Dispense Universitarie: Python per Cyber Security & Ethical Hacking

Modulo: Linguaggi di Programmazione | Livello: Master

---

1. Introduzione al Python in Ambito Security

1. 📖 Spiegazione Approfondita

Python si è affermato come standard de facto nell'industria della sicurezza informatica. A differenza di linguaggi di basso livello che richiedono una gestione complessa della memoria, Python è un linguaggio **interpretato, orientato agli oggetti e multipiattaforma**. Questo significa che il codice sorgente viene eseguito direttamente da un interprete senza necessità di compilazione preventiva, rendendolo portabile su diversi sistemi operativi (incluso Kali Linux, dove è preinstallato).

L'importanza strategica di Python nella Cybersecurity deriva da quattro pilastri fondamentali:

1. **Automazione:** Permette di scriptare attività ripetitive che richiederebbero ore se svolte manualmente.
2. **Penetration Testing:** È la base per la creazione di exploit personalizzati e strumenti di attacco/difesa.
3. **Integrazione:** Facilita la connessione tra diversi software e librerie esistenti.
4. **Prototipazione Rapida:** La sua sintassi semplice permette di passare dall'idea al codice funzionante (PoC - Proof of Concept) in tempi brevissimi.

2. 🛠 Sintassi & Comandi (Cleaning)

Per verificare l'installazione e la versione su sistemi Linux (es. Kali):

> `python -V` *Output atteso:* `Python 3.11.9` (o versione corrente installata).

3. 🧠 Analisi Tecnica & Memorizzazione

- **Definizione Chiave:** Python è un linguaggio *interpretato* e *dinamicamente tipizzato*, essenziale per l'automazione offensiva e difensiva.
- **Contesto Reale:** Un analista SOC usa Python per parsare log di firewall giganti; un Ethical Hacker lo usa per scrivere script che testano vulnerabilità SQL Injection in massa.

4. ⚠ Focus Esame

- **Domanda:** Perché Python è definito "interpretato"?
- **Risposta:** Perché non richiede un compilatore che trasformi il codice in eseguibile binario prima dell'esecuzione; il codice viene letto ed eseguito riga per riga dall'interprete.

---

2. Programmazione Orientata agli Oggetti (OOP)

1. 📖 Spiegazione Approfondita

Python supporta nativamente il paradigma OOP (*Object Oriented Programming*). Questo approccio serve a modellare il software basandosi su concetti del mondo reale per migliorare la manutenibilità e il riutilizzo del codice.

Il paradigma si basa su due concetti cardine:

- **Classe:** È il "progetto" o lo stampo astratto. Definisce le caratteristiche (attributi) e le azioni possibili (metodi) comuni a tutti gli elementi di quel tipo.
- **Oggetto:** È l'istanza concreta della classe. È l'entità specifica creata seguendo le regole della classe, ma con i propri dati specifici.

Esempio Concettuale: La classe è "Automobile" (attributi generici: colore, velocità; metodi: accelera, frena). L'oggetto è una specifica "Fiat Punto" (colore: verde, velocità: 65 km/h).

I vantaggi dell'OOP includono:

1. **Riutilizzo del codice:** Scrivi la classe una volta, crei infiniti oggetti.
2. **Modellazione reale:** Mappa concetti fisici in strutture dati logiche.
3. **Manutenibilità:** Il codice è modulare; modificare la classe aggiorna il comportamento di tutti gli oggetti derivati.

3. 🧠 Analisi Tecnica & Memorizzazione

- **Definizione Chiave:** Una Classe è una definizione astratta; un Oggetto è un'istanza che incapsula dati (attributi) e comportamenti (metodi).

4. 🔗 Cross-Connection Master

In ambito **Malware Development**, l'OOP è usato per creare malware modulari (es. classe `Ransomware` con metodi `encrypt()` e `decrypt()`), permettendo di generare varianti del virus cambiando solo specifici attributi senza riscrivere il core logic.

---

3. Modalità di Utilizzo e Sintassi Base

1. 📖 Spiegazione Approfondita

Python può essere eseguito in due modalità distinte:

1. **Prompt Interattivo:** Accessibile digitando `python` nel terminale. Esegue il codice riga per riga ("al momento") con feedback immediato (`>>>`). Utile per test rapidi, calcoli o debug, ma il codice non viene salvato.
2. **Modalità Script (Convenzionale):** Si scrive il codice in un file di testo con estensione `.py` (es. `exploit.py`) e lo si esegue interamente.

Indentazione: A differenza del linguaggio C che usa le parentesi graffe `{}` per delimitare i blocchi di codice, Python utilizza l'**indentazione** (spazi o tabulazioni). L'indentazione non è estetica ma *sintattica*: definisce dove inizia e finisce un blocco (es. dentro un `if` o un ciclo `while`). Inoltre, non sono necessari i punti e virgola (`;`) per terminare le istruzioni.

2. 🛠 Sintassi & Comandi (Cleaning)

Esecuzione di uno script:

```
python nome_file.py
```

Esempio di blocco indentato:

```
def saluta(nome):
    if nome:
        # Questo blocco è dentro l'if
        print(f"Ciao, {nome}!")
    else:
        # Questo blocco è dentro l'else
        print("Ciao, mondo!")
```

5. ⚠ Focus Esame

- **Trabocchetto:** In Python, mischiare tabulazioni e spazi genera un `IndentationError`. È fondamentale usare un editor configurato correttamente.
- **Nota:** Le variabili sono *Case Sensitive* (`Nome` è diverso da `nome`).

---

4. Variabili e Tipizzazione Dinamica

1. 📖 Spiegazione Approfondita

In Python non esiste la dichiarazione esplicita del tipo di variabile (come `int n;` in C). Il linguaggio è **dinamicamente tipizzato**: il tipo della variabile viene determinato automaticamente dall'interprete in base al valore assegnato a runtime.

Regole di Nomenclatura:

- Devono iniziare con lettera o underscore (`_`).
- Possono contenere numeri (ma non all'inizio).
- Non possono essere parole chiave riservate (es. `if`, `class`, `while`).
- Convenzione consigliata: `snake_case` per variabili (es. `mio_nome`), `PascalCase` per classi.

Tipi di Dati Fondamentali:

- **int:** Interi (es. `10`).
- **float:** Numeri con virgola (es. `3.14`).
- **str:** Stringhe di testo (es. `"Alice"`).
- **bool:** Booleani (`True`, `False`).
- **list:** Sequenze ordinate mutabili (es. `[1, 2, 3]`).
- **tuple:** Sequenze ordinate immutabili (es. `(10, 20)`).
- **dict:** Coppie chiave-valore (es. `{"nome": "Alice"}`).

3. 🧠 Analisi Tecnica & Memorizzazione

- **Definizione Chiave:** In Python le variabili sono *riferimenti* a oggetti in memoria. L'assegnazione `x = 10` crea un oggetto intero 10 e "attacca" l'etichetta `x` ad esso.

---

5. Operatori e Stringhe

1. 📖 Spiegazione Approfondita

Operatori Aritmetici: Python supporta le operazioni classiche: `+` (addizione), `-` (sottrazione), `*` (moltiplicazione), `/` (divisione float), `//` (divisione intera), `**` (potenza), `%` (modulo/resto).

Manipolazione Stringhe: Le stringhe possono essere delimitate da doppi apici `"` o singoli `'`. Sono oggetti che possiedono metodi nativi per la loro manipolazione:

- **Concatenazione:** Si usa l'operatore `+` per unire due stringhe.
- **Metodi utili:**
  - `.split(sep)`: Divide la stringa in una lista di sottostringhe basandosi su un separatore.
  - `.upper()`: Converte tutto in maiuscolo.

2. 🛠 Sintassi & Comandi (Cleaning)

Esempio manipolazione stringa:

```
x = "Hello, World"
print(x.split(sep=",")) # Output: ['Hello', ' World']
print(x.upper())        # Output: HELLO, WORLD

y = " this is Python"
print(x + y)            # Output: Hello, World this is Python
```

5. ⚠ Focus Esame

- **Nota:** Il metodo `.split()` senza argomenti divide per spazi bianchi di default.
- **Attenzione:** Le stringhe in Python sono *immutabili*. `x.upper()` non modifica `x`, ma restituisce una *nuova* stringa modificata.

---

6. Input/Output e Controllo Flusso

1. 📖 Spiegazione Approfondita

Interazione Utente:

- **Output:** `print()` scrive a schermo.
- **Input:** `input("Messaggio")` sospende l'esecuzione, mostra il messaggio e attende che l'utente scriva qualcosa seguito da Invio.

Valori Booleani (Truthiness): Per il controllo del flusso, Python valuta la "verità" delle espressioni. Sono considerati **False**: `0`, `False`, `None`, stringa vuota `""`. Tutto il resto è considerato **True**.

Strutture Condizionali (if-elif-else): Permettono di ramificare l'esecuzione.

- `if`: valuta la prima condizione.
- `elif` (else if): valuta condizioni successive se la precedente è falsa.
- `else`: esegue il codice se nessuna condizione precedente è vera.

2. 🛠 Sintassi & Comandi (Cleaning)

Esempio Input e If:

```
# L'input restituisce SEMPRE una stringa, necessario casting a int per
confronti numerici
x = int(input("Inserisci un numero: "))

if x > 10:
    print("Maggiore di 10")
elif x == 10:
    print("Uguale a 10")
else:
    print("Minore di 10")
```

Operatori Logici:

- `and`, `or`, `not` (al posto di `&&`, `||`, `!` del C).
- Confronto: `==`, `!=`, `>`, `<`, `>=`, `<=`.

5. ⚠ Focus Esame

- **Trabocchetto Classico:** La funzione `input()` restituisce sempre una **Stringa** (`str`). Se chiedi un'età e vuoi fare `if eta > 18`, il codice andrà in errore se non converti prima con `int(input(...))`.

---

7. Cicli (Loops)

1. 📖 Spiegazione Approfondita

I cicli permettono l'esecuzione ripetuta di blocchi di codice.

Ciclo While: Esegue il blocco finché la condizione rimane `True`. È ideale quando non si conosce a priori il numero di iterazioni necessarie (es. "continua a provare password finché non entri").

Ciclo For: In Python il `for` è diverso dal C (dove si incrementa un contatore). In Python, il `for` **itera su una sequenza** (lista, stringa, range). Esegue il blocco per ogni elemento presente nella collezione.

Funzione Range: Supporta il ciclo `for` generando sequenze numeriche.

- `range(stop)`: da 0 a stop-1.
- `range(start, stop)`: da start a stop-1.
- `range(start, stop, step)`: come sopra ma con incremento definito da step.

2. 🛠 Sintassi & Comandi (Cleaning)

While:

```
n = 0
while n < 5:
    print(n)
    n += 1 # Python non ha n++
```

For con Range:

```
# Stampa 0, 1, 2
for i in range(3):
    print("i:", i)
```

4. 🔗 Cross-Connection Master

In **Brute Force**, si usa spesso un ciclo `for` per iterare su una lista di password (`wordlist`), oppure un `while` per mantenere attiva una reverse shell finché il server non chiude la connessione.

---

8. Strutture Dati: Liste

1. 📖 Spiegazione Approfondita

Le **Liste** sono raccolte ordinate e mutabili di elementi. Possono essere eterogenee (contenere int, str e altre liste contemporaneamente). Si definiscono con parentesi quadre `[]`.

Accesso: Si usa l'indice numerico partendo da 0. **Modificabilità:** È possibile cambiare il valore di un elemento accedendo al suo indice (es. `lista[0] = "nuovo"`).

Metodi Principali:

- `append(elemento)`: Aggiunge in coda.
- `insert(indice, elemento)`: Inserisce in una posizione specifica traslando gli altri.
- `remove(valore)`: Rimuove la prima occorrenza di un valore specifico.
- `pop()`: Rimuove e restituisce l'ultimo elemento (o quello all'indice specificato).
- `del lista[i]`: Cancella l'elemento all'indice `i`.

2. 🛠 Sintassi & Comandi (Cleaning)

```
lista = [1, "ciao", 3.14]
lista.append("nuovo")       # [1, "ciao", 3.14, "nuovo"]
print(lista[1])             # Output: ciao
del lista[0]                # Rimuove l'1
```

3. 🧠 Analisi Tecnica & Memorizzazione

- **Concept:** Le liste sono vettori dinamici. Non serve specificare la dimensione all'inizio.

---

9. Strutture Dati: Dizionari

1. 📖 Spiegazione Approfondita

I **Dizionari** (`dict`) sono collezioni non ordinate di coppie **Chiave-Valore**. Si definiscono con parentesi graffe `{}`. A differenza delle liste (indicizzate per posizione numerica), i dizionari sono indicizzati per **chiave**. Le chiavi devono essere oggetti immutabili (stringhe, numeri), i valori possono essere qualsiasi cosa.

Operazioni:

- Accesso: `dizionario["chiave"]`.
- Modifica/Aggiunta: `dizionario["chiave"] = nuovo_valore`. Se la chiave esiste, aggiorna il valore; se non esiste, la crea.
- Rimozione: `pop("chiave")` o `del dizionario["chiave"]`.

Metodi di Iterazione:

- `.keys()`: Restituisce tutte le chiavi.
- `.values()`: Restituisce tutti i valori.

2. 🛠️ Sintassi & Comandi (Cleaning)

```
utente = {"nome": "Alice", "role": "admin"}
print(utente["role"])       # Output: admin
utente["role"] = "user"     # Modifica
utente["ip"] = "127.0.0.1"  # Aggiunta nuova coppia
```

4. 🔗 Cross-Connection Master

I dizionari sono onnipresenti nel parsing di dati **JSON** (API response) o nella gestione di header HTTP in script di attacco Web.

---

10. Funzioni

1. 📖 Spiegazione Approfondita

Le funzioni sono blocchi di codice riutilizzabili definiti dalla keyword `def`. Permettono di applicare il principio DRY (Don't Repeat Yourself). Possono accettare **parametri** in ingresso e restituire un risultato tramite `return`.

Passaggio Parametri:

1. **Posizionale:** L'ordine conta (il primo valore va al primo parametro).

2. **Per Nome (Keyword):** Si specifica `parametro=valore` nella chiamata. L'ordine non conta. È possibile mixarli, ma i posizionali devono precedere quelli per nome.

Scope (Visibilità):

- **Variabili Locali:** Definite dentro una funzione. Visibili solo lì.
- **Variabili Globali:** Definite fuori. Visibili ovunque in lettura.
- **Modifica Global:** Per modificare una variabile globale *dentro* una funzione, bisogna dichiararla con la keyword `global nome_var`. Altrimenti Python crea una nuova variabile locale con lo stesso nome.

2. 🛠 Sintassi & Comandi (Cleaning)

Definizione e Scope:

```
var_globale = 10

def raddoppia_globale():
    global var_globale  # Senza questo, var_globale sarebbe considerata
locale
    var_globale = var_globale * 2

raddoppia_globale()
print(var_globale) # Output: 20
```

5. ⚠ Focus Esame

- **Domanda:** Cosa succede se assegno un valore a una variabile globale dentro una funzione senza usare `global`?
- **Risposta:** Viene creata una nuova variabile *locale* che nasconde (shadowing) quella globale. La globale originale rimane invariata.

---

11. Moduli

1. 📖 Spiegazione Approfondita

I moduli sono file `.py` contenenti funzioni, classi e variabili che possono essere importati in altri script. Servono a organizzare il codice in file separati.

Metodi di Importazione:

1. `import modulo`: Importa tutto il modulo. Per usare una funzione serve la dot notation: `modulo.funzione()`.
2. `from modulo import funzione`: Importa solo una specifica funzione. Si usa direttamente: `funzione()`.
3. `from modulo import *`: Importa tutto nello spazio dei nomi corrente (sconsigliato per rischio conflitti di nomi).

2. 🛠 Sintassi & Comandi (Cleaning)

File `tools.py`:

```
def hack():
    return "Hacked!"
```

File `main.py`:

```
from tools import hack
print(hack())
```

3. 🧠 Analisi Tecnica & Memorizzazione

L'uso dei moduli è la base per utilizzare le potenti librerie di sicurezza di Python come `scapy` (reti), `requests` (web) o `socket` (connessioni).