Da neofita di Python a campione

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 permette 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:

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.

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 Esecuzione del programma

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 istruzione per istruzione, 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.

1.5 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.

1.5.1 Analisi dei requisiti

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.

1.5.2 Disegno architetturale

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. 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.

1.5.3 Disegno funzionale

Il **disegno funzionale** dettaglia come ogni componente del sistema realizza le proprie funzionalità. In questa fase si descrivono le operazioni specifiche che ogni componente deve eseguire, spesso utilizzando pseudocodice o diagrammi di attività per rappresentare il flusso di operazioni.

1.5.4 Implementazione

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.

1.5.5 Testing e debugging

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 possono essere di diversi tipi, come test unitari, test di integrazione e test di sistema.
- **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.

1.5.6 Distribuzione e manutenzione

Una volta che il software è stato testato e ritenuto pronto, si passa alla fase di **distribuzione** (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.

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 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 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 nelle lingue naturali, che esprime comandi per compiere azioni, i programmi imperativi sono una sequenza di comandi che il computer deve eseguire. 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. I linguaggi imperativi di alto livello utilizzano variabili e istruzioni più complesse, ma seguono ancora lo stesso paradigma per mantenere la coerenza nella traduzione (compilazione o interpretazione) del codice.

2.2.1 Esempio in Assembly

Assembly è un linguaggio a basso livello strettamente legato all'hardware del computer. Ecco un esempio di un semplice programma scritto per l'architettura x86, utilizzando la sintassi dell'assembler NASM (Netwide Assembler). Il codice somma 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

section .text
    global _start

_start:
    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

    ; Print result (pseudo-syscall for simplicity)
    ; In realtà, si dovrebbe convertire il risultato in ASCII e chiamare le syscall appropri

    ; Terminazione del programma
    mov eax, 1          ; Codice di sistema per l'uscita
    int 0x80            ; Interruzione per chiamare il kernel
```

Le sezioni del codice:

- Dati: La sezione `.data` è utilizzata per dichiarare variabili statiche inizializzate.

- Testo: La sezione `.text` contiene il codice eseguibile. L'etichetta `_start` indica il punto di ingresso del programma.
- Registri: I registri come `al` e `eax` sono utilizzati per operazioni aritmetiche e per la memorizzazione temporanea dei dati.
- Chiamate di sistema: L'interruzione `int 0x80` è usata per eseguire chiamate di sistema in Linux.

L'Assembly x86 è usato nello sviluppo di:

- Sistemi operativi: Utilizzato nello sviluppo di kernel e driver.
- Applicazioni embedded (microcontrollori nei dispositivi medici, sistemi di controllo nei veicoli, dispositivi IoT, ecc.): 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 troviamo Python, un linguaggio di alto livello noto per la leggibilità ed eleganza. Ecco il medesimo esempio:

```python
# Definizione delle variabili
num1 = 5
num2 = 3

# Somma dei due numeri
result = num1 + num2

# Stampa del risultato
print("Il risultato è:", result)
```

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.
- Versatilità: Il linguaggio è progettato per una ben definita architettura e, quindi, ha una scarsa applicabilità ad altre.
- Precisione: Il programmatore ha un controllo dettagliato su ogni singola operazione.
- 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, permettendo di ignorare i dettagli dell'hardware.

- Semplicità: Il codice è più breve e leggibile, facilitando la comprensione e la manutenzione.
- Versatilità: Il linguaggio è applicabile senza modifiche a un elevato numero di architetture hardware-software.
- Produttività: I programmatori possono concentrarsi sulla logica del problema senza preoccuparsi dei dettagli implementativi.

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.