Da neofita di Python a campione

Antonio Montano

2024-05-24

Indice

Prefazione

Parte I

Prima parte: I fondamenti

1 I linguaggi di programmazione, i programmi e i programmatori

Partiamo da alcuni concetti basilari a cui collegare quelli che approfondiremo nel corso.

1.1 Definizioni

La **programmazione** è il processo di progettazione e scrittura di **istruzioni**, note 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++ e JavaScript. 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.

Quando un programma viene scritto e salvato in un file di testo, il computer deve eseguirlo per produrre le azioni desiderate. Questo processo si svolge in diverse fasi:

- Compilazione o interpretazione: Il codice sorgente, scritto in un linguaggio di alto livello leggibile dall'uomo, deve essere trasformato in un linguaggio macchina comprensibile dal computer. Questo avviene attraverso due possibili processi:
  - Compilazione: In linguaggi come C++ o Java, un compilatore traduce tutto il codice sorgente in linguaggio macchina, creando un file eseguibile. Questo file può poi essere eseguito direttamente dalla CPU.
  - Interpretazione: In linguaggi come Python o JavaScript, un interprete legge ed esegue il codice sorgente riga per riga, traducendolo in linguaggio macchina al momento dell'esecuzione.
- Esecuzione: Una volta che il programma è stato compilato (nel caso dei linguaggi compilati) o viene interpretato (nel caso dei linguaggi interpretati), il computer può iniziare ad eseguire le istruzioni. La CPU (central processing unit) legge queste istruzioni dal file eseguibile o dall'interprete e le esegue una per una. Durante questa fase, la CPU manipola i dati e produce i risultati desiderati.
- Interazione con i componenti hardware: Durante l'esecuzione, il programma può interagire con vari componenti hardware del computer. Ad esempio, può leggere e scrivere dati nella memoria, accedere ai dischi rigidi per salvare o recuperare informazioni, comunicare attraverso la rete, e interagire con dispositivi di input/output come tastiere e monitor. Questa interazione permette al programma di eseguire compiti complessi e di fornire output all'utente.

1.2 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.

1.2.1 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.

1.2.2 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 tipi 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 supportano più di un paradigma di programmazione, rendendo difficile assegnare un linguaggio a una sola categoria. Come ha affermato Bjarne Stroustrup, il creatore di C++, *un linguaggio di programmazione non è semplicemente supportare un certo paradigma, ma abilitare un certo stile di programmazione* (Stroustrup 1997).

2.1 Linguaggi di programmazione imperativa

La **programmazione imperativa** si concentra sull'esecuzione di istruzioni sequenziali che modificano lo stato del programma. Le istruzioni indicano al computer cosa fare passo dopo passo. Esempi di linguaggi che permettono il paradigma imperativo sono Assembly, C, Go, Python, per diversi casi d'uso:

- Assembly: Utilizzato nella programmazione a basso livello, come nello sviluppo di firmware e driver di dispositivi.
- C: Utilizzato per lo sviluppo di sistemi operativi e software di sistema, dove il controllo dettagliato delle operazioni è cruciale.
- Go: Sviluppato da Google, è utilizzato per costruire applicazioni di rete e sistemi scalabili, noto per la sua efficienza e facilità di utilizzo nelle applicazioni concorrenti.
- Python: Utilizzato in vari campi e noto per la sua semplicità e leggibilità, supporta la programmazione imperativa con l'uso di dichiarazioni di controllo e assegnazioni di variabili.

2.2 Linguaggi procedurali

La **programmazione procedurale** è un sottotipo di programmazione imperativa che organizza il codice in blocchi chiamati procedure o funzioni. Questi blocchi possono essere riutilizzati in diverse parti del programma per evitare ripetizioni e migliorare l'organizzazione del codice. Esempi sono Fortran, Pascal, C, Go, Python e i relativi casi d'uso:

- Fortran: Molto utilizzato in applicazioni scientifiche e di ingegneria per calcoli numerici ad alta precisione.
- Pascal: Storicamente utilizzato nei corsi di informativa per insegnare i fondamenti della programmazione.
- Go, Python: Supportano la programmazione procedurale grazie alla possibilità di definire funzioni e organizzare il codice in moduli.
- C: Anche se C non supporta i moduli nel senso moderno, utilizza file header (.h) e file sorgente (.c) per separare e organizzare il codice.

2.3 Linguaggi orientati agli oggetti

I **linguaggi orientati agli oggetti** modellano il problema come un insieme di oggetti che interagiscono tra loro per svolgere un compito. Gli oggetti sono istanze di classi, che possono contenere dati e metodi per manipolare quei dati. La programmazione orientata agli oggetti è estremamente utile per progettare architetture software complesse grazie ai suoi concetti di modularità, riutilizzabilità, astrazione, ereditarietà e polimorfismo. Alcuni linguaggi ad oggetti sono Java, Python, C++, Rust, Scala e i casi d'uso:

- Java: Ampiamente utilizzato per lo sviluppo di applicazioni aziendali, applicazioni Android e sistemi di backend.
- C++: Utilizzato in applicazioni ad alte prestazioni come videogiochi, motori grafici e software di simulazione. Il C++ supporta i template, che permettono la scrittura di codice generico e la metaprogrammazione, consentendo al codice di essere più flessibile e riutilizzabile.
- Rust: Concepito per garantire la sicurezza della memoria e la concorrenza, offre anche supporto per la programmazione orientata agli oggetti.
- Scala: Utilizzato per sviluppare applicazioni scalabili e sistemi distribuiti, spesso usato per l'elaborazione di grandi moli di dati. Scala supporta i generics, che sono simili ai template in C++, permettendo di scrivere codice generico e riutilizzabile.

2.4 Linguaggi funzionali

I **linguaggi funzionali** si concentrano sulla valutazione di espressioni e funzioni, trattandole alla stregua di equazioni matematiche. La programmazione funzionale enfatizza l'uso di funzioni pure (cioè hanno come unico effetto quello di produrre un output) e l'immutabilità dei dati. Alcuni esempi: Haskell, Lisp, ML, Scala e i casi d'uso:

- Haskell: Utilizzato nella ricerca accademica, nello sviluppo di software finanziario e nei sistemi di calcolo parallelo.

- Lisp: Storicamente utilizzato nell'intelligenza artificiale e nello sviluppo di software di simulazione. ELIZA, uno dei primi chatbot che potevano simulare una conversazione umana, era scritto in Lisp. John McCarthy nel 1959 introdusse la gestione automatica della memoria (garbage collection).
- Meta language (ML): Utilizzato nello sviluppo di compilatori, nell'analisi formale di programmi e in applicazioni finanziarie. ML ha diversi dialetti importanti, ognuno dei quali ha influenzato significativamente la programmazione funzionale e lo sviluppo di linguaggi di programmazione, come F# e OCaml.
- Scala: Abilita sia la programmazione orientata agli oggetti che la programmazione funzionale, rendendolo un linguaggio versatile per vari tipi di applicazioni.
- Python: Sebbene Python non sia un linguaggio di programmazione funzionale puro come l'Haskell, offre comunque molte funzionalità che facilitano lo stile di programmazione funzionale, ad esempio la funzioni di prima classe, quelle anonime dette lambda e le funzioni di ordine superiore (map, filter, reduce).

2.5 Altri paradigmi di programmazione

Oltre ai paradigmi principali sopra menzionati, esistono altri paradigmi di programmazione meno comuni ma altrettanto importanti in certi contesti.

- **Logico**: Prolog, che sta per programming in logic, è stato sviluppato nei primi anni '70 da Alain Colmerauer e Robert Kowalski. È uno dei linguaggi più noti per la programmazione logica e ha giocato un ruolo significativo nello sviluppo dell'intelligenza artificiale.
- **Concorrente**: Uno dei più diffusi linguaggi abilitanti la programmazione concirrente è l'Erlang, utilizzato nello sviluppo di sistemi distribuiti e applicazioni che richiedono alta disponibilità.
- **Dichiarativo**: La programmazione dichiarativa si concentra sul "cosa" deve essere fatto piuttosto che sul "come" farlo. In altre parole, in un linguaggio dichiarativo, il programmatore specifica il risultato desiderato, lasciando al sistema il compito di determinare come ottenerlo. Questo approccio contrasta con la programmazione imperativa, dove il programmatore deve fornire una sequenza dettagliata di passi per raggiungere il risultato. Un esempio è lo structured query language (SQL), standard de facto per interrogare e manipolare database relazionali. Altri ben noti linguaggi dichiarativi sono: CSS, XQuery, VHDL, RegEx, Makefile.

2.6 In sintesi

I paradigmi di programmazione offrono diversi approcci per risolvere problemi e progettare sistemi software. Ogni paradigma ha i suoi punti di forza e indirizza specifiche esigenze nel processo di sviluppo del software. La comprensione e l'utilizzo dei vari paradigmi permette

ai programmatori di scegliere l'approccio più appropriato per il problema in questione e di scrivere codice più efficace, mantenibile e riutilizzabile:

- Programmazione imperativa: Ottimale per problemi che richiedono una sequenza di istruzioni dettagliate e un controllo preciso sullo stato del programma.
- Programmazione procedurale: Favorisce la modularità e la riusabilità del codice tramite la suddivisione in procedure o funzioni.
- Programmazione orientata agli oggetti: Eccelle nella gestione di sistemi complessi grazie alla modularità, riusabilità, astrazione, ereditarietà e polimorfismo.
- Programmazione funzionale: Promuove funzioni pure, immutabilità e composizionalità, facilitando il ragionamento e la verifica del comportamento del sistema.
- Programmazione logica: Ideale per problemi che possono essere espressi in termini di relazioni logiche, come l'intelligenza artificiale e la risoluzione di vincoli.
- Programmazione dichiarativa: Si concentra sul "cosa" piuttosto che sul "come", rendendo il codice più leggibile e permettendo l'ottimizzazione automatica.

Alcuni linguaggi di programmazione, come Python e C++, sono noti per il loro supporto a molteplici paradigmi, rendendoli strumenti versatili e potenti nel repertorio di un programmatore.

3 Sintassi dei linguaggi di programmazione e algoritmi dei programmi

3.1 Che cos'è un algoritmo?

Una definizione informale di **algoritmo** è un insieme di istruzioni precise e finite che descrivono come eseguire un compito specifico o risolvere un problema. Pensate a un algoritmo come a una ricetta di cucina: la ricetta fornisce una serie di passaggi chiari da seguire per preparare un piatto.

Formalmente, 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 di un algoritmo includono:

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

3.2 Sintassi e semantica dei linguaggi di programmazione

I linguaggi di programmazione sono strumenti utilizzati per implementare algoritmi in modo che possano essere eseguiti da un computer. Un linguaggio di programmazione ha due componenti principali: la **sintassi** e la **semantica**.

3.2.1 Sintassi

La **sintassi** di un linguaggio di programmazione è l'insieme di regole che definiscono come devono essere scritte le 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.

Gli elementi principali della sintassi includono:

- 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 variabili e valori, come `+`, `-`, `*`, `/`, `=`, `==`, ecc.
- Separatori e 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 definiti dall'utente.
- Letterali: Rappresentazioni di valori costanti nel codice, come numeri (`123`), stringhe (`"hello"`), caratteri (`'a'`), ecc.
- Espressioni: Combinazioni di variabili, operatori e funzioni che vengono valutate per produrre un valore.

Questi elementi sintattici sono utilizzati per comporre istruzioni semplici e istruzioni complesse.

3.2.1.1 Istruzioni Semplici

Le **istruzioni semplici** sono costituite da singole operazioni che possono essere eseguite direttamente. Queste istruzioni sono fondamentali e spesso coinvolgono i seguenti elementi:

- Assegnazione: Utilizza un operatore di assegnazione (`=`) per attribuire un valore a una variabile, che possiamo pensare come un nome simbolico rappresentante una posizione dove è memorizzato un valore. Esempio:

  ```
  x = 5
  ```

  `x`: Identificatore (nome) della variabile. `=`: Operatore di assegnazione. `5`: Letterale numerico intero.

- Input/output: Utilizza parole chiave o funzioni di libreria per leggere valori dall'input o scrivere valori all'output. Esempio:

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

  `print`: Parola chiave o identificatore di funzione di libreria. `"Hello, World!"`: Letterale stringa. L'esecuzione dell'istruzione produce `"Hello, World!"` in output.

- Operazioni aritmetiche: Utilizza operatori aritmetici per eseguire operazioni su variabili e valori. Esempio: `plaintext y = 7 + 2.3` `y`: Identificatore della variabile. `=`: Operatore di assegnazione del risultato della valutazione dell'operazione a destra. `7`: Letterale numerico intero. `+`: Operatore aritmetico. `2.3`: Letterale numerico decimale. L'esecuzione dell'istruzione fa in modo che la variabile `y` sia associato a 9.3.

- Espressioni: Combinazioni di variabili, operatori e valori che producono un risultato. Esempio: `plaintext 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. 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.

3.2.1.2 Istruzioni Complesse

Le **istruzioni complesse** sono costituite da più istruzioni semplici e possono includere strutture di controllo del flusso. Esempi includono:

- Condizioni: Istruzioni che eseguono un blocco di codice solo se una condizione è vera. Esempio: `plaintext if (x > 0) { print("x è positivo") }` `if`: Parola chiave che introduce la condizione. `(x > 0)`: Condizione composta da: `x`: Identificatore della 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: `plaintext 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 del ciclo e`i++`: Incremento della variabile `i`. `{ ... }`: Delimitatori che racchiudono il blocco di codice. `somma = somma + i`: Operazione aritmetica.

3.2.2 Semantica

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 molto complessi e non tutti presenti in uno specifico linguaggio. Di seguito ne sono elencati alcuni tra i più diffusi:

- 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).

- Tipo di dati: I tipi di dati definiscono il dominio dei valori che una variabile può assumere e le operazioni che possono essere eseguite su quei valori. Un tipo di dato determina la natura del valore (ad esempio, numero intero, carattere, booleano) e le operazioni che possono essere effettuate su di esso. Generalmente si distinguono in:

  - Tipo primitivo: I tipi di dati fondamentali forniti da un linguaggio, come integer, float, boolean e character.
  - Tipo complesso: Tipo di dati costituiti da più tipi primitivi, come array, struct e oggetti.
  - Tipo di dati utente: Tipo definito dall'utente, come classi e tipi personalizzati, che permette di creare strutture dati più complesse e specifiche per il problema da risolvere.

- 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.

- 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.

- 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.

- Controllo di flusso: Determina l'ordine in cui le istruzioni vengono eseguite e alcuni esempi sono:

  - Condizionali: Strutture che permettono al programma di prendere decisioni (`if`, `else`, `switch`/`case`).
  - Cicli: Strutture che ripetono un blocco di codice (`for`, `while`, `do-while`).
  - Eccezioni: Meccanismi per gestire errori e condizioni anomale (`try`, `catch`, `throw`), permettendo al programma di continuare l'esecuzione in modo controllato.

- Classi e oggetti: Le classi sono trutture che definiscono proprietà (variabili) e comportamenti (metodi) comuni a tutti gli oggetti di quel tipo. Le classi rappresentano il modello o il blueprint da cui vengono creati gli oggetti. L'oggetto è l'istanza concreta di una classe. Gli oggetti sono entità che combinano dati e comportamenti secondo la struttura definita dalla loro classe. Si applicano i seguenti:

  - Encapsulamento: Nasconde i dettagli interni di un oggetto e mostra solo le interfacce necessarie, migliorando la modularità e la manutenzione del codice.
  - Ereditarietà: Permette di creare nuove classi basate su classi esistenti, riutilizzando e estendendo il comportamento delle classi base.
  - Polimorfismo: Consente a metodi di comportarsi diversamente a seconda dell'oggetto su cui vengono invocati, fornendo flessibilità e estendibilità.

- Gestione della memoria utilizzata dal programma: La gestione della memoria è fondamentale per il funzionamento efficiente di un programma. Ne esistono diverse modalità:

  - Allocazione dinamica: La memoria è allocata e deallocata a runtime, permettendo una gestione flessibile delle risorse.
  - Garbage collection: Automatizza la deallocazione della memoria non utilizzata, riducendo il rischio di sfruttamento non ottimale (memory leak) e semplificando la gestione della memoria.

- Spazio di nomi (in inglese namespace): I namespace organizzano variabili, funzioni e altri identificatori per evitare conflitti di nome.
- Moduli e librerie: I moduli e le librerie suddividono il codice in unità riutilizzabili e organizzate, da importare in programmi. I moduli possono definire degli spazi di nomi.
- Concorrenza: La concorrenza permette l'esecuzione parallela di più sequenze di istruzioni, migliorando le prestazioni e la reattività. Alcuni concetti relativi sono:

  - Thread: Un thread è la più piccola unità di elaborazione che può essere eseguita in modo indipendente. I thread consentono l'esecuzione parallela di codice all'interno di un programma.
  - Sincronizzazione: Meccanismi per gestire l'accesso concorrente alle risorse condivise, prevenendo condizioni di gara e garantendo la consistenza dei dati.
  - Lock e mutex: Meccanismi per prevenire condizioni di corsa (in inglese race condition), cioè un fenomeno che si presenta nei sistemi concorrenti quando, in presenza di una sequenza di processi multipli, il risultato finale dell'esecuzione dei processi dipende dalla temporizzazione o dalla sequenza con cui vengono eseguiti. Garantiscono, pertanto, l'accesso sicuro alle risorse condivise.

  - Async/await: Gestione di operazioni asincrone, migliorando l'efficienza e la reattività delle applicazioni.

- Input/output (I/O): L'input/output gestisce la comunicazione tra il programma e l'ambiente esterno.
  - File I/O: Lettura e scrittura su file per memorizzare e recuperare dati persistenti.
  - Network I/O: Comunicazione attraverso reti per inviare e ricevere dati tra sistemi diversi.
  - Standard I/O: Interazione con l'utente tramite input da tastiera e output su schermo.

- Annotazioni e metadati: Le annotazioni e i metadati forniscono informazioni aggiuntive al compilatore o al runtime, influenzando il comportamento del programma o fornendo dettagli utili per la documentazione e l'analisi del codice.
  - Annotazioni: Informazioni extra utilizzate per specificare comportamenti speciali o configurazioni. Ad esempio, in Java, le annotazioni possono essere utilizzate per indicare che un metodo è obsoleto (`@Deprecated`), per sovrascrivere un metodo della superclasse (`@Override`), o per specificare la relazione tra entità nel contesto di framework come JPA (`@Entity`, `@Table`). In Python, le annotazioni sono utilizzate principalmente per indicare i tipi di variabili, parametri di funzione e valori di ritorno (type hint). Non influenzano direttamente il comportamento del programma, ma sono utili per la documentazione e il type checking anche automatico.
  - Docstring: Commenti strutturati che documentano il codice, spesso utilizzati per generare documentazione automatica. In Python, ad esempio, le docstring possono essere utilizzate per descrivere il funzionamento di moduli, classi, metodi e funzioni, rendendo il codice più leggibile e comprensibile.

- Macro e metaprogrammazione: Le macro e la metaprogrammazione permettono di scrivere codice che manipola altre porzioni di codice.
  - Macro: Sequenze di istruzioni predefinite che possono essere inserite nel codice durante la fase di precompilazione. In C, sono utilizzate con il preprocessore per definire costanti, funzioni inline e codice condizionale.
  - Metaprogrammazione: Tecniche per scrivere codice che genera o modifica altre parti del codice a runtime o a compile-time, migliorando la flessibilità e il riutilizzo del codice. In Python include l'uso di decoratori, metaclassi e introspezione.

Questi elementi semantici combinati determinano il comportamento e la logica di un programma, influenzando il modo in cui il codice viene scritto, eseguito e mantenuto.

3.3 Esempio di algoritmo

Consideriamo un esempio semplice di algoritmo per calcolare la somma dei numeri da 1 a n. In input si avrà un numero intero `n` e in output il risultato. In pseudocodice si può rappresentare così:

```
Inizializza somma a 0
Per ogni numero i da 1 a n:
  Aggiungi i a somma
Restituisci somma
```

O, in alternativa, possiamo definire una funzione che implementa l'algoritmo:

```
funzione calcola_somma(n):
  somma = 0
  per i da 1 a n:
    somma = somma + i
  ritorna somma
```

Effettuaimo una analisi dettagliata della funzione per indicare quali elementi sintattici e semantici sono presenti. Partiamo dalla prima riga:

```
funzione calcola_somma(n):
```

- Sintassi:
  - `funzione`: Parola chiave che introduce la definizione di una funzione.
  - `calcola_somma`: Identificatore della funzione.
  - `(n)`: Delimitatori che contengono un identificatore.
- Semantica:
  - Definisce una funzione chiamata `calcola_somma` che accetta un parametro `n`.

La seconda riga ha l'inizializzazione della variabile che conterrà il risultato:

```
somma = 0
```

- Sintassi:
  - `somma`: Identificatore della variabile.
  - `=`: Operatore di assegnazione.
  - `0`: Letterale numero intero.
- Semantica:

  - Inizializza la variabile `somma` a `0`.

A seguire la definizione di un ciclo:

```
per i da 1 a n:
```

- Sintassi:
  - `per`: Parola chiave che introduce il ciclo.
  - `i`: Identificatore della variabile di controllo del ciclo.
  - `da 1 a n`: Espressione di controllo del ciclo che indica l'intervallo.
- Semantica:
  - Itera la variabile `i` da `1` a `n`.

Un assegnamento per accumulare i valori nella variabile di ritorno:

```
somma = somma + i
```

- Sintassi:
  - `somma`: Identificatore della variabile.
  - `=`: Operatore di assegnazione.
  - `somma + i`: Espressione aritmetica composta da:
    - `somma`: Identificatore della variabile.
    - `+`: Operatore aritmetico.
    - `i`: Identificatore della variabile.
- Semantica:
  - Aggiunge il valore di `i` alla variabile `somma` e assegna il risultato a `somma`.

E finalmente il risultato del calcolo viene restituito al chiamante:

```
ritorna somma
```

- Sintassi:
  - `ritorna`: Parola chiave che indica la restituzione di un valore.
  - `somma`: Identificatore della variabile.
- Semantica:
  - Restituisce il valore della variabile `somma` come risultato della funzione.

Abbiamo così dissezionato un algoritmo molto semplice per illustrare come sintassi e semantica di un linguaggio abbiano ruoli distinti e complementari in un programma. È importante comprendere che un buon programmatore deve avere tutte e tre le competenze, cioè conoscere le specificità formali del linguaggio (o di più linguaggi), quindi, la sua sintassi e semantica e saper comporre algoritmi, che potrà realizzare grazie proprio a quelle.

Parte II

Seconda parte: Le basi di Python

4 introduzione a Python

Python è un linguaggio di programmazione multiparadigma rilasciato da Guido van Rossum nel 1991, 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.

4.1 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.

4.2 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: Python supporta la programmazione imperativa, che si basa sull'esecuzione di istruzioni in una sequenza specifica. 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 di 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. Sfruttando la modularità, si possono creare collezioni di strumenti pronti all'uso.

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.

4.3 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.

4.4 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 e anche come sorgente sul sito ufficiale python.org.

4.4.1 L'interprete

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

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

[1] PEP 20 – The Zen of Python

4.4.2 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.

4.4.3 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, solo allo scopo di illustrarne la varietà, includono:

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

4.4.4 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:

[2] Documentazione delle librerie standard di Python

- 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.

4.4.5 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.

4.5 L'algoritmo di ordinamento bubble sort

Per chiudere il capitolo sul primo approccio a Python, possiamo confrontare un algoritmo di media complessità 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++, e Java.

4.5.1 Python

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
```

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

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

bubble_sort(arr)

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

4.5.2 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("Sorted array is: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
```

```
    return 0;
}
```

4.5.3 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 << "Sorted array is: ";
    bs.printArray(arr, n);
```

```
    return 0;
}
```

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

4.5.5 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!("Sorted array is: {:?}", arr);
}
```

4.5.6 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("Sorted array is: " + 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 leggibile, mentre C richiede una gestione manuale della memoria e una sintassi più dettagliata.

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.

5 Scaricare e installare Python

5.1 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: Clicca su "Download Python 3.12.x" (assicurati di scaricare la versione più recente).
   - Per macOS: Clicca su "Download Python 3.12.x".
   - 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`).

5.2 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.

5.3 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

[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

Python.

5.3.1 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.

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.

5.3.2 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.

5.4 Windows

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

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

    3. 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:
    1. Apri il prompt dei comandi.
    2. 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
```

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

```
python hello.txt
```

    oppure, se il tuo sistema utilizza `python3`: `plaintext  python3 hello.txt`
4. Visualizza il risultato:

```
Hello, World!
```

5.5 macOS

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

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

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

```
cd ~/Documents
```

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

```
python3 hello.txt
```

4. Visualizza il risultato:

```
Hello, World!
```

5.6 Linux

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

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

    3. 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: `bash mv hello.txt hello`

3. Esegui il file Python:

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

```
cd ~/Documenti
```

    3. 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.

5.6.1 IDE

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

5.7 IDLE

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`).

5.8 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...`).

5.9 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`.

5.9.1 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.

5.10 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.

5.11 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.

5.12 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>
```

3. Salva il file con estensione `.html` (ad esempio, `hello.html`).

4. Apri il file salvato in un browser web. Vedrai l'output `Hello, World!` direttamente nella pagina.

5.12.1 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.

5.13 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.

5.14 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.

5.15 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.

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.

Stroustrup, Bjarne. 1997. *The C++ Programming Language*. 3rd ed. Reading, MA, USA: Addison-Wesley.