Da neofita di Python a campione

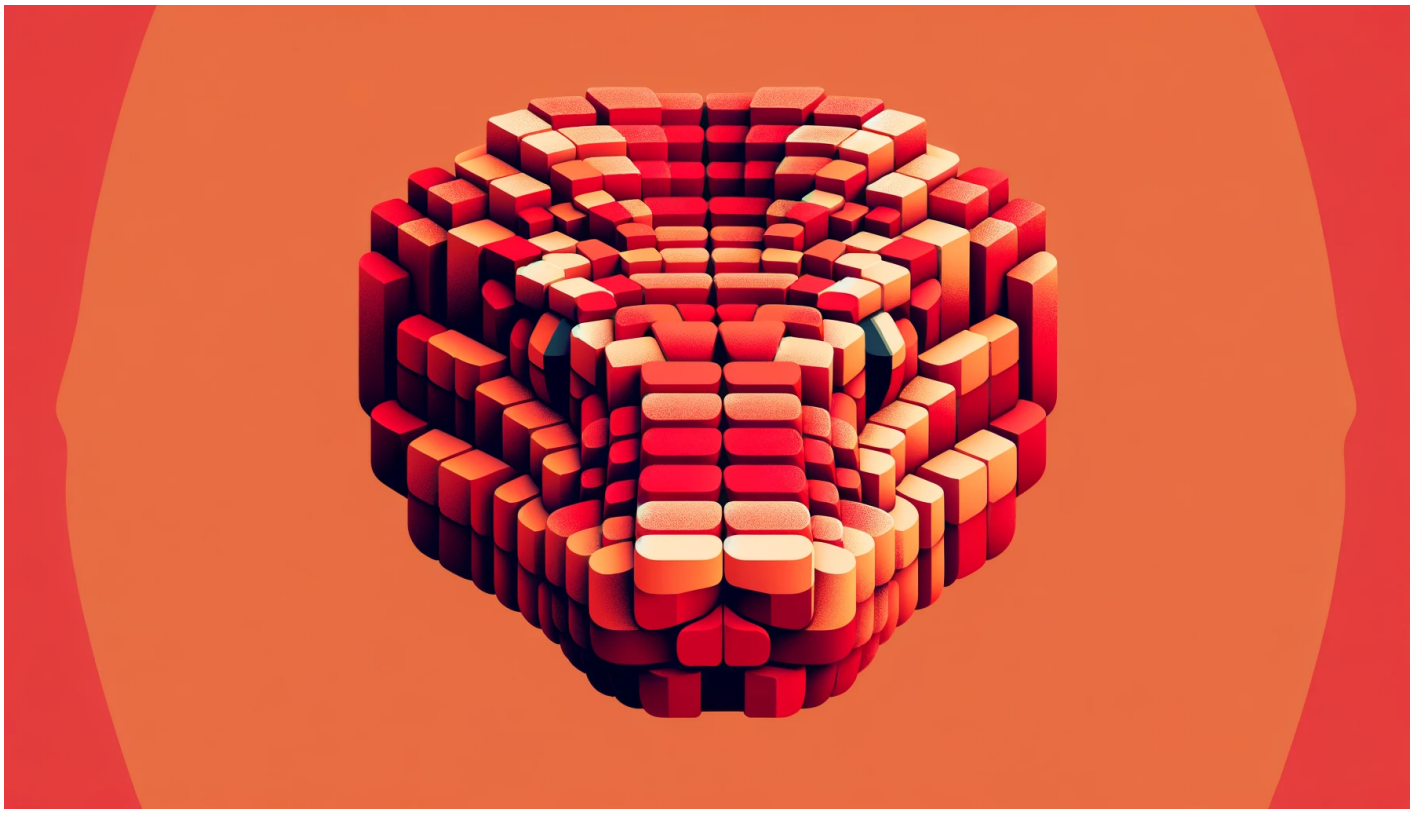
Corso di Python per tutti

Antonio Montano

2024-05-24

Indice

Prefazione



Parte I.

Prima parte: I fondamenti della programmazione

1. I linguaggi di programmazione, i programmi e i programmatori

Partiamo da alcuni concetti basilari che ci permettono di contestualizzare più facilmente quelli che introdurremo via via nel corso.

1.1. Definizioni

La **programmazione** è il processo di progettazione e scrittura di **istruzioni**, nella forma statica, ad esempio un file di testo, identificate come **codice sorgente**, che un computer può ricevere per eseguire compiti predefiniti. Queste istruzioni sono codificate in un **linguaggio di programmazione**, che traduce le idee e gli algoritmi del programmatore, in un formato comprensibile ed eseguibile dal computer.

Un **programma** informatico è una sequenza di istruzioni scritte per eseguire una specifica operazione o un insieme di operazioni su un computer. Queste istruzioni sono codificate in un linguaggio che il computer può comprendere e utilizzare per eseguire attività come calcoli, manipolazione di dati, controllo di dispositivi e interazione con l'utente. Pensate a un programma come a una ricetta di cucina. La ricetta elenca gli ingredienti necessari (dati) e fornisce istruzioni passo-passo (algoritmo) per preparare un piatto. Allo stesso modo, un programma informatico specifica i dati da usare e le istruzioni da seguire per ottenere un risultato desiderato.

Un linguaggio di programmazione è un linguaggio formale che fornisce un insieme di regole e sintassi per scrivere programmi informatici. Questi linguaggi permettono ai programmatori di comunicare con i computer e di creare software. Alcuni esempi di linguaggi di programmazione includono Python, Java, C++, SQL, Rust, Haskell, Prolog, C, Assembly, Fortran, JavaScript e altre centinaia (o forse migliaia).

1.2. Linguaggi naturali e di programmazione

I linguaggi di programmazione hanno dei punti in comune e delle differenze coi linguaggi naturali (come l'italiano o l'inglese). Quest'ultime sono principalmente:

1. Precisione e rigidità: I linguaggi di programmazione sono estremamente precisi e rigidi. Ogni istruzione deve essere scritta in un modo specifico affinché il computer possa comprenderla ed eseguirla correttamente. Anche un piccolo errore di sintassi può impedire il funzionamento di un programma.

2. Ambiguità: I linguaggi naturali sono spesso ambigui e aperti a interpretazioni. Le stesse parole possono avere significati diversi a seconda del contesto. I linguaggi di programmazione, invece, sono progettati per essere privi di ambiguità; ogni istruzione ha un significato preciso e univoco.

3. Vocabolario limitato: I linguaggi naturali hanno un vocabolario vastissimo e in continua espansione. I linguaggi di programmazione, al contrario, hanno un vocabolario limitato costituito da parole chiave e comandi definiti dal linguaggio stesso.

4. Forma di mediazione: I linguaggi naturali sono direttamente utilizzati per comunicare, quelli di programmazione non sono comprensibili immediatamente dai computer, ma devono essere tradotti in una forma opportuna per mezzo di programmi ad hoc.

1.3. Algoritmi

Un **algoritmo** è "un insieme di regole che definiscono con precisione una sequenza di operazioni" (Harold Stone, *Introduction to Computer Organization and Data Structures*, 1971 (Stone 1971)). Tale definizione, per quanto generica, coglie i due aspetti fondanti, cioè regole, intese come prescrizioni sintattiche o semantiche, che si traducono in operazioni richieste ad un agente, umano o informatico.

In altre parole, un algoritmo è una sequenza ben definita di passi o operazioni ben codificata, che, a partire da un input, produce un output in un tempo finito e, quindi, presenta la seguente serie di caratteristiche:

- Finitudine: L'algoritmo deve terminare dopo un numero finito di passi.
- Determinismo: Ogni passo dell'algoritmo deve essere definito in modo preciso e non ambiguo.
- Input: L'algoritmo riceve zero o più dati in ingresso.
- Output: L'algoritmo produce uno o più risultati.
- Effettività: Ogni operazione dell'algoritmo deve essere fattibile ed eseguibile in un tempo finito.

Gli algoritmi sono tradotti in codice sorgente attraverso un linguaggio di programmazione per creare programmi. In altre parole, un programma è la realizzazione pratica e funzionante degli algoritmi ideati dal programmatore.

1.4. Dal codice sorgente all'esecuzione

Per comprendere come un programma scritto in un linguaggio di programmazione passi dal file di testo contenente il codice sorgente all'esecuzione delle istruzioni da parte della CPU, è fondamentale capire che questo processo richiede un programma che interpreti il codice sorgente. Tale programma può essere un **compilatore** o un **interprete**, le due macrocategorie che definiscono come il codice sorgente viene tradotto ed eseguito.

Un **compilatore** è un programma che traduce l'intero codice sorgente di un programma scritto in un linguaggio di alto livello (come C o C++) in codice macchina, che è il linguaggio comprensibile direttamente dalla CPU. Questa traduzione avviene una sola volta, generando un file eseguibile che può essere eseguito direttamente dalla CPU.

Un **interprete**, invece, è un programma che esegue il codice sorgente direttamente, istruzione per istruzione, senza produrre un file eseguibile separato. L'interprete legge una riga di codice, la traduce in codice macchina e la esegue immediatamente. Questo processo viene ripetuto per ogni riga del codice sorgente.

È importante notare che alcuni linguaggi di programmazione possono essere sia compilati che interpretati, a seconda dell'implementazione disponibile. Ad esempio, Java utilizza sia la compilazione (per generare bytecode) che l'interpretazione, e la Java Virtual Machine (JVM) spesso utilizza anche la compilazione just-in-time (JIT) per tradurre il bytecode in codice macchina nativo durante l'esecuzione.

Detto ciò i passaggi macro perché un programma sia eseguito e possa produrre gli effetti pronosticati, sono:

1. Il programmatore scrive il codice sorgente utilizzando un editor di testo o un ambiente di sviluppo integrato (integrated development environment, IDE). Questo codice contiene le istruzioni del programma, scritte secondo la sintassi del linguaggio di programmazione scelto.
2. L'interpreto o il compilatore vengono eseguiti con input il programma e un componente, l'analizzatore lessicale, legge il codice sorgente e lo divide in lessemi, che sono sequenze di caratteri che corrispondono agli elementi *atomici* del linguaggio. Ogni lessema viene identificato come un token specifico, come una parola chiave, un operatore o un identificatore.
3. A seguire un secondo compoenente, il parser, riceve la sequenza di token dall'analizzatore lessicale e costruisce un albero di sintassi, che rappresenta la struttura grammaticale del programma. Il parser verifica che il codice rispetti le regole sintattiche del linguaggio.
4. Un altro componente effettua la verifica che il programma abbia un senso logico. Ad esempio, controlla che le variabili siano dichiarate prima di essere utilizzate e che i tipi di dati siano compatibili con le operazioni eseguite su di essi.
5. Il compilatore, a questo punto, genera una rappresentazione intermedia del programma, che è più vicina al linguaggio macchina ma ancora indipendente dall'architettura specifica del computer. Ciò è tipico dei linguaggi compilati, anche se alcuni interpreti possono generare un bytecode intermedio.
6. Il compilatore ottimizza codice intermedio al fine di migliorare le prestazioni del programma, riducendo il numero di istruzioni o migliorando l'efficienza delle operazioni.
7. Il codice intermedio ottimizzato viene tradotto in codice macchina, che è specifico per l'architettura del computer su cui il programma verrà eseguito.
8. Linking: Il codice macchina viene combinato con altre librerie e moduli necessari per formare un eseguibile completo.
9. Esecuzione: L'eseguibile viene caricato nella memoria del computer e il processore esegue le istruzioni, portando a termine le operazioni definite nel programma.

Nel caso di un interprete, i passaggi di generazione del codice intermedio e macchina possono essere sostituiti da una valutazione diretta delle istruzioni del programma, eseguendole una per una. In pratica, l'interprete traduce ogni singola istruzione del codice sorgente in un formato conprensibile dalla CPU e passa questa istruzione alla CPU stessa per l'esecuzione. Questo processo continua fino a quando tutte le istruzioni del programma non sono state eseguite.

1.5. Ciclo di vita del software

Un **software** è composto da uno o più programmi e, quando eseguito, realizza un compito con un grado di utilità specifico. La gerarchia concetttuale, dal più generale all'elemento più granulare, è: software, programmi, istruzioni.

Così come il disegno dei programmi è quello computazionale degli algoritmi, il disegno del software è funzionale per determinare i suoi obiettivi e architetturale per la decomposizione nei programmi.

Per creare il software, quindi, è necessario percorrere una sequenza di fasi ben definita che, concisamente, è:

- La progettazione di un'applicazione inizia con la fase di **analisi dei requisiti**, in cui si identificano cosa deve fare il software, chi sono gli utenti e quali sono i requisiti funzionali e non funzionali che deve soddisfare.

- Segue il **disegno funzionale** che dettaglia come ogni componente del sistema possa rispondere alle funzionalità richieste. In questa fase si descrivono le operazioni specifiche che ogni componente deve eseguire, utilizzando diagrammi di processo per rappresentare il flusso di attività al fine di rispondere ai requisiti.
- Il **disegno architetturale** riguarda l'organizzazione ad alto livello del sistema software. In questa fase si definiscono i componenti principali del sistema e come essi interagiscono tra di loro per supportare le attività di processo. Questo include la suddivisione del sistema in moduli o componenti, la definizione delle interfacce tra di essi e l'uso di tecniche di modellazione per rappresentare l'architettura del sistema.
- Una volta che l'architettura è stata progettata, si passa alla fase di **implementazione**, in cui i programmatori scrivono il codice sorgente nei linguaggi di programmazione scelti.
- Dopo l'implementazione, è essenziale verificare che il software funzioni correttamente:
  - **Testing**: Scrivere ed eseguire test per verificare che il software soddisfi i requisiti specificati. I test sono di diversi generi in funzione dell'oggetto di verifica, come test unitari, per segmenti di codice, test di integrazione, per componenti, e test di sistema nella sua interezza.
  - **Debugging**: Identificare e correggere gli errori (bug) nel codice. Questo può includere l'uso di strumenti di debugging per tracciare l'esecuzione del programma e trovare i punti in cui si verificano gli errori.
- Una volta che il software è stato testato e ritenuto pronto, si passa alla fasi di messa a disposizione delle funzionalità agli utenti (in inglese, *deployment*):
  - **Distribuzione**: Rilasciare il software agli utenti finali, che può includere l'installazione su server, la distribuzione di applicazioni desktop o il rilascio di app mobile.
  - **Manutenzione**: Continuare a supportare il software dopo il rilascio. Questo include la correzione di bug scoperti dopo il rilascio, l'aggiornamento del software per miglioramenti e nuove funzionalità, e l'adattamento a nuovi requisiti o ambienti.

La complessità del processo induce la necessità di avere dei team con qualità indiivduali diverse e il programmatore, oltre alle competenze specifiche, deve saper interpretare i vari artifatti di disegno e saperli tramutare in algoritmi e codice sorgente.

1.6. L'Impatto dell'intelligenza artificiale generativa sulla programmazione

Con l'avvento dell'**intelligenza artificiale generativa** (IA generativa), la programmazione ha subito una trasformazione significativa. Prima dell'IA generativa, i programmatori dovevano tutti scrivere manualmente ogni riga di codice, seguendo rigorosamente la sintassi e le regole del linguaggio di programmazione scelto. Questo processo richiedeva una conoscenza approfondita degli algoritmi, delle strutture dati e delle migliori pratiche di programmazione.

Inoltre, i programmatori dovevano creare ogni funzione, classe e modulo a mano, assicurandosi che ogni dettaglio fosse corretto, identificavano e correggevano gli errori nel codice con un processo lungo e laborioso, che comportava anche la scrittura di casi di test e l'esecuzione di sessioni di esecuzione di tali casi. Infine, dovebano scrivere documentazione dettagliata per spiegare il funzionamento del codice e facilitare la manutenzione futura.

1.6.1. Attività del programmatore con l'IA Generativa

L'IA generativa ha introdotto nuovi strumenti e metodologie che stanno cambiando il modo in cui i programmatori lavorano:

1. Generazione automatica del codice: Gli strumenti di IA generativa possono creare porzioni di codice basate su descrizioni ad alto livello fornite dai programmatori. Questo permette di velocizzare notevolmente lo sviluppo iniziale e ridurre gli errori di sintassi.
2. Assistenza nel debugging: L'IA può identificare potenziali bug e suggerire correzioni, rendendo il processo di debugging più efficiente e meno dispendioso in termini di tempo.
3. Ottimizzazione automatica: Gli algoritmi di IA possono analizzare il codice e suggerire o applicare automaticamente ottimizzazioni per migliorare le prestazioni.
4. Generazione di casi di test: L'IA può creare casi di test per verificare la correttezza del codice, coprendo una gamma più ampia di scenari di quanto un programmatore potrebbe fare manualmente.
5. Documentazione automatica: L'IA può generare documentazione leggendo e interpretando il codice, riducendo il carico di lavoro manuale e garantendo una documentazione coerente e aggiornata.

1.6.2. L'Importanza di imparare a programmare nell'era dell'IA generativa

Nonostante l'avvento dell'IA generativa, imparare a programmare rimane fondamentale per diverse ragioni. La programmazione non è solo una competenza tecnica, ma anche un modo di pensare e risolvere problemi. Comprendere i fondamenti della programmazione è essenziale per utilizzare efficacemente gli strumenti di IA generativa. Senza una solida base, è difficile sfruttare appieno queste tecnologie. Inoltre, la programmazione insegna a scomporre problemi complessi in parti più gestibili e a trovare soluzioni logiche e sequenziali, una competenza preziosa in molti campi.

Anche con l'IA generativa, esisteranno sempre situazioni in cui sarà necessario personalizzare o ottimizzare il codice per esigenze specifiche. La conoscenza della programmazione permette di fare queste modifiche con sicurezza. Inoltre, quando qualcosa va storto, è indispensabile sapere come leggere e comprendere il codice per identificare e risolvere i problemi. L'IA può assistere, ma la comprensione umana rimane cruciale per interventi mirati.

Imparare a programmare consente di sperimentare nuove idee e prototipare rapidamente soluzioni innovative. La creatività è potenziata dalla capacità di tradurre idee in codice funzionante. Sapere programmare aiuta anche a comprendere i limiti e le potenzialità degli strumenti di IA generativa, permettendo di usarli in modo più strategico ed efficace.

La tecnologia evolve rapidamente, e con una conoscenza della programmazione si è meglio preparati ad adattarsi alle nuove tecnologie e metodologie che emergeranno in futuro. Inoltre, la programmazione è una competenza trasversale applicabile in numerosi settori, dalla biologia computazionale alla finanza, dall'ingegneria all'arte digitale. Avere questa competenza amplia notevolmente le opportunità di carriera.

Infine, la programmazione è una porta d'accesso a ruoli più avanzati e specializzati nel campo della tecnologia, come l'ingegneria del software, la scienza dei dati e la ricerca sull'IA. Conoscere i principi della programmazione aiuta a comprendere meglio come funzionano gli algoritmi di IA, permettendo di contribuire attivamente allo sviluppo di nuove tecnologie.

2. I paradigmi di programmazione

I linguaggi di programmazione possono essere classificati in diversi tipologie in base al loro scopo e alla loro struttura.

Una delle classificazioni più importanti è quella del **paradigma di programmazione**, che definisce il modello e gli stili di risoluzione dei problemi che un linguaggio supporta per mezzo della codificazione degli algoritmi.

Tuttavia, è importante notare che molti linguaggi moderni sfruttano efficacemente più di un paradigma di programmazione, rendendo difficile assegnare un linguaggio a una sola categoria. Come ha affermato Bjarne Stroustrup, il creatore di C++:

> Le funzionalità dei linguaggi esistono per fornire supporto agli stili di programmazione. Per favore, non considerate una singola funzionalità di linguaggio come una soluzione, ma come un mattoncino da un insieme variegato che può essere combinato per esprimere soluzioni.
>
> I principi generali per il design e la programmazione possono essere espressi semplicemente:
>
> - Esprimere idee direttamente nel codice.
> - Esprimere idee indipendenti in modo indipendente nel codice.
> - Rappresentare le relazioni tra le idee direttamente nel codice.
> - Combinare idee espresse nel codice liberamente, solo dove le combinazioni hanno senso.
> - Esprimere idee semplici in modo semplice.
>
> Questi sono ideali condivisi da molte persone, ma i linguaggi progettati per supportarli possono differire notevolmente. Una ragione fondamentale per questo è che un linguaggio incorpora una serie di compromessi ingegneristici che riflettono le diverse necessità, gusti e storie di vari individui e comunità.(Stroustrup 2013, 10)

2.1. L'importanza dei paradigmi di programmazione

Comprendere i paradigmi di programmazione è fondamentale per diversi motivi:

- Approccio alla risoluzione dei problemi: Ogni paradigma offre una visione diversa su come affrontare e risolvere problemi. Conoscere vari paradigmi permette ai programmatori di scegliere l'approccio più adatto in base al problema specifico. Ad esempio, per problemi che richiedono una manipolazione di stati, la programmazione imperativa può essere più intuitiva. Al contrario, per problemi che richiedono trasformazioni di dati senza effetti collaterali, la programmazione funzionale potrebbe essere più adatta.
- Versatilità e adattabilità: I linguaggi moderni che supportano più paradigmi permettono ai programmatori di essere più versatili e adattabili. Possono utilizzare il paradigma più efficiente per diverse parti del progetto, migliorando sia la leggibilità che le prestazioni del codice.

- Manutenzione del codice: La comprensione dei paradigmi aiuta nella scrittura di codice più chiaro e manutenibile. Ad esempio, il paradigma orientato agli oggetti può essere utile per organizzare grandi basi di codice in moduli e componenti riutilizzabili, migliorando la gestione del progetto.
- Evoluzione professionale: La conoscenza dei vari paradigmi arricchisce le competenze di un programmatore, rendendolo più competitivo nel mercato del lavoro. Conoscere più paradigmi permette di comprendere e lavorare con una gamma più ampia di linguaggi di programmazione e tecnologie.
- Ottimizzazione del codice: Alcuni paradigmi sono più efficienti in determinate situazioni. Ad esempio, la programmazione concorrente è essenziale per lo sviluppo di software che richiede alta prestazione e scalabilità, come nei sistemi distribuiti. Comprendere come implementare la concorrenza in vari paradigmi permette di scrivere codice più efficiente.

2.2. Paradigma imperativo

La **programmazione imperativa**, a differenza della programmazione dichiarativa, è un paradigma di programmazione che descrive l'esecuzione di un programma come una serie di istruzioni che cambiano il suo stato. In modo simile al modo imperativo delle lingue naturali, che esprime comandi per compiere azioni, i programmi imperativi sono una sequenza di comandi che il computer deve eseguire in sequenza. Un caso particolare di programmazione imperativa è quella procedurale.

I linguaggi di programmazione imperativa si contrappongono ad altri tipi di linguaggi, come quelli funzionali e logici. I linguaggi di programmazione funzionale, come Haskell, non producono sequenze di istruzioni e non hanno uno stato globale come i linguaggi imperativi. I linguaggi di programmazione logica, come Prolog, sono caratterizzati dalla definizione di cosa deve essere calcolato, piuttosto che come deve avvenire il calcolo, a differenza di un linguaggio di programmazione imperativo.

L'implementazione hardware di quasi tutti i computer è imperativa perché è progettata per eseguire il codice macchina, che è scritto in stile imperativo. Da questa prospettiva a basso livello, lo stato del programma è definito dal contenuto della memoria e dalle istruzioni nel linguaggio macchina nativo del processore. Al contrario, i linguaggi imperativi di alto livello sono caratterizzati da un modello dati e istruzioni che risultano più facilmente usabili come strumenti di espressione di passi algoritmici.

2.2.1. Esempio in assembly

Assembly è una categoria di linguaggi di basso livello, cioè strettamente legati all'hardware del computer, tanto che ogni processore ha il suo *dialetto*. Un esempio di un semplice programma scritto per l'architettura x86, utilizzando la sintassi dell'assembler NASM (Netwide Assembler), è il seguente che effettua la somma di due numeri e stampa il risultato:

```
section .data
    num1 db 5           ; Definisce il primo numero
    num2 db 3           ; Definisce il secondo numero
    result db 0         ; Variabile per memorizzare il risultato
    msg db 'Result: ', 0 ; Messaggio di output

section .bss
    result_str resb 4   ; Buffer per la stringa del risultato

section .text
```

```
    global _start

_start:
    ; Somma num1 e num2
    mov al, [num1]      ; Carica il primo numero in AL
    add al, [num2]      ; Aggiunge il secondo numero a AL
    mov [result], al    ; Memorizza il risultato in result

    ; Converti il risultato in stringa ASCII
    mov eax, [result]   ; Carica il risultato in EAX
    add eax, '0'        ; Converti il valore numerico in carattere ASCII
    mov [result_str], eax ; Memorizza il carattere ASCII in result_str

    ; Stampa il messaggio
    mov eax, 4          ; syscall numero per sys_write
    mov ebx, 1          ; file descriptor 1 (stdout)
    mov ecx, msg        ; puntatore al messaggio
    mov edx, 8          ; lunghezza del messaggio
    int 0x80            ; chiamata di sistema

    ; Stampa il risultato
    mov eax, 4          ; syscall numero per sys_write
    mov ebx, 1          ; file descriptor 1 (stdout)
    mov ecx, result_str ; puntatore alla stringa del risultato
    mov edx, 1          ; lunghezza della stringa del risultato
    int 0x80            ; chiamata di sistema

    ; Terminazione del programma
    mov eax, 1          ; codice di sistema per l'uscita
    xor ebx, ebx        ; codice di ritorno 0
    int 0x80            ; interruzione per chiamare il kernel
```

Le sezioni del codice:

- La sezione `.data` definisce i dati statici `num1`, `num2`, `result` e `msg`.
- Sezione `.bss` alloca lo spazio per `result_str`, che conterrà la stringa del risultato.
- Sezione `.text`definisce `_start` come punto di ingresso del programma e:
  - Implementa la logica principale del programma.
  - Somma i valori di `num1` e `num2`.
  - Converte il risultato numerico in una stringa ASCII.
  - Utilizza chiamate al sistema operativo per scrivere il messaggio e il risultato su stdout.
  - Termina il programma.

L'assembly è usato nello sviluppo di:

- Sistemi operativi, ad esempio il kernel, che ha il controllo del sistema e i driver, cioè i programmi utili alla comunicazione coll'hardware.

- Applicazioni *embedded*: Microcontrollori di dispositivi medici, sistemi di controllo di veicoli, dispositivi IoT, ecc., cioè)dove è necessaria un'ottimizzazione estrema delle risorse computazionali.
- Applicazioni HPC (*high performance computing*): Il focus qui è eseguire calcoli intensivi e complessi in tempi relativamente brevi. Queste applicazioni richiedono un numero di operazioni per unità di tempo elevato e sono ottimizzate per sfruttare al massimo le risorse hardware disponibili, come CPU, GPU e memoria.

2.2.2. Esempio in Python

All'altro estremo della immediatezza di comprensione del testo del codice per un essere umano, troviamo Python, un linguaggio di alto livello noto per la leggibilità ed eleganza.

Ecco il medesimo esempio, visibilmente più conciso e certamente intuibile anche avendo basi limitate di programmazione:

```
num1 = 5                                                    ①
num2 = 3

result = num1 + num2                                        ②

print("Il risultato è: ", result)                           ③
```

① Definizione delle variabili che identificano gli addendi.
② Somma dei due numeri.
③ Stampa del risultato della somma.

2.2.3. Analisi comparativa

Assembly:

- Basso livello di astrazione: Assembly lavora direttamente con i registri della CPU e la memoria, quindi non astrae granché della complessità dell'hardware.
- Scarsa versatilità: Il linguaggio è progettato per una ben definita architettura e, quindi, ha una scarsa applicabilità ad altre, anche se alcuni dialetti di assembly presentano delle similitudini.
- Elevata precisione: Il programmatore ha un controllo dettagliato su ogni singola operazione compiuta dal processore, perché c'è una corrispondenza col codice macchina.
- Complessità: Ogni operazione deve essere definita esplicitamente e in sequenza, il che rende il codice più lungo e difficile da leggere.

Python:

- Alto livello di astrazione: Python fornisce un'astrazione più elevata sia dei dati che delle istruzioni, permettendo di ignorare i dettagli dei diversi hardware.
- Elevata semplicità: Il codice è più breve e leggibile, facilitando la comprensione e la manutenzione.
- Elevata versatilità: Il linguaggio è applicabile senza modifiche a un elevato numero di architetture hardware-software.

- Produttività: I programmatori possono concentrarsi sulla complessità intrinseca del problema, senza preoccuparsi di molti dettagli implementativi del processo di esecuzione.

2.3. Paradigma procedurale

La **programmazione procedurale** è un paradigma di programmazione, derivato da quella imperativa, che organizza il codice in unità chiamate procedure o funzioni. Ogni procedura o funzione è un blocco di codice che può essere richiamato da altre parti del programma, promuovendo la riutilizzabilità e la modularità del codice.

La programmazione procedurale è una naturale evoluzione della imperativa e uno dei paradigmi più antichi e ampiamente utilizzati. Ha avuto origine negli anni '60 e '70 con linguaggi come Fortan, COBOL e C, tutt'oggi rilevanti. Questi linguaggi hanno introdotto concetti fondamentali come funzioni, sottoprogrammi e la separazione tra codice e dati. Il C, in particolare, ha avuto un impatto duraturo sulla programmazione procedurale, diventando uno standard de facto per lo sviluppo di sistemi operativi e software di sistema.

I vantaggi principali sono:

- Modularità: La programmazione procedurale incoraggia la suddivisione del codice in funzioni o procedure più piccole e gestibili. Questo facilita la comprensione, la manutenzione e il riutilizzo del codice.

- Riutilizzabilità: Le funzioni possono essere riutilizzate in diverse parti del programma o in progetti diversi, riducendo la duplicazione del codice e migliorando l'efficienza dello sviluppo.

- Struttura e organizzazione: Il codice procedurale è generalmente più strutturato e organizzato, facilitando la lettura e la gestione del progetto software.

- Facilità di debug e testing: La suddivisione del programma in funzioni isolate rende più facile individuare e correggere errori, oltre a testare parti specifiche del codice.

D'altro canto, presenta anche degli svantaggi che hanno spinto i ricercatori a continuare l'innovazione:

- Scalabilità limitata: Nei progetti molto grandi, la programmazione procedurale può diventare difficile da gestire. La mancanza di meccanismi di astrazione avanzati, come quelli offerti dalla programmazione orientata agli oggetti, può complicare la gestione della complessità.

- Gestione dello stato: La programmazione procedurale si basa spesso su variabili globali per condividere stato tra le funzioni, il che può portare a bug difficili da individuare e risolvere.

- Difficoltà nell'aggiornamento: Le modifiche a una funzione possono richiedere aggiornamenti in tutte le parti del programma che la utilizzano, aumentando il rischio di introdurre nuovi errori.

- Meno Adatta per Applicazioni Moderne: Per applicazioni complesse e moderne che richiedono la gestione di eventi, interfacce utente complesse e modellazione del dominio, la programmazione procedurale può essere meno efficace rispetto ad altri paradigmi come quello orientato agli oggetti.

2.3.1. Funzioni e procedure

Nella programmazione procedurale, il codice è suddiviso in unità elementari chiamate **funzioni** e **procedure**. La differenza principale tra le due è la seguente:

- Funzione: Una funzione è un blocco di codice che esegue un compito specifico e restituisce un valore. Le funzioni sono utilizzate per calcoli o operazioni che producono un risultato. Ad esempio, una funzione che calcola la somma di due numeri in linguaggio C:

```c
int somma(int a, int b) {
  return a + b;
}
```

- Procedura: Una procedura è simile a una funzione, ma non restituisce un valore. È utilizzata per eseguire azioni o operazioni che non necessitano di un risultato. Ad esempio, una procedura che stampa un messaggio in Pascal:

```pascal
procedure stampaMessaggio;
begin
  writeln('Ciao, Mondo!');
end;
```

2.3.2. Creazione di librerie

Un altro aspetto importante della programmazione procedurale è la possibilità di creare **librerie**, che sono collezioni di funzioni e procedure riutilizzabili. Le librerie permettono di organizzare e condividere codice comune tra diversi progetti, aumentando la produttività e riducendo la duplicazione del codice, nonché abilitando un modello commerciale che mette a disposizione del software prodotto da aziende o comunità specializzate.

Esempio di una semplice libreria ipotetica di *somme* in C:

- File header (`mialibreria.h`):

```c
#ifndef MIALIBRERIA_H
#define MIALIBRERIA_H

int somma_interi(int a, int b);

char* somma_stringhe(const char* a, const char* b);

int somma_array(int arr[], int n);

void stampa_messaggio(const char* messaggio,
                      void* risultato,
                      char tipo);

#endif
```

- File di implementazione (`mialibreria.c`):

```c
#include "mialibreria.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int somma_interi(int a, int b) {
  return a + b;
}

char* somma_stringhe(const char* a, const char* b) {
  char* risultato = malloc(strlen(a) + strlen(b) + 1);                    ①

  if (risultato) {                                                        ②
    strcpy(risultato, a);                                                 ③
    strcat(risultato, b);                                                 ④
  }

  return risultato;
}

int somma_array_interi(int arr[], int n) {
  int somma = 0;

  for (int i = 0; i < n; i++) {
    somma += arr[i];
  }

  return somma;
}

void stampa_messaggio(const char* messaggio,
                      void* risultato,
                      char tipo) {
  printf("%s", messaggio);

  if (tipo == 'i') {
    printf("%d\n", *(int*)risultato);                                     ⑤
  } else if (tipo == 's') {
    printf("%s\n", (char*)risultato);                                     ⑥
  }
}
```

① Allocazione della memoria per la somma delle due stringhe e +1 per il carattere di terminazione `\0`.
② Controllo se la funzione `malloc` ha avuto successo nell'allocare la memoria richiesta. Se `risultato` è `NULL`, significa che `malloc` ha fallito e il blocco di codice all'interno dell'`if` viene saltato, evitando così di tentare di accedere a memoria non valida.
③ Se l'allocazione ha avuto successo, copia la prima stringa nel risultato.
④ Concatenazione della seconda stringa nel risultato.
⑤ Stampa del risultato se il tipo è intero.
⑥ Stampa del risultato se il tipo è una stringa.

- File principale (`main.c`):

```c
#include "mialibreria.h"
#include <stdio.h>
#include <stdlib.h>

int main() {
  int risultato = somma_interi(5, 3);                                    ①
  stampa_messaggio("Il risultato della somma di interi è: ",
                   &risultato, 'i');

  char* risultato_stringhe = somma_stringhe("Ciao, ", "mondo!");          ②
  stampa_messaggio("Il risultato della somma di stringhe è: ",
                   risultato_stringhe, 's');
  free(risultato_stringhe);                                              ③

  int array[] = {1, 2, 3, 4, 5};
  int risultato_array = somma_array_interi(array, 5);                    ④
  stampa_messaggio("Il risultato della somma dell'array di interi è: ",
                   &risultato_array, 'i');

  return 0;
}
```

① Chiamata della funzione per la somma di due interi.
② Chiamata della funzione per la somma di due stringhe (implementata come una concatenazione).
③ Liberazione della memoria allocata per la stringa risultante.
④ Chiamata della funzione per la somma di un array di interi.

E il medesimo, ma in Python:

```python
def somma_interi(a, b):
  return a + b

def somma_stringhe(a, b):                                                ①
  return a + b

def somma_array(arr):                                                    ②
  return sum(arr)                                                        ③

risultato_interi = somma_interi(3, 5)
print(f"Il risultato della somma di interi è: {risultato_interi}")

risultato_stringhe = somma_stringhe("Ciao, ", "mondo!")
print(f"Il risultato della somma di stringhe è: {risultato_stringhe}")

array_interi = [1, 2, 3, 4, 5]
risultato_array = somma_array(array_interi)
print(f"Il risultato della somma dell'array è: {risultato_array}")
```

① Il codice di `somma_interi` e `somma_stringhe` è identico e questo ci suggerisce che una delle due è ridondante.

② La funzione ora prende in input solo l'array e non c'è bisogno di inserire anche la sua dimensione.
③ In Python, per evitare errori quando si usa la funzione `sum`, l'array (o lista) deve contenere elementi che supportano l'operazione di addizione tra di loro. Tipicamente, si usano numeri (interi o a virgola mobile), ma è possibile anche sommare altri tipi di elementi se l'operazione di addizione è definita per quel tipo di dato.

Anche qui Python appare più semplice e immediato, sicuramente vincente sul piano della comprensione del codice e della immediatezza di utilizzo. In realtà, Python e C hanno sia una forte complementarietà sulle applicazioni, sia una dipendenza perché molte librerie e l'interprete stesso di Python sono in C.

2.4. Paradigma di orientamento agli oggetti

La **programmazione orientata agli oggetti** (in inglese *object-oriented programming*, OOP) è un paradigma di programmazione che organizza il software in termini di *oggetti*, ciascuno dei quali rappresenta un'istanza di una matrice detta *classe*. Una classe definisce un tipo di dato che include attributi (dati) e metodi (funzionalità). Gli oggetti interagiscono tra loro attraverso messaggi, permettendo una struttura modulare e intuitiva.

L'OOP è emersa negli anni '60 e '70 con il linguaggio Simula, il primo linguaggio di programmazione a supportare questo paradigma. Tuttavia, è stato con Smalltalk, sviluppato negli anni '70 da Alan Kay e altri presso Xerox PARC, che l'OOP ha guadagnato popolarità. Il paradigma è stato ulteriormente consolidato con il linguaggio C++ negli anni '80 e con Java negli anni '90, rendendolo uno dei più utilizzati per lo sviluppo software moderno. Oggi numerosi sono i linguaggi a oggetti, ad esempio Python, C#, Ruby, Java, Swift, Javascript, ecc. ed altri lo supportano come PHP (dalla versione 5) e financo il Fortran nella versione 2003.

Rispetto ai paradigmi precedenti, l'OOP introduce diversi concetti chiave che ineriscono al disegno architetturale di software:

- **Classe** e **oggetto**: La classe è un modello o schema per creare oggetti. Contiene definizioni di attributi e metodi. L'oggetto è un'istanza di una classe e rappresenta un'entità concreta nel programma con stato e comportamento mutevoli.
- **Incapsulamento**: Nasconde i dettagli interni di un oggetto e mostra solo le interfacce necessarie agli altri oggetti. Migliora la modularità e protegge l'integrità dei dati.
- **Ereditarietà** (relazione *is-a*): Permette a una classe di estenderne un'altra, ereditandone attributi e metodi. Favorisce il riuso del codice e facilita l'estensione delle funzionalità. Si usa quando una classe può essere considerata una specializzazione di un'altra. Ad esempio, un `Gatto` è un `Animale`, quindi la classe `Gatto` eredita dalla classe `Animale`.
- **Polimorfismo**: Consente a oggetti di classi diverse di essere trattati come oggetti di una classe comune. Facilita l'uso di un'interfaccia uniforme per operazioni diverse. Il polimorfismo è strettamente legato all'ereditarietà e permette di usare un metodo in modi diversi a seconda dell'oggetto che lo invoca. Ad esempio, un metodo `muovi()` può comportarsi diversamente se invocato su un oggetto di classe `Gatto` rispetto a un oggetto di classe `Uccello`, ma entrambi sono trattati come `Animale`.
- **Astrazione**: Permette di definire interfacce di alto livello per oggetti, senza esporre i dettagli implementativi. Facilita la comprensione e la gestione della complessità del sistema, perché, assieme a ereditarietà e polimorfismo, permette di pensare in modo più naturale, basando la decomposizione del problema anche su relazioni di tipo gerarchico e concettuale. Attraverso l'astrazione, si definiscono classi e interfacce che rappresentano concetti generici, come `Forma` o `Veicolo`, senza specificare i dettagli concreti delle implementazioni.

- **Composizione** (relazione *has-a*): Permette a una classe di contenere altre classi come parte dei suoi attributi. È una forma di relazione che indica che un oggetto è composto da uno o più oggetti di altre classi. Si usa quando una classe ha bisogno di utilizzare funzionalità di altre classi ma non rappresenta una specializzazione di quelle classi. Ad esempio, una classe `Auto` può avere un oggetto `Motore` come attributo, indicando che *un'Auto ha un Motore*.

I vantaggi principali dell'OOP sono:

- **Modularità**: Le classi e gli oggetti favoriscono la suddivisione del codice in moduli indipendenti, in una forma più granulare rispetto al paradigma procedurale. Non solo le istruzioni sono raggruppate per soddisfare una specifica operazione, ma possono essere viste come più operazioni su uno stato associato. La modularità è rafforzata dalle relazioni *has-a* e *is-a*, che aiutano a organizzare il codice in componenti logicamente separati e interconnessi.
- **Riutilizzabilità**: L'uso di classi e l'ereditarietà (relazione *is-a*) consentono di riutilizzare il codice in nuovi progetti senza riscriverlo, limitando gli effetti collaterali sul codice con cui interagiscono. Le classi base possono essere estese per creare nuove classi con funzionalità aggiuntive, mantenendo al contempo la compatibilità con il codice esistente.
- **Facilità di manutenzione**: L'incapsulamento e l'astrazione riducono la complessità perché permettono una migliore assegnazione logica dei principi usati nella progettazione dell'applicazione alle singole classi. Ciò facilita la manutenzione del codice, poiché nella modifica si possono individuare rapidamente le istruzioni impattate. La relazione *has-a* contribuisce ulteriormente alla manutenzione isolando le responsabilità all'interno delle classi.
- **Estendibilità**: Le classi possono essere estese (relazione *is-a*) per aggiungere nuove funzionalità senza modificare il codice già preesistente, riducendo così gli impatti per il codice che ne dipende. Questo approccio facilita l'integrazione di nuove caratteristiche e miglioramenti, mantenendo la stabilità del sistema.

Anche se sussistono dei caveat:

- Complessità iniziale: L'OOP può essere complesso da apprendere e implementare correttamente per i nuovi programmatori.
- Overhead di prestazioni: L'uso intensivo di oggetti può introdurre un overhead di memoria e prestazioni rispetto alla programmazione procedurale.
- Abuso di ereditarietà: L'uso improprio dell'ereditarietà può portare a gerarchie di classi troppo complesse e difficili da gestire, quindi, producendo un effetto opposto ad una delle ragioni di esistenza del concetto, cioè la semplicità di comunicazione della progettazione del software.

2.4.1. Esempio in Java

In questo esempio, la classe `Animale` rappresenta una tipo di dato generico con un attributo `nome` e un metodo `faiVerso`. La classe `Cane` specializza `Animale`, usando l'attributo `nome` e sovrascrivendo il metodo `faiVerso`, per fornire un'implementazione coerente colle sue caratteristiche. La classe `Main` crea un'istanza di `Cane` e chiama il suo metodo `faiVerso` (*annotato* con `@Override`[1]), dimostrando il polimorfismo e l'ereditarietà:

[1] In Java, l'annotazione `@Override` è opzionale, ma altamente consigliata. Non omettere l'annotazione `@Override` non causerà un errore di compilazione o di runtime. Tuttavia, l'uso di `@Override` offre dei vantaggi importanti perché, innanzitutto, il compilatore può verificare che il metodo stia effettivamente sovrascrivendo uno nella classe base e segnalare un errore in caso contrario. Inoltre, l'annotazione migliora la leggibilità del codice perché indica chiaramente al lettore come il metodo è inteso rispetto all'ereditarietà.

```java
class Animale {                                                    ①
  String nome;

  Animale(String nome) {
    this.nome = nome;
  }

  void faiVerso() {
    System.out.println("L'animale fa un verso");
  }
}

class Cane extends Animale {                                       ②

  Cane(String nome) {
    super(nome);
  }

  @Override                                                        ③
  void faiVerso() {
    System.out.println("Il cane abbaia");
  }
}

public class Main {
  public static void main(String[] args) {
    Animale mioCane = new Cane("Fido");

    mioCane.faiVerso();                                            ④
  }
}
```

① Definizione della classe `Animale` che ha il doppio compito di provvedere all'implementazione per una caratteristica comune (`nome` e `descrizione()`) e una particolare (`faiVerso()`).
② Definizione della classe derivata `Cane`.
③ `@Override` indica in esplicito che il `faiVerso()` del `Cane` sovrascrive (non eredita) il `faiVerso()` di `Animale`.
④ Output: `Il cane abbaia`.

In realtà, se gli oggetti devono rappresentare animali reali vorrà dire che non deve essere possibile crearne dalla matrice `Animale`. Vediamo, quindi, come implementare il medesimo esempio con una classe *astratta*, cioè una classe che non può essere usata per generare direttamente oggetti, sempre in Java.

Nel caso pratico, ogni animale ha il suo verso, quindi dobbiamo costringere il programmatore che vuole implementare classi corrispondenti ad animali reali, ad aggiungere tassativamente il metodo `faiVerso()` per comunicarne la caratteristica distintiva. Una modalità è marchiare `Animale` e il suo metodo da caratterizzare (`faiVerso()`), con costrutti ad hoc perché siano, rispettivamente, identificata come classe astratta (per mezzo della parola riservata `abstract`) e metodo da implementare. Al contempo, `Cane` non subisce specifiche modifiche sintattiche, ma deve rispettare il vincolo (implementare `faiVerso()`) perché, ereditando le caratteristiche di `Animale`, possa essere una classe concreta, cioè da cui si possono creare oggetti. Il codice risultate è:

```
abstract class Animale {                                                    (1)
  String nome;

  Animale(String nome) {
    this.nome = nome;
  }

  abstract String faiVerso();                                               (2)

  String descrizione() {                                                    (3)
    return "L'animale si chiama " + nome;
  }
}

class Cane extends Animale {                                                (4)

  Cane(String nome) {
    super(nome);
  }

  @Override
  String faiVerso() {
    return "Il cane abbaia";
  }
}

class Coccodrillo extends Animale {                                         (5)

  Coccodrillo(String nome) {
    super(nome);
  }

  @Override
  String faiVerso() {                                                       (6)
    return "";
  }
}

public class Main {
  public static void main(String[] args) {
    Animale mioCane = new Cane("Fido");

    System.out.println(mioCane.descrizione());                              (7)
    System.out.println(mioCane.faiVerso());                                 (8)

    Animale mioCoccodrillo = new Coccodrillo("Crocky");

    System.out.println(mioCoccodrillo.descrizione());                       (9)
    System.out.println(mioCoccodrillo.faiVerso());                          (10)
```

```
  }
}
```

① Definizione della classe astratta che ha il doppio compito di fornire una implementazione di default per una caratteristica comune (`nome`) e un vincolo di implementazione nelle classe derivate per una seconda caratteristica comune non implementabile nello stesso modo per tutte (`faiVerso()`).
② Metodo astratto `faiVerso()` che le classi corrispondenti ad animali reali dovranno implementare e che dovrà resituire una stringa.
③ Metodo concreto `faiVerso()` che restituisce una stringa.
④ Definizione della classe derivata `Cane`.
⑤ Definizione della classe derivata `Coccodrillo`.
⑥ Il coccodrillo non emette versi!
⑦ Stampa: `L'animale si chiama Fido`.
⑧ Stampa: `Il cane abbaia`.
⑨ Stampa: `L'animale si chiama Crocky`.
⑩ Non stampa nulla perché il coccodrillo non emette versi!

2.4.2. Template

I **template**, o generics, non sono specifici dell'OOP, anche se sono spesso associati a essa. I template permettono di scrivere funzioni, classi, e altri costrutti di codice in modo generico, cioè indipendente dal tipo dei dati che manipolano. Questo concetto è particolarmente utile per creare librerie e moduli riutilizzabili e flessibili.

Ad esempio, definiamo la classe `Box` nel modo seguente:

```
template <typename T>                                                    ①
class Box {                                                              ②
  T value;                                                               ③

public:
  void setValue(T val) { value = val; }                                  ④

  T getValue() { return value; }                                         ⑤
};
```

① La keyword `template` definisce un template di classe che può lavorare con qualsiasi tipo `T` specificato al momento dell'uso.
② Dichiarazione della classe `Box` che utilizza il template di tipo `T`.
③ Dichiarazione del membro dati `value` di tipo `T`, che rappresenta il valore contenuto nella scatola.
④ Metodo pubblico `setValue` che imposta il valore del membro dati `value` con il parametro `val` di tipo `T`.
⑤ Metodo pubblico `getValue` che restituisce il valore del membro dati `value` di tipo `T`.

`Box` può contenere un valore di qualsiasi tipo specificato al momento della creazione dell'istanza per mezzo del template `T`:

```
Box<int> intBox;                                                         ①

intBox.setValue(123);                                                    ②
int x = intBox.getValue();                                               ③
```

```
Box<std::string> stringBox;                                    ④

stringBox.setValue("Hello, World!");                           ⑤
std::string str = stringBox.getValue();                        ⑥
```

① Creazione di un'istanza di `Box` con tipo `int`, chiamata `intBox`.
② Chiamata del metodo `setValue` per impostare il valore di `intBox` a `123`.
③ Chiamata del metodo `getValue` per ottenere il valore di `intBox` e assegnarlo alla variabile `x` di tipo `int`.
④ Creazione di un'istanza di `Box` con tipo `std::string`, chiamata `stringBox`.
⑤ Chiamata del metodo `setValue` per impostare il valore di `stringBox` a `"Hello, World!"`.
⑥ Chiamata del metodo `getValue` per ottenere il valore di `stringBox` e assegnarlo alla variabile `str` di tipo `std::string`.

Anche nei linguaggi non orientati agli oggetti, i template trovano applicazione. Ad esempio, in Rust, un linguaggio di programmazione sistemistica non puramente OOP, il codice seguente restituisce il valore più grande di una lista:

```
fn largest<T: PartialOrd>(list: &[T]) -> &T {                  ①
    let mut largest = &list[0];                                ②

    for item in list {                                         ③
        if item > largest {                                    ④
            largest = item;                                    ⑤
        }
    }
    largest                                                    ⑥
}

fn main() {                                                    ⑦
    let numbers = vec![34, 50, 25, 100, 65];                   ⑧
    let max = largest(&numbers);                               ⑨

    println!("The largest number is {}", max);                 ⑩
}
```

① Definizione della funzione generica `largest` che accetta una lista di riferimenti a un tipo `T` che implementa il tratto `PartialOrd` e restituisce un riferimento a un valore di tipo `T`.
② Inizializzazione della variabile `largest` con il primo elemento della lista.
③ Iterazione attraverso ogni elemento della lista.
④ Controllo se l'elemento corrente `item` è maggiore di `largest`.
⑤ Se `item` è maggiore, aggiornamento della variabile `largest` con `item`.
⑥ Restituzione di `largest`, che è il riferimento al più grande elemento trovato nella lista.
⑦ Definizione della funzione `main`, punto di ingresso del programma.
⑧ Creazione di un vettore di numeri interi `numbers`.
⑨ Chiamata della funzione `largest` con un riferimento a `numbers` e assegnazione del risultato a `max`.
⑩ Stampa del valore più grande trovato nella lista usando la macro `println!`.

2.4.3. Metaprogrammazione

La metaprogrammazione è un paradigma che consente al programma di trattare il codice come dati, permettendo al codice di generare, manipolare o eseguire altro codice. Anche questo concetto non è esclusivo dell'OOP. In C++, la metaprogrammazione è strettamente legata ai template. Un esempio classico è la template metaprogramming (TMP), che permette di eseguire calcoli a tempo di compilazione.

Un esempio è il codice seguente di calcolo del fattoriale:

```
template<int N>
struct Factorial {
    static const int value = N * Factorial<N - 1>::value;                    ①
};

template<>
struct Factorial<0> {
    static const int value = 1;                                              ②
};
```

① Questa riga definisce un membro statico `value` della struttura `Factorial`. Per un dato `N`, il valore viene calcolato come `N` moltiplicato per il valore del fattoriale di `N - 1`. Questo è un esempio di ricorsione a livello di metaprogrammazione template.
② Questa riga è una specializzazione del template `Factorial` per il caso base quando `N` è 0. In questo caso, `value` è definito come `1`, terminando la ricorsione template.

La metaprogrammazione è presente anche in linguaggi non OOP come Lisp, che utilizza le macro per trasformare e generare codice. Un esempio è il codice proposto di seguito dove è definita la macro `when`, che prende due parametri in input, cioè `test` e `body`, ove `test` è un'espressione condizionale e `body` un insieme di istruzioni da eeseguire se la condizione è vera:

```
(defmacro when (test &rest body)
  `(if ,test
       (progn ,@body)))
```

Commento riga per riga:

1. Definizione di una macro chiamata `when`, che accetta un `test` e un numero variabile di espressioni (`body`).
2. La macro espande in un'espressione `if` che valuta `test`. Se `test` è vero, esegue le espressioni contenute in `body`.
3. `progn` è utilizzato per racchiudere ed eseguire tutte le espressioni in `body` in sequenza. L'operatore `,@` è usato per spalmare gli elementi di `body` nell'espressione `progn`.

Vediamo un esempio pratico di come si utilizza la macro `when`. Il test è valutare se `x` è maggiore di `10` e, nel caso, stampare `"x is greater than 10"` e poi assegnare `x` a `0`. Chiamiamo la macro con i due parametri:

```
(when (> x 10)
  (print "x is greater than 10")
  (setf x 0))
```

Commento riga per riga:

1. Invocazione della macro `when` con la condizione `> x 10`.
2. Se la condizione è vera, viene eseguita l'istruzione `(print "x is greater than 10")`, che stampa il messaggio.
3. Successivamente, viene eseguita l'istruzione `(setf x 0)`, che assegna il valore `0` a `x`.

Questo viene espanso in:

```
(if (> x 10)
    (progn
      (print "x is greater than 10")
      (setf x 0)))
```

Commento riga per riga:

1. L'istruzione `if` valuta la condizione `> x 10`.
2. Se la condizione è vera, viene eseguito il blocco `progn`.
3. All'interno del blocco `progn`, viene eseguita l'istruzione `(print "x is greater than 10")`.
4. Infine, viene eseguita l'istruzione `(setf x 0)` all'interno del blocco `progn`.

2.5. Paradigma dichiarativo

La **programmazione dichiarativa** è un paradigma di programmazione che si focalizza sul *cosa* deve essere calcolato piuttosto che sul *come* calcolarlo. In altre parole, i programmi dichiarativi descrivono il risultato desiderato senza specificare esplicitamente i passaggi per ottenerlo. Questo è in netto contrasto con la programmazione imperativa, dove si fornisce una sequenza dettagliata di istruzioni per modificare lo stato del programma.

La programmazione dichiarativa ha radici nella logica e nella matematica, ed è emersa come un importante paradigma negli anni '70 e '80 con l'avvento di linguaggi come Prolog (per la programmazione logica) e SQL (per la gestione dei database). La programmazione funzionale, con linguaggi come Haskell, è anch'essa una forma di programmazione dichiarativa.

I concetti principali associati alla programazione dichiarativa sono:

- Descrizione del risultato: I programmi dichiarativi descrivono le proprietà del risultato desiderato senza specificare l'algoritmo per ottenerlo. Esempio: In SQL, per ottenere tutti i record di una tabella con un certo valore, si scrive una query che descrive la condizione, non un algoritmo che scorre i record uno per uno.
- Assenza di stato esplicito: La programmazione dichiarativa evita l'uso esplicito di variabili di stato e di aggiornamenti di stato. Ciò riduce i rischi di effetti collaterali e rende il codice più facile da comprendere e verificare.
- Idempotenza: Le espressioni dichiarative sono spesso idempotenti, cioè possono essere eseguite più volte senza cambiare il risultato. Questo è particolarmente utile per la concorrenza e la parallelizzazione.

Il vantaggio principale è relativo alla sua chiarezza perché ci si concentra sul risultato desiderato piuttosto che sui dettagli di implementazione.

La programmazione imperativa specifica come ottenere un risultato mediante una sequenza di istruzioni, modificando lo stato del programma. La programmazione dichiarativa, al contrario, specifica cosa deve essere ottenuto senza descrivere i dettagli di implementazione. In termini di livello di astrazione, la programmazione dichiarativa si trova a un livello superiore rispetto a quella imperativa.

2.5.1. Linguaggi

Ecco una lista di alcuni linguaggi di programmazione dichiarativi:

1. SQL (Structured Query Language): Utilizzato per la gestione e l'interrogazione di database relazionali.
2. Prolog: Un linguaggio di programmazione logica usato principalmente per applicazioni di intelligenza artificiale e linguistica computazionale.
3. HTML (HyperText Markup Language): Utilizzato per creare e strutturare pagine web.
4. CSS (Cascading Style Sheets): Utilizzato per descrivere la presentazione delle pagine web scritte in HTML o XML.
5. XSLT (Extensible Stylesheet Language Transformations): Un linguaggio per trasformare documenti XML in altri formati.
6. Haskell: Un linguaggio funzionale che è anche dichiarativo, noto per la sua pura implementazione della programmazione funzionale.
7. Erlang: Un linguaggio utilizzato per sistemi concorrenti e distribuiti, con caratteristiche dichiarative.
8. VHDL (VHSIC Hardware Description Language): Utilizzato per descrivere il comportamento e la struttura di sistemi digitali.
9. Verilog: Un altro linguaggio di descrizione hardware usato per la modellazione di sistemi elettronici.
10. XQuery: Un linguaggio di query per interrogare documenti XML.

Questi linguaggi rappresentano diversi ambiti di applicazione, dai database alla descrizione hardware, e sono accomunati dall'approccio dichiarativo nel quale si specifica cosa ottenere piuttosto che come ottenerlo.

Nota

SQL è uno degli esempi più diffusi di linguaggio di programmazione dichiarativo. Le query SQL descrivono i risultati desiderati piuttosto che le procedure operative.
Una stored procedure in PL/SQL (Procedural Language/SQL) combina SQL con elementi di linguaggi di programmazione procedurali come blocchi di codice, condizioni e cicli. PL/SQL è quindi un linguaggio procedurale, poiché consente di specificare "come" ottenere i risultati attraverso un flusso di controllo esplicito, rendendolo non puramente dichiarativo. PL/SQL è utilizzato principalmente con il database Oracle.
Un'alternativa a PL/SQL è T-SQL (Transact-SQL), utilizzato con Microsoft SQL Server e Sybase ASE. Anche T-SQL estende SQL con funzionalità procedurali simili, consentendo la scrittura di istruzioni condizionali, cicli e la gestione delle transazioni. Come PL/SQL, T-SQL è un linguaggio procedurale e non puramente dichiarativo.
Esistono anche estensioni ad oggetti come il PL/pgSQL (Procedural Language/PostgreSQL) per il database PostgreSQL.

2.5.2. Esempi

Esempio di una query SQL che estrae tutti i nomi degli utenti con età maggiore di 30:

Certamente! Ecco il codice SQL con i commenti identificati da un ID progressivo e l'elenco esplicativo:

```sql
SELECT nome                ①
FROM utenti                ②
WHERE età > 30;            ③
```

① Seleziona la colonna `nome`.
② Dalla tabella `utenti`.
③ Per le righe dove la colonna `età` è maggiore di 30.

In Prolog, si definiscono fatti e regole che descrivono relazioni logiche. Il motore di inferenza di Prolog utilizza queste definizioni per risolvere query, senza richiedere un algoritmo dettagliato. Di seguito, sono definiti due fatti (le prime due righe) e due regole (la terza e la quarta) e quindi si effettua una query che dà come risultato `true`:

```prolog
genitore(padre, figlio).
genitore(madre, figlio).
antenato(X, Y) :- genitore(X, Y).
antenato(X, Y) :- genitore(X, Z), antenato(Z, Y).

?- antenato(padre, figlio).
```

Commento riga per riga:

1. Questa regola dichiara che `padre` è genitore di `figlio`.
2. Questa regola dichiara che `madre` è genitore di `figlio`.
3. Questa regola stabilisce che `X` è antenato di `Y` se `X` è genitore di `Y`.
4. Questa regola stabilisce che `X` è antenato di `Y` se `X` è genitore di `Z` e `Z` è antenato di `Y`.
5. Riga vuota.
6. Questa è una query che chiede se `padre` è un antenato di `figlio`.

2.6. Paradigma funzionale

La **programmazione funzionale** è un paradigma di programmazione che tratta il calcolo come la valutazione di funzioni matematiche ed evita lo stato mutabile e i dati modificabili. I programmi funzionali sono costruiti applicando e componendo funzioni. Questo paradigma è stato ispirato dal calcolo lambda, una formalizzazione matematica del concetto di funzione. La programmazione funzionale è un paradigma alternativo alla programmazione imperativa, che descrive la computazione come una sequenza di istruzioni che modificano lo stato del programma.

La programmazione funzionale ha radici storiche che risalgono agli anni '30, con il lavoro di Alonzo Church sul calcolo lambda. I linguaggi di programmazione funzionale hanno iniziato a svilupparsi negli anni '50 e '60 con Lisp, ma è stato negli anni '70 e '80 che linguaggi come ML e Haskell hanno consolidato questo paradigma. Haskell, in particolare, è stato progettato per esplorare nuove idee in programmazione funzionale e ha avuto un impatto significativo sulla ricerca e sulla pratica del software.

La programmazione funzionale è una forma di programmazione dichiarativa che si basa su funzioni pure e immutabilità. Entrambi i paradigmi evitano stati mutabili e si concentrano sul risultato finale, ma la programmazione funzionale utilizza funzioni matematiche come unità fondamentali di calcolo.

Concetti fondamentali:

- Immutabilità: I dati sono immutabili, il che significa che una volta creati non possono essere modificati. Questo riduce il rischio di effetti collaterali e rende il codice più prevedibile.
- Funzioni di prima classe e di ordine superiore: Le funzioni possono essere passate come argomenti a altre funzioni, ritornate da funzioni, e assegnate a variabili. Le funzioni di ordine superiore accettano altre funzioni come argomenti o restituiscono funzioni.
- Purezza: Le funzioni pure sono funzioni che, dato lo stesso input, restituiscono sempre lo stesso output e non causano effetti collaterali. Questo rende il comportamento del programma più facile da comprendere e prevedere.
- Trasparenza referenziale: Un'espressione è trasparentemente referenziale se può essere sostituita dal suo valore senza cambiare il comportamento del programma. Questo facilita l'ottimizzazione e il reasonig sul codice.
- Ricorsione: È spesso utilizzata al posto di loop iterativi per eseguire ripetizioni, poiché si adatta meglio alla natura immutabile dei dati e alla definizione di funzioni.
- Composizione di funzioni: Consente di costruire funzioni complesse combinando funzioni più semplici. Questo favorisce la modularità e la riusabilità del codice.

Il paradigma funzionale ha diversi vantaggi:

- Prevedibilità e facilità di test: Le funzioni pure e l'immutabilità rendono il codice più prevedibile e più facile da testare, poiché non ci sono stati mutabili o effetti collaterali nascosti.
- Concorrenza: La programmazione funzionale è ben adatta alla programmazione concorrente e parallela, poiché l'assenza di stato mutabile riduce i problemi di sincronizzazione e competizione per le risorse.
- Modularità e riutilizzabilità: La composizione di funzioni e la trasparenza referenziale facilitano la creazione di codice modulare e riutilizzabile.

E qualche svantaggio:

- Curva di apprendimento: La programmazione funzionale può essere difficile da apprendere per chi proviene da paradigmi imperativi o orientati agli oggetti, a causa dei concetti matematici sottostanti e della diversa mentalità necessaria.
- Prestazioni: In alcuni casi, l'uso intensivo di funzioni ricorsive può portare a problemi di prestazioni, come il consumo di memoria per le chiamate ricorsive. Tuttavia, molte implementazioni moderne offrono ottimizzazioni come la ricorsione di coda (in inglese, *tail recursion*).
- Disponibilità di librerie e strumenti: Alcuni linguaggi funzionali potrebbero non avere la stessa ampiezza di librerie e strumenti disponibili rispetto ai linguaggi imperativi più diffusi.

2.6.1. Linguaggi

Oltre a Haskell, ci sono molti altri linguaggi funzionali, tra cui:

- Erlang: Utilizzato per sistemi concorrenti e distribuiti.
- Elixir: Costruito a partire da Erlang, è utilizzato per applicazioni web scalabili.
- F#: Parte della piattaforma .NET, combina la programmazione funzionale con lo OOP.
- Scala: Anch'esso combina programmazione funzionale e orientata agli oggetti ed è interoperabile con Java.
- OCaml: Conosciuto per le sue prestazioni e sintassi espressiva.
- Lisp: Uno dei linguaggi più antichi, multi-paradigma con forti influenze funzionali.
- Clojure: Dialetto di Lisp per la JVM, adatto alla concorrenza.
- Scheme: Dialetto di Lisp spesso usato nell'educazione.
- ML: Linguaggio influente che ha portato allo sviluppo di OCaml e F#.
- Racket: Derivato da Scheme, usato nella ricerca accademica.

2.6.2. Esempio in Haskell

Di seguito due funzioni, la prima `sumToN` è pura e somma i primi `n` numeri. `(*2)` è una funzione che prende un argomento e lo moltiplica per 2 e ciò rende la seconda funzione `applyFunction` una vera funzione di ordine superiore, poiché accetta `(*2)` come argomento oltre ad una lista, producendo come risultato il raddoppio di tutti i suoi elementi:

Certamente! Ecco il codice con i commenti identificati da un ID progressivo e l'elenco esplicativo che include le descrizioni:

```haskell
sumToN :: Integer -> Integer
sumToN n = sum [1..n]                                                    ①

applyFunction :: (a -> b) -> [a] -> [b]
applyFunction f lst = map f lst                                          ②

main = do
    print (sumToN 10)                                                    ③
    print (applyFunction (*2) [1, 2, 3, 4])                              ④
```

① Definizione di una funzione pura che calcola la somma dei numeri da 1 a `n`.
② Funzione di ordine superiore che accetta una funzione e una lista.
③ Nel `main`, stampa il risultato di `sumToN 10`, che è 55.
④ Nel `main`, stampa il risultato di `applyFunction (*2) [1, 2, 3, 4]`, che è `[2, 4, 6, 8]`.

3. La sintassi dei linguaggi di programmazione

Abbiamo visto come i linguaggi di programmazione siano degli insiemi di regole formali con cui si scrivono programmi eseguibili da computer. I linguaggio di programmazione ha due componenti principali: la **sintassi** e la **semantica**.

Partiamo dalla **sintassi** di un linguaggio di programmazione che possiamo considerare come l'insieme di regole che definisce la struttura e la forma delle istruzioni, cioè le unità logiche di esecuzione del programma. È come la grammatica in una lingua naturale e stabilisce quali combinazioni di simboli sono considerate costrutti validi nel linguaggio.

Per esempio, la sintassi determina quali parole chiave, operatori, separatori e altri elementi sono ammessi e in quale ordine devono apparire. Una sintassi corretta è fondamentale per garantire che il programma sia privo di errori formali e possa essere eseguito senza problemi.

L'importanza della comprensione della sintassi è simile alla buona conoscenza per un linguaggio naturale:

1. Leggibilità del codice: Una corretta comprensione della sintassi permette di scrivere codice più chiaro e comprensibile agli altri programmatori. Una buona formattazione e organizzazione del codice facilita la manutenzione e la collaborazione su progetti di programmazione.
2. Efficienza nella risoluzione dei problemi: Conoscere bene la sintassi del linguaggio aiuta a trovare soluzioni efficienti ai problemi, poiché si è consapevoli delle strutture e delle funzionalità native del linguaggio che possono essere utilizzate per risolvere determinati compiti.
3. Sviluppo di codice robusto e sicuro: Una comprensione approfondita della sintassi aiuta a scrivere codice più robusto e sicuro, riducendo il rischio di bug e vulnerabilità nel software.
4. Adattabilità a nuovi contesti e tecnologie: Con una solida conoscenza della sintassi di base, è più facile imparare nuovi concetti, framework e librerie nel linguaggio di programmazione, consentendoci di sfruttare il lavoro e l'innovazione prodotta da altri.
5. Possibilità di esplorazione creativa: Capire la sintassi di un linguaggio offre la flessibilità necessaria per sperimentare e innovare, consentendo ai programmatori di creare soluzioni originali e creative ai problemi.

3.1. Token

Gli elementi atomici della sintassi sono i **token**. Essi compongono tutte le istruzioni e possono essere sia prodotti dal programmatore che generati dall'analisi del testo da parte dell'interprete o compilatore. La comprensione di quali token siano validi, ci permette sia di scrivere istruzioni corrette, sia di sfruttare appieno i costrutti del linguaggio:

- Parole chiave: Sono termini riservati del linguaggio che hanno significati specifici e non possono essere utilizzati per altri scopi, come `if`, `else`, `while`, `for`, ecc.
- Operatori: Simboli utilizzati per eseguire operazioni su identificatori e letterali, come `+`, `-`, `*`, `/`, `=`, `==`, ecc.

- Delimitatori: Caratteri utilizzati per separare elementi del codice, come punto e virgola (`;`), parentesi tonde (`()`), parentesi quadre (`[]`), parentesi graffe (`{}`), ecc.
- Identificatori: Nomi utilizzati per identificare variabili, funzioni, classi, e altri oggetti.
- Letterali: Rappresentazioni di valori costanti nel codice, come numeri (`123`), stringhe (`"hello"`), caratteri (`'a'`), ecc.
- Commento: Non fanno parte della logica del programma e sono ignorati nell'esecuzione.
- Spazi e tabulazioni: Sono gruppi di caratteri non visualizzabili e spesso ignorati.

Un **lessema** è una sequenza di caratteri nel programma sorgente che corrisponde al pattern di un token ed è identificata dall'**analizzatore lessicale** come un'istanza di quel token. Un **token** è una coppia composta da un nome di token e un valore attributo opzionale. Il nome del token è un simbolo astratto che rappresenta un tipo di unità lessicale, come una particolare parola chiave o una sequenza di caratteri di input che denota un identificatore. Un **pattern** è una descrizione della forma che possono assumere i lessemi di un token. Ad esempio, nel caso di una parola chiave come token, il pattern è semplicemente la sequenza di caratteri che forma la parola chiave. Per gli identificatori e altri token, il pattern è una struttura più complessa che corrisponde a molte stringhe.

Un esempio per visualizzare i concetti introdotti:

```
if x == 10:
```

- Token coinvolti:
  - `if`: Parola chiave.
  - `NAME`: Identificatore.
  - `EQEQUAL`: Operatore.
  - `NUMBER`: Letterale numerico.
  - `COLON`: Delimitatore.
- Lessemi:
  - Il lessema per il token `if` è la sequenza di caratteri "if".
  - Il lessema per il token `NAME` è "x".
  - Il lessema per il token `EQEQUAL` è "==".
  - Il lessema per il token `NUMBER` "10".
  - Il lessema per il token `COLON` ":".
- Pattern:
  - Il pattern per il token `if` è la stringa esatta "if".
  - Il pattern per un identificatore è una sequenza di lettere e numeri che inizia con una lettera.
  - Il pattern per l'operatore `==` è la stringa esatta "==".
  - Il pattern per un letterale numerico è una sequenza di cifre.
  - Il pattern per il delimitatore `:` è la stringa esatta ":".

3.2. Analizzatore lessicale e parser

L'**analizzatore lessicale** (o *lexer*) è un componente del compilatore o interprete che prende in input il codice sorgente del programma e lo divide in lessemi. Esso confronta ciascun lessema con i pattern definiti per il linguaggio di programmazione e genera una sequenza di token. Questi token sono poi passati al parser.

Ad esempio, il codice `if x == 10:` viene trasformato in una sequenza di token: `[IF, NAME(x), EQEQUAL, NUMBER(10), COLON]`.

Il **parser** è un altro componente del compilatore o interprete che prende in input la sequenza di token generata dall'analizzatore lessicale e verifica che la sequenza rispetti le regole sintattiche del linguaggio di programmazione. Il parser analizza i token per formare una struttura gerarchica che rappresenti le relazioni grammaticali tra di essi. Questa struttura interna è spesso un albero di sintassi (*parse tree* o *syntax tree*), che riflette la struttura grammaticale del codice sorgente, solitamente descritta usando una forma standard di notazione come la BNF (Backus-Naur Form) o varianti di essa [1]. L'albero di sintassi ottenuto viene utilizzato per le successive fasi di compilazione o interpretazione, come quella di analisi semantica e di generazione del codice eseguibile. Ad esempio, il parser può verificare che le espressioni aritmetiche siano ben formate, che le istruzioni siano correttamente annidate e che le dichiarazioni di variabili siano valide.

3.3. Espressioni

Un'espressione è una combinazione di lessemi che viene valutata per produrre un risultato. Le espressioni sono fondamentali nei linguaggi di programmazione perché permettono di eseguire calcoli, prendere decisioni e manipolare dati.

Ecco alcune tipologie di espressioni (notazioni in Python, ma non molto dissimili da altri linguaggi):

1. Espressioni aritmetiche: Combinano letterali numerici, identificatori valorizzabili in numeri e operatori aritmetici per eseguire calcoli matematici. Esempi: `5 + 3`, `y / 4.0`, `"Hello, " + "world!"`.

2. Espressioni logiche: Applicano operatori logici per valutare condizioni e produrre valori booleani (vero o falso) a letterali e identificatori. Esempi: `x or 5`, `not y`, `a and b`.

3. Espressioni di confronto: Confrontano due valori usando operatori di confronto e restituiscono valori booleani, sempre a partire da letterali e identificatori. Esempi: `x < y`, `x != 42`, `a >= b`.

4. Espressioni di chiamata a funzione: Invocano identificatori particolari, funzioni e metodi di oggetti, spesso con parametri definiti da identificatori e letterali, per eseguire operazioni più complesse. Esempi: `max(a, b)`, `sin(theta)`, `my_function(x, 42)`.

5. Espressioni di manipolazione di contenitori di dati: Creano e manipolano strutture dati come liste, dizionari, tuple e insiemi contenenti identificatori e letterali. Esempi: `[1, x, 3]`, `{ 'key': 'value' }`, `( 'y', 42 )`

6. Espressioni condizionali (ternarie): Valutano espressioni e restituiscono un valore basato sul risultato. Esempi: `x if x > y else y`, `'Even' if n % 2 == 0 else 'Odd'`.

[1] La BNF (Backus-Naur form o Backus normal form) è una metasintassi, ovvero un formalismo attraverso cui è possibile descrivere la sintassi di linguaggi formali (il prefisso meta ha proprio a che vedere con la natura circolare di questa definizione). Si tratta di uno strumento molto usato per descrivere in modo preciso e non ambiguo la sintassi dei linguaggi di programmazione, dei protocolli di rete e così via, benché non manchino in letteratura esempi di sue applicazioni a contesti anche non informatici e addirittura non tecnologici. Un esempio è la grammatica di Python.

Le espressioni si possono comporre in espressioni più complesse come accade per quelle matematiche pur che siano rispettate le regole di compatibilità tra operatori, identificatori e letterali; esempio `(x < y) and sin(theta)`.

3.4. Istruzioni semplici

Le **istruzioni semplici** sono operazioni atomiche secondo il linguaggio e sono costituite da tutti i tipi di lessemi per compiere operazioni di base. I linguaggi di programmazione presentano delle istruzioni *condivise* nel senso di fondamentali che hanno solo minime differenze ed altre più particolari, perché dipendenti da specificità dalla progettazione del linguaggio. Ciò potrebbe essere anche solo questione di sintassi più che rappresentanti nuovi concetti.

Di seguito un elenco di istruzioni semplici, alcune standard cioè presenti in tutti o la gran parte dei linguaggi considerati, altre specifiche ma che mettono in evidenza aspetti rilevanti di progettazione:

- Espressioni: È eseguibile dal compilatore o interprete, quindi, è una delle istruzioni semplici più importanti quando a sé stante, ma sono presenti anche all'interno di istruzioni semplici e composte. Alcuni esempi in vari linguaggi di somma di due identificatori:
  - In Python: `x + y`.
  - In Java: `x + y;`.
  - In C: `x + y;`.
  - In C++: `x + y;`.
- Dichiarazione di variabili: La dichiarazione di una variabile introduce una nuova variabile nel programma e specifica il suo tipo[2]. La dichiarazione non assegna necessariamente un valore iniziale alla variabile. Esempi:
  - In Python la dichiarazione avviene automaticamente con l'assegnazione, anche se è possibile annotare il tipo di una variabile, ad esempio `x: int = 5`, anche se ciò non costringe il programmatore a utilizzare `x` on interi.
  - In Java: `int x;`.
  - In C: `int x;`.
  - In C++: `int x;`.
- Assegnazione: Utilizza un operatore di assegnazione (ad esempio, `=`) per attribuire un valore rappresentato da un letterale, una espressione o un identificatore, ad un identificatore di variabile, che possiamo pensare come un nome simbolico rappresentante una posizione dove è memorizzato un valore. In alcuni linguaggi deve essere preceduta dalla dichiarazione. Esempio:
  - In Python:

    ```
    z = (x * 2) + (y / 2)
    ```

    * `z`: Identificatore della variabile.
    * `=`: Operatore di assegnazione.
    * `(x * 2)`: Espressione che moltiplica `x` per `2`.
    * `(y / 2)`: Espressione che divide `y` per `2`.

[2] Spiegheremo il concetto di tipo a breve, per ora si può pensare ad esso come l'insieme dei possibili valori e operazioni che si possono effettuare su di essi.

    - `+`: Operatore aritmetico che somma i risultati delle due espressioni in una più complessa. L'esecuzione dell'istruzione produce un risultato valido solo se `x` e `y` sono associate a valori numerici e ciò perché non tutte le istruzioni sintatticamente corrette sono semanticamente corrette. D'altronde ciò non deve essere preso come regola, perché se `*` fosse un operatore che ripete quanto a sinistra un numero di volte definito dal valore di destra e `/` la divisione del valore di sinistra in parti di numero pari a quanto a destra, allora `x` e `y` potrebbero essere stringhe.
  - In Java: `z = (x * 2) + (y / 2);`.
  - In C: `z = (x * 2) + (y / 2);`.
  - In C++: `z = (x * 2) + (y / 2);`.
- Assegnazione aumentata: Combina un'operazione e un'assegnazione in un'unica istruzione. Esempi per una assegnazione di una variabile del valore ottenuto dalla somma di quello proprio con il numero intero 1:
  - In Python: `x += 1` è come scrivere `x = x + 1`.
  - In Java: `x += 1;`.
  - In C: `x += 1;`.
  - In C++: `x += 1;`.
- Istruzioni di input/output: Sono espressioni particolari ma generalmente evidenziate perché permettono di interagire con l'utente o di produrre output, spesso con sintassi ad hoc. Esempi:
  - In Python: `print("Hello, World!")`.
  - In Java: `System.out.println("Hello, World!");`.
  - In C: `printf("Hello, World!\n");`.
  - In C++: `std::cout << "Hello, World!" << std::endl;`.
- Istruzioni di controllo del flusso: Permettono di interrompere o continuare l'esecuzione di cicli o di saltare a una specifica etichetta nel codice. Esempi:
  - In Python: `break`, `continue`.
  - In Java: `break;`, `continue;`.
  - In C: `break;`, `continue;`, `goto label;`.
  - In C++: `break;`, `continue;`, `goto label;`.
- Gestione della vita degli oggetti: Include la creazione, l'utilizzo e la distruzione dei dati presenti nella memoria del computer. In alcuni linguaggi ciò è parzialmente o totalmente a carico del programmatore, mentre, all'altro estremo, è completamente gestito dal linguaggio. Esempi:
  - Creazione di oggetti:
    - In C++: `int* ptr = new int;`.
    - In Java: `String str = new String("Hello, world!");`
    - In Python: Non è presente una istruzione specifica giacché l'espressione `MyClass()` crea un oggetto di tipo `MyClass`.
    - In C: La creazione di oggetti è spesso gestita manualmente attraverso l'allocazione di memoria dinamica con funzioni come `malloc`.
  - Distruzione di oggetti:
    - In Python: La gestione della memoria è automatica tramite il garbage collector.
    - In Java: In Java, la gestione della memoria è affidata al garbage collector.
    - In C++: `delete ptr;`.

    - In C: `free(ptr);` (richiede `#include <stdlib.h>`).
  - Eliminazione di variabili:
    - In Python: `del x`.

- Ritorno di valori: Utilizzata all'interno di funzioni per restituire un valore. Esempi:
  - In Python: `return x`.
  - In Java: `return x;`.
  - In C: `return x;`.
  - In C++: `return x;`.

- Generazione di eccezioni: Utilizzata per generare e inviare un'eccezione, cioè una interruzione della sequenza ordinaria delle istruzione per segnalare una anomalia. Esempi:
  - In Python: `raise ValueError("Invalid input")`.
  - In Java: `throw new IllegalArgumentException("Invalid input");`.
  - In C++: `throw std::invalid_argument("Invalid input");`.
  - In C: Non esiste un equivalente diretto, ma si possono utilizzare meccanismi come `setjmp` e `longjmp` per la gestione degli errori.

- Importazione di moduli: Permettono di importare moduli o parti di essi, cioè di utilizzare funzioni, classi, variabili e altri identificatori definiti in altri file o librerie. Esempi:
  - In Python: `import math`, `from math import sqrt`
  - In Java: `import java.util.List;`
  - In C: `#include <stdio.h>`
  - In C++: `#include <iostream>`

- Dichiarazioni globali e non locali: Permettono di dichiarare variabili che esistono nell'ambito globale o non locale. Esempi:
  - In Python: `global x`, `nonlocal y`.
  - In Java: Le variabili globali non sono supportate direttamente; si utilizzano campi statici delle classi.
  - In C: Le variabili globali sono dichiarate al di fuori di qualsiasi funzione.
  - In C++: Le variabili globali sono dichiarate al di fuori di qualsiasi funzione.

- Assert: Utilizzata per verificare se una condizione è vera e, in caso contrario, sollevare un'eccezione. Esempi:
  - In Python: `assert x > 0, "x deve essere positivo"`.
  - In Java: `assert x > 0 : "x deve essere positivo";`.
  - In C: `assert(x > 0);` (richiede `#include <assert.h>`).
  - In C++: `assert(x > 0);` (richiede `#include <cassert>`).

3.5. Istruzioni composte e blocchi di codice

Le **istruzioni composte** sono costituite da più istruzioni semplici e possono includere strutture di controllo del flusso, come condizioni (`if`), cicli (`for`, `while`) ed eccezioni (`try`, `catch`). Queste istruzioni sono utilizzate per organizzare il flusso di esecuzione del programma e possono contenere altre istruzioni semplici o composte al loro interno.

Un **blocco di codice** è una sezione del codice che raggruppa una serie di istruzioni che devono essere eseguite insieme. I blocchi di codice sono spesso utilizzati all'interno delle istruzioni composte per delimitare il gruppo di istruzioni che devono essere eseguite in determinate condizioni o iterazioni.

In molti linguaggi di programmazione, i blocchi di codice sono delimitati da parentesi graffe (`{}`), mentre in altri linguaggi, come Python, l'indentazione è utilizzata per indicare l'inizio e la fine di un blocco di codice.

Alcuni esempi di istruzione e blocco di codice:

- Esempio in C:

```c
if (x > 0) {                          ①
  printf("x è positivo\n");           ②

  y = x * 2;
}
```

  In questo esempio:

  1. `if (x > 0)` e quanto nelle parentesi graffe è un'istruzione composta. `if` è una parola chiave seguita da una espressione tra delimitatori.
  2. `{ printf("x è positivo\n"); y = x * 2; }` è un blocco di codice che viene eseguito se la condizione dell'istruzione `if` è vera. Sono presenti diversi delimitatori, una espressione e una istruzione di assegnamento.

- Esempio in Python:

```python
if x > 0:                             ①
  print("x è positivo")               ②

  y = x * 2
```

  In questo esempio:

  1. `if x > 0:` è un'istruzione composta.
  2. Le righe indentate sotto l'istruzione `if`, cioè `print("x è positivo")` e `y = x * 2`, costituiscono un blocco di codice che viene eseguito se la condizione dell'istruzione `if` è vera.

Alcuni esempi di istruzione e blocco di codice:

- Esempio in C:

```c
if (x > 0) {                          ①
  printf("x è positivo\n");           ②
  y = x * 2;
}
```

  In questo esempio:

  1. `if (x > 0)` e quanto nelle parentesi graffe è un'istruzione composta.
  2. `{ printf("x è positivo\n"); y = x * 2; }` è un blocco di codice che viene eseguito se la condizione dell'istruzione `if` è vera.

- Esempio in Python:

```
if x > 0:                                                    ①
  print("x è positivo")                                      ②
  y = x * 2
```

In questo esempio:

1. `if x > 0:` è un'istruzione composta.
2. Le righe indentate sotto l'istruzione `if` (`print("x è positivo")` e `y = x * 2`) costituiscono un blocco di codice che viene eseguito se la condizione dell'istruzione `if` è vera.

Di seguito sono elencate le istruzioni composte principali, con spiegazioni e semplici esempi di sintassi per Python, Java, C e C++:

- Condizionali (`if`, `else if`, `else`): Le istruzioni condizionali permettono l'esecuzione di blocchi di codice basati su espressioni logiche.
  - Python:

```python
if x > 0:
  print("x è positivo")

elif x == 0:
  print("x è zero")

else:
  print("x è negativo")
```

  - Java:

```java
if (x > 0) {
  System.out.println("x è positivo");
} else if (x == 0) {
  System.out.println("x è zero");
} else {
  System.out.println("x è negativo");
}
```

  - C:

```c
if (x > 0) {
  printf("x è positivo\n");
} else if (x == 0) {
  printf("x è zero\n");
} else {
  printf("x è negativo\n");
}
```

  - C++:

```cpp
if (x > 0) {
  std::cout << "x è positivo" << std::endl;
} else if (x == 0) {
  std::cout << "x è zero" << std::endl;
} else {
```

```
  std::cout << "x è negativo" << std::endl;
}
```

- Cicli (`for`): I cicli `for` permettono di iterare su un insieme di valori o di ripetere l'esecuzione di un blocco di codice per un numero specificato di volte.
  - Python:

```
for i in range(5):
  print(i)
```

  - Java:

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

  - C:

```
for (int i = 0; i < 5; i++) {
  printf("%d\n", i);
}
```

  - C++:

```
for (int i = 0; i < 5; i++) {
  std::cout << i << std::endl;
}
```

Cicli (`while`): I cicli `while` ripetono l'esecuzione di un blocco di codice finché una condizione specificata rimane vera.

- Python:

```
while x > 0:
  print(x)

  x -= 1
```

- Java:

```
while (x > 0) {
  System.out.println(x);

  x--;
}
```

- C:

```
while (x > 0) {
  printf("%d\n", x);

  x--;
}
```

- C++:

```cpp
while (x > 0) {
  std::cout << x << std::endl;

  x--;
}
```

- Gestione delle eccezioni (`try`, `catch`): Le istruzioni di gestione delle eccezioni permettono di gestire errori o condizioni anomale che possono verificarsi durante l'esecuzione del programma.

  - Python:

```python
try:
  value = int(input("Inserisci un numero: "))

except ValueError:
  print("Input non valido")
```

  - Java:

```java
try {
  int value = Integer.parseInt(input);

} catch (NumberFormatException e) {
  System.out.println("Input non valido");
}
```

  - C++:

```cpp
try {
  int value = std::stoi(input);

} catch (const std::invalid_argument& e) {
  std::cout << "Input non valido" << std::endl;
}
```

  - C: C non ha un supporto nativo per la gestione delle eccezioni, ma si possono usare meccanismi come `setjmp` e `longjmp`.

```c
#include <setjmp.h>

jmp_buf buf;                                                    ①

void error() {
  longjmp(buf, 1);                                              ②
}

int main() {
  if (setjmp(buf)) {                                            ③
    printf("Errore rilevato\n");                                ④

  } else {
    error();                                                    ⑤
```

```
  }
  return 0;
}
```

① Dichiarazione di una variabile di tipo `jmp_buf`.
② Salta al punto salvato in buf con valore di ritorno 1.
③ Salva il contesto di esecuzione attuale in `buf`.
④ Esegue se `longjmp` viene chiamato.
⑤ Chiama la funzione error, che salta indietro al punto `setjmp`.

- Selezione multipla (`switch`, `case`, `default`): Le istruzioni di selezione multipla permettono di eseguire uno tra diversi blocchi di codice basati sul valore di un'espressione.
  - Python (a partire da Python 3.10 con `match`):

```
match x:
  case 0:
    print("x è zero")

  case 1:
    print("x è uno")

  case _:
    print("x è un altro numero")
```

  - Java:

```
switch (x) {
  case 0:
    System.out.println("x è zero");

    break;

  case 1:
    System.out.println("x è uno");

    break;

  default:
    System.out.println("x è un altro numero");

    break;
}
```

  - C:

```
switch (x) {
  case 0:
    printf("x è zero\n");

    break;
```

```
  case 1:
    printf("x è uno\n");

    break;

  default:
    printf("x è un altro numero\n");

    break;
}
```

– C++:

```
switch (x) {
  case 0:
    std::cout << "x è zero" << std::endl;

    break;

  case 1:
    std::cout << "x è uno" << std::endl;

    break;

  default:
    std::cout << "x è un altro numero" << std::endl;

    break;
}
```

3.6. Organizzazione delle istruzioni in un programma

Certamente, ecco il paragrafo completato:

3.7. Organizzazione delle istruzioni in un programma

Il programma è solitamente salvato in un file di testo in righe. Queste righe possono essere classificate in righe fisiche e righe logiche.

Una **riga fisica** è una linea di testo nel file sorgente del programma, terminata da un carattere di a capo.

Esempio:

```
int x = 10;                                                    ①
```

① Questa è una riga fisica.

Una **riga logica** è una singola istruzione, che può estendersi su una o più righe fisiche.

Esempio di riga logica con più righe fisiche:

```
int y = (10 + 20 + 30 +                    ①
         40 + 50);                         ②
```

① Prima riga fisica della riga logica.
② Seconda riga fisica della riga logica.

Il concetto di righe fisiche e logiche esiste perché le istruzioni (o righe logiche) possono essere lunghe e composte, richiedendo più righe fisiche per migliorare la leggibilità e la gestione del codice.

In molti linguaggi di programmazione, l'uso di righe fisiche e logiche facilita l'organizzazione e la formattazione del codice. Ad esempio:

- **Python** utilizza l'indentazione per definire i blocchi di codice, quindi una riga logica che si estende su più righe fisiche deve continuare con una corretta indentazione. Inoltre, è possibile usare il carattere di continuazione (\) per indicare che una riga logica prosegue sulla riga successiva:

```
result = (10 + 20 + 30 + \
          40 + 50)
```

- **Java** e **C** utilizzano le parentesi graffe ({}) per delimitare i blocchi di codice, e le istruzioni possono estendersi su più righe fisiche senza il bisogno di un carattere di continuazione, grazie al punto e virgola (;) che termina le istruzioni:

```
int y = (10 + 20 + 30 +
         40 + 50);
```

```
int y = (10 + 20 + 30 +
         40 + 50);
```

L'uso corretto di righe fisiche e logiche migliora la leggibilità del codice, rendendolo più facile da capire e mantenere. Inoltre, una buona formattazione del codice facilita il lavoro di squadra, poiché gli sviluppatori possono facilmente seguire e comprendere la logica implementata da altri.

4. Le variabili e le funzioni

La **semantica** di un linguaggio di programmazione definisce il significato delle istruzioni sintatticamente corrette. In altre parole, la semantica specifica cosa fa un programma quando viene eseguito, descrivendo l'effetto delle istruzioni sullo stato del sistema. Gli elementi semantici sono numerosi, possono essere anche complessi e non tutti presenti in uno specifico linguaggio.

Tra i primi elementi semantici, richiamati già molte volte, troviamo le variabili e le funzioni.

4.1. Variabili

Le **variabili** sono uno dei concetti fondamentali nella programmazione, essenziali per la manipolazione e la gestione dei dati. Una variabile è un nome simbolico associato a una locazione di memoria che può contenere uno o più valori di un certo tipo di dato. Questo concetto permette agli sviluppatori di astrarre dalla memoria fisica e concentrarsi sulla logica del programma.

La gestione delle variabili varia tra i diversi linguaggi di programmazione, quindi esploreremo le variabili con particolare attenzione a Python, Java, C e C++.

4.1.1. Dichiarazione e inizializzazione

La dichiarazione di una variabile è il processo mediante il quale si introduce una variabile nel programma, specificandone il nome e, in molti casi, il tipo di dato che essa può contenere. Questo processo informa il compilatore o l'interprete che una certa variabile esiste e può essere utilizzata nel codice. Abbiamo visto che esiste una istruzione specifica in alcuni linguaggi, mentre in altri è implicita nella assegnazione.

L'inizializzazione di una variabile è il processo di assegnazione di un valore iniziale alla variabile. Questo può avvenire contestualmente alla dichiarazione o in un'istruzione separata successiva, quella di assegnamento.

Esempi:

- In Python, le variabili sono dichiarate automaticamente al momento dell'assegnazione del valore. Non è necessario specificare il tipo di dato, poiché Python è dinamicamente tipizzato, cioè determina durante l'esecuzione il tipo di dato del valore associato alla variabile.

```
x = 10          ①
x = "Hello"     ②
```

① Dichiarazione e inizializzazione.
② Cambia il tipo di `x` dinamicamente a stringa.

- Java: In Java, le variabili devono essere dichiarate con un tipo di dato esplicito. La dichiarazione può avvenire contestualmente all'inizializzazione o separatamente.

```
int x;          ①
x = 10;         ②
int y = 20;     ③
```

1. Dichiarazione.
2. Inizializzazione.
3. Dichiarazione e inizializzazione.

- C: In C, la dichiarazione delle variabili richiede la specifica del tipo di dato. La dichiarazione e l'inizializzazione possono essere separate o combinate.

```
int x;          ①
x = 10;         ②
int y = 20;     ③
```

1. Dichiarazione.
2. Inizializzazione.
3. Dichiarazione e inizializzazione.

- C++: Simile al C, ma con funzionalità aggiuntive come l'inizializzazione a lista.

```
int x;                          ①
x = 10;                         ②
int y = 20;                     ③
int z{30};                      ④
int arr[5] = {1, 2, 3, 4, 5};   ⑤
```

① Dichiarazione.
② Inizializzazione.
③ Dichiarazione e inizializzazione.
④ Dichiarazione e inizializzazione a lista di `z` con l'intero 30.
⑤ Dichiarazione e inizializzazione a lista di un array con 5 valori predefiniti.

4.1.2. Ambito delle variabili

L'ambito di una variabile rappresenta la porzione del codice in cui l'identificatore della variabile è definito e, quindi, può essere utilizzato. Gli approcci dei diversi linguaggi sono diversi, infatti Java, C e C++ hanno una gestione dell'ambito delle variabili per cui quelle dichiarate all'interno di un blocco sono limitate a quel blocco e non sono visibili al di fuori di esso. In Python, invece, le variabili definite all'interno di un blocco di un'istruzione composta (come un ciclo for o una condizione if) rimangono accessibili anche al di fuori del blocco, purché siano ancora nel medesimo ambito di funzione o modulo e, soprattutto quel blocco sia stato eseguito.

- Globale: Le variabili globali sono dichiarate al di fuori di qualsiasi blocco e sono accessibili ovunque nel programma.

```
int globalVar = 10;                 ①

void function() {                   ②
  printf("%d\n", globalVar);        ③
}
```

① Variabile globale.
② Inizio del blocco.
③ Accesso alla variabile globale.

- Locale: Le variabili locali sono dichiarate all'interno di un blocco, come una funzione o un loop, e sono accessibili solo all'interno di quel blocco.

```
public class Main {
  public static void main(String[] args) {
    if (true) {
      int x = 10;                                                    ①
    }

    System.out.println(x);                                           ②

    for (int i = 0; i < 10; i++) {
      int y = i;                                                     ③
    }

    System.out.println(y);                                           ④
  }
}
```

① Variabile locale al blocco `if`.
② Errore: `x` non è visibile qui!
③ Variabile locale al blocco `for`.
④ Errore: `y` non è visibile qui.

- In Python, una variabile definita all'interno di un blocco di un'istruzione composta, come all'interno di un ciclo `for` o di una condizione `if`, rimane accessibile anche dopo l'esecuzione del blocco:

```
for i in range(10):
  loopVar = i  #

print(loopVar)  #
```

1. Variabile locale al ciclo.
2. `loopVar` è ancora accessibile qui.

4.1.3. Visibilità delle variabili

La visibilità si riferisce alla possibilità che in una regione di codice una certa variabile possa essere *vista* e utilizzata. Anche se correlata all'ambito, la visibilità può essere influenzata da altri fattori come la modularità e gli spazi di nomi (*namespace*).

- Consideriamo un esempio in C++ per illustrare la differenza tra ambito e visibilità:

```
#include <iostream>

int globalVar = 10;                                                  ①

void function() {
```

```cpp
  int localVar = 5;                                                    ②

  std::cout << globalVar << std::endl;                                 ③
  std::cout << localVar << std::endl;                                  ④
}

int main() {
  function();

  std::cout << globalVar << std::endl;                                 ⑤
  std::cout << localVar << std::endl;                                  ⑥

  return 0;
}
```

① Variabile globale (ambito globale).
② Variabile locale (ambito locale alla funzione).
③ Visibilità `globaleVar` all'interno della funzione.
④ Visibilità `localVar` all'interno della funzione.
⑤ Visibilità `globalVar` all'interno di `main`.
⑥ Errore: `localVar` non è visibile qui (ambito locale alla funzione function).

- In Python, le variabili definite all'interno di una funzione sono locali a quella funzione, ma le variabili definite all'interno di un blocco (come un ciclo `for` o un `if`) sono visibili all'interno della funzione o del modulo in cui si trovano:

```python
globalVar = 10                                                         ①

def function():
  localVar = 5                                                         ②

  if True:
    blockVar = 20                                                      ③

  print(localVar)                                                      ④
  print(blockVar)                                                      ⑤

function()

print(globalVar)                                                       ⑥
print(localVar)                                                        ⑦
print(blockVar)                                                        ⑧
```

① Variabile globale.
② Variabile locale.
③ Visibile all'interno della funzione.
④ Visibile.
⑤ Visibile.
⑥ Visibile.
⑦ Errore: non visibile al di fuori della funzione.
⑧ Errore: non visibile al di fuori della funzione.

4.1.4. Durata di vita degli oggetti referenziati

La durata di vita descrive per quanto tempo un oggetto rimane in memoria durante l'esecuzione del programma. Questo è distinto dalla variabile (o puntatore) che fa riferimento all'oggetto.

In alcuni linguaggi di programmazione è presente il **garbage collector**, cioè un processo avviato dal compilatore o interprete che si occupa di rendere nuovamente disponibili le aree di memoria precedentemente occupate da oggetti non più referenziati da variabili. Questo accade quando l'esecuzione del programma raggiunge regioni di codice dove quelle variabili non sono più visibili. In questo modo, la visibilità è legata alla durata di vita degli oggetti, rendendo la gestione della memoria non più una preoccupazione del programmatore.

- Variabile automatica: L'oggetto esiste solo durante l'esecuzione del blocco di codice in cui è stata dichiarata la variabile a cui è associato.

```
void function() {
  int autoVar = 10;                                                    ①
}

printf("autoVar cancellata!");                                         ②
```

① Dichiarazione di `autoVar` e creazione in memoria di un oggetto corrispondente all'intero 10.
② Prima di questa istruzione l'oggetto 10 non è più presente in memoria.

- Variabile statica: La variabile esiste per tutta la durata del programma, ma è accessibile solo all'interno del blocco in cui è dichiarata.

```
void function() {
  static int staticVar = 10;                                           ①
}
```

① Variabile statica ottenuta con una parola chiave ad hoc in fase di dichiarazione.

- Variabile dinamica: Le variabili dinamiche sono utilizzate per riservare memoria che persiste oltre la durata del blocco di codice in cui sono state create. L'oggetto è creato in memoria e deve essere cancellato esplicitamente dall'utente, utilizzando funzioni di gestione della memoria come `delete`. La variabile che punta all'oggetto è separata dall'oggetto stesso, quindi se la variabile non è più visibile, l'oggetto continuerà a rimanere in memoria e non sarà più eliminabile, causando una perdita di memoria (*memory leak*).

```
#include <iostream>

void function() {
  int* dynamicVar = new int(10);                                       ①
  std::cout << "dynamicVar: " << *dynamicVar << std::endl;              ②
}

int main() {
  function();                                                          ③

  int* safeDynamicVar = new int(20);                                   ④

  std::cout << "safeDynamicVar: " << *safeDynamicVar << std::endl;     ⑤

  delete safeDynamicVar;                                               ⑥
```

```
  return 0;
```

① Allocazione dinamica di un intero all'interno della funzione `function()`.
② Stampa del valore puntato da `dynamicVar`. Prima della chiusura del blocco non viene deallocata `dynamicVar` per dimostrare il problema di perdita di memoria
③ Chiamata alla funzione `function()`. Dopo l'uscita dalla funzione, `dynamicVar` non è più accessibile, causando una perdita di memoria poiché non è stata deallocata.
④ Allocazione dinamica di un intero.
⑤ Stampa del valore puntato da `safeDynamicVar`.
⑥ Deallocazione dinamica dell'intero. Corretto uso di allocazione e deallocazione dinamica.

- In Python, la gestione della memoria è automatica grazie al garbage collector. Quando non ci sono più riferimenti di variabili a un oggetto, il garbage collector lo rimuove dalla memoria.

4.2. Funzioni

Le funzioni sono blocchi di codice riutilizzabili che eseguono una serie di istruzioni. Questi costrutti sono fondamentali per la strutturazione e la modularizzazione del codice, consentendo di definire operazioni che possono essere invocate più volte durante l'esecuzione di un programma. La distinzione tra funzioni e metodi è che le funzioni sono indipendenti, mentre i metodi sono associati a oggetti o classi.

4.2.1. Parametri e argomenti

I **parametri** e gli **argomenti** sono strumenti fondamentali per passare informazioni alle funzioni e influenzarne il comportamento.

- Parametri o parametri formali: I parametri sono definiti nella dichiarazione della funzione e rappresentano i nomi delle variabili che la funzione utilizzerà per accedere ai dati passati.
- Argomenti o parametri attuali: Gli argomenti sono i valori effettivi passati alla funzione quando viene chiamata.

Esempio in Python:

```
def add(a, b):                    ①
  return a + b

result = add(3, 4)                ②

print(result)                     ③
```

① `a` e `b` sono parametri della funzione `add`.
② `3` e `4` sono argomenti passati alla funzione `add`.
③ Il risultato della funzione `add` viene stampato.

4.2.2. Valore di ritorno

Il valore di ritorno è il risultato prodotto da una funzione, che può essere utilizzato nell'istruzione chiamante. Una funzione può restituire un valore utilizzando la parola chiave `return`.

Esempio in Java:

```
public class Main {
  public static int add(int a, int b) {                    ①
    return a + b;                                           ②
  }

  public static void main(String[] args) {
    int result = add(3, 4);                                 ③

    System.out.println(result);                             ④
  }
}
```

① Dichiarazione della funzione `add` che accetta due parametri interi.
② La funzione `add` ritorna la somma di `a` e `b`.
③ Chiamata della funzione `add` con argomenti `3` e `4`.
④ Il risultato della funzione `add` viene stampato.

4.2.3. Ricorsione

La **ricorsione** è la capacità di una funzione di chiamare se stessa, utile per risolvere problemi che possono essere suddivisi in sottoproblemi simili. Ogni chiamata ricorsiva deve avvicinarsi a una condizione di terminazione per evitare loop infiniti.

Esempio in C++ (calcolo del fattoriale):

```
#include <iostream>

int factorial(int n) {                                      ①
  if (n <= 1) return 1;                                     ②

  return n * factorial(n - 1);                              ③
}

int main() {
  int result = factorial(5);                                ④

  std::cout << result << std::endl;                         ⑤

  return 0;
}
```

① Dichiarazione della funzione `factorial`.
② Condizione di terminazione: se `n` è minore o uguale a 1, ritorna 1.

③ Chiamata ricorsiva: `factorial` chiama se stessa con `n - 1`.
④ Chiamata della funzione `factorial` con argomento `5`.
⑤ Il risultato della funzione `factorial` viene stampato.

4.2.4. Funzioni di prima classe

Le **funzioni di prima classe** sono funzioni che possono essere trattate come qualsiasi altra variabile. Possono essere assegnate a variabili, passate come argomenti e ritornate da altre funzioni.

- Esempio in Python:

```python
def greet(name):
  return f"Hello, {name}!"

say_hello = greet                                                    ①

print(say_hello("World"))                                            ②
```

① La funzione `greet` viene assegnata alla variabile `say_hello`.
② `say_hello` viene chiamata con l'argomento `"World"`.

- Esempio in C++:

```cpp
#include <iostream>
#include <functional>

void greet(const std::string& name) {
  std::cout << "Hello, " << name << "!" << std::endl;
}

int main() {
  std::function<void(const std::string&)> say_hello = greet;         ①

  say_hello("World");                                                ②

  return 0;
```

1. La funzione `greet` viene assegnata alla variabile `say_hello` utilizzando `std::function`.
2. `say_hello` viene chiamata con l'argomento `"World"`.

4.2.5. Funzioni di ordine superiore

Le **funzioni di ordine superiore** sono funzioni che accettano altre funzioni come argomenti e/o ritornano funzioni come risultati. Sono fondamentali per la programmazione funzionale.

- Esempio in Python:

```python
def add(a):
  def inner(b):
    return a + b
```

```
  return inner                                                    ①

add_five = add(5)                                                 ②

print(add_five(3))                                                ③
```

① La funzione `add` ritorna una nuova funzione `inner` che somma `a` al suo argomento `b`.
② `add(5)` ritorna una nuova funzione che somma 5 al suo argomento.
③ La funzione risultante viene chiamata con l'argomento `3`, restituendo `8`.

- Esempio in C++:

```
#include <iostream>
#include <functional>

std::function<int(int)> add(int a) {                              ①
  return [a](int b) { return a + b; };                            ②
}

int main() {
  auto add_five = add(5);                                         ③

  std::cout << add_five(3) << std::endl;                          ④

  return 0;
}
```

1. Dichiarazione della funzione `add` che ritorna un `std::function<int(int)>`.
2. `add` ritorna una funzione lambda che somma `a` al suo argomento `b`.
3. `add(5)` ritorna una nuova funzione che somma `5` al suo argomento.
4. La funzione risultante viene chiamata con l'argomento `3`, restituendo `8`.

4.2.6. Ambito e visibilità

L'ambito e la visibilità degli identificatori delle funzioni sono concetti sono simili a quelli delle variabili, ma presentano alcune differenze chiave che è importante comprendere.

4.2.6.1. Ambito

Per le funzioni distinguiamo sempre i seguenti:

- Ambito globale: Una funzione dichiarata a livello globale, cioè al di fuori di qualsiasi altra funzione o blocco di codice, ha un ambito globale. Questo significa che la funzione è visibile e può essere chiamata da qualsiasi punto del programma dopo la sua dichiarazione.

  Esempio in C++:

```cpp
#include <iostream>

void globalFunction() {                                                    ①
  std::cout << "Funzione globale" << std::endl;
}

int main() {
  globalFunction();                                                        ②

  return 0;
```

① Dichiarazione della funzione `globalFunction` a livello globale.
② Chiamata della funzione `globalFunction` all'interno di `main`.

Il comportamento è identico in Java e C. In Python, le funzioni definite a livello globale hanno ambito globale.

- Ambito locale: Una funzione dichiarata all'interno di un blocco di codice (come all'interno di una funzione o di una classe) ha un ambito locale. La funzione è visibile e può essere chiamata solo all'interno di quel blocco.

  Esempio in Python:

```python
def outerFunction():
  def localFunction():                                                     ①
    print("Funzione locale")

  localFunction()                                                          ②

outerFunction()

localFunction()                                                            ③
```

① Dichiarazione della funzione `localFunction` all'interno di `outerFunction`.
② Chiamata della funzione `localFunction` all'interno di `outerFunction`.
③ Chiamata a `localFunction` al di fuori di `outerFunction`, che genera un errore poiché `localFunction` non è visibile a questo livello.

In Java e C++, le funzioni dichiarate all'interno di un blocco (come metodi all'interno di una classe) sono accessibili solo all'interno di quel blocco, simile a Python.

In C, le funzioni locali non sono standard, ma è possibile ottenere un comportamento simile usando funzioni statiche o funzioni inline definite all'interno di un file sorgente specifico.

4.2.6.2. Visibilità

La visibilità si riferisce a dove nel codice l'identificatore di una funzione può essere utilizzato. La visibilità è strettamente legata all'ambito, ma può essere influenzata anche da altre considerazioni come la modularità e le regole di accesso.

- Visibilità Globale: Le funzioni con ambito globale sono visibili ovunque nel programma come per le variabili.

- Visibilità Locale: Le funzioni con ambito locale sono visibili solo all'interno del blocco in cui sono dichiarate. Questo è utile per creare funzioni di supporto (*helper*) o interne che non devono essere accessibili dall'esterno.

  Esempio in Python:

```
def outerFunction():
  def helperFunction():                                                    ①
    print("Funzione helper")

  helperFunction()                                                         ②

  print("Funzione esterna")

outerFunction()
```

  ① Dichiarazione della funzione `helperFunction` all'interno di `outerFunction`.
  ② Chiamata della funzione `helperFunction` all'interno di `outerFunction`.

4.2.6.3. Differenze tra funzioni con variabili e oggetti

Sebbene l'ambito e la visibilità delle funzioni condividano concetti simili con le variabili e gli oggetti, ci sono alcune differenze chiave:

- Durata di vita: Le variabili locali (automatiche) hanno una durata di vita limitata al blocco di codice in cui sono dichiarate. Quando il controllo esce dal blocco, la memoria allocata per la variabile viene liberata. Le funzioni, tuttavia, non vengono "distrutte" quando il controllo esce dal loro ambito; semplicemente non sono più visibili e chiamabili. In Python, le variabili definite all'interno di un blocco di un'istruzione composta rimangono accessibili finché sono nello stesso ambito di funzione o modulo, mentre le funzioni definite all'interno di un'altra funzione (nested functions) sono visibili solo all'interno di quella funzione.
- Allocazione dinamica: In C++, le variabili e gli oggetti possono essere allocati dinamicamente usando `new` e deallocati usando `delete`. Le funzioni non richiedono un'allocazione esplicita di memoria; la loro dichiarazione è sufficiente per renderle utilizzabili nell'ambito definito.

4.3. Spazi di nomi, moduli e file

In molti linguaggi di programmazione, la gestione dell'ambito e della visibilità delle variabili e delle funzioni può essere ulteriormente organizzata utilizzando spazio di nomi (*namespace*), moduli, header e file separati. Questa organizzazione aiuta a evitare conflitti di nome e a mantenere il codice più modulare e manutenibile.

4.3.1. Python

In Python, i moduli sono file che contengono definizioni di variabili, funzioni e classi. I moduli possono essere importati in altri moduli o script per riutilizzare il codice. Quando un modulo viene importato, gli identificatori definiti in quel modulo (come variabili, funzioni e classi) diventano accessibili attraverso il nome del modulo. Sebbene i moduli siano spesso implementati come file separati, è possibile definirli anche all'interno di un file di codice sorgente principale.

Esempio di modulo (`mymodule.py`):

```python
# mymodule.py
my_var = 10                                          ①

def my_function():
    print("Funzione del modulo")                     ②
```

① Variabile globale nel modulo.
② Funzione nel modulo.

Importazione di un modulo in un altro file sorgente (`main.py`):

```python
# main.py
import mymodule                                      ①

print(mymodule.my_var)                               ②
mymodule.my_function()                               ③
```

① Importazione del modulo `mymodule`.
② Accesso alla variabile `my_var` dal modulo `mymodule`.
③ Chiamata della funzione `my_function` dal modulo `mymodule`.

4.3.2. Java

In Java, i pacchetti (*package*) sono utilizzati per organizzare le classi in namespace separati. Ogni classe deve dichiarare il pacchetto di appartenenza.

Esempio di classe in un pacchetto (`mypackage/MyClass.java`):

```java
// mypackage/MyClass.java
package mypackage;                                   ①

public class MyClass {
    public static int myVar = 10;                    ②

    public static void myMethod() {
        System.out.println("Metodo del pacchetto");  ③
    }
}
```

① Dichiarazione del pacchetto `mypackage`.
② Variabile globale di classe.
③ Metodo della classe.

Importazione di una classe da un pacchetto in un'altra classe (`Main.java`):

```java
// Main.java
import mypackage.MyClass;                            ①

public class Main {
    public static void main(String[] args) {
```

```
        System.out.println(MyClass.myVar);                                    ②
        MyClass.myMethod();                                                   ③
    }
}
```

① Importazione della classe `MyClass` dal pacchetto `mypackage`.
② Accesso alla variabile `myVar` dalla classe `MyClass`.
③ Chiamata del metodo `myMethod` dalla classe `MyClass`.

4.3.3. C

In C, i file di intestazione (*header file*) sono utilizzati per dichiarare funzioni e variabili che possono essere utilizzate in più file sorgente, a mo' di libreria.

Esempio di file di intestazione (`mymodule.h`):

```
// mymodule.h
#ifndef MYMODULE_H
#define MYMODULE_H

extern int myVar;                                                             ①

void myFunction();                                                            ②

#endif
```

① Dichiarazione della variabile globale `myVar`.
② Dichiarazione della funzione `myFunction`.

Esempio di file sorgente (`mymodule.c`):

```
#include "mymodule.h"

int myVar = 10;                                                               ①

void myFunction() {
    printf("Funzione del modulo\n");                                          ②
}
```

① Definizione della variabile globale `myVar`.
② Definizione della funzione `myFunction`.

Utilizzo del file di intestazione in un altro file sorgente (`main.c`):

```
#include <stdio.h>
#include "mymodule.h"                                                         ①

int main() {
    printf("%d\n", myVar);                                                    ②
```

```
    myFunction();                                                    ③

    return 0;
}
```

① Inclusione del file di intestazione `mymodule.h`.
② Accesso alla variabile `myVar` dichiarata in `mymodule.h`.
③ Chiamata della funzione `myFunction` dichiarata in `mymodule.h`.

4.3.4. C++

In C++, la parola chiave `namespace` è utilizzata per organizzare le classi, le funzioni e le variabili in spazi di nomi separati, simili ai pacchetti in Java.

Esempio di dichiarazione di uno spazio di nomi (`mymodule.h`):

```
#ifndef MYMODULE_H
#define MYMODULE_H

namespace mynamespace {                                              ①
    extern int myVar;                                                ②

    void myFunction();                                               ③
}

#endif
```

① Dichiarazione dello spazio di nomi `mynamespace`.
② Dichiarazione della variabile globale `myVar` all'interno dello spazio di nomi.
③ Dichiarazione della funzione `myFunction` all'interno dello spazio di nomi.

Esempio di definizione dello spazio di nomi (`mymodule.cpp`):

```
#include "mymodule.h"
#include <iostream>

namespace mynamespace {
    int myVar = 10;                                                  ①

    void myFunction() {
        std::cout << "Funzione del namespace" << std::endl;          ②
    }
}
```

① Definizione della variabile globale `myVar` all'interno dello spazio di nomi.
② Definizione della funzione `myFunction` all'interno dello spazio di nomi.

Utilizzo dello spazio di nomi in un altro file sorgente (`main.cpp`):

```
#include "mymodule.h"
#include <iostream>

int main() {
    std::cout << mynamespace::myVar << std::endl;                    ①
    mynamespace::myFunction();                                       ②
    return 0;
}
```

① Accesso alla variabile `myVar` all'interno dello spazio di nomi `mynamespace`.
② Chiamata della funzione `myFunction` all'interno dello spazio di nomi `mynamespace`.

4.3.5. Impatti

L'uso di spazio di nomi, moduli e file di intestazione influisce sull'ambito e sulla visibilità delle variabili e delle funzioni. In generale, questi meccanismi consentono una maggiore modularità e organizzazione del codice, facilitando la gestione di grandi progetti.

- Ambito: L'ambito delle variabili e delle funzioni può essere limitato a uno spazio di nomi o a un modulo, riducendo il rischio di conflitti di nome.
- Visibilità: Le variabili e le funzioni dichiarate in namespace o moduli possono essere visibili solo all'interno di quel namespace o modulo, a meno che non vengano esplicitamente esportate.
- Durata di vita degli oggetti: La durata di vita degli oggetti non è direttamente influenzata dallo spazio di nomi o moduli, ma l'organizzazione del codice può rendere più chiaro quando e dove gli oggetti vengono creati e distrutti.

5. Il modello dati dei linguaggi di programmazione

Un **modello dati** è una rappresentazione formale dei tipi di dati del linguaggio e delle operazioni che possono essere eseguite su di essi. Esso definisce le strutture fondamentali attraverso le quali i dati vengono organizzati, memorizzati, manipolati e interagiscono all'interno del programma.

Le componenti il modello dati sono:

1. Tipi di dati:

   - Tipi primitivi: Questi sono i tipi di dati fondamentali che il linguaggio supporta nativamente, come numeri interi, numeri in virgola mobile, caratteri e booleani.

   - Tipi compositi: Questi sono tipi di dati costruiti combinando tipi primitivi. Esempi comuni includono array, liste, tuple, set e dizionari.

   - Tipi di dati definiti dall'utente: Questi sono tipi di dati che possono essere definiti dagli utenti del linguaggio, come le `struct` in C oppure le classi in Python o C++, che permettono di creare tipi di dati personalizzati.

2. Operazioni:

   - Operazioni aritmetiche: Operazioni che possono essere eseguite sui tipi di dati, come addizione, sottrazione, moltiplicazione e divisione per i numeri.

   - Operazioni logiche: Operazioni che coinvolgono valori booleani, come AND, OR e NOT.

   - Operazioni di sequenza: Operazioni che si possono eseguire su sequenze di dati, come l'indicizzazione, la *slicing* e l'iterazione.

   - Altre operazioni ad hoc per il tipo di dato.

3. Regole di comportamento:

   - Mutabilità: Determina se un oggetto può essere modificato dopo la sua creazione. Oggetti mutabili, come liste e dizionari in Python, possono essere cambiati. Oggetti immutabili, come tuple e stringhe, non possono essere modificati dopo la loro creazione.

   - Copia e clonazione: Regole che determinano come i dati vengono copiati. Per esempio, in Python, la copia di una lista crea una nuova lista con gli stessi elementi, mentre la copia di un intero crea solo un riferimento allo stesso valore.

5.1. Linguaggi procedurali

Nei linguaggi di programmazione procedurali, il modello dati è incentrato su tipi di dati semplici e compositi che supportano lo stile di programmazione orientato alle funzioni e procedure. Alcune caratteristiche tipiche includono:

- Tipi primitivi: Numeri interi, numeri a virgola mobile, caratteri e booleani.
- Strutture composite: Array, strutture (`struct`) e unioni (`union`). Gli array permettono di gestire collezioni di elementi dello stesso tipo, mentre le strutture permettono di combinare vari tipi di dati sotto un unico nome. Le unioni consentono di memorizzare diversi tipi di dati nello stesso spazio di memoria, ma solo uno di essi può essere attivo alla volta.
- Operazioni basate su funzioni: Le operazioni sui dati vengono eseguite attraverso funzioni che manipolano i valori passati come argomenti.

Esempio in C:

```c
#include <stdio.h>
#include <string.h>

#define MAX_DATI 100

union Valore {                                                    ①
  int intero;
  float decimale;
  char carattere;
};

struct Dato {                                                     ②
  char tipo;
  // 'i' per int, 'f' per float, 'c' per char
  union Valore valore;
};

void stampa_dato(struct Dato d) {                                 ③
  switch (d.tipo) {
    case 'i':
      printf("Intero: %d\n", d.valore.intero);
      break;

    case 'f':
      printf("Float: %f\n", d.valore.decimale);
      break;

    case 'c':
      printf("Carattere: %c\n", d.valore.carattere);
      break;

    default:
      printf("Tipo sconosciuto\n");
```

```
      break;
  }
}

int confronta_dato(struct Dato d1, struct Dato d2) {
  if (d1.tipo != d2.tipo) return 0;

  switch (d1.tipo) {
    case 'i': return d1.valore.intero == d2.valore.intero;

    case 'f': return d1.valore.decimale == d2.valore.decimale;

    case 'c': return d1.valore.carattere == d2.valore.carattere;

    default: return 0;
  }
}

void inserisci_dato(struct Dato dati[], int *count, struct Dato nuovo_dato) {          ④
  if (*count < MAX_DATI) {
    dati[*count] = nuovo_dato;

    (*count)++;

  } else {
    printf("Array pieno, impossibile inserire nuovo dato.\n");
  }
}

void cancella_dato(struct Dato dati[], int *count, struct Dato dato_da_cancellare) {   ⑤
  for (int i = 0; i < *count; i++) {
    if (confronta_dato(dati[i], dato_da_cancellare)) {
      for (int j = i; j < *count - 1; j++) {
        dati[j] = dati[j + 1];
      }

      (*count)--;

      i--;
    }
  }
}

int main() {
  struct Dato dati[MAX_DATI];                                                          ⑥
  int count = 0;

  struct Dato dato1 = {'i', .valore.intero = 42};
  struct Dato dato2 = {'f', .valore.decimale = 3.14};
```

```
  struct Dato dato3 = {'c', .valore.carattere = 'A'};

  inserisci_dato(dati, &count, dato1);                                    ⑦
  inserisci_dato(dati, &count, dato2);
  inserisci_dato(dati, &count, dato3);

  for (int i = 0; i < count; i++) {                                       ⑧
    stampa_dato(dati[i]);
  }

  cancella_dato(dati, &count, dato1);                                     ⑨

  printf("Dopo cancellazione:\n");

  for (int i = 0; i < count; i++) {
    stampa_dato(dati[i]);
  }

  return 0;
}
```

① Definizione di una `union`.
② Definizione di una `struct` che include la `union`.
③ Funzione per stampare i valori in base al tipo.
④ Funzione per inserire un nuovo dato alla fine dell'array.
⑤ Funzione per cancellare tutte le occorrenze di un dato dall'array.
⑥ Definizione di un array di `struct` Dato.
⑦ Inserimento di dati nell'array.
⑧ Stampa dei dati nell'array.
⑨ Cancellazione di un dato specifico e ristampa dell'array.

L'esempio mostra come nel modello dati del linguaggio C possono essere definiti dei tipi compositi (`Dato`, `Valore`) e delle operazioni su quelli (`stampa_dato`, `confronta_dato`, `inserisci_dato`, `cancella_dato`). Il codice, pur realizzante una semplice libreria, appare *slegato*, cioè con funzioni che si applicano a tipi di dati specifici solo dall'interpretazione degli identificatori della funzione stessa e dei suoi parametri, cioè senza un legame esplicito e non ambiguo, tra tipo e funzione.

5.2. Linguaggi orientati agli oggetti

La programmazione orientata agli oggetti è un paradigma che utilizza **oggetti** per rappresentare concetti ed entità del mondo reale o astratto. Questo approccio si basa su un processo mentale fondamentale per risolvere problemi complessi: la decomposizione. Un problema complesso è più facilmente risolvibile se diviso in parti più piccole, ciascuna delle quali possiede uno stato e la possibilità di interagire con le altre parti. Questa divisione può essere effettuata per gradi, come se si osservasse sempre più da vicino il problema, effettivamente continunandone la specificazione, fino a raggiungere un livello sufficientemente di dettaglio da poter essere realizzato come istruzioni, codificate in costrutti permessi dalla sintassi del linguaggio, dell'oggetto.

5.2.1. Oggetti

Lo stato di un oggetto è definito dai suoi attributi, i cui valori possono essere altri oggetti già disponibili, sia definiti dall'utente che dal linguaggio. L'interazione tra diversi oggetti avviene attraverso i metodi, che sono funzioni associate agli oggetti che possono modificare lo stato dell'oggetto o invocare metodi su altri oggetti.

I membri di un oggetto (attributi e metodi) possono avere diverse limitazioni di accesso, definite dal concetto di visibilità:

- Pubblica: Gli attributi e i metodi pubblici sono accessibili da qualsiasi parte del programma. Questa visibilità permette a qualsiasi altro oggetto o funzione di interagire con questi membri.
- Privata: Gli attributi e i metodi privati sono accessibili solo da altri membri dell'oggetto e rispondono alla esigenza di separare il codice di interfaccia da quello utile al funzionamento interno.
- Protetta: Gli attributi e i metodi protetti sono accessibili da tutti i membri del medesimo oggetto ma, a differenza dei privati, anche da quelli degli oggetti derivati. Questo fornisce un livello intermedio di accesso, utile per la gestione dell'ereditarietà.

L'**incapsulamento** è il principio su cui si basa la gestione della visibilità e guida la separazione del codice realizzante le specificità di un oggetto, da come è fruito dagli altri oggetti. Questo protegge l'integrità del suo stato e ne facilita la manutenzione del codice stesso, permettendo modifiche di implementazione, senza impatti sul codice esterno fintantoché non si cambiano i membri pubblici. Inoltre, se ben sfruttata nella progettazione, rende il codice più comprensibile e riduce la superficie d'attacco.

5.2.2. Classi

Un oggetto può essere generato da una struttura statica che ne definisce tutte le caratteristiche, la **classe**, oppure può essere creato a partire da un altro oggetto esistente, noto come **prototipo**.

Nella programmazione ad oggetti basata su classi, ogni oggetto è un'istanza *vivente* di una classe predefinita, che ne rappresenta il progetto o l'archetipo. La classe definisce i membri e la visibilità, quindi, in definitiva tutte le proprietà comuni agli oggetti dello stesso tipo o matrice. Gli oggetti vengono creati chiamando un metodo speciale della classe, noto come costruttore e, all'atto della loro vita, un secondo metodo, il distruttore, che si occupa di effettuare le azioni di terminazione.

La classe può inoltre definire metodi e attributi particolari, che possono essere ereditati da altre classi, cioè possono essere utilizzati da quest'ultime al pari dei propri membri. In tal modo, il linguaggio permette la costruzioni di gerarchie di classi che modellano relazioni di specializzazione, dalla più generale alla più particolare.

Ciò, oltre ad essere uno strumento di progettazione utile di per sé, facilita il riuso del codice per mezzo dell'estensione, al posto della modifica, di funzionalità. La classe che eredita da un'altra classe si definisce *derivata* dalla classe che, a sua volta, è detta *base*.

5.2.3. Prototipi

Alternativamente, alcuni linguaggi usano il concetto di prototipo, in cui gli oggetti sono le entità principali e non esiste una matrice separata come la classe. In questo paradigma, ogni oggetto può servire da prototipo per altri e ciò significa che, invece di creare nuove istanze di una classe, si creano nuovi oggetti clonando o estendendo quelli esistenti. È possibile aggiungere o modificare proprietà e metodi di un oggetto prototipo e, in tal caso, queste modifiche si propagheranno in tutti gli oggetti che derivano da esso.

Il paradigma basato su prototipi offre maggiore flessibilità e dinamismo rispetto a quello basato su classi, poiché la struttura degli oggetti può essere modificata in modo dinamico. D'altronde, questo approccio può anche introdurre complessità e rendere più difficile la gestione delle gerarchie di oggetti e la comprensione del codice, poiché non esistono strutture fisse come le classi.

5.2.4. Esempi di gerarchie di classi e prototipi

Vediamo le differenze tra classi e prototipi, riprendendo l'esempio in Java nella versione semplificata (senza astrazione):

```java
class Animale {
    String nome;

    Animale(String nome) {
        this.nome = nome;
    }

    void faiVerso() {
        System.out.println("L'animale fa un verso");
    }
}

class Cane extends Animale {

    Cane(String nome) {
        super(nome);
    }

    @Override
    void faiVerso() {
        System.out.println("Il cane abbaia");
    }
}

public class Main {
    public static void main(String[] args) {
        Animale mioCane = new Cane("Fido");

        mioCane.faiVerso();
    }
}
```

Implementiamo il medesimo programma in Javascript[1], linguaggio che usa il concetto di prototipo:

[1] In JavaScript, le classi come sintassi sono state introdotte in ECMAScript 6 (ES6), per semplificare la creazione di oggetti e la gestione dell'ereditarietà prototipale. Tuttavia, è importante capire che sotto il cofano, JavaScript non utilizza classi nel senso tradizionale come in linguaggi come Java o C++ e non esiste un meccanismo nativo per creare classi astratte, anche se è possibile simulare il comportamento delle classi astratte utilizzando varie tecniche. Una comune è quella di lanciare un'eccezione se un metodo funzionalmente astratto non viene sovrascritto nella classe derivata.

```
let Animale = {                                                    ①
    nome: "Generic",

    init: function(nome) {
        this.nome = nome;
    },

    faiVerso: function() {
        console.log("L'animale fa un verso");
    }
};

let Cane = Object.create(Animale);                                 ②

Cane.faiVerso = function() {                                       ③
    console.log("Il cane abbaia");
};

let mioCane = Object.create(Cane);                                 ④
mioCane.init("Fido");                                              ⑤

mioCane.faiVerso();                                                ⑥
```

① Definizione dell'oggetto prototipo `Animale`.
② Creazione di un nuovo oggetto basato sul prototipo `Animale`.
③ Viene creato un nuovo oggetto `Cane` basato sul prototipo `Animale`, usando `Object.create(Animale)`. Questo permette a `Cane` di ereditare proprietà e metodi da `Animale`. Il metodo `faiVerso` viene sovrascritto nell'oggetto `Cane` per specificare il comportamento da cane.
④ Un nuovo oggetto `mioCane` viene creato basandosi sul prototipo `Cane` usando `Object.create(Cane)`.
⑤ Il metodo `init` viene chiamato per inizializzare il nome dell'oggetto `mioCane`.
⑥ Quando viene chiamato `mioCane.faiVerso()`, il metodo sovrascritto nell'oggetto `Cane` viene eseguito, mostrando `Il cane abbaia`.

5.2.5. Ereditarietà

Come abbiamo visto, l'ereditarietà è un meccanismo che permette a una classe di ereditare membri da un'altra classe. Essa si può presentare singola o **multipla**, ove la prima consente a una classe derivata di estendere solo una classe base. Questo è il modello di ereditarietà più comune e supportato da molti linguaggi di programmazione orientati agli oggetti, come Java e C#.

L'ereditarietà multipla è tale da permettere a una classe di ricevere attributi e metodi contemporaneamente da più classi base. Questo meccanismo risponde all'esigenza di specializzare più concetti allo stesso tempo. Va sottolineato che è uno strumento potente prono, però, ad abusi, perché può introdurre complessità nella gestione delle gerarchie di classi e causare conflitti quando lo stesso metodo è ereditato da più classi, situazione nota come *problema del diamante*. Pertanto, alcuni linguaggi ne limitano l'applicazione, come Java che consente solo l'ereditarietà multipla di interfacce, ma non di classi. Altri, come Go, non supportano l'ereditarietà per scelta di progettazione. Go enfatizza la composizione rispetto all'ereditarietà per promuovere uno stile di programmazione più essenziale e flessibile. La composizione consente di costruire comportamenti complessi

aggregando oggetti più semplici, evitando le complicazioni delle gerarchie di classi multilivello. Il C++, invece, supporta completamente l'ereditarietà multipla.

5.2.6. Interfacce e classi astratte

Le **interfacce** e le **classi astratte** sono due concetti fondamentali nella programmazione orientata agli oggetti, che consentono di definire contratti che le classi concrete devono rispettare.

Un'interfaccia è un contratto che specifica un insieme di metodi che una classe deve implementare, senza fornire l'implementazione effettiva di questi metodi. Sono utilizzate per definire comportamenti comuni che possono essere condivisi da classi diverse, indipendentemente dalla loro posizione nella gerarchia delle classi. Le classi che implementano un'interfaccia devono fornire una definizione concreta per tutti i metodi dichiarati nell'interfaccia. In Java, ad esempio, le interfacce sono definite con la parola chiave `interface`.

Una classe astratta è una classe che non può essere istanziata direttamente. Può contenere sia metodi astratti (senza codice al loro interno, che devono essere implementati dalle classi derivate) sia metodi concreti (con codice allinterno, che possono essere utilizzati dalle classi derivate). Le classi astratte sono utilizzate per fornire una base comune con alcune implementazioni di default e lasciare ad altre classi il compito di completare l'implementazione. In Java, le classi astratte sono definite con la parola chiave `abstract`.

Esempio di interfaccia, classa astratta e ereditarietà multipla in Java:

```
interface Domesticazione {                                                    ①
  void assegnaAddomesticato(boolean addomesticato);                           ②
  boolean ottieniAddomesticato();                                             ③
}

abstract class Animale {                                                      ④
  String nome;

  Animale(String nome) {
    this.nome = nome;
  }

  abstract String faiVerso();                                                 ⑤

  String descrizione() {                                                      ⑥
    return "L'animale si chiama " + nome;
  }
}

class Cane extends Animale implements Domesticazione {                       ⑦
  private boolean addomesticato;                                              ⑧

  Cane(String nome) {
    super(nome);
  }

  @Override
  String faiVerso() {                                                         ⑨
```

```
    return "Il cane abbaia";
  }

  @Override
  public void assegnaAddomesticato(boolean addomesticato) {                    (10)
    this.addomesticato = addomesticato;
  }

  @Override
  public boolean ottieniAddomesticato() {                                      (11)
    return addomesticato;
  }
}

class Coccodrillo extends Animale {                                            (12)

  Coccodrillo(String nome) {
    super(nome);
  }

  @Override
  String faiVerso() {
    return "";
  }
}

public class Main {
  public static void main(String[] args) {
    Cane mioCane = new Cane("Fido");                                           (13)

    System.out.println(mioCane.descrizione());
    System.out.println(mioCane.faiVerso());

    mioCane.assegnaAddomesticato(true);
    System.out.println("Cane addomesticato: " +
                       mioCane.ottieniAddomesticato());

    Coccodrillo mioCoccodrillo = new Coccodrillo("Crocky");                    (14)

    System.out.println(mioCoccodrillo.descrizione());
    System.out.println(mioCoccodrillo.faiVerso());
  }
}
```

① Interfaccia che definisce una proprietà che gli animali possono possedere, la domesticazione. Da notare che la domesticazione è una proprietà *complementare* alle altre caratterizzanti l'animale, addirittura non aprioristica.
② Metodo per impostare lo stato di addomesticamento dell'animale.
③ Metodo per verificare se è addomesticato.
④ Classe astratta che ha l'implementazione di una caratteristica condivisa dalle classi derivate, `descrizione()`,

e un metodo astratto per una seconda, `faiVerso()`, che, deve essere sempre presente negli oggetti di tipo base animale, ma non ne è comume l'implementazione.

⑤ Metodo astratto.
⑥ Metodo concreto.
⑦ Il cane è un animale che può essere addomesticato, quindi la classe `Cane` deriva `Animale` (cioè deve implementare necessariamente `faiVerso()`) e implementa `Domesticazione` (cioè deve implementare `assegnaAddomesticato()` e `ottieniAddomesticato()`). `descrizione()` viene ereditato colla implementazione di `Animale`.
⑧ Variabile utile a registrare se il cane è stato addomesticato.
⑨ `Cane` implementa `faiVerso()` di `Animale`.
⑩ `Cane` implementa `assegnaAddomesticato()` di `Domesticazione`.
⑪ `Cane` implementa `ottieniAddomesticato()` di `Domesticazione`.
⑫ Il coccodrillo non è addomesticabile, quindi, `Coccodrillo` non implementa l'interfaccia `Domesticazione`, ma è comunque un animale quindi deriva `Animale` e ne implementa l'unico metodo astratto `faiVerso()`. Non essendo addomesticabile, non ha neanche l'attributo `addomesticato`.
⑬ Creazione dell'oggetto `Cane`.
⑭ Creazione dell'oggetto `Coccodrillo`.

Le interfacce e le classi astratte sono strumenti potenti per promuovere la riusabilità del codice e l'estensibilità dei sistemi software, poiché permettono di definire contratti chiari e di implementare diverse versioni di una funzionalità senza modificare il codice preesistente.

5.2.7. Polimorfismo

Il **polimorfismo** è un concetto chiave della programmazione orientata agli oggetti che permette a oggetti di classi diverse di essere trattati come oggetti di una classe comune. È uno strumento complementare all'ereditarietà, nelle mani del programmatore, utile a modellare comportamenti comuni per oggetti di tipi diversi, permettendo al codice di interagire con questi oggetti senza conoscere esattamente il loro tipo specifico. In termini pratici, il polimorfismo permette di chiamare metodi su oggetti di tipi diversi e ottenere comportamenti specifici a seconda del tipo di oggetto su cui viene chiamato il medesimo metodo.

Il concetto di polimorfismo è strettamente legato all'idea di contratto tra oggetti. Questo contratto è definito dalle interfacce o dalle classi base e specifica quali metodi devono essere implementati dalle classi derivate. Quando un oggetto di una classe derivata è trattato come un oggetto della classe base o di un'interfaccia, si garantisce che esso rispetti il contratto definito dalla classe base o dall'interfaccia.

Esistono due tipi principali di polimorfismo:

- Polimorfismo statico: Conosciuto soprattutto come **overloading**, si verifica quando più metodi nella stessa classe hanno lo stesso nome ma firme diverse (diverso numero o tipo di parametri). Il compilatore decide quale metodo chiamare in base alla firma del metodo.

Esempio in Java che supporta l'overloading:

```
class Esempio {
  void stampa(int a) {
    System.out.println("Intero: " + a);
  }

  void stampa(String a) {
    System.out.println("Stringa: " + a);
```

```
  }
}

public class Main {
  public static void main(String[] args) {
    Esempio es = new Esempio();

    es.stampa(5);                                                    ①
    es.stampa("ciao");                                               ②
  }
}
```

① Chiama il metodo `stampa(int a)`.
② Chiama il metodo `stampa(String a)`.

- Polimorfismo dinamico: Noto come **overriding**, si verifica quando una classe derivata fornisce una specifica implementazione di un metodo già definito nella sua classe base. L'implementazione da chiamare è determinata a runtime, cioè a tempo di esecuzione e non compilazione, in base al tipo dell'oggetto.

Esempio in Java riprendendo l'esempio con gli animali:

```
class Animale {
  void faiVerso() {
    System.out.println("L'animale fa un verso");
  }
}

class Cane extends Animale {
  @Override
  void faiVerso() {
    System.out.println("Il cane abbaia");
  }
}

public class Main {
  public static void main(String[] args) {
    Animale mioAnimale = new Cane();                                 ①

    mioAnimale.faiVerso();                                           ②
  }
}
```

① L'oggetto `mioAnimale` è dichiarato come tipo `Animale` ma istanziato come `Cane`. Questo è un esempio di polimorfismo.
② Il metodo `faiVerso()` viene chiamato sull'oggetto `mioAnimale`, ma viene eseguita la versione del metodo `faiVerso()` definita nella classe `Cane`, grazie al polimorfismo.

Il polimorfismo è strettamente legato all'ereditarietà, poiché l'ereditarietà è spesso il meccanismo che permette al polimorfismo di funzionare. Quando una classe derivata estende una classe base e sovrascrive i suoi metodi, permette agli oggetti della classe derivata di essere trattati come oggetti della classe base ma di comportarsi in modo specifico alla classe derivata.

I linguaggi di programmazione hanno delle differenze in relazione al supporto del polimorfismo:

- Java: Supporta sia l'overloading che l'overriding.
- C++: Supporta sia l'overloading che l'overriding. Fornisce meccanismi per specificare il tipo di legame (statico o dinamico) usando parole chiave come `virtual`.
- Python: Supporta l'overriding, ma non l'overloading nello stesso senso di Java o C++. Python permette la definizione di metodi con argomenti predefiniti o argomenti variabili per ottenere un effetto simile all'overloading.

Esempio in Java da confrontare con quello seguente in Python:

```
class Animale {
  void faiVerso() {                                                    ①
    System.out.println("L'animale fa un verso");
  }
}

class Cane extends Animale {
  @Override
  void faiVerso() {                                                    ②
    System.out.println("Il cane abbaia");
  }

  void faiVerso(String suono) {                                        ③
    System.out.println("Il cane fa: " + suono);
  }
}

public class Main {
  public static void main(String[] args) {
    Animale mioAnimale = new Cane();                                   ④
    mioAnimale.faiVerso();                                             ⑤

    Cane mioCane = new Cane();
    mioCane.faiVerso("bau");                                           ⑥
  }
}
```

① Metodo `faiVerso()` definito nella classe base `Animale`.
② Overriding del metodo `faiVerso()` nella classe derivata `Cane`.
③ Overloading del metodo `faiVerso()` nella classe derivata `Cane`.
④ Dichiarazione di un oggetto di tipo `Animale`, ma istanziato come `Cane`.
⑤ Chiamata al metodo `faiVerso()`, che esegue la versione del metodo nella classe `Cane` grazie al polimorfismo.
⑥ Chiamata al metodo `faiVerso(String suono)`, che dimostra l'overloading del metodo nella classe `Cane`.

E in Python diventa:

```
class Animale:
  def fai_verso(self):                                                    ①
    print("L'animale fa un verso")

class Cane(Animale):
  def fai_verso(self):                                                    ②
    print("Il cane abbaia")

  def fai_verso_con_suono(self, suono):                                   ③
    print(f"Il cane fa: {suono}")

mio_animale = Cane()                                                      ④
mio_animale.fai_verso()                                                   ⑤

mio_cane = Cane()
mio_cane.fai_verso_con_suono("bau")                                       ⑥
```

① Metodo `fai_verso()` definito nella classe base `Animale`.
② Overriding del metodo `fai_verso()` nella classe derivata `Cane`.
③ Definizione di un metodo aggiuntivo `fai_verso_con_suono` nella classe derivata `Cane` (Python non supporta l'overloading nello stesso senso di Java).
④ Dichiarazione e istanziazione di un oggetto `mio_animale` come `Cane`.
⑤ Chiamata al metodo `fai_verso()`, che esegue la versione del metodo nella classe `Cane` grazie al polimorfismo.
⑥ Chiamata al metodo `fai_verso_con_suono(suono)`, che dimostra una forma di polimorfismo simile all'overloading in Python.

L'overriding è possibile grazie al **dynamic dispatch**, un meccanismo che consente di selezionare a runtime il metodo corretto da invocare in base al tipo effettivo dell'oggetto. Lo **static disptach**, al contrario, avviene al tempo di compilazione.

Ma il dispatch, cioè l'individuazione del metodo da eseguire, può essere singolo (**single dispatch**), così come presente nella maggior parte dei linguaggi orientati agli oggetti come Java e C++, e dove la scelta del metodo dipende solo dal tipo dell'oggetto sul quale il metodo stesso viene chiamato. Questo tipo di dispatch è sufficiente per supportare il polimorfismo detto di *sottotipo*, dove le classi derivate possono sovrascrivere i metodi della classe base e il metodo corretto viene selezionato a runtime in base al tipo effettivo dell'oggetto.

Il **multiple dispatch**, invece, estende ulteriormente le capacità del polimorfismo permettendo la selezione del metodo da invocare basandosi sui tipi runtime di più di un argomento. Questo è particolarmente utile in scenari dove il comportamento dipende da combinazioni di tipi di oggetti, e non solo dal tipo dell'oggetto su cui il metodo è chiamato. Linguaggi come Julia e CLOS (Common Lisp Object System)supportano nativamente il multiple dispatch, mentre linguaggi come Java e C++ non lo supportano direttamente ma possono emularlo attraverso pattern come il *visitor*.

5.2.8. Altri concetti

Dopo aver compreso i concetti fondamentali della programmazione orientata agli oggetti (OOP), come oggetti, classi, prototipi, ereditarietà e polimorfismo, è importante esplorare altri aspetti avanzati che contribuiscono alla potenza e alla flessibilità di questo paradigma.

5.2.8.1. Mixin e trait

I **mixin** e i **trait** sono concetti che permettono di aggiungere funzionalità a una classe senza utilizzare l'ereditarietà classica.

I mixin sono classi che offrono metodi che possono essere utilizzati da altre classi senza essere una classe base di queste ultime. Permettono di combinare comportamenti comuni tra diverse classi.

Esempio:

```python
class MixinA:
    def metodo_a(self):
        print("Metodo A")

class MixinB:
    def metodo_b(self):
        print("Metodo B")

class ClasseConMixin(MixinA, MixinB):
    pass

obj = ClasseConMixin()
obj.metodo_a()
obj.metodo_b()
```

I trait sono simili ai mixin e permettono di definire metodi che possono essere riutilizzati in diverse classi. Sono supportati nativamente in linguaggi come Scala e Rust.

```scala
trait TraitA {
    def metodoA(): Unit = println("Metodo A")
}

trait TraitB {
    def metodoB(): Unit = println("Metodo B")
}

class ClasseConTrait extends TraitA with TraitB

val obj = new ClasseConTrait()
obj.metodoA()
obj.metodoB()
```

5.2.8.2. Duck Typing

Il **duck typing** è un concetto che si applica principalmente nei linguaggi dinamici, dove l'importanza è data al comportamento degli oggetti piuttosto che alla loro appartenenza a una specifica classe. Se un oggetto implementa i metodi richiesti da una certa operazione, allora può essere utilizzato per quella operazione, indipendentemente dal suo tipo.

Esempio:

```
class Anatra:
    def quack(self):
        print("Quack!")

class Persona:
    def quack(self):
        print("Sono una persona che imita un'anatra")

def fai_quack(oggetto):
    oggetto.quack()

anatra = Anatra()
persona = Persona()

fai_quack(anatra)
fai_quack(persona)
```

6. Altri concetti semantici dei linguaggi di programmazione

Dopo aver esplorato variabili, funzioni e oggetti, ci sono altri concetti semantici essenziali nei linguaggi di programmazione che completano la comprensione del comportamento dei programmi. Questi concetti includono la concorrenza, l'input/output (I/O), le annotazioni e i metadati, e le macro e la metaprogrammazione.

6.1. Concorrenza

La concorrenza è la capacità di un programma di eseguire più sequenze di istruzioni in parallelo, migliorando le prestazioni e la reattività. La concorrenza è particolarmente utile in applicazioni che richiedono l'elaborazione simultanea di compiti indipendenti, come server web, sistemi di gestione di basi di dati e applicazioni interattive.

Un concetto fondamentale della concorrenza è il **thread**, che rappresenta la più piccola unità di elaborazione eseguibile in modo indipendente. I thread permettono l'esecuzione parallela di codice all'interno di un programma, ma introducono la necessità di gestire l'accesso concorrente alle risorse condivise.

La **sincronizzazione** è essenziale per evitare condizioni di gara, che si verificano quando il risultato dell'esecuzione dipende dalla sequenza temporale in cui i thread accedono alle risorse. Meccanismi come i **lock** e i **mutex** garantiscono che solo un thread alla volta possa accedere a una risorsa condivisa, prevenendo conflitti e garantendo la consistenza dei dati.

In linguaggi moderni, come Python e JavaScript, la gestione delle operazioni asincrone è facilitata da costrutti come **async/await**. Questi costrutti migliorano l'efficienza e la reattività delle applicazioni, permettendo di eseguire operazioni di I/O senza bloccare il thread principale.

6.2. Input/Output (I/O)

L'input/output (I/O) gestisce la comunicazione tra un programma e il suo ambiente esterno. Il **File I/O** permette la lettura e la scrittura su file, consentendo di memorizzare e recuperare dati persistenti. In molti linguaggi, come C e Python, le operazioni di file I/O sono supportate da funzioni o metodi che aprono, leggono, scrivono e chiudono file.

Il **Network I/O** facilita la comunicazione tra sistemi diversi attraverso reti, consentendo di inviare e ricevere dati tra computer. Linguaggi come Java e Python offrono librerie per la gestione delle connessioni di rete, il trasferimento di dati e la comunicazione tra client e server.

Lo **Standard I/O** comprende l'interazione con l'utente tramite input da tastiera e output su schermo. In C, funzioni come `scanf` e `printf` gestiscono lo standard I/O, mentre in Python si utilizzano `input` e `print`.

6.3. Annotazioni e Metadati

Le annotazioni e i metadati forniscono informazioni aggiuntive al compilatore o al runtime, influenzando il comportamento del programma o fornendo dettagli utili per la documentazione e l'analisi del codice.

Le **annotazioni** sono utilizzate per specificare comportamenti speciali o configurazioni. In Java, le annotazioni come `@Deprecated` indicano che un metodo è obsoleto, `@Override` segnala che un metodo sovrascrive un metodo della superclasse, e `@Entity` e `@Table` in JPA (Jakarta Persistence) definiscono la relazione tra entità e tabelle nel contesto di un database. In Python, le annotazioni dei tipi (type hint) indicano i tipi delle variabili, dei parametri di funzione e dei valori di ritorno, migliorando la leggibilità e facilitando il type checking automatico.

Le **docstring** in Python sono commenti strutturati che documentano il codice. Utilizzate per descrivere moduli, classi, metodi e funzioni, le docstring rendono il codice più leggibile e comprensibile e possono essere utilizzate per generare documentazione automatica.

6.4. Macro e Metaprogrammazione

Le macro e la metaprogrammazione permettono di scrivere codice che manipola altre porzioni di codice, migliorando la flessibilità e il riutilizzo.

Le **macro** sono sequenze di istruzioni predefinite che possono essere inserite nel codice durante la fase di precompilazione. In C, le macro sono utilizzate con il preprocessore per definire costanti, funzioni inline e codice condizionale. Le macro permettono di evitare la duplicazione di codice, ma possono anche introdurre complessità e difficoltà di debug.

La **metaprogrammazione** consiste nello scrivere codice che genera o modifica altre parti del codice a runtime o a compile-time. In Python, la metaprogrammazione include l'uso di decoratori, che sono funzioni che modificano il comportamento di altre funzioni, e metaclassi, che permettono di controllare la creazione e il comportamento delle classi. L'introspezione, che consente di esaminare gli oggetti durante l'esecuzione del programma, è un'altra potente tecnica di metaprogrammazione.

Parte II.

Seconda parte: Le basi di Python

7. Introduzione a Python

Python è un linguaggio di programmazione multiparadigma, cioè abilita o supporta più paradigmi di programmazione, e multipiattaforma, potendo essere installato e utilizzato su gran parte dei sistemi operativi e hardware.

La storia di Python inizia colla pubblicazione del codice sorgente, da parte del suo creatore Guido van Rossum, nel 1991, nella versione 0.9.0. Circa tre anni dopo, nel 1994, raggiungerà la prima versione definitiva, quindi nove anni dopo il C++ e un anno prima di PHP e due rispetto a Java.

Python offre una combinazione unica di eleganza, semplicità, praticità e versatilità. Questa eleganza e semplicità derivano dal fatto che è stato progettato per essere molto simile al linguaggio naturale inglese, rendendo il codice leggibile e comprensibile. La sintassi di Python è pulita e minimalista, evitando simboli superflui come parentesi graffe e punti e virgola, e utilizzando indentazioni per definire blocchi di codice, il che forza una struttura coerente e essenziale. La semantica del linguaggio è intuitiva e coerente, il che riduce la curva di apprendimento e minimizza gli errori, anche per programmatori non professionali.

Python è gestito dalla Python Software Foundation (PSF), un'organizzazione no-profit che si occupa dello sviluppo e della promozione del linguaggio. Il sito ufficiale di Python, python.org, è la risorsa principale dove è possibile trovare la documentazione ufficiale, scaricare il linguaggio, e accedere a tutorial, guide e altre risorse utili.

Il processo di aggiornamento di Python è trasparente e comunitario. Le proposte di miglioramento del linguaggio vengono discusse attraverso le *Python Enhancement Proposals* (proposte di aggiornamento di Python, PEP), documenti di design che forniscono informazioni alle comunità di Python su nuove funzionalità proposte, miglioramenti al linguaggio e altre questioni correlate. Le PEP vengono valutate e accettate da un comitato di sviluppo centrale.

Python è distribuito sotto la Python Software Foundation License, una licenza open-source che consente l'uso, la modifica e la distribuzione del linguaggio senza costi. Questa licenza garantisce che Python rimanga libero e accessibile a tutti, permettendo l'uso commerciale e non commerciale. Per quanto riguarda i documenti e la documentazione, essi sono generalmente distribuiti sotto la licenza Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA), che consente di condividere e adattare il materiale, purché venga attribuito correttamente, non sia utilizzato per scopi commerciali e sia distribuito con la stessa licenza.

Tecnicamente, Python è un linguaggio interpretato e dinamico. Essere interpretato significa che il codice sorgente di Python viene eseguito direttamente dall'interprete, senza la necessità di una fase di compilazione separata. Questo offre vantaggi come la facilità di esecuzione del codice e il supporto per il debugging interattivo, ma può comportare una velocità di esecuzione inferiore rispetto ai linguaggi compilati.

Essere dinamico significa che molti aspetti del linguaggio, come i tipi delle variabili, vengono determinati a runtime piuttosto che a compile-time. Questo consente una maggiore flessibilità e facilita lo sviluppo rapido, poiché non è necessario dichiarare esplicitamente i tipi di variabili. Tuttavia, questo approccio può anche portare a errori di tipo che vengono rilevati solo durante l'esecuzione del programma.

Diventerai rapidamente produttivo con Python grazie alla sua coerenza e regolarità, alla sua ricca libreria standard e ai numerosi pacchetti e strumenti di terze parti prontamente disponibili. Python è facile da imparare, quindi è molto adatto se sei nuovo alla programmazione, ma è anche potente abbastanza per i più

sofisticati esperti. Questa semplicità ha attratto una comunità ampia e attiva che ha contribuito sia alle librerie di programmi incluse nell'implementazione ufficiale, che a molte librerie scaricabili liberamente, ampliando ulteriormente l'ecosistema di Python.

7.1. Perché Python è un linguaggio di alto livello?

Python è considerato un linguaggio di programmazione di alto livello, cioè utilizza un livello di astrazione elevato rispetto alla complessità dell'ambiente in cui i suoi programmi sono eseguiti. Il programmatore ha a disposizione una sintassi che è più intuitiva rispetto ad altri linguaggi come Java, C++, PHP tradizionalmente anch'essi definiti di alto livello.

Infatti, consente ai programmatori di scrivere codice in modo più concettuale e indipendente dalle caratteristiche degli hardware, anche molto diversi, su cui è disponibile. Ad esempio, invece di preoccuparsi di allocare e deallocare memoria manualmente, Python gestisce queste operazioni automaticamente. Questo libera il programmatore dai dettagli del sistema operativo e dell'elettronica, permettendogli di concentrarsi sulla logica del problema da risolvere.

Ciò ha un effetto importante sulla versatilità perché spesso è utilizzato come *interfaccia utente* per linguaggi di livello più basso come C, C++ o Fortran. Questo permette a Python di sfruttare le prestazioni dei linguaggi compilati per le parti critiche e computazionalmente intensive del codice, mantenendo al contempo una sintassi semplice e leggibile per la maggior parte del programma. Buoni compilatori per i linguaggi compilati classici possono sì generare codice binario che gira più velocemente di Python, tuttavia, nella maggior parte dei casi, le prestazioni delle applicazioni codificate in Python sono sufficienti.

7.2. Python come linguaggio multiparadigma

Python è un linguaggio di programmazione multiparadigma, il che significa che supporta diversi paradigmi di programmazione, permettendo di mescolare e combinare gli stili a seconda delle necessità dell'applicazione. Ecco alcuni dei paradigmi supportati da Python:

- Programmazione imperativa: Puoi scrivere ed eseguire script Python direttamente dalla linea di comando, permettendo un approccio interattivo e immediato alla programmazione, come se fosse una calcolatrice.
- Programmazione procedurale: In Python, è possibile organizzare il codice in funzioni e moduli, rendendo più semplice la gestione e la riutilizzabilità del codice. Puoi raccogliere il codice in file separati e importarli come moduli, migliorando la struttura e la leggibilità del programma.
- Programmazione orientata agli oggetti: Python supporta pienamente la programmazione orientata agli oggetti, consentendo la definizione di classi e oggetti. Questo paradigma è utile per modellare dati complessi e relazioni tra essi. Le caratteristiche orientate agli oggetti di Python sono concettualmente simili a quelle del C++, ma più semplici da usare.
- Programmazione funzionale: Python include funzionalità di programmazione funzionale, come funzioni di prima classe e di ordine superiore, lambda e strumenti come `map`, `filter` e `reduce`.

Questa flessibilità rende Python adatto a una vasta gamma di applicazioni e consente ai programmatori di scegliere l'approccio più adatto al problema da risolvere.

7.3. Regole formali e esperienziali

Python non è solo un linguaggio con regole sintattiche precise e ben progettate, ma possiede anche una propria filosofia, un insieme di regole di buon senso esperienziali che sono complementari alla sintassi formale. Questa filosofia è spesso riassunta nel **zen di Python**, una raccolta di aforismi che catturano i principi fondamentali del design di Python. Tali principi aiutano i programmatori a comprendere e utilizzare al meglio le potenzialità del linguaggio e dell'ecosistema Python.

Ecco alcuni dei principi dello zen di Python[1]:

- La leggibilità conta: Il codice dovrebbe essere scritto in modo che sia facile da leggere e comprendere.
- Esplicito è meglio di implicito: È preferibile scrivere codice chiaro e diretto piuttosto che utilizzare scorciatoie criptiche.
- Semplice è meglio di complesso: Il codice dovrebbe essere il più semplice possibile per risolvere il problema.
- Complesso è meglio di complicato: Quando la semplicità non è sufficiente, la complessità è accettabile, ma il codice non dovrebbe mai essere complicato.
- Pratico batte puro: Le soluzioni pragmatiche sono preferibili alle soluzioni eleganti ma poco pratiche.

Questi principi, insieme alle regole sintattiche, guidano il programmatore nell'adottare buone pratiche di sviluppo e nel creare codice che sia non solo funzionale ma anche mantenibile e comprensibile da altri.

7.4. L'ecosistema

Fino ad ora abbiamo visto Python come linguaggio, ma è molto di più: Python è anche una vasta collezione di strumenti e risorse a disposizione degli sviluppatori, strutturata in un ecosistema completo, di cui il linguaggio ne rappresenta la parte formale. Questo ecosistema è disponibile completamente, anche come sorgente, sul sito ufficiale python.org.

7.4.1. L'interprete

L'interprete Python è lo strumento di esecuzione dei programmi. È il software che legge ed esegue il codice Python. Python è un linguaggio interpretato, il che significa che il codice viene eseguito direttamente dall'interprete, senza bisogno di essere compilato in un linguaggio macchina. Esistono diverse implementazioni dell'interprete Python:

- **CPython**: L'implementazione di riferimento dell'interprete Python, scritta in C. È la versione più utilizzata e quella ufficiale.
- PyPy: Un interprete alternativo che utilizza tecniche di compilazione just-in-time (JIT) per migliorare le prestazioni.
- Jython: Un'implementazione di Python che gira sulla JVM (Java Virtual Machine).
- IronPython: Un'implementazione di Python integrata col .NET Framework della Microsoft.

[1] PEP 20 – The Zen of Python

7.4.2. L'ambiente di sviluppo

IDLE (integrated development and learning environment) è l'ambiente di sviluppo integrato ufficiale per Python. È incluso nell'installazione standard di Python ed è progettato per essere semplice e facile da usare, ideale per i principianti. Offre diverse funzionalità utili:

- Editor di codice: Con evidenziazione della sintassi, indentazione automatica e controllo degli errori.
- Shell interattiva: Permette di eseguire codice Python in modo interattivo.
- Strumenti di debug: Include un debugger integrato con punti di interruzione e stepping.

7.4.3. Le librerie standard

Una delle caratteristiche più potenti di Python è il vasto insieme di librerie[2] utilizzabili in CPython e IDLE, che fornisce moduli e pacchetti per quasi ogni necessità di programmazione. Alcuni esempi, tra le decine e al solo allo scopo di illustrarne la varietà, includono:

- `os`: Fornisce funzioni per interagire con il sistema operativo.
- `sys`: Offre accesso a funzioni e oggetti del runtime di Python.
- `datetime`: Consente di lavorare con date e orari.
- `json`: Permette di leggere e scrivere dati in formato JSON.
- `re`: Supporta la manipolazione di stringhe tramite espressioni regolari.
- `http`: Include moduli per l'implementazione di client e server HTTP.
- `unittest`: Fornisce un framework per il testing del codice.
- `math` e `cmath`: Contengono funzioni matematiche di base e complesse.
- `itertools`, `functools`, `operator`: Offrono supporto per il paradigma di programmazione funzionale.
- `csv`: Gestisce la lettura e scrittura di file CSV.
- `typing`: Fornisce supporto per l'annotazione dei tipi di variabili, funzioni e classi.
- `email`: Permette di creare, gestire e inviare email, facilitando la manipolazione di messaggi email MIME.
- `hashlib`: Implementa algoritmi di hash sicuri come SHA-256 e MD5.
- `asyncio`: Supporta la programmazione asincrona per la scrittura di codice concorrente e a bassa latenza.
- `wave`: Fornisce strumenti per leggere e scrivere file audio WAV.

[2] Documentazione delle librerie standard di Python

7.4.4. Moduli di estensione

Python supporta l'estensione del suo core tramite moduli scritti in C, C++ o altri linguaggi. Questi moduli permettono di ottimizzare parti critiche del codice o di interfacciarsi con librerie e API esterne:

- **Cython**: Permette di scrivere moduli C estesi utilizzando una sintassi simile a Python. Cython è ampiamente utilizzato per migliorare le prestazioni di parti critiche del codice, specialmente in applicazioni scientifiche e di calcolo numerico. Ad esempio, molte librerie scientifiche popolari come SciPy e scikit-learn utilizzano Cython per accelerare le operazioni computazionalmente intensive.
- **ctypes**: Permette di chiamare funzioni in librerie dinamiche C direttamente da Python. È utile per interfacciarsi con librerie esistenti scritte in C, rendendo Python estremamente versatile per l'integrazione con altre tecnologie. Ciò è utile in applicazioni che devono interfacciarsi con hardware specifico o utilizzare librerie legacy.
- **CFFI** (C Foreign Function Interface): Un'altra interfaccia per chiamare librerie C da Python. È progettata per essere facile da usare e per supportare l'uso di librerie C complesse con Python. CFFI è utilizzato in progetti come PyPy e gevent, permettendo di scrivere codice ad alte prestazioni e di gestire le chiamate a funzioni C in modo efficiente.

7.4.5. Le utility e gli strumenti aggiuntivi

Python include anche una serie di strumenti e utility che facilitano lo sviluppo e la gestione dei progetti:

- **pip**: Il gestore dei pacchetti di Python. Permette di installare e gestire moduli aggiuntivi, cioè non inclusi nello standard.
- **venv**: Uno strumento per creare ambienti virtuali isolati, che permettono di gestire separatamente le dipendenze di diversi progetti.
- Documentazione: Python include una documentazione dettagliata, accessibile tramite il comando pydoc o attraverso il sito ufficiale.

7.5. Un esempio di algorimo in Python: il bubble sort

Per chiudere il capitolo sul primo approccio a Python, possiamo confrontare un algoritmo, di bassa complessità ma non triviale, in diversi linguaggi di programmazione. Un buon esempio potrebbe essere l'implementazione dell'algoritmo di ordinamento *bubble sort* di una lista di valori. Vediamo come viene scritto in Python, C, C++, Java, Rust e Scala:

- Python in versione procedurale:

```python
def bubble_sort(arr):
  n = len(arr)

  for i in range(n):
    for j in range(0, n-i-1):
      if arr[j] > arr[j+1]:
        arr[j], arr[j+1] = arr[j+1], arr[j]

# Esempio di utilizzo
```

```
arr = [64, 34, 25, 12, 22, 11, 90]

bubble_sort(arr)

print("Array ordinato con bubble sort: ", arr)
```

- Python in versione sintatticamente orientata agli oggetti, ma praticamente procedurale:

```
class BubbleSort:
  @staticmethod
  def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
      for j in range(0, n-i-1):
        if arr[j] > arr[j+1]:
          arr[j], arr[j+1] = arr[j+1], arr[j]

# Esempio di utilizzo
arr = [64, 34, 25, 12, 22, 11, 90]

BubbleSort.bubble_sort(arr)

print("Array ordinato con bubble sort: ", arr)
```

- Python in versione orientata agli oggetti, con una interfaccia di ordinamento implementata con due algoritmi (bubble e insertion sort):

```
from abc import ABC, abstractmethod                                    ①

# Classe astratta per algoritmi di ordinamento
class SortAlgorithm(ABC):                                              ②
  def __init__(self, arr):
    self._arr = arr

  @abstractmethod
  def sort(self):                                                      ③
    # Metodo astratto che deve essere implementato dalle sottoclassi
    pass

  def get_array(self):
    # Metodo per ottenere l'array corrente
    return self._arr

  def set_array(self, arr):
    # Metodo per impostare un nuovo array
    self._arr = arr

# Implementazione dell'algoritmo di bubble sort
class BubbleSort(SortAlgorithm):                                       ④
  def sort(self):
```

```
    n = len(self._arr)

    for i in range(n):
      for j in range(0, n-i-1):
        if self._arr[j] > self._arr[j+1]:
          self._arr[j], self._arr[j+1] = self._arr[j+1], self._arr[j]

# Implementazione dell'algoritmo di insertion sort
class InsertionSort(SortAlgorithm):
  def sort(self):
    for i in range(1, len(self._arr)):
      key = self._arr[i]

      j = i - 1

      while j >= 0 and key < self._arr[j]:
        self._arr[j + 1] = self._arr[j]

        j -= 1

      self._arr[j + 1] = key

# Esempio di utilizzo con bubble sort
arr = [64, 34, 25, 12, 22, 11, 90]

bubble_sorter = BubbleSort(arr)

bubble_sorter.sort()

print("Array ordinato con bubble sort: ", bubble_sorter.get_array())

# Esempio di utilizzo con insertion sort
arr = [64, 34, 25, 12, 22, 11, 90]

insertion_sorter = InsertionSort(arr)

insertion_sorter.sort()

print("Array ordinato con insertion sort: ", insertion_sorter.get_array())
```

① Importiamo `ABC` e `abstractmethod` dal modulo `abc` per definire la classe astratta.
② `SortAlgorithm` è una classe astratta che rappresenta l'interfaccia di algoritmi di ordinamento.
③ `sort` è un metodo astratto che deve essere implementato nelle sottoclassi.
④ `BubbleSort` è una sottoclasse di `SortAlgorithm` che implementa l'algoritmo di ordinamento a bolle. Idem per `InsertionSort`.

- Python in versione funzionale:

```
def bubble_sort(arr):
  def sort_pass(arr, n):                                                    ①
```

```
    if n == 1:
      return arr

    new_arr = arr[:]                                                    ②

    for i in range(n - 1):
      if new_arr[i] > new_arr[i + 1]:
        new_arr[i], new_arr[i + 1] = new_arr[i + 1], new_arr[i]

    return sort_pass(new_arr, n - 1)                                    ③

  return sort_pass(arr, len(arr))                                       ④

# Esempio di utilizzo
arr = [64, 34, 25, 12, 22, 11, 90]

sorted_arr = bubble_sort(arr)

print("Sorted array is:", sorted_arr)
```

① All'interno di `bubble_sort`, è definita una funzione interna `sort_pass` che esegue un singolo passaggio dell'algoritmo di ordinamento a bolle.
② Viene creata una copia dell'array arr chiamata `new_arr`. Poi, per ogni coppia di elementi `(new_arr[i], new_arr[i + 1])`, se `new_arr[i]` è maggiore di `new_arr[i + 1]`, vengono scambiati.
③ La funzione `sort_pass` viene chiamata ricorsivamente con `new_arr` e decrementando n di 1.
④ La funzione `bubble_sort` avvia il processo chiamando `sort_pass` con l'array completo e la sua lunghezza.

- C:

```
#include <stdio.h>

void bubble_sort(int arr[], int n) {
  int i, j, temp;

  for (i = 0; i < n-1; i++) {
    for (j = 0; j < n-i-1; j++) {
      if (arr[j] > arr[j+1]) {
        temp = arr[j];

        arr[j] = arr[j+1];

        arr[j+1] = temp;
      }
    }
  }
}

int main() {
  int arr[] = {64, 34, 25, 12, 22, 11, 90};
```

```
  int n = sizeof(arr)/sizeof(arr[0]);

  bubble_sort(arr, n);

  printf("Array ordinato con bubble sort: ");

  for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }

  return 0;
}
```

- C++:

```
#include <iostream>
using namespace std;

class BubbleSort {
public:
  void sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
      for (int j = 0; j < n-i-1; j++) {
        if (arr[j] > arr[j+1]) {
          int temp = arr[j];

          arr[j] = arr[j+1];

          arr[j+1] = temp;
        }
      }
    }
  }

  void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
    }

    cout << endl;
  }
};

int main() {
  int arr[] = {64, 34, 25, 12, 22, 11, 90};
  int n = sizeof(arr)/sizeof(arr[0]);

  BubbleSort bs;
  bs.sort(arr, n);
```

```
  cout << "Array ordinato con bubble sort: ";
  bs.printArray(arr, n);

  return 0;
}
```

- Java:

```
public class BubbleSort {

  public static void bubbleSort(int arr[]) {
    int n = arr.length;

    for (int i = 0; i < n-1; i++) {
      for (int j = 0; j < n-i-1; j++) {
        if (arr[j] > arr[j+1]) {

          int temp = arr[j];

          arr[j] = arr[j+1];

          arr[j+1] = temp;
        }
      }
    }
  }

  public static void main(String args[]) {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};

    bubbleSort(arr);

    System.out.println("Array ordinato con bubble sort: ");

    for (int i = 0; i < arr.length; i++) {
      System.out.print(arr[i] + " ");
    }
  }
}
```

- Rust:

```
fn bubble_sort(arr: &mut [i32]) {
  let n = arr.len();

  for i in 0..n {
    for j in 0..n-i-1 {
      if arr[j] > arr[j+1] {
        arr.swap(j, j+1);
      }
    }
```

```
  }
}

fn main() {
  let mut arr = [64, 34, 25, 12, 22, 11, 90];

  bubble_sort(&mut arr);

  println!("Array ordinato con bubble sort: {:?}", arr);
}
```

- Scala:

```
object BubbleSort {
  def bubbleSort(arr: Array[Int]): Unit = {
    val n = arr.length

    for (i <- 0 until n) {
      for (j <- 0 until n - i - 1) {
        if (arr(j) > arr(j + 1)) {
          val temp = arr(j)

          arr(j) = arr(j + 1)

          arr(j + 1) = temp
        }
      }
    }
  }

  def main(args: Array[String]): Unit = {
    val arr = Array(64, 34, 25, 12, 22, 11, 90)

    bubbleSort(arr)

    println("Array ordinato con bubble sort: " + arr.mkString(", "))
  }
}
```

Confrontando questi esempi, possiamo osservare le differenze sintattiche e di stile tra Python ed altri, importanti, linguaggi. Python si distingue per la sua sintassi concisa e espressiva soprattutto nella versione procedurale. L'implementazione colla gerarchia di oggetti ha un piccolo incremento di complessità che è ripagato dalla possibilità di creare gerarchie di algoritmi di ordinamento, con impatti nulli sul codice preesistente.

La versione procedurale in Python e l'implementazione C, già a primo acchito, presentano un evidente diverso grado di chiarezza del codice. Inoltre, la riga `int n = sizeof(arr)/sizeof(arr[0]);` in C si rende necessaria per calcolare il numero di valori a partire dalle dimensioni totale della lista e del singolo elemento, rispetto a `n = len(arr)` di Python, dove chiediamo direttamente il numero di valori.

Il C++ e Java aggiungono caratteristiche relative agli oggetti e funzionalità di alto livello rispetto a C, al prezzo di una sintassi più complessa e verbosa. Rust e Scala sono linguaggi più moderni e si pongono nel mezzo tra C,

C++ e Java e Python.

8. Scaricare e installare Python

8.1. Scaricamento

1. Visita il sito ufficiale di Python: Vai su python.org.
2. Naviga alla pagina di download: Clicca su *Downloads* nel menu principale.
3. Scarica il pacchetto di installazione:
    - Per Windows: Cerca Python 3.12.x e fai partire il download (assicurati di scaricare la versione più recente).
    - Per macOS: Come per Windows.
    - Per Linux: Python è spesso preinstallato. Se non lo è, usa il gestore di pacchetti della tua distribuzione (ad esempio `apt` per Ubuntu: `sudo apt-get install python3`).

8.2. Installazione

1. Esegui il file di installazione:
    - Su Windows: Esegui il file `.exe` scaricato. Assicurati di selezionare l'opzione `Add Python to PATH` durante l'installazione.
    - Su macOS: Apri il file `.pkg` scaricato e segui le istruzioni.
    - Su Linux: Usa il gestore di pacchetti per installare Python.
2. Verifica l'installazione:
    - Apri il terminale (Command Prompt su Windows, Terminal su macOS e Linux).
    - Digita `python --version` o `python3 --version` e premi Invio. Dovresti vedere la versione di Python installata.

8.3. Esecuzione del primo programma: "Hello, World!"

È consuetudine eseguire come primo programma la visualizzazione della stringa "Hello, World!"[1]. Possiamo farlo in diversi modi e ciò è una delle caratteristiche più apprezzate di Python.

[1] La tradizione del programma "Hello, World!" ha una lunga storia che risale ai primi giorni della programmazione. Questo semplice programma è generalmente il primo esempio utilizzato per introdurre i nuovi programmatori alla sintassi e alla struttura di un linguaggio di programmazione. Il programma "Hello, World!" è diventato famoso grazie a Brian Kernighan, che lo ha incluso nel suo libro (Kernighan e Ritchie 1988) pubblicato nel 1978. Tuttavia, il suo utilizzo risale a un testo precedente di Kernighan, (Kernighan 1973), pubblicato nel 1973, dove veniva utilizzato un esempio simile.

8.3.1. REPL

Il primo modo prevede l'utilizzo del REPL di Python. Il REPL (read-eval-print loop) è un ambiente interattivo di esecuzione di comandi Python generato dall'interprete, secondo il ciclo:

1. Read: Legge un input dell'utente.
2. Eval: Valuta l'input.
3. Print: Visualizza il risultato dell'esecuzione.
4. Loop: Ripete il ciclo.

Eseguiamo il nostro primo "Hello, World!":

1. Apri il terminale ed esegui l'interprete Python digitando `python` o `python3` e premi il tasto di invio della tastiera.
2. Scrivi ed esegui il programma:

```
print("Hello, World!")
```

Premi il tasto di invio per vedere il risultato immediatamente.

> **Attenzione**
>
> Il REPL e l'interprete Python sono strettamente collegati, ma non sono esattamente la stessa cosa. Quando avvii l'interprete Python senza specificare un file di script da eseguire (digitando semplicemente `python` o `python3` nel terminale), entri in modalità REPL. Nel REPL, l'interprete Python legge l'input direttamente dall'utente, lo esegue, stampa il risultato e poi attende il prossimo input. In sintesi, l'interprete può eseguire programmi Python completi salvati in file, il REPL è progettato per un'esecuzione interattiva e immediata di singole istruzioni.

8.3.2. Interprete

Un altro modo per eseguire il nostro programma "Hello, World!" è utilizzare l'interprete Python per eseguire un file di codice sorgente. Questo metodo è utile per scrivere programmi più complessi e per mantenere il codice per usi futuri.

Ecco come fare sui diversi sistemi operativi.

8.4. Windows

1. Crea un file di testo:
   i. Apri il tuo editor di testo preferito, come Notepad.
   ii. Scrivi il seguente codice nel file:

```
print("Hello, World!")
```

   iii. Salva il file con il nome `hello.txt`.

2. Rinomina il file (facoltativo): se desideri mantenere il file senza estensione `.txt`, puoi rinominarlo in `hello` direttamente dall'Esplora file.
3. Esegui il file Python:
   i. Apri il prompt dei comandi.
   ii. Naviga fino alla directory in cui hai salvato il file. Ad esempio, se il file si trova nella cartella `Documenti`, puoi digitare:

```
cd %HOMEPATH%\Documenti
```

   iii. Esegui l'interprete Python passando come argomento il file che hai creato:

```
python hello.txt
```

   iv. oppure, se il tuo sistema utilizza `python3`:

```
python3 hello.txt
```

4. Visualizza il risultato:

```
Hello, World!
```

8.5. macOS

1. Crea un file di testo:
   i. Apri il tuo editor di testo preferito, come TextEdit.
   ii. Scrivi il seguente codice nel file:

```
print("Hello, World!")
```

   iii. Salva il file con il nome `hello.txt`.
2. Rinomina il file (facoltativo): se desideri mantenere il file senza estensione `.txt`, puoi rinominarlo in `hello` direttamente dal Finder.
3. Esegui il file Python:
   i. Apri il terminale del sistema operativo.
   ii. Naviga fino alla directory in cui hai salvato il file. Ad esempio, se il file si trova nella cartella `Documenti`, puoi digitare:

```
cd ~/Documents
```

   iii. Esegui l'interprete Python passando come argomento il file che hai creato:

```
python3 hello.txt
```

4. Visualizza il risultato:

```
Hello, World!
```

8.6. Linux

1. Crea un file di testo:
   i. Apri il tuo editor di testo preferito, come Gedit o Nano.
   ii. Scrivi il seguente codice nel file:

```
print("Hello, World!")
```

   iii. Salva il file con il nome `hello.txt`.

2. Rinomina il file (facoltativo): se desideri mantenere il file senza estensione `.txt`, puoi rinominarlo in `hello` utilizzando il comando `mv` nel terminale:

```
mv hello.txt hello
```

3. Esegui il file Python:
   i. Apri il terminale del sistema operativo.
   ii. Naviga fino alla directory in cui hai salvato il file. Ad esempio, se il file si trova nella cartella `Documenti`, puoi digitare:

```
cd ~/Documenti
```

   iii. Esegui l'interprete Python passando come argomento il file che hai creato:

```
python3 hello.txt
```

4. Visualizza il risultato:

```
Hello, World!
```

Con queste istruzioni, dovresti essere in grado di eseguire il programma "Hello, World!" utilizzando un file Python su Windows, macOS e Linux.

8.6.1. IDE

Utilizzo di un IDE (integrated development environment) installato sul computer. Ecco alcuni dei più comuni e gratuiti.

8.7. IDLE

È incluso con l'installazione di Python.

1. Avvia IDLE.
2. Crea un nuovo file (`File -> New File`).
3. Scrivi il programma:

```
print("Hello, World!")
```

4. Salva il file (`File -> Salva`).
5. Esegui il programma (`Run -> Run Module`).

8.8. PyCharm

Proprietario ma con una versione liberamente fruibile.

1. Scarica e installa PyCharm da jetbrains.com/pycharm/download.
2. Crea un nuovo progetto associando l'interprete Python.
3. Crea un nuovo file Python (`File -> New -> Python File`).
4. Scrivi il programma:

```
print("Hello, World!")
```

5. Esegui il programma (`Run -> Run...`).

8.9. Visual Studio Code

Proprietario ma liberamente fruibile.

1. Scarica e installa VS Code da code.visualstudio.com.
2. Installa l'estensione Python.
3. Apri o crea una nuova cartella di progetto.
4. Crea un nuovo file Python (`File -> Nuovo file`).
5. Scrivi il programma:

```
print("Hello, World!")
```

6. Salva il file con estensione `.py`, ad esempio `hello_world.py`.
7. Esegui il programma utilizzando il terminale integrato (`Visualizza -> Terminale`) e digitando `python hello_world.py`.

8.9.1. Esecuzione nel browser

Puoi eseguire Python direttamente nel browser, senza installare nulla. Anche qui abbiamo diverse alternative, sia eseguendo il codice localmente, che utilizzando piattaforme online.

8.10. Repl.it

1. Visita repl.it.
2. Crea un nuovo progetto selezionando Python.
3. Scrivi il programma:

```
print("Hello, World!")
```

4. Clicca su "Run" per eseguire il programma.

8.11. Google Colab

1. Visita colab.research.google.com.
2. Crea un nuovo notebook.
3. In una cella di codice, scrivi:

```
print("Hello, World!")
```

4. Premi il pulsante di esecuzione accanto alla cella.

8.12. PyScript

1. Visita il sito ufficiale di PyScript per ulteriori informazioni su come iniziare.
2. Crea un file HTML con il seguente contenuto:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello, World with PyScript</title>
    <link rel="stylesheet" href="https://pyscript.net/latest/pyscript.css">
    <script defer src="https://pyscript.net/latest/pyscript.js"></script>
</head>
<body>
    <py-script>
        print("Hello, World!")
    </py-script>
</body>
</html>
```

4. Salva il file con estensione `.html` (ad esempio, `hello.html`).
5. Apri il file salvato in un browser web. Vedrai l'output `Hello, World!` direttamente nella pagina.

8.12.1. Jupyter Notebook

Jupyter Notebook è un ambiente di sviluppo interattivo per la programmazione che permette di creare e condividere documenti contenenti codice eseguibile, visualizzazioni, testo formattato e altro ancora. Originariamente sviluppato come parte del progetto IPython, Jupyter supporta non solo Python, ma anche numerosi altri linguaggi di programmazione attraverso i cosiddetti kernel tra cui R, Julia e Scala.

8.13. Uso locale

1. Assicurati di avere Python e Jupyter installati sul tuo computer. Se non li hai, puoi installarli utilizzando Anaconda o pip:

```
pip install notebook
```

2. Avvia Jupyter Notebook dal terminale:

```
jupyter notebook
```

3. Crea un nuovo notebook Python.

4. In una cella di codice, scrivi:

```
print("Hello, World!")
```

5. Premi `Shift + Enter` per eseguire la cella.

8.14. JupyterHub

1. Visita l'istanza di JupyterHub della tua istituzione o azienda (maggiori informazioni).

2. Accedi con le tue credenziali.

3. Crea un nuovo notebook Python.

4. In una cella di codice, scrivi:

```
print("Hello, World!")
```

5. Premi `Shift + Enter` per eseguire la cella.

8.15. Binder

1. Visita mybinder.org.
2. Inserisci l'URL del repository GitHub che contiene il tuo notebook o il tuo progetto Python.
3. Clicca su "Launch".
4. Una volta avviato l'ambiente, crea un nuovo notebook o apri uno esistente.
5. In una cella di codice, scrivi:

```
print("Hello, World!")
```

6. Premi `Shift + Enter` per eseguire la cella.

Binder è un servizio simile a Colab, anche se quest'ultimo offre strumenti generalmente più avanzati in termini di risorse computazionali e collaborazione. Binder di contro è basato su GitHub e ciò può essere utile in alcuni contesti.

9. La sintassi

Per iniziare ad imparare Python come linguaggio, partiamo dalla sua stuttura lessicale, cioè dall'insieme delle sue regole sintattiche più significative, sia per imparare le regole di composizione di programmi comprensibili all'interprete e sfrutturne appieno tutte le potenzialità, sia per essere in grado di utilizzare il codice scritto da altri.

Ogni programma Python è costituito da una serie di file di testo contenenti il codice sorgente con una certa codifica, il default essendo l'UTF-8, ed ogni file si può vedere come una sequenza di istruzioni, righe e token. Le istruzioni danno la granularità dell'algoritmo, le righe definiscono come queste istruzioni sono distribuite nel testo e, infine, i token sono gli elementi atomici che hanno un significato per il linguaggio.

9.1. Premessa

9.1.1. Elementi semantici

Per dare un senso, anche intuitivo, agli esempi presentati assieme ai concetti sintattici che saranno introdotti nel seguito, è opportuno definire, informalmente, alcuni elementi semantici, a partire da nozioni di base:

- Variabile: È un nome dato ad un valore presente in memoria.
- Espressione: È una combinazione di elementi di sintassi che può essere valutata per produrre un valore. In altre parole, un'espressione è un insieme di elementi come letterali, variabili, accessi ad attributi, operatori o chiamate di funzione che restituiscono tutti un valore.
- Funzione: È un insieme di istruzioni che può essere parametrizzato da una serie di input predefiniti e può avere una serie di output, a cui è associato un nome.
- Classe: È una definizione che raggruppa un insieme di attributi e operazioni che agiscono sugli attributi della propria o di altre classi.
- Oggetto: È un'istanza di una classe, cioè un particolare valore di una classe che è stato creato in memoria.
- Modulo: È un file contenente definizioni di variabili, funzioni e classi che possono essere importate e utilizzate in altri programmi o moduli.

9.1.2. Le funzioni `print()` e `help()`

Nel seguito useremo molto la funzione `print()` che permette di visualizzare informazioni sullo schermo. Ciò è fondamentale per capire cosa sta succedendo nel programma in modo immediato ed è una semplice alternativa a strumenti di *ispezione* o *debugging* messi a diposizione da IDE.

Usare `print()` è facile: basta scrivere `print(espressione)` e mettere nelle parentesi una `espressione` che si vuole visualizzare.

Ad esempio, per stampare `"Ciao, mondo!"`, si scrive:

```
print("Ciao, mondo!")
```

Nel REPL, invece, ciò si ottiene semplicemente dando l'invio:

```
>>> "Ciao, mondo!"
'Ciao, mondo!'
```

Una seconda funzione molto utile è `help()`. Fornisce informazioni sulla documentazione di moduli, funzioni, classi e metodi.

Ad esempio, per ottenere informazioni sulla funzione `print()`, possiamo scrivere:

```
>>> help(print)                                                    (1)
```

(1) Output di `help(print)`:

```
Help on built-in function print in module builtins:
print(*args, sep=' ', end='\n', file=None, flush=False)
  Prints the values to a stream, or to sys.stdout by default.

  sep
    string inserted between values, default a space.
  end
    string appended after the last value, default a newline.
  file
    a file-like object (stream); defaults to the current sys.stdout.
  flush
    whether to forcibly flush the stream.
```

Nel REPL offre la massima utilità, perché eseguendo `help()` si passa dalla modalità interattiva del REPL a quella di navigazione del contenuto testuale della risposta, funzionalità molto comoda soprattutto per testi lunghi. Per ritornare al REPL, basta premere `q` e poi `Invio`.

9.2. Righe

Le righe sono di due tipi: **logiche** e **fisiche**. Le seconde sono le più facilmente individuabili nel testo di un programma, perché sono terminate da un carattere di a capo. Una o più righe fisiche costituiscono una riga logica che corrisponde ad una istruzione. Esiste una eccezione, poco usata e consigliata in Python, per cui una riga fisica contiene più istruzioni separate da `;`.

Vi sono due modi per dividere una riga logica in righe fisiche. Il primo è terminare con il backslash (\, poco usata la traduzione *barra rovesciata* o simili) tutte le righe fisiche meno l'ultima (intendendo con ciò che il backslash precede l'a capo):

```
x = 1 + 2 + \                                                    ①
     3

if x > 5 and \                                                   ②
   x < 9:                                                        ③

  print("5 < x < 9")
```

① L'istruzione di assegnamento è spezzata su due righe fisiche.
② L'istruzione condizionale ha due espressioni che devono essere entrambe vere, ognuna su una riga fisica.
③ Non importa quanto sono indentate le righe fisiche successive alla prima e ciò può essere sfruttato per incrementare la leggibilità, ad esempio, allineando le espressioni `x > 5` e `x < 9` in colonna.

Il secondo è per mezzo di parentesi, giacché tutte le righe fisiche che seguono una con parentesi tonda `(`, quadra `[` o graffa `{` aperta, fino a quella con l'analoga parentesi chiusa, sono unite in una logica. Le regole di indentazione, che vedremo nel seguito, si applicano solo alla prima riga fisica.

Esempi sintatticamente corretti ma sconsigliabili, per l'inerente illeggibilità:

```
x = (1 + 2                                                       ①
     + 3 + 4)

y = [1, 2,
           3, 4 +                                                ②
      5]

z = [1, 2                                                        ③
     , 3, 4]
```

① L'espressione è spezzata su due righe fisiche e le parentesi tonde rappresentano un'alternativa all'uso del backslash.
② Le righe fisiche della lista non hanno la stessa indentazione e una espressione è spezzata su due righe.
③ La lista è spezzata su due righe fisiche e un delimitatore inizia la riga anziché terminare la precedente.

9.3. Commenti

Un **commento** inizia con un carattere cancelletto (`#`) e termina alla fine della riga fisica. I commenti non possono coesistere con il backslash come separatore di riga logica, giacché entrambi devono chiudere la riga fisica.

Esempi non sintatticamente corretti:

```
x = 1 + 2 + \ # Commento                                         ①
     3

if x > 5 and # Commento \                                        ②
   x < 9:
  print("5 < x < 9")
```

① Il backslash deve terminare la riga fisica, quindi non può essere seguito da un commento. Se necessario può andare o alla riga successiva, scelta consigliata, o alla precedente. L'interprete segnalerà l'errore `SyntaxError`.
② Il commento rende il backslash parte di esso quindi non segnala più la fine della riga fisica e, all'esecuzione, si avrà anche qui un errore di tipo `SyntaxError`, perché `and` deve essere seguito da un'espressione.

9.4. Indentazione

Indentazione significa che spazi o, in alternativa, tabulazioni precedono un carattere che non sia nessuno dei due. Il numero di spazi, ottenuto dopo la trasformazione delle tabulazioni in spazi, si definisce livello di indentazione.

L'indentazione del codice è il modo che Python utilizza per raggruppare le istruzioni in un blocco, ove tutte devono presentare la medesima indentazione. La prima riga logica che ha una indentazione minore della precedente, segnala che il blocco è stato chiuso proprio da quest'ultima. Anche le clausole di un'istruzione composta devono avere la stessa indentazione.

La prima istruzione di un file o la prima inserita al prompt `>>>` del REPL non deve presentare spazi o tabulazioni, cioè ha un livello di indentazione pari a 0.

Alcuni esempi:

- Definizione di una funzione:

```
def somma(a, b):                                                    ①
  risultato = a + b                                                 ②

  return risultato
```

① Prima riga senza indentazione.
② Questa riga e la successiva appartengono allo stesso blocco e, pertanto, hanno la medesima indentazione.

- Test di condizione:

```
x = 10

if x < 0:                                                           ①
    print("x è negativo")                                           ②

elif x == 0:
      print("x è zero")

else:
        print("x è positivo")
```

① Le tre clausole `if` , `then` e `else` della medesima istruzione composta di test di condizione, devono avere identica indentazione.
② I tre blocchi hanno come unico vincolo quello di avere un livello maggiore di indentazione della riga precedente. I blocchi corrispondenti alle diverse clausole non devono avere lo stesso livello di indentazione, anche se è buona prassi farlo.

> **Attenzione**
>
> Non si possono avere sia spazi che tabulazioni per definire il livello di indentazione nello stesso file. Ciò perché renderebbe ambiguo il numero di spazi che si ottiene dopo la trasformazione delle tabulazioni in spazi. Quindi, o si usano spazi, scelta raccomandata, o tabulazioni.

9.5. Token

Le righe logiche sono composte da token che si categorizzano in parole chiave, identificatori, operatori, delimitatori e letterali. I token sono separati da un numero arbitrario di spazi e tabulazioni. Ad esempio:

```
x = 1  +  2  +  3

if    x > 5    and    x < 9:
  print("5 < x < 9")
```

9.5.1. Identificatori

Un identificatore è un nome assegnato ad un oggetto, cioè una variabile, una funzione, una classe, un modulo e altro. Esso è *case sensitive* cioè `python` e `Python` sono due identificatori diversi.

Alcuni esempi:

```
intero = 42                                                    (1)
decimale = 3.14                                                (2)
testo = "Ciao, mondo!"                                         (3)
lista = [1, 2, 3]                                              (4)
dizionario = {"chiave": "valore"}                              (5)

def mia_funzione():                                            (6)
  print("Questa è una funzione")

class MiaClasse:                                               (7)
  def __init__(self, valore):                                  (8)
    self.valore = valore                                       (9)

  def metodo(self):                                            (10)
    print("Questo è un metodo della classe")

import math                                                    (11)

def mio_generatore():                                          (12)
  yield 1
  yield 2
  yield 3

mio_oggetto = MiaClasse(10)                                    (13)
```

① Identificatore di numero intero: `intero`.
② Identificatore di numero decimale: `decimale`.
③ Identificatore di stringa: `testo`.
④ Identificatore di lista: `lista`.
⑤ Identificatore di dizionario: `dizionario`.
⑥ Identificatore di funzione: `mia_funzione`.
⑦ Identificatore di classe: `MiaClasse`.
⑧ Identificatore di metodo e parametro: `__init__` e `valore`.
⑨ Identificatore di attributo: `valore`.
⑩ Identificatore di metodo: `metodo`.
⑪ Identificatore di modulo: `math`.
⑫ Identificatore di generatore: `mio_generatore`.
⑬ Identificatore di oggetto: `mio_oggetto`.

9.5.2. Parole chiave

Le parole chiave sono parole che non possono essere usate per scopi diversi da quelli predefiniti nel linguaggio e, quindi, non possono essere usate come identificatori. Ad esempio, `True` che rappresenta il valore logico di verità, non può essere usato per definire ad esempio una variabile.

Esistono anche delle parole chiave contestuali, cioè che sono tali solo in alcuni contesti ed altrove possono essee usate come identificatori. Usiamo il codice seguente per ottenere una lista di parole chiave e parole chiave contestuali:

```
import keyword                                          ①

parole_chiave = keyword.kwlist                          ②

parole_chiave_contestuale = keyword.softkwlist          ③

print(parole_chiave)                                    ④

print(parole_chiave_contestuale)                        ⑤
```

① Importa il modulo `keyword`.
② Ottiene la lista delle parole chiave.
③ Ottiene la lista delle parole chiave contestuali.
④ Stampa la lista delle parole chiave.
⑤ Stampa la lista delle parole chiave contestuali.

Nella tabella seguente invece un elenco completo con breve descrizione:

Tabella 9.1.: Parole chiave di Python

Parola chiave	Descrizione
Valori booleani	
`False`	Valore booleano falso
`True`	Valore booleano vero
Operatori logici	

Parola chiave	Descrizione
`and`	Operatore logico AND
`or`	Operatore logico OR
`not`	Operatore logico NOT
Operatori di controllo di flusso	
`if`	Utilizzato per creare un'istruzione condizionale
`elif`	Utilizzato per aggiungere condizioni in un blocco if
`else`	Utilizzato per specificare il blocco di codice da eseguire se le condizioni precedenti sono false
`for`	Utilizzato per creare un ciclo for
`while`	Utilizzato per creare un ciclo while
`break`	Interrompe il ciclo in corso
`continue`	Salta l'iterazione corrente del ciclo e passa alla successiva
`pass`	Indica un blocco di codice vuoto
`return`	Utilizzato per restituire un valore da una funzione
Gestione delle eccezioni	
`try`	Utilizzato per definire un blocco di codice da eseguire e gestire le eccezioni
`except`	Utilizzato per catturare le eccezioni in un blocco try-except
`finally`	Blocco di codice che viene eseguito alla fine di un blocco try, indipendentemente dal fatto che si sia verificata un'eccezione
`raise`	Utilizzato per sollevare un'eccezione
Definizione delle funzioni e classi	
`def`	Utilizzato per definire una funzione
`class`	Utilizzato per definire una classe
`lambda`	Utilizzato per creare funzioni anonime
Gestione contesto di dichiarazione di variabili	
`global`	Utilizzato per dichiarare variabili globali
`nonlocal`	Utilizzato per dichiarare variabili non locali
Operazioni su moduli	
`import`	Utilizzato per importare moduli
`from`	Utilizzato per importare specifici elementi da un modulo
`as`	Utilizzato per creare alias, ad esempio negli import
Operatori di identità e appartenenza	
`in`	Utilizzato per verificare se un valore esiste in una sequenza
`is`	Operatore di confronto di identità
Gestione delle risorse	
`with`	Utilizzato per garantire un'azione di pulizia come il rilascio delle risorse
Programmazione asincrona	
`async`	Utilizzato per definire funzioni asincrone
`await`	Utilizzato per attendere un risultato in una funzione asincrona
Varie	
`del`	Utilizzato per eliminare oggetti
`assert`	Utilizzato per le asserzioni, verifica che un'espressione sia vera
`yield`	Utilizzato per restituire un generatore da una funzione
`None`	Rappresenta l'assenza di valore o un valore nullo

Parola chiave	Descrizione
Parole chiave contestuali	
`match`	Utilizzato nell'istruzione `match` per il pattern matching
`case`	Utilizzato nell'istruzione `match` per definire un ramo
`_`	Utilizzato come identificatore speciale nell'istruzione `match` per indicare un pattern di default o ignorare valori
`type`	Utilizzato in specifici contesti per dichiarazioni di tipo

Esempi di uso di parole chiave contestuali:

- `match`, `case` e `_`:

```
def process_value(value):                                    ①
  match value:                                               ②
    case 1:                                                  ③
      print("Uno")

    case 2:
      print("Due")

    case _:                                                  ④
      print("Altro")

match = "Questo è un identificatore valido"                  ⑤

process_value(1)  # Output: Uno
process_value(2)  # Output: Due
process_value(3)  # Output: Altro

print(match)                                                 ⑥
```

① Definiamo una funzione che utilizza il pattern matching.
② Uso di `match` come parola chiave.
③ Uso di `case` come parola chiave.
④ Uso di `_` come parola chiave.
⑤ Utilizzo di `match` come identificatore per una variabile.
⑥ Output: `Questo è un identificatore valido`.

- `type`:

```
from typing import TypeAlias

type Point = tuple[float, float]                             ①

def distanza(p1: Point, p2: Point) -> float:                 ②
  return ((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2) ** 0.5

punto1: Point = (1.0, 2.0)                                   ③
punto2: Point = (4.0, 6.0)
```

```
print(distanza(punto1, punto2))                                                    ④

print(type(punto1))                                                                ⑤
```

① Uso di `type` come parola chiave.
② Utilizzo dell'alias di tipo `Point` per i parametri `p1` e `p2` della funzione.
③ Definizione delle variabili `punto1` e `punto2` come `Point` utilizzando l'alias di tipo. Questo rende chiaro che entrambe le variabili sono tuple di due `float`, anche se l'interprete non controlla la coerenza tra oggetto e alias di tipo.
④ Output: `5.0`.
⑤ Uso di `type` identificatore di una funzione. Output: `<class 'tuple'>`.

9.5.3. Classi riservate di identificatori

Alcune classi di identificatori (oltre alle parole chiave) hanno significati speciali in Python. Queste classi sono identificate dai pattern di caratteri di sottolineatura (underscore) all'inizio e alla fine dei nomi. Tuttavia, l'uso di questi identificatori non impone limitazioni rigide al programmatore, ma è importante seguire le convenzioni per evitare ambiguità e problemi di compatibilità.

Identificatori speciali:

- `_`:
  - Non importato da `from module import *`: Gli identificatori che iniziano con un singolo underscore non vengono importati con un'istruzione di importazione globale. Questo è un meccanismo per indicare che tali variabili o funzioni sono destinate ad essere *private* al modulo e non dovrebbero essere usate direttamente da altri moduli. Esempio:

In un file `example.py`:

```
_private_variable = "Variabile da non esportare'"                                  ①
```

① Nel modulo `example.py` viene definita la variabile come privata.

In altro modulo diverso da `example.py`:

```
from example import *                                                              ①

print(_private_variable)                                                           ②
```

① Dal modulo `example` vengono importatati tutti gli identificatori pubblici.
② Genera un errore: `NameError: name '_private_variable' is not defined`.

  - Pattern nei match: Nel contesto di un pattern di corrispondenza all'interno di un'istruzione `match`, `_` è una parola chiave contestuale che denota un *wildcard* (carattere jolly), coem indicato sopra.
  - Interprete interattivo: L'interprete interattivo rende disponibile il risultato dell'ultima valutazione nella variabile `_`. Il valore di `_` è memorizzato nel modulo `builtins`, insieme ad altre funzioni e variabili predefinite come `print()`, permettendo l'accesso globale a `_` durante una sessione interattiva. Esempio:

```
# Eseguire nel REPL
result = 5 + 3

print(_)                                                                    ①
```

① Output: `8` (nell'interprete interattivo).

– Altro uso: Altrove, `_` è un identificatore regolare. Viene spesso usato per nominare elementi speciali per l'utente, ma non speciali per Python stesso. Il nome `_` è comunemente usato in congiunzione con l'internazionalizzazione (vedi la documentazione del modulo `gettext` per ulteriori informazioni su questa convenzione) ed è anche comunemente utilizzato per variabili non usate. Esempio:

```
_ = "Valore non usato"                                                      ①

import gettext

gettext.install('myapplication')

print(_('Hello, world'))                                                    ②
```

① Uso di `_` come variabile regolare.
② Uso di `_` per internazionalizzazione.

- `__*__`: Questi nomi, informalmente noti come nomi *dunder*[1], sono definiti dall'interprete e dalla sua implementazione (inclusa la libreria standard). Altri potrebbero essere definiti nelle versioni future di Python. Qualsiasi uso di nomi `__*__`, in qualsiasi contesto, che non segua l'uso esplicitamente documentato, è soggetto a discontinuazione senza preavviso. Esempio:

```
class MyClass:
  def __init__(self, value):                                               ①
    self.__value = value

  def __str__(self):
    return f"MyClass con valore {self.__value}"                            ②

obj = MyClass(10)

print(obj)                                                                 ③
```

① Dunder per metodo chiamato alla creazione dell'oggetto.
② Dunder chiamato da `print` con parametro l'oggetto.
③ Output: `MyClass` con valore `10`.

- `__*`: I nomi in questa categoria, quando utilizzati all'interno di una definizione di classe, vengono riscritti dal compilatore (processo noto come name *mangling*) per evitare conflitti di nome tra attributi "privati" delle classi base e delle classi derivate. Questo aiuta a garantire che gli attributi destinati ad essere privati non vengano accidentalmente sovrascritti nelle sottoclassi. Esempio:

```
class BaseClass:
  def __init__(self):
    self.__private_attr = "Base"
```

[1] I nomi con doppio underscore (`__`) sono chiamati *dunder* come abbreviazione di *double underscore*.

```
class DerivedClass(BaseClass):
  def __init__(self):
    super().__init__()

    self.__private_attr = "Derived"

base_obj = BaseClass()
derived_obj = DerivedClass()

print(base_obj._BaseClass__private_attr)                    ①
print(derived_obj._DerivedClass__private_attr)              ②
```

① Accesso al nome di `BaseClass`.
② Accesso al nome di `DerivedClass`.

9.5.4. Operatori

Gli operatori sono rappresentati da simboli non alfanumerici e, quando applicati a uno o più identificatori, letterali o espressioni (definiti genericamente operandi), producono un valore di risultato. Attenzione a non confondere la definizione di operatore come token, considerata qui, con quella di operatore come funzionalità algoritmica, poiché alcune parole chiave sono operatori algoritmici e anche le funzioni possono agire come operatori.

Esempi:

```
x = 5
y = 10

z = x + y                                                   ①

sum = 3 + 4                                                 ②

result = (x * y) + (z / 2)                                  ③
```

① Utilizza l'operatore `+` sugli identificatori `x` e `y`.
② Utilizza l'operatore `+` su letterali numerici.
③ Utilizza vari operatori su espressioni.

In tabella l'elenco degli operatori:

Tabella 9.2.: Operatori di Python

Operatore	Descrizione
Aritmetici	
`+`	Addizione
`-`	Sottrazione
`*`	Moltiplicazione
`/`	Divisione
`//`	Divisione intera

Operatore	Descrizione
`%`	Modulo
`**`	Esponenziazione
`@`	Matrice (operatore di moltiplicazione)
Di confronto	
`<`	Minore
`>`	Maggiore
`<=`	Minore o uguale
`>=`	Maggiore o uguale
`==`	Uguale
`!=`	Diverso
Sui bit	
`&`	AND bit a bit
`|`	OR bit a bit
`^`	XOR bit a bit
`~`	NOT bit a bit
`<<`	Shift a sinistra
`>>`	Shift a destra
Di assegnazione	
`:=`	Operatore di assegnazione in espressione (walrus o tricheco)

Una caratteristica molto potente del linguaggio è che gli operatori sono utilizzabili anche sui tipi creati dall'utente con logica determinata dalla implementazione. Pertanto, non vanno visti come limitati alle espressioni artimetiche o di assegnazione sui letterali definiti dal linguaggio, ma anche sugli identificatori corrispondenti a tipi più complessi, anche definiti nel programma o importati.

Un esempio è dato dall'operatore `@` che, nel codice seguente, è applicato ad un nuovo tipo di oggetto corrispondente al concetto matematico di matrice. La classe `Matrice` presenta gli attributi per memorizzare gli elementi numerici e anche l'implementazione dell'operazione matematica di moltiplicazione, nel metodo `__matmul__`. Quando l'interprete trova `@`, chiama `__matmul__` dell'operando di sinistra e gli passa l'oggetto corrispondente all'operando di destra.

Codice:

```
class Matrice:
  def __init__(self, righe):
    self.righe = righe
    self.num_righe = len(righe)
    self.num_colonne = len(righe[0]) if righe else 0

  def __matmul__(self, altra):                                          ①
    if self.num_colonne != altra.num_righe:                             ②
      raise ValueError("Non è possibile moltiplicare le matrici: "
                       "dimensioni incompatibili.")

    risultato = [[0 for _ in range(altra.num_colonne)]                  ③
                 for _ in range(self.num_righe)]

    for i in range(self.num_righe):                                     ④
```

```
      for j in range(altra.num_colonne):
        for k in range(self.num_colonne):
          risultato[i][j] += (self.righe[i][k] *
                              altra.righe[k][j])

    return Matrice(risultato)

  def __repr__(self):                                                    (5)
    return '\n'.join([' '.join(map(str, riga)) for riga in self.righe])

A = Matrice([[1, 2],                                                     (6)
             [3, 4]])
B = Matrice([[5, 6],                                                     (7)
             [7, 8]])

C = A @ B                                                                (8)

print("Matrice A:")
print(A)                                                                 (9)

print("Matrice B:")
print(B)

print("Risultato di A @ B:")
print(C)                                                                 (10)
```

① Metodo di implementazione della moltiplicazione di matrici, chiamato da `@` usato come operatore su oggetti di tipo `Matrice`.
② Controlla se le dimensioni sono compatibili per la moltiplicazione.
③ Inizializza la matrice risultato con zeri.
④ Esegue la moltiplicazione delle matrici.
⑤ Chiamato da `print()` quando applicato su oggetti di tipo `Matrice`.
⑥ Rappresentazione leggibile della matrice operando di sinistra.
⑦ Rappresentazione leggibile della matrice operando di destra.
⑧ Moltiplicazione di matrici utilizzando l'operatore `@` che chiama `__matmul__()` per il calcolo effettivo.
⑨ Chiama `__repr__()` di `Matrice` per ottenere la stringa su due righe per `A`: `1 2` e `3 4`.
⑩ Chiama `__repr__()` di `Matrice` per ottenere la stringa su due righe per `C`: `19 22` e `43 50`.

9.5.5. Delimitatori

In Python, alcuni token servono come delimitatori nella grammatica del linguaggio. I delimitatori sono caratteri che separano le varie componenti del codice, come espressioni, blocchi di codice, parametri di funzioni e istruzioni.

La seguente tabella include tutti i delimitatori e i principali utilizzi:

Tabella 9.3.: Delimitatori di Python

Delimitatore	Descrizione
`(`	Utilizzata per raggruppare espressioni, chiamate di funzione e definizioni di tupla
`)`	Utilizzata per chiudere le parentesi tonde aperte
`[`	Utilizzate per definire liste e accedere agli elementi delle liste, tuple, o stringhe
`]`	Utilizzate per chiudere le parentesi quadre aperte
`{`	Utilizzate per definire dizionari e set
`}`	Utilizzate per chiudere le parentesi graffe aperte
`,`	Utilizzata per separare elementi in liste, tuple, e argomenti nelle chiamate di funzione
`:`	Utilizzato per definire blocchi di codice (come in `if`, `for`, `while`, `def`, `class`) e per gli slice
`.`	Utilizzato per accedere agli attributi di un oggetto. Può apparire in letterali decimnali e immaginari
`;`	Utilizzato per separare istruzioni multiple sulla stessa riga
`@`	Utilizzato per dichiarare decoratori per funzioni e metodi
`=`	Operatore utilizzato per assegnare valori a variabili
`->`	Annotazione del tipo di ritorno delle funzioni
`+=`	Assegnazione aumentata con addizione. Aggiunge il valore a destra a quello a sinistra e assegna il risultato alla variabile a sinistra. Come i successivi, è sia un delimitatore che un operatore
`-=`	Assegnazione aumentata con sottrazione
`*=`	Assegnazione aumentata con moltiplicazione
`/=`	Assegnazione aumentata con divisione
`//=`	Assegnazione aumentata con divisione intera
`%=`	Assegnazione aumentata con modulo
`@=`	Assegnazione aumentata con moltiplicazione di matrici
`&=`	Assegnazione aumentata con AND bit a bit
`|=`	Assegnazione aumentata con OR bit a bit
`^=`	Assegnazione aumentata con XOR bit a bit
`>>=`	Assegnazione aumentata con shift a destra
`<<=`	Assegnazione aumentata con shift a sinistra
`**=`	Assegnazione aumentata con esponenziazione

Una sequenza di tre punti, comunemente indicata come ellissi anche al difuori dei linguaggi di programmazione,[2] è trattata come un token a sé e corrisponde ad un oggetto predefinito chiamato Ellipsis, con applicazioni in diversi contesti:

```
print(type(...))                                                    ①

def funzione_da_completare():
  ...                                                               ②
```

[2] L'ellissi è usata, ad esempio, in C per dichiarare funzioni che accettano un numero variaible di parametri e i Javascript come operatore per espandere gli array o le proprietà di un oggetto.

```
class ClasseEsempio:
  def metodo_da_completare(self):
    ...

from typing import Callable

def funzione_variadica(func: Callable[..., int]):                    ③
  pass

import numpy as np

array = np.array([[[1, 2, 3],    [4, 5, 6]],
                  [[7, 8, 9], [10, 11, 12]]])

print(array[..., 1])                                                 ④
```

① Otteniamo il tipo dell'oggetto ellissi. L'output è `<class 'ellipsis'>`.
② Utilizzo come segnaposto per indicare che la funzione è da completare. Da notare che chiamare la funzione `funzione_da_completare()` non dà errore.
③ L'uso di `Callable[..., int]` indica una funzione che può accettare un numero variabile di argomenti di qualsiasi tipo e restituire un valore di tipo int.
④ `numpy` è una libreria di calcolo matriciale molto diffusa. L'ellissi è utilizzata per effettuare uno sezione complessa della matrice secondo tutte le dimensioni precedenti all'ultima. In altre parole, l'ellissi permette di selezionare interamente tutte le dimensioni tranne l'ultima specificata. Il risultato stampato in console è su due righe: `[[ 2  5]` e `[ 8 11]]`.

Alcuni caratteri ASCII hanno un significato speciale come parte di altri token o sono significativi per l'analizzatore lessicale:

Tabella 9.4.: Caratteri speciali di Python

Carattere	Descrizione
'	Utilizzato per definire stringhe di caratteri.
"	Utilizzato per definire stringhe di caratteri.
#	Simbolo di commento. Utilizzato per indicare un commento, che viene ignorato dall'interprete Python.
\	Backslash. Utilizzato per caratteri di escape nelle stringhe e per continuare le righe di codice su più righe fisiche.

Alcuni caratteri ASCII non sono utilizzati in Python e la loro presenza al difuori dei letterali di stringa e dei commenti genera un errore: `$`, `?`, `` ` ``.

9.5.6. Letterali

I letterali sono notazioni per valori costanti di alcuni tipi predefiniti nel linguaggio. Esistono diversi tipi di letterali, ognuno rappresenta un tipo di dato specifico e ha una sintassi particolare.

9.5.6.1. Numerici

I letterali numerici includono interi, numeri a virgola mobile e numeri immaginari:

- Interi, possono essere scritti in base decimale, ottale, esadecimale o binaria:
  - Decimale: `42`.
  - Ottale: `0o12`, `0O7`.
  - Esadecimale: `0xA`, `0X1F`.
  - Binario: `0b1010`, `0B11`.
- Virgola mobile, possono essere rappresentati con una parte intera e una decimale, oppure con notazione scientifica:
  - Virgola mobile: `3.14`, `0.001`.
  - Notazione scientifica: `1e10`, `2.5E-3`.
- Numeri immaginari, ottenuti da un letterale intero o a virgola mobile con un suffisso `j` o `J`: `4j`, `4.j`, `3.14e-10j`.

9.5.6.2. Stringhe

I letterali di stringa possono essere racchiusi tra virgolette singole o doppie. Possono anche essere multi-linea se racchiusi tra triple virgolette singole o doppie:

- Stringhe racchiuse tra virgolette singole o doppie:
  - Singole: `'ciao'`.
  - Doppie: `"mondo"`.
- Stringhe multi-linea racchiuse tra triple virgolette singole o doppie:
  - Triple singole: `'''testo multi-linea'''`.
  - Triple doppie: `"""testo multi-linea"""`.

Le stringhe tra tripli apici possono avere degli a capo e degli apici (non tripli) all'interno.

Esempio:

```
stringa_multilinea = """Questa è una stringa
molto "importante"."""

print(stringa_multilinea)
```

Tutte le stringhe sono codificate in Unicode, con il prefisso `b` la stringa è di tipo byte ed è limitata ai 128 caratteri dell'ASCII. Se si prepone `r`, che sta per *raw* cioè grezzo, allora la codifica è sempre Unicode ma i caratteri di escape[3] non sono intepretati.

Alcuni esempi comuni includono:

- `\n` per una nuova linea (linefeed).

[3] In Python, il carattere di escape `\` è utilizzato nelle stringhe per inserire caratteri speciali che non possono essere facilmente digitati sulla tastiera o che hanno significati speciali.

- `\t` per una tabulazione.
- `\\` per inserire un backslash.
- `\'` per il singolo apice.
- `\"` per i doppi apici.

9.5.6.3. F-stringhe

Le f-stringhe (stringhe formattate) sono racchiuse tra virgolette singole, doppie o triple e sono precedute dal prefisso `f` o `F`. Il *formattato* nel nome deriva dalla possibilità di includere espressioni Python all'interno che sono valutate all'atto della esecuzione della istruzione contenente la stringa. Si possono avere stringhe formattate grezze ma non byte.

Esempio:

```
nome = "Python"

f_stringa = f'Ciao, {nome.upper()}!'                                    ①

definizione = "Linguaggio"

f_stringa_multi_linea = f'''Questo è un esempio
di f-stringa multi-linea
in {definizione.lower() + ' ' + nome}'''                                ②

print(f_stringa)                                                        ③

print(f_stringa_multi_linea)                                            ④
```

① Viene chiamato il metodo della stringa `lower()` per avere il maiuscolo.
② Usiamo un'espressione di concatenazione di stringhe colla prima generata da un metodo che mette in minuscolo la stringa di `definizione`.
③ Output: `Ciao, python!`.
④ Output composto dalle tre righe `Questo è un esempio`, `di f-stringa multi-linea` e `in linguaggio Python`.

10. Il modello dati

Per comprendere il **modello dati** di Python, dobbiamo conoscere i tipi di dati del linguaggio e le relative operazioni, con ciò intendendo che dobbiamo sia elencare quelli predefiniti, che apprendere le modalità di definizione di nuovi tipi.

10.1. Elementi di programmazione orientata agli oggetti

Gli oggetti sono l'astrazione dei dati definita in Python. Ogni oggetto è caratterizzato da un'**identità**, un **tipo** e un **valore**. L'identità di un oggetto non cambia una volta creato e possiamo pensarlo come l'indirizzo dell'oggetto in memoria[1]. Python permette di ricavarlo per mezzo della funzione predefinita (*built-in*) `id()`:

```
s = "Hello"

print(id(s))                                                    ①
```

① Output: un numero come `4467381744`.

Il tipo di un oggetto determina le operazioni che l'oggetto supporta per la manipolazione del proprio stato o di quello di altri oggetti, e definisce anche i possibili valori per gli oggetti di quel tipo (*dominio* dei valori, per prendere in prestito un termine dalla matematica). Il linguaggio fornisce una funzione per conoscere il tipo dell'oggetto:

```
s = "Hello"

print(type(s))                                                  ①
```

① Output: `<class 'str'>`.

10.1.1. Tipi e classi

In Python, i tipi di dati sono implementati come **classi**. Una classe è una definizione che specifica una struttura di dati e il comportamento associato attraverso **attributi** e **metodi**. Gli attributi sono variabili legate alla classe o alle sue istanze e rappresentano lo stato, mentre i metodi sono funzioni che definiscono il comportamento.

Un'**istanza** (o **oggetto**) è un'implementazione concreta della classe. Mentre di una classe esiste una sola definizione, si possono creare infinite istanze della stessa classe, a meno che non siano state implementate limitazioni particolari. Ogni istanza ha il proprio stato (valore degli attributi). Gli attributi e i metodi possono essere membri sia delle classi che delle istanze. In generale, ogni oggetto può avere i propri attributi e metodi, ma può anche accedere agli attributi e ai metodi della classe a cui appartiene.

Gli attributi possono essere di classe o di istanza:

[1] In CPython è effettivamente implementato così.

- Gli **attributi di classe** sono condivisi tra tutte le istanze della classe.
- Gli **attributi di istanza** sono specifici di ogni istanza.

I **metodi di istanza** operano su istanze specifiche della classe e possono accedere e modificare gli attributi dell'istanza stessa. I **metodi di classe**, invece, operano a livello della classe e possono accedere e modificare gli attributi della classe attraverso il parametro `cls`. Infine, i **metodi statici** sono funzioni associate alla classe che non dipendono né dalla classe né dalle sue istanze; non possono modificare lo stato della classe o delle sue istanze.

Ecco un esempio di definizione di una classe in Python, utile, per il momento, solo a mostrare la sintassi dei membri:

```
class Esempio:
  attributo_di_classe = 'Valore di classe'                          ①

  def __init__(self, attributo):
    self.attributo_di_istanza = attributo                           ②

  def metodo_di_istanza(self):                                      ③
    return f"Metodo di istanza: {self.attributo_di_istanza}"

  @classmethod
  def metodo_di_classe(cls):                                        ④
    return f"Metodo di classe: {cls.attributo_di_classe}"

  @staticmethod
  def metodo_statico():                                             ⑤
    return "Metodo statico"

oggetto = Esempio("Valore di istanza")                              ⑥

print(oggetto.metodo_di_istanza())                                  ⑦
print(Esempio.metodo_di_classe())                                   ⑧
print(Esempio.metodo_statico())                                     ⑨
```

① Attributo di classe.
② Attributo di istanza.
③ Metodo di istanza.
④ Metodo di classe, preceduto da `@classmethod`.
⑤ Metodo statico, preceduto da `@staticmethod`.
⑥ Creazione di un oggetto della classe `Esempio` associato alla variabile `oggetto`.
⑦ Output: `Metodo di istanza: Valore di istanza`.
⑧ Output: `Metodo di classe: Valore di classe`.
⑨ Output: `Metodo statico`.

In questo esempio, `Esempio` è una classe con un attributo di classe, un attributo di istanza, un metodo di istanza, un metodo di classe e un metodo statico. La classe definisce la struttura e il comportamento, mentre l'oggetto `oggetto` è un'istanza concreta della classe.

10.1.2. Creazione di oggetti

Una volta definita una classe, si possono creare nuove istanze o oggetti di quella classe, il che rappresenta un modo efficiente per riutilizzare funzionalità nel proprio codice. Questo passaggio è spesso chiamato costruzione o **istanziazione** degli oggetti. Lo strumento responsabile per la specializzazione di questo processo, è comunemente noto come **costruttore** della classe.

In Python, il costruttore è il metodo speciale `__init__` e può avere un numero arbitrario di parametri. Questo metodo viene eseguito automaticamente dall'interprete quando un nuovo oggetto della classe viene creato ed è il luogo possiamo scrivere delle istruzioni utili all'inizializzazione dell'oggetto, anche usando gli argomenti passati. La sintassi per l'istanziazione è data dall'identificatore della classe, seguita da una coppia di parentesi tonde, con all'interno l'elenco degli argomenti.

Esempio:

```
s = str(4)                                                    ①
```

① L'identificatore della classe delle stringhe `str` è seguito da un unico argomento, `4` , dato da un letterale intero, passato al costruttore per l'inizializzazione.

Il metodo `__init__` è sempre presente: se non è definito esplicitamente nella classe, viene ereditato dalla classe base `object`.

La creazione di un oggetto avviene in diversi contesti:

- Uso di letterali. Ad esempio, per creare un oggetto stringa, si usano i letterali stringa e analogamente per gli altri:

  ```
  s = "Hello"                                                 ①

  n = 42                                                      ②

  f = 3.14                                                    ③
  ```

  ① Creazione di un oggetto di tipo `str`.
  ② Creazione di un oggetto di tipo `int`.
  ③ Creazione di un oggetto di tipo `float`.

- Chiamata del costruttore della classe, ad esempio a quello della classe `list` passiamo una sequenza di interi:

  ```
  lista = list((1, 2, 3))
  ```

- Utilizzo di funzioni e metodi che restituiscono oggetti. Ad esempio, il metodo `upper()` della classe `str` restituisce un nuovo oggetto di tipo `str`, ottenuto a partire da quello su cui è eseguito:

  ```
  s1 = "Hello, World!"

  su_1 = s1.upper()                                           ①

  su_2 = "Hello, World!".upper()                              ②
  ```

  ① `upper()` crea un secondo oggetto stringa che viene associato alla variabile `su_1`.
  ② Sintassi alternativa che ha il medesimo effetto.

- Clonazione o copia utilizzando la funzione `copy` del modulo `copy`:

```
import copy

lista_originale = [1, 2, 3]                    ①

lista_copiata = copy.copy(lista_originale)     ②
```

① `lista_originale` identifica un oggetto lista.
② `copy()` produce una copia di `lista_originale` identificata da `lista_copiata`.

10.1.3. Accesso a attributi e metodi

In Python, si utilizza la notazione con il punto `.` per accedere agli attributi e ai metodi di un oggetto o di una classe, semplicemente accodando all'identificatore della variabile (che rappresenta un'istanza della classe) o a quello della classe, il punto e l'identificatore del membro (attributo o metodo).

Esempio su `str`:

```
s = "hello world"                              ①

s_upper = s.upper()                            ②

print(s_upper)                                 ③

print(s.__class__)                             ④
```

① Creazione di un oggetto stringa.
② Chiamata del metodo `upper()` usando l'identificatore `s`.
③ Output: `HELLO WORLD`.
④ Output dell'attributo `__class__`: `<class 'str'>`.

10.1.4. Gerarchie di classi

Le classi sono organizzate in una gerarchia. In cima alla gerarchia delle classi di Python c'è la classe `object`, con un nome po' infelice, da cui derivano tutte le altre classi. Questo significa che ogni classe in Python eredita gli attributi e i metodi della classe `object`.

La funzione `isinstance()` è utile per investigare questa gerarchia, poiché permette di verificare se un oggetto è un'istanza di una determinata classe o di una sua classe derivata. `isinstance()`, pertanto, ha due parametri, l'identificatore dell'oggetto e l'identificatore della classe e restituisce un valore logico, `True` o `False`.

Ad esempio:

```
s = "Hello"

print(isinstance(s, str))                      ①

print(isinstance(s, object))                   ②
```

① Output: `True`. `isinstance` conferma che `s` è un'istanza della classe `str`.

② Output: `True`. `isinstance` conferma che `s` è un'istanza anche della classe `object`, da cui deriva `str`.

La funzione `issubclass()` è altrettanto utile per esplorare le gerarchie delle classi, in quanto permette di verificare se una classe è una sottoclasse di un'altra. Ha due parametri, cioè il nome della classe derivata e quella base e restituisce un valore logico.

Ad esempio:

```
print(issubclass(str, object))                                    ①
```

① Output: `True`. `issubclass` conferma che `str` è una sottoclasse di `object`.

10.1.5. Mutabilità e immutabilità

Gli oggetti il cui valore può cambiare sono detti **mutabili**, mentre gli oggetti il cui valore non lo può, una volta creati, sono chiamati **immutabili**. L'immutabilità di un oggetto è determinata dalla progettazione del suo tipo. Ad esempio, per i tipi definiti nel linguaggio, numeri interi e in virgola mobile, stringhe e tuple sono immutabili, mentre dizionari e liste sono mutabili.

Alcuni oggetti, chiamati **contenitori**, contengono riferimenti ad altri oggetti. Esempi di contenitori sono *tuple*, *liste* e *dizionari*, strutture dati di base definite in molti linguaggi di programmazione. I riferimenti ad altri oggetti contribuiscono al valore di un contenitore e, nella maggior parte dei casi, quando parliamo del valore di un contenitore, intendiamo proprio i valori degli oggetti contenuti. Ad esempio:

```
t = (1, 2, 3)                                                     ①

l = ["Qui", "Quo", "Qua"]                                         ②
```

① Tupla con tre oggetti al suo interno: `1`, `2`, `3`. Diciamo che la tupla contiene i tre valori interi.
② Lista con tre oggetti all'interno: `"Qui"`, `"Quo"`, `"Qua"`. Diciamo che la lista contiene le tre stringhe.

La mutabilità di un contenitore si riferisce alla identità degli oggetti referenziati e non al loro valore, quindi, ad esempio, se una tupla contiene un riferimento ad un oggetto mutabile, il valore della tupla cambia se quell'oggetto mutabile viene modificato:

```
s = ([1, 2, 3], ["Qui", "Quo", "Qua"])

print(s)                                                          ①

print(id(s[0]))                                                   ②
print(id(s[1]))                                                   ③

s[1][0] = "Huey"                                                  ④
s[1][1] = "Dewey"
s[1][2] = "Louie"

print(s)                                                          ⑤

print(id(s[0]))                                                   ⑥
print(id(s[1]))                                                   ⑦
```

① Output: `([1, 2, 3], ['Qui', 'Quo', 'Qua'])`.
② Modifico gli elementi della lista, traducendoli in inglese.
③ Output dell'identità del primo oggetto contenuto (su ogni computer e sessione sarà diverso): `4361472384`.
④ Output dell'identità del secondo oggetto contenuto: `4361474176`.
⑤ Output: `([1, 2, 3], ['Huey', 'Dewey', 'Louie'])`. La tupla è cambiata!
⑥ Output: `4361472384`, l'identità del primo oggetto contenuto non è cambiata!
⑦ Output: `4361474176`, l'identità del secondo oggetto contenuto non è cambiata!

Ciò è un po' più generale, perché per gli oggetti di tipo immutabile, quando sono assegnati ad una variaible o risultato di un'espressione, a parità di valore, potrebbero avere la stessa identità, cioè essere lo stesso oggetto. Il condizionale è dovuto alla presenza e applicazione logiche di ottimizzazione, ad esempio della memoria, implementate nell'interprete.

Ad esempio, dopo:

```
s1 = "Qui, Quo, Qua"

print(id(s1))                                                    ①

s2 = "Qui, Quo, Qua"

print(id(s2))                                                    ②
```

① Output dell'identità di `s1`: `4432491760`.
② Output dell'identità di `s2`: `4432491760`, cioè è la medesima identità dell'oggetto il cui nome è `s1`.

Per gli oggetti mutabili, questo non accade anche in casi su cui potrebbe essere ottimizzante:

```
l1 = []

print(id(l1))                                                    ①

l2 = []

print(id(l2))                                                    ②
```

① Output dell'identità di `l1`: `4454287744`.
② Output dell'identità di `l2`: `4454289536`, cioè le identità sono diverse.

Attenzione però ad alcune *scorciatoie* di Python:

```
c = d = []
```

è equivalente a:

```
c = []
d = c
```

cioè l'oggetto lista ha due nomi nel programma.

10.1.6. Tipi hashable

I tipi di dati possono essere classificati come **hashable** o non hashable. Un tipo è considerato hashable se gli oggetti creati da quel tipo hanno un valore di hash[2] che rimane costante per tutta la durata della loro vita e possono essere confrontati con altri oggetti. Questo è garantito per i tipi immutabili, come numeri, stringhe e tuple.

Una funzione di hash prende un valore (come una stringa o un numero) e restituisce un numero di lunghezza fissa, chiamato **hash**. Questo hash viene utilizzato come indice per memorizzare il valore nella tabella di hash. Quando si desidera cercare un elemento, la funzione di hash calcola l'indice corrispondente, consentendo un accesso diretto all'elemento. Questo processo rende la ricerca, l'inserimento e la cancellazione degli elementi molto efficienti, con un tempo di accesso medio costante (O(1)).

Gli oggetti di tipo hashable possiedono le seguenti caratteristiche principali:

1. Valore di hash costante: Gli oggetti devono avere un valore di hash che non cambia durante la loro vita. Questo è tipico degli oggetti immutabili.
2. Comparabilità: Gli oggetti devono poter essere confrontati con altri oggetti per determinare se sono uguali. Due oggetti che sono considerati uguali devono avere lo stesso valore di hash.

La proprietà di essere hashable è strettamente legata all'immutabilità. Gli oggetti immutabili sono naturalmente hashable perché il loro stato non cambia dopo la creazione, garantendo che il valore di hash rimanga costante. Ad esempio, tipi di dati immutabili come numeri, stringhe e tuple possono essere considerati hashable. Il viceversa non è vero, cioè esistono alcuni tipi mutabili che sono hashable.

10.1.7. Eliminazione

Gli oggetti non vengono mai eliminati esplicitamente dall'utente, ma, quando diventano inaccessibili, possono essere raccolti dal garbage collector, che è eseguito contemporaneamente al codice del programma, come parte dell'interprete. L'implementazione specifica quando e come la memoria debba essere liberata.

Alcuni oggetti contengono riferimenti a risorse *esterne* rispetto al programma, come file aperti, connessioni di rete, finestre (*graphical user interface*, GUI) o dispositivi hardware. Queste risorse generalmente non vengono liberate dal garbage collector, perché le azioni corrispondenti sono particolari della risorsa stessa. Pertanto, i loro oggetti forniscono anche un metodo esplicito di rilascio, solitamente chiamato `close()`. È molto importante tener conto di ciò per evitare effetti indesiderati.

10.2. Tipi predefiniti

Python mette a disposizione molti tipi *generali*, cioè utili alla gestione di dati comunemente presenti in algoritmi. Esistono, inoltre, sia meccanismi di estensione della distribuzione Python che modificano l'installazione sul proprio computer, sia la possibilità conferita al programmatore di creare i propri tipi o di importarli da librerie prodotte da terze parti.

I tipi predefiniti sono immediatamente disponibili, cioè non necessitano di alcuna azione da parte del programmatore, come, ad esempio, l'importazione di moduli.

[2] In informatica, le tabelle hash (hash table) sono una struttura dati essenziale per l'efficienza e la velocità di accesso ai dati. Le tabelle hash sono costituite da coppie indice-valore e sono progettate per sostituire la ricerca sequenziale in array, che può essere lenta soprattutto per grandi quantità di dati.

Per ragioni di esposizione, distingueremo tra tipi predefiniti fondamentali, cioè che possono essere introdotti con un insieme limitato di concetti, tipi legati alla astrazione del paradigma di orientamento agli oggetti e, infine, costruzioni del linguaggio legate all'esecuzione asincrona.

10.2.1. Tipi predefiniti fondamentali

Prima di entrare nel dettaglio per ognuno, di seguito un elenco dei tipi predefiniti fondamentali:

- Numeri: Sia interi relativi che reali, cioè in virgola mobile, sia complessi.
- Sequenze: Contenitori caratterizzati da un ordinamento degli oggetti al loro interno, sia mutabili che immutabili.
- Insiemi: Contenitori non caratterizzati da un ordinamento degli oggetti.
- Dizionari: Contenitori di coppie di oggetti, rispettivamente, chiave e valore.
- Altri: `None`, `NotImplemented`, `Ellipsis`.

10.2.2. Numeri

I tipi predefiniti numerici corrispondono agli interi, i numeri a virgola mobile e quelli complessi. Sono immutabili, il che significa che effettuando una operazione su un oggetto numerico si produrrà sempre un nuovo oggetto e non una modifica del precedente.

Esistono anche altri tipi numerici nelle librerie fornite coll'interprete, per i numeri decimali a precisione arbitraria e i numeri razionali come frazioni.

I letterali numerici non hanno segno, cioè `+` e `-` sono operatori unari che precedono la rappresentazione di un numero, quindi `-3` è una espressione che diventa il valore di un oggetto intero di valore `-3`. Nei letterali numerici si può inserire un trattino basso (*underscore*) tra le cifre o dopo gli specificatori della base per gli interi.

10.2.2.1. Interi

Esistono due tipi di interi. Il primo è `int`, per gli usuali numeri interi, a cui corrispondono diverse rappresentazioni sintattiche in basi diverse, come i letterali decimali, binari, ottali o esadecimali:

- Un letterale decimale inizia sempre con una cifra diversa da zero, esempi: `10`, `1_0`.
- Quello binario inizia con `0b` o `0B` seguito da una sequenza di cifre binarie (0 o 1), esempi: `0b1010`, `0B1010`, `0b_10_10`, `0b1_0_10`.
- L'ottale inizia con `0o` o `0O` seguito da una sequenza di cifre ottali (da 0 a 7), esempi: `0o12`, `0O12`.
- L'esadecimale inizia con `0x` o `0X` seguito da una sequenza di cifre esadecimali (da 0 a 9 e da A a F, in maiuscolo o minuscolo), esempi: `0xA`, `0XA`.

Gli interi in Python sono illimitati, al netto della finitezza della memoria del computer disponibile per la loro rappresentazione.

```
large_int = 10**100

print(f"Un intero molto grande: {large_int}")                                    ①
```

① Output: `100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000`

L'altro tipo è `bool`, che ha solo due oggetti possibili, che si comportano come quelli corrispondenti agli interi `1` e `0` per rappresentare i valori logici, rispettivamente, di vero e falso. `bool` come classe deriva da `int` che, a sua volta, deriva da `object`:

```python
print(issubclass(bool, int))        ①
print(issubclass(int, object))      ②
```

① Output: `True`.
② Output: `True`.

10.2.2.2. Numeri in virgola mobile

I numeri in virgola mobile in Python sono istanze della classe `float` e sono utilizzati per rappresentare numeri reali con una parte decimale. I letterali corrispondenti sono una sequenza di cifre in base 10, e solo in base 10, che include un punto (`.`), un suffisso di notazione scientifica (`e` o `E`, opzionalmente seguito da `+` o `-` e da una o più cifre), o entrambi. Il carattere iniziale di un letterale in virgola mobile non può essere `e` o `E`, ma può essere qualsiasi cifra (anche zeri a differenza degli interi) o un punto (`.`) seguito da una cifra.

Esempi di literali in virgola mobile: `0.`, `0.1`, `.0`, `1.`, `1.0`, `1e0`, `1.e0`, `1.0E0`, `1_0_0.0_1`. Attenzione: `1` è un intero, `1.` non lo è.

Per conoscere l'intervallo esatto e la precisione dei valori in virgola mobile sulla piattaforma in uso, e molti altri dettagli, è possibile usare `sys.float_info` nel modo seguente:

```python
import sys

float_info = sys.float_info        ①

print(f"Massimo valore rappresentabile (max): {float_info.max}")        ②
print(f"Minimo valore rappresentabile positivo (min): {float_info.min}")        ③
print(f"Massimo esponente base 2 (max_exp): {float_info.max_exp}")        ④
print(f"Minimo esponente base 2 (min_exp): {float_info.min_exp}")        ⑤
print(f"Massimo esponente base 10 (max_10_exp): {float_info.max_10_exp}")        ⑥
print(f"Minimo esponente base 10 (min_10_exp): {float_info.min_10_exp}")        ⑦
print(f"Precisione in bit (dig): {float_info.dig}")        ⑧
print(f"Numero di bit di mantissa (mant_dig): {float_info.mant_dig}")        ⑨
print(f"Epsilon macchina (epsilon): {float_info.epsilon}")        ⑩
```

① Oggetto con all'interno le informazioni relative alla gestione dei numeri in virgola mobile della macchina.
② Output: `Massimo valore rappresentabile (max): 1.7976931348623157e+308`. Il massimo valore rappresentabile per un numero in virgola mobile.
③ Output: `Minimo valore rappresentabile positivo (min): 2.2250738585072014e-308`. Il minimo valore positivo rappresentabile per un numero in virgola mobile.
④ Output: `Massimo esponente base 2 (max_exp): 1024`. Il massimo esponente base 2.
⑤ Output: `Minimo esponente base 2 (min_exp): -1021`. Il minimo esponente base 2.
⑥ Output: `Massimo esponente base 10 (max_10_exp): 308`. Il massimo esponente base 10.
⑦ Output: `Minimo esponente base 10 (min_10_exp): -307`. Il minimo esponente base 10.

⑧ Output: `Precisione in bit (dig): 15`. La precisione in bit, cioè il numero di cifre decimali che possono essere rappresentate senza perdita di precisione.
⑨ Output: `Numero di bit di mantissa (mant_dig): 53`. Il numero di bit nella mantissa.
⑩ Output: `Epsilon macchina (epsilon): 2.220446049250313e-16`. La differenza tra 1 e il numero più piccolo maggiore di 1 che può essere rappresentato.

10.2.2.3. Numeri complessi

Un numero complesso è composto da due valori in virgola mobile, uno per la parte reale e uno per la parte immaginaria. In Python, i numeri complessi sono istanze della classe `complex`, che presenta due attributi di sola lettura `real` e `imag`, rispettivamente per la parte reale e immaginaria, di tipo `float`.

Un letterale immaginario può essere specificato come qualsiasi letterale decimale in virgola mobile o intero seguito da una `j` o `J`: `0j`, `0.j`, `0.0j`, `.0j`, `1j`, `1.j`, `1.0j`, `1e0j`, `1.e0j`, `1.0e0j`. La `j` alla fine del letterale immaginario indica la radice quadrata di `-1`. Per denotare un qualsiasi numero complesso costante, si potrà sommare o sottrarre un letterale in virgola mobile (o letterale intero) e uno immaginario.

```
z = 42+3.14j                    ①

print(z)                        ②

print(z.real)                   ③
print(z.imag)                   ④

print(type(z.real))             ⑤
print(type(z.imag))             ⑥
print(type(z))                  ⑦
```

① Assegnamento di un numero complesso alla variabile `z`.
② Output: `(42+3.14j)`.
③ Output: `42.0`.
④ Output: `3.14`.
⑤ Output: `<class 'float'>`.
⑥ Output: `<class 'float'>`.
⑦ Output: `<class 'complex'>`.

10.2.3. Sequenze

Una sequenza è un contenitore ordinato di oggetti, il cui indice è un intero che parte da `0`. Se la sequenza è referenziata da una variabile `c`, per ottenere il numero totale di oggetti contenuti si userà la funzione predefinita `len()`, cioè `len(c)`, e per l'oggetto i-esimo le parentesi quadre e l'indice, `c[i]`. È possibile usare indici negativi che sono interpretati come la somma di tale indice col numero totale di oggetti contenuti, quindi `-i` è trattato come `len(c)-i` e l'indice risultante dovrà sempre essere compreso tra `0` e `len(c)-1`.

Si può *sezionare* (*slicing*) la sequenza per ottenere una nuova sottosequenza, dello stesso tipo, di oggetti originariamente contigui tra gli indici `i` e `j`, cioè con indici `x` tali che `i x<j`, usando `c[i:j]`. Aggiungendo un terzo parametro di *passo*, `k`, si possono selezionare solo gli oggetti con indici `i+l*k` e `l 0`, che siano compresi tra i due indici, cioè `i i+l*k<j`, con `c[i:j:k]`.

Le sequenze sono categorizzate in base alla mutabilità:

- Immutabili: stringhe, tuple, bytes.
- Mutabili: liste e array di bytes.

10.2.3.1. Iterabilità

Le sequenze sono iterabili, cioè possono essere ottenuti tutti gli oggetti contenuti nell'ordine corretto, per mezzo di un oggetto ad hoc detto **iteratore**. Un iteratore è un oggetto separato dalla sequenza stessa, che mantiene uno stato interno per tenere traccia dell'elemento successivo da restituire. Questo permette di iterare sulla sequenza senza modificarla direttamente. L'iteratore prende in input un oggetto iterabile e ne restituisce i valori uno alla volta secondo un certo protocollo. Per le sequenze, questo protocollo consiste nel partire dal primo elemento e procedere fino all'ultimo.

10.2.3.2. Stringhe

Una stringa in Python è un oggetto che si può creare a partire da un letterale composto di un numero non negativo di caratteri Unicode racchiusi o tra apici singoli `'`, oppure doppi `"`. Per inserire un a capo nella stringa dovrà essere usato `\n`, mentre per spezzarla su due righe fisiche dovrà essere usato un singolo backslash alla fine della prima riga fisica per indicare la continuazione nella riga fisica successiva.

Esempi:

```
s1 = "Hello " \
    "World!"                                                    ①

print(s1)                                                       ②

s2 = "Hello \n" \
    "World!"                                                    ③

print(s2)                                                       ④
```

① Stringa su due righe fisiche ma senza a capo all'interno.
② Output: `Hello World!`.
③ Stringa su due righe fisiche con un a capo all'interno.
④ Output: `Hello World!`.

Alternativamente, si possono usare letterali con coppie di tripli apici singoli `'''` o doppi `"""`, dove la differenza è che è possibile inserire un a capo nell'editor e sarà mantenuto nella stringa. Un singolo backslash non può essere presente.

```
s1 = """Hello
World!"""                                                       ①

print(s1)                                                       ②

s2 = """Hello \n
        World!"""                                               ③

print(s2)                                                       ④
```

① Stringa su due righe fisiche ma senza a capo all'interno.
② Output: `Hello World!`.
③ Stringa su due righe fisiche con un a capo all'interno.
④ Output: `Hello`

`World!`.

Nei letterali stringa si possono inserire caratteri non stampabili o caratteri che non sono disponibili a tastiera, usando le cosiddette sequenze di escape, come da tabella seguente:

Tabella 10.1.: Sequenze di escape

Sequenza	Significato	Codice ASCII/ISO	Esempio di stringa Python
`\<newline>`	Ignora fine linea	-	`"Questo è un testo\ con una linea continuata"`
`\\`	Backslash	`0x5c`	`"C:\\percorso\\al\\file"`
`\'`	Apice singolo	`0x27`	`"L'apice singolo: \' esempio"`
`\"`	Apice doppio	`0x22`	`"L'apice doppio: \" esempio"`
`\a`	Campanello	`0x07`	`"Suono del campanello\a"`
`\b`	Backspace	`0x08`	`"Carattere di backspace\b"`
`\f`	Form feed	`0x0c`	`"Form feed\f esempio"`
`\n`	Nuova linea	`0x0a`	`"Nuova linea\n esempio"`
`\r`	Ritorno carrello	`0x0d`	`"Ritorno carrello\r esempio"`
`\t`	Tabulazione	`0x09`	`"Tabulazione\tesempio"`
`\v`	Tabulazione verticale	`0x0b`	`"Tabulazione verticale\v esempio"`
`\DDD`	Valore ottale DDD del codice Unicode del carattere (solo per caratteri ASCII)	`DDD` (in ottale)	`"Valore ottale: \101 esempio"` (`\101` rappresenta `'A'` che in ASCII è `65`)
`\xXX`	Valore esadecimale XX del codice Unicode del carattere	`XX` (in esadecimale)	`"Valore esadecimale: \x41 esempio"` (`\x41` rappresenta 'A')
`\uXXXX`	Carattere Unicode con valore esadecimale a 4 cifre	`XXXX` (in esadecimale)	`"Carattere cinese: \u4e2d"` (`\u4e2d` rappresenta ' ')
`\UXXXXXXXX`	Carattere Unicode con valore esadecimale a 8 cifre	`XXXXXXXX` (in esadecinale)	`"Carattere: \U0001f600"` (`\U0001f600` rappresenta ' ')

Sequenza	Significato	Codice ASCII/ISO	Esempio di stringa Python
`\N{name}`	Carattere Unicode	-	`"Carattere Unicode: \N{LATIN CAPITAL LETTER A} esempio"`

Esistono anche i letterali di stringhe *grezze* (*raw*), sintatticamente identiche alle altre a meno di un suffisso `r` o `R`, comportante che le sequenze di escape non siano interpretate. Si usano comunemente per esprimere pattern di espressioni regolari o percorsi di file in Windows.

```
stringa = "C:\\Users\\username\\Documents\\file.txt"  ①

print(stringa)  ②

stringa_raw = r"C:\Users\username\Documents\file.txt"  ③

print(stringa_raw)  ④
```

① Perché un path sia corretto in Windows è necessario usare il backslash per escape del backslash come separatore di path.
② Output: `C:\Users\username\Documents\file.txt`.
③ Definendo la stringa come grezza allora il backslash è intepretato come tale, quindi separatore di path.
④ Output: `C:\Users\username\Documents\file.txt`.

Dopo aver identificato il letterale stringa, l'inteprete crea l'oggetto stringa in memoria con tipo `str`. Alternativamente, si può crearla in altri modi:

- Sebbene non sia comune usare il costruttore per creare una stringa da un letterale stringa, è comunque possibile:

  ```
  s = str("Hello, World!")  ①

  print(s)  ②
  ```

  ① Passo `"Hello, World!"` al costruttore di `str`.
  ② Output: `Hello, World!`.

- Possiamo creare una stringa a partire da un numero (intero o in virgola mobile):

  ```
  s = str(42)

  print(s)  ①

  s = str(3.14)

  print(s)  ②
  ```

  ① Output: `3.14`.
  ② Output: `42`.

- Si può creare una stringa concatenando i caratteri di una lista:

```
cl = ['H', 'e', 'l', 'l', 'o']

s = ''.join(cl)

print(s)                                                                    (1)
```

(1) Output: `Hello`.

- Se un oggetto definisce il metodo speciale `__str__()` che restituisce una stringa, il costruttore di `str` lo chiama per ottenerla:

```
l = ["Hello", " ", "World!"]

s = str(l)

print(s)                                                                    (1)
```

(1) Output: `['Hello', ' ', 'World!']`.

- Si può creare una nuova stringa da qualsiasi oggetto iterabile, come liste o tuple, utilizzando il metodo `join` che inserisce tra gli elementi della sequenza la stringa dell'oggetto di cui è membro:

```
lista = ['Python', 'è', 'fantastico']

s = ' '.join(lista)

print(s)                                                                    (1)

tupla = ('Hello', 'World!')

s = ' '.join(tupla)

print(s)                                                                    (2)
```

(1) Output: `Python è fantastico`. Lo spazio tra le parole è stato inserito perché `join` è stato chiamato su un oggetto la cui stringa era data dal solo carattere di spazio.
(2) Output: `Hello World!`.

10.2.3.3. Oggetti `bytes`

Un oggetto `bytes` è un array immutabile i cui elementi sono byte a 8 bit, rappresentati da interi nel range da 0 a 255. Gli oggetti `bytes` sono utili per gestire dati binari, come quelli letti o scritti su file o trasmessi su reti.

Esistono diversi modi per creare oggetti `bytes`:

- È possibile creare oggetti `bytes` utilizzando letterali bytes, che sono simili ai letterali stringa, ma sono preceduti dal prefisso `b`. Ad esempio:

```
b'abc'                                                                      (1)
```

(1) Crea un oggetto bytes con i byte corrispondenti ai caratteri ASCII `'a'`, `'b'`, `'c'`.

- È possibile creare oggetti `bytes` utilizzando il costruttore `bytes()`, che può accettare diversi tipi di argomenti:
  - Senza argomenti, crea un oggetto `bytes` vuoto:

    ```
    bytes()  ①
    ```

    ① Output: `b''`.
  - Da un oggetto iterabile contenente interi (ognuno dei quali deve essere nel range da 0 a 255):

    ```
    bytes([97, 98, 99])  ①
    ```

    ① Output: `b'abc'`.
  - Da una stringa e una codifica:

    ```
    bytes('abc', 'utf-8')  # Output: b'abc'
    ```

    1. Output: `b'abc'`.
- È possibile creare una stringa a partire da un oggetto `bytes` specificando la codifica:

```
b = b'Hello, World!'  ①

s = str(b, 'ASCII')  ②

print(s)  ③
```

① Oggetto di tipo `bytes`.
② Creazione di una stringa da oggetto `bytes` che è codificato in ASCII.
③ Output: `Hello World!`.

Alcuni esempi di utilizzo degli oggetti `bytes`:

- Creazione e accesso:

```
b = b'hello'

print(b[0])  ①

print(b[1:3])  ②
```

① Output: `104` (ASCII per 'h').
② Output: `b'el'`.

- Concatenazione e ripetizione:

```
b1 = b'hello'
b2 = b'world'

b3 = b1 + b2

print(b3)  ①

b4 = b1 * 3
```

```
print(b4)                                                                    ②
```

① Output: `b'helloworld'`.
② Output: `b'hellohellohello'`.

- Conversione da e verso stringhe:

```
s = "hello"
b = s.encode('utf-8')                                                        ①
print(b)                                                                     ②

s2 = b.decode('utf-8')                                                       ③
print(s2)                                                                    ④
```

① Converte la stringa in `bytes` usando la codifica UTF-8.
② Output: `b'hello'`.
③ Decodifica i `bytes` in una stringa.
④ Output: `'hello'`.

10.2.3.4. Tuple

Le tuple in Python sono sequenze ordinate immutabili, in cui gli oggetti contenuti possono essere di tipi diversi. La classe è `tuple` che deriva da `object`.

Per creare una tupla, si può utilizzare una serie di espressioni separate da virgole (`,`), come elementi della tupla. Si può opzionalmente mettere una virgola ridondante dopo l'ultimo elemento, che è necessaria se si ha un solo elemento. Si possono raggruppare gli elementi della tupla tra parentesi, ma le parentesi sono necessarie solo quando le virgole avrebbero altrimenti un altro significato (ad esempio, nelle chiamate di funzione), o per denotare tuple vuote o annidate.

Esempi di tuple costruite con letterali e espressioni:

```
t1 = 42., "Hello", 0x42                                                      ①
t2 = (42,)                                                                   ②
t3 = 21 + 21,                                                                ③
t4 = ()                                                                      ④
```

① Tupla con tre oggetti contenuti da letterale numerico, letterale stringa e ancora letterale numerico.
② Tupla con un solo oggetto da letterale numerico.
③ Tupla con un solo oggetto da espressione.
④ Tupla vuota.

Possiamo creare le tuple anche usando il costruttore di `tuple`, che accetta come argomento un iterabile:

```
t1 = tuple()                                                                 ①

t2 = tuple("Hello")                                                          ②

t3 = tuple([1, 2, 3])                                                        ③
```

① Crea una tupla vuota.
② Crea una tupla con 5 stringhe, una per ogni carattere.
③ Crea una tupla con una lista di 3 interi.

Esempi di operazioni sulle tuple:

- Accesso e numero di elementi:

```
t1 = 42., "Hello", 0x42

print(t1[1])                                                    ①

print(t1[1:3])                                                  ②

print(42. in t1)                                                ③

print(len(t1))                                                  ④

print(t1[2:6])                                                  ⑤

print(t1[4:6])                                                  ⑥
```

① Output: `"Hello"`.
② Output: `3`.
③ Output: `('Hello', 66)`. Slicing di tupla che produce una nuova tupla con 2 oggetti.
④ Test di appartenenza di `42.` nella tupla `(42.0, 'Hello', 66, 42)`. Output: `True`.
⑤ Slicing con indice destro oltre la lunghezza della lista. Output: `(66,)`.
⑥ Slicing con entrambi gli indici oltre la lunghezza della lista. Output: `()`.

- Concatenazione:

```
t1 = 42., "Hello", 0x42
t2 = (42,)

t3 = t1 + t2

print(t3)                                                       ①
```

① Concatenazione di tuple che produce una nuova tupla con 4 oggetti contenuti. Output: `(42.0, 'Hello', 66, 42)`.

10.2.4. Liste

Le liste in Python sono sequenze ordinate mutabili, in cui gli oggetti contenuti possono essere di tipi diversi. La classe è `list` che deriva da `object`.

Per creare una lista, si utilizza una serie di espressioni, separate da virgole (`,`), all'interno di parentesi quadre, per indicare gli elementi della lista. Si può opzionalmente mettere una virgola ridondante dopo l'ultimo elemento. Per denotare una lista vuota, si utilizza una coppia di parentesi quadre vuote.

Esempi di liste costruite con letterali e espressioni:

```
l1 = [42., "Hello", 0x42]                                    ①

l2 = [21 + 21]                                               ②

l3 = []                                                      ③
```

① Lista con tre oggetti: un letterale numerico, un letterale stringa e un altro letterale numerico.
② Lista con un solo oggetto creato da un'espressione.
③ Lista vuota.

Possiamo creare le liste usando il costruttore della classe `list`, passando come argomento un iterabile o nulla:

```
l1 = list()                                                  ①

l2 = list("Hello")                                           ②
```

① Crea una lista vuota.
② Crea una lista con 5 stringhe, una per ogni carattere della stringa "Hello".

Le operazioni comuni sulle liste:

- Accesso e modifica:

```
l1 = [42., "Hello", 0x42]

print(l1[1])                                                 ①

print(len(l1))                                               ②

l1[1] = "Ciao"

print(l1)                                                    ③
```

① Output: `"Hello"`. Accesso all'elemento in posizione 1 della lista.
② Output: `3`. Numero totale di elementi nella lista `l1`.
③ Modifica della lista con l'assegnazione di un nuovo valore a un indice. Output: `[42.0, 'Ciao', 66]`.

- Concatenazione:

```
l1 = [42., "Hello", 0x42]
l2 = [42]
l3 = l1 + l2

print(l3)                                                    ①

l2 += l1
print(l2)                                                    ②
```

① Concatenazione di liste che produce una nuova lista. Output: `[42.0, 'Hello', 66, 42]`.
② Concatenazione sul posto modificando `l2`. Output: `[42, 42.0, 'Hello', 66]`.

- Estensione e aggiunta di elementi:

```
l1 = [42., "Hello", 0x42]
l4 = []

l4.extend(l1)

print(l4)                                                    ①

l5 = []

l5.append(l1)

print(l5)                                                    ②
```

① Estende `l4` aggiungendo gli elementi di `l1`. Output: `[42.0, 'Hello', 66]`.
② Aggiunge `l1` come un singolo elemento alla fine di `l5`. Output: `[[42.0, 'Hello', 66]]`.

- Slicing, rimozione e riassegnazione di elementi:

```
l1 = [42., "Hello", 0x2A, "Hello", 0o52, "Hello", 42]
l2 = [42]

print(l1[1:3])                                               ①

l1.remove("Hello")
print(l1)                                                    ②

l1.pop(0)
print(l1)                                                    ③

del l1[0:2]
print(l1)                                                    ④

l1[1:3] = ["Ciao", 24]
print(l1)                                                    ⑤

l1[1:3] = []
print(l1)                                                    ⑥
```

① Slicing di lista che produce una nuova lista. Output: `['Hello', 66]`.
② Rimozione della prima occorrenza di elemento dalla lista usando `remove()` che, scorrendo la sequenza, applica un test di uguaglianza per stabilire che l'oggetto nella lista sia uguale a quello passato come argomento e, quindi, lo elimina. `remove()` non restituisce l'elemento eliminato. Output: `[42.0, 42, 'Hello', 42, 'Hello', 42]`.
③ Rimozione di un elemento della lista usando `pop()` per eliminare l'elemento in posizione indicata dall'indice passato come argomento, qui `0`. `pop()` restituisce l'elemento eliminato. Output: `[42, 'Hello', 42, 'Hello', 42]`.
④ Rimozione di elementi con slicing usando la parola chiave `del`. Output: `[42, 'Hello', 42]`.
⑤ Riassegnazione usando lo slicing. Output: `[42, 'Ciao', 24]`.
⑥ Eliminazione per mezzo di assegnazione e slicing. Output: `[42]`.

10.2.5. Insiemi

Python ha due tipi predefiniti di insiemi per rappresentare collezioni di elementi unici con ordine arbitrario: `set` e `frozenset`.

Gli elementi in un `set` possono essere di tipi diversi, ma devono essere tutti hashable. Le istanze del tipo `set` sono mutabili e quindi non hashable, mentre le istanze del tipo `frozenset` sono immutabili e hashable.

Non è possibile avere un `set` i cui elementi siano altri `set`, ma è possibile avere un `set`, o un `frozenset`, i cui elementi siano `frozenset`.

Entrambi i tipi `set` e `frozenset` derivano direttamente dalla classe base `object`.

10.2.5.1. `set`

Per denotare un `set`, si utilizza una serie di espressioni separate da virgole all'interno di parentesi graffe. Si può opzionalmente mettere una virgola ridondante dopo l'ultimo elemento.

Esempi di letterali di `set`:

```
s = {42, 3.14, 'hello'}  ①

s = {100}  ②

s = set()  ③
```

① `set` definito per mezzo di letterali e delimitatori.
② Non esiste un letterale per un `set` vuoto, pertanto si deve usare il costruttore `set()`.

Si può creare una istanza di `set` chiamando il costruttore senza argomenti, per un `set` vuoto, o con un oggetto iterabile (per un `set` i cui elementi sono quelli dell'iterabile).

I `set` sono mutabili, il che significa che una volta creati, possono essere modificati. Supportano operazioni come aggiunta, rimozione e controllo dell'esistenza di elementi.

- Aggiunta di elementi:

  ```
  s = {1, 2, 3}

  s.add(4)

  print(s)  ①
  ```

  ① `add()` aggiunge un elemento. Output: `{1, 2, 3, 4}`.

- Rimozione di elementi:

  ```
  s = {1, 2, 3, 4, 3}  ①

  s.remove(3)

  print(s)  ②
  ```

① Quando crei un `set` con elementi duplicati, come `{1, 2, 3, 4, 3}`, Python rimuove automaticamente i duplicati. Quindi, il `set` `s` diventerà `{1, 2, 3, 4}`.
② `remove()` elimina l'elemento specificato dal `set`. Output: `{1, 2, 4}`.

- Controllo dell'esistenza di un elemento:

```
s = {1, 2, 3}

print(2 in s)                                                    ①

print(5 in s)                                                    ②
```

① Output: `True`.
② Output: `False`.

- Operazioni insiemistiche:

```
s1 = set([1, 2, 3])
s2 = set([3, 4, 5])

print(s1 | s2)                                                   ①

print(s1 & s2)                                                   ②

print(s1 - s2)                                                   ③
```

① Operazione di unione insiemistica. Output: `set({1, 2, 3, 4, 5})`.
② Operazione di intersezione insiemistica. Output: `set({3})`.
③ Operazione di differenza insiemistica. Output: `set({1, 2})`.

10.2.5.2. `frozenset`

Allo stesso modo, si può creare un `frozenset` per mezzo del costruttore, senza argomenti o con un iterabile.

```
#
fs = frozenset([1, 2, 3, 4])

print(fs)                                                        ①

fs_empty = frozenset()

print(fs_empty)                                                  ②
```

① Creazione di un `frozenset` da una lista. Output: `frozenset({1, 2, 3, 4})`.
② Creazione di un `frozenset` vuoto. Output: `frozenset()`.

I `frozenset` sono immutabili, il che significa che una volta creati, non possono essere modificati. Supportano operazioni di lettura come controllo dell'esistenza di elementi e operazioni insiemistiche (unione, intersezione, differenza), ma non supportano operazioni di modifica. Il comportamento è identico alle operazioni di `set`.

10.2.6. Mappature

Le mappature rappresentano insiemi finiti di oggetti indicizzati da insiemi di indici arbitrari. La notazione con le parentesi quadre, `a[k]`, seleziona l'elemento indicizzato da `k` nella mappatura `a` e può essere utilizzata all'interno di espressioni oppure a sinistra di assegnazioni o istruzioni `del`. La funzione `len()` restituisce il numero di elementi in una mappatura.

Attualmente, esiste un solo tipo di mappatura predefinita in Python: il **dizionario**. È mutabile e la classe corrispondente è `dict` che deriva da `object`.

I dizionari rappresentano insiemi finiti di oggetti indicizzati da valori quasi arbitrari, detti **chiavi**. Le chiavi nei dizionari devono essere di tipi arbitrati ma hashable e, come oggetti, uniche. La ragione è che l'implementazione efficiente dei dizionari richiede che il valore hash di una chiave rimanga costante.

I dizionari preservano l'ordine di inserimento, il che significa che le chiavi verranno prodotte nello stesso ordine in cui sono state aggiunte sequenzialmente al dizionario. Sostituire una chiave esistente non cambia l'ordine; tuttavia, rimuovere una chiave e reinserirla la aggiungerà alla fine invece di mantenerne la posizione precedente.

I dizionari si possono creare per mezzo di una serie di coppie di espressioni, separate da virgole, all'interno di parentesi graffe. Le chiavi e i valori sono separati da due punti. È possibile inserire una virgola dopo l'ultimo elemento.

Esempi:

```
d1 = {"a": 42, "b": 2, 42: 3, "b": 24}                                    ①

print(d1)                                                                 ②

x = 2.

d2 = {2**2: "uno", 2 * x: "due", "hello".upper(): x, }

print(d2)                                                                 ③

d3 = {}                                                                   ④
```

① Dizionario con tre coppie chiave-valore.
② Le chiavi devono essere uniche quindi l'interprete mantiene solo una chiave con un valore arbitrario. Output: `{'a': 42, 'b': 24, 42: 3}`.
③ Dizionario con due coppie chiave-valore e virgola opzionale in fondo. Output: `{4: 'due', 'HELLO': 2.0}`.
④ Dizionario vuoto.

Possiamo creare i dizionari anche usando la classe `dict` oltre che i letterali:

```
d1 = dict()                                                               ①

d2 = dict(a=1, b=2, c=3)                                                  ②

d3 = dict([("a", 1), ("b", 2), ("c", 3)])

print(d3)                                                                 ③
```

① Crea un dizionario vuoto.
② Crea un dizionario con tre coppie chiave-valore specificate come argomenti.
③ Crea un dizionario a partire da un iterabile di tuple. Output: `{'a': 1, 'b': 2, 'c': 3}`.

Operazioni sui dizionari:

- Accesso agli elementi:

```
d = {"a": 1, "b": 2, "c": 3}

print(d["a"])                                                    ①

print(d.get("b"))                                                ②

print(d.keys())                                                  ③

print(d.values())                                                ④

print(d.items())                                                 ⑤
```

① Accesso al valore associato alla chiave `"a"`. Output: `1`.
② Utilizzo del metodo `get()` per accedere al valore associato alla chiave `"b"`. Output: `2`.
③ Accesso a tutte le chiavi del dizionario. Output: `dict_keys(['a', 'b', 'c'])`.
④ Accesso a tutti i valori del dizionario. Output: `dict_values([1, 2, 3])`.
⑤ Accesso a tutte le coppie chiave-valore del dizionario. Output: `dict_items([('a', 1), ('b', 2), ('c', 3)])`.

- Modifica degli elementi:

```
d = {"a": 1, "b": 2, "c": 3}

d["b"] = 20                                                      ①

d["d"] = 4                                                       ②

print(d)                                                         ③
```

① Modifica del valore associato alla chiave `"b"`. Output: `{"a": 1, "b": 20, "c": 3}`.
② Aggiunta di una nuova coppia chiave-valore. Output: `{"a": 1, "b": 20, "c": 3, "d": 4}`.
③ Output del dizionario dopo le modifiche: `{"a": 1, "b": 20, "c": 3, "d": 4}`.

- Rimozione degli elementi:

```
d = {"a": 1, "b": 2, "c": 3}

del d["b"]                                                       ①

print(d)                                                         ②

valore = d.pop("c")                                              ③

print(d)                                                         ④
```

```
print(valore)                                                    ⑤

d.clear()                                                        ⑥

print(d)                                                         ⑦
```

① Rimozione della coppia chiave-valore con chiave `"b"` usando `del`.
② Output dopo la rimozione con `del`. Output: `{"a": 1, "c": 3}`.
③ Rimozione della coppia chiave-valore con chiave `"c"` usando `pop()`, che restituisce il valore associato.
④ Output dopo la rimozione con `pop()`. Output: `{"a": 1}`.
⑤ Valore rimosso con `pop()`. Output: `3`.
⑥ Rimozione di tutte le coppie chiave-valore usando `clear()`.
⑦ Output dopo l'uso di `clear()`. Output: `{}`.

- Operazioni di controllo:

```
d = {"a": 1, "b": 2, "c": 3}

print("a" in d)                                                  ①

print("z" in d)                                                  ②
```

① Verifica se la chiave `"a"` è presente nel dizionario. Output: `True`.
② Verifica se la chiave `"z"` è presente nel dizionario. Output: `False`.

Lo spacchettamento (*unpacking*) è una funzionalità che permette di combinare i contenuti di più dizionari in un unico dizionario. Si utilizza l'operatore `**` per esplodere i contenuti del singolo dizionario. Alternativamente, si può effettuare l'unione di dizionari per mezzo dell'operatore `|`.

Esempi:

```
d1 = {'a': 1, 'b': 2}
d2 = {'c': 3, 'd': 4}

d3 = {**d1, **d2}                                                ①

print(d3)                                                        ②

d4 = d1 | d2                                                     ③

print(d4)                                                        ④

d5 = dict.fromkeys('a', 1)                                       ⑤

print(d5)                                                        ⑥

d6 = dict.fromkeys(['a', 'b', 'c'])                              ⑦

print(d6)                                                        ⑧
```

① Spacchettamento dei dizionari `d1` e `d2` in un nuovo dizionario `d3`.

② Stampa del dizionario `d3`. Output: `{'a': 1, 'b': 2, 'c': 3, 'd': 4}`.
③ Creazione di un nuovo dizionario `d4` unendo `d1` e `d2` per mezzo dell'operatore `|`.
④ Stampa del dizionario `d4`. Output: `{'a': 1, 'x': 5, 'c': 2}`.
⑤ Creazione di un dizionario `d5` usando `dict.fromkeys` con le chiavi dalla stringa `'a'` e valore `1`.
⑥ Stampa del dizionario `d5`. Output: `{'a': 1}`.
⑦ Creazione di un dizionario `d6` usando `dict.fromkeys` con le chiavi dalla lista `[1, 2, 3]` e valori `None`.
⑧ Stampa del dizionario `d6`. Output: `{'a': None, 'b': None, 'c': None}`.

10.2.7. None

`None` è un oggetto predefinito in Python che rappresenta un valore nullo. Non ha metodi né altri attributi. `None` può essere utilizzato per fare riferimento ad un oggetto qualsiasi, oppure quando si vuole indicare l'assenza di un oggetto.

Le funzioni restituiscono `None` come risultato a meno che `return` non sia seguito da elenco di oggetti. `None` è hashable e può essere utilizzato come chiave di un dizionario. `None` è un oggetto della classe `NoneType`, che deriva da `object` ed è unico, cioè non esiste una seconda istanza di `NoneType`.

Esempi di utilizzo di `None`:

```
x = None                                                    ①

def funzione():
    pass

print(funzione())                                           ②

d = {None: "valore"}                                        ③

print(d)                                                    ④
```

① Assegnazione di `None` a una variabile.
② Le funzioni che non hanno un'istruzione di ritorno specifica restituiscono `None`. Output: `None`.
③ `None` può essere utilizzato come chiave in un dizionario.
④ Output: `{None: 'valore'}`.

10.2.8. Ellissi

L'ellissi, scritta come tre punti consecutivi senza spazi intermedi (`...`), è un oggetto speciale in Python utilizzato in applicazioni numeriche o come alternativa a `None` quando `None` è un valore valido.

Ad esempio, per inizializzare un dizionario che può accettare `None` come valore legittimo, si può inizializzare con `...` come indicatore di "nessun valore fornito, neanche `None`". `Ellipsis` è hashable e può essere utilizzato come chiave di un dizionario. `Ellipsis` è un oggetto della classe `ellipsis`, che deriva da `object` ed è unica come `None`.

Esempi di utilizzo di `Ellipsis`:

```
e = ...                                                                    ①

print(e)                                                                   ②

d = {None: "valore1", ...: "valore2"}                                      ③

print(d)                                                                   ④

def funzione():
    ...

print(funzione())                                                          ⑤
```

① Assegnazione di `Ellipsis` a una variabile.
② Output: `Ellipsis`.
③ `Ellipsis` può essere utilizzato come chiave in un dizionario.
④ Output: `{None: 'valore1', Ellipsis: 'valore2'}`.
⑤ Le funzioni possono contenere `Ellipsis` come segnaposto per future implementazioni. Output: `None`.

11. Istruzioni

In Python, un programma è composto da una sequenza di istruzioni che l'interprete esegue una dopo l'altra. Le istruzioni sono i comandi fondamentali inviati al sistema operativo per la generazione delle attività computazionali da parte dell'hardware.

Ogni istruzione rappresenta un'azione, come la creazione di una variabile, l'esecuzione di una iterazione di ciclo, la definizione di una funzione o la stampa di un messaggio sullo schermo.

Le istruzioni si distinguono in **semplici** e **composte**:

- Le istruzioni semplici sono quelle che non contengono altre istruzioni al loro interno. Sono eseguite dall'interprete come un singolo blocco di codice.
- Le istruzioni composte contengono altre istruzioni al loro interno. Queste istruzioni definiscono blocchi di codice che possono includere altre istruzioni semplici o composte e che non possono essere vuoti.

11.1. Istruzione di gestione identificatori

11.1.1. Assegnamenti

Gli assegnamenti in Python sono istruzioni semplici che collegano valori a variabili utilizzando l'operatore `=`. L'assegnamento in Python è una definizione e non può mai far parte di un'espressione. Per eseguire un'assegnamento come parte di un'espressione, è necessario utilizzare l'operatore `:=` (noto come operatore "walrus").

Esempio di assegnamento semplice:

```
x = 10                                                                    ①
y = 20                                                                    ②
print(x + y)                                                              ③
```

① Assegna il valore `10` alla variabile `x`.
② Assegna il valore `20` alla variabile `y`.
③ Output della somma di `x` e `y`: `30`.

Esempio di assegnamento con l'operatore walrus:

```
if (n := 10) > 5:                                                         ①
    print(n)                                                              ②
```

① Assegna il valore `10` a `n` e verifica se `n` è maggiore di `5`.
② Output di `n`: `10`.

11.1.2. Importazione di moduli

L'istruzione semplice `import` viene utilizzata per importare moduli in un programma Python, permettendo l'accesso alle funzioni, classi e variabili definite al loro interno.

Tutti gli identificatori definiti nel modulo `nome_modulo` si importano colla sintassi `import nome_modulo` e dal quel punto fino alla fine del modulo importatore, sono accessibili colla notazione data dal nome del modulo seguito dal punto e l'identificatore di interesse, cioè `nome_modulo.nome_variabile` dove `nome_variabile` è l'identificatore importato.

Una sintassi alternativa è `from nome_modulo import *`, per cui gli identificatori sono utilizzabili senza preporre `nome_modulo.`.

Infine, si possono importare identificatori particolari usando `from nome_modulo import` seguito dall'elenco degli identificatori necessari, separati da virgole.

Esempi:

- Importazione di tutti gli identificatori di un modulo:

```
import math                          ①

print(math.sqrt(4))                  ②
```

① Si importa il modulo `math`.
② Output: `2.0`.

- Importazione di identificatori particolari:

```
from math import sqrt, pi

print(sqrt(4))                       ①

print(pi)                            ②
```

① Output: `2.0`.
② Output: `3.141592653589793`.

- Importazione di un modulo con un alias:

```
import numpy as np                   ①

array = np.array([1, 2, 3])

print(array)                         ②
```

① `numpy` è una librarie non facente parte dello standard Python, importata solitamente con un identificatore abbreviato in `np`.
② Output: `[1 2 3]`.

- Importazione di tutti gli identificatori di un modulo con accesso semplificato:

```
from math import *

print(sqrt(4))                                                    ①

print(pi)                                                         ②
```

① Output: `2.0`.
② Output: `3.141592653589793`.

È importante notare che l'istruzione `import` carica e inizializza il modulo solo una volta per sessione del programma. Se il modulo è già stato importato in precedenza, Python utilizza la versione già caricata, riducendo così il tempo di esecuzione e il consumo di memoria.

Quando si importa un modulo, Python cerca il modulo nelle directory specificate nella variabile `sys.path`. Questa variabile include la directory corrente, le directory specificate nella variabile d'ambiente `PYTHONPATH`, e le directory di installazione predefinite.

Esempio:

```
import sys

print(sys.path)                                                   ①
```

① Elenco delle directory del computer dove Python cerca i moduli.

11.2. Istruzioni di controllo di flusso

Il flusso di controllo di un programma regola l'ordine in cui le istruzioni vengno eseguite. Il flusso di controllo di un programma Python dipende principalmente da istruzioni condizionali, cicli e chiamate a funzioni.

Anche il sollevamento e la gestione delle eccezioni influenzano il flusso di controllo (tramite le istruzioni `try` e `with`).

11.2.1. Istruzione di esecuzione condizionale

Spesso è necessario eseguire alcune istruzioni solo quando una certa condizione è vera o scegliere le istruzioni da eseguire a seconda di condizioni mutuamente esclusive. L'istruzione composta `if`, che comprende le clausole `if`, `elif` ed `else`, consente di eseguire condizionalmente blocchi di istruzioni.

La sintassi dell'istruzione `if` in pseudocodice è la seguente:

```
if espressione:                                                   ①
  istruzione(i)                                                   ②

elif espressione:                                                 ③
  istruzione(i)                                                   ④

elif espressione:                                                 ⑤
  istruzione(i)                                                   ⑥
```

```
else:                                                                    ⑦
  istruzione(i)                                                          ⑧
```

① Clausola `if` con una condizione, cioè una espressione con valore interpretato come logico.
② Blocco di codice eseguito se la condizione `if` è vera.
③ Clausola `elif` con una condizione.
④ Blocco di codice eseguito se la condizione `elif` è vera.
⑤ Clausola `elif` con una condizione.
⑥ Blocco di codice eseguito se la condizione `elif` è vera.
⑦ Clausola `else` eseguita se tutte le condizioni precedenti sono false.
⑧ Blocco di codice eseguito dalla clausola `else`.

Le clausole `elif` ed `else` sono opzionali. Ecco un tipico esempio di istruzione `if` con tutti e tre i tipi di clausole:

```
if x < 0:                                                                ①
  print('x è negativo')                                                  ②

elif x % 2:                                                              ③
  print('x è positivo e dispari')                                        ④

else:
  print('x è pari e non negativo')                                       ⑤
```

① Controlla se `x` è negativo.
② Stampa `"x è negativo"` se la condizione è vera.
③ Controlla se `x` è positivo e dispari.
④ Stampa `"x è positivo e dispari"` se la condizione è vera.
⑤ Stampa `"x è pari e non negativo"` se nessuna delle condizioni precedenti è vera.

Ogni clausola controlla una o più istruzioni raggruppate in un blocco di codice: si posizionano le istruzioni del blocco su righe logiche separate dopo la riga contenente la parola chiave della clausola (nota come riga intestazione della clausola), con un'indentazione tipicamente di quattro spazi oltre la riga intestazione. Il blocco termina quando l'indentazione torna al livello della riga intestazione della clausola o ulteriormente a sinistra (questo è lo stile imposto da PEP 8).

È possibile utilizzare qualsiasi espressione Python come condizione in una clausola `if` o `elif`. Utilizzare un'espressione in questo modo è noto come usarla in un contesto booleano. In questo contesto, qualsiasi valore viene considerato vero o falso. Qualsiasi numero diverso da zero o contenitore non vuoto (stringa, tupla, lista, dizionario, set, ecc.) viene valutato come vero, mentre zero (0, di qualsiasi tipo numerico), `None` e contenitori vuoti vengono valutati come falsi.

Per testare un valore `x` in un contesto booleano, utilizzare lo stile di codifica seguente:

```
if x:
```

Questa è la forma più chiara e più "Pythonica".

Non utilizzare nessuna delle seguenti forme:

```
if x is True:

if x == True:

if bool(x):
```

C'è una differenza cruciale tra dire che un'espressione restituisce `True` (significa che l'espressione restituisce il valore 1 con il tipo `bool`) e dire che un'espressione viene valutata come vera (significa che l'espressione restituisce qualsiasi risultato che è vero in un contesto booleano). Quando si testa un'espressione, ad esempio in una clausola `if`, interessa solo come viene valutata, non cosa, precisamente, restituisce. Come menzionato in precedenza, "valutata come vera" è spesso espresso informalmente come "è veritiera", e "valutata come falsa" come "è falsa".

Quando la condizione della clausola `if` viene valutata come vera, le istruzioni all'interno della clausola `if` vengono eseguite, quindi l'intera istruzione `if` termina. Altrimenti, Python valuta la condizione di ciascuna clausola `elif`, in ordine. Le istruzioni all'interno della prima clausola `elif` la cui condizione viene valutata come vera, se presente, vengono eseguite e l'intera istruzione `if` termina. Altrimenti, quando esiste una clausola `else`, essa viene eseguita. In ogni caso, le istruzioni successive all'intera costruzione `if`, allo stesso livello, vengono eseguite successivamente.

11.2.2. Istruzione di pattern matching

L'istruzione `match` introduce il pattern matching strutturale nel linguaggio Python. Questa funzionalità consente di testare facilmente la struttura e il contenuto degli oggetti Python.

La struttura sintattica generale dell'istruzione `match` è la seguente:

```
match espressione:                                    ①
  case pattern1 [if guard1]:                          ②
    istruzione(i)                                     ③

  case pattern2 [if guard2]:
    istruzione(i)                                     ④

  ...

  case _:                                             ⑤
    istruzione(i)                                     ⑥
```

① La parola chiave `match` seguita da un'espressione il cui valore diventa il soggetto del matching.
② Clausole `case` indentate che controllano l'esecuzione del blocco di codice che contengono. Possono includere un'opzione `if guard` per ulteriori controlli.
③ Blocco di istruzioni da eseguire se `pattern1` corrisponde.
④ Blocco di istruzioni da eseguire se `pattern2` corrisponde.
⑤ Pattern wildcard che corrisponde a qualsiasi valore.
⑥ Blocco di codice eseguito se nessun altro pattern corrisponde.

Durante l'esecuzione, Python prima valuta l'espressione, quindi testa il valore risultante contro il pattern in ciascuna clausola `case` a turno, dall'inizio alla fine, fino a quando uno corrisponde. A quel punto, il blocco di codice indentato all'interno della clausola `case` corrispondente viene eseguito. Un pattern può fare due cose:

- Verificare che il soggetto sia un oggetto con una struttura particolare.
- Associare componenti corrispondenti a nomi per un uso successivo (di solito all'interno della clausola `case` associata).

Quando un pattern corrisponde al soggetto, il `guard` consente un controllo finale prima della selezione della clausola per l'esecuzione. Tutti i binding dei nomi del pattern sono già avvenuti, e si possono usare nel `guard`. Quando non c'è un `guard`, o quando il `guard` viene valutato come vero, il blocco di codice indentato della clausola `case` viene eseguito, dopo di che l'esecuzione dell'istruzione `match` è completa e non vengono controllate ulteriori clausole.

L'istruzione `match`, di per sé, non prevede un'azione predefinita. Se ne serve una, l'ultima clausola `case` deve specificare un pattern wildcard, uno la cui sintassi garantisce che corrisponda a qualsiasi valore del soggetto. È un errore di sintassi seguire una clausola `case` con un pattern wildcard con ulteriori clausole `case`.

Gli elementi del pattern non possono essere creati in anticipo, associati a variabili e riutilizzati in più punti. La sintassi del pattern è valida solo immediatamente dopo la parola chiave `case`, quindi non c'è modo di eseguire tale assegnazione. Per ogni esecuzione di un'istruzione `match`, l'interprete è libero di memorizzare nella cache le espressioni del pattern che si ripetono all'interno delle clausole, ma la cache inizia vuota per ogni nuova esecuzione.

Esempio di utilizzo dell'istruzione `match`:

```
def azione(comando, livello):                                    (1)
  match comando:                                                 (2)
    case "start" if livello > 1:                                 (3)
      print("Avvio con livello alto")                            (4)

    case "start":                                                (5)
      print("Avvio con livello basso")                           (6)

    case "stop" if livello > 1:                                  (7)
      print("Arresto con livello alto")                          (8)

    case "stop":                                                 (9)
      print("Arresto con livello basso")                         (10)

    case "pause":                                                (11)
      print("Pausa")                                             (12)

    case _:                                                      (13)
      print("Comando sconosciuto")                               (14)

azione("start", 2)                                               (15)
azione("start", 1)                                               (16)
azione("pause", 3)                                               (17)
azione("exit", 0)                                                (18)
```

(1) Definizione della funzione `azione` con due parametri: `comando` e `livello`.
(2) Inizia il blocco `match` per il valore `comando`.
(3) Pattern `"start"` con guard che verifica se `livello` è maggiore di 1.
(4) Output se `comando` è `"start"` e `livello` è maggiore di 1.

⑤ Pattern `"start"` senza guard.
⑥ Output se `comando` è `"start"` e `livello` è minore o uguale a 1.
⑦ Pattern `"stop"` con guard che verifica se `livello` è maggiore di 1.
⑧ Output se `comando` è `"stop"` e `livello` è maggiore di 1.
⑨ Pattern `"stop"` senza guard.
⑩ Output se `comando` è `"stop"` e `livello` è minore o uguale a 1.
⑪ Pattern `"pause"`.
⑫ Output se `comando` è `"pause"`.
⑬ Pattern wildcard che corrisponde a qualsiasi altro valore.
⑭ Output se `comando` non corrisponde a nessun altro pattern.
⑮ Chiamata a `azione` con `"start"` e livello `2`. Output: `Avvio con livello alto`.
⑯ Chiamata a `azione` con `"start"` e livello `1`. Output: `Avvio con livello basso`.
⑰ Chiamata a `azione` con `"pause"` e livello `3`. Output: `Pausa`.
⑱ Chiamata a `azione` con `"exit"` e livello `0`. Output: `Comando sconosciuto`.

In questo esempio, la guardia `if livello > 1` aggiunge una condizione extra per i casi `"start"` e `"stop"`, permettendo di distinguere tra diversi livelli di comando.

11.2.2.1. Pattern letterali

I pattern letterali corrispondono a valori letterali come interi, float, stringhe, ecc. La corrispondenza viene effettuata confrontando il valore del soggetto con il valore del pattern.

Esempio:

```
def controlla_valore(valore):                                    ①
  match valore:                                                  ②
    case 1:                                                      ③
      return "Uno"                                               ④

    case "ciao":                                                 ⑤
      return "Saluto"                                            ⑥

    case True:                                                   ⑦
      return "Vero"                                              ⑧

    case None:                                                   ⑨
      return "Nessuno"                                           ⑩

    case _:                                                      ⑪
      return "Altro"                                             ⑫

print(controlla_valore(1))                                       ⑬
print(controlla_valore("ciao"))                                  ⑭
print(controlla_valore(False))                                   ⑮
```

① Definizione della funzione `controlla_valore`.
② Inizia il blocco `match` per il valore `valore`.
③ Pattern letterale `1`.

④ Output se `valore` è `1`.
⑤ Pattern letterale `"ciao"`.
⑥ Output se `valore` è `"ciao"`.
⑦ Pattern letterale `True`.
⑧ Output se `valore` è `True`.
⑨ Pattern letterale `None`.
⑩ Output se `valore` è `None`.
⑪ Pattern wildcard che corrisponde a qualsiasi altro valore.
⑫ Output se `valore` non corrisponde a nessun altro pattern.
⑬ Chiamata a `controlla_valore` con `1`. Output: `Uno`.
⑭ Chiamata a `controlla_valore` con `"ciao"`. Output: `Saluto`.
⑮ Chiamata a `controlla_valore` con `False`. Output: `Altro`.

11.2.2.2. Pattern di cattura

I pattern di cattura utilizzano nomi non qualificati (nomi senza punti) per catturare valori all'interno di un pattern. Questi nomi sono wildcard che corrispondono a qualsiasi valore, ma con un effetto collaterale: il nome viene associato all'oggetto corrispondente nella corrispondente espressione di pattern matching. I binding creati dai pattern di cattura rimangono disponibili dopo l'esecuzione dell'istruzione `match`, permettendo alle istruzioni all'interno del blocco `case` e al codice successivo di processare i valori catturati.

Un pattern di cattura semplice associa il nome della variabile al valore corrispondente. Se il pattern di cattura è combinato con altre forme di pattern, può catturare parti specifiche del soggetto.

Esempio di utilizzo dei pattern di cattura:

```
def descrivi_valore(valore):                                    ①
  match valore:                                                 ②
    case x if x < 0:                                            ③
      print(f"{x} è un numero negativo")                        ④

    case x if x == 0:                                           ⑤
      print(f"{x} è zero")                                      ⑥

    case x if x > 0 and x % 2 == 0:                             ⑦
      print(f"{x} è un numero positivo pari")                   ⑧

    case x if x > 0 and x % 2 != 0:                             ⑨
      print(f"{x} è un numero positivo dispari")                ⑩

    case _ if isinstance(valore, str):                          ⑪
      print(f'"{valore}" è una stringa')                        ⑫

    case _ if isinstance(valore, list):                         ⑬
      print(f"{valore} è una lista")                            ⑭

    case _:                                                     ⑮
      print("Tipo di valore non riconosciuto")                  ⑯

# Esempi di utilizzo della funzione
```

```
descrivi_valore(-5)                                    (17)
descrivi_valore(0)                                     (18)
descrivi_valore(4)                                     (19)
descrivi_valore(7)                                     (20)
descrivi_valore("ciao")                                (21)
descrivi_valore([1, 2, 3])                             (22)
descrivi_valore({"chiave": "valore"})                  (23)
```

(1) Definizione della funzione `descrivi_valore`.
(2) Inizio del blocco `match` per il valore `valore`.
(3) Pattern di cattura `x` con guard `x < 0`.
(4) Output se `valore` è un numero negativo. Esempio di output: `-5 è un numero negativo`.
(5) Pattern di cattura `x` con guard `x == 0`.
(6) Output se `valore` è zero. Esempio di output: `0 è zero`.
(7) Pattern di cattura `x` con guard `x > 0 and x % 2 == 0`.
(8) Output se `valore` è un numero positivo pari. Esempio di output: `4 è un numero positivo pari`.
(9) Pattern di cattura `x` con guard `x > 0 and x % 2 != 0`.
(10) Output se `valore` è un numero positivo dispari. Esempio di output: `7 è un numero positivo dispari`.
(11) Pattern di guard `isinstance(valore, str)`.
(12) Output se `valore` è una stringa. Esempio di output: `"ciao" è una stringa`.
(13) Pattern di guard `isinstance(valore, list)`.
(14) Output se `valore` è una lista. Esempio di output: `[1, 2, 3] è una lista`.
(15) Pattern wildcard che corrisponde a qualsiasi altro valore.
(16) Output se `valore` non corrisponde a nessun altro pattern. Esempio di output: `Tipo di valore non riconosciuto`.
(17) Chiamata a `descrivi_valore` con `-5`. Output: `-5 è un numero negativo`.
(18) Chiamata a `descrivi_valore` con `0`. Output: `0 è zero`.
(19) Chiamata a `descrivi_valore` con `4`. Output: `4 è un numero positivo pari`.
(20) Chiamata a `descrivi_valore` con `7`. Output: `7 è un numero positivo dispari`.
(21) Chiamata a `descrivi_valore` con `"ciao"`. Output: `"ciao" è una stringa`.
(22) Chiamata a `descrivi_valore` con `[1, 2, 3]`. Output: `[1, 2, 3] è una lista`.
(23) Chiamata a `descrivi_valore` con `{"chiave": "valore"}`. Output: `Tipo di valore non riconosciuto`.

11.2.2.3. Pattern a valore

I pattern a valore utilizzano nomi qualificati per rappresentare valori piuttosto che catturarli. In questo modo, puoi fare riferimento a valori specifici all'interno di un pattern senza rischiare di sovrascrivere variabili esistenti. I nomi qualificati possono essere attributi di una classe o attributi di istanze di classe.

Poiché i nomi semplici catturano i valori durante il pattern matching, è necessario utilizzare riferimenti agli attributi (nomi qualificati come `nome.attr`) per esprimere valori che possono cambiare tra le diverse esecuzioni dello stesso statement `match`.

Esempio di utilizzo dei pattern a valore:

```
class Valori:
  V1 = 42
  V2 = "ciao"
```

```
  V3 = [1, 2, 3]

obj = Valori()
obj.V4 = 99                                                    (1)

def controlla_valore(valore):                                  (2)
  match valore:                                                (3)
    case Valori.V1:                                            (4)
      print("Valore uguale a 42")                              (5)

    case Valori.V2:                                            (6)
      print('Valore uguale a "ciao"')                          (7)

    case Valori.V3:                                            (8)
      print("Valore uguale a [1, 2, 3]")                       (9)

    case obj.V4:                                               (10)
      print("Valore uguale a 99")                              (11)

    case _:                                                    (12)
      print("Valore non riconosciuto")                         (13)

# Esempi di utilizzo della funzione
controlla_valore(42)                                           (14)
controlla_valore("ciao")                                       (15)
controlla_valore([1, 2, 3])                                    (16)
controlla_valore(99)                                           (17)
controlla_valore(100)                                          (18)
```

(1) Assegna un nuovo attributo `V4` all'istanza `obj` della classe `Valori`.
(2) Definizione della funzione `controlla_valore`.
(3) Inizia il blocco `match` per il valore `valore`.
(4) Pattern a valore per `Valori.V1`.
(5) Output se `valore` è uguale a `42`. Esempio di output: `Valore uguale a 42`.
(6) Pattern a valore per `Valori.V2`.
(7) Output se `valore` è uguale a `"ciao"`. Esempio di output: `Valore uguale a "ciao"`.
(8) Pattern a valore per `Valori.V3`.
(9) Output se `valore` è uguale a `[1, 2, 3]`. Esempio di output: `Valore uguale a [1, 2, 3]`.
(10) Pattern a valore per `obj.V4`.
(11) Output se `valore` è uguale a `99`. Esempio di output: `Valore uguale a 99`.
(12) Pattern wildcard che corrisponde a qualsiasi altro valore.
(13) Output se `valore` non corrisponde a nessun altro pattern. Esempio di output: `Valore non riconosciuto`.
(14) Chiamata a `controlla_valore` con `42`. Output: `Valore uguale a 42`.
(15) Chiamata a `controlla_valore` con `"ciao"`. Output: `Valore uguale a "ciao"`.
(16) Chiamata a `controlla_valore` con `[1, 2, 3]`. Output: `Valore uguale a [1, 2, 3]`.
(17) Chiamata a `controlla_valore` con `99`. Output: `Valore uguale a 99`.
(18) Chiamata a `controlla_valore` con `100`. Output: `Valore non riconosciuto`.

In questo esempio, `Valori.V1`, `Valori.V2`, e `Valori.V3` sono attributi della classe `Valori`, mentre `obj.V4` è

un attributo dell'istanza `obj` della classe `Valori`.

11.2.2.4. Pattern OR

I pattern OR utilizzano l'operatore `|` per combinare più pattern. Il match ha successo se uno qualsiasi dei pattern combinati corrisponde al soggetto.

Esempio di utilizzo dei pattern OR:

```
def descrivi_numero(numero):                                    (1)
  match numero:                                                 (2)
    case 0 | 1:                                                 (3)
      print("Numero è 0 o 1")                                   (4)

    case 2 | 3:                                                 (5)
      print("Numero è 2 o 3")                                   (6)

    case _ if numero > 3:                                       (7)
      print("Numero è maggiore di 3")                           (8)

    case _:                                                     (9)
      print("Numero non riconosciuto")                          (10)

# Esempi di utilizzo della funzione
descrivi_numero(0)                                              (11)
descrivi_numero(2)                                              (12)
descrivi_numero(5)                                              (13)
descrivi_numero(-1)                                             (14)
```

(1) Definizione della funzione `descrivi_numero`.
(2) Inizia il blocco `match` per il valore `numero`.
(3) Pattern OR per `0 | 1`.
(4) Output se `numero` è `0` o `1`. Esempio di output: `Numero è 0 o 1`.
(5) Pattern OR per `2 | 3`.
(6) Output se `numero` è `2` o `3`. Esempio di output: `Numero è 2 o 3`.
(7) Pattern di guard per `numero > 3`.
(8) Output se `numero` è maggiore di `3`. Esempio di output: `Numero è maggiore di 3`.
(9) Pattern wildcard che corrisponde a qualsiasi altro valore.
(10) Output se `numero` non corrisponde a nessun altro pattern. Esempio di output: `Numero non riconosciuto`.
(11) Chiamata a `descrivi_numero` con `0`. Output: `Numero è 0 o 1`.
(12) Chiamata a `descrivi_numero` con `2`. Output: `Numero è 2 o 3`.
(13) Chiamata a `descrivi_numero` con `5`. Output: `Numero è maggiore di 3`.
(14) Chiamata a `descrivi_numero` con `-1`. Output: `Numero non riconosciuto`.

11.2.2.5. Pattern di gruppo

I pattern di gruppo utilizzano parentesi per raggruppare parti di un pattern, consentendo la combinazione di pattern complessi. Questa funzionalità è utile quando si vuole applicare operazioni di pattern matching su componenti specifici di un soggetto.

Esempio di utilizzo dei pattern di gruppo:

```
def analizza_tupla(tupla):                                                    ①
  match tupla:                                                                ②
    case (x, (y, z)):                                                         ③
      print(f"Primo elemento: {x}, Secondo elemento: ({y}, {z})")            ④

    case (x, y):                                                              ⑤
      print(f"Primo elemento: {x}, Secondo elemento: {y}")                    ⑥

    case _:                                                                   ⑦
      print("Pattern non riconosciuto")                                       ⑧

# Esempi di utilizzo della funzione
analizza_tupla((1, (2, 3)))                                                   ⑨
analizza_tupla((1, 4))                                                        ⑩
analizza_tupla((1, 2, 3))                                                     ⑪
```

① Definizione della funzione `analizza_tupla`.
② Inizia il blocco `match` per il valore `tupla`.
③ Pattern di gruppo `(x, (y, z))`.
④ Output se `tupla` corrisponde al pattern `(x, (y, z))`. Esempio di output: `Primo elemento: 1, Secondo elemento: (2, 3)`.
⑤ Pattern di gruppo `(x, y)`.
⑥ Output se `tupla` corrisponde al pattern `(x, y)`. Esempio di output: `Primo elemento: 1, Secondo elemento: 4`.
⑦ Pattern wildcard che corrisponde a qualsiasi altro valore.
⑧ Output se `tupla` non corrisponde a nessun altro pattern. Esempio di output: `Pattern non riconosciuto`.
⑨ Chiamata a `analizza_tupla` con `(1, (2, 3))`. Output: `Primo elemento: 1, Secondo elemento: (2, 3)`.
⑩ Chiamata a `analizza_tupla` con `(1, 4)`. Output: `Primo elemento: 1, Secondo elemento: 4`.
⑪ Chiamata a `analizza_tupla` con `(1, 2, 3)`. Output: `Pattern non riconosciuto`.

In questo esempio, il pattern `(x, (y, z))` confronta la tupla `tupla` per verificare se il secondo elemento è una tupla di due elementi, mentre il pattern `(x, y)` confronta la tupla `tupla` per verificare se ha esattamente due elementi. Questo esempio mostra come utilizzare i pattern di gruppo in Python per eseguire il pattern matching su tuple.

11.2.2.6. Pattern di sequenza

I pattern di sequenza corrispondono a sequenze come liste o tuple. Ogni elemento della sequenza viene confrontato con il pattern corrispondente. È possibile utilizzare l'asterisco (`*`) per catturare più elementi in una sottosequenza.

Esempio:

```
def verifica_sequenza(sequenza):                                              ①
  match sequenza:                                                             ②
    case [1, 2, 3]:                                                           ③
      return "Sequenza 1, 2, 3"                                               ④
```

```
    case [x, y, z]:                                                    ⑤
      return f"Sequenza generica: {x}, {y}, {z}"                       ⑥

    case [x, *y, z]:                                                   ⑦
      return f"Sequenza con primo e ultimo elemento: {x}, {z}, e mediano: {y}"  ⑧

    case _:                                                            ⑨
      return "Altro"                                                   ⑩

print(verifica_sequenza([1, 2, 3]))                                    ⑪
print(verifica_sequenza([4, 5, 6]))                                    ⑫
print(verifica_sequenza([7, 8, 9, 10]))                                ⑬
print(verifica_sequenza([7, 8]))                                       ⑭
```

① Definizione della funzione `verifica_sequenza`.
② Inizia il blocco `match` per il valore `sequenza`.
③ Pattern di sequenza `[1, 2, 3]`.
④ Output se `sequenza` è `[1, 2, 3]`.
⑤ Pattern di sequenza generico `[x, y, z]` per una sequenza di esattamente tre elementi.
⑥ Output con i valori catturati.
⑦ Pattern di sequenza con l'uso dell'asterisco (`*y`) per catturare tutti gli elementi intermedi tra il primo (`x`) e l'ultimo (`z`).
⑧ Output con il primo, l'ultimo e gli elementi intermedi della sequenza.
⑨ Pattern wildcard che corrisponde a qualsiasi altro valore.
⑩ Output se `sequenza` non corrisponde a nessun altro pattern.
⑪ Chiamata a `verifica_sequenza` con `[1, 2, 3]`. Output: `Sequenza 1, 2, 3`.
⑫ Chiamata a `verifica_sequenza` con `[4, 5, 6]`. Output: `Sequenza generica: 4, 5, 6`.
⑬ Chiamata a `verifica_sequenza` con `[7, 8, 9, 10]`. Output: `Sequenza con primo e ultimo elemento: 7, 10, e mediano: [8, 9]`.
⑭ Chiamata a `verifica_sequenza` con `[7, 8]`. Output: `Altro`.

In questo esempio, l'uso dell'asterisco `*` nel pattern `[x, *y, z]` permette di catturare una sottosequenza di lunghezza arbitraria tra il primo e l'ultimo elemento della sequenza. Questo rende possibile gestire sequenze di lunghezza variabile, mentre il pattern `[x, y, z]` corrisponde solo a sequenze di esattamente tre elementi.

11.2.2.7. Pattern `as`

È possibile utilizzare i pattern `as` per catturare i valori abbinati da pattern più complessi o componenti di un pattern, che i semplici pattern di cattura non possono gestire. Quando `P1` è un pattern, allora `P1 as name` è anche un pattern; quando `P1` ha successo, Python associa il valore abbinato al nome `name` nel namespace locale.

Esempio di utilizzo del pattern `as`:

```
def analizza_comando(comando):
  match comando:
    case ("start", param) as c:
      print(f"Avvio con parametro {param}. Comando completo: {c}")        ①
```

```
    case ("stop", param) as c:
      print(f"Arresto con parametro {param}. Comando completo: {c}")          ②

    case ("pause", param) as c:
      print(f"Pausa con parametro {param}. Comando completo: {c}")            ③

    case _ as c:
      print(f"Comando sconosciuto: {c}")                                      ④

# Esempi di utilizzo
analizza_comando(("start", 5))                                                ⑤
analizza_comando(("stop", 10))                                                ⑥
analizza_comando(("pause", 15))                                               ⑦
analizza_comando(("exit", 20))                                                ⑧
```

① Caso in cui il comando è un avvio con un parametro.
② Caso in cui il comando è un arresto con un parametro.
③ Caso in cui il comando è una pausa con un parametro.
④ Caso wildcard che cattura qualsiasi altro comando.
⑤ Chiamata a `analizza_comando` con `("start", 5)`. Output: `Avvio con parametro 5. Comando completo: ('start', 5)`.
⑥ Chiamata a `analizza_comando` con `("stop", 10)`. Output: `Arresto con parametro 10. Comando completo: ('stop', 10)`.
⑦ Chiamata a `analizza_comando` con `("pause", 15)`. Output: `Pausa con parametro 15. Comando completo: ('pause', 15)`.
⑧ Chiamata a `analizza_comando` con `("exit", 20)`. Output: `Comando sconosciuto: ('exit', 20)`.

11.2.2.8. Pattern di mappatura

I pattern di mappatura corrispondono alle mappature definite nel linguaggio come i dizionari. Ogni coppia chiave-valore viene confrontata con il pattern corrispondente.

Esempio:

```
def verifica_mappatura(mappatura):                                            ①
  match mappatura:                                                            ②
    case {"a": 1, "b": 2}:                                                    ③
      return "Mappatura a=1, b=2"                                             ④

    case {"a": x, "b": y}:                                                    ⑤
      return f"Mappatura generica: a={x}, b={y}"                              ⑥

    case _:                                                                   ⑦
      return "Altro"                                                          ⑧

print(verifica_mappatura({"a": 1, "b": 2}))                                   ⑨
print(verifica_mappatura({"a": 3, "b": 4}))                                   ⑩
print(verifica_mappatura({"c": 5}))                                           ⑪
```

① Definizione della funzione `verifica_mappatura`.
② Inizia il blocco `match` per il valore `mappatura`.
③ Pattern di mappatura `{"a": 1, "b": 2}`.
④ Output se `mappatura` è `{"a": 1, "b": 2}`.
⑤ Pattern di mappatura generico `{"a": x, "b": y}`.
⑥ Output con i valori catturati.
⑦ Pattern wildcard che corrisponde a qualsiasi altro valore.
⑧ Output se `mappatura` non corrisponde a nessun altro pattern.
⑨ Chiamata a `verifica_mappatura` con `{"a": 1, "b": 2}`. Output: `Mappatura a=1, b=2`.
⑩ Chiamata a `verifica_mappatura` con `{"a": 3, "b": 4}`. Output: `Mappatura generica: a=3, b=4`.
⑪ Chiamata a `verifica_mappatura` con `{"c": 5}`. Output: `Altro`.

11.2.2.9. Pattern di classe

I pattern di classe permettono di verificare se un oggetto è un'istanza di una particolare classe e di accedere ai suoi attributi. Un pattern di classe ha la forma generale `nome_o_attr(patterns)`, dove `nome_o_attr` è un nome semplice o qualificato legato a una classe, e `patterns` è una lista separata da virgole di specifiche di pattern.

Se non ci sono specifiche di pattern, il pattern di classe corrisponde a qualsiasi istanza della classe data. Se ci sono specifiche di pattern posizionali, queste devono precedere qualsiasi specifica di pattern nominata.

Le classi predefinite di Python come `bool`, `bytearray`, `bytes`, `dict`, `float`, `frozenset`, `int`, `list`, `set`, `str` e `tuple` sono tutte configurate per accettare un singolo pattern posizionale, che viene confrontato con il valore dell'istanza.

Esempio:

```
class Punto:
  def __init__(self, x, y):
    self.x = x
    self.y = y

def descrivi_punto(p):
  match p:
    case Punto(0, 0):
      return "Punto all'origine"                                    ①

    case Punto(x, 0):
      return f"Punto sull'asse x a {x}"                             ②

    case Punto(0, y):
      return f"Punto sull'asse y a {y}"                             ③

    case Punto(x, y) if x == y:
      return f"Punto sulla bisettrice x=y a ({x}, {y})"             ④

    case Punto(x, y):
      return f"Punto a ({x}, {y})"                                  ⑤
```

```
    case _:
      return "Non è un punto"                                                    ⑥

# Esempi di utilizzo
p1 = Punto(0, 0)
p2 = Punto(3, 0)
p3 = Punto(0, 4)
p4 = Punto(2, 2)
p5 = Punto(1, 5)

print(descrivi_punto(p1))                                                        ⑦
print(descrivi_punto(p2))                                                        ⑧
print(descrivi_punto(p3))                                                        ⑨
print(descrivi_punto(p4))                                                        ⑩
print(descrivi_punto(p5))                                                        ⑪
```

① Pattern di classe che verifica se `p` è un'istanza di `Punto` con `x` e `y` uguali a `0`.
② Pattern di classe che verifica se `p` è un'istanza di `Punto` con `y` uguale a `0`.
③ Pattern di classe che verifica se `p` è un'istanza di `Punto` con `x` uguale a `0`.
④ Pattern di classe con guard che verifica se `p` è un'istanza di `Punto` con `x` uguale a `y`.
⑤ Pattern di classe generico che verifica se `p` è un'istanza di `Punto` e cattura i valori di `x` e `y`.
⑥ Pattern wildcard che corrisponde a qualsiasi altro valore.
⑦ Chiamata a `descrivi_punto` con `Punto(0, 0)`. Output: `Punto all'origine`.
⑧ Chiamata a `descrivi_punto` con `Punto(3, 0)`. Output: `Punto sull'asse x a 3`.
⑨ Chiamata a `descrivi_punto` con `Punto(0, 4)`. Output: `Punto sull'asse y a 4`.
⑩ Chiamata a `descrivi_punto` con `Punto(2, 2)`. Output: `Punto sulla bisettrice x=y a (2, 2)`.
⑪ Chiamata a `descrivi_punto` con `Punto(1, 5)`. Output: `Punto a (1, 5)`.

In questo esempio, `Punto` è una classe con attributi `x` e `y`. I pattern di classe permettono di verificare se un oggetto è un'istanza di `Punto` e di accedere ai suoi attributi per ulteriori controlli.

11.3. Cicli

11.3.1. Istruzione `while`

L'istruzione composta `while` ripete l'esecuzione di un blocco di istruzioni fintantoché un'espressione condizionale risulta vera. La clausola `else` viene eseguita quando il ciclo `while` termina naturalmente (cioè, la condizione del ciclo diventa falsa).

Ecco la sintassi completa dello pseudocodice per l'istruzione `while` con la clausola `else`:

```
while espressione:                                                               ①
  istruzione(i)                                                                  ②

else:                                                                            ③
  istruzione(i)                                                                  ④
```

① L'interprete valuta l'espressione condizionale.

② Se l'espressione condizionale è vera, esegue il blocco di istruzioni all'interno del ciclo `while`. Al termine del blocco, torna a valutare l'espressione condizionale.
③ Se l'espressione condizionale è falsa, esegue il blocco di istruzioni all'interno della clausola `else`.
④ Il blocco di istruzioni all'interno della clausola `else` viene eseguito solo se il ciclo `while` termina naturalmente (cioè, la condizione del ciclo diventa falsa).

Esempio pratico con `while` e `else`:

```
x = 3

while x > 0:                                                    ①
  print(x)                                                      ②

  x -= 1

else:                                                           ③
  print("Ciclo terminato")                                      ④
```

① Inizia il ciclo `while` finché `x` è maggiore di 0.
② Stampa il valore corrente di `x`.
③ La clausola `else` viene eseguita quando il ciclo `while` termina naturalmente.
④ Stampa "Ciclo terminato".

L'istruzione `while` può anche includere una clausola `else` e le istruzioni `break` e `continue`. L'istruzione `break` interrompe il ciclo `while`, al pari di `return` se il ciclo si trova in una funzione. L'istruzione `continue` salta l'iterazione corrente e passa alla successiva.

Esempio:

```
x = 10

while x > 0:                                                    ①
  x -= 1

  if x == 5:
    break                                                       ②

  if x % 2 == 0:
    continue                                                    ③

  print(x)                                                      ④
```

① Inizia il ciclo `while` finché `x` è maggiore di `0`.
② Interrompe il ciclo quando `x` è uguale a `5`.
③ Salta l'iterazione corrente se `x` è pari.
④ Stampa il valore di `x` se non è stato saltato.

11.3.2. Istruzione `for`

L'istruzione `for` ripete l'esecuzione di un blocco di istruzioni controllata da un'espressione iterabile. La sintassi è:

```
for indice in iterabile:                                    ①
    istruzione(i)                                           ②
```

① `indice` è normalmente un identificatore che funge da variabile di controllo del ciclo e viene associato a ciascun elemento dell'`iterabile`.
② `istruzione(i)` rappresenta una o più istruzioni che vengono eseguite per ogni elemento dell'iterabile.

Esempio tipico di `for`:

```
for lettera in 'ciao':                                      ①
    print(f'Dammi una {lettera}...')                        ②
```

① Inizia il ciclo `for` su ogni carattere della stringa `'ciao'`.
② Stampa `"Dammi una [lettera]..."` per ogni lettera della stringa.

L'istruzione `for` può anche includere una clausola `else` che viene eseguita se il ciclo termina normalmente (cioè, non viene interrotto da un'istruzione `break` o `return`).

```
numeri = [1, 2, 3, 4, 5]

for numero in numeri:                                       ①
  if numero == 3:                                           ②
    break

  print(numero)                                             ③

else:                                                       ④
    print("Ciclo completato")                               ⑤
```

① Inizia il ciclo `for` sulla lista `numeri`.
② Se il numero è 3, interrompe il ciclo con `break`.
③ Stampa il numero corrente.
④ La clausola `else` viene eseguita se il ciclo `for` termina naturalmente.
⑤ Stampa "Ciclo completato".

L'iterabile può essere qualsiasi espressione iterabile in Python. In particolare, qualsiasi sequenza è iterabile. L'interprete chiama implicitamente la funzione built-in `iter()` sull'iterabile, producendo un iteratore.

11.3.3. Iteratori e iterabili

Un iteratore è un oggetto che permette di attraversare una collezione di elementi, uno alla volta. Gli iteratori sono utilizzati per rappresentare flussi di dati o collezioni di elementi che non sono necessariamente tutti disponibili in memoria contemporaneamente. Pertanto:

- Un oggetto è iterabile se può restituire un iteratore. Un oggetto iterabile implementa il metodo `__iter__()` che deve restituire un iteratore.
- Un oggetto è un iteratore se implementa i metodi `__iter__()` e `__next__()`. Il metodo `__next__()` restituisce il prossimo elemento della collezione e solleva l'eccezione `StopIteration` quando non ci sono più elementi.

Esempio di iteratore:

```
numeri = [1, 2, 3]                    ①
iteratore = iter(numeri)              ②

print(next(iteratore))                ③
print(next(iteratore))                ④
print(next(iteratore))                ⑤
print(next(iteratore))                ⑥
```

① Definizione di una lista di numeri.
② Creazione di un iteratore dall'iterabile numeri.
③ Output: `1`.
④ Output: `2`.
⑤ Output: `3`.
⑥ Solleva l'eccezione `StopIteration` perché non ci sono più elementi.

L'istruzione `for` chiama implicitamente `iter` sul suo iterabile per ottenere un iteratore. Lo pseudocodice seguente mostra cosa accade dietro le quinte:

```
_iteratore_temporaneo = iter(contenitore)

while True:
    try:
        x = next(_iteratore_temporaneo)

    except StopIteration:
        break

    istruzione(i)
```

11.3.4. La funzione `range`

La funzione `range` genera una sequenza di numeri interi. È comunemente usata nei cicli `for`.

```
for i in range(5):                    ①
  print(i)                            ②
```

① Inizia il ciclo `for` sui numeri da `0` a `4`.
② Stampa il numero corrente.

`range` può accettare fino a tre argomenti: `start`, `stop`, e `step`.

```
for i in range(1, 10, 2):             ①
  print(i)                            ②
```

① Inizia il ciclo `for` sui numeri da `1` a `9`, con incremento di `2`.
② Stampa il numero corrente.

L'oggetto `range` è un iterabile ma non un iteratore. Tuttavia, è possibile ottenere un iteratore chiamando `iter()` su un oggetto `range`.

```
r = range(5)
i = iter(r)

print(next(i))  ①
print(next(i))  ②
```

① Stampa il primo numero dell'oggetto `range`: `0`.
② Stampa il secondo numero dell'oggetto `range`: `1`.

11.3.5. Spacchettamento nei cicli `for`

È possibile utilizzare un indice composto da più identificatori, come in un'assegnazione con spacchettamento. In questo caso, gli elementi dell'iteratore devono essere essi stessi iterabili, ciascuno con esattamente tanti elementi quanti sono gli identificatori nell'indice.

Esempio di ciclo `for` su un dizionario:

```
d = {'a': 1, 'b': 2, 'c': 3}

for chiave, valore in d.items():  ①
  if chiave and valore:  ②
    print(chiave, valore)  ③
```

① Inizia il ciclo `for` sugli elementi del dizionario `d`.
② Verifica se sia la chiave che il valore sono "truthy".
③ Stampa la coppia chiave-valore.

È possibile usare un identificatore preceduto da un asterisco `*` nel target. Questo identificatore verrà associato a una lista di tutti gli elementi non assegnati ad altri target.

```
lista = [1, 2, 3, 4, 5]

for primo, *centro, ultimo in [lista]:  ①
  print(primo)  ②
  print(centro)  ③
  print(ultimo)  ④
```

① Inizia il ciclo `for` sulla lista `lista`.
② Stampa il primo elemento della lista.
③ Stampa tutti gli elementi centrali della lista.
④ Stampa l'ultimo elemento della lista.

11.4. Comprensioni

Le comprensioni *comprehension* in Python sono un modo conciso e leggibile per creare nuove sequenze (liste, insiemi, dizionari) da iterabili esistenti. Sono un esempio di supporto alla programmazione funzionale in quanto permettono di creare nuove collezioni attraverso la trasformazione e il filtraggio di elementi, senza modificare l'iterabile originale. Questa capacità di trasformare dati in modo dichiarativo, senza effetti collaterali, è un principio fondamentale della programmazione funzionale.

11.4.1. Liste

Le comprensioni di liste sono uno degli utilizzi più comuni. Permettono di ispezionare ogni elemento di un iterabile e costruire una nuova lista aggiungendo i risultati di un'espressione calcolata su alcuni o tutti gli elementi. Una compresa di lista ha la seguente sintassi:

```
[espressione for indice in iterabile clausole]
```

- `indice` e `iterabile` in ogni clausola `for` di una compresa di lista hanno la stessa sintassi e significato di quelli in una normale istruzione `for`.
- `espressione` può essere qualsiasi espressione Python valida e viene calcolata per ogni elemento della lista risultante.
- `clausole` è una serie di zero o più clausole, ciascuna con la forma `for indice in iterabile` o `if espressione`.

Esempio semplice:

```
result = [x + 1 for x in range(5)]                                    ①

print(result)                                                         ②
```

① Crea una lista di numeri incrementati di 1 da 0 a 4.
② Output: `[1, 2, 3, 4, 5]`.

Esempio con condizione:

```
result = [x + 1 for x in range(5) if x > 2]                           ①

print(result)                                                         ②
```

① Crea una lista con i numeri da 1 a 5, ma solo per i numeri maggiori di 2.
② Output: `[4, 5]`.

Esempio con annidamento:

```
lista_di_liste = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

result = [x for sublist in lista_di_liste for x in sublist]           ①
print(result)                                                         ②
```

① Appiattisce una lista di liste in una singola lista.
② Output: `[1, 2, 3, 4, 5, 6, 7, 8, 9]`.

11.4.2. Insiemi

Le comprensioni di insiemi hanno la stessa sintassi e semantica delle comprese di liste, ad eccezione che sono racchiuse tra parentesi graffe `{}` invece che tra parentesi quadre `[]`. Il risultato è un insieme.

Esempio:

```
s = {n // 2 for n in range(10)}                                    ①

print(sorted(s))                                                   ②
```

① Crea un insieme con i risultati della divisione intera di `n` per 2, da 0 a 9.
② Output: `[0, 1, 2, 3, 4]`.

11.4.3. Dizionari

Le comprensioni di dizionari hanno la stessa sintassi delle comprese di insiemi, ma invece di una singola espressione prima della clausola `for`, si usano due espressioni separate da un due punti `:` `chiave:valore`. Il risultato è un dizionario che mantiene l'ordine di inserimento.

Esempio:

```
d = {s: i for i, s in enumerate(['zero', 'uno', 'due'])}           ①

print(d)                                                           ②
```

① Crea un dizionario associando ogni stringa al suo indice nella lista.
② Output: `{'zero': 0, 'uno': 1, 'due': 2}`.

11.5. Gestione anomalie

11.5.1. Istruzioni `try` e `raise`

Python supporta la gestione delle eccezioni con l'istruzione composta `try`, che include le clausole `try`, `except`, `finally` ed `else`. Il codice può anche sollevare esplicitamente un'eccezione con l'istruzione `raise`. Quando il codice solleva un'eccezione, il normale flusso di controllo del programma si interrompe e Python cerca un gestore di eccezioni adatto.

Esempio di utilizzo di `try` e `raise`:

```
def dividi(a, b):                                                  ①
  try:                                                             ②
    return a / b                                                   ③

  except ZeroDivisionError:                                        ④
    raise ValueError("Divisione per zero non consentita")          ⑤

try:                                                               ⑥
  risultato = dividi(10, 0)                                        ⑦
```

```
except ValueError as e:                                                    ⑧
  print(e)                                                                 ⑨
```

① Definizione della funzione `dividi`.
② Inizia il blocco `try`.
③ Tentativo di divisione.
④ Gestione dell'eccezione `ZeroDivisionError`.
⑤ Sollevamento di una nuova eccezione `ValueError`.
⑥ Inizia un altro blocco `try`.
⑦ Tentativo di chiamare `dividi` con un denominatore pari a zero.
⑧ Gestione dell'eccezione `ValueError`.
⑨ Output del messaggio di errore: `Divisione per zero non consentita`.

11.5.2. Istruzione di controllo condizioni anomale

L'istruzione `assert condizione, messaggio` viene utilizzata per eseguire controlli durante l'esecuzione del programma. Se la `condizione` è falsa, viene sollevata un'eccezione `AssertionError` che include il `messaggio`. Può essere usata per il debugging e per verificare condizioni che dovrebbero essere sempre vere in un punto specifico del codice.

Esempi di utilizzo:

```
x = 5

assert x > 0, "x deve essere maggiore di zero"                             ①
assert x < 0, "x deve essere minore di zero"                               ②
```

① Non genera errore perché `x > 0` è vero.
② Genera `AssertionError` con il messaggio `"x deve essere minore di zero"` perché `x < 0` è falso.

È importante notare che le asserzioni possono essere disabilitate a livello di runtime utilizzando l'opzione `-O` (ottimizzazione) quando si esegue lo script Python. Questo rimuoverà tutte le istruzioni `assert` dal bytecode generato.

```
python -O script.py
```

11.6. Altre istruzioni

11.6.1. Istruzione `pass`

Il corpo di un'istruzione composta in Python non può essere vuoto; deve contenere almeno un'istruzione. Si può utilizzare l'istruzione `pass`, che non esegue alcuna azione, come segnaposto esplicito quando è richiesta un'istruzione sintatticamente ma non c'è nulla da fare.

Esempio di utilizzo di `pass`:

```
if True:                                                                    ①
  pass                                                                      ②
```

① Condizione sempre vera.
② Segnaposto che non esegue alcuna azione.

```
def funzione_non_implementata():                                            ①
  pass                                                                      ②
```

① Definizione di una funzione.
② Segnaposto per una funzione non ancora implementata.

```
class ClasseVuota:                                                          ①
  pass                                                                      ②
```

① Definizione di una classe.
② Segnaposto per una classe non ancora implementata.

11.6.2. Istruzione `with`

L'istruzione composta `with` può spesso essere un'alternativa più leggibile e utile all'istruzione `try/finally`. Essa consente di gestire risorse in modo efficiente e sicuro, assicurando che le risorse siano correttamente rilasciate dopo l'uso. Per essere gestita dall'istruzione `with`, una risorsa deve implementare il protocollo del context manager, che richiede i metodi speciali `__enter__` e `__exit__`.

Un esempio comune è l'uso di `with` per gestire i file:

```
with open('file.txt', 'r') as file:                                         ①
  contenuto = file.read()                                                   ②

  print(contenuto)                                                          ③
```

① Apre il file `file.txt` in modalità lettura.
② Legge il contenuto del file.
③ Output del contenuto del file.

Esempio di un contesto personalizzato:

```
class GestoreRisorsa:
  def __enter__(self):                                                      ①
    print("Risorsa acquisita")                                              ②

    return self

  def __exit__(self, tipo, valore, traceback):                              ③
    print("Risorsa rilasciata")                                             ④

with GestoreRisorsa() as risorsa:                                           ⑤
  print("Usando la risorsa")                                                ⑥
```

① Metodo `__enter__` che viene chiamato all'inizio del blocco `with`.
② Output indicante che la risorsa è stata acquisita.
③ Metodo `__exit__` che viene chiamato alla fine del blocco `with`, indipendentemente dal fatto che ci sia stata un'eccezione o meno.
④ Output indicante che la risorsa è stata rilasciata.
⑤ Inizia il blocco `with` usando il `GestoreRisorsa`.
⑥ Output indicante l'uso della risorsa.

11.6.3. Istruzioni di ritorno

Le istruzioni di ritorno vengono utilizzate per restituire valori da una funzione. Esistono due tipi principali di istruzioni di ritorno: `return` e `yield`.

11.6.3.1. `return`

L'istruzione `return` viene utilizzata per restituire un valore da una funzione e terminare l'esecuzione della funzione.

```
def somma(a, b):                                                    ①
  return a + b                                                      ②

print(somma(3, 4))                                                  ③
```

① Definizione della funzione `somma`.
② Restituisce la somma di `a` e `b`.
③ Output della somma: `7`.

11.6.3.2. `yield`

L'istruzione `yield` viene utilizzata per restituire un valore da una funzione generatore senza terminare l'esecuzione della funzione. La funzione può riprendere l'esecuzione dal punto in cui è stata interrotta al successivo ciclo di iterazione.

```
def generatore():                                                   ①
    yield 1                                                         ②
    yield 2                                                         ③
    yield 3                                                         ④

for valore in generatore():                                         ⑤
    print(valore)                                                   ⑥
```

① Definizione della funzione generatore `generatore`.
② Restituisce `1` e sospende l'esecuzione.
③ Restituisce `2` e sospende l'esecuzione.
④ Restituisce `3` e sospende l'esecuzione.
⑤ Itera sui valori restituiti dal generatore.
⑥ Output dei valori: `1`, `2`, `3`.

11.6.4. Modificatori di ambito

Le istruzioni `global` e `nonlocal` sono utilizzate per modificare la visibilità delle variabili all'interno delle funzioni.

L'istruzione `global` viene utilizzata per dichiarare che una variabile all'interno di una funzione fa riferimento a una variabile globale, cioè una variabile definita al di fuori di qualsiasi funzione. Senza `global`, tutte le assegnazioni di variabili all'interno di una funzione sono considerate locali alla funzione stessa.

Esempio di utilizzo di `global`:

```
x = 10                                          (1)

def modifica_global():
  global x                                      (2)

  x = 20

modifica_global()

print(x)                                        (3)
```

(1) `x` è una variabile globale.
(2) L'istruzione `global` dichiara che `x` fa riferimento alla variabile globale `x`, permettendo alla funzione di modificarla.
(3) Output: 20.

L'istruzione `nonlocal` viene utilizzata per dichiarare che una variabile all'interno di una funzione fa riferimento a una variabile non locale, cioè una variabile definita in un contesto esterno ma non globale (ad esempio, all'interno di una funzione contenente). Senza `nonlocal`, tutte le assegnazioni di variabili all'interno di una funzione sono considerate locali alla funzione stessa.

Esempio di utilizzo di `nonlocal`:

```
def esterna():
  x = 10                                        (1)

  def interna():
    nonlocal x                                  (2)

    x = 20

  interna()

  print(x)

esterna()                                       (3)
```

(1) `x` è una variabile locale alla funzione `esterna`.
(2) L'istruzione `nonlocal` dichiara che `x` fa riferimento alla variabile non locale `x`, permettendo alla funzione `interna` di modificarla.
(3) Output: 20.

11.6.5. Alias di tipo

L'istruzione `type` viene utilizzata per creare alias di tipo. Questo consente di assegnare nomi significativi ai tipi di dati complessi, migliorando la leggibilità del codice. Si possono definire anche alias di tipi generici, cioè tipi parametrizzati da altri tipi.

È importante sottolineare che gli alias non sono da intendere come utili al controllo statico dei tipi durante l'esecuzione, ma come annotazione utile per strumenti di analisi del codice e miglioramento della leggibilità.

Esempi:

- Definizione di alias di tipo:

```
type lista_coppie = list[tuple[str, int]]                    ①
```

① `lista_coppie` è un alias di `list[tuple[str, int]]`.

- Definizione di alias di tipo generico:

```
type contenitore_ordinato[T] = list[t] | tuple[T, ...]       ①
```

① `contenitore_ordinato` può essere o una lista o una tupla con zero o più elementi.

11.6.6. Eliminazione di identificatori e elementi in contenitori

L'istruzione `del` viene utilizzata per eliminare identificatori come variabili o di funzione e oggetti da contenitori, come elementi singoli o in sezioni (slice) di liste oppure chiavi di dizionari. `del` rimuovendo i riferimenti agli oggetti, segnala al garbage collector la possibilità di liberazione delle risorse associate.

Esempi di utilizzo:

- Eliminazione di una variabile:

```
x = 10

del x                                                        ①

print(x)                                                     ②
```

① Elimina la variabile `x`.
② Dà errore perché `x` non esiste.

- Eliminazione di un elemento da un contenitore di oggetti di tipo lista per indice:

```
lista = [1, 2, 3, 4]

del lista[0]                                                 ①

print(lista)                                                 ②
```

① Elimina il primo elemento dalla lista.
② Output: `[2, 3, 4]`.

- Eliminare di elementi contigui da una lista:

```
lista = [1, 2, 3, 4, 5, 6]

del lista[1:4]                                                    ①

print(lista)                                                      ②
```

① Elimina gli elementi dal secondo al quarto.
② Output: `[1, 5, 6]`.

- Eliminazione di una chiave da un dizionario:

```
dizionario = {'a': 1, 'b': 2, 'c': 3}

del dizionario['a']                                               ①

print(dizionario)                                                 ②
```

① Elimina la chiave `'a'` dal dizionario.
② Output: `{'b': 2, 'c': 3}`.

- Eliminazione di un attributo da un oggetto:

```
class ClasseSemplice:
  def __init__(self):
    self.attr = 42

oggetto_semplice = ClasseSemplice()

del oggetto_semplice.attr                                         ①
```

① Elimina l'attributo `'attr'` dall'oggetto `'oggetto_semplice'`.

- Eliminazione dell'identificatore di una funzione:

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

del somma_semplice                                                ①

somma_semplice(a, b)                                              ②
```

① Elimina il riferimento `somma_semplice` alla funzione.
② Errore. Output: `NameError: name 'somma_semplice' is not defined`.

- Eliminazione dell'identificatore di un modulo:

```
import math

print(math.sqrt(4))                                               ①

del math
```

```
print(math)                                                                    ②
```

① Output: `2.0`.
② Errore. Output: `NameError: name 'math' is not defined. Did you forget to import 'math'?`.

12. Esercizi

Legenda livelli:

Neofita: Adatto a chi è alle prime armi con la programmazione. Gli esercizi di questo livello richiedono una conoscenza basilare della sintassi di Python e dei concetti fondamentali come variabili, semplici espressioni, dimistichezza coll'esecuzione dell'interprete.

Principiante: Gli esercizi a questo livello sono pensati per chi ha familiarità con i costrutti di base di Python e vuole iniziare a esplorare le strutture dati come liste, tuple e dizionari.

Principiante evoluto: Questi esercizi richiedono una comprensione più approfondita dei costrutti disponibili e delle operazioni sulle strutture dati fornite dal linguaggio. Gli studenti dovrebbero essere in grado di scrivere funzioni e manipolare collezioni di dati, usando la documentazione del linguaggio.

Autonomo: A questo livello, gli studenti devono saper gestire concetti come il controllo del workflow di esecuzione per mezzo delle eccezioni, l'uso di moduli e pacchetti standard, nonché di quelli esterni al linguaggio. Gli studenti devono saper effettuare un debugging, fornire codice documentato e più organizzato e modulare.

Intermedio: Gli esercizi richiedono la capacità di lavorare con librerie esterne, creare e gestire oggetti complessi, e utilizzare tecniche di programmazione più avanzate come le comprensioni di lista e le espressioni lambda, sapendo applicare pienamente sia lo stile orientato agli oggetti che quello funzionale. Gli studenti devono produrre codice robusto per mezzo di tecniche come i test di unità.

Esperto: A questo livello, gli esercizi implicano la padronanza di concetti avanzati come il decoratori, i generatori, e la manipolazione avanzata dei dati. Gli studenti dovrebbero essere in grado di risolvere problemi complessi con soluzioni eleganti.

Esperto evoluto: Gli studenti devono avere competenze solide di disegno di software basato sul paradigma dell'orientamento agli oggetti e funzionale, nonché nell'ottimizzazione del codice per la performance.

Maestro: Gli esercizi a questo livello richiedono la conoscenza approfondita di Python, comprese le tecniche di programmazione asincrona, il threading e la gestione di progetti di grandi dimensioni in diversi ambiti.

Maestro evoluto: A questo livello, gli studenti devono padroneggiare la creazione di librerie proprie anche usando le modalità di estensione di Python.

Guru: Questo è il livello più alto di difficoltà, dove gli esercizi richiedono una comprensione approfondita e una padronanza assoluta di Python. Gli studenti devono essere in grado di risolvere problemi estremamente complessi, ottimizzare il codice a livello di prestazioni e memoria, e applicare concetti avanzati di ingegneria del software.

12.1. Python come calcolatrice

Primi esperimenti con Python.

12.1.1. Numeri interi e in virgola mobile

12.2. Problema

Usare gli operatori matematici su costanti numeriche e osservare i risultati e gli errori nel REPL, perché è più immediato rispetto all'esecuzione completa del programma e permette di prendere dimistichezza velocemente con dei costrutti di base del linguaggio.

12.3. Soluzione

Il codice seguente può essere eseguito sia nel REPL, riga per riga, sia come programma.

> Suggerimento
>
> Usando il REPL, basterà digitare l'espressione senza assegnamento per ottenere il risultato.

```
# Moltiplicazione
x = 5 * 2
print(x)

x = 5 * 2.
print(x) # Cosa notiamo?

# Divisione in virgola mobile
x = 5 / 2
print(x)

x = 4 / 2
print(x) # Cosa notiamo?

x = 4 / 2.
print(x)

# Confronto
x = 5 > 2
print(x)

x = 5 > 2.
print(x) # Cosa notiamo?

# Diversità
x = 4 != 4.
print(x) # Cosa notiamo?

x = 0 != (1 - 1)
print(x) # Cosa notiamo?
```

12.3.1. Stringhe

12.4. Problema

Usare gli operatori su stringhe, sempre nel REPL.

12.5. Soluzione

```
s = "Hello" + ' ' + 'World!'
print(s)

ss = s

ss *= 2
print(ss)
print(s) # Cosa notiamo per s e ss?

# Appartenenza
b = 'el' in s
print(b)

b = 'oo' not in s
print(b)

# Confronto
b = "Ciao Mondo!" < s
print(b) # È rispettato l'ordine lessicografico?

l_s= len(s)
print(l_s)

# Slicing della stringa come contenitore di caratteri
s_ = ss[:l_s]
print(s_)

l_ss = len(ss)
print(l_ss)

# Modo alternativo di ottenere la stringa originale solo usando ss
s_ = ss[:int(l_ss / 2)]
print(s_)

# Metodo per rendere la stringa in maiuscolo
su = s.upper()
print(su)

# Uguaglianza
```

```
b = s == su
print(b) # Cosa notiamo?
```

12.5.1. Espressioni

12.6. Problema

Costruire delle espressioni per comprendere come mischiare numeri e stringhe, la precedenza degli operatori e le conversioni di tipo, sempre nel REPL.

12.7. Soluzione

```
n = 42
s = "42"

# Congiunzione
b = n and s
print(b) # Cosa notiamo?

# Disgiunzione
b = n or s
print(b)

# Negazione e congiunzione
b = n and not s
print(b) # Cosa notiamo?

# Conversione di tipo in stringa e appartenenza
b = str(2) in s
print(b)

# Conversione di tipo in intero e divisione
b = int(s) / 2
print(b)

# Espressione con precedenza data dall'ordine degli operatori
e = 2 + n * 3
print(e)

# Espressione con precedenza modificata colle parentesi
e = (2 + n) * 3
print(e) # Cosa notiamo?
```

12.8. Numeri pari o dispari

Definire una funzione che prende in input un numero intero e restituisce una stringa di `Pari` o `Dispari`.

12.8.1. Riscaldamento

12.9. Problema

Sperimentiamo l'operatore modulo `%`, che restituisce il resto della divisione di due interi, con diversi input sia pari che dispari usando un test condizionale.

12.10. Soluzione

```
n = 42

if n % 2 == 0:
  print("Pari")

else:
  print("Dispari")
```

12.10.1. Svolgimento

12.11. Problema

Inserire le istruzioni in una funzione che prende in input un parametro, il numero intero, e restituisce una stringa, `Pari` o `Dispari`. Sperimentare soluzioni diverse.

12.12. Soluzione 1

Usiamo l'operatore modulo `%` che restituisce il resto della divisione di due interi all'interno di una funzione. Questa prende in input un numero intero e restituisce la stringa richiesta.

```
def pari_o_dispari(n):
  if n % 2 == 0:
    return "Pari"

  else:
    return "Dispari"

risultato = pari_o_dispari(42)

print(risultato)
```

```
risultato = pari_o_dispari(73)

print(risultato)
```

12.13. Soluzione 2

Usiamo l'operatore modulo `%` per il test di parità sul numero intero e la funzione `isinstance` per verificare il tipo in input.

```
def pari_o_dispari(n):
  if not isinstance(n, int):
    return "Errore: l'input deve essere un numero intero!"

  if n % 2 == 0:
    return "Pari"

  else:
    return "Dispari"

risultato = pari_o_dispari(42)

print(risultato)

risultato = pari_o_dispari(73)

print(risultato)
```

12.14. Soluzione 3

Usiamo l'operatore modulo `%`, la funzione `isinstance` per verificare il tipo in input e `assert` in caso di input non corretto.

```
def pari_o_dispari(n):
  assert isinstance(n, int), \
    "Errore: l'input deve essere un numero intero!"

  if n % 2 == 0:
    return "Pari"

  else:
    return "Dispari"

risultato = pari_o_dispari(42)

print(risultato)
```

```
risultato = pari_o_dispari(73)

print(risultato)

'''
risultato = pari_o_dispari("42")

print(risultato)

risultato = pari_o_dispari(73.)

print(risultato)
'''
```

12.15. Soluzione 4

Usiamo la funzione `divmod` che restituisce il quoziente e il resto della divisione di due interi. Per ottenere documentazione su essa basterà digitare `help(divmod)` nel REPL.

```
def pari_o_dispari(n):
  _, remainder = divmod(n, 2)

  return "Pari" if remainder == 0 else "Dispari"

risultato = pari_o_dispari(42)

print(risultato)

risultato = pari_o_dispari(73)

print(risultato)
```

12.16. Rimozione di duplicati da una lista preservando l'ordinamento

12.17. Problema

Scrivere una funzione che prende in input una lista e ne rimuove i duplicati, preservando l'ordinamento.

12.18. Soluzione 1

Usiamo un ciclo `for` per iterare attraverso la lista originale e una lista di appoggio per memorizzare gli elementi unici. Gli elementi vengono aggiunti alla lista di appoggio solo se non sono già presenti in essa, preservando così l'ordine originale.

```
def rimuovi_duplicati(lista):
  lista_senza_duplicati = []

  for elemento in lista:
    if elemento not in lista_senza_duplicati:
      lista_senza_duplicati.append(elemento)

  return lista_senza_duplicati

# Esempio di utilizzo
lista = [4, 2, 2, 3, 1, 4, 5]

print(rimuovi_duplicati(lista))
```

12.19. Soluzione 2

Usiamo un ciclo while per iterare attraverso la lista originale. Un `set` viene utilizzato per memorizzare gli elementi già visti e una lista di appoggio per memorizzare gli elementi unici. Gli elementi vengono aggiunti alla lista di appoggio solo se non sono già presenti nel `set`.

```
def rimuovi_duplicati(lista):
  lista_senza_duplicati = []

  visti = set()

  i = 0
  while i < len(lista):
    if lista[i] not in visti:
      lista_senza_duplicati.append(lista[i])

      visti.add(lista[i])

    i += 1

  return lista_senza_duplicati

# Esempio di utilizzo
lista = [4, 2, 2, 3, 1, 4, 5]

print(rimuovi_duplicati(lista))
```

12.20. Soluzione 3

Usiamo un dizionario per memorizzare gli elementi unici, sfruttando il fatto che i dizionari preservano l'ordine di inserimento a partire da Python 3.7. Gli elementi vengono aggiunti al dizionario come chiavi, e infine si restituisce la lista delle chiavi del dizionario.

```
def rimuovi_duplicati(lista):
  return list(dict.fromkeys(lista))

# Esempio di utilizzo
lista = [4, 2, 2, 3, 1, 4, 5]

print(rimuovi_duplicati(lista))
```

12.21. Soluzione 4

Utilizziamo una list comprehension per creare una nuova lista. Un `set` viene usato per tenere traccia degli elementi già visti, e gli elementi vengono aggiunti alla lista finale solo se non sono già presenti nel `set`.

```
def rimuovi_duplicati(lista):
    visti = set()

    lista_senza_duplicati = [elemento
                             for elemento in lista
                             if elemento not in visti and not visti.add(elemento)]

    return lista_senza_duplicati

# Esempio di utilizzo
lista = [4, 2, 2, 3, 1, 4, 5]

print(rimuovi_duplicati(lista))
```

12.22. Rimozione di duplicati da una lista e ordinamento

12.23. Problema

Scrivere una funzione che prende in input una lista e ne rimuove i duplicati, ordinando il risultato.

12.24. Soluzione 1

Usiamo un ciclo `for` per iterare attraverso la lista originale e una lista di appoggio per memorizzare gli elementi unici. Gli elementi vengono aggiunti alla lista di appoggio solo se non sono già presenti in essa. Dopo aver rimosso i duplicati, ordiniamo la lista risultante.

```
def rimuovi_duplicati(lista):
  lista_senza_duplicati = []

  for elemento in lista:
    if elemento not in lista_senza_duplicati:
```

```
      lista_senza_duplicati.append(elemento)

  return sorted(lista_senza_duplicati)

# Esempio di utilizzo
lista = [4, 2, 2, 3, 1, 4, 5]

print(rimuovi_duplicati(lista))
```

12.25. Soluzione 2

Usiamo un ciclo `while` per iterare attraverso la lista originale. Un `set` viene utilizzato per memorizzare gli elementi già visti e una lista di appoggio per memorizzare gli elementi unici. Gli elementi vengono aggiunti alla lista di appoggio solo se non sono già presenti nel `set`. Dopo aver rimosso i duplicati, ordiniamo la lista risultante.

```
def rimuovi_duplicati(lista):
  lista_senza_duplicati = []
  visti = set()

  i = 0
  while i < len(lista):
    if lista[i] not in visti:
      lista_senza_duplicati.append(lista[i])

      visti.add(lista[i])

    i += 1

  return sorted(lista_senza_duplicati)

# Esempio di utilizzo
lista = [4, 2, 2, 3, 1, 4, 5]

print(rimuovi_duplicati(lista))
```

12.26. Soluzione 3

Usiamo un dizionario per memorizzare gli elementi unici, sfruttando il fatto che i dizionari preservano l'ordine di inserimento a partire da Python 3.7. Gli elementi vengono aggiunti al dizionario come chiavi. Dopo aver rimosso i duplicati, ordiniamo la lista delle chiavi del dizionario.

```
def rimuovi_duplicati(lista):
  return sorted(dict.fromkeys(lista))

# Esempio di utilizzo
```

```
lista = [4, 2, 2, 3, 1, 4, 5]

print(rimuovi_duplicati(lista))
```

12.27. Soluzione 4

Utilizziamo una list comprehension per creare una nuova lista. Un `set` viene usato per tenere traccia degli elementi già visti, e gli elementi vengono aggiunti alla lista finale solo se non sono già presenti nel `set`. Dopo aver rimosso i duplicati, ordiniamo la lista risultante.

```
def rimuovi_duplicati(lista):
  visti = set()

  lista_senza_duplicati = [x for x in lista if not (x in visti or visti.add(x))]

  return sorted(lista_senza_duplicati)

# Esempio di utilizzo
lista = [4, 2, 2, 3, 1, 4, 5]

print(rimuovi_duplicati(lista))
```

12.28. Calcolo del fattoriale di un numero

12.29. Problema

Scrivere una funzione che prende in input un numero intero positivo e restituisce il suo fattoriale.

> Suggerimento
>
> Il fattoriale di un numero n è il prodotto di tutti i numeri interi positivi minori o uguali a n ed è denotato come n!.

12.30. Soluzione 1

Usiamo un ciclo `for` per calcolare il fattoriale. Partiamo da 1 e moltiplichiamo progressivamente tutti i numeri fino a n.

```
def fattoriale(n):
  risultato = 1

  for i in range(1, n + 1):
    risultato *= i
```

```
  return risultato

# Esempio di utilizzo
numero = 5

print(fattoriale(numero))                                                    ①
```

① Output: 120.

12.31. Soluzione 2

Usiamo un ciclo `while` per calcolare il fattoriale. Partiamo da 1 e moltiplichiamo progressivamente tutti i numeri fino a n, utilizzando una variabile di iterazione.

```
def fattoriale(n):
  risultato = 1
  i = 1

  while i <= n:
    risultato *= i

    i += 1

  return risultato

# Esempio di utilizzo
numero = 5

print(fattoriale(numero))                                                    ①
```

① Output: 120.

12.32. Soluzione 3

Utilizziamo la ricorsione per calcolare il fattoriale. La funzione richiama se stessa riducendo il problema fino a raggiungere il caso base $n = 1$.

```
def fattoriale(n):
  if n == 0 or n == 1:
    return 1

  else:
    return n * fattoriale(n - 1)

# Esempio di utilizzo
numero = 5
```

```
print(fattoriale(numero))                                                    ①
```

① Output: 120.

12.33. Soluzione 4

Usiamo la funzione `reduce` del modulo `functools` per calcolare il fattoriale. Questa soluzione utilizza un approccio funzionale per ridurre una sequenza di numeri a un singolo valore.

```
from functools import reduce

def fattoriale(n):
  return reduce(lambda x, y: x * y, range(1, n + 1), 1)

# Esempio di utilizzo
numero = 5

print(fattoriale(numero))                                                    ①
```

① Output: 120.

12.34. Contare le parole in una frase in modo semplificato

12.35. Problema

Scrivere una funzione che prende in input una stringa contenente una frase e restituisce un dizionario con il conteggio di ciascuna parola nella frase. Le frasi non devono contenere punteggiatura e il confronto tiene conto della differenza tra lettere maiuscole e minuscole.

12.36. Soluzione 1

Usiamo un ciclo `for` per iterare attraverso le parole della frase, aggiornando il conteggio di ciascuna parola in un dizionario.

```
def conta_parole(frase):
  parole = frase.split()
  conteggio = {}

  for parola in parole:
    if parola in conteggio:
      conteggio[parola] += 1

    else:
      conteggio[parola] = 1
```

```
  return conteggio

# Esempio di utilizzo
frase = "ciao ciao come stai ciao"

print(conta_parole(frase))                                                    ①
```

1. Output: `{'ciao': 3, 'come': 1, 'stai': 1}`.

12.37. Soluzione 2

Usiamo il metodo `get` del dizionario per aggiornare il conteggio delle parole in un dizionario.

```
def conta_parole(frase):
  parole = frase.split()
  conteggio = {}

  for parola in parole:
    conteggio[parola] = conteggio.get(parola, 0) + 1
  return conteggio

# Esempio di utilizzo
frase = "ciao ciao come stai ciao"

print(conta_parole(frase))
```

12.38. Soluzione 3

Usiamo il modulo `collections` e il `defaultdict` per semplificare il conteggio delle parole.

```
from collections import defaultdict

def conta_parole(frase):
  parole = frase.split()
  conteggio = defaultdict(int)

  for parola in parole:
    conteggio[parola] += 1

  return dict(conteggio)

# Esempio di utilizzo
frase = "ciao ciao come stai ciao"

print(conta_parole(frase))
```

12.39. Soluzione 4

Usiamo il modulo `collections` e `Counter` per contare le parole nella frase in modo conciso.

```
from collections import Counter

def conta_parole(frase):
    parole = frase.split()

    conteggio = Counter(parole)

    return dict(conteggio)

# Esempio di utilizzo
frase = "ciao ciao come stai ciao"

print(conta_parole(frase))
```

12.40. Contare le parole in una frase con esattezza

12.41. Problema

Scrivere una funzione che prende in input una stringa contenente una frase e un flag `maiuscolo_minuscolo` che controlla se il conteggio delle parole debba tener conto del maiuscolo o minuscolo. Inoltre considera parole senza terner conto di eventuale punteggiatura nel calcolo. La funzione restituisce un dizionario con il conteggio di ciascuna parola nella frase.

> Suggerimento
>
> Si può usare `string.punctuation` del modulo `string` che contiene tutti i caratteri di punteggiatura disponibili in Python. Include caratteri come `!"#$%&'()*+,-./:;<=>?@[\]^_``{|}~`.

12.42. Soluzione 1

Usiamo un ciclo `for` per iterare attraverso le parole della frase, aggiornando il conteggio di ciascuna parola in un dizionario. Il comportamento è determinato dal flag `maiuscolo_minuscolo`.

```
import string

def conta_parole(frase, maiuscolo_minuscolo=False):
  if not maiuscolo_minuscolo:
    frase = frase.lower()

  frase = ''.join(carattere for carattere in frase if carattere not in string.punctuation)

  parole = frase.split()
```

```
  conteggio = {}

  for parola in parole:
    if parola in conteggio:
      conteggio[parola] += 1

    else:
      conteggio[parola] = 1

  return conteggio

# Esempio di utilizzo
frase = "Ciao, ciao! Come stai? Ciao."

print(conta_parole(frase, maiuscolo_minuscolo=False))                    ①
print(conta_parole(frase, maiuscolo_minuscolo=True))                     ②
```

① Output: `{'ciao': 3, 'come': 1, 'stai': 1}`.
② Output: `{'Ciao': 2, 'ciao': 1, 'Come': 1, 'stai': 1}`.

12.43. Soluzione 2

Usiamo il metodo `get` del dizionario per aggiornare il conteggio delle parole in un dizionario. Il comportamento è determinato dal flag `maiuscolo_minuscolo`.

```
import string

def conta_parole(frase, maiuscolo_minuscolo=False):
  if not maiuscolo_minuscolo:
    frase = frase.lower()

  frase = ''.join(carattere for carattere in frase if carattere not in string.punctuation)

  parole = frase.split()

  conteggio = {}

  for parola in parole:
    conteggio[parola] = conteggio.get(parola, 0) + 1

  return conteggio

# Esempio di utilizzo
frase = "Ciao, ciao! Come stai? Ciao."

print(conta_parole(frase, maiuscolo_minuscolo=False))
print(conta_parole(frase, maiuscolo_minuscolo=True))
```

12.44. Soluzione 3

Usiamo il modulo `collections` e il `defaultdict` per semplificare il conteggio delle parole. Il comportamento è determinato dal flag `maiuscolo_minuscolo`.

```
import string
from collections import defaultdict

def conta_parole(frase, maiuscolo_minuscolo=False):
  if not maiuscolo_minuscolo:
    frase = frase.lower()

  frase = ''.join(carattere for carattere in frase if carattere not in string.punctuation)

  parole = frase.split()

  conteggio = defaultdict(int)

  for parola in parole:
    conteggio[parola] += 1

  return dict(conteggio)

# Esempio di utilizzo
frase = "Ciao, ciao! Come stai? Ciao."

print(conta_parole(frase, maiuscolo_minuscolo=False))
print(conta_parole(frase, maiuscolo_minuscolo=True))
```

12.45. Soluzione 4

Usiamo il modulo `collections` e `Counter` per contare le parole nella frase in modo conciso. Il comportamento è determinato dal flag `maiuscolo_minuscolo`.

```
import string
from collections import Counter

def conta_parole(frase, maiuscolo_minuscolo=False):
  if not maiuscolo_minuscolo:
    frase = frase.lower()

  frase = ''.join(carattere for carattere in frase if carattere not in string.punctuation)

  parole = frase.split()

  conteggio = Counter(parole)

  return dict(conteggio)
```

```
# Esempio di utilizzo
frase = "Ciao, ciao! Come stai? Ciao."

print(conta_parole(frase, maiuscolo_minuscolo=False))
print(conta_parole(frase, maiuscolo_minuscolo=True))
```

Riferimenti

Kernighan, Brian W. 1973. «A Tutorial Introduction to the Programming Language B». Murray Hill, NJ: Bell Laboratories.

Kernighan, Brian W., e Dennis M. Ritchie. 1988. *The C Programming Language*. 2nd ed. Englewood Cliffs, NJ: Prentice Hall.

Stone, Harold S. 1971. *Introduction to Computer Organization and Data Structures*. USA: https://dl.acm.org/doi/10.5555/578826; McGraw-Hill, Inc.

Stroustrup, Bjarne. 2013. *The C++ Programming Language*. 4th ed. https://dl.acm.org/doi/10.5555/2543987; Addison-Wesley Professional.