ChatGPT

Come “pensa” l’interprete Python: 50 punti chiave

Valutazione delle espressioni e operatori

- **Ordine di valutazione deterministico:** Python valuta le espressioni da sinistra a destra, rispettando la precedenza degli operatori. Ad esempio, in un’assegnazione `x = expr`, prima viene valutata la parte destra e solo dopo il risultato viene assegnato a `x` [1]. Questo approccio garantisce un comportamento coerente e prevedibile nell’ordine di esecuzione delle sotto-espressioni.
- **Precedenza e associatività degli operatori:** L’interprete applica le regole di precedenza standard (ad esempio, la moltiplicazione viene eseguita prima dell’addizione) e un ordine associativo predefinito. Operatori dello stesso livello sono generalmente valutati da sinistra a destra, **eccetto** alcuni come l’esponenziazione `**` (destro-associativo) [2]. Questo “schema di precedenza” spiega perché un’espressione come `2 + 3 * 4` dà 14 e non 20.
- **Valutazione “short-circuit” nei booleani:** Per gli operatori logici `and` e `or`, Python utilizza una valutazione cortocircuitata: valuta l’operando successivo solo se necessario. In `A and B`, B viene valutato solo se A è *vero*; in `A or B`, B viene valutato solo se A è *falso*. Questo comportamento è implementato per ragioni di efficienza (evitando calcoli inutili) e sicurezza (impedendo ad esempio accessi non validi se il primo operando già determina l’esito).

Nomi, variabili e oggetti

- **Variabili come riferimenti, non scatole:** In Python le variabili non contengono direttamente valori, ma sono etichette (riferimenti) associate ad oggetti in memoria [3]. Un’istruzione di assegnazione come `x = 5` non copia il valore 5 in `x`, ma collega il nome `x` all’oggetto numerico `5` già esistente [4]. Questo modello di *name binding* spiega perché assegnare `y = x` fa puntare `y` allo stesso oggetto di `x` invece di crearne una copia indipendente.
- **Un nome, più etichette:** Poiché i nomi sono riferimenti, è possibile che più variabili puntino allo stesso oggetto. Ad esempio, dopo `a = []` e `b = a`, sia `a` che `b` fanno riferimento alla stessa lista in memoria (hanno “etichette” diverse attaccate allo stesso oggetto lista) [3]. Questo significa che una modifica tramite `a` (come `a.append(1)`) sarà visibile anche leggendo da `b`, proprio perché l’oggetto sottostante è unico.
- **Identità degli oggetti:** Ogni oggetto Python ha un’identità unica e immutabile durante la sua vita, recuperabile con `id(obj)` [5]. L’operatore `is` confronta le identità, verificando se due riferimenti puntano al *medesimo* oggetto. Ad esempio, dopo `c = []` e `d = []`, `c is d` restituisce False perché sono due liste distinte, mentre assegnando `e = d`, allora `e is d` sarebbe True (entrambi i nomi riferiscono lo stesso oggetto) [6]. Questa distinzione spiega perché usare `==` (uguaglianza per valore) e `is` (uguaglianza per identità) può dare risultati diversi.
- **Mutabilità vs. immutabilità:** Alcuni tipi di oggetti in Python sono mutabili (possono cambiare stato) mentre altri sono immutabili (una volta creati, il loro valore non può essere modificato) [7]. Numeri, stringhe e tuple, ad esempio, sono immutabili; liste, dizionari e insiemi sono mutabili. L’interprete *pensa* in termini di oggetti: modificare un oggetto mutabile tramite un riferimento influisce su tutti i nomi che lo condividono, mentre “modificare” un immutabile in

realtà crea un nuovo oggetto. Questo è il motivo per cui facendo `x = 10; y = x; x += 1`, la variabile `y` rimane 10 (il numero 10 è immutabile, quindi `x += 1` crea un nuovo oggetto 11 senza toccare l'oggetto condiviso iniziale).

- **Riutilizzo degli oggetti immutabili comuni:** Per ottimizzazione, Python può riutilizzare lo **stesso oggetto** per valori immutabili identici, invece di crearne copie multiple. Ad esempio, l'implementazione CPython potrebbe utilizzare un'unica istanza per tutti gli `1` (interi con valore uno) presenti nel programma [8]. Questo significa che dopo `a = 1; b = 1`, è possibile (ma non garantito) che `a is b` risulti True perché entrambi puntano allo stesso oggetto integer 1 in memoria [8]. Ciò avviene perché interi e stringhe sono immutabili e quindi sicuri da condividere: Python ne approfitta per risparmiare memoria e tempo (ad esempio, mantiene una cache di piccoli interi e stringhe internate).
- **Oggetti mutabili sempre distinti quando creati:** Al contrario, ogni volta che si crea un nuovo oggetto mutabile, esso sarà *unico* in memoria. Ad esempio, eseguendo `c = []` e `d = []`, Python garantisce due liste distinte con identità diverse [9]. Solo un'assegnazione diretta (come `e = d`) può farli puntare allo stesso oggetto. Questo design evita che modifiche accidentali a una struttura di dati influenzino altre variabili a meno che non siano esplicitamente alias dello stesso oggetto (ad esempio con liste condivise intenzionalmente).
- **Piccole ottimizzazioni di interning:** Alcuni oggetti molto utilizzati sono **preallocati** o internati. Ad esempio, in CPython gli interi compresi grossomodo tra -5 e 256 sono allocati una volta e riutilizzati per qualsiasi variabile che assuma quei valori, anziché creare sempre nuovi oggetti [8]. Lo stesso accade con certe stringhe immutabili (come le stringhe brevi o quelle che somigliano a identificatori): l'interprete può riutilizzare l'oggetto esistente. Questo spiega perché `256 is 256` può risultare True (oggetto condiviso dalla cache), mentre `257 is 257` è False (oggetti distinti non cacheati). Tali micro-ottimizzazioni migliorano le prestazioni e l'uso di memoria per valori molto frequenti, pur essendo dettagli di implementazione.

Funzioni e passaggio di parametri

- **Chiamata di funzione "per assegnazione":** Python adotta un modello di passaggio argomenti per *assegnazione di riferimento*. Quando si chiama una funzione, per ogni parametro viene creato un nuovo nome locale che riferisce lo stesso oggetto passato come argomento [10]. In altre parole, all'interno della funzione i parametri sono come variabili locali inizializzate con gli oggetti forniti negli argomenti. Questo è il motivo per cui modificare un oggetto mutabile all'interno della funzione (es. una lista) si riflette all'esterno (perché la lista è la medesima), mentre riassegnare il parametro a un nuovo oggetto non ha effetto sulla variabile originale del chiamante.
- **Valutazione degli argomenti (ordine e completezza):** Prima di eseguire il corpo di una funzione chiamata, l'interprete valuta tutte le espressioni degli argomenti nell'ordine da sinistra a destra [11]. Solo dopo aver ottenuto tutti i valori, li assegna ai parametri formali e inizia l'esecuzione della funzione. Questo garantisce che gli effetti collaterali negli argomenti avvengano *prima* dell'ingresso nella funzione. Inoltre, Python verifica che il numero e tipo di argomenti corrisponda ai parametri: se mancano argomenti per qualche parametro (senza default) o ce ne sono in eccesso non previsti, viene sollevato un `TypeError` [12]. Allo stesso modo, gli argomenti keyword vengono associati ai parametri per nome [13] e quelli extra non corrispondenti vengono gestiti tramite parametri variadici (`*args` e `**kwargs`).
- **Valori di default e oggetti mutabili:** I valori di default dei parametri sono valutati **una sola volta** al momento della definizione della funzione, non ad ogni chiamata. Ciò significa che se un parametro ha come default un oggetto mutabile (es. una lista vuota), quella *stessa* lista viene riutilizzata in tutte le invocazioni che non forniscono un argomento per quel parametro [14]. È un comportamento intenzionale (per efficienza), ma spesso sorprende: per evitare effetti

indesiderati, si consiglia di usare default immutabili (come `None`) e dentro la funzione sostituirli con nuovi oggetti mutabili se necessario.

- **Ogni chiamata ha il suo ambiente:** Quando una funzione viene invocata, Python crea un nuovo *frame* di esecuzione con un proprio spazio di nomi locale (un dizionario separato) per le variabili della funzione [15]. In questo frame, i nomi dei parametri vengono legati agli oggetti argomento, e man mano che il codice della funzione viene eseguito, eventuali variabili locali aggiuntive vengono create in questo contesto. È possibile vedere il contenuto dello scope locale di una funzione usando `locals()` all'interno di essa (che restituisce un dizionario delle variabili locali) [16]. Una volta terminata la funzione (ritornata), il suo frame viene distrutto e le variabili locali (se non catturate altrove) diventano inaccessibili, a meno che non fossero chiuse in una *closure*.

Scope delle variabili e risoluzione dei nomi

- **Regola LEGB (Local, Enclosing, Global, Built-in):** Quando l'interprete deve risolvere un nome (variabile, funzione, ecc.), segue l'ordine di ricerca LEGB [17]. In pratica, cerca prima nello **scope Locale** corrente (ad esempio dentro la funzione in cui ci si trova), poi negli **scope Enclosing** esterni (nel caso di funzioni annidate, risale alle funzioni che le contengono), quindi nello **scope Globale** del modulo e infine nello scope dei nomi **Built-in** (definiti di default da Python, come funzioni intrinseche). Viene utilizzata la **prima definizione trovata** in questo ordine; se il nome non esiste in nessuno di questi ambienti, l'interprete solleva un `NameError` [17]. Questo meccanismo a livelli evita collisioni di nomi e permette la definizione di variabili locali senza interferire con quelle globali omonime.
- **Variabili locali di default:** All'interno di una definizione di funzione (o di un blocco), qualsiasi assegnamento a un nome fa sì che quel nome sia trattato come **locale** in quello scope, a meno che non sia dichiarato esplicitamente come `global` o `nonlocal` [18]. Ciò significa, ad esempio, che se dentro una funzione usi `x = 5`, Python considera `x` una variabile locale e non guarderà a una `x` globale omonima. Se provi a leggere il valore di `x` prima di assegnarlo in quella funzione, otterrai un `UnboundLocalError` perché l'interprete ha già deciso che `x` è locale (dunque non cerca nel globale) ma non gli è stato ancora assegnato un valore [19]. Questo comportamento può sembrare sottile, ma impedisce di *inavvertitamente* modificare variabili globali da dentro le funzioni senza accorgersene.
- **Keyword `global` e `nonlocal`:** Se si vuole indicare all'interprete che una certa variabile all'interno di una funzione si riferisce a quella definita nel **globale** (modulo) e non deve essere considerata locale, si usa la dichiarazione `global nomeVar`. Dopo questa dichiarazione, tutte le assegnazioni e riferimenti a quel nome nella funzione agiranno sulla variabile globale [20]. Analogamente, la keyword `nonlocal` permette a una funzione interna di riferirsi a una variabile definita nel suo **enclosing scope** (funzione esterna) invece di crearne una locale nuova [21]. Se tale variabile non esiste nello scope esterno immediato, Python solleva un errore in fase di compilazione (`SyntaxError`) [21]. Questi meccanismi espliciti sono stati introdotti per gestire le variabili in contesti annidati mantenendo la chiarezza: senza di essi, una funzione non può modificare variabili fuori dal proprio scope se non restituendo valori.
- **Scope isolato nelle comprehension e generator:** Le *list comprehension* e le *generator expression* hanno uno scope locale proprio e non perdono le loro variabili di controllo all'esterno. Ad esempio, dopo aver eseguito una list comprehension `[x for x in range(10)]`, la variabile `x` usata all'interno non rimane definita fuori [22]. Python implementa le comprehension in modo simile a funzioni temporanee: questo design (introdotto in Python 3) evita che variabili temporanee delle comprehension inquinino lo scope circostante o sovrascrivano valori esistenti. Un caso particolare è quello delle comprehension nelle definizioni di classe: poiché lo scope della comprehension è isolato, non ha accesso diretto ai nomi della classe in definizione, il che può causare errori se si tenta di usarli (comportamento voluto per coerenza di scope).

Compilazione in bytecode ed esecuzione

- **Compilazione del codice sorgente in bytecode:** Prima di eseguire effettivamente il codice Python, l'interprete **CPython** lo compila in un formato intermedio chiamato *bytecode*. Ogni volta che avviamo uno script o importiamo un modulo, il codice Python viene tradotto in istruzioni bytecode (rappresentate da numeri operazione e argomenti) che la macchina virtuale di Python può capire [23] . Questo avviene automaticamente e in maniera trasparente per l'utente (non c'è bisogno di un passo di compilazione manuale): l'obiettivo è migliorare la velocità di esecuzione, perché il bytecode è più snello da interpretare rispetto al sorgente testuale, ed è indipendente dalla piattaforma [23] .
- **Python Virtual Machine (PVM):** Il bytecode Python viene eseguito da un *interprete virtuale* interno, spesso chiamato Python Virtual Machine. La PVM è una macchina virtuale stack-based che legge ed esegue una istruzione bytecode alla volta [24] . Ogni istruzione bytecode rappresenta un'operazione elementare (caricare un valore, sommare, chiamare una funzione, ecc.) e manipola uno **stack** di esecuzione interno. Questo strato di astrazione permette al codice Python di girare su qualsiasi sistema per cui esista un interprete Python, fungendo da tramite tra il programma Python e l'hardware reale [25] . In pratica, Python *non esegue direttamente* il codice sorgente, ma il sorgente -> viene compilato in bytecode -> che viene eseguito dalla PVM.
- **Fasi di parse ed esecuzione:** Internamente, l'interprete segue diverse fasi quando esegue un programma. Prima effettua il **parsing** lessicale e sintattico: il sorgente viene spezzato in token e trasformato in un **AST** (Abstract Syntax Tree), una rappresentazione ad albero della struttura del codice [26] [27] . Segue un'eventuale analisi semantica e ottimizzazione semplice dell'AST [28] (ad esempio risoluzione di costanti, controllo di nomi), dopodiché l'AST viene **compilato** in bytecode vero e proprio [29] . A questo punto, l'interprete entra nella fase di **esecuzione**: percorre le istruzioni bytecode con la sua macchina virtuale, effettuando le operazioni richieste passo dopo passo. Questo pipeline (parse -> compile -> execute) spiega perché certi errori di sintassi vengono rilevati prima dell'esecuzione, mentre altri problemi (come NameError, TypeError) emergono solo quando il bytecode corrispondente viene eseguito.
- **Esempio di bytecode generato:** Consideriamo il codice Python:

```
a = 10
b = 20
c = a + b
print(c)
```

Durante la compilazione, questo sarà tradotto in una sequenza di istruzioni bytecode come:

```
0 LOAD_CONST 10    ; carica il valore costante 10
2 STORE_NAME a     ; assegna 10 al nome a
4 LOAD_CONST 20    ; carica 20
6 STORE_NAME b     ; assegna 20 al nome b
8 LOAD_NAME a      ; carica il valore di a
10 LOAD_NAME b     ; carica il valore di b
12 BINARY_ADD      ; somma a + b
14 STORE_NAME c    ; assegna il risultato al nome c
16 LOAD_NAME print ; carica la funzione print
18 LOAD_NAME c     ; carica il valore di c
20 CALL_FUNCTION 1 ; chiama print con 1 argomento
22 POP_TOP         ; elimina il valore di ritorno di print
```

```
24 LOAD_CONST None ; carica None (valore di ritorno implicito)
26 RETURN_VALUE    ; termina la funzione/script
```

Ogni istruzione bytecode è identificata da un indice (offset) e compie un'azione. Ad esempio `LOAD_CONST 10` mette il valore `10` sullo stack, `BINARY_ADD` sostituisce i due elementi in cima allo stack con la loro somma, e così via [30] [31]. Questo esempio illustra come un codice sorgente semplice venga scomposto in passi molto elementari che la VM esegue in sequenza.

- **Ciclo di esecuzione interprete (fetch-decode-execute):** La PVM esegue il bytecode tramite un loop in cui ripete: preleva la prossima istruzione, la decodifica e la esegue. Internamente, un program counter tiene traccia dell'istruzione corrente. Questo design *interpretativo* è facile da portare su diverse piattaforme, ma aggiunge un certo overhead: ogni operazione di alto livello in Python coinvolge più istruzioni bytecode e quindi più iterazioni del loop dell'interprete. In CPython, questo loop è implementato in C (nel file `ceval.c`), quindi è altamente ottimizzato, ma comunque le performance di Python sono inferiori a quelle di un equivalente programma compilato direttamente in linguaggio macchina, proprio perché c'è questa "traduzione al volo" passo passo invece che esecuzione nativa pura.

Prestazioni e ottimizzazioni del compilatore

- **Vantaggi del bytecode (portabilità e rapidità):** L'uso del bytecode offre due grandi benefici: **indipendenza dalla piattaforma** ed **efficienza di esecuzione**. Essendo il bytecode una rappresentazione intermedia non legata all'hardware, un programma Python può girare su qualsiasi sistema con un interprete compatibile, senza ricompilazioni [32]. Inoltre, il bytecode è più compatto e vicino al linguaggio macchina rispetto al sorgente: il Python Virtual Machine può quindi interpretarlo molto più velocemente di quanto farebbe direttamente con il codice sorgente ad alto livello [33]. In altre parole, saltando ogni volta le fasi di parsing e costruzione dell'AST (se il codice non cambia) e avendo istruzioni semplificate, si riduce il lavoro runtime e si velocizza l'avvio e l'esecuzione dello script [34] [35]. (Da notare che, pur caricando il bytecode da cache, la **velocità di esecuzione** del programma in sé rimane la stessa; il guadagno è nei tempi di import e startup).
- **Ottimizzazioni in fase di compilazione:** Durante la generazione del bytecode, l'interprete applica alcune ottimizzazioni automatiche. Ad esempio, effettua il *constant folding*: espressioni costituite da soli letterali (come `2+3` o tuple di costanti) possono essere valutate dal compilatore stesso e sostituite direttamente con il risultato, riducendo il lavoro a runtime. Il compilatore inoltre elimina istruzioni bytecode ridondanti e salti inutili, snellendo il flusso di esecuzione [36]. Alcune di queste ottimizzazioni avvengono a livello di AST, altre a livello di *peephole optimizer* sul bytecode. L'obiettivo è migliorare le prestazioni senza alterare la semantica: ad esempio, una stringa letterale ripetuta più volte nel codice viene interna e referenziata una sola volta; sequenze di carichi e immediati store possono venire combinate. Queste ottimizzazioni sono volutamente conservative (Python privilegia la correttezza e la trasparenza), ma contribuiscono a rendere il bytecode più efficiente da eseguire.
- **Cache dei bytecode compilati (`__pycache__`):** Quando importi un modulo, Python cerca di evitare di ricompilare inutilmente il codice sorgente ad ogni esecuzione. Se hai già importato quel modulo in precedenza e il file `.py` non è cambiato, l'interprete caricherà una versione compilata in bytecode dal file `.pyc` corrispondente nella cartella `__pycache__` [37]. In pratica, alla prima importazione di un modulo, Python crea nella directory `__pycache__` un file `.pyc` ottimizzato per la versione corrente dell'interprete. Ai run successivi, rilevando che il timestamp del sorgente combacia con quello registrato nel `.pyc`, salterà la fase di parsing/compilazione e userà direttamente il bytecode cache, guadagnando tempo. Questo meccanismo accelera soprattutto i programmi grandi con molti import. (Nota: l'uso del bytecode cache

influisce sul tempo di *caricamento* dei moduli, non sulle performance del codice una volta importato, che dipendono comunque dalla velocità dell'interprete nel far girare il bytecode).

- **Cache dei moduli importati (singleton dei moduli):** A complemento della cache su disco, Python mantiene anche una **cache in memoria** dei moduli importati, accessibile via `sys.modules` [38]. Ciò significa che se uno stesso modulo viene importato più volte (anche con nomi diversi, ad esempio tramite importazioni multiple in parti diverse del programma), in realtà Python eseguirà il codice del modulo una sola volta e poi riutilizzerà l'oggetto modulo già caricato. L'oggetto modulo funge da *singleton*: le successive import dello stesso modulo restituiscono il riferimento all'istanza esistente (a meno che non si usi esplicitamente `importlib.reload()` per forzarne il ricaricamento). Questo design evita duplicazioni e potenziali inconsistenze, oltre a velocizzare ulteriormente le importazioni successive poiché né la compilazione né l'esecuzione del modulo vengono ripetute.
- **Niente compilazione a codice macchina (assenza di JIT):** L'interprete CPython di default *non* converte il bytecode in codice macchina nativo a runtime (cioè non ha un JIT compiler incorporato). Invece, esegue il bytecode tramite l'interprete virtuale per l'intera durata del programma [39]. Questa scelta, dovuta a ragioni storiche e di semplicità, punta sulla massima portabilità e sull'affidabilità dell'esecuzione, a scapito di possibili ottimizzazioni specifiche hardware. Alcune implementazioni alternative di Python (ad esempio **PyPy**) includono un JIT che traduce dinamicamente alcune parti critiche in codice nativo per velocizzarle. CPython però finora ha privilegiato una implementazione più semplice e prevedibile, delegando le prestazioni a miglioramenti incrementali dell'interprete stesso (scritto in C) e all'uso di estensioni in linguaggi compilati per operazioni pesanti. In sintesi, il bytecode Python è *interpretato* e non "turbo-compilato" in assembly durante l'esecuzione – una differenza chiave rispetto a linguaggi come Java o C# che hanno macchine virtuali con JIT.

Gestione della memoria e garbage collection

- **Reference counting immediato:** Python adotta un sistema di **garbage collection automatica** per gestire la memoria. In CPython, il meccanismo principale è il *reference counting*: ogni oggetto tiene traccia del numero di riferimenti ad esso esistenti [40]. Ogni volta che un nome (o un contenitore) punta a quell'oggetto, il contatore aumenta; quando un riferimento esce dallo scope o viene riassegnato, il contatore diminuisce. Quando il reference count di un oggetto scende a zero (nessuno lo referenzia più), l'oggetto viene immediatamente distrutto e la memoria liberata. Questo significa che tipicamente gli oggetti vengono deallocati *appena* diventano inutilizzati, il che riduce il rischio di consumare memoria inutilmente. Ad esempio, nel codice:

```
x = []   # crea una lista, refcount=1
y = x    # secondo riferimento, refcount=2
del x    # elimina riferimento x, refcount=1
del y    # elimina riferimento y, refcount=0, la lista viene liberata
subito
```

  La lista viene deallocata immediatamente dopo l'ultimo `del`. Il reference counting fornisce un comportamento prevedibile (deterministico) per la maggior parte delle deallocazioni in Python.
- **Garbage collector per oggetti referenziati ciclicamente:** Il reference counting da solo non può gestire correttamente strutture dati circolari (ad esempio due oggetti che puntano l'uno all'altro e non sono più accessibili dall'esterno). Per questo, CPython integra un *garbage collector aggiuntivo* che rileva e raccoglie i cicli di riferimento. Periodicamente (o su richiesta, tramite il modulo `gc`), l'interprete analizza gli oggetti allocati alla ricerca di gruppi ciclici isolati e li

dealloca se nessun riferimento esterno li può raggiungere [40]. Questo garbage collector opera in modo generazionale (dividendo gli oggetti per "anzianità" per ottimizzare le scansioni) e può essere disabilitato o invocato manualmente all'occorrenza. La presenza di due livelli di GC (reference count immediato e collezione di cicli ritardata) garantisce sia prontezza nella liberazione della memoria nella maggior parte dei casi, sia la capacità di recuperare memoria in situazioni che altrimenti produrrebbero *memory leak*.

- **Finalizzazione e metodo `__del__`:** Python permette di definire un metodo speciale `__del__(self)` in una classe, chiamato *distruttore*, che l'interprete invoca quando un oggetto di quella classe sta per essere garbage-collectato. Tuttavia, l'istante esatto in cui `__del__` viene eseguito non è garantito in modo rigoroso: per oggetti gestiti da reference count `__del__` viene chiamato immediatamente al raggiungimento di refcount zero, ma in presenza di cicli o con altri interpreti, la distruzione può essere posticipata o non avvenire affatto se il programma termina prima [41]. Inoltre, cicli di oggetti che implementano `__del__` non vengono raccolti automaticamente per evitare esecuzioni indeterminate dei distruttori. Per queste ragioni, `__del__` è da usare con cautela. In generale, è preferibile usare costrutti come `with` o i metodi di contesto (`__enter__/__exit__`) per garantire il rilascio tempestivo di risorse esterne (file, connessioni) anziché fare affidamento sul GC. Python infatti **non garantisce** quando (o se) un oggetto verrà distrutto se rimangono riferimenti ciclici o se il programma chiude prima [42]. Quindi, se un oggetto gestisce risorse cruciali, meglio prevedere un metodo esplicito di chiusura (ad es. `close()`) o usare un context manager.
- **Free list e memoria ottimizzata:** L'interprete Python implementa varie ottimizzazioni interne per la gestione della memoria. Per alcuni tipi di oggetti come interi piccoli, float, frame delle funzioni, ecc., CPython utilizza delle *free list* (liste di oggetti precedentemente deallocati tenuti pronti a essere riutilizzati) o degli allocator specializzati. Ad esempio, può conservare in cache alcuni oggetti numerici liberati di recente invece di restituirli al sistema operativo, così da riutilizzarli prontamente per future allocazioni. Questo a volte può portare a comportamenti inattesi se osservati superficialmente: ad esempio, due interi di valore uguale potrebbero essere in realtà lo stesso oggetto in memoria per via del caching, e un confronto con `is` potrebbe dare True [43]. Tali dettagli sono specifici dell'implementazione e trasparenti allo sviluppatore, ma sono pensati per ridurre il overhead di creazione/distruzione di oggetti nelle operazioni intensive. L'idea di base è che Python gestisce la memoria in modo da bilanciare performance e semplicità: lo sviluppatore in genere non deve occuparsi manualmente di allocare o liberare memoria (lo fa l'interprete), ma può fidarsi che sotto il cofano esistono meccanismi per rendere il tutto efficiente.

Eccezioni ed errori

- **Errori di sintassi vs eccezioni runtime:** Python distingue chiaramente i **SyntaxError** (errori rilevati durante l'analisi del codice) dalle **eccezioni** generate durante l'esecuzione. I SyntaxError vengono segnalati prima ancora che il programma inizi a girare (l'interprete non riesce a compilare il bytecode se ci sono errori di sintassi) [44]. Le eccezioni runtime, invece, sono errori che accadono mentre il programma è in esecuzione, ad esempio un tentativo di dividere per zero (`ZeroDivisionError`) o di accedere a una variabile non definita (`NameError`) [45] [46]. Queste eccezioni non sono necessariamente fatali: Python fornisce un meccanismo (try/except) per **gestirle** e continuare l'esecuzione. Se però un'eccezione *propaga* fino al livello principale senza essere intercettata, il programma viene interrotto e l'interprete stampa uno *stack traceback* con il tipo di eccezione e il messaggio associato.
- **Sollevamento delle eccezioni (raise):** Quando si verifica un errore a runtime, l'interprete *lancia* (raise) un'eccezione creando un oggetto che rappresenta l'errore. Questo oggetto contiene informazioni sul tipo di errore (classe dell'eccezione, ad esempio `ValueError`), un eventuale messaggio o altri dati (accesibili via `.args`), e soprattutto un *traceback* che registra la sequenza

di chiamate (stack) che ha portato all'errore [46] [47]. Il traceback è ciò che viene stampato per aiutare a debuggare: esso mostra i file e numeri di linea attraversati fino al punto dell'eccezione. Creare e mantenere il traceback ha un costo, ma è fondamentale per capire gli errori. Python opta per sollevare eccezioni (anziché restituire codici di errore) perché ciò forza il programma a occuparsene o a terminare: è un modo per evitare che errori passino inosservati, rendendo il codice più sicuro e chiaro.

- **Propagazione e unwinding dello stack:** Una volta lanciata, un'eccezione interrompe il flusso normale del programma e l'interprete inizia a *risalire lo stack* alla ricerca di un gestore appropriato. Questo processo è detto *stack unwinding*: l'interprete esce dalla funzione corrente (abbandonando le istruzioni rimanenti in essa) e torna al punto della chiamata; se anche lì non c'è un gestore, prosegue risalendo alla funzione chiamante precedente, e così via. Ogni blocco try/except lungo questo percorso viene ispezionato: se il tipo dell'eccezione corrisponde a uno degli except, l'esecuzione riprende da quel blocco gestore [48]. Se nessuna funzione in call stack gestisce l'errore, si arriva fino al livello principale del programma: a quel punto l'eccezione è *non gestita* e l'interprete stampa il traceback (mostrando tutte le funzioni srotolate) e termina il programma. Questo meccanismo garantisce che un'eccezione o viene trattata esplicitamente, oppure non passa inosservata.
- **Gestione delle eccezioni (try/except):** Per intercettare e gestire un'eccezione, Python offre il costrutto `try/except`. Il codice che potrebbe generare errori viene posto nel blocco `try`, e uno o più blocchi `except` specificano quali eccezioni catturare e come reagire. Quando un'eccezione avviene nel `try`, il resto di quel blocco viene saltato e l'interprete cerca un `except` con un tipo di eccezione corrispondente [49]. Se lo trova, esegue quel blocco `except` e poi continua dopo il try/except, permettendo al programma di proseguire. Se ci sono più `except`, solo il primo con tipo compatibile viene eseguito (l'ordine è quindi importante) [50]. È buona pratica catturare solo le eccezioni previste e lasciare propagare le altre: in questo modo si evita di nascondere bug imprevisti. Inoltre, Python consente di catturare temporaneamente l'oggetto eccezione (ad esempio `except Exception as e:`) per ispezionarlo o loggarlo, e persino di rilanciarlo (`raise`) se necessario per propagarlo ulteriormente.
- **Clausola `finally` e gestione deterministica:** Il blocco `finally` in Python assicura l'esecuzione di codice di pulizia *a prescindere* dall'esito del try. Se presente, il `finally` viene eseguito come ultimo passo quando si esce dal `try`, sia che si esca normalmente (nessuna eccezione) sia che si esca per un'eccezione [51]. In caso di eccezione non gestita nello stesso try, il codice nel finally gira comunque, dopodiché l'eccezione continua a propagare. Questo garantisce, ad esempio, che risorse aperte vengano chiuse:

```
f = open("file.txt")
try:
    # operazioni sul file
    ...
finally:
    f.close()  # viene eseguito sempre, errore o non errore
```

Qui `f.close()` avviene sicuramente. In modo simile, il costrutto `with` (che internamente usa un protocollo con metodi `__enter__` / `__exit__`) è pensato per garantire la chiamata di cleanup (il `__exit__`) automaticamente [52]. In definitiva, `finally` e `with` forniscono sicurezza nella gestione di risorse ed evitano fughe: l'interprete li implementa proprio chiamando in automatico il codice di pulizia al momento giusto, sollevando lo sviluppatore dall'onere di ricordarsi di farlo in ogni possibile percorso di esecuzione (inclusi quelli con errore).

- **Context manager (`with`):** Il **context manager** è un protocollo che l'interprete utilizza per oggetti che devono eseguire azioni di setup e teardown (inizio e fine). Quando si scrive `with`

`obj as var: ...`, Python chiama `obj.__enter__()` all'ingresso del blocco e assegna il suo eventuale risultato a `var`, poi esegue il blocco interno. Alla fine (sia che il blocco termini normalmente, sia per un'eccezione), l'interprete chiama `obj.__exit__(exc_type, exc_value, exc_tb)` passando i dettagli di un'eventuale eccezione occorsa. `__exit__` può gestire l'eccezione (ritornando True per sopprimerla) o semplicemente effettuare pulizie. Questo protocollo spiega ad esempio perché usare `with open("file") as f:` è consigliato: il file ha un context manager che chiude il file in `__exit__`, evitando di lasciarlo aperto per errore [52]. In generale, molti oggetti (file, socket, lock, ecc.) implementano `__enter__/__exit__` per sfruttare il `with` ed assicurare rilascio di risorse. L'interprete supporta nativamente questo pattern, traducendo il `with` in bytecode come se fosse un try/finally che chiama i due metodi al momento giusto.

Concorrenza e GIL

- **Il Global Interpreter Lock (GIL):** Nella versione CPython, l'interprete utilizza un *Global Interpreter Lock*, ossia un lucchetto globale che permette a un solo thread Python per volta di eseguire codice bytecode [53]. In un programma multi-threading Python, anche se ci sono più thread di sistema, solo uno alla volta può essere in stato di esecuzione di istruzioni Python (gli altri o dormono, o aspettano il lock). Il motivo storico di questa scelta è legato alla gestione semplificata della memoria: Python usa il reference counting per il GC, il quale richiede protezione da race condition su quel contatore [54]. Introdurre un lock globale sull'interprete è stata una soluzione semplice per evitare che due thread aggiornino contemporaneamente i contatori di riferimento causando corruzione di memoria. Il GIL garantisce quindi l'integrità dello stato interno di Python senza dover mettere lock fine-grained su ogni oggetto (evitando rischi di deadlock complessi) [55]. **Perché** mantenerlo? Perché rimuoverlo richiederebbe un radicale cambiamento nella gestione di tutti gli oggetti e probabilmente introdurrebbe overhead notevole su single-thread. Il rovescio della medaglia è che il GIL limita il parallelismo su CPU multiple: i thread Python non possono sfruttare più core in parallelo per calcoli puri (CPU-bound), bensì si "intercambiano" sul singolo core. In situazioni IO-bound, invece, mentre un thread è bloccato su I/O, il GIL viene rilasciato e un altro thread Python può eseguire, quindi la concorrenza a livello di I/O è possibile. Da Python 3.2+ il GIL è stato migliorato per essere più *fair* tra thread, ma resta un elemento fondamentale del "come funziona" interno di CPython. (È in corso ricerca per rendere il GIL disattivabile in future versioni sperimentali, ma al 2025 CPython utilizza ancora il GIL di default).

Modello ad oggetti e introspezione

- **Tutto è oggetto:** Python adotta un modello di design fortemente orientato agli oggetti – **ogni entità** in Python è un oggetto con un tipo. Numeri, stringhe, funzioni, classi, moduli… tutti questi sono oggetti che possiedono attributi e metodi. Ad esempio, un numero intero è un'istanza della classe `int` e ha metodi (es. `.__add__`, `bit_length()`), una funzione è un oggetto di classe `function` con attributi come `__name__`, `__doc__`, ecc. Questo approccio unificato semplifica l'implementazione interna (c'è un'infrastruttura comune per gestire identità, tipi e garbage collection) e offre all'utente un potente sistema di introspezione e metaprogrammazione [56]. Il rovescio della medaglia è che anche operazioni semplici (come sommare due numeri) avvengono tramite meccanismi di oggetti (in questo caso invocando metodi speciali sotto al cofano), il che aggiunge un lieve overhead rispetto a operazioni primitive in linguaggi a basso livello. Ma per Python la chiarezza e la coerenza dell'approccio orientato agli oggetti hanno la precedenza sulle micro-ottimizzazioni: in cambio, l'interprete sfrutta implementazioni in C sotto il cofano per rendere veloci le operazioni su questi oggetti.

- **Lookup degli attributi e MRO:** Quando accediamo a un attributo `obj.attr` o invochiamo un metodo `obj.method()`, l'interprete deve determinare a quale definizione di quell'attributo riferirsi. Python implementa un algoritmo di risoluzione degli attributi che, per gli oggetti user-defined, segue l'ordine stabilito dalla **Method Resolution Order (MRO)**. In caso di eredità semplice, questo è banale: cerca nell'istanza, poi nella classe, poi nelle superclassi in ordine gerarchico. Con l'**ereditarietà multipla**, Python utilizza il *C3 linearization* per costruire un ordine lineare di ricerca tra le superclassi [57]. Il risultato è che l'interprete cercherà l'attributo seguendo questo ordine lineare (che garantisce coerenza e che ogni classe base sia visitata al massimo una volta). Ad esempio, se una classe D eredita da B e C, a loro volta sottoclassi di A, l'MRO potrebbe essere `[D, B, C, A, object]` [58] [59]. Una chiamata a `d.some_method()` cercherà `some_method` in D, poi in B, poi C, poi A, ecc., secondo quell'ordine. Questo meccanismo risolve elegantemente il "diamond problem" dell'ereditarietà multipla, assicurando che ogni metodo venga risolto in modo deterministico e senza ambiguità. L'MRO è calcolato staticamente alla definizione della classe, così il lookup a runtime è efficiente (non necessita di ricombinare gli ordini ogni volta). Inoltre, l'uso corretto di `super()` nelle classi con più basi consente di delegare chiamate al metodo successivo nell'MRO, permettendo a tutte le classi base di inizializzarsi o eseguire la loro parte di logica senza duplicazioni o omissioni [60] [61].
- **Metodi speciali e dispatch dinamico:** Molte operazioni sintattiche di Python (come `+`, gli operatori di confronto, l'indicizzazione `obj[key]`, ecc.) sono implementate invocando *metodi speciali* sugli oggetti coinvolti. Ad esempio, valutando l'espressione `x + y`, l'interprete in realtà chiamerà `x.__add__(y)` sotto al cofano [62]. Se la classe di `x` non implementa `__add__` o restituisce `NotImplemented` per il tipo di y, allora Python proverà a chiamare `y.__radd__(x)` [63]. Questo sistema di *dynamic dispatch* basato sul tipo reale degli oggetti rende Python molto flessibile e polimorfo: nuovi tipi possono definire il proprio comportamento con gli operatori sovrascrivendo questi metodi speciali. Ad esempio, una classe personalizzata può definire `__lt__` per stabilire cosa significhi `<` tra sue istanze. L'interprete, anziché avere un rigido set di operazioni built-in, delega agli oggetti la logica, chiamando tali metodi al bisogno. Ciò comporta internamente un piccolo overhead (una risoluzione di metodo) ma offre un sistema unificato e estensibile: *come* avviene un'operazione dipende dal tipo dei suoi operand, e tipi diversi possono avere implementazioni diverse mantenendo la stessa sintassi a livello di codice sorgente.
- **Introspezione runtime:** Python fornisce strumenti potenti per introspezione, sfruttando il fatto che gli oggetti portano con sé molte informazioni. Si può usare la funzione built-in `dir()` per elencare gli attributi di un oggetto, `type()` per ottenere il tipo, e il modulo `inspect` per ispezionare la firma di una funzione, il sorgente di un oggetto se disponibile, o il contenuto di uno stack frame durante un traceback. L'interprete aggiorna costantemente queste informazioni: ad esempio, quando definisci una funzione, l'oggetto funzione creato contiene un riferimento al suo oggetto codice (`func.__code__`), alla tupla di costanti, ai nomi locali, ecc. Questo consente a debugger e altri tool (come IDE o sistemi di test) di esaminare lo stato interno di un programma Python mentre è in esecuzione. Python addirittura permette di modificare a runtime alcuni attributi, rendendo possibile la *metaprogrammazione*: ad esempio si possono aggiungere attributi a oggetti o persino modificare il bytecode di una funzione (sconsigliato, ma teoricamente fattibile). Tutto ciò riflette la filosofia "batteries included" di Python: l'interprete espone molte delle sue strutture interne in modo sicuro, permettendo ai programmatori di ottenere informazioni dettagliate e costruire strumenti sopra il linguaggio stesso. Un rovescio della medaglia è che l'interprete deve talvolta sacrificare un po' di efficienza per mantenere aggiornati questi metadata (come i debug symbols per il traceback), ma il beneficio in termini di usabilità e debug è enorme.
- **Hook di debug e tracing:** Python consente perfino di inserire *hook* durante l'esecuzione di un programma. Funzioni come `sys.settrace()` e `sys.setprofile()` permettono di registrare callback che l'interprete chiamerà ad ogni evento di esecuzione (ad esempio ogni

nuova linea di codice, ogni chiamata di funzione, ogni eccezione lanciata). Questo è il meccanismo alla base di debugger interattivi come `pdb` e di profiler come `cProfile`. Quando attivi un trace hook, l'interprete Python in modalità esecuzione verifica a ogni istruzione se c'è un trace attivo e in caso affermativo chiama la funzione di tracing passando i dettagli dell'evento. Ciò ovviamente rallenta drasticamente l'esecuzione (il debug è più lento del run normale), ma è uno strumento inestimabile per monitorare e comprendere il flusso di un programma. Il design interno dell'interprete, quindi, prevede punti in cui può cedere il controllo a funzioni esterne per scopi di introspezione. Questo è un esempio di come Python privilegi la *trasparenza* e la possibilità di estendere/modificare il comportamento (per debugging, test coverage, ecc.) rispetto alla massima velocità: l'interprete è "hook-friendly", pronto a collaborare con strumenti di alto livello che ne interrogano lo stato.

In conclusione, il "modello mentale" dell'interprete Python è quello di una macchina virtuale orientata agli oggetti, altamente dinamica e riflessiva. Valuta il codice in modo deterministico (sinistra-destra, precedenze), risolve i nomi attraverso scope annidati (LEGB), lega nomi ad oggetti tramite riferimenti (evitando copie implicite), gestisce memoria automaticamente (refcount + GC) e privilegia chiarezza e flessibilità (eccezioni invece di codici di errore, hooking, dynamic dispatch) anche a costo di qualche overhead in più. Questa filosofia progettuale rende Python estremamente produttivo e potente, sia per principianti che per esperti, poiché permette di ragionare sul codice in modo logico/alto livello mentre l'interprete si occupa dei dettagli implementativi sotto il cofano.

---

1 2 11 12 13 14 43 6. Expressions — Python 3.13.5 documentation
https://docs.python.org/3/reference/expressions.html

3 6 Python variables, references and mutability - Bite code!
https://www.bitecode.dev/p/python-variables-references-and-mutability

4 How to Copy Objects in Python: Shallow vs Deep Copy Explained – Real Python
https://realpython.com/python-copy/

5 7 8 9 40 41 42 56 62 63 3. Data model — Python 3.13.5 documentation
https://docs.python.org/3/reference/datamodel.html

10 15 16 Pass by Reference in Python: Background and Best Practices – Real Python
https://realpython.com/python-pass-by-reference/

17 Python Scope & the LEGB Rule: Resolving Names in Your Code – Real Python
https://realpython.com/python-scope-legb-rule/

18 19 20 21 22 4. Execution model — Python 3.13.5 documentation
https://docs.python.org/3/reference/executionmodel.html

23 Python is an interpreted language with a compiler | nicole@web
https://ntietz.com/blog/python-is-an-interpreted-language-with-a-compiler/

24 25 26 27 28 29 30 31 32 33 36 How Python's Bytecode Compilation Works | Medium
https://medium.com/@AlexanderObregon/how-pythons-bytecode-compilation-works-507e5be82185

34 35 37 38 39 What Is the __pycache__ Folder in Python? – Real Python
https://realpython.com/python-pycache/

44 45 46 47 48 49 50 51 52 8. Errors and Exceptions — Python 3.13.5 documentation
https://docs.python.org/3/tutorial/errors.html

53 54 55 What Is the Python Global Interpreter Lock (GIL)? – Real Python
https://realpython.com/python-gil/

57 58 59 60 61 Python’s Multiple Inheritance and Method Resolution Order (MRO)! - DEV Community
https://dev.to/nkpydev/pythons-multiple-inheritance-and-method-resolution-order-mro-51i7