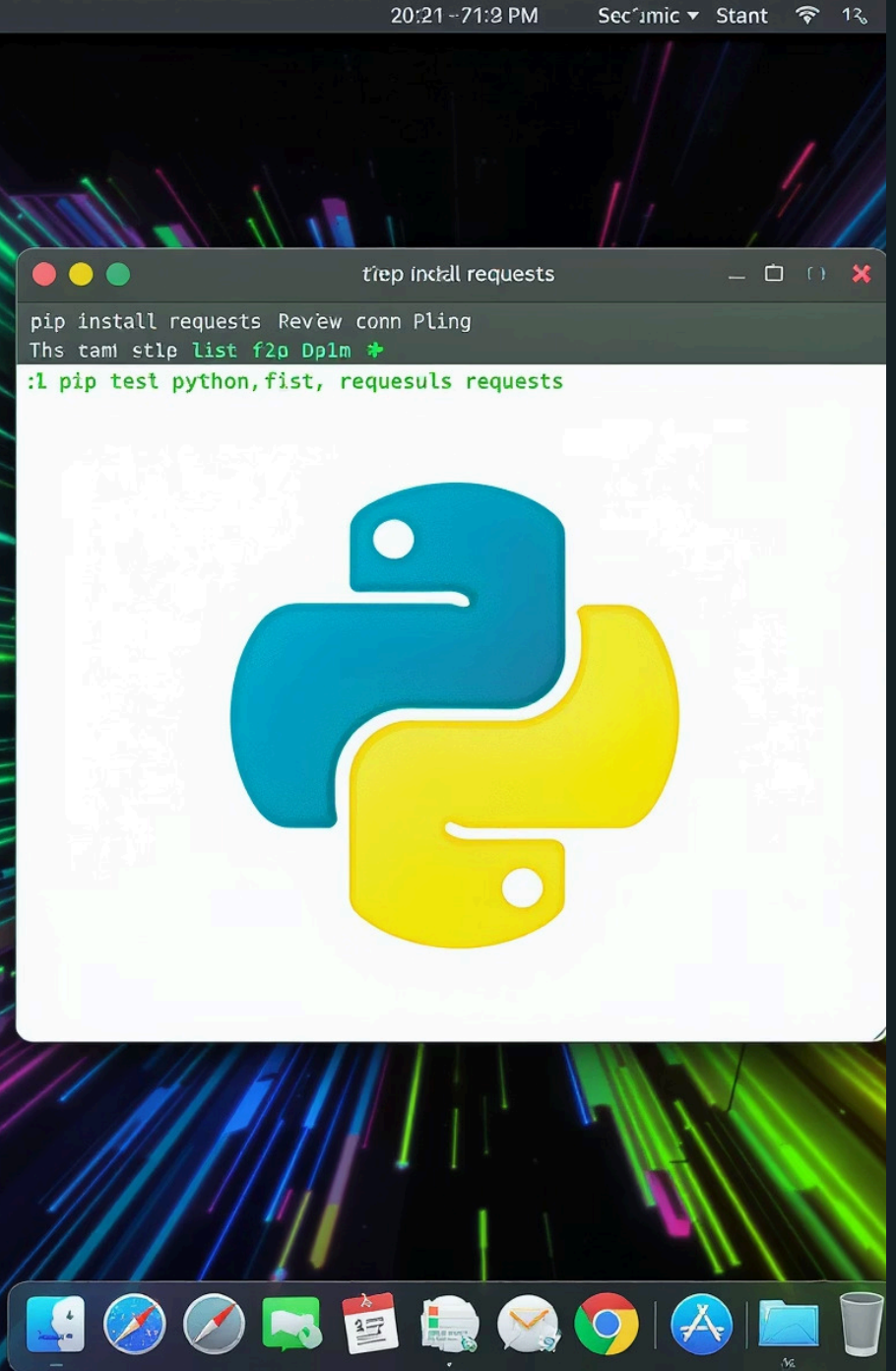
# Requests: Interagire con il Web in Python

Benvenuti a questo corso sulla libreria Requests di Python! Oggi esploreremo uno strumento fondamentale per qualsiasi sviluppatore Python che voglia interagire con il web.

Requests è una libreria elegante e semplice che consente di inviare richieste HTTP senza sforzo. Nelle prossime slide, impareremo a utilizzarla per comunicare con API web, scaricare dati e molto altro ancora.

Il nostro viaggio ci porterà dalle basi fino a tecniche più avanzate, fornendovi tutte le competenze necessarie per integrare i vostri progetti Python con servizi web esterni.




# Installazione della Libreria Requests

## Installazione con pip

Il modo più comune per installare Requests è tramite pip, il gestore di pacchetti di Python. Apri il terminale e digita:

**pip install requests**

## Installazione con conda

Se utilizzi Anaconda, puoi installare Requests usando conda:

**conda install -c anaconda requests**

## Verifica dell'installazione

Per verificare che l'installazione sia avvenuta correttamente, apri Python e prova a importare la libreria:

**import requests**

Una volta installata la libreria, saremo pronti a iniziare a fare le nostre prime richieste web. L'installazione è semplice e veloce, rendendo Requests accessibile anche ai principianti.


# La Tua Prima Chiamata GET

## Importa la libreria

Il primo passo è importare la libreria Requests nel tuo script Python:

**import requests**

## Esegui una richiesta GET

Usa il metodo **requests.get()** per fare una semplice richiesta a un URL:

**response = requests.get('https://api.example.com/data')**

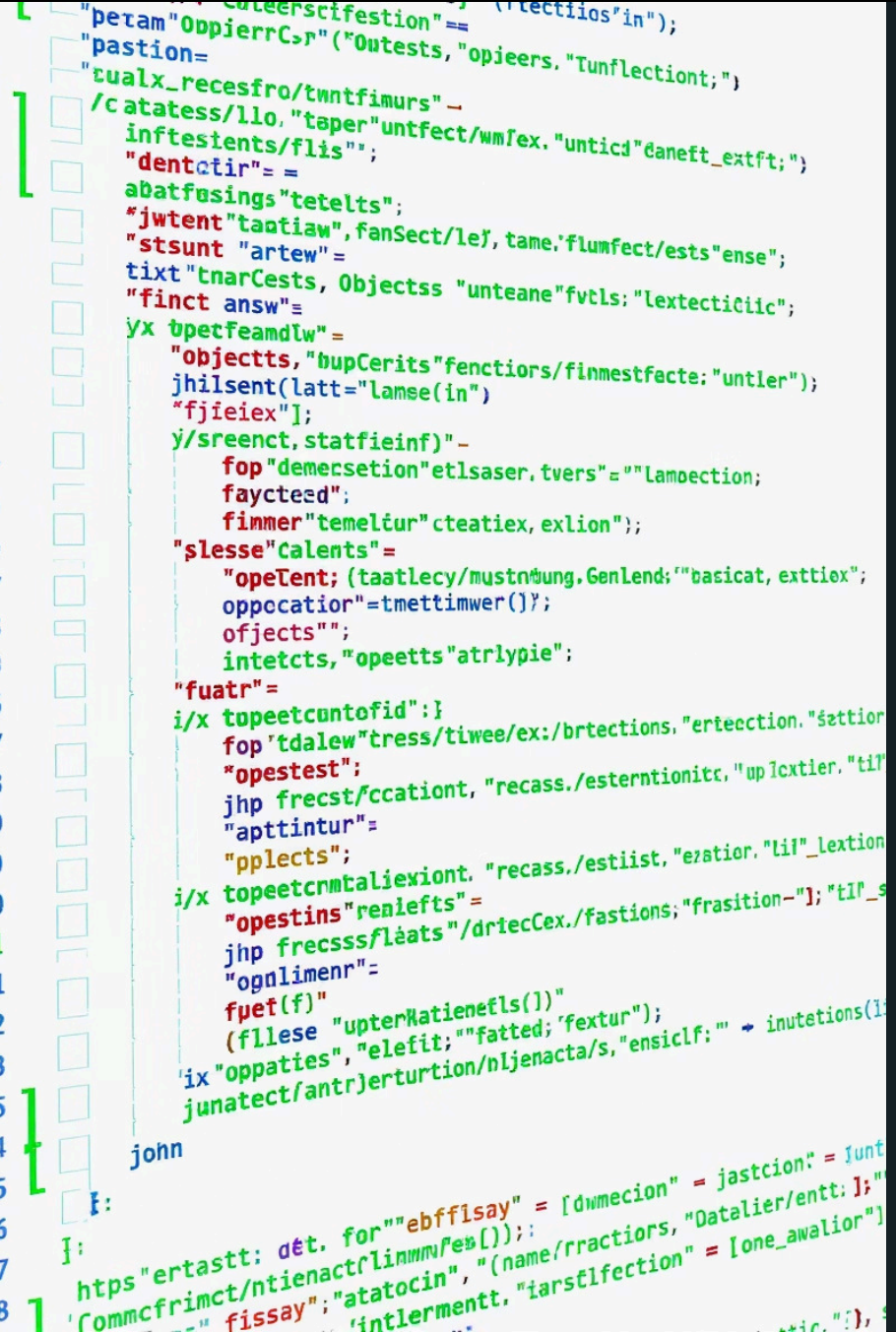
## Esamina la risposta

Puoi esaminare la risposta in vari modi:

**print(response.text)** # Contenuto testuale della risposta

**print(response.status_code)** # Codice di stato HTTP (200, 404, ecc.)

Congratulazioni! Hai appena eseguito la tua prima richiesta HTTP con Python. Questa operazione di base è il fondamento di qualsiasi interazione con API web e servizi online.



# Parsing delle Risposte JSON

## Ricevi dati JSON

Molte API web restituiscono dati in formato JSON, un formato leggero per lo scambio di dati.

## Converti in oggetto Python

Requests può convertire automaticamente la risposta JSON in un dizionario Python:

```
data = response.json()
```

## Accedi ai dati

Ora puoi accedere ai dati come faresti con qualsiasi dizionario:

```
print(data['results'][0]['name'])
```

Il metodo **.json()** converte la risposta JSON in strutture dati Python come dizionari e liste, facilitando l'elaborazione dei dati. Ricorda che questo metodo solleverà un'eccezione se la risposta non contiene JSON valido.

# Gestione degli Errori HTTP

### Controlla status_code

Verifica il codice di stato HTTP per capire l'esito della richiesta:

**if response.status_code == 200:**

### Cattura le eccezioni

Usa try/except per gestire elegantemente gli errori nelle richieste.

### Gestisci codici di errore

I codici 4xx indicano errori del client, i codici 5xx indicano errori del server.

### Usa .raise_for_status()

Questo metodo solleva un'eccezione se la richiesta non è andata a buon fine:

**response.raise_for_status()**

Una gestione appropriata degli errori è essenziale per creare applicazioni robuste. Ricorda che le API possono non essere sempre disponibili o possono restituire errori inaspettati, quindi è importante pianificare di conseguenza.


# Parametri nelle Richieste GET

### Filtri

Usa i parametri GET per filtrare i risultati dell'API:

**{'category': 'electronics', 'min_price': 100}**

### Ricerca

Invia termini di ricerca come parametri:

**{'q': 'python programming', 'sort': 'relevance'}**

### Paginazione

Gestisci risultati paginati con parametri:

**{'page': 2, 'limit': 20}**

I parametri GET vengono passati come dizionario al parametro **params** del metodo **requests.get()**:

**response = requests.get('https://api.example.com/search', params={'q': 'python', 'limit': 10})**

Requests si occuperà automaticamente di codificare correttamente i parametri nell'URL, trasformando l'esempio sopra in: https://api.example.com/search?q=python&limit=10


# Chiamate POST con Dati JSON

## Prepara i dati JSON

Crea un dizionario Python con i dati da inviare:

**data = {'name': 'Mario', 'email': 'mario@example.com'}**

## Invia la richiesta POST

Usa il metodo **requests.post()** con il parametro **json**:

**response = requests.post('https://api.example.com/users', json=data)**

## Verifica la risposta

Controlla il codice di stato e la risposta del server:

**if response.status_code == 201: print("Utente creato con successo!")**

Il parametro **json** si occupa automaticamente di convertire il dizionario Python in una stringa JSON e imposta l'header **Content-Type: application/json**. Questo è molto comodo quando si lavora con API RESTful moderne.

# Header Personalizzati

## Token di autenticazione

Molte API richiedono token di accesso negli header:

**headers = {'Authorization': 'Bearer your_token_here'}**

## Preferenze linguistiche

Specifica la lingua preferita per la risposta:

**headers = {'Accept-Language': 'it-IT'}**

## Tipo di contenuto

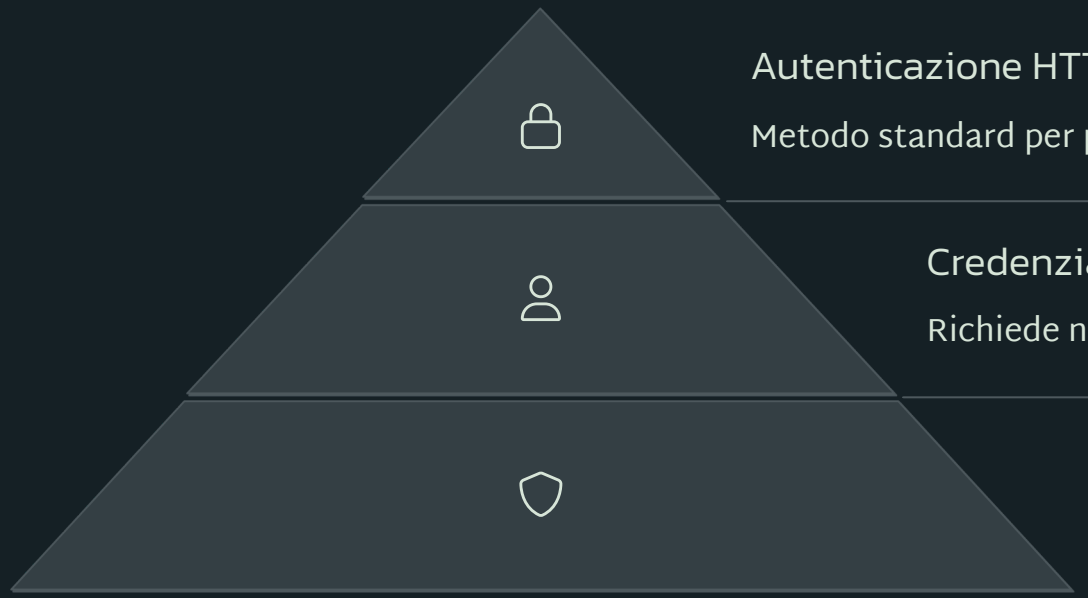
Specifica il formato dei dati che stai inviando:

**headers = {'Content-Type': 'application/json'}**

Gli header HTTP vengono passati come dizionario al parametro **headers** delle funzioni di richiesta:

response = requests.get('https://api.example.com/data', headers={'User-Agent': 'MyApp/1.0', 'Accept': 'application/json'})

# Autenticazione Base



### Autenticazione HTTP Basic

Metodo standard per proteggere API con nome utente e password

### Credenziali

Richiede nome utente e password per l'accesso

### Sicurezza

Deve essere usata con HTTPS per proteggere le credenziali

Requests semplifica l'autenticazione HTTP Basic con il parametro **auth**:

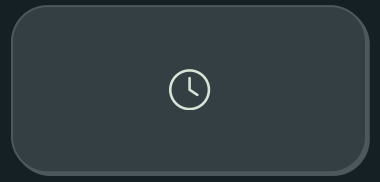
```
from requests.auth import HTTPBasicAuth

response = requests.get('https://api.example.com/secure', auth=HTTPBasicAuth('username', 'password'))
```

Esiste anche una forma abbreviata più semplice:

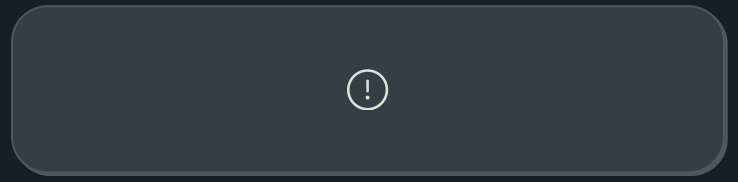
```
response = requests.get('https://api.example.com/secure', auth=('username', 'password'))
```

# Gestione dei Timeout



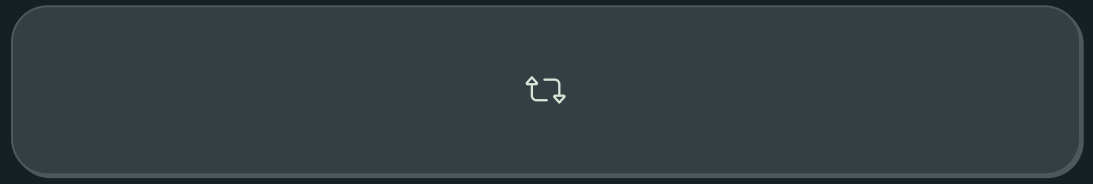
### Imposta un timeout

Evita che le richieste bloccanti rallentino la tua applicazione



### Gestisci le eccezioni

Cattura Timeout Error per gestire i tempi di risposta eccessivi



### Implementa strategie di retry

Riprova la richiesta dopo un timeout, se appropriato

Per impostare un timeout, aggiungi il parametro **timeout** alla tua richiesta:

```
try:

response = requests.get('https://api.example.com/data', timeout=5) # 5 secondi

except requests.exceptions.Timeout:

print("La richiesta è andata in timeout")
```

Puoi anche specificare timeout separati per la connessione e la lettura: timeout=(3, 10)

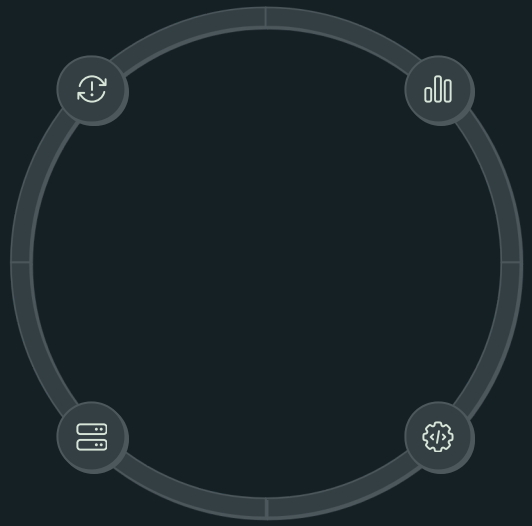
# Retry Automatici

### Configurazione dei retry

Usa HTTPAdapter per configurare i tentativi automatici

### Backoff esponenziale

Aumenta progressivamente il tempo tra i tentativi



### Errori di connessione

Riprova automaticamente in caso di problemi di rete

### Status code specifici

Configura i retry solo per certi codici di errore

Ecco come configurare i retry automatici con Requests:

```
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.util.retry import Retry
retry_strategy = Retry(total=3, backoff_factor=1, status_forcelist=[429, 500, 502, 503, 504])
adapter = HTTPAdapter(max_retries=retry_strategy)
session = requests.Session()
session.mount("https://", adapter)
```

# Logging delle Chiamate HTTP

## Configurazione del logging

Python include un modulo di logging integrato che può essere utilizzato per registrare le richieste HTTP:

```
import logging

logging.basicConfig(level=logging.DEBUG)

logging.getLogger("requests").setLevel(logging.DEBUG)
```

## Informazioni registrate

Con il logging abilitato, vedrai dettagli su:

- URL richiesti
- Header inviati e ricevuti
- Corpo delle richieste
- Codici di stato

## Logging personalizzato

Per un maggiore controllo, puoi creare il tuo sistema di logging:

```
def log_request(response):

print(f"{response.request.method} {response.url}")

print(f"Status: {response.status_code}")
```

Il logging è essenziale durante lo sviluppo e il debug di applicazioni che interagiscono con API. Ti aiuta a capire esattamente cosa viene inviato e ricevuto, facilitando l'identificazione dei problemi.

# Download di File tramite API

## Streaming di file grandi

Per file di grandi dimensioni, è importante utilizzare lo streaming per evitare di caricare l'intero file in memoria:

**response = requests.get('https://example.com/file.pdf', stream=True)**

Il parametro **stream=True** ritarda il download del corpo della risposta fino a quando non accedi al contenuto.

## Salvare il file

Una volta ottenuta la risposta, puoi salvare il contenuto su disco in modo efficiente:

**with open('file.pdf', 'wb') as f:**

**for chunk in response.iter_content(chunk_size=8192):**

**f.write(chunk)**

Il metodo **iter_content()** permette di scaricare il file a blocchi, riducendo l'utilizzo della memoria.

Quando scarichi file binari, assicurati di aprire il file in modalità binaria (**'wb'**) per evitare problemi di codifica dei caratteri. Il parametro **chunk_size** può essere regolato in base alle tue esigenze.


# Upload di File con Requests

### Prepara il file

Apri il file che desideri caricare:

**files = {'document': open('report.pdf', 'rb')}**

### Invia la richiesta

Usa il metodo POST con il parametro files:

**response = requests.post('https://api.example.com/upload', files=files)**

### Verifica il risultato

Controlla la risposta per confermare il successo:

**if response.status_code == 201: print("File caricato con successo!")**

### Chiudi il file

Assicurati di chiudere il file dopo l'upload:

**files['document'].close()**

Per caricare più file contemporaneamente, aggiungi più elementi al dizionario **files**. Puoi anche specificare un nome di file diverso sul server:

files = {'document': ('report2023.pdf', open('report.pdf', 'rb'), 'application/pdf')}

# Gestione delle Eccezioni di Rete

## Timeout

Si verifica quando una richiesta impiega troppo tempo. Puoi gestirla con:

**except requests.exceptions.Timeout:**

Utile per implementare fallback o notificare l'utente del problema di connessione.

## ConnectionError

Si verifica quando non è possibile connettersi al server:

**except requests.exceptions.ConnectionError:**

Può essere causato da problemi DNS, server offline o errori di rete.


## SSLError

Si verifica quando ci sono problemi con la connessione sicura:

**except requests.exceptions.SSLError:**

Può essere dovuto a certificati scaduti o non validi sul server.

Una gestione robusta delle eccezioni è fondamentale per le applicazioni che dipendono da servizi di rete. Ricorda che la classe base **requests.exceptions.RequestException** può essere usata per catturare qualsiasi errore di Requests.