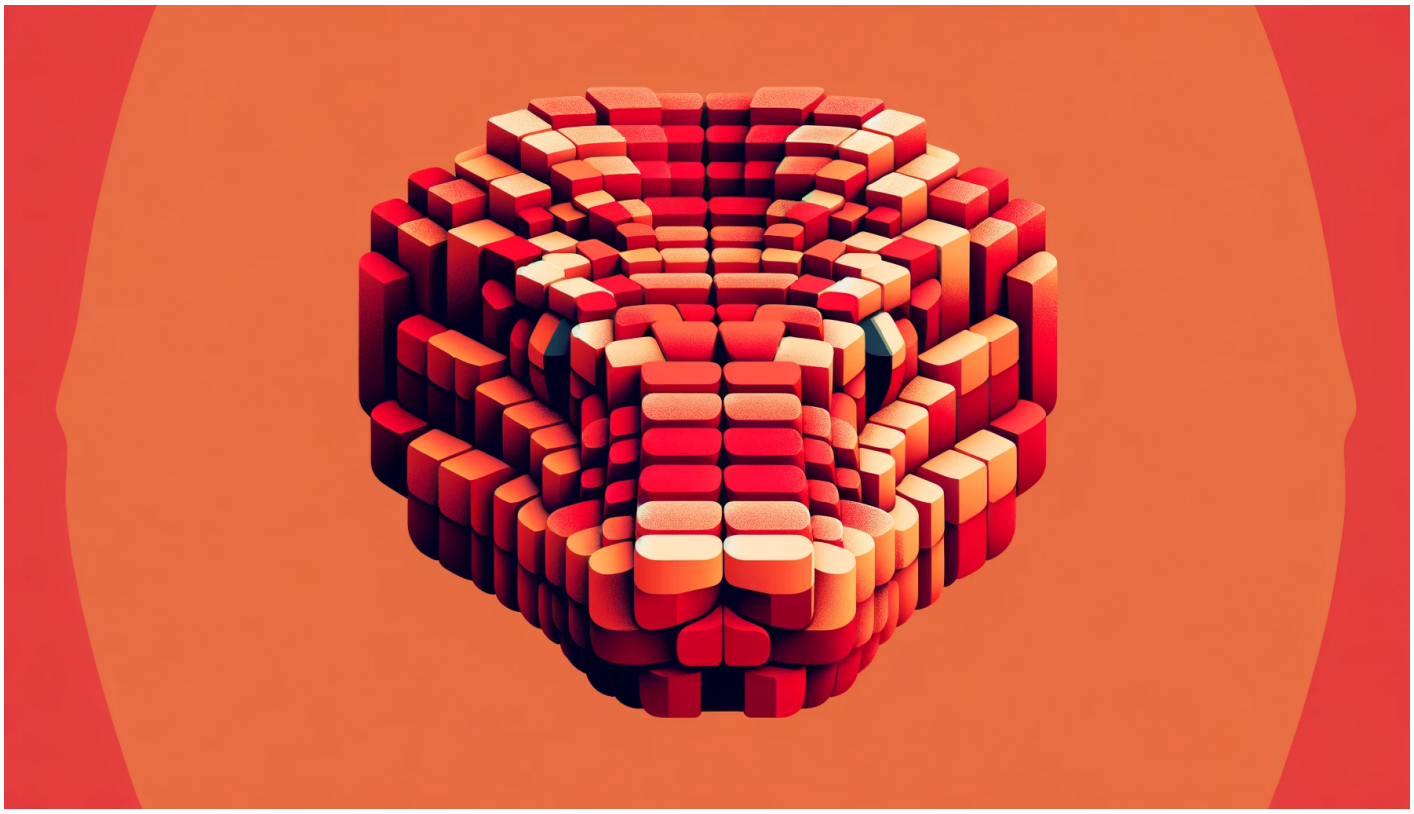
Da neofita di Python a campione

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:

- 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. 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.

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 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 (pseudocodice per semplicità)
```

```
    ; In realtà, si dovrebbe convertire il risultato in ASCII e chiamare le syscall appropriate.

    ; 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. `int 0x80`

L'Assembly x86 è usato nello sviluppo di:

- Sistemi operativi: Utilizzato nello sviluppo di kernel e driver.
- 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 troviamo Python, un linguaggio di alto livello noto per la leggibilità ed eleganza.

Ecco il medesimo esempio:

```
# 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.

2.3.1. Funzioni e procedure

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

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

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

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

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

2.3.2. Creazione di librerie

Un altro aspetto importante della programmazione procedurale è la possibilità di creare **librerie**, che sono collezioni di funzioni e procedure riutilizzabili. Le librerie permettono di organizzare e condividere codice comune tra diversi progetti, aumentando la produttività e riducendo la duplicazione del codice.

Esempio di una semplice libreria in C:

- File header (`mialibreria.h`):

```c
#ifndef MIALIBRERIA_H
#define MIALIBRERIA_H

int somma(int a, int b);
void stampaMessaggio(const char* messaggio);
```

```
#endif
```

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

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

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

void stampaMessaggio(const char* messaggio) {
    printf("%s\n", messaggio);
}
```

- File principale (`main.c`):

```
#include "mialibreria.h"

int main() {
    int risultato = somma(5, 3);
    stampaMessaggio("Il risultato è:");
    printf("%d\n", risultato);
    return 0;
}
```

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 "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 con stato e comportamento.
- Incapsulamento: Nasconde i dettagli interni di un oggetto e mostra solo le interfacce necessarie. Migliora la modularità e protegge l'integrità dei dati.
- Ereditarietà: Permette a una classe di estenderne un'altra, ereditandone attributi e metodi. Favorisce il riuso del codice e facilita l'estensione delle funzionalità.

- 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.
- 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.

I vantaggi principali dell'OOP sono:

- Modularità: Le classi e gli oggetti favoriscono la suddivisione del codice in moduli indipendenti, migliorando la manutenibilità e la riusabilità del software.
- Riutilizzabilità: L'uso di classi e l'ereditarietà consentono di riutilizzare il codice in nuovi progetti senza riscriverlo.
- Facilità di manutenzione: L'incapsulamento e l'astrazione riducono la complessità e facilitano la manutenzione del codice.
- Estendibilità: Le classi possono essere estese per aggiungere nuove funzionalità senza modificare il codice esistente.
- Affidabilità: Il polimorfismo e l'ereditarietà migliorano l'affidabilità del codice, poiché le modifiche possono essere fatte in una classe base e propagate alle classi derivate.

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.

2.4.1. Esempio in Java

In questo esempio, la classe `Animale` rappresenta una classe base con un attributo `nome` e un metodo `faiVerso`. La classe `Cane` estende `Animale` e sovrascrive il metodo `faiVerso` per fornire un'implementazione specifica. La classe `Main` crea un'istanza di `Cane` e chiama il metodo `faiVerso`, dimostrando il polimorfismo e l'ereditarietà:

```
// Definizione della classe base
class Animale {
    String nome;

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

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

// Definizione della classe derivata
```

```
class Cane extends Animale {

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

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

// Classe principale
public class Main {
    public static void main(String[] args) {
        Animale mioCane = new Cane("Fido");
        mioCane.faiVerso(); // Stampa: Il cane abbaia
    }
}
```

Vediamo come implementare il medesimo esempio con una classe astratta in Java. `Animale` è la classe astratta, cioè che non può essere istanziata in un oggetto e `Cane` è una classe concreta che la estende:

```
// Definizione della classe astratta
abstract class Animale {
    String nome;

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

    // Metodo astratto
    abstract void faiVerso();

    // Metodo concreto
    void descrizione() {
        System.out.println("L'animale si chiama " + nome);
    }
}

// Definizione della classe derivata
class Cane extends Animale {

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

    @Override
    void faiVerso() {
```

```
        System.out.println("Il cane abbaia");
    }
}

// Classe principale
public class Main {
    public static void main(String[] args) {
        Animale mioCane = new Cane("Fido");
        mioCane.descrizione(); // Stampa: L'animale si chiama Fido
        mioCane.faiVerso();    // Stampa: Il cane abbaia
    }
}
```

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>
class Box {
    T value;
public:
    void setValue(T val) { value = val; }
    T getValue() { return value; }
};
```

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

```
// Box contiene un intero
Box<int> intBox;
intBox.setValue(123);
int x = intBox.getValue();

// Box contiene una stringa
Box<std::string> stringBox;
stringBox.setValue("Hello, World!");
std::string str = stringBox.getValue();
```

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 {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];
    let max = largest(&numbers);
    println!("The largest number is {}", max);
}
```

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

La metaprogrammazione è presente anche in linguaggi non OOP come Lisp, che utilizza le macro per trasformare e generare codice.

Nel codice sotto proposto 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:

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

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:

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

Questo viene espanso in:

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

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 significativi 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:

```sql
SELECT nome
FROM utenti
WHERE età > 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`:

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

?- antenato(padre, 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. 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:

```haskell
-- Definizione di una funzione pura che calcola la somma dei numeri da 1 a n
sumToN :: Integer -> Integer
sumToN n = sum [1..n]

-- Funzione di ordine superiore che accetta una funzione e una lista
applyFunction :: (a -> b) -> [a] -> [b]
applyFunction f lst = map f lst

-- Utilizzo delle funzioni
main = do
    print (sumToN 10) -- Stampa 55
    print (applyFunction (*2) [1, 2, 3, 4]) -- Stampa [2, 4, 6, 8]
```

2.6.2. 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.

3. Sintassi e semantica dei linguaggi di programmazione e algoritmi dei programmi

3.1. 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. 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.

Partiamo dagli elementi atomici della sintassi, detti **token**, per poi risalire fino al programma:

- 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 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 ":".

3.2.1. 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.

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 viene utilizzato per ulteriori fasi di compilazione o interpretazione, come l'analisi semantica e la generazione del codice. 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.

3.2.2. Espressioni

Un'espressione è una combinazione di lessemi che viene valutata per produrre un risultato.

Esempi di espressioni includono:

- `5 + 3`.

[1] PEP 20 – The Zen of Python

- `x * 2`.
- `y / 4.0`.
- `max(a, b)`.
- `"Hello, " + "world!"`.

3.2.3. 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:

- Assegnazione: Utilizza un operatore di assegnazione (ad esempio, `=`) 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 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.

- Assegnazione ad espressione: Combinazione di variabili, operatori e valori che producono un risultato assegnato ad una variabile. 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.

3.2.4. Istruzioni complesse e blocchi di codice

Le **istruzioni complesse** 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 complesse 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 complesse per delimitare il gruppo di istruzioni che devono essere eseguite in determinate condizioni o iterazioni.

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

Alcuni esempi di istruzione e blocco di codice:

- Esempio in C:

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

  In questo esempio:

  - `if (x > 0)` è un'istruzione complessa.
  - `{ 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:

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

In questo esempio: - `if x > 0:` è un'istruzione complessa. - 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:

```c
#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:

```c
#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.

3.2.5. 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:

```c
int x = 10; // 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:

```c
int y = (10 + 20 + 30 + // Prima riga fisica della riga logica
         40 + 50); // Seconda riga fisica della riga logica
```

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

3.3. 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.
- 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.

- 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 (Jakarta Persistence; `@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.

3.4. Applicazione dei concetti di sintassi e semantica

Dissezioniamo 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.

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 (cioè mimiamo il rigore sintattico di un linguaggio di programmazione, ma utilizziamo la lingua italiana) 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
```

Effettuiamo 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 di variabile, `+` operatore aritmetico e `i` identificatore di 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.

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, 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.

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: 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.

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.

[1] PEP 20 – The Zen of Python

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, 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.

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, 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.

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:

- 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 bassa complessità ma non triviale, in diversi linguaggi di programmazione. Un buon esempio potrebbe essere l'implementazione dell'algoritmo di ordinamento *bubble sort* di una lista di valori. Vediamo come viene scritto in Python, C, C++, Java, Rust e Scala:

- Python in versione procedurale:

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

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

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

bubble_sort(arr)

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

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

```python
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):

```python
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:

```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:

```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 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: 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`).

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 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.

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.

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:
    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!
```

5.5. 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!
```

5.6. 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.

5.6.1. IDE

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

5.7. 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`).

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>
```

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.

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.

6. 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.

6.1. 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:

```
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 Questa riga e la successiva non hanno la stessa indentazione, anche se è da evitare perché poco leggibile.
③ La lista è spezzata su due righe fisiche e, anche qui, così è poco leggibile.

6.2. 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:

```
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 la precedente.
② Il commento rende il backslash parte di esso quindi non segnala più la fine della riga fisica e, all'esecuzione, si avrà un SyntaxError perché `and` deve essere seguito da un'espressione.

6.3. 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.

- Ciclo e 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.

6.4. 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")
```

6.4.1. 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
```

6.4.2. 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
Gestione delle risorse	

Parola chiave	Descrizione
`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 `_`:

```
# Definiamo una funzione che utilizza il pattern matching
def process_value(value):
  match value:
    case 1:
      print("Uno")
    case 2:
      print("Due")
    case _:
      print("Altro")

# Utilizzo di `match` come identificatore per una variabile
match = "Questo è un identificatore valido"

# 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
```

- `type`:

```
from typing import TypeAlias

# Dichiarazione di un alias di tipo
type Point = tuple[float, float]
```

```
# 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
```

6.4.3. 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.
  - Pattern nei match: Nel contesto di un pattern di corrispondenza all'interno di un'istruzione `match`, `_` è una soft keyword che denota un wildcard (carattere jolly).
  - Interprete interattivo: L'interprete interattivo rende disponibile il risultato dell'ultima valutazione nella variabile `_`. Questo risultato è memorizzato nel modulo builtins, insieme alle funzioni built-in come `print`.
  - Altro uso: Altrove, `_` è un identificatore regolare. Viene spesso usato per nominare elementi "speciali", ma non è speciale 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:

```
# Utilizzo di _ come identificatore in vari contesti

# Non importato con from module import *
_private_variable = "Questa variabile non verrà importata con 'from module import *'"

# Utilizzo di _ come wildcard in un'istruzione match
def process_value(value):
  match value:
    case 1:
      print("Uno")
    case _:
      print("Altro")

process_value(2)  # Output: Altro
```

```
# Uso di _ nell'interprete interattivo
result = 5 + 3
print(_)  # Output: 8 (nell'interprete interattivo)

# Uso di _ come variabile regolare
_ = "Valore non usato"

# Uso di _ per internazionalizzazione
import gettext
gettext.install('myapplication')
print(_('Hello, world'))
```

- `__*__`
  - Nomi definiti dal sistema: Questi nomi, informalmente noti come nomi "dunder", sono definiti dall'interprete e dalla sua implementazione (inclusa la libreria standard). I nomi di sistema attuali sono discussi nella sezione dei nomi dei metodi speciali e altrove. 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 rottura senza preavviso.

6.4.3.1. Esempio

```
# Utilizzo di nomi "dunder"
class MyClass:
    def __init__(self, value):
        self.__value = value  # Questo è un nome "dunder" per un attributo privato

    def __str__(self):
        return f"MyClass con valore {self.__value}"

obj = MyClass(10)
print(obj)  # Output: MyClass con valore 10
```

- `__*`
  - Nomi privati della classe: I nomi in questa categoria, quando utilizzati nel contesto di una definizione di classe, vengono riscritti per utilizzare una forma mangled per evitare conflitti di nome tra attributi "privati" delle classi base e derivate.

6.4.3.2. Esempio

```
# Utilizzo di nomi privati della classe
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()

# Accesso ai nomi riscritti (name mangling)
print(base_obj._BaseClass__private_attr)  # Output: Base
print(derived_obj._DerivedClass__private_attr)  # Output: Derived
```

In sintesi, Python riserva certe classi di identificatori che hanno significati speciali e seguono regole specifiche, principalmente per garantire l'integrità del codice e la compatibilità tra versioni diverse del linguaggio. Tuttavia, queste convenzioni non pongono limitazioni rigide al programmatore, che può comunque utilizzare questi identificatori secondo necessità, purché consapevole delle loro implicazioni.

6.4.4. 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 # Utilizza l'operatore + sugli identificatori x e y

sum = 3 + 4 # Utilizza l'operatore + su letterali

result = (x * y) + (z / 2) # Utilizza vari operatori su espressioni
```

In tabella l'elenco degli operatori:

Tipo di operatore	Operatore	Descrizione
Aritmetici	`+`	Addizione
	`-`	Sottrazione
	`*`	Moltiplicazione
	`/`	Divisione
	`//`	Divisione intera
	`%`	Modulo
	`**`	Esponenziazione
	`@`	Matrice (operatore di moltiplicazione)
Confronto	`<`	Minore
	`>`	Maggiore
	`<=`	Minore o uguale
	`>=`	Maggiore o uguale

Tipo di operatore	Operatore	Descrizione
	`==`	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):
        # Controlla se le dimensioni sono compatibili per la moltiplicazione
        if self.num_colonne != altra.num_righe:
            raise ValueError("Non è possibile moltiplicare le matrici: "
                             "dimensioni incompatibili.")

        # Inizializza la matrice risultato con zeri
        risultato = [[0 for _ in range(altra.num_colonne)]
                     for _ in range(self.num_righe)]

        # Esegue la moltiplicazione delle matrici
        for i in range(self.num_righe):
            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):
        # Rappresentazione leggibile della matrice
        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)
```

Infine, `@` è anche un delimitatore.

6.4.5. 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
`=`	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

Delimitatore	Descrizione
`@=`	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,[1] è trattata come un token a sé e corrisponde ad un oggetto predefinito chiamato Ellipsis, con applicazioni in diversi contesti:

```
print(type(...))  # <class 'ellipsis'>                                    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
# Output:
# [[ 2  5]
#  [ 8 11]]
```

① Otteniamo il tipo dell'oggetto ellissi.
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.

Alcuni caratteri ASCII hanno un significato speciale come parte di altri token o sono significativi per l'analizzatore lessicale:

[1] 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.

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

6.4.6. 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.

6.4.6.1. 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`.

6.4.6.2. 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[2] non sono intepretati.

Alcuni esempi comuni includono:

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

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

6.4.6.3. 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}!'
f_stringa_multi_linea = f'''Questo è
un esempio di
f-stringa multi-linea con {nome}'''

print(f_stringa)
print(f_stringa_multi_linea)
```

6.5. Istruzioni

Un programma Pythom è una sequenza di istruzioni che si distinguono in **semplici** o **composte**.

[2] 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.

6.5.1. 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.

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
```

6.5.2. 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.

Alcuni esempi:

- 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 (non consigliato):

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

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.