Da neofita di Python a campione

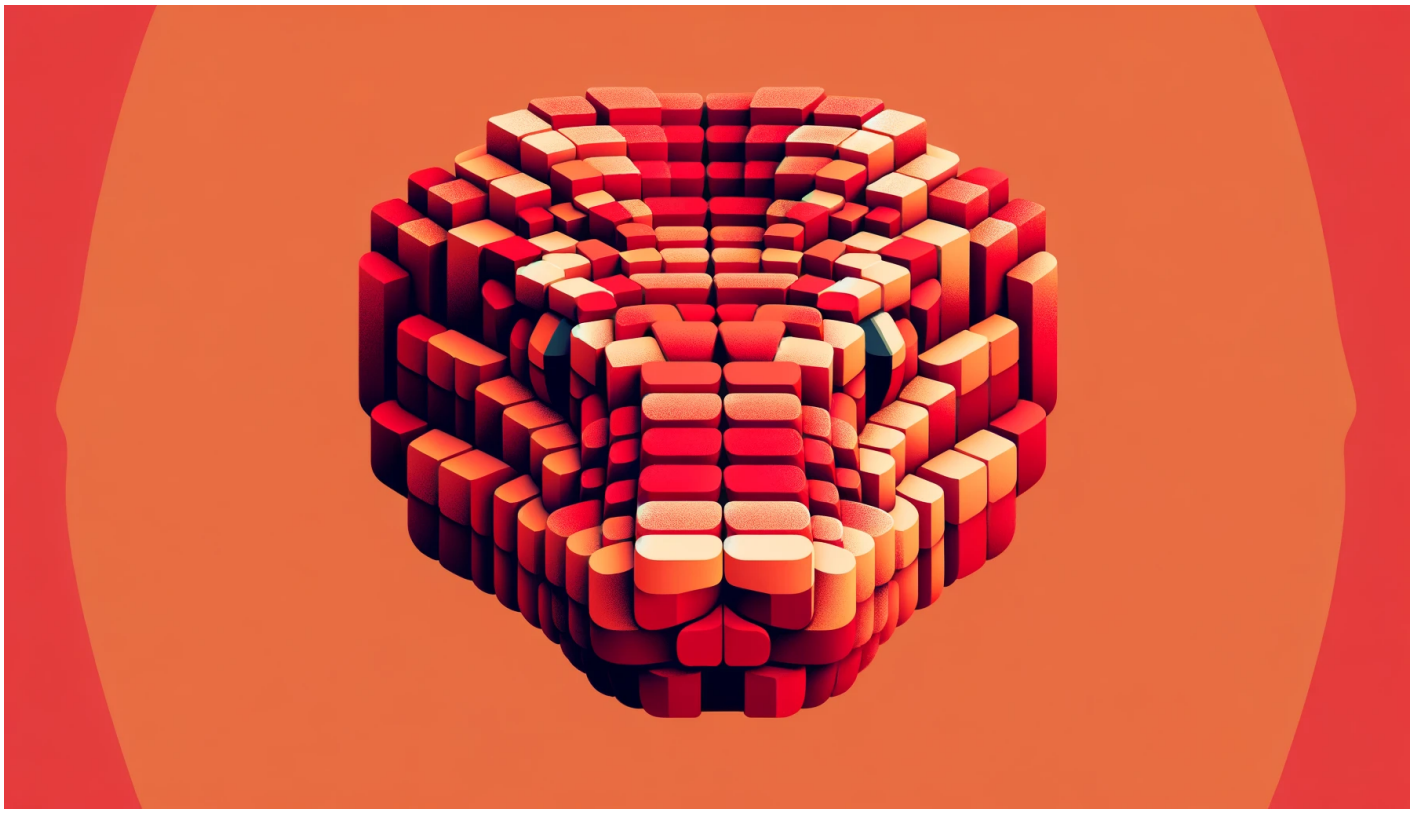
Corso di Python per tutti

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 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 seguire 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 differiscono dai linguaggi naturali (come l'italiano o l'inglese) in diversi modi:

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.

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)). Gli algoritmi sono alla base della programmazione perché rappresentano il disegno teorico computazionale dei programmi.

Più precisamente, un algoritmo è una sequenza ben definita di passi o operazioni che, a partire da un input, produce un output in un tempo finito. Le proprietà principali seguenti esprimono in modo più completo le caratteristiche che un algoritmo deve possedere:

- 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, 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 è, quindi: 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 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. 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. 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:

```
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:

```
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.

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

- File header (`mialibreria.h`):

```
#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`):

```
#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`):

```
#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:

```
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 float), ma è possibile anche sommare altri tipi di elementi se l'operazione di addizione è definita per quel tipo.

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, 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:

```
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);
  }
```

[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.

```
  @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:

```rust
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:

```cpp
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. 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.

3.2.1. 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):

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)`.

3.3. Istruzioni semplici

Le **istruzioni semplici** sono operazioni atomiche secondo il linguaggio e sono costituite da lessemi ed espressioni per compiere operazioni di base. Gli esempi principali includono:

- Espressioni: È eseguibile dal compilatore o interprete quindi è una delle istruzioni semplici più importanti quando a sé stante. Sono presenti all'interno di istruzioni semplici e composte.
- 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. Esempio:

```
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`.
  - `+`: 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.

3.4. 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:

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

  y = x * 2;
}
```

In questo esempio:

- `if (x > 0)` è un'istruzione composta.
- `{ 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:
  print("x è positivo")

  y = x * 2
```

In questo esempio:

- `if x > 0:` è un'istruzione composta.
- 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.

Altri esempi:

- Condizioni: Istruzioni che eseguono un blocco di codice solo se una condizione è vera. Esempio:

```
#include <stdio.h>

x = 42;

if (x > 0) {
    printf("x è positivo\n");
}
```

- `if`: Parola chiave che introduce la condizione.
- `(x > 0)`: Condizione composta da: `x` identificatore di variabile, `>` operatore di confronto e `0` letterale numerico intero.
- `{ ... }`: Delimitatori che racchiudono il blocco di codice.
- `print("x è positivo")`: Istruzione di output.

- Cicli: Istruzioni che ripetono un blocco di codice. Esempio:

```
#include <stdio.h>

int n = 42;
int somma = 0;
int i;

for (i = 0; i < n; i++) {
    somma = somma + i;
}
```

- `for`: Parola chiave che introduce il ciclo.
- `(i = 0; i < n; i++)`: Espressione di controllo del ciclo composta da: `i = 0` assegnazione iniziale, `i < n` condizione di ciclo e `i++` incremento della variabile `i`.
- `{ ... }`: Delimitatori che racchiudono il blocco di codice.
- `somma = somma + i`: Operazione aritmetica.

3.5. Organizzazione del codice 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.

4. Variabili e 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 gli elementi semantici più importanti, troviamo le variabili e le funzioni.

4.1. Variabili

- Variabile: È un nome simbolico associato a locazione di memoria che può contenere uno o più valori. È fondamentale per la manipolazione di dati perché sono un mezzo per astrarre dalla costante memorizzata. Le variabili possono essere associate a diversi tipi di dati e durate di vita. La semantica delle variabili include la loro dichiarazione, inizializzazione, uso e visibilità:
  - Dichiarazione: 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. La dichiarazione informa il compilatore o l'interprete che una certa variabile esiste e può essere utilizzata nel codice.
  - Inizializzazione: L'inizializzazione di una variabile è il processo di assegnare un valore iniziale alla variabile. L'inizializzazione può avvenire contestualmente alla dichiarazione o in un'istruzione separata successiva.
  - Visibilità: Indica dove la variabile può essere utilizzata all'interno del codice (ad esempio, variabili locali o globali).
  - Durata di Vita: Descrive per quanto tempo la variabile rimane in memoria durante l'esecuzione del programma (ad esempio, automatica, statica, dinamica).
- Ambito (in inglese, *scope*): L'ambito rappresenta la porzione del codice in cui un identificatore (come una variabile o una funzione) è definito e, quindi, esiste. L'ambito determina dove un identificatore può essere dichiarato e utilizzato. Tipicamente gli ambiti sono:
  - Globale: Identificatori dichiarati a livello globale, accessibili ovunque nel programma.
  - Locale: Identificatori dichiarati all'interno di un blocco, come una funzione o un loop, e accessibili solo all'interno di quel blocco.
  - Statico e dinamico: L'ambito statico è determinato a tempo di compilazione, mentre l'ambito dinamico è determinato a runtime, influenzando come e dove gli identificatori possono essere utilizzati.
- Visibilità: La visibilità si riferisce a dove nel codice un identificatore può essere visto e utilizzato. Anche se correlata all'ambito, la visibilità può essere influenzata da altri fattori come la modularità e i namespace, che organizzano e separano gli identificatori per evitare conflitti di nome. La visibilità è generalmente:

  - Globale: Un identificatore dichiarato con visibilità globale può essere utilizzato in qualsiasi parte del programma.
  - Locale: Un identificatore dichiarato con visibilità locale è visibile solo all'interno del blocco di codice in cui è stato dichiarato.

4.2. Funzioni

- Funzioni e metodi: Le funzioni e i metodi sono blocchi di codice riutilizzabili che eseguono una serie di istruzioni. Alcuni concetti collegati sono:
  - Parametri e argomenti: Valori passati alle funzioni per influenzarne il comportamento. I parametri sono definiti nella dichiarazione della funzione, mentre gli argomenti sono i valori effettivi passati quando la funzione è chiamata.
  - Valore di ritorno: Il risultato prodotto da una funzione, che può essere utilizzato nell'istruzione chiamante.
  - Overloading: Definizione di più funzioni con lo stesso nome ma diversi parametri, consentendo diverse implementazioni basate sui tipi e il numero di argomenti.
  - Ricorsione: Capacità di una funzione di chiamare se stessa, utile per risolvere problemi che possono essere suddivisi in sottoproblemi simili.
  - Funzioni di prima classe: Le funzioni possono essere assegnate a variabili, passate come argomenti e ritornate da altre funzioni.
  - Funzioni di ordine superiore: Funzioni che accettano altre funzioni come argomenti e/o ritornano funzioni come risultati.
- Durata di vita delle variabili: La durata di vita delle variabili si riferisce a quanto tempo una variabile rimane in memoria durante l'esecuzione del programma. Alcune tipologie di durata:
  - Automatica: Variabili che esistono solo durante l'esecuzione del blocco in cui sono dichiarate.
  - Statica: Variabili che esistono per tutta la durata del programma e mantengono il loro valore tra diverse chiamate di funzione.
  - Dinamica: Variabili allocate dinamicamente durante l'esecuzione del programma, solitamente gestite manualmente dall'utente (ad esempio, usando `malloc/free` in C) o automaticamente tramite garbage collection.
- Durata di vita di altri identificatori:
  - Funzioni: Le funzioni stesse generalmente hanno una durata di vita che coincide con la durata del programma. Tuttavia, i puntatori a funzione e le chiusure (in inglese, closures) possono avere durate di vita diverse in alcuni linguaggi.
  - Classi e oggetti: Le classi hanno una durata di vita che coincide con la durata del programma, mentre gli oggetti (istanze di classi) hanno durate di vita dinamiche, determinate dalla loro allocazione e deallocazione.
  - Moduli: In linguaggi come Python, i moduli hanno una durata di vita che coincide con la durata del programma o del processo di importazione.

5. Modello dati

Un **modello dati** è una rappresentazione formale dei tipi di dati 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):

```
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:

```
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.

```
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)
```

Parte II.

Seconda parte: Le basi di Python

6. Introduzione a Python

Python è un linguaggio di programmazione multiparadigma rilasciato da Guido van Rossum nel 1991, quindi dopo il C++ e prima di Java e PHP. È multiparadigma, cioè abilita o supporta più paradigmi di programmazione, e multipiattaforma, potendo essere installato e utilizzato su gran parte dei sistemi operativi e hardware.

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 leggibile. La semantica del linguaggio è intuitiva e coerente, il che riduce la curva di apprendimento e minimizza gli errori.

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.

6.1. 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.

6.2. 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.

6.3. 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.

[1] PEP 20 – The Zen of Python

6.4. 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.

6.4.1. 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.

6.4.2. 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.

6.4.3. 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.

[2] Documentazione delle librerie standard di Python

- `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.

6.4.4. 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.

6.4.5. Utility e 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.

6.5. L'algoritmo di ordinamento 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:

```
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.

7. Scaricare e installare Python

7.1. 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`).

7.2. 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.

7.3. 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.

7.3.1. 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.

7.3.2. 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.

7.4. 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!
```

7.5. 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!
```

7.6. 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.

7.6.1. IDE

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

7.7. 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`).

7.8. 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...`).

7.9. 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`.

7.9.1. 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.

7.10. 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.

7.11. 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.

7.12. 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.

7.12.1. 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.

7.13. 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.

7.14. 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.

7.15. 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.

8. La struttura lessicale di Python

Per iniziare ad imparare Python come linguaggio, partiamo da una semplificazione della stuttura lessicale, cioè dall'insieme di regole sintattiche più significative, sia per comprendere le regole di composizione di programmi comprensibili all'interprete, sia per sfrutturne appieno tutte le potenzialità.

Ogni programma Python è costituito da una serie di file di testo contenenti il codice sorgente con una certa codifica, il default è 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.

8.1. 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.

8.2. 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.

8.3. 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` hanno identica indentazione.
② I tre blocchi hanno come unico vincolo quello di avere un livello maggiore 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.

8.4. 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")
```

8.4.1. 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  # Identificatore di numero intero
decimale = 3.14  # Identificatore di numero decimale
testo = "Ciao, mondo!"  # Identificatore di stringa
lista = [1, 2, 3]  # Identificatore di lista
dizionario = {"chiave": "valore"}  # Identificatore di dizionario

def mia_funzione(): # Identificatore di funzione
  print("Questa è una funzione")

# Classe
class MiaClasse: # Identificatore di classe
  def __init__(self, valore): # Identificatore di metodo e parametro
    self.valore = valore  # Identificatore di attributo

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

import math  # Identificatore di modulo

def mio_generatore(): # Identificatore di generatore
  yield 1
  yield 2
  yield 3

mio_oggetto = MiaClasse(10)  # Identificatore di istanza
```

8.4.2. 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

# Otteniamo la lista delle parole chiave
parole_chiave = keyword.kwlist

# Otteniamo la lista delle parole chiave contestuali
parole_chiave_contestuale = keyword.softkwlist

# Stampiamo la lista delle parole chiave
print(parole_chiave)

# Stampiamo la lista delle parole chiave contestuali
print(parole_chiave_contestuale)
```

Nella tabella seguente invece un elenco completo con breve descrizione:

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

Parola chiave	Descrizione
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
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

# Test della funzione
process_value(1)  # Output: Uno
process_value(2)  # Output: Due
process_value(3)  # Output: Altro

# Stampa della variabile `match`
print(match)  # Output: Questo è un identificatore valido
```

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.

- `type`:

```
from typing import TypeAlias

type Point = tuple[float, float]                                    ①

# Utilizzo dell'alias di tipo
def distanza(p1: Point, p2: Point) -> float:
  return ((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2) ** 0.5

# Test della funzione con alias di tipo
punto1: Point = (1.0, 2.0)
punto2: Point = (4.0, 6.0)

print(distanza(punto1, punto2))  # Output: 5.0

print(type(punto1)) # Output: <class 'tuple'>                       ②
```

① Uso di `type` come parola chiave.
② Uso di `type` identificatore di una funzione.

8.4.3. 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:

```
# Nel modulo example.py`
_private_variable = "Variabile da non esportare'"                   ①

# In altro modulo diverso da example.py
from example import *

print(_private_variable)                                            ②
```

① Nel modulo `example.py` viene definita la variabile come privata.
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:

```
result = 5 + 3
print(_)  # 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:

[1] I nomi con doppio underscore (`__`) sono chiamati *dunder* come abbreviazione di *double underscore*.

```
class BaseClass:
  def __init__(self):
    self.__private_attr = "Base"

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`.

8.4.4. 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 risultato. Attenzione a non confondere la definizione di operatore come token, come 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.
③ Utilizza vari operatori su espressioni.

In tabella l'elenco degli operatori:

Tipo di operatore	Operatore	Descrizione
Aritmetici	`+`	Addizione
	`-`	Sottrazione
	`*`	Moltiplicazione
	`/`	Divisione
	`//`	Divisione intera

Tipo di operatore	Operatore	Descrizione
	`%`	Modulo
	`**`	Esponenziazione
	`@`	Matrice (operatore di moltiplicazione)
Confronto	`<`	Minore
	`>`	Maggiore
	`<=`	Minore o uguale
	`>=`	Maggiore o uguale
	`==`	Uguale
	`!=`	Diverso
Bitwise	`&`	AND bit a bit
	`|`	OR bit a bit
	`^`	XOR bit a bit
	`~`	NOT bit a bit
	`<<`	Shift a sinistra
	`>>`	Shift a destra
Assegnazione	`:=`	Operatore di assegnazione in espressione (walrus o tricheco)

Esempio su `@` che illustra un aspetto importante: il comportamento degli operatori può (o meglio, deve) essere definito quando si creano dei tipi di oggetto. Infatti, nel codice seguente, è definita una matrice assieme a una delle operazioni matematiche più comuni che è la moltiplicazione, implementata per mezzo di `__matmul__`:

```
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):
    if self.num_colonne != altra.num_righe:                          ①
      raise ValueError("Non è possibile moltiplicare le matrici: "
                       "dimensioni incompatibili.")

    risultato = [[0 for _ in range(altra.num_colonne)]               ②
          for _ in range(self.num_righe)]

    for i in range(self.num_righe):                                  ③
      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):                                                ④
    return '\n'.join([' '.join(map(str, riga)) for riga in self.righe])

# Definizione di due matrici
```

```
A = Matrice([[1, 2],
             [3, 4]])
B = Matrice([[5, 6],
             [7, 8]])

# Moltiplicazione di matrici utilizzando l'operatore @
C = A @ B

print("Matrice A:")
print(A)

print("Matrice B:")
print(B)

print("Risultato di A @ B:")
print(C)                                                    (5)
```

(1) Controlla se le dimensioni sono compatibili per la moltiplicazione.
(2) Inizializza la matrice risultato con zeri.
(3) Esegue la moltiplicazione delle matrici.
(4) Rappresentazione leggibile della matrice.
(5) Chiama `__matmul__` per ottenere la stringa su due righe: `19 22` e `43 50`.

Infine, `@` è anche un delimitatore.

8.4.5. 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:

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

Delimitatore	Descrizione
`=`	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

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] 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.

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:

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: `$`, `?`, `` ` ``.

8.4.6. 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.

8.4.6.1. Numerici

I letterali numerici includono interi, numeri a virgola mobile e numeri complessi:

- Interi, possono essere scritti in base decimale, ottale, esadecimale o binaria:
  - Decimale: `10`, `-3`.
  - 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`.
- Complessi, appresentati da una parte reale e una parte immaginaria: `3+4j`, `-1-0.5j`.

8.4.6.2. 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.

8.4.6.3. F-stringhe

Le f-stringhe (stringhe formattate) sono racchiuse tra virgolette singole, doppie o triple e sono precedute dal prefisso `f` o `F`. Permettono di includere espressioni Python all'interno.

Si possono avere stringhe formattate grezze ma non byte.

Esempio:

```
nome = "Python"

f_stringa = f'Ciao, {nome.upper()}!'                                    (1)

definizione = "Linguaggio"
```

[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.
Alcuni esempi comuni includono:

- `\n` per una nuova linea (linefeed).
- `\t` per una tabulazione.
- `\\` per inserire un backslash.
- `\'` per un apostrofo.
- `\"` per una doppia virgoletta.

Questi caratteri permettono di includere simboli speciali nelle stringhe senza interrompere la sintassi del codice.

```
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.
③ Output: `Output: Ciao, python!`.
④ Output composto dalle tre righe `Questo è un esempio`, `di f-stringa multi-linea` e `in linguaggio Python`.

8.5. Istruzioni

Un programma Python è una sequenza di istruzioni che si distinguono in **semplici** e **composte**.

8.5.1. Istruzioni semplici

Un'istruzione semplice è sempre contenuta in una riga logica, che può presentare più istruzioni semplici separate da `;`. È permesso ma scaonsigliato perché in pochi casi porta a codice leggibile.

In Python, le istruzioni semplici sono:

- Assegnazione, attribuisce un valore a una variabile:

```
x = 10
```

- Istruzioni di importazione, permettono di utilizzare gli identificatori definiti in altri moduli:

```
import math
from math import sqrt
```

- Istruzioni di controllo del flusso nei cicli:
  - `break`: Interrompe un ciclo.
  - `continue`: Salta all'iterazione successiva di un ciclo.
  - `pass`: Non esegue alcuna operazione.

```
for i in range(10):
    if i == 5:
        break
```

- Istruzioni di gestione dell'uscita da una funzione, della generazione di valori o del sollevamento di eccezioni:
  - `return`: Restituisce un valore da una funzione.
  - `yield`: Restituisce un generatore.
  - `raise`: Solleva un'eccezione per segnalare uan condizione di errore.

```
def func():
    return x
```

- Istruzioni di asserzione in cui si verifica una condizione e viene generata un'eccezione se la condizione è falsa:

```
assert x > 0, "x deve essere positivo"
```

- Istruzioni di diichiarazione di modifica dell'ambito di variabili:

```
global x
nonlocal y
```

Un'espressione è una istruzione semplice ed, infatti, inserita nel REPL, ne viene prodotto il risultato della valutazione. D'altronde, una espressione è spesso utilizzata per chiamare funzioni che hanno effetti collaterali, come, ad esempio, produrre un output:

- Produzione di output:

```
# Funzione che stampa un messaggio
def stampa_messaggio(messaggio):
  print(messaggio)

# Istruzione di espressione che chiama la funzione
# con un effetto collaterale (stampa del messaggio)
stampa_messaggio("Ciao, mondo!")
```

- Modifica di parametri:

```
# Funzione che modifica un argomento mutabile (lista)
def aggiungi_elemento(lista, elemento):
  lista.append(elemento)

# Lista iniziale
numeri = [1, 2, 3]

# Istruzione di espressione che chiama la funzione
# con un effetto collaterale (modifica dell'argomento)
aggiungi_elemento(numeri, 4)

print(numeri)  # Output: [1, 2, 3, 4]
```

- Modifica di variabili globali:

```
# Variabile globale
contatore = 0

# Funzione che modifica una variabile globale
def incrementa_contatore():
  global contatore

  contatore += 1

# Istruzione di espressione che chiama la funzione
```

```
# con un effetto collaterale (modifica della variabile globale)
incrementa_contatore()

print(contatore)  # Output: 1
```

- Lancio di eccezioni:

```
# Funzione che solleva un'eccezione
def solleva_eccezione(messaggio):
  raise ValueError(messaggio)

# Istruzione di espressione che chiama la funzione con un effetto collaterale (sollevamento di
try:
  solleva_eccezione("Qualcosa è andato storto!")

except ValueError as e:
  print(e)  # Output: Qualcosa è andato storto!
```

Un assegnamento con `=` è anch'esso un'istruzione semplice e non può mai essere all'interno di una espressione, dove, invece, si può usare l'operatore *tricheco* `:=`:

```
if (n := len("Python")) > 5:
  print(f"La lunghezza della stringa è {n}")
  # Output: La lunghezza della stringa è 6
```

8.5.2. Istruzioni composte

Una istruzione composta è costituita da altre istruzioni (semplici o composte). Il controllo dell'esecuzione delle istruzioni componenti avviene per mezzo di una o più clausole che iniziano tutte con una parola chiave, sono terminate da `:` e seguite da un blocco di codice. Ogni blocco deve avere almeno una istruzione semplice, ma può non avere una propria riga logica, cioè stare sulla stessa riga fisica e logica del `:`.

Alcuni esempi del rapporto tra istruzioni e righe:

- Blocco di istruzioni separato su più righe con medesima indentazione:

```
if x > 0:
  print("x è positivo")

  x += 1

  print(f"x ora è {x}")
```

- Blocco come singola istruzione sulla stessa riga logica:

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

- Diverse istruzioni semplici sulla stessa riga logica (non consigliato):

```
if x > 0: print("x è positivo"); x += 1; print(f"x ora è {x}")
```

Ed ecco le principali istruzioni composte:

- `if`, controlla l'esecuzione di un blocco di codice in base a una condizione:

```
if condizione:
    # blocco di codice
```

- `for`, itera su una sequenza (come una lista, una tupla o una stringa):

```
for elemento in sequenza:
    # blocco di codice
```

- `while`, esegue un blocco di codice finché una condizione è vera:

```
while condizione:
    # blocco di codice
```

- `try`, gestisce gli errori che possono verificarsi durante l'esecuzione di un blocco di codice:

```
try:
    # blocco di codice
except Eccezione:
    # blocco di codice per gestire l'eccezione
```

- `with`, gestisce l'allocazione e la deallocazione di risorse:

```
with open('file.txt', 'r') as file:
    # blocco di codice
```

- `def`, definisce una funzione:

```
def nome_funzione(parametri):
    # blocco di codice
```

- `class`, definisce una classe:

```
class NomeClasse:
    # blocco di codice
```

- `match`, esegue il pattern matching su un valore:

```
match valore:
    case pattern:
        # blocco di codice
```

9. Modello dati

Un **modello dati** è una rappresentazione astratta che definisce come i dati sono organizzati, archiviati e manipolati all'interno di un sistema informatico. In altre parole, un modello dati stabilisce le strutture dei dati e le relazioni tra essi, fornendo un quadro concettuale per comprendere e gestire le informazioni.

9.0.1. Componenti di un Modello Dati

Un modello dati tipicamente include le seguenti componenti:

1. **Entità**: Le entità rappresentano oggetti o concetti del mondo reale che possiedono attributi e hanno un'esistenza indipendente. Ad esempio, in un database di una biblioteca, entità possono essere "Libri", "Autori", e "Utenti".

2. **Attributi**: Gli attributi sono proprietà o caratteristiche delle entità. Per esempio, l'entità "Libro" può avere attributi come "Titolo", "Autore", "Anno di pubblicazione" e "ISBN".

3. **Relazioni**: Le relazioni definiscono le associazioni tra le entità. Ad esempio, una relazione può indicare che "Autore" scrive "Libro" o che "Utente" prende in prestito "Libro".

4. **Vincoli**: I vincoli sono regole che limitano i dati ammessi all'interno del modello, garantendo l'integrità e la coerenza dei dati. Un esempio di vincolo è che un ISBN deve essere univoco per ogni libro.

9.0.2. Tipi di Modelli Dati

Esistono diversi tipi di modelli dati, ciascuno con il proprio livello di astrazione e applicazioni specifiche:

1. **Modello Dati Concettuale**: Questo modello fornisce una visione ad alto livello della struttura dei dati, senza preoccuparsi di come i dati saranno fisicamente implementati. È spesso rappresentato mediante diagrammi E/R (Entità-Relazione).

2. **Modello Dati Logico**: Questo modello traduce il modello concettuale in strutture che possono essere implementate in un sistema di gestione di database (DBMS). Definisce tabelle, colonne, chiavi primarie e chiavi esterne.

3. **Modello Dati Fisico**: Questo modello descrive l'implementazione effettiva dei dati nel DBMS, includendo dettagli come i tipi di dati specifici, gli indici, le partizioni, e le modalità di memorizzazione.

9.0.3. Importanza del Modello Dati

Un modello dati ben progettato è fondamentale per diverse ragioni:

- **Organizzazione dei Dati**: Fornisce una struttura chiara e logica per l'archiviazione e l'accesso ai dati.
- **Efficienza**: Ottimizza le operazioni di database, migliorando la velocità e la scalabilità.
- **Integrità dei Dati**: Assicura che i dati siano accurati, coerenti e privi di errori.
- **Manutenzione**: Facilita la manutenzione e l'espansione del sistema, consentendo aggiornamenti e modifiche più agevoli.

9.0.4. Conclusione

Un modello dati è essenziale per la progettazione e la gestione efficiente dei dati in qualsiasi sistema informatico. Comprendere i principi dei modelli dati è cruciale per sviluppatori, amministratori di database e analisti di dati, poiché fornisce le basi per costruire applicazioni solide e scalabili.

L'operazione di un programma Python si basa sui dati che gestisce. I valori dei dati in Python sono noti come oggetti; ogni oggetto, alias valore, ha un tipo. Il tipo di un oggetto determina quali operazioni l'oggetto supporta (in altre parole, quali operazioni è possibile eseguire sul valore). Il tipo determina anche gli attributi e gli elementi dell'oggetto (se presenti) e se l'oggetto può essere modificato. Un oggetto che può essere modificato è noto come oggetto mutabile, mentre uno che non può essere modificato è un oggetto immutabile. Trattiamo gli attributi e gli elementi degli oggetti nella sezione "Object attributes and items".

La funzione incorporata `type(obj)` accetta qualsiasi oggetto come argomento e restituisce l'oggetto di tipo che rappresenta il tipo di `obj`. La funzione incorporata `isinstance(obj, type)` restituisce True quando l'oggetto `obj` ha il tipo `type` (o qualsiasi sua sottoclasse); altrimenti, restituisce False. L'argomento `type` di `isinstance` può anche essere una tupla di tipi (Python 3.10+ o più tipi uniti con l'operatore `|`), nel qual caso restituisce True se il tipo di `obj` corrisponde a uno qualsiasi dei tipi dati, o a qualsiasi sottoclasse di quei tipi.

Python ha tipi incorporati per tipi di dati fondamentali come numeri, stringhe, tuple, liste, dizionari e insiemi, come trattato nelle sezioni seguenti.

10. Esercizi

10.1. Funzioni e istruzioni composte

10.1.1. Numeri pari o dispari

Definire una funzione che prende in input un numero intero e restituisce una stringa di `Pari` o `Dispari`.

10.1.1.1. Riscaldamento: sperimentazione dell'operatore modulo

> Suggerimento
>
> Usare l'operatore modulo `%` che restituisce il resto della divisione di due interi con diversi input sia pari che dispari ed anche non numerici per vedere cosa accade.

```
n = 42

if n % 2 == 0:
  print("Pari")

else:
  print(Dispari)
```

10.1.1.2. Soluzione 1: test con operatore modulo

> Suggerimento
>
> Usare 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)
```

```
Pari
Dispari
```

10.1.1.3. Soluzione 2: test con operatore modulo e controllo degli input

> Suggerimento
>
> Usare l'operatore modulo `%` 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)
```

```
Pari
Dispari
```

10.1.1.4. Soluzione 3: test con operatore modulo e generazione di errore su input non intero

> Suggerimento
>
> Usare 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)
'''
```

```
Pari
Dispari
```

```
'\nrisultato = pari_o_dispari("42")\n\nprint(risultato)\n\nrisultato = pari_o_dispari(73.)\n\nprint
```

10.1.1.5. Soluzione 4: uso di funzione built-in

> **Suggerimento**
>
> Usare la funzione `divmod` che restituisce il quoziente e il resto della divisione di due interi.

```
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)
```

```
Pari
Dispari
```

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.