)
```

Decoratori

Lo abbiamo già definito

Un decoratore è:

Una funzione che riceve una funzione e restituisce una nuova funzione arricchita/modificata.

Decoratori

Sintassi dei decoratori

- **Forma esplicita**:

saluta = decoratore(saluta)

Forma pythonistica:

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

Decoratori

Primo esempio di decoratore

Esempio pratico:

```
def decoratore(funzione):
    def wrapper():
        print("Prima")
        funzione()
        print("Dopo")
    return wrapper

@decoratore
def saluta():
    print("Ciao!")

saluta()
```

Decoratori

Casi d’uso comuni

- Logging
- Misurazione dei tempi
- Controllo accessi
- Caching
- Validazione

Decoratori

Esempio: misurazione del tempo

Dimostrazione pratica:

```
import time

def timer(funzione):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = funzione(*args, **kwargs)
        end = time.time()
        print(f"Tempo: {end-start:.2f} sec")
        return result
    return wrapper

@timer
def lavoro():
    time.sleep(1)
```

Decoratori

Decoratori multipli

- Più decoratori possono avvolgere una funzione.
- Ordine: quello più vicino alla funzione è applicato per primo.

Decoratori

Esempi reali

Ecco alcuni esempi concreti di decoratori Python, che vanno da applicazioni semplici a quelle più complesse, per illustrarne l'utilità in scenari reali:

Decoratore di logging

Un semplice decoratore di registrazione può essere utilizzato per registrare i punti di ingresso e di uscita delle funzioni, il che è utile per il debug e il monitoraggio del flusso dell'applicazione.

Decoratori

```
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Function {func.__name__} returned {result}")
        return result
    return wrapper

@logger
def add(x, y):
    return x + y

add(5, 3)
```

Decoratori

Controllo degli accessi

È possibile utilizzare un decoratore per applicare il controllo degli accessi a determinate funzioni, limitandone l'esecuzione in base ai ruoli o alle autorizzazioni degli utenti.

Decoratori

```
def admin_required(func):
    def wrapper(user, *args, **kwargs):
        if user != 'admin':
            raise Exception("This function requires admin privileges")
        return func(*args, **kwargs)
    return wrapper

@admin_required
def delete_database():
    print("Database deleted!")

delete_database('admin')  # Works
delete_database('guest')  # Raises Exception
```

Decoratori

Decoratori nelle classi

- Non solo funzioni!

- Esempi integrati in Python:
  - `@staticmethod`
  - `@classmethod`
  - `@property`

Decoratori

@staticmethod:

- Trasforma un metodo in **metodo statico**.
- Non riceve né l'istanza (`self`) né la classe (`cls`) come primo argomento.
- È essenzialmente una **funzione normale** "ospitata" all'interno della classe, utile per **ragioni di organizzazione**.
- Può essere chiamato sia sulla classe sia sull'istanza.

Decoratori

```
class Matematica:
    @staticmethod
    def addizione(a, b):
        return a + b

# Chiamata su classe
print(Matematica.addizione(2, 3))  # 5

# Chiamata su istanza
m = Matematica()
print(m.addizione(4, 6))  # 10
```

Quando usarlo: se la funzione non ha bisogno di accedere né ai dati della classe né dell’istanza.

Decoratori

`@classmethod:`

- Trasforma un metodo in **metodo di classe**.
- Riceve come primo argomento la **classe stessa** (`cls`), non l'istanza.
- Utile quando vuoi scrivere metodi che lavorano **sulla classe nel suo insieme** (es. factory methods, alternative constructors).

Decoratori

```
class Persona:
    def __init__(self, nome, eta):
        self.nome = nome
        self.eta = eta

    @classmethod
    def da_stringa(cls, testo):
        nome, eta = testo.split("-")
        return cls(nome, int(eta))

# Creo oggetto con init normale
p1 = Persona("Marco", 30)

# Creo oggetto con metodo di classe
p2 = Persona.da_stringa("Anna-25")

print(p1.nome, p1.eta)  # Marco 30
print(p2.nome, p2.eta)  # Anna 25
```

Decoratori

Quando usarlo: se vuoi creare metodi che operano a livello di classe (come *costruttori alternativi* o *metodi condivisi*).

Decoratori

`@property:`

- Permette di accedere a un **metodo come se fosse un attributo**.
- Si usa per **controllare l'accesso agli attributi** (getter, setter, deleter) mantenendo la sintassi pulita.
- È una forma di **incapsulamento "pythonic"**.

Decoratori

```
class Rettangolo:
    def __init__(self, base, altezza):
        self._base = base
        self._altezza = altezza

    @property
    def area(self):
        return self._base * self._altezza

r = Rettangolo(4, 5)
print(r.area)  # 20  (notare: senza parentesi!)
```

Decoratori

Estensione con setter

```
class Persona:
    def __init__(self, nome):
        self._nome = nome

    @property
    def nome(self):
        return self._nome

    @nome.setter
    def nome(self, nuovo_nome):
        if len(nuovo_nome) < 2:
            raise ValueError("Nome troppo corto!")
        self._nome = nuovo_nome
```

Decoratori

Decoratore	Primo argomento ricevuto	Uso principale
`@staticmethod`	Nessuno	Funzioni indipendenti, organizzazione dentro la classe
`@classmethod`	La **classe** (`cls`)	Metodi legati alla classe, factory methods
`@property`	L'**istanza** (`self`)	Getter/setter in stile attributo, incapsulamento

Decoratori

Un decoratore è applicato al momento della definizione della funzione

- Quando Python legge il `def`, applica subito il decoratore.
- Non è al momento della chiamata della funzione.

➡️ Quindi:

```
def decoratore(f):
    print("Decoratore applicato")
    return f

@decoratore
def funzione():
    print("Funzione chiamata")

# Output subito:
# Decoratore applicato
```

Decoratori

Ordine dei decoratori multipli

- Se metti più decoratori, vengono applicati **dal basso verso l'alto**.
- Quello più vicino alla funzione è applicato per primo.

```
@A
@B
def f(): pass

# Equivale a:
f = A(B(f))
```

Decoratori

Decoratori possono restituire *qualunque oggetto*

- Non devono per forza restituire una funzione!
- Possono restituire **classi, metodi, persino valori**.
- Esempio: un decoratore che sostituisce la funzione con una costante:

```
def sempre_cinque(_):
    return lambda: 5

@sempre_cinque
def calcolo():
    return 42

print(calcolo())  # 5
```

Decoratori

Decoratori e testing

- Decoratori possono rendere il codice più difficile da testare.
- Alcuni test devono bypassare i decoratori (ad esempio quelli che limitano l’accesso o aggiungono ritardi).
- Soluzioni:
  - Estrarre la logica nel wrapper in una funzione separata.

Decoratori sono “stackable middleware”

- Concettualmente sono molto simili ai **middleware nei framework web**: ogni decoratore aggiunge uno strato.
- Flask, Django, FastAPI, Click (CLI) sfruttano massicciamente questo concetto.

Moduli os e sys in Python

os: interfaccia tra Python e il sistema operativo.

sys: informazioni e interazione con l’interprete Python.

Giacomo Di Salvo

Moduli os e sys in Python

os

- Serve per:
  - Interagire con il **file system**
  - Gestire **processi**
  - Lavorare con **variabili d’ambiente**
- Portabile: funziona su Windows, Linux, macOS (con differenze minori).

os – Operazioni sui file e directory

Moduli os e sys in Python

```
import os

print(os.getcwd())       # Directory corrente
os.mkdir("nuova_cartella")  # Crea una cartella
os.listdir(".")          # Lista dei file nella cartella
os.rename("vecchio.txt", "nuovo.txt")  # Rinomina file
os.remove("file.txt")     # Elimina file
os.rmdir("vuota")        # Elimina cartella vuota
```

Giacomo Di Salvo

Moduli os e sys in Python

os – Percorsi

- os.path fornisce funzioni per lavorare con i percorsi.

•import os

•percorso = "/home/utente/file.txt"
•print(os.path.basename(percorso)) # "file.txt"
•print(os.path.dirname(percorso))   # "/home/utente"
•print(os.path.exists(percorso))   # True/False
•print(os.path.join("cartella", "file.txt")) # "cartella/file.txt"

Moduli os e sys in Python

os – Variabili d’ambiente

```
import os

print(os.environ["PATH"])     # Mostra variabile PATH
os.environ["NUOVA_VAR"] = "ciao"
print(os.environ.get("NUOVA_VAR"))  # "ciao"
```

Moduli os e sys in Python

os – Eseguire comandi di sistema

```
import os

os.system("echo Hello World")
```

Giacomo Di Salvo

Moduli os e sys in Python

Modulo sys – Panoramica

- Fornisce accesso a:
  - Informazioni sull’interprete Python
  - Argomenti da linea di comando
  - Uscita dal programma
  - Moduli caricati

Giacomo Di Salvo

Moduli os e sys in Python

sys – Argomenti da linea di comando

Esempio pratico:

```
import sys

print(sys.argv)   # Lista degli argomenti
# Esempio: python script.py ciao mondo
# Output: ["script.py", "ciao", "mondo"]
```

Moduli os e sys in Python

sys – Uscita dal programma

Esempio pratico:

```
import sys

print("Inizio programma")
sys.exit(0)   # Interrompe subito il programma
print("Non verrà mai eseguito")
```

Moduli os e sys in Python

sys – Informazioni sul sistema

Esempio pratico:

```
import sys

print(sys.version)    # Versione di Python
print(sys.platform)   # Piattaforma (win32, linux, darwin)
```

Moduli os e sys in Python

sys – Input/Output

```
import sys

sys.stdout.write("Scrivo direttamente su stdout\n")
nome = sys.stdin.readline()  # Legge da input
```

Moduli os e sys in Python

```
import os
import sys

def main():
    # Controllo che l'utente abbia passato un argomento
    if len(sys.argv) < 2:
        print("Uso: python script.py <percorso>")
        sys.exit(1)

    percorso = sys.argv[1]

    # Controllo se il percorso esiste
    if not os.path.exists(percorso):
        print(f"Errore: il percorso '{percorso}' non esiste.")
        sys.exit(1)

    # Se è un file, mostro info
    if os.path.isfile(percorso):
        print(f"'{percorso}' è un FILE")
        print(f"Dimensione: {os.path.getsize(percorso)} byte")

    # Se è una cartella, mostro il contenuto
    elif os.path.isdir(percorso):
        print(f"'{percorso}' è una CARTELLA, contenuto:")
        for elemento in os.listdir(percorso):
            print(" -", elemento)

    else:
        print(f"'{percorso}' non è né file né cartella")

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

Giacomo Di Salvo

Moduli os e sys in Python

- **Uso di sys**
  - `sys.argv` → prende gli argomenti passati da riga di comando.
  - `sys.exit()` → esce con codice di errore se qualcosa non va.
- **Uso di os**
  - `os.path.exists()` → controlla se il percorso esiste.
  - `os.path.isfile()` / `os.path.isdir()` → distingue file e cartella.
  - `os.path.getsize()` → legge la dimensione di un file.
  - `os.listdir()` → elenca i contenuti di una cartella.

$ python script.py documento.txt

'documento.txt' è un FILE

Dimensione: 512 byte

$ python script.py cartella/

'cartella/' è una CARTELLA, contenuto:

- foto.jpg

- app.py

- dati.csv

Moduli os e sys in Python

Esercizi sui Decoratori

- Scrivi un decoratore che **stampa “Inizio” e “Fine”** prima e dopo l’esecuzione di una funzione.
- Crea un decoratore che conta **quante volte** una funzione viene chiamata.
- Realizza un decoratore che stampa il **tempo di esecuzione** di una funzione.

Moduli os e sys in Python

Esercizi sui Decoratori

- Scrivi un decoratore che accetta un **parametro n** e ripete l’esecuzione della funzione n volte.
- Crea un decoratore che permette l’esecuzione della funzione **solo se l’utente ha un ruolo = "admin"**, altrimenti stampa un messaggio di errore.
- Realizza un decoratore che **memorizza i risultati** di una funzione costosa in una cache (memoization).

Moduli `os` e `sys` in Python

Esercizi su `os` (File e Cartelle)

- Scrivi un programma che, dato un percorso da input, stampa se è un file o una cartella.
- Realizza uno script che crea una cartella chiamata `backup/` e copia dentro tutti i file `.txt` presenti nella cartella corrente.
- Scrivi un programma che elenca tutti i file di una cartella e mostra per ciascuno la **dimensione in byte**.

Moduli `os` e `sys` in Python

Esercizi su `sys` (Argomenti e Interprete)

- Scrivi uno script che riceve due numeri da riga di comando e stampa la loro somma.
- Realizza un programma che mostra: versione di Python, piattaforma e percorso dei moduli importati.
- Scrivi uno script che prende un file come argomento e lo legge riga per riga, stampandolo su `stdout`.

Moduli os e sys in Python

Esercizio Finale (Mix os + sys + decoratori)

- Scrivi un programma che:
- Riceve da riga di comando un percorso.
- Se è un file, stampa la dimensione.
- Se è una cartella, elenca i file.
- Usa un decoratore per loggare ogni chiamata a funzione con timestamp.

Moduli `os` e `sys` in Python

Cosa sono i moduli in Python

- Un **modulo** è un file Python (`.py`) che contiene funzioni, classi o variabili.
- I moduli permettono di **organizzare il codice** in parti riutilizzabili e più leggibili.
- Python ha:
  - **Moduli built-in** (es. `os`, `sys`, `math`)
  - **Moduli di terze parti** (installati con `pip`, es. `requests`, `numpy`)
  - **Moduli personalizzati** (creati da te)

Un modulo non è altro che **un file Python** che puoi importare in altri programmi.

Moduli `os` e `sys` in Python

Creare un Modulo Personalizzato

- Crea un file chiamato **`mio_modulo.py`** con dentro funzioni o variabili:

```
# mio_modulo.py
def saluta(nome):
    return f"Ciao, {nome}!"

PI_GRECO = 3.14159
```

Moduli `os` e `sys` in Python

In un altro file **`main.py`**, importa e usa il modulo:

```
import mio_modulo

print(mio_modulo.saluta("Alice"))
print("Valore di PI:", mio_modulo.PI_GRECO)
```

Moduli os e sys in Python

Come Importare un Modulo

- `import modulo`
  Importa tutto il modulo (accedi con `modulo.funzione`).
- `from modulo import funzione`
  Importa solo quella funzione (usi direttamente `funzione()`).
- `from modulo import *`
  Importa tutto senza prefisso (**sconsigliato**, può creare conflitti).
- `import modulo as alias`
  Importa con un nome abbreviato:

Moduli os e sys in Python

```
import mio_modulo as kekw
print(kekw.saluta("Bob"))
```

Giacomo Di Salvo

Moduli `os` e `sys` in Python

La variabile `__name__`

- Ogni modulo in Python ha una variabile speciale chiamata **`__name__`**.
- Se un file viene **eseguito direttamente**, allora `__name__ == "__main__"`.
- Se un file viene **importato come modulo**, allora `__name__` assume il **nome del file** (senza `.py`).

Moduli `os` e `sys` in Python

Uso di `if __name__ == "__main__":`

Questa condizione serve a distinguere due casi:

- **Il file viene eseguito direttamente** → esegue il codice dentro `if`.
- **Il file viene importato come modulo** → il codice dentro `if` **non viene eseguito**.

```
# calcoli.py
def somma(a, b):
    return a + b

if __name__ == "__main__":
    # Questo codice si esegue solo se avvio calcoli.py
    print("Somma 2+3 =", somma(2, 3))
```

Moduli `os` e `sys` in Python

Avvio con `python calcoli.py` → stampa `Somma 2+3 = 5`

Importo in un altro file con `import calcoli` → non stampa nulla (solo funzioni disponibili).

Giacomo Di Salvo

Moduli `os` e `sys` in Python

Vantaggi di `if __name__ == "__main__":`

- Evita che del codice venga eseguito **accidentalmente** quando importiamo il modulo.
- Permette di avere un file che sia sia:
  modulo riutilizzabile (quando importato)
  programma eseguibile (quando lanciato direttamente).

Moduli `os` e `sys` in Python

Esercizi sui Moduli

- Crea un modulo `matematica.py` che contenga una funzione `somma(a, b)` e `moltiplica(a, b)`. Importalo in un file principale e usalo.
- Crea un modulo `geometria.py` con una costante `PI = 3.14159` e una funzione `area_cerchio(r)`. Importalo in un programma e calcola l'area di un cerchio con raggio 5.
- Crea un modulo `saluti.py` con una funzione `ciao(nome)` che ritorna un saluto. Importa e usa questa funzione in un file `main.py`.

Moduli `os` e `sys` in Python

Esercizi su Import e Alias

- Crea un modulo `conversioni.py` con due funzioni:
  - `km_to_miglia(km)`
  - `miglia_to_km(miglia)`
    `Importa il modulo in un file principale usando un` **alias** e prova entrambe le funzioni.
- Scrivi un modulo `stringhe.py` con una funzione `conta_vocali(s)`. Importa **solo quella funzione** in un file principale e usala.

Moduli `os` e `sys` in Python

Esercizi su `__name__`

- Crea un modulo `test.py` che stampi il valore di `__name__`. Eseguilo direttamente e poi importalo in un altro file. Vedi le differenze osservate.

- Scrivi un modulo `calcoli.py` con una funzione `quadrato(x)` e un blocco `if __name__ == "__main__":` che calcoli il quadrato di 10. Importalo in un altro file e osserva cosa succede.

Moduli `shutil` e `requests` in Python

Shell Utilities

- Fornisce **funzioni di alto livello** per:
- Copiare file e directory
- Spostare file
- Cancellare directory
- Creare archivi (zip, tar, ecc.)
- Lavora **sopra os**, semplificando operazioni comples

Giacomo Di Salvo

Moduli `shutil` e `requests` in Python

`shutil.copy(src, dst)` → copia contenuto + permessi.

`shutil.copy2(src, dst)` → copia contenuto + metadati (timestamp, permessi, ecc.).

- Utile per backup o duplicati.

`copy2()` è preferibile quando è importante mantenere **integrità dei metadati**.

Moduli `shutil` e `requests` in Python

Copiare directory

- **`shutil.copytree(src, dst)`** → copia ricorsiva dell'intera cartella.
- Crea la nuova cartella e copia tutto il contenuto (file + sottocartelle).
- Dal Python 3.8 → parametro `dirs_exist_ok=True` permette di copiare anche in cartelle già esistenti.

shutil.copytree("cartella_origine", "cartella_copia")

è molto utile per clonare directory (es. ambienti di progetto, backup).

Moduli `shutil` e `requests` in Python

Spostare e rinominare

- **`shutil.move(src, dst)`** → sposta file o directory.
- Se la destinazione è sullo stesso disco → è una semplice **rename operation**.
- Se è su disco diverso → avviene una vera **copia + rimozione**

shutil.move("file.txt", "nuova_cartella/file_rinominato.txt")

Moduli `shutil` e `requests` in Python

- **`shutil.rmtree(path)`** → elimina una directory in modo **ricorsivo** (contenuto incluso).

Attenzione: l'eliminazione è **definitiva** (non passa dal cestino).

- Per eliminare singoli file → usare `os.remove()`.

```
shutil.rmtree("cartella_da_eliminare")
```

usare con cautela.

Moduli `shutil` e `requests` in Python

Creare archivi

- **`shutil.make_archive(base_name, format, root_dir)`** → crea un archivio a partire da una cartella.
- Formati supportati: `"zip"`, `"tar"`, `"gztar"`, `"bztar"`, `"xztar"`.
- Utile per **backup automatici** o distribuzione di progetti.

shutil.make_archive("backup", "zip", "cartella_origine")

genera direttamente `backup.zip` (senza librerie esterne)

Moduli `shutil` e `requests` in Python

Estrarre archivi

- **`shutil.unpack_archive(filename, extract_dir)`** → estrae un archivio.
- Riconosce automaticamente il formato in base all'estensione.
- Supporta diversi formati standard (`zip`, `tar`, ecc.).

shutil.unpack_archive("backup.zip", "cartella_destinazione")

utile per importare rapidamente pacchetti o backup.

Moduli `shutil` e `requests` in Python

Altre funzioni utili

- **`shutil.disk_usage(path)`** → restituisce spazio totale, usato e libero del disco.
- **`shutil.copyfileobj(src, dst)`** → copia contenuti tra oggetti file (es. flussi).
- Utile per operazioni a basso livello con file molto grandi.

```
total, used, free = shutil.disk_usage("/")
print(f"Totale: {total // (2**30)} GB, Libero: {free // (2**30)} GB")
```

Moduli `shutil` e `requests` in Python

`requests` è una libreria di terze parti

(da installare con `pip install requests`).

- Semplifica enormemente l'interazione con il protocollo **HTTP/HTTPS**.
- Obiettivo: **leggibilità e semplicità** rispetto a moduli più complessi come `urllib`.

Moduli `shutil` e `requests` in Python

Supporta:

- Richieste GET, POST, PUT, DELETE, ecc.
- Invio di parametri e dati in vari formati (query string, form, JSON).
- Gestione delle sessioni e dei cookie.
- Gestione delle autenticazioni e headers personalizzati.
- Download/upload file.

Moduli `shutil` e `requests` in Python

- Metodo **`get()`** → ottiene una risorsa da un server.

```
import requests


response = requests.get("https://jsonplaceholder.typicode.com/posts/1")


print(response.status_code)  # codice HTTP (200 = OK)
print(response.headers)      # intestazioni della risposta
print(response.text)         # contenuto come stringa
print(response.json())       # parsing diretto JSON (se supportato)
```

Moduli `shutil` e `requests` in Python

- `status_code` → stato della risposta (200, 404, 500, ...).
- `headers` → metadati (content-type, server, ecc.).
- `text` → corpo della risposta come stringa.
- `json()` → parsing automatico se il server restituisce JSON.

Moduli `shutil` e `requests` in Python

Richieste con parametri

I parametri possono essere passati nella query string (`?key=value`).

```
payload = {"userId": 1}
response =
requests.get("https://jsonplaceholder.typicode.com/po
sts", params=payload)


print(response.url)    # mostra l’URL finale con
parametri
print(response.json()) # restituisce solo i post di
userId=1
```

Moduli `shutil` e `requests` in Python

Richieste POST

- Usata per **inviare dati** al server (form, JSON, ecc.).

```
data = {"title": "Nuovo post", "body": "Contenuto", "userId": 1}
response = requests.post("https://jsonplaceholder.typicode.com/posts", json=data)

print(response.status_code)  # 201 = Created
print(response.json())      # il server ritorna i dati salvati
```

Moduli `shutil` e `requests` in Python

- `data=` → invia dati come **form-urlencoded**.
- `json=` → invia direttamente dati JSON con `Content-Type: application/json`.

Headers personalizzati

- Possiamo inviare **informazioni extra** con la richiesta.

```
headers = {"User-Agent": "CorsoPython/1.0"}
response =
requests.get("https://httpbin.org/headers",
headers=headers)


print(response.json())
```

Moduli `shutil` e `requests` in Python

Cosa sono gli headers

- Gli **HTTP headers** sono **metadati** che accompagnano una richiesta (**request**) o una risposta (**response**) HTTP.
- Non fanno parte del contenuto principale (body), ma forniscono **informazioni aggiuntive** su:
  - **Chi manda la richiesta** (es. browser, applicazione).
  - **Tipo di dati** inviati o accettati.
  - **Autenticazione o autorizzazione**.
  - **Gestione delle connessioni** (caching, compressione, ecc.).

Moduli `shutil` e `requests` in Python

GET /pagina HTTP/1.1

Host: www.example.com

User-Agent: Mozilla/5.0

Accept: text/html

Authorization: Bearer <token>

- `Host` → dominio richiesto.
- `User-Agent` → identifica il client (browser o app).
- `Accept` → specifica i formati che il client può ricevere (`html`, `json`, ecc.).
- `Authorization` → contiene token o credenziali per accedere a risorse protette.

Moduli `shutil` e `requests` in Python

Il server risponde con headers che descrivono la risposta:

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1234
Set-Cookie: sessionid=abcd1234

- `Content-Type` → indica che il corpo della risposta è JSON.
- `Content-Length` → lunghezza in byte del contenuto.
- `Set-Cookie` → invia cookie al client.

Moduli `shutil` e `requests` in Python

Headers in `requests`

Con Python puoi **leggere** e **inviare** headers facilmente:

Come leggere headers della risposta

```
import requests

r = requests.get("https://httpbin.org/get")
print(r.headers)  # mostra gli headers della risposta
```

Moduli `shutil` e `requests` in Python

Headers più comuni

Request headers (client → server)

- `User-Agent` → identifica il client.
- `Accept` → formati accettati (`application/json`, `text/html`).
- `Authorization` → token o credenziali.
- `Content-Type` → tipo di dati inviati (es. `application/json`).

Response headers (server → client)

- `Content-Type` → tipo di risposta (`application/json`, `text/html`).
- `Content-Length` → dimensione del contenuto.
- `Set-Cookie` → gestione sessioni.
- `Server` → tecnologia del server web (Apache, Nginx, ecc.).

Moduli `shutil` e `requests` in Python

Gli **headers sono come etichette** che accompagnano ogni messaggio HTTP e permettono al client e al server di **capirsi meglio** (quale formato usare, come autenticarsi, come gestire le connessioni).

Moduli `shutil` e `requests` in Python

Sessioni e Cookie

- Con `Session()` possiamo mantenere **cookie, headers, autenticazioni** tra richieste.

```
session = requests.Session()
session.headers.update({"User-Agent": "SessionTest/1.0"})

r1 = session.get("https://httpbin.org/cookies/set/sessioncookie/123")
r2 = session.get("https://httpbin.org/cookies")

print(r2.json())  # mantiene il cookie della sessione
```

Moduli `shutil` e `requests` in Python

Timeout e gestione errori

```
try:
    response = requests.get("https://httpbin.org/delay/5", timeout=3)
except requests.exceptions.Timeout:
    print("La richiesta è scaduta!")
```

- `timeout` evita che il programma resti bloccato indefinitamente.
- Eccezioni comuni:
- `Timeout`
- `ConnectionError`
- `HTTPError`

Moduli `shutil` e `requests` in Python

Autenticazione

Basic Auth

```
from requests.auth import HTTPBasicAuth

response = requests.get("https://httpbin.org/basic-auth/user/pass",
                        auth=HTTPBasicAuth("user", "pass"))
print(response.status_code)  # 200 se ok
```

Moduli `shutil` e `requests` in Python

- Supporta Basic, Digest, OAuth (con librerie aggiuntive).
- Molte API moderne usano **token nei headers** (`Authorization: Bearer <token>`).

Moduli `shutil` e `requests` in Python

Esercizi Requests

Scrivi un programma che:

- Faccia una richiesta `GET` all'endpoint
- `https://jsonplaceholder.typicode.com/posts/1`

- Stampi lo **status code**, gli **headers** e il **contenuto JSON** della risposta.

Moduli `shutil` e `requests` in Python

Query Parameters

Scrivi un programma che:

- Usi `requests.get()` per chiamare
- `https://jsonplaceholder.typicode.com/posts`

- Aggiunga il parametro `userId=2` alla richiesta.
- Stampi tutti i titoli dei post restituiti.

Moduli `shutil` e `requests` in Python

Scrivi un programma che:

- Invi una richiesta POST a
- https://jsonplaceholder.typicode.com/posts

- Includa un JSON con:
- {"title": "Python", "body": "Sto studiando requests", "userId": 10}

- Stampi la risposta del server in formato JSON.

Moduli `shutil` e `requests` in Python

Scrivi un programma che:

- Faccia una richiesta GET a
- `https://httpbin.org/headers`

- Includa negli headers:
  - `User-Agent: CorsoPython/2.0`
  - `Authorization: Bearer 12345`
- Stampi la risposta in JSON e verifichi che gli headers siano stati ricevuti.

Moduli `shutil` e `requests` in Python

Scrivi un programma che:

- Scarichi il logo di Python da
- `https://www.python.org/static/img/python-logo.png`

- Salvi il file con nome **`logo.png`**.
- Stampi un messaggio di conferma quando il file è stato salvato.

Moduli `shutil` e `requests` in Python

Sessioni e Cookie

Scrivi un programma che:

- Crei una **sessione** `requests.Session()`.

- Setti un cookie su `https://httpbin.org/cookies/set/testcookie/12345`

- Recuperi i cookie da `https://httpbin.org/cookies` e stampi la risposta.

Moduli `shutil` e `requests` in Python

Scrivi un programma che:

- Faccia una richiesta GET a
- `https://httpbin.org/delay/5`

- Imposti `timeout=3`.

- Gestisca l'eccezione `requests.exceptions.Timeout` stampando un messaggio di errore.

?

Q&A