

INFORMATICA & PENSIERO COMPUTAZIONALE

Pensare in codice

Imparare l'informatica e a programmare da zero,
in italiano, con il linguaggio Leo

ROSARIO ZAPPALÀ

leoitalia.com

Pensare in codice

Imparare l’informatica e a programmare da zero, in italiano, con Leo

Rosario Zappalà

Prefazione
Nota dell’autore — Perché Leo
Capitolo 1 — Il primo programma
Capitolo 2 — Variabili e costanti
Capitolo 3 — I numeri
Capitolo 4 — I testi
Capitolo 5 — Liste e dizionari
Capitolo 6 — Intervalli e fette
Capitolo 7 — Prendere decisioni
Capitolo 8 — Ripetere
Capitolo 9 — Le funzioni
Capitolo 10 — Quando qualcosa va storto
Capitolo 11 — Classi e oggetti
Capitolo 12 — Dividere il programma in moduli
Capitolo 13 — File e cartelle
Capitolo 14 — Fare più cose insieme
Capitolo 15 — Le sequenze pigre
Capitolo 16 — Un tour della libreria
Capitolo 17 — Finestre e Leo Studio
Capitolo 18 — App per Android
Capitolo 19 — Cybersecurity difensiva

Capitolo 20 — Distribuire i programmi

Appendice A — Riferimento rapido della sintassi

Appendice B — Errori comuni e come risolverli

Appendice C — Un confronto onesto con Python

Appendice D — Glossario

Le pagine interattive

A mio padre.

Hai preso il linguaggio della vita — quello che a me restava oscuro — e me l'hai tradotto in parole che potevo capire, con la pazienza di chi ha già vissuto. Mi hai prestato i tuoi occhi per guardare il mondo.

Da te ho imparato a trasmutare i problemi in opportunità, le delusioni in forza.

Ciò che sono, lo devo a te: ai tuoi valori, al tuo esempio.

Leone, grazie di essere stato un padre così formidabile.

A te dedico questo libro, e ogni riga del progetto Leo.

Prefazione

L’alfabeto di questo secolo

Prova a contare, in una giornata qualunque, quante volte un programma decide qualcosa al posto tuo. La sveglia al mattino. Il percorso che evita il traffico. Il pagamento alla cassa, la notizia che ti compare per prima, la pratica che un ufficio accetta o respinge, la diagnosi che una macchina suggerisce al medico. Il mondo, oggi, è scritto in software. Non è un’esagerazione da entusiasti: è l’aria che respiriamo quando viviamo e lavoriamo.

Eppure prova a chiedere, in una stanza piena di gente, chi sa *scrivere* un programma. Non usarlo: scriverlo. Si alzano pochissime mani. È una delle sproporzioni più strane del nostro tempo: dipendiamo da qualcosa che quasi nessuno sa fare. Un secolo fa saper leggere e far di conto divideva chi poteva partecipare pienamente alla vita civile da chi ne restava ai margini. Oggi una nuova alfabetizzazione traccia una linea quasi identica. Più silenziosa, ma altrettanto netta.

La spiegazione comoda è che programmare sia roba per pochi, per menti particolari. Non è vero, e questo libro parte proprio dal mandare all’aria quell’idea. La ragione vera è un’altra, più banale e più ingiusta. Per decenni programmare ha preteso due fatiche in una. Imparare a *pensare* in modo nuovo e, nello stesso momento, azzuffarsi con una lingua straniera e con strumenti ostici. Parole chiave in inglese, messaggi d’errore incomprensibili, simboli da decifrare prima ancora di capire l’idea che c’è sotto. Quanta gente capace si è fermata lì, convinta di non essere portata. Stavano solo pagando due volte lo stesso biglietto.

Perché il cuore della programmazione non è la lingua in cui la scrivi. È la **logica**. Prendere un problema grande e spezzarlo in passi piccoli. Dire le cose con precisione, senza “più o meno”. Prevedere cosa può andare storto prima

che accada. Distinguere la causa dall'effetto. Ripetere senza stancarsi e cambiare rotta quando i dati cambiano. Chi impara a programmare, prima ancora di imparare un linguaggio, impara a pensare così. E quel modo di pensare non resta chiuso dentro il computer: torna utile quando organizzi un lavoro, quando smonti un ragionamento sbagliato, quando prendi una decisione guardando i fatti invece delle sensazioni.

Da qui nasce **Leo**, e da qui nasce questo libro. Leo è un linguaggio di programmazione completo in cui tutto è in italiano: le parole, gli errori, la libreria. Non per vezzo nazionale. Per togliere di mezzo una delle due fatiche e lasciarti tutta l'energia per quella che conta davvero: imparare a pensare da programmatore. Quando l'ostacolo della lingua sparisce, la logica resta nuda, visibile, e diventa sorprendentemente più facile da afferrare. E attenzione: Leo non è un giocattolo didattico. È un linguaggio vero, con i numeri esatti, gli oggetti, il parallelismo, novecento funzioni pronte. Quello che impari qui è reale e te lo porti dietro. Domani, se vorrai, passare a un altro linguaggio sarà solo cambiare le parole. Non reimparare a ragionare.

Questo libro, allora, ha due anime che non si pestano i piedi. È un corso per **imparare a programmare** partendo da zero, una riga alla volta, senza dare nulla per scontato. Ed è, nello stesso movimento, un modo per **allenare la logica**. Per lo studente che vuole un vantaggio, per il professionista che vuole finalmente capire gli strumenti che usa ogni giorno, per chiunque abbia sempre pensato "io non sono capace" e meriti di scoprire che non era vero.

Non ti prometto che sarà senza sforzo: imparare qualcosa che vale non lo è mai. Ti prometto però che lo sforzo andrà tutto dove serve. Niente lingua straniera da combattere, niente muri messi lì per nulla. Solo tu, un problema, e il piacere preciso di vederlo risolversi, passo dopo passo, con parole che già conosci.

Si comincia dalla prima riga.

Nota dell’autore — Perché Leo

Mi appassionai all’informatica per caso, l’estate dopo la seconda media.

Avevo passato quasi tutte le vacanze a giocare a *Counter-Strike* con mio cugino, su un piccolo server privato tutto nostro. Anche quel pomeriggio eravamo lì, davanti allo schermo — finché, di colpo, lo schermo diventò blu. Il computer si spense e ricominciò a riavviarsi da solo, all’infinito: si accendeva, ripartiva, si riaccendeva, senza mai arrivare da nessuna parte.

Fu mio cugino a rimetterlo in piedi. Lo formatò, reinstallò i driver, entrò in quel menu scarno e misterioso che è il BIOS e si muoveva lì dentro come se fosse casa sua. Io lo guardavo e basta — ma, mentre guardavo, qualcosa dentro di me si accese. Quando il computer tornò a funzionare non mi bastava: volevo *capire*. Cos’era successo? E soprattutto, cos’aveva fatto lui, davvero, per rimediare? Cominciai a studiare, un argomento dopo l’altro, per dare un nome a quella che a me era sembrata magia.

E quella magia, giorno dopo giorno, diventava più affascinante.

Fu per coltivarla che scelsi l’istituto tecnico industriale, indirizzo informatica. Ed è lì che, quasi subito, capii una cosa che non mi ha più lasciato. Stavamo imparando due cose insieme: una logica nuova — il modo di pensare della programmazione — e una lingua che non era la nostra, l’inglese. Due fatiche sovrapposte. E partivamo svantaggiati: sapevo per esperienza che i nostri coetanei europei, e non solo, con l’inglese erano molto più avanti di noi. Loro combattevano una battaglia sola; noi due.

Allora cominciai a chiedermi: *e se il codice si potesse scrivere in italiano?* Forse sarebbe stato più semplice. Forse si sarebbe potuto cominciare a studiarlo già alle medie, come la matematica. E forse — con l’ambizione un po’ spavalda di quell’età — forse avrebbe aiutato l’Italia a colmare il divario con le grandi potenze, anche là dove quel divario pesa di più: nella sicurezza informatica. Perché sì, già allora ero un ragazzo fissato con l’hacking, e non

sopportavo l'idea che il nostro Paese fosse fra gli ultimi della classifica, invece che fra i primi.

Per anni è rimasta una domanda, di quelle che ti torni a fare senza mai trovare il tempo di rispondere. Poi, nel 2026, ho smesso di chiedermelo e ho cominciato a costruirla.

Leo è la risposta a quel ragazzo seduto in classe. Un linguaggio di programmazione vero, potente come quelli che usano i professionisti, ma che parla italiano: `se`, `per ogni`, `mentre`, `stampa`. Non un giocattolo e non una traduzione di facciata — uno strumento con cui fare sul serio, che però toglie di mezzo l'unico ostacolo che con il saper programmare non c'entra niente: la lingua.

E se si chiama Leo, un motivo più intimo c'è — ma quello l'ho affidato alla pagina della dedica, non a questa.

Questo libro nasce dalla stessa idea. Non voglio insegnarti un linguaggio: voglio insegnarti a *pensare in codice*. La logica che c'è sotto a ogni programma, a ogni app, a ogni sito — spiegata nella tua lingua, con qualcosa da toccare a ogni pagina. Se alla fine guarderai un computer e capirai cosa sta facendo, e perché, avrò vinto.

Buona lettura. E benvenuto nell'alfabeto di questo secolo.

— Rosario Zappalà # Introduzione

Benvenuto. Mettiti comodo, perché stai per imparare a dare ordini a una macchina che farà *esattamente* quello che le dici. Né più né meno. Ed è proprio lì che si nasconde tutto il divertimento, e qualche sano grattacapo. Questo è un libro su **Leo**, un linguaggio di programmazione pensato e scritto in italiano. Se non hai mai scritto una riga di codice, sei nel posto giusto: si parte dalla prima riga e non si dà niente per scontato. Se invece bazzichi già altri linguaggi, qui ti senti subito a casa. Funzioni, classi, moduli, eccezioni,

parallelismo: c'è tutto. Solo che stavolta è raccontato nella lingua che parli ogni giorno.

> 💡 **L'idea — Il programma è una ricetta**
>
> Pensa a una ricetta di cucina: elenca gli ingredienti e i passi nell'ordine giusto, e chi la segue ottiene il piatto. Un programma è la stessa cosa, con una differenza importante: il computer esegue i passi alla lettera, senza mai «capire cosa intendi». Se la ricetta dice «aggiungi il sale» ma non dice quanto, una persona improvvisa; il computer, no. Imparare a programmare è soprattutto imparare a essere precisi.

> ☕ **Pausa — Il serpente che non c'era**
>
> Il linguaggio Python, su cui Leo poggia, non prende il nome dal serpente ma dai **Monty Python**: il suo creatore, Guido van Rossum, era un fan del loro show comico. Morale: anche nei nomi della tecnologia, dietro le cose serissime si nasconde spesso una risata.

Che cos'è Leo

Leo è un linguaggio di programmazione completo in cui le parole chiave, i messaggi d'errore, la libreria e la documentazione sono **tutti in italiano**. Un programma Leo si legge quasi come una frase:

```
se voto >= 18 allora
    scrivi "Promosso"
altrimenti
    scrivi "Da ripassare"
fine
```

Niente da tradurre mentalmente. Non devi passare da `if`, `then`, `else`: le istruzioni dicono quello che fanno, nella lingua in cui pensi. E non è uno scherzo né un giocattolo. Dentro Leo ci sono i tipi di dato seri (interi senza limiti, decimali, denaro esatto, frazioni, date), le funzioni come valori, le chiusure, le eccezioni, i moduli, gli oggetti con l'ereditarietà, il parallelismo su più processori, le sequenze pigre e oltre novecento funzioni di libreria già pronte, dalla statistica al web, dalla crittografia alle immagini.

Leo **gira sull'interprete Python**, e qui ci mettiamo la faccia con onestà: Leo non reinventa la macchina sottostante, si appoggia a una delle piattaforme più solide e collaudate al mondo. Da lì eredita stabilità, portabilità e la possibilità, per chi un giorno la vorrà, di attingere all'immenso ecosistema di pacchetti Python. Ma il linguaggio che scrivi e leggi è Leo: pulito, in italiano, coerente dall'inizio alla fine.

> 🦁 **Curiosità — Il primo programma aveva quasi due secoli**
>
> Nel 1843 la matematica Ada Lovelace pubblicò, in appendice a una memoria tradotta dal francese, quello che è considerato il primo algoritmo pensato per essere eseguito da una macchina: una procedura per calcolare i numeri di Bernoulli sulla macchina analitica di Charles Babbage. Quella macchina non fu mai completata, e per vedere un vero computer si dovette aspettare circa un secolo. Il primo «programma», insomma, fu scritto per un calcolatore che non esisteva ancora.

La filosofia

Tre idee guidano tutto il progetto, e vale la pena nominarle subito perché tornano in ogni capitolo.

L'italiano non è una decorazione. Programmare è, prima di tutto, scrivere per essere capiti: dal computer e dalle persone. Una lingua vicina rende il codice più leggibile a chi lo scrive per la prima volta e più chiaro a chi lo rilegge dopo mesi. Quando un programma Leo si rompe, non ti ritrovi davanti un muro di inglese tecnico. L'errore ti spiega in italiano che cosa è successo, su quale riga, e spesso ti mette già in mano la correzione.

La semplicità è una qualità, non una mancanza. In Leo le cose comuni si scrivono in modo breve e diretto. Stampare a schermo è `scrivi`; sommare i numeri di una lista è `somma`; scrivere un importo in lettere è `euro_in_lettere`. Ma semplice non vuol dire limitato: quando serve potenza, c'è. Il punto è che non devi pagare la complessità in anticipo, prima ancora di averne bisogno.

Niente magia nascosta. Leo preferisce dirti le cose in faccia. Non converte i tipi di nascosto (`"3" + 4` è un errore, non un risultato a sorpresa); non crea variabili per un refuso; non lascia che un calcolo sbagliato passi inosservato. Molti controlli avvengono *prima* di eseguire il programma, così tanti errori li scopri subito invece che a metà di un lavoro lungo.

Come è fatto questo libro

Questo è un libro che si legge **in ordine**. Non è un elenco di riferimento da consultare a salti (per quello ci sono la Guida al linguaggio e le schede di libreria in `docs/`). È un percorso che parte dal primo `scrivi` e arriva, pagina dopo pagina, a costruire applicazioni con finestre, app per il telefono e programmi distribuibili.

Il viaggio è diviso in cinque parti e un gruppo di appendici:

- **Parte I — Le basi.** Il primo programma, le variabili e le costanti, i numeri in tutte le loro forme, i testi, le liste e i dizionari, e come prendere una "fetta" di una sequenza.

- **Parte II — Il flusso.** Come il programma decide (`se`, `scegli`) e ripete (`per ogni`, `mentre`, `ripeti`), come si scrivono le funzioni e come si gestiscono gli errori.
- **Parte III — Organizzare.** Classi e oggetti, l'ereditarietà, la divisione del codice in moduli, e il lavoro con file e cartelle.
- **Parte IV — Potenza.** Il parallelismo, le sequenze pigre e un tour ragionato della libreria, area per area.
- **Parte V — Fare cose vere.** Le interfacce grafiche con Leo Studio, le app per Android, la cybersecurity difensiva e la distribuzione dei programmi.
- **Appendici.** Un riferimento rapido della sintassi, un catalogo degli errori più comuni, un confronto onesto con Python e un glossario.

Ogni capitolo ha la stessa forma: una spiegazione chiara, esempi che puoi copiare, e in fondo un piccolo **«prova tu»** per mettere subito le mani in pasta. Tutti gli esempi di questo libro sono stati **eseguiti davvero** con `python3 -m leo` prima di finire in queste pagine. Quando leggi un output, è l'output reale del programma, non una ricostruzione a memoria.

Che cosa ti serve

Per seguire la prima metà del libro basta Python 3.10 o successivo e la cartella di Leo. Dalla riga di comando, un programma si esegue così:

```
python3 -m leo saluto.leo
```

Nel resto del libro scriveremo semplicemente `leo saluto.leo`: su Windows è il collegamento *Leo - riga di comando* (o `leo.bat`), altrove è `python3 -m leo`. Per i capitoli con le finestre serve in più **PySide6**; lo installeremo a tempo debito. Per ora, apri un editor di testo qualunque: stiamo per scrivere il primo programma.

Una nota di metodo, prima di partire. Imparare a programmare non è mandare a memoria una lista di comandi. È prendere l'abitudine di provare. Tieni aperto un terminale accanto al libro. Quando incontri un esempio, scrivilo (non copiarlo e basta: scriverlo a mano fa la differenza), eseguilo, poi **rompilo** di proposito. Cambia un numero, togli una parentesi, scambia due righe, e guarda che cosa dice Leo. Gli errori, in questo libro, non sono nemici: sono la via più veloce per capire come funziona davvero il linguaggio.

Buona lettura, e buon divertimento.

Capitolo 1 — Il primo programma

Un programma, in fondo, è solo un testo. Una lista di istruzioni che il computer esegue una dopo l'altra, senza discutere. In Leo lo salvi in un file che finisce in `.leo` e lo lanci con un comando. Niente da compilare a mano, niente da configurare. Scrivi, salvi, esegui. Fine della cerimonia.

> **💡 L'idea — Chi esegue non sottintende nulla**
>
> Immagina di dare indicazioni stradali a qualcuno che fa esattamente quello che dici, parola per parola, senza aggiungere niente di suo. Se dimentichi di dire «gira a destra», tira dritto; se dici due volte «prosegui», prosegue due volte. Il computer è questo esecutore fedele e letterale. Non è cattiveria: è l'assenza totale di sottinteso, ed è anche ciò che rende i programmi affidabili una volta scritti bene.

Cominciamo dalla tradizione. Apri un editor di testo: va bene qualunque cosa, persino il Blocco note, purché salvi in **UTF-8**. Scrivi questa riga e salva il file come `ciao.leo`:

```
scrivi "Ciao, mondo!"
```

Ora apri un terminale nella cartella dove hai salvato il file ed esegui:

```
leo ciao.leo
```

Il programma stampa:

```
Ciao, mondo!
```

Congratulazioni: hai scritto ed eseguito il tuo primo programma. Adesso smontiamolo, pezzo per pezzo. In quella riga c'è già parecchio.

> 🦁 **Curiosità — «Ciao, mondo» viene da lontano**
>
> L'usanza di far scrivere al principiante un programma che stampa «Hello, World» risale a un tutorial del linguaggio B scritto da Brian Kernighan nei primi anni Settanta, e divenne celebre con il libro sul linguaggio C che Kernighan firmò con Dennis Ritchie nel 1978. Da allora quasi ogni manuale di programmazione comincia così, in ogni lingua. Il nostro «Ciao, mondo!» è l'edizione italiana di una tradizione lunga mezzo secolo.

`scrivi`: il comando per parlare

`scrivi` è il comando che mostra qualcosa sullo schermo. (Ha anche un nome più tecnico, `stampa`: sono la stessa identica cosa, due parole per lo stesso comando. Nel libro useremo quasi sempre `scrivi`, perché si legge meglio.) Quello che segue `scrivi` è ciò che vuoi mostrare.

In Leo i comandi all'inizio di una riga **non hanno bisogno delle parentesi**: la regola, che approfondiremo via via, è che *le parentesi vogliono dire «calcola»*. Quindi puoi scrivere il testo così com'è, senza nemmeno le virgolette:

```
scrivi Ciao, mondo!
scrivi Oggi imparo a programmare.
```

Questo stampa:

```
Ciao, mondo!
Oggi imparo a programmare.
```

Esiste anche la **forma classica**, con le parentesi e le virgolette: `scrivi("Ciao, mondo!")`. È perfettamente valida e fa esattamente la stessa cosa. La vedrai spesso, perché quando il testo contiene caratteri particolari (parentesi, segni di punteggiatura usati per calcolare) le virgolette tolgono ogni ambiguità. Per ora tieni a mente che entrambe vanno bene; sceglieremo di volta in volta la più chiara.

Ogni `scrivi` manda a capo dopo aver stampato. Più righe di programma diventano più righe sullo schermo:

```
scrivi "Riga uno"
scrivi "Riga due"
```

```
Riga uno
Riga due
```

I commenti: note per gli umani

Non tutto quello che scrivi in un file è destinato al computer. A volte vuoi lasciare una nota a te stesso o a chi leggerà il codice dopo di te. Questi appunti si chiamano **commenti** e Leo li ignora completamente durante l'esecuzione.

Un commento comincia con `#` e arriva fino a fine riga. Per un commento su più righe si usano `#[` e `]#`:

```
# Questo è un commento: Leo lo ignora
scrivi "Il commento qui sopra non viene stampato"
#[ Un commento
   su più righe ]#
scrivi "Fatto"
```

```
Il commento qui sopra non viene stampato
Fatto
```

I commenti sembrano inutili quando il programma è di tre righe. Diventano preziosi quando è di trecento. Prendi l'abitudine di spiegare *perché* fai una cosa, non *cosa* fai: il «cosa» lo dice già il codice.

Inserire valori nel testo

Finché scriviamo solo frasi fisse, il programma dice sempre la stessa cosa. Si fa interessante quando il testo cambia a seconda dei dati. In Leo, dentro un testo tra virgolette, puoi inserire un valore mettendolo tra **graffe** `{…}`:

```
variabile nome = "Giulia"
variabile anni = 10
scrivi "Ciao {nome}, hai {anni + 1} anni"
```

```
Ciao Giulia, hai 11 anni
```

Tra le graffe puoi mettere non solo una variabile ma qualunque calcolo: `{anni + 1}` viene calcolato e il risultato prende il suo posto. Questo si chiama **interpolazione** e sarà il nostro pane quotidiano; gli dedicheremo più spazio nel capitolo sui testi.

Nella forma senza virgolette lo stesso risultato si ottiene con le parentesi tonde che, ricordi?, vogliono dire «calcola»:

```
scrivi Ciao (nome), hai (anni + 1) anni
```

Le due forme producono la stessa riga. Useremo di solito quella con le graffe, perché mostra a colpo d'occhio dove finisce il testo e dove comincia il valore.

Chiedere qualcosa a chi usa il programma

Un programma che parla soltanto è un monologo. Per ascoltare c'è `chiedi`: mostra una domanda e restituisce, come testo, quello che la persona digita e invia con Invio.

```
variabile nome = chiedi "Come ti chiami?"
scrivi "Piacere di conoscerti, {nome}!"
```

Eseguendo e rispondendo `Ada`, il programma stampa:

```
Piacere di conoscerti, Ada!
```

`chiedi` restituisce **sempre un testo**, anche se la persona scrive un numero. Per trasformarlo in un numero vero useremo `intero(...)` o `decimale(...)`: lo vedremo tra pochi capitoli. Per ora ci basta sapere che il programma può dialogare.

Dove comincia un programma: `inizio` … `fine`

I programmi piccoli si leggono dall'alto in basso e partono dalla prima riga. Man mano che crescono, conviene segnare con chiarezza *da dove* parte il programma, tenendo le funzioni (che vedremo nel Capitolo 9) separate. Per questo Leo offre il blocco `inizio` … `fine`:

```
inizio
    scrivi "Il programma parte da qui."
    saluta("Leo")
fine

funzione saluta(nome)
    scrivi "Ciao, {nome}!"
fine
```

```
Il programma parte da qui.
Ciao, Leo!
```

Il codice dentro `inizio` è quello che deve partire subito; le funzioni stanno fuori, prima o dopo, pronte quando il blocco le chiama (nota che `saluta` è definita *sotto* `inizio` ma viene usata lo stesso: il blocco parte dopo che tutto il file è stato letto). Il blocco `inizio` è facoltativo, e i programmi senza di esso funzionano benissimo, ma è la forma consigliata per i programmi nuovi. È quella che adotteremo quando le cose si faranno serie.

Prova tu

1. Scrivi un programma che saluta te stesso per nome e poi stampa, su una riga separata, in che anno pensi di finire questo libro.
2. Modifica l'esempio di `chiedi` perché faccia due domande — il nome e la città — e poi stampi una frase sola che usa entrambe le risposte, con l'interpolazione `{…}`.
3. Prendi uno qualunque dei programmi di questo capitolo, togli una virgoletta di chiusura e rieseguilo. Leggi con calma il messaggio d'errore: su quale riga ti manda? Che cosa ti propone? Poi rimetti la virgoletta a posto.

Capitolo 2 — Variabili e costanti

Un programma che non ricorda nulla può fare ben poco. Le **variabili** sono i cassetti in cui il programma mette i dati per ritrovarli più tardi: un punteggio, un nome, una lista della spesa. Ogni cassetto ha un nome che scegli tu, e dentro ci metti un valore.

Dichiarare e cambiare

In Leo una variabile si **dichiara** la prima volta con la parola `variabile`, seguita dal nome, un segno di uguale e il valore iniziale. Dopo, per cambiarle valore, basta il nome e il segno di uguale:

```
variabile punti = 0
scrivi "All'inizio: {punti}"
punti = 10
scrivi "Dopo aver segnato: {punti}"
punti += 5
scrivi "Con il bonus: {punti}"
```

```
All'inizio: 0
Dopo aver segnato: 10
Con il bonus: 15
```

Nota la differenza. La **prima** volta serve `variabile`, perché stai creando il cassetto; le volte successive no, perché il cassetto esiste già e lo stai solo riempiendo di nuovo. L'ultima riga usa `+=`, una scorciatoia che vuol dire «aggiungi a quello che c'è già»: `punti += 5` è identico a `punti = punti + 5`. Esistono anche `-=`, `*=` e `/=`, con lo stesso senso.

> 💡 **L'idea — L'etichetta e il contenuto**
>
> Una variabile è come un armadietto con un'etichetta. Il nome che scegli è l'etichetta incollata sullo sportello; il valore è ciò che hai messo dentro. Quando scrivi `punti = 10` stai cambiando il contenuto dell'armadietto, non l'etichetta: il nome resta lo stesso, cambia solo quello che custodisce. Ecco perché puoi riempirlo e svuotarlo quante volte vuoi senza doverlo ri-dichiarare.

Questa regola, `variabile` solo alla nascita, non è una formalità noiosa. Ti protegge dai refusi. Se scrivi per sbaglio `pnuti = 3` pensando a `punti`, Leo non crea di nascosto una variabile nuova. Si ferma e te lo dice.

```
ErroreNome alla riga 1, colonna 1 di programma.leo

    1 | pnuti = 3
      | ^^^^^

`pnuti` non esiste ancora: una variabile nuova si crea con `variabile`.
Suggerimento: Scrivi `variabile pnuti = 3`; dopo, per cambiarle valore,
basta `pnuti = ...`.
```

In un linguaggio che crea le variabili al volo, quel refuso sarebbe passato inosservato e il bug si sarebbe manifestato molto più tardi, molto più lontano.

Stampare il valore, non il nome

Qui c'è una trappola gentile in cui tutti inciampano almeno una volta. Ricordi che `scrivi` senza parentesi tratta quello che segue come testo? Allora questa riga:

```
scrivi punti
```

…non stampa il valore di `punti`: stampa la parola `punti`. Per mostrare il *valore* hai due strade, entrambe corrette:

```
scrivi(punti)              # forma con parentesi: calcola e stampa il
valore
scrivi "Punti: {punti}"    # interpolazione: il valore entra nel testo
```

Tienilo a mente: **le parentesi (o le graffe dentro un testo) vogliono dire «calcola»**. Senza di esse, `scrivi` prende le parole alla lettera. È la stessa regola del capitolo scorso, vista da un'altra angolazione. Dopo un po' diventa naturale.

Le costanti: valori che non devono cambiare

A volte un valore non deve mai cambiare per tutta la vita del programma: l'aliquota IVA, il numero di giocatori, il titolo di una finestra. Per questi casi c'è la **costante**, che si dichiara con `costante`:

```
costante IVA = 22
variabile prezzo = 100
scrivi "Prezzo con IVA: {prezzo + prezzo * IVA / 100}"
```

```
Prezzo con IVA: 122.0
```

(Il risultato è `122.0`, con il punto e lo zero: la divisione `/` in Leo dà sempre un numero decimale. Ne parleremo nel prossimo capitolo.)

La differenza con una variabile è semplice: una costante, una volta dichiarata, **non si può più cambiare**. Se ci provi, Leo ti ferma:

```
ErroreValore alla riga 2, colonna 1 di programma.leo

    2 | IVA = 10
      | ^^^

`IVA` è una costante: non si può cambiare il suo valore.
Suggerimento: Se deve cambiare, dichiarala con `variabile` invece di
`costante`.
```

Non è una punizione: è una rete di sicurezza. Dichiarare costante un valore che non deve cambiare documenta la tua intenzione e impedisce che un errore distratto lo modifichi. Usa `costante` ogni volta che puoi; `variabile` solo quando il valore deve davvero cambiare.

> 🦁 **Curiosità — La costante che definisce il metro**
>
> La velocità della luce nel vuoto vale esattamente 299.792.458 metri al secondo. «Esattamente» non è un modo di dire: dal 1983 quel numero è fissato per convenzione, e il metro viene definito a partire da esso, come la distanza che la luce percorre in una frazione precisa di secondo. In natura come nel codice, una costante serve proprio a questo: dare un nome stabile a un valore su cui tutto il resto può appoggiarsi.

Più valori in una volta sola

Una dichiarazione (o un'assegnazione) può riguardare più nomi insieme, separati da virgole. È comodo quando i valori arrivano da una lista o quando vuoi assegnarne diversi in una riga:

```
variabile nome, anni = ["Beatrice", 24]
scrivi "{nome} ha {anni} anni"
variabile a, b = 1, 2
a, b = b, a
scrivi "a vale {a}, b vale {b}"
```

```
Beatrice ha 24 anni
a vale 2, b vale 1
```

L'ultima riga merita attenzione: `a, b = b, a` **scambia** i due valori senza bisogno di una variabile d'appoggio. Funziona perché Leo calcola prima tutti i valori a destra e solo dopo li assegna. Se nomi e valori non sono lo stesso numero, l'errore te lo dice con precisione: *«Ci sono 2 nomi ma 3 valori.»*

Aumentare e diminuire a parole

Oltre a `+=` e `-=`, Leo offre due parole che si leggono come una frase: `aumenta` e `diminuisci`. Dopo `di` scrivi di quanto; senza `di`, vuol dire di uno:

```
variabile punti = 0
variabile vite = 3
aumenta punti di 10
diminuisci vite
scrivi "Punti: {punti}, vite: {vite}"
```

```
Punti: 10, vite: 2
```

Sono perfettamente equivalenti a `punti += 10` e `vite -= 1`: scegli la forma che in quel punto si legge meglio.

Dove vive una variabile

Ogni blocco di codice ha un suo spazio, e lo vedremo meglio parlando di cicli e funzioni. Una variabile dichiarata *dentro* un blocco con `variabile` esiste solo lì dentro; fuori non si vede. Per cambiare invece una variabile che esisteva già prima, basta il segno di uguale, senza `variabile`:

```
variabile totale = 0
per ogni n in [1, 2, 3] fai
    variabile doppio = n * 2     # esiste solo dentro il ciclo
    totale += doppio             # cambia la variabile di fuori
fine
scrivi "Il totale è {totale}"
```

```
Il totale è 12
```

La variabile `doppio` nasce e muore a ogni giro del ciclo; `totale`, dichiarata fuori, sopravvive e accumula. Questa separazione si chiama **visibilità** (o *ambito*) e tiene i programmi ordinati: le variabili temporanee non «sporcano» il resto del codice. Torneremo sull'argomento quando parleremo di funzioni, dove conta davvero.

I nomi

Qualche parola sui nomi delle variabili, perché nomi buoni valgono più di molti commenti. Un nome inizia con una lettera o un trattino basso `_` e prosegue con lettere, cifre e trattini bassi. Le lettere accentate sono ammesse, quindi `età` e `città` sono nomi legittimi. Scegli nomi che dicono *cosa contengono*: `media_voti` è meglio di `m`, `prezzo_finale` è meglio di `x`. Il tuo io del mese prossimo ti ringrazierà.

Prova tu

1. Dichiara una costante con il tuo anno di nascita e una variabile con l'anno corrente, poi stampa quanti anni compi quest'anno.

2. Crea tre variabili `a`, `b`, `c` con tre valori a tuo piacere su una riga sola, poi ruotale: `a` prende il valore di `b`, `b` quello di `c`, `c` quello di `a`. Stampa i tre valori prima e dopo.
3. Prova a scrivere `scrivi prezzo` (senza parentesi) in uno degli esempi e guarda cosa succede. Poi correggilo in `scrivi(prezzo)`. Hai capito la differenza?

Capitolo 3 — I numeri

I numeri sono dappertutto. Punteggi, prezzi, misure, conteggi: aprine uno qualunque e ci trovi dei numeri sotto. Leo li prende sul serio e ti dà più tipi di numero, ciascuno tagliato per un mestiere preciso. In questo capitolo li incontri tutti: gli interi, i decimali, il denaro e le frazioni.

> ☕ **Pausa — Mezzo byte si chiama «morso»**
>
> Otto bit formano un **byte**. E quattro bit — mezzo byte — hanno un nome ufficiale e buffo: **nibble**, cioè «morsetto» in inglese. È un gioco di parole: *byte* si pronuncia come *bite* (morso), quindi mezzo boccone è giusto un *morsetto*. Qualcuno, agli albori dell'informatica, aveva fame e senso dell'umorismo.

Interi e decimali

Un **intero** è un numero senza virgola: `42`, `-7`, `0`. Un **decimale** la virgola ce l'ha, ma in Leo si chiama **punto**: scrivi `3.14`, non `3,14`. Vale così in quasi tutti i linguaggi di programmazione, e ti ci abitui in un attimo. La virgola, in Leo, fa un altro lavoro: separa le cose, gli elementi di una lista, gli argomenti di una funzione. Mai i decimali.

> 💡 **L'idea — Contare o misurare**
>
> Gli interi sono come contare oggetti: tre mele sono tre mele, un numero netto, senza sfumature. I decimali sono come misurare con un righello: dici «2,7 centimetri», ma sotto la tacca c'è sempre un pelo in più o in meno che non riesci a leggere. Per questo gli interi sono esatti per natura, mentre i decimali portano con sé, quasi sempre, una piccola approssimazione, come vedremo tra poco con i soldi.

Le operazioni le conosci già dalla scuola:

```
scrivi(7 + 2)
scrivi(7 - 2)
scrivi(7 * 2)
scrivi(7 / 2)
scrivi(7 // 2)
scrivi(7 % 3)
scrivi(2 ^ 10)
```

```
9
5
14
3.5
3
1
1024
```

Tre di queste meritano una parola. La divisione `/` dà **sempre** un decimale, anche quando il conto tornerebbe intero: `7 / 2` è `3.5`, e `10 / 2` sarebbe `5.0`. Vuoi solo il *quoziente* intero, cioè quante volte ci sta senza badare al resto? Usi la doppia barra `//`: `7 // 2` è `3`. Il **resto** di quella divisione te lo dà `%`: `7 % 3` è `1`, sette diviso tre fa due con resto uno. La potenza si scrive con l'accento circonflesso `^`: `2 ^ 10` è `1024`.

Gli interi di Leo **non hanno limiti di grandezza**. Puoi calcolare numeri enormi senza che «traboccino», cosa che in molti linguaggi manda fuori strada i risultati:

```
scrivi(1_000_000 + 1)
scrivi(2 ^ 100)
```

```
1000001
1267650600228229401496703205376
```

Hai notato il trattino basso in `1_000_000`? Serve solo a te, per leggere meglio le cifre. Leo lo ignora. Scrivi `1_000_000` invece di `1000000` e ringrazierai te stesso quando dovrai rileggerti.

Qualche funzione utile

Alcune operazioni comuni sono funzioni già pronte, sempre a portata di mano:

```
variabile pi = 3.14159
scrivi(arrotonda(pi, 2))
scrivi(arrotonda(2.5))
scrivi(radice(16))
scrivi(assoluto(-7))
scrivi(minimo([3, 9, 4]))
scrivi(massimo([3, 9, 4]))
scrivi(somma([10, 20, 30]))
```

```
3.14
3
4.0
7
3
9
60
```

`arrotonda(numero, cifre)` arrotonda a quante cifre decimali vuoi; senza indicare le cifre ti dà un intero, e arrotonda le metà *lontano da zero* (`arrotonda(2.5)` è `3`, come ti hanno insegnato a scuola, non come fa di nascosto qualche altro linguaggio). `radice` è la radice quadrata, `assoluto` toglie il segno, `minimo` e `massimo` scelgono tra i valori di una lista, `somma` li addiziona tutti. Nel Capitolo 15, parlando della libreria, ne incontreremo molte altre, su tutte la statistica (`media`, `mediana`, `deviazione`).

Scrivere i numeri come si deve

Un conto è calcolare `1234.5`. Un altro è *mostrarlo* a una persona. Un prezzo vuole due decimali; un numero grande si legge meglio con i punti delle migliaia. Dentro un testo, dopo l'espressione metti i due punti e un formato:

```
variabile prezzo = 1234.5
scrivi "{prezzo:2}"
scrivi "{1234567:migliaia}"
scrivi "{prezzo:migliaia 2}"
```

```
1234.50
1.234.567
1.234,50
```

`{prezzo:2}` forza due cifre decimali (`1234.50`, con lo zero finale che un prezzo deve avere). `{n:migliaia}` mette i punti delle migliaia all'italiana. `{prezzo:migliaia 2}` fa tutte e due le cose, ed è il formato perfetto per gli importi: `1.234,50`, punto per le migliaia e virgola per i decimali, identico a una fattura. Questa formattazione vale solo per i numeri. Se sbagli il formato, Leo te lo dice e ti suggerisce quello giusto.

Il denaro: i centesimi non si perdono

Qui arriviamo a una delle idee più belle di Leo. I numeri decimali del computer, quelli che hai visto sopra, seguono uno standard tecnico (si chiama IEEE-754) che è velocissimo ma *non è esatto* con i decimali. Ti suona astratto? Guarda qui:

```
scrivi(0.1 + 0.2)
```

```
0.30000000000000004
```

Dieci centesimi più venti centesimi dovrebbero fare trenta centesimi tondi. E invece spunta quella coda di zeri con un `4` appiccicato in fondo. Per i calcoli scientifici è irrilevante. Per i soldi è inaccettabile. Per questo Leo ha un tipo pensato apposta per il denaro, `soldi`, che tiene i centesimi **esatti**:

```
variabile conto = soldi("10.00")
variabile spesa = soldi("3.50")
scrivi(conto.meno(spesa))
scrivi(soldi("0.10").piu(soldi("0.20")))
```

```
6,50 €
0,30 €
```

Niente code di zeri: trenta centesimi sono trenta centesimi, punto. Nota due cose. Un importo si crea da un testo (`soldi("10.00")`), così non eredita l'imprecisione dei decimali fin dalla culla; e le operazioni si scrivono a parole, `.piu(...)`, `.meno(...)`, perché qui non facciamo aritmetica qualunque, facciamo i conti della serva e li vogliamo giusti al centesimo. Il risultato esce già all'italiana, con la virgola e il simbolo dell'euro.

> **Curiosità — Quando un'unità sbagliata costa una sonda**
>
> Nel 1999 la NASA perse la sonda Mars Climate Orbiter, un progetto da circa 327 milioni di dollari, proprio mentre si avvicinava a Marte. La causa fu banale e amara: un gruppo di lavoro aveva usato le unità di misura anglosassoni e un altro il sistema metrico, e nessuno dei due convertì i numeri. I conti erano «giusti», ma nelle unità sbagliate. È il promemoria perfetto del perché, con i numeri, contano tanto il tipo e l'unità quanto il valore.

C'è anche un tocco di eleganza tutto italiano: `euro_in_lettere` scrive un importo in parole, come sugli assegni.

```
scrivi(euro_in_lettere(19.90))
```

```
diciannove euro e novanta centesimi
```

Le frazioni: niente approssimazioni

Un terzo più un sesto fa esattamente un mezzo. Prova a dirlo con i decimali e ti ritrovi `0.333...` più `0.166...`, con tutte le approssimazioni al seguito. Leo ha il tipo `frazione`, che tiene i numeri razionali **esatti** e li semplifica da solo:

```
variabile meta = frazione(1, 2)
variabile terzo = frazione(1, 3)
scrivi(meta.piu(terzo))
scrivi(meta.per(terzo))
scrivi(frazione(2, 4))
```

```
5/6
1/6
1/2
```

Un mezzo più un terzo fa cinque sesti, esatto; un mezzo per un terzo fa un sesto; e `frazione(2, 4)` si semplifica da sé in `1/2`. È il tipo giusto per la matematica, la didattica e ogni volta che un'approssimazione sarebbe un tradimento.

Trasformare un tipo nell'altro

Leo non converte mai i tipi di nascosto: `"3" + 4` è un errore, non `7` né `"34"`. Questa severità ti salva da una montagna di bug. Ogni tanto, però, una conversione la vuoi sul serio, per esempio quando `chiedi` ti ha restituito un testo e tu vuoi un numero. Le conversioni si fanno a mano, con funzioni dal nome trasparente:

```
variabile testo_numero = "42"
variabile n = intero(testo_numero)
scrivi(n + 1)
scrivi(decimale("3.5"))
scrivi(testo(3.5))
```

```
43
3.5
3.5
```

`intero(...)` trasforma un testo in un numero intero; `decimale(...)` in un decimale; `testo(...)` fa il contrario, trasforma qualunque valore nel suo testo. Se la conversione non ha senso, come `intero("ciao")`, ottieni un `ErroreValore` chiaro, non un risultato a caso.

Prova tu

1. Chiedi all'utente due numeri (ricorda: `chiedi` dà un testo, trasformalo con `intero`) e stampa somma, differenza, prodotto e quoziente.
2. Scrivi un piccolo «conto del ristorante»: parti da `soldi("0.00")`, aggiungi tre o quattro voci con `.piu(...)`, poi stampa il totale e lo stesso totale in lettere con `euro_in_lettere`.
3. Verifica con le frazioni che `1/2 + 1/3 + 1/6` fa esattamente `1`. Poi prova lo stesso conto con i decimali (`0.5 + 0.333...`) e osserva la differenza.

Capitolo 4 — I testi

Un **testo** (gli informatici lo chiamano *stringa*) è una sequenza di caratteri: una parola, una frase, un paragrafo intero. In Leo i testi vanno tra virgolette doppie, e li userai di continuo. Perché quasi ogni programma, prima o poi, deve dire o leggere qualcosa.

> 💡 **L'idea — Una collana di perline**
>
> Immagina un testo come una collana di perline: ogni perlina è un carattere, infilato in un ordine preciso. La parola «Leo» è tre perline — `L`, `e`, `o` — in quella sequenza e non in un'altra. Da questa immagine seguono due cose naturali: puoi contare le perline (la lunghezza del testo) e puoi sfilarne un tratto (una «fetta» di caratteri), proprio come faremo più avanti.

L'interpolazione: testi che si riempiono da soli

L'abbiamo già incrociata, ma ora le diamo il posto che merita. Dentro un testo tra virgolette, tutto quello che metti tra **graffe** `{…}` viene calcolato, e il risultato prende quel posto. Si chiama **interpolazione**:

```
variabile nome = "Beatrice"
scrivi "Ciao, {nome}! Il tuo nome ha {lunghezza(nome)} lettere."
scrivi "Due più tre fa {2 + 3}."
```

```
Ciao, Beatrice! Il tuo nome ha 8 lettere.
Due più tre fa 5.
```

Tra le graffe ci infili qualunque espressione: una variabile (`{nome}`), una chiamata di funzione (`{lunghezza(nome)}`), un calcolo (`{2 + 3}`), persino un altro testo. È il modo più leggibile per costruire frasi a partire dai dati, e lo sceglieremo quasi sempre al posto della concatenazione con il `+`.

A proposito: i testi si possono anche **unire** con il `+` (`"ab" + "cd"` dà `"abcd"`) e **ripetere** con il `*` (`"ab" * 3` dà `"ababab"`). Ma occhio alla regola d'oro di Leo: i tipi non si mescolano di nascosto. `"Ne mancano " + 3` è un errore, perché stai sommando un testo e un numero. O usi l'interpolazione, `"Ne mancano {3}"`, oppure converti a mano: `"Ne mancano " + testo(3)`.

I caratteri speciali

Certi caratteri non li puoi scrivere direttamente dentro un testo. O perché servirebbero a chiudere il testo stesso (le virgolette), o perché non sono stampabili (l'a-capo). Per questi c'è la barra rovesciata `\`, seguita da una lettera che dice cosa vuoi:

```
scrivi "Prima riga\nSeconda riga"
scrivi "Colonna1\tColonna2"
scrivi "Lei disse \"ciao\" e sorrise"
scrivi "Un backslash: \\"
scrivi "Graffe vere: \{ciao\}"
```

```
Prima riga
Seconda riga
Colonna1    Colonna2
Lei disse "ciao" e sorrise
Un backslash: \
Graffe vere: {ciao}
```

In breve: `\n` è l'a-capo, `\t` è la tabulazione, `\"` è una virgoletta che non chiude il testo, `\\` è una barra rovesciata vera, e `\{` e `\}` sono graffe letterali, utili quando vuoi davvero scrivere una graffa senza interpolare.

Testi su più righe

Quando un testo è lungo, una lettera, un messaggio, un piccolo documento, infilare `\n` dappertutto diventa una pena. Per questo ci sono i testi tra **tre virgolette** `"""`, che vanno a capo liberamente e capiscono comunque l'interpolazione. C'è anche un'eleganza nascosta: se le virgolette di chiusura stanno da sole sulla loro riga, il loro rientro viene tolto da tutte le righe, così il testo può rientrare insieme al codice senza portarsi dietro gli spazi.

```
variabile nome = "Beatrice"
variabile lettera = """
    Caro {nome},
      grazie della visita.
    """
scrivi(lettera)
```

```
Caro Beatrice,
  grazie della visita.
```

Guarda: il rientro di quattro spazi, quello del codice, è sparito, mentre i due spazi *in più* davanti a «grazie» sono rimasti. Leo toglie solo il rientro comune, quello segnato dalle virgolette di chiusura.

Entrare dentro un testo

Un testo è fatto di caratteri in fila, numerati a partire da **zero**. Con le parentesi quadre prendi il carattere a una certa posizione; un indice negativo conta dalla fine (`-1` è l'ultimo):

```
variabile nome = "Beatrice"
scrivi(nome[0])
scrivi(nome[-1])
```

```
B
e
```

> **Curiosità — Ogni emoji ha un numero**
>
> Dietro ogni carattere che vedi c'è un numero, stabilito da uno standard internazionale chiamato Unicode. Non solo le lettere dell'alfabeto: anche gli ideogrammi cinesi, l'alfabeto greco, i simboli matematici e persino le emoji hanno ciascuno il proprio codice. I caratteri definiti da Unicode sono oltre centomila, e crescono a ogni nuova versione. Quando Leo conta le lettere di una parola, in fondo conta questi caratteri, qualunque sia la lingua.

Una cosa importante, che all'inizio spiazza: in Leo **i testi non si modificano**. Non puoi cambiare una lettera «sul posto». Tutti i metodi che sembrano cambiare un testo in realtà ne restituiscono uno *nuovo*, e lasciano intatto l'originale. È una scelta che rende i programmi più prevedibili.

I metodi dei testi

Un **metodo** è una funzione che appartiene a un valore e si chiama con il punto. I testi ne hanno parecchi; ecco i più usati:

```
variabile nome = "Beatrice"
scrivi(nome.maiuscolo())
scrivi(nome.minuscolo())
scrivi("  spazi  ".pulisci())
scrivi("ciao mondo".dividi())
scrivi("-".unisci(["2026", "09", "25"]))
scrivi("banana".sostituisci("a", "o"))
scrivi("Beatrice".contiene("tri"))
```

```
BEATRICE
beatrice
spazi
["ciao", "mondo"]
2026-09-25
bonono
vero
```

Leggiamoli con calma. `.maiuscolo()` e `.minuscolo()` cambiano il caso delle lettere. `.pulisci()` toglie gli spazi all'inizio e alla fine, utilissimo con quello che scrive un utente. `.dividi()` spezza un testo in una **lista** di pezzi: senza argomenti divide sugli spazi, ma puoi anche indicargli un separatore. `.unisci(lista)` fa il contrario: prende una lista di testi e li incolla mettendo in mezzo il testo su cui lo chiami (qui `"-"`). `.sostituisci(vecchio, nuovo)` cambia tutte le occorrenze. `.contiene(pezzo)` dice se un pezzo è dentro.

Ci sono anche `.inizia_con(...)`, `.finisce_con(...)` e altri; li trovi nel riferimento in appendice. Ricorda: nessuno di questi tocca il testo di partenza, tutti ne restituiscono uno nuovo.

Prova tu

1. Chiedi all'utente nome e cognome in due domande separate, poi stampa le iniziali (per esempio `B.R.`) usando l'indicizzazione `[0]` e `.maiuscolo()`.
2. Prendi una frase a tua scelta, dividila in parole con `.dividi()` e stampa quante parole contiene (suggerimento: `lunghezza` funziona anche sulle liste).
3. Scrivi una «lettera tipo» su più righe con `"""`, lasciando dei buchi come `{nome}` e `{data}`, poi riempila con variabili diverse e osserva come cambia.

Capitolo 5 — Liste e dizionari

Una variabile da sola tiene un valore. Ma i dati, nella vita vera, arrivano in gruppo: i voti di una classe, la spesa della settimana, l'anagrafica di una persona. Per questi casi Leo ha due contenitori fondamentali, che ti seguiranno in ogni programma di una certa stazza: le **liste** e i **dizionari**.

Le liste: cose in fila

Una lista è una sequenza ordinata di valori, scritti tra parentesi quadre e separati da virgole. Gli elementi sono numerati a partire da **zero**; con le quadre prendi quello alla posizione che vuoi, e un indice negativo conta dalla fine:

```
variabile voti = [28, 30, 24]
scrivi(voti[0])
scrivi(voti[-1])
```

```
28
24
```

> 🦁 **Curiosità — Perché si conta da zero**
>
> Non è un capriccio di Leo: in quasi tutti i linguaggi il primo elemento di una lista ha indice 0, non 1. Nel 1982 il grande informatico Edsger Dijkstra scrisse un breve e celebre appunto (siglato EWD831) per spiegare perché contare da zero rende i conti più puliti, soprattutto quando si lavora con gli intervalli. Una volta fattaci l'abitudine, tornare indietro sembra scomodo.

Al contrario dei testi, le liste **si possono modificare**. Puoi cambiare un elemento, aggiungerne, toglierne:

```
variabile voti = [28, 30, 24]
voti[1] = 29
voti.aggiungi(27)
scrivi(voti)
scrivi(lunghezza(voti))
scrivi(27 in voti)
scrivi(somma(voti))
```

```
[28, 29, 24, 27]
4
vero
108
```

Qui vedi quattro operazioni comunissime: `voti[1] = 29` sostituisce il secondo elemento; `voti.aggiungi(27)` ne attacca uno in fondo; `lunghezza(voti)` conta gli elementi; `27 in voti` controlla se un valore è presente (dà `vero` o `falso`); `somma(voti)` addiziona tutti i numeri. Quella parolina `in` vale oro: la ritroverai nei cicli e nelle condizioni.

I metodi delle liste

Oltre ad `aggiungi`, le liste hanno diversi metodi per rimescolare e cercare:

```
variabile numeri = [3, 1, 2]
numeri.ordina()
scrivi(numeri)
numeri.inverti()
scrivi(numeri)
scrivi(numeri.trova(3))
numeri.rimuovi(3)
scrivi(numeri)
scrivi(numeri.estrai())
scrivi(numeri)
```

```
[1, 2, 3]
[3, 2, 1]
0
[2, 1]
1
[2]
```

`.ordina()` mette in ordine (crescente di serie), `.inverti()` rovescia, `.trova(x)` dà la posizione del primo elemento uguale a `x`, `.rimuovi(x)` toglie la prima occorrenza di `x`, `.estrai()` toglie e restituisce l'ultimo elemento (o quello all'indice che indichi). Occhio a una sottigliezza: questi metodi **cambiano la lista sul posto** e non restituiscono la lista modificata. Ecco perché scriviamo `numeri.ordina()` su una riga e `scrivi(numeri)` sull'altra, e non `scrivi(numeri.ordina())` (che stamperebbe `nulla`). È l'opposto dei metodi dei testi, che invece restituiscono sempre un valore nuovo. La ragione è proprio quella: i testi non si modificano, le liste sì.

I dizionari: cose con un'etichetta

In una lista ritrovi un valore sapendo la sua *posizione*. In un dizionario lo ritrovi sapendo la sua *etichetta*, che si chiama **chiave**. Un dizionario è un insieme di coppie «chiave → valore», scritte tra graffe:

> 💡 **L'idea — Come una rubrica**
>
> Un dizionario funziona proprio come un vero dizionario di carta, o come la rubrica del telefono: non scorri tutte le pagine una per una. Vai diritto alla parola che cerchi — la chiave — e leggi la sua definizione, cioè il valore. È questa la differenza con la lista, dove a contare è invece la posizione.

```
variabile persona = {"nome": "Ada", "eta": 36}
scrivi(persona["nome"])
persona["citta"] = "Londra"
scrivi(persona.prendi("email", "-"))
scrivi(persona.ha("eta"))
per ogni chiave in persona fai
    scrivi "{chiave}: {persona[chiave]}"
fine
```

```
Ada
-
vero
nome: Ada
eta: 36
citta: Londra
```

Con le quadre e la chiave leggi un valore (`persona["nome"]`) e, assegnando a una chiave nuova, la aggiungi (`persona["citta"] = "Londra"`). E se chiedi una chiave che non esiste? `persona["email"]` ti darebbe un errore (`ErroreChiave`). Per questo esistono due vie sicure: `.prendi(chiave, predefinito)` restituisce il valore o, se la chiave manca, il valore di riserva che indichi (qui `"-"`); e `.ha(chiave)` dice semplicemente se la chiave c'è. Infine, un ciclo `per ogni chiave in persona` scorre tutte le chiavi: lo vedremo nel dettaglio nel prossimo capitolo.

Le chiavi di un dizionario di solito sono testi, ma possono anche essere interi o valori logici. I valori, invece, possono essere qualunque cosa: numeri, testi, altre liste, altri dizionari. Annidando liste e dizionari rappresenti dati anche molto ricchi. Pensa a una rubrica, dove ogni contatto è un dizionario e tutti i contatti stanno in una lista.

Costruire liste in una riga

Capita spesso di voler costruire una lista calcolando ogni elemento a partire da un'altra sequenza. Leo ha una forma compatta ed elegante per farlo, la

comprensione di lista:

```
variabile quadrati = [n * n per ogni n in 1..5]
scrivi(quadrati)
variabile pari = [n per ogni n in 1..10 se n % 2 == 0]
scrivi(pari)
```

```
[1, 4, 9, 16, 25]
[2, 4, 6, 8, 10]
```

Si legge quasi come una frase: «`n * n`, per ogni `n` da 1 a 5» costruisce la lista dei quadrati; e aggiungendo una condizione `se` tieni solo gli elementi che vuoi, qui i numeri pari da 1 a 10. È una scrittura che condensa in una riga quello che altrimenti vorrebbe un ciclo di tre o quattro. All'inizio può sembrarti densa. Con un po' di pratica diventa una delle cose che renderanno i tuoi programmi più leggibili.

(Hai notato `1..5` e `1..10`? Sono **intervalli**: un modo breve per dire «tutti i numeri da qui a lì». Li incontreremo di nuovo, da protagonisti, nel prossimo capitolo.)

Prova tu

1. Parti da una lista della spesa con quattro voci. Aggiungine una, togline una, ordina la lista in ordine alfabetico e stampa quante voci sono rimaste.
2. Costruisci un dizionario che descrive un libro (titolo, autore, anno), poi stampalo in una frase sola usando l'interpolazione. Aggiungi la chiave `pagine` e ristampalo.
3. Usando una comprensione, costruisci la lista dei multipli di 3 da 1 a 30. Poi modificala perché tenga solo quelli che sono anche multipli di 2.

Capitolo 6 — Intervalli e fette

Chiudiamo la Parte I con un'idea piccola ma che torna di continuo: prendere un *pezzo* di una sequenza. In Leo si fa con gli intervalli, e la notazione è una delle più pulite che il linguaggio ti offra.

Gli intervalli

Un **intervallo** è un modo breve per dire «tutti i numeri da qui a lì». Si scrive con due punti `..` tra gli estremi, che sono **entrambi compresi**:

```
per ogni i in 1..5 fai
    scrivi(i)
fine
```

```
1
2
3
4
5
```

`1..5` sono i numeri 1, 2, 3, 4, 5: cinque numeri, estremi inclusi. È una scelta deliberata di Leo. L'intervallo `1..5` contiene esattamente i numeri che un italiano direbbe «da uno a cinque», senza i trucchi del «fino a cinque escluso» di altri linguaggi. E funzionano anche all'indietro: `10..1` conta da dieci a uno.

Un intervallo non serve solo ai cicli: puoi usarlo ovunque serva una sequenza di numeri, per esempio `somma(1..100)` o `media(1..10)`.

> **Curiosità — Dove finisce una fetta**
>
> Leo include entrambi gli estremi, ma molti linguaggi diffusi fanno il contrario: la fetta parte dall'inizio indicato e si ferma *appena prima* del fondo. La ragione è elegante quanto la scelta di Leo: con gli intervalli «mezzi aperti» una fetta `da 0 a 3` e una `da 3 a 5` combaciano alla perfezione, senza sovrapporsi né lasciare buchi. Due filosofie diverse, ciascuna con la sua logica.

Le fette (slicing)

Lo stesso `..`, messo tra le parentesi quadre, prende una **fetta** di una lista o di un testo. Gli estremi sono, di nuovo, entrambi inclusi e contati da zero; un estremo negativo conta dalla fine:

```
variabile voti = [4, 6, 7, 9, 10]
scrivi(voti[0..2])
scrivi(voti[2..-1])
scrivi(voti[0..-1])
scrivi("Beatrice"[0..2])
scrivi("Beatrice"[2..-1])
```

```
[4, 6, 7]
[7, 9, 10]
[4, 6, 7, 9, 10]
Bea
atrice
```

Leggiamole. `voti[0..2]` prende dall'elemento 0 all'elemento 2 compreso, i primi tre. `voti[2..-1]` va dall'elemento 2 fino all'ultimo (`-1`), cioè tutta la coda. `voti[0..-1]` è una scorciatoia utilissima: «dal primo all'ultimo», cioè **tutta** la sequenza. E poiché la stessa notazione vale per i testi, `"Beatrice"[0..2]` è `"Bea"` e `"Beatrice"[2..-1]` è `"atrice"`.

Una fetta **mantiene il tipo**: una fetta di lista è una lista, una fetta di testo è un testo.

> 💡 **L'idea — Come affettare il pane**
>
> Prendere una fetta di una lista o di un testo è come tagliare una fetta di pane: dici da dove a dove vuoi tagliare e ottieni esattamente quel pezzo, mentre il resto della pagnotta resta lì, intero, pronto per la prossima fetta. L'originale non si rovina: ne esce solo il pezzo che hai chiesto.

I bordi, senza drammi

Che succede se chiedi una fetta più grande della sequenza, o «al contrario»? Leo non va in errore: si comporta nel modo più ragionevole.

```
variabile voti = [4, 6, 7, 9, 10]
scrivi(voti[0..100])
scrivi(voti[3..1])
```

```
[4, 6, 7, 9, 10]
[]
```

Se un estremo sfora, viene accostato al limite: `voti[0..100]` ti dà tutta la lista, senza una lamentela. Se l'inizio viene *dopo* la fine (`3..1`), la fetta è semplicemente vuota, `[]`. Questa tolleranza ti risparmia una montagna di controlli manuali.

Una fetta è una copia

C'è un dettaglio che vale oro quando i programmi crescono: prendere una fetta crea una **lista nuova**. Modificarla non tocca l'originale.

```
variabile voti = [4, 6, 7, 9, 10]
variabile copia = voti[0..-1]
copia.aggiungi(99)
scrivi(voti)
scrivi(copia)
```

```
[4, 6, 7, 9, 10]
[4, 6, 7, 9, 10, 99]
```

`voti[0..-1]` ha prodotto una copia completa: aggiungendo `99` alla copia, l'originale è rimasto intatto. È il modo idiomatico, in Leo, per «fare una copia di una lista» prima di lavorarci sopra senza rischiare di rovinare i dati di partenza.

Per rovesciare una lista, invece, non usare una fetta: c'è il metodo apposito `.inverti()`, che hai visto nel capitolo scorso.

Prova tu

1. Data la lista `[10, 20, 30, 40, 50, 60]`, stampa separatamente la prima metà e la seconda metà usando due fette.
2. Prendi una parola lunga e stampa le sue ultime tre lettere con una fetta che usa indici negativi.
3. Fai una copia di una lista con `[0..-1]`, ordina la copia e stampa entrambe per convincerti che l'originale non è cambiato.

Con questo si chiude la Parte I. Ora sai dichiarare dati, manipolarli e mostrarli. Nella Parte II daremo ai programmi la capacità di **decidere** e di **ripetere**: è lì che il codice comincia davvero a prendere vita.

Capitolo 7 — Prendere decisioni

Finora i nostri programmi eseguivano le istruzioni una dopo l'altra: tutte, sempre, nello stesso ordine. Ordinati e prevedibili. Ma un programma utile deve *scegliere*: se il voto è alto, promuovi; altrimenti, rimanda. Questo è il **controllo di flusso**, e il suo primo strumento è la condizione.

`se`, `altrimenti se`, `altrimenti`

La condizione più semplice è `se`: esegue un blocco di codice solo se qualcosa è vero. Spesso si accompagna ad `altrimenti`, che copre il caso contrario, e ad `altrimenti se`, per i casi intermedi:

```
variabile voto = 16
se voto >= 18 allora
    scrivi "Promosso"
altrimenti se voto >= 15 allora
    scrivi "Orale di recupero"
altrimenti
    scrivi "Respinto"
fine
```

```
Orale di recupero
```

> 💡 **L'idea — Il buttafuori all'ingresso**
>
> Una condizione è come il buttafuori davanti a un locale: controlla un requisito — hai l'età giusta? sei sulla lista? — e in base alla risposta ti manda di qua o di là. Il `se` è il primo controllo, gli `altrimenti se` sono i controlli successivi, e `altrimenti` è la porta che resta aperta per tutti gli altri.

La struttura si legge da sola. Leo controlla la prima condizione (`voto >= 18`): falsa, perché 16 non arriva a 18. Passa alla seconda (`voto >= 15`): vera. Allora esegue il suo blocco e **salta tutto il resto**. Se nessuna condizione reggesse, scatterebbe `altrimenti`. Guarda la forma: ogni condizione apre il blocco con `allora`, e un **solo** `fine` chiude l'intera catena. Puoi mettere quanti `altrimenti se` vuoi, e sia `altrimenti se` sia `altrimenti` sono facoltativi.

> 🦁 **Curiosità — Una frase quasi italiana**
>
> Il costrutto «se… altrimenti» è vecchio quanto la programmazione stessa: c'era già nei primissimi linguaggi degli anni Cinquanta, e da allora non se n'è più andato. Quello che cambia, da un linguaggio all'altro, è soltanto la forma. Leo ha scelto parole nostre, così che `se voto >= 18 allora` si legga quasi come una frase in italiano, al posto del classico `if ... then` inglese.

Due comodità. Se `allora` è l'ultima cosa sulla riga, puoi ometterlo: la condizione va a capo da sola. E dentro la condizione di un `se` puoi scrivere `=` al posto di `==` per il confronto, perché lì non c'è ambiguità:

```
variabile risposta = "si"
se risposta = "si"
    scrivi "Evviva!"
fine
```

```
Evviva!
```

(Fuori dalle condizioni, ricordiamolo, `=` assegna un valore e `==` confronta: due cose diverse. Dentro `se` e `mentre`, Leo accetta `=` come se fosse `==` per gentilezza, ma se vuoi puoi sempre scrivere `==`.)

Confrontare e combinare

Le condizioni nascono da **confronti**. Gli operatori sono quelli che ti aspetti: `==` (uguale), `!=` (diverso), `<`, `<=`, `>`, `>=`. A questi si aggiungono `in` (per cercare un elemento in una lista o un pezzo in un testo) ed `è` / `non è` (per confrontare con `nulla`).

Più confronti si combinano con gli operatori logici, scritti a parole: `e` (entrambi veri), `o` (almeno uno vero), `non` (nega). E Leo ha un paio di tocchi che rendono le condizioni quasi italiane:

```
variabile eta = 20
se eta >= 18 e eta < 65 allora
    scrivi "Adulto in età lavorativa"
fine
variabile x = 5
se 0 < x < 10 allora
    scrivi "x è tra 0 e 10"
fine
se non (x in [1, 2, 3]) allora
    scrivi "x non è tra i primi tre"
fine
```

```
Adulto in età lavorativa
x è tra 0 e 10
x non è tra i primi tre
```

La riga `0 < x < 10` è esattamente ciò che scriveresti in matematica: Leo concatena i confronti e li interpreta come «x maggiore di 0 **e** minore di 10». Attenzione a `non`: vale solo per il valore subito dopo, quindi per negare un confronto intero servono le parentesi, come in `non (x in [1, 2, 3])`.

Un dettaglio che conta nei programmi seri: `e` e `o` si fermano appena sanno come va a finire. In `a e b`, se `a` è falso, `b` non viene nemmeno calcolato: tanto il risultato è già falso. Questo si chiama *valutazione pigra* e ti lascia

scrivere condizioni come `se lista ha elementi e lista[0] > 0` senza rischiare errori sul secondo pezzo.

Infine, una regola di Leo che fa da guardrail: in `se` e `mentre` sono ammessi **solo** `vero` e `falso`, non «qualcosa che assomiglia al vero». Per sapere se una lista è vuota non scrivi `se lista` ma `se non vuota(lista)`. Una riga in più. In cambio, toglie di mezzo una categoria intera di malintesi.

Tanti casi: `scegli`

Quando uno stesso valore va confrontato con molte possibilità, una catena di `altrimenti se` diventa pesante. Per questi casi c'è `scegli`, più ordinato e più leggibile:

```
variabile voto = 7
scegli voto
    caso 10 allora
        scrivi "eccellente"
    caso 8, 9 allora
        scrivi "molto bene"
    caso 6..7 allora
        scrivi "sufficiente"
    altrimenti
        scrivi "da ripassare"
fine scegli
```

```
sufficiente
```

Il valore dopo `scegli` si calcola una volta sola, poi si confronta con ogni `caso`: vince il primo che corrisponde. Ogni `caso` può essere un valore singolo, più valori separati da virgole (`8, 9`), un intervallo (`6..7`, estremi compresi) o una lista di alternative. L'`altrimenti` finale, facoltativo, cattura tutto ciò che non ha trovato posto. Puoi chiudere con `fine` o, se preferisci la chiarezza, con `fine scegli`: Leo controlla che corrisponda.

`scegli` dice tutto della filosofia del linguaggio: la stessa logica la scriveresti anche con tanti `se`, ma una forma dedicata, quando il caso lo chiede, rende l'intenzione evidente a colpo d'occhio.

Prova tu

1. Scrivi un programma che chiede l'ora (un numero da 0 a 23) e saluta con «Buongiorno», «Buon pomeriggio», «Buonasera» o «Buonanotte» secondo la fascia oraria.
2. Chiedi un voto da 1 a 10 e usa `scegli` per stampare il giudizio corrispondente (inventa tu i giudizi per i voti bassi).
3. Scrivi una condizione che stampa «Numero valido» solo se un numero è compreso tra 1 e 90 **e** non è uno dei numeri `[13, 17]`. Prova diversi valori.

Capitolo 8 — Ripetere

L'altra metà del controllo di flusso è la **ripetizione**. I computer sono instancabili: fanno la stessa cosa un milione di volte senza mai annoiarsi. Il nostro compito è dirgli *cosa* ripetere e *quando fermarsi*. Leo offre tre modi di ciclare, ciascuno con il suo carattere.

> 💡 **L'idea — Giri di pista**
>
> Un ciclo è come correre dei giri attorno a una pista: ripeti sempre lo stesso percorso e, a ogni giro, controlli se è il momento di fermarti — hai finito i giri previsti, oppure è successo qualcosa che ti dice basta. Senza quel controllo continueresti a girare all'infinito.

`mentre`: finché una condizione regge

`mentre` ripete un blocco *finché* una condizione resta vera. Controlla la condizione prima di ogni giro; appena diventa falsa, esce.

```
variabile n = 3
mentre n > 0 fai
    scrivi "Manca {n}"
    n -= 1
fine
scrivi "Via!"
```

```
Manca 3
Manca 2
Manca 1
Via!
```

Il blocco si apre con `fai` (che, come `allora`, puoi omettere se è l'ultima cosa sulla riga) e si chiude con `fine`. Una cosa a cui stare attenti: dentro il ciclo deve succedere *qualcosa* che prima o poi renda la condizione falsa, qui `n -= 1`. Se te ne dimentichi, il ciclo gira per sempre. È il classico «ciclo infinito». Capita a tutti: se un programma sembra bloccato, è il primo indiziato.

> 🦁 **Curiosità — Insapona, risciacqua, ripeti**
>
> C'è una classica battuta tra programmatori: le istruzioni sul flacone dello shampoo — «insapona, risciacqua, ripeti» — descrivono un ciclo infinito, perché non dicono mai *quando* smettere. Preso alla lettera, resteresti sotto la doccia per sempre. È esattamente l'errore di un `mentre` senza una via d'uscita: la morale è che ogni ciclo ha bisogno di un modo per fermarsi.

`per ogni`: scorrere una sequenza

Molto spesso non vuoi ripetere «finché», ma «per ciascun elemento» di una sequenza: una lista, un testo, un intervallo, le chiavi di un dizionario. Per questo c'è `per ogni`:

```
per ogni frutto in ["mela", "pera", "fico"] fai
    scrivi "Mi piace la {frutto}"
fine
per ogni i in 1..3 fai
    scrivi(i * i)
fine
```

```
Mi piace la mela
Mi piace la pera
Mi piace la fico
1
4
9
```

A ogni giro, la variabile dopo `per ogni` (qui `frutto`, poi `i`) prende il valore dell'elemento successivo. Con un intervallo `1..3` ottieni il classico ciclo a conteggio. Dopo il nome va *sempre* `in`: se scrivi per sbaglio `per ogni i da 1 a 3`, Leo capisce l'intenzione e ti propone `in 1..3`.

`per ogni` sa anche scorrere **due cose insieme**. Su un dizionario prende chiave e valore; su una lista di coppie, i due elementi di ciascuna:

```
variabile capitali = {"Italia": "Roma", "Francia": "Parigi"}
per ogni paese, capitale in capitali fai
    scrivi "{capitale} è la capitale di {paese}"
fine
per ogni x, y in [[1, 2], [3, 4]] fai
    scrivi(x + y)
fine
```

```
Roma è la capitale di Italia
Parigi è la capitale di Francia
3
7
```

`ripeti`: un numero fisso di volte

Quando vuoi solo ripetere *n* volte e non ti importa di un indice, `ripeti` è il più diretto:

```
ripeti 3 volte
    scrivi "Ciao!"
fine
```

```
Ciao!
Ciao!
Ciao!
```

Nota che `ripeti 3 volte` non vuole né `allora` né `fai`: è già completo così. Esiste anche `ripeti per sempre`, per i cicli che vanno avanti finché non li fermi dall'interno. Utile nei giochi e nei programmi interattivi.

Uscire e saltare: `interrompi` e `continua`

Dentro un ciclo hai due comandi per alterarne il corso. `interrompi` esce subito dal ciclo; `continua` salta al giro successivo senza eseguire il resto del blocco:

```
per ogni i in 1..100 fai
    se i % 2 == 0 allora
        continua
    fine
    se i > 9 allora
        interrompi
    fine
    scrivi(i)
fine
```

```
1
3
5
7
9
```

Seguiamo il ragionamento: per ogni numero da 1 a 100, se è pari (`continua`) saltiamo; se ha superato il 9 (`interrompi`) usciamo del tutto; altrimenti lo stampiamo. Risultato: i numeri dispari da 1 a 9. Entrambi i comandi agiscono sul ciclo più interno, quello che li contiene direttamente.

Ripetere nel tempo: `ogni`

I cicli visti finora girano il più velocemente possibile. A volte, invece, vuoi che qualcosa accada a intervalli regolari *mentre il tempo scorre*: un orologio, un conto alla rovescia, l'aggiornamento di una schermata. Per questo c'è `ogni`:

```
variabile secondi = 0
ogni 1 secondo fai
    secondi += 1
    scrivi "{secondi}..."
    se secondi >= 3 allora
        ferma_temporizzatori()
    fine
fine
```

```
1...
2...
3...
```

`ogni 1 secondo` esegue il blocco una volta al secondo. Il programma resta vivo finché c'è un temporizzatore attivo; qui lo fermiamo noi dopo tre giri con `ferma_temporizzatori()`. Esiste anche `dopo`, gemello di `ogni`, che esegue il blocco una sola volta, più tardi (`dopo 3 secondi fai ... fine`). Questi strumenti diventano centrali nei programmi con le finestre, dove li ritroveremo nella Parte V. Per ora ti basti sapere che esistono e che si misurano in `secondi`, `millisecondi` e `minuti`.

Prova tu

1. Stampa la tavola pitagorica del 7 (da `7 x 1` a `7 x 10`) con un `per ogni` su `1..10`.
2. Chiedi all'utente una parola e stampala lettera per lettera, una per riga, con `per ogni lettera in parola`.
3. Usa un `mentre` per trovare il primo numero il cui quadrato supera 1000. (Suggerimento: parti da 1 e aumenta finché `n * n` non supera 1000, poi stampa `n`.)

Capitolo 9 — Le funzioni

Man mano che un programma cresce, ti ritrovi a ripetere gli stessi gruppi di istruzioni. Le **funzioni** servono a questo: dare un nome a un blocco di codice, così da richiamarlo quando serve invece di riscriverlo ogni volta. Sono il mattone con cui si costruiscono i programmi grandi. Ogni funzione fa *una cosa*, la fa bene, e il resto del programma la usa senza doversi preoccupare di *come*.

> 🦁 **Curiosità — Una parola presa in prestito**
>
> Il termine «funzione» arriva dalla matematica, dove indica una regola che a ogni ingresso associa un'uscita: dai un numero, ne ottieni un altro. In programmazione l'idea è la stessa, con un vantaggio pratico enorme: scrivi una cosa *una volta* e la riusi cento. Questo principio ha persino un nome e una sigla, DRY, da «Don't Repeat Yourself» — non ripeterti — reso celebre dal libro *The Pragmatic Programmer* del 1999.

> ☕ **Pausa — La parola «algoritmo» è il nome di una persona**
>
> La parola **algoritmo** — cioè una sequenza precisa di passi per risolvere un problema, esattamente quello che fa una funzione — viene dal nome del matematico persiano **al-Khwārizmī**, vissuto intorno all'anno 800. I suoi libri portarono in Europa i numeri che usiamo oggi e i procedimenti di calcolo; la versione latina del suo nome, *Algoritmi*, ci ha lasciato la parola. (E dal titolo di un suo libro, *al-jabr*, viene «algebra».)

Definire e chiamare

Una funzione si definisce con `funzione`, seguita dal nome e, tra parentesi, dai **parametri** — i dati che riceve. Dentro, `restituisci` dice quale valore mandare indietro:

```
funzione saluta(nome, saluto = "Ciao")
    restituisci "{saluto}, {nome}!"
fine

scrivi(saluta("Giulia"))
scrivi(saluta("Marco", saluto = "Buongiorno"))
```

```
Ciao, Giulia!
Buongiorno, Marco!
```

> 💡 **L'idea — Come un distributore automatico**
>
> Una funzione è come un distributore automatico: inserisci qualcosa (gli argomenti), premi il pulsante e ne esce un risultato. Non hai bisogno di sapere come funziona dentro — ti fidi di quello che restituisce. È questo il bello delle funzioni: una volta scritte bene, le usi senza più pensare a *come* fanno il loro mestiere.

Guarda il parametro `saluto = "Ciao"`: è un **valore predefinito**. Se chi chiama la funzione non lo fornisce (come nel primo caso), vale `"Ciao"`; se lo fornisce (secondo caso), vince il suo. E nota come, nella seconda chiamata, l'abbiamo passato **per nome**: `saluto = "Buongiorno"`. Passare gli argomenti per nome rende le chiamate leggibili, soprattutto quando i parametri sono tanti.

Una regola da tenere a mente: una funzione va **definita prima di essere chiamata**. Se la metti dopo il punto in cui la usi, spostala sopra. Oppure usa il blocco `inizio` del Capitolo 1, che parte quando tutto il file è già stato letto.

Le funzioni come comandi

Hai notato nel Capitolo 1 che `scrivi` si usa senza parentesi? La stessa cortesia vale per le tue funzioni, a una condizione: che **non abbiano** `restituisci`, cioè che facciano qualcosa invece di calcolare un valore. In quel caso puoi chiamarle come comandi, con gli argomenti separati da virgole e senza parentesi:

```
funzione saluta(nome)
    scrivi "Ciao, {nome}!"
fine

saluta "Ada Lovelace"
```

```
Ciao, Ada Lovelace!
```

Le funzioni che invece **restituiscono** un valore vogliono le parentesi, come in un foglio di calcolo: `saluta("Giulia")`, `radice(16)`, `media_di(voti...)`. La regola è coerente con tutto il resto: le parentesi dicono «calcola e dammi un valore».

Le funzioni sono valori

Ecco un'idea che all'inizio spiazza e poi diventa indispensabile: in Leo una funzione è un **valore** come un numero o un testo. Puoi metterla in una variabile, passarla a un'altra funzione, restituirla. Puoi anche creare una funzione *senza nome* — una funzione anonima — al volo:

```
variabile doppio = funzione (x) restituisci x * 2 fine
scrivi(doppio(21))
variabile voti = [4, 6, 7, 9, 10]
scrivi(mappa(voti, doppio))
scrivi(filtra(voti, funzione (v) restituisci v >= 6 fine))
```

```
42
[8, 12, 14, 18, 20]
[6, 7, 9, 10]
```

`mappa(lista, funzione)` applica una funzione a ogni elemento e restituisce la nuova lista; `filtra(lista, funzione)` tiene solo gli elementi per cui la funzione dà `vero`. A entrambe passiamo una funzione: nel primo caso `doppio` (per nome), nel secondo una funzione anonima scritta sul posto. Questo stile, passare comportamenti e non solo dati, è potentissimo, e lo ritroveremo parlando della libreria.

Le chiusure: funzioni che ricordano

Quando definisci una funzione dentro un'altra, quella interna **ricorda** le variabili del posto in cui è nata, anche dopo che la funzione esterna è terminata. Questo meccanismo si chiama **chiusura**, ed è più facile vederlo che spiegarlo:

```
funzione crea_contatore()
    variabile conteggio = 0
    funzione incrementa()
        conteggio += 1
        restituisci conteggio
    fine
    restituisci incrementa
fine

variabile conta = crea_contatore()
conta()
conta()
scrivi(conta())
```

```
3
```

`crea_contatore` crea una variabile `conteggio` e restituisce la funzione `incrementa`, che la usa. Anche se `crea_contatore` è già finita, la funzione `conta` continua ad avere accesso al *suo* `conteggio`, che cresce a ogni chiamata. Ogni contatore creato così ha un conteggio tutto suo, indipendente dagli altri. Le chiusure sono il modo elegante di «incapsulare» uno stato senza bisogno di variabili globali.

Quanti argomenti vuoi

A volte non sai in anticipo quanti argomenti riceverà una funzione. Tre puntini dopo l'ultimo parametro li raccolgono tutti in una lista; e tre puntini nella *chiamata* fanno il contrario, srotolando una lista in tanti argomenti:

```
funzione media_di(valori...)
    restituisci somma(valori) / lunghezza(valori)
fine

scrivi(media_di(6, 7, 8))
variabile voti = [5, 7, 9]
scrivi(media_di(voti...))
```

```
7.0
7.0
```

La stessa funzione accetta sia argomenti sciolti (`media_di(6, 7, 8)`) sia una lista già pronta srotolata con i puntini (`media_di(voti...)`).

La ricorsione

Una funzione può chiamare **sé stessa**. Sembra un gioco di prestigio, ma è il modo naturale per risolvere i problemi che si definiscono in termini di sé stessi, come il fattoriale, dove `5!` è `5 x 4!`:

```
funzione fattoriale(n)
    se n <= 1 allora
        restituisci 1
    fine
    restituisci n * fattoriale(n - 1)
fine

scrivi(fattoriale(5))
```

```
120
```

La chiave di ogni ricorsione è il **caso base**, qui `se n <= 1`, che ferma la discesa. Senza, la funzione si richiamerebbe all'infinito; Leo ti protegge fermandosi dopo 1000 chiamate annidate con un `ErroreRicorsione`, ma il caso base devi sempre scriverlo tu.

Prova tu

1. Scrivi una funzione `area_rettangolo(base, altezza)` che restituisce l'area, e una `perimetro_rettangolo(base, altezza)` che restituisce il perimetro. Stampale per un paio di rettangoli.
2. Scrivi una funzione `applica_sconto(prezzo, percento = 10)` con uno sconto predefinito del 10%, e provala sia con lo sconto predefinito sia con uno diverso passato per nome.
3. Usando `filtra` e una funzione anonima, estrai da una lista di numeri solo quelli maggiori della loro media. (Suggerimento: calcola prima la `media`, poi filtra.)

Capitolo 10 — Quando qualcosa va storto

Diciamocelo: la prima scritta rossa che il computer ti sputa addosso fa una certa impressione. Sembra di aver rotto qualcosa di irreparabile. Spoiler: no. Gli errori non sono l'eccezione. Sono la normalità, e col tempo diventano quasi degli alleati. Un file sparito, un utente che scrive «trenta» invece di «30», una divisione che inciampa sullo zero: il mondo è imperfetto, e un buon programma lo mette in conto. *Prevede* gli imprevisti e li gestisce con grazia. La bella notizia è che Leo, quando qualcosa va storto, non borbotta in geroglifico: te lo dice in italiano, col dito puntato sul punto esatto.

> 💡 **L'idea — Un allarme antifumo**
>
> Pensa a un'eccezione come all'allarme antifumo di casa: quando sente il pericolo interrompe tutto e lo segnala subito, con un suono che non puoi ignorare. È molto meglio di una casa che brucia in silenzio. Un errore che si fa sentire al momento giusto ti permette di intervenire prima che il guaio si propaghi.

Leggere un errore

Partiamo da un errore *non* gestito, per imparare a leggerlo. Questo programma divide per zero:

```
funzione dividi(a, b)
    restituisci a / b
fine
scrivi(dividi(10, 0))
```

```
ErroreDivisione alla riga 2, colonna 19 di programma.leo

    2 |     restituisci a / b
      |                   ^

Divisione per zero.

Chiamate (dalla più recente):
    programma.leo:2  dentro `dividi(a = 10, b = 0)`
    programma.leo:4  nel programma principale
```

Ogni messaggio di Leo ha la stessa struttura, e vale la pena imparare a leggerla. In alto: il **tipo** di errore (`ErroreDivisione`) e *dove* è successo (riga 2, colonna 19). Poi la riga di codice incriminata, con un accento circonflesso sotto il punto esatto. Poi la **spiegazione** in italiano («Divisione per zero.»). E infine, cosa rara e preziosa, l'elenco delle **chiamate**: da dove è partito tutto. Qui leggiamo che `dividi` è stato chiamato con `a = 10, b = 0` dalla riga 4 del programma principale. Nei programmi grandi questa traccia è una bussola: ti porta dritto all'origine del problema.

> 🦁 **Curiosità — Il primo «bug» era un insetto vero**
>
> Nel 1947, cercando un guasto nel calcolatore Harvard Mark II, i tecnici del gruppo di Grace Hopper trovarono una falena incastrata in un relè: la staccarono, la incollarono nel registro di bordo e annotarono «primo caso reale di bug trovato». Da lì chiamiamo «bug» un errore e «debugging» il lavoro di scovarlo. Per onestà: la parola «bug» per un difetto tecnico girava già ai tempi di Edison; questo è però il caso famoso dell'insetto preso con le mani nel sacco.

I tipi di errore più comuni hanno nomi che si spiegano da soli: `ErroreNome` (una variabile che non esiste), `ErroreTipo` (operazioni tra tipi incompatibili, come `"3" + 4`), `ErroreValore` (il tipo è giusto ma il valore

no, come `intero("ciao")`), `ErroreIndice` (fuori dai limiti di una lista), `ErroreChiave` (chiave assente in un dizionario), `ErroreFile` (file che non si apre).

`prova`, `cattura`, `infine`

Per *gestire* un errore invece di subirlo, si racchiude il codice rischioso in un blocco `prova`, e si dice con `cattura` cosa fare se qualcosa va storto:

```
prova
    variabile n = intero("non un numero")
    scrivi(n)
cattura ErroreValore come e
    scrivi "Serve un numero: {e.messaggio}"
infine
    scrivi "Ho finito di provare."
fine
```

```
Serve un numero: Non posso trasformare il testo "non un numero" in un
intero.
Ho finito di provare.
```

Leggiamo il flusso. Leo esegue il blocco `prova`. La prima riga solleva un `ErroreValore` (`intero` non sa cosa farsene di `"non un numero"`), quindi la riga `scrivi(n)` viene saltata e il controllo passa a `cattura ErroreValore come e`. Lì, la variabile `e` rappresenta l'errore: ha i campi `tipo`, `messaggio`, `riga`, `colonna`, `suggerimento`. Infine — ed è letterale — il blocco `infine` si esegue **sempre**, sia che ci sia stato un errore sia che tutto sia filato liscio: è il posto giusto per le pulizie (chiudere un file, rilasciare una risorsa).

Puoi avere più `cattura`, uno per tipo di errore, e vince il primo adatto. Un `cattura` senza tipo (o `cattura come e` da solo) cattura qualunque errore: usalo come rete finale, dopo quelli specifici.

Sollevare i propri errori: `lancia`

Gli errori non li subisci soltanto: puoi anche **sollevarli** tu, quando il tuo codice incontra una situazione che non deve accadere. Si usa `lancia`:

```
funzione controlla_voto(voto)
    se voto < 0 o voto > 30 allora
        lancia ErroreValore("Il voto deve essere tra 0 e 30")
    fine
    restituisci "Voto accettato: {voto}"
fine

prova
    scrivi(controlla_voto(35))
cattura ErroreValore come e
    scrivi "Rifiutato: {e.messaggio}"
fine
```

```
Rifiutato: Il voto deve essere tra 0 e 30
```

`lancia ErroreValore("...")` interrompe la funzione e propaga l'errore verso chi l'ha chiamata, esattamente come farebbe un errore «naturale». Chi chiama può catturarlo, come qui, o lasciarlo salire. È il modo pulito per dire «questo input non è valido»: meglio un errore chiaro e immediato che un risultato sbagliato portato avanti di nascosto.

Mettere alla prova il codice: `verifica`

C'è un modo meraviglioso per dormire sonni tranquilli: scrivere, accanto alle funzioni, dei piccoli controlli che ne verificano il comportamento. In Leo si fa con i blocchi `verifica "nome" fai ... fine`:

```
funzione doppio(n)
    restituisci n * 2
fine

verifica "doppio di un numero" fai
    aspettati doppio(21) == 42
    aspettati doppio(0) == 0
fine

verifica "un testo non numerico" fai
    aspettati errore ErroreValore fai
        intero("trenta")
    fine
fine
```

Quando esegui normalmente il programma, questi blocchi **non partono**: sono lì in attesa. Per lanciarli c'è un comando apposito, `leo verifica`:

```
leo verifica programma.leo
```

```
programma.leo
  ok  doppio di un numero
  ok  un testo non numerico

2 verifiche: 2 riuscite.
```

Dentro un blocco `verifica`, `aspettati condizione` controlla che qualcosa sia vero; con un confronto (`doppio(21) == 42`), se fallisce, il rapporto mostra il valore atteso e quello ottenuto. E `aspettati errore TipoErrore fai ... fine` controlla che un pezzo di codice *dia* proprio quell'errore: perfetto per collaudare la gestione dei casi limite. Scrivere verifiche mentre scrivi il codice è una di quelle abitudini che distinguono i programmi che reggono nel tempo. Quando un domani modificherai `doppio`, le verifiche ti diranno subito se hai rotto qualcosa.

Esiste anche una `verifica` «al volo», da usare dentro il programma come controllo rapido: `verifica doppio(21) == 42` ferma il programma con

un errore se la condizione è falsa. È utile per sancire una cosa che *deve* essere vera a un certo punto dell'esecuzione.

Prova tu

1. Scrivi un programma che chiede un numero all'utente e lo raddoppia, ma gestisce con `prova` / `cattura` il caso in cui l'utente non scrive un numero valido, stampando un messaggio gentile invece di andare in errore.
2. Scrivi una funzione `radice_sicura(x)` che lancia un `ErroreValore` se `x` è negativo, e altrimenti restituisce `radice(x)`. Collaudala con un blocco `verifica`.
3. Prendi la funzione `fattoriale` del capitolo scorso e aggiungi una verifica che controlli `fattoriale(5) == 120` e che `fattoriale(0) == 1`.

Finisce qui la Parte II. Hai dato ai tuoi programmi la capacità di decidere, ripetere, organizzarsi in funzioni e reagire agli imprevisti. Nella Parte III impareremo a *organizzare* programmi più grandi: oggetti, moduli, file.

Capitolo 11 — Classi e oggetti

Finora abbiamo tenuto i dati da una parte (variabili, liste, dizionari) e il comportamento dall'altra (funzioni). Ma guarda la realtà: le due cose quasi sempre viaggiano insieme. Un conto in banca *ha* un saldo e *sa* versare e prelevare. Una persona *ha* un nome e *sa* salutare. Le **classi** servono proprio a questo: unire dati e comportamento in un unico stampo, da cui poi sforni tanti **oggetti** indipendenti. È il cuore della cosiddetta «programmazione a oggetti», e in Leo prende una forma che si legge quasi da sé.

> 🦁 **Curiosità — Nata in Norvegia, battezzata pensando alle cellule**
>
> Le idee alla base della programmazione a oggetti nascono negli anni '60 col linguaggio Simula, creato a Oslo da Ole-Johan Dahl e Kristen Nygaard. L'espressione «object-oriented» la coniò invece Alan Kay, che raccontò di essersi ispirato alle cellule biologiche: tante unità autonome, ciascuna coi propri dati, che collaborano scambiandosi messaggi. Non è poi così diverso da quello che facciamo noi con gli oggetti.

Definire una classe

Una classe si definisce con `classe`. Dentro, i dati si dichiarano con `ha` e i comportamenti con `azione`. All'interno di un'azione, l'oggetto su cui stai lavorando si chiama `questo`:

```
classe Persona
    ha nome
    ha eta = 0
    azione saluta()
        scrivi "Ciao, sono {questo.nome} e ho {questo.eta} anni"
    fine
    azione compleanno()
        questo.eta = questo.eta + 1
    fine
fine

variabile p = Persona("Ada", 36)
p.saluta()
p.compleanno()
scrivi "Dopo il compleanno: {p.eta}"
variabile q = nuovo Persona("Marco")
q.saluta()
```

```
Ciao, sono Ada e ho 36 anni
Dopo il compleanno: 37
Ciao, sono Marco e ho 0 anni
```

Smontiamolo con calma. `ha nome` dichiara un dato (un *campo*) obbligatorio; `ha eta = 0` dichiara un campo con valore predefinito, che puoi quindi omettere alla creazione. Le `azione` sono funzioni che appartengono all'oggetto: dentro, `questo.nome` e `questo.eta` leggono e cambiano i dati di *quel* particolare oggetto. Solo di quello.

Per creare un oggetto, cioè un'*istanza* della classe, scrivi il nome della classe con i valori tra parentesi: `Persona("Ada", 36)`. Preferisci una forma più discorsiva? Leo accetta anche `nuovo Persona("Marco")`: le due scritture sono identiche, scegli quella che ti suona meglio. Poi chiami i metodi col punto: `p.saluta()`, `p.compleanno()`.

E fai attenzione a una cosa: `p` e `q` sono **indipendenti**. Il compleanno di Ada non sfiora Marco. Ogni oggetto si porta dietro la propria copia dei dati, e questo è esattamente ciò che rende le classi così utili. Uno stampo, infiniti oggetti distinti.

> 💡 **L'idea — Lo stampino dei biscotti**
>
> Una classe è come lo stampino per i biscotti, e gli oggetti sono i biscotti. Lo stampino decide la forma — quali campi e quali azioni ci saranno — ma non è qualcosa che si mangia; ogni biscotto che ne esce è invece concreto, con la sua glassa e le sue decorazioni. Allo stesso modo `Persona` descrive la forma, e Ada e Marco sono due biscotti ben distinti usciti dallo stesso stampino.

Campi, tipi e protezioni

I campi possono avere un **tipo**, che Leo controlla alla creazione:

```
classe Punto
    ha x: intero
    ha y: intero
fine
```

Con questa definizione, `Punto(3, 4)` passa senza problemi, ma `Punto("ciao")` ti blocca con un errore chiaro: manca `y`, e `x` dovrebbe essere un intero. Dichiarare i tipi è facoltativo. Ma nei programmi seri ti fa pescare gli sbagli sul nascere, e fidati, è tempo guadagnato.

Vediamo una classe un po' più in carne: un conto corrente che si difende dagli usi scorretti sollevando un errore (ricordi `lancia` dal Capitolo 10?).

```
classe Conto
    ha intestatario
    ha saldo = 0
    azione versa(importo)
        questo.saldo = questo.saldo + importo
    fine
    azione preleva(importo)
        se importo > questo.saldo allora
            lancia ErroreValore("Fondi insufficienti")
        fine
        questo.saldo = questo.saldo - importo
    fine
    azione descrivi() dà testo
        restituisci "{questo.intestatario}: {questo.saldo} euro"
    fine
fine

variabile c = Conto("Giulia")
c.versa(100)
c.versa(50)
c.preleva(30)
scrivi(c.descrivi())
```

```
Giulia: 120 euro
```

Un'azione può anche restituire un valore con `restituisci`, come una funzione qualunque; la notazione `azione descrivi() dà testo` dichiara — facoltativamente — che il risultato è un testo. Nota cosa fa l'oggetto `Conto`: racchiude i suoi dati e li protegge. L'unico modo di muovere il saldo è attraverso `versa` e `preleva`, e sono loro a imporre le regole (niente scoperti). Questo è il senso dell'incapsulamento: tenere insieme i dati e le regole che li governano, così nessuno mette le mani dove non deve.

L'ereditarietà: costruire sul già fatto

Capita di continuo che classi diverse condividano gran parte del comportamento e cambino solo in qualche dettaglio. Un cane e un gatto sono entrambi animali: hanno un nome, si presentano allo stesso modo, ma fanno

un verso diverso. L'**ereditarietà**, con la parola `estende`, lascia che una classe parta da un'altra e cambi solo ciò che serve:

```
classe Animale
    ha nome
    azione verso() dà testo
        restituisci "..."
    fine
    azione presentati()
        scrivi "{questo.nome} fa {questo.verso()}"
    fine
fine

classe Cane estende Animale
    azione verso() dà testo
        restituisci "bau"
    fine
fine

classe Gatto estende Animale
    azione verso() dà testo
        restituisci "miao"
    fine
fine

per ogni a in [Cane("Fido"), Gatto("Romeo"), Animale("Misterioso")] fai
    a.presentati()
fine
```

```
Fido fa bau
Romeo fa miao
Misterioso fa ...
```

`Cane` e `Gatto` **estendono** `Animale`: ereditano il campo `nome` e il metodo `presentati`, ma ridefiniscono `verso` a modo loro. La magia sta tutta nella riga `presentati`, scritta una volta sola dentro `Animale`: quando chiama `questo.verso()`, Leo sceglie la versione giusta per *quell'oggetto* — `bau` per il cane, `miao` per il gatto, `...` per l'animale generico. Questo meccanismo, per cui lo stesso metodo si comporta in modo diverso a seconda del tipo reale dell'oggetto, si chiama **polimorfismo**, ed è

ciò che rende l'ereditarietà davvero potente. Vuoi aggiungere una `Mucca` che fa `muu`? Non devi toccare `presentati` nemmeno con un dito.

Richiamare il genitore

A volte ridefinisci un metodo ma vuoi comunque riusare la versione della classe da cui erediti. Per questo c'è la parola `genitore`:

```
classe Veicolo
    ha nome
    azione descrivi() dà testo
        restituisci "Veicolo {questo.nome}"
    fine
fine

classe Auto estende Veicolo
    azione descrivi() dà testo
        restituisci genitore.descrivi() + " (un'automobile)"
    fine
fine

scrivi(Auto("Panda").descrivi())
```

```
Veicolo Panda (un'automobile)
```

`genitore.descrivi()` chiama la versione di `Veicolo`, e `Auto` ci costruisce sopra aggiungendo la sua parte. È il modo pulito per *estendere* un comportamento invece di riscriverlo da zero.

Quando usare le classi

Una parola di equilibrio, perché le classi sono utili ma non obbligatorie. Se un dato è semplice, un dizionario basta e avanza. Le classi guadagnano terreno quando i dati e le regole che li governano cominciano a intrecciarsi: quando ti serve garantire invarianti (il saldo non va sotto zero), gestire tanti oggetti simili ma distinti, o reggere una gerarchia di tipi che condividono

comportamento. Non sentirti in dovere di incapsulare tutto in una classe dal primo giorno. Introducile quando è il programma a chiedertele.

Prova tu

1. Scrivi una classe `Rettangolo` con i campi `base` e `altezza` e le azioni `area()` e `perimetro()`. Creane un paio e stampane le misure.
2. Aggiungi alla classe `Conto` un metodo `trasferisci(altro, importo)` che preleva da `questo` e versa su un altro conto. Collauda un trasferimento tra due conti.
3. Crea una classe `Quadrato` che **estende** `Rettangolo` e, nel costruttore o in un metodo, imposta base e altezza uguali. Verifica che erediti `area` e `perimetro`.

Capitolo 12 — Dividere il programma in moduli

Un file solo va benissimo per i programmi piccoli. Ma quando un progetto cresce, tenere tutto in un file diventa come vivere in una stanza sola: ogni cosa si trova sempre peggio. La via d'uscita è spezzare il codice in più file, ciascuno col suo tema — le forme geometriche di qua, la gestione dei dati di là — e poi comporli. In Leo questo lavoro lo fa `importa`.

> 💡 **L'idea — I cassetti della cucina**
>
> Dividere il codice in moduli è come organizzare una cucina in cassetti: le posate in uno, le spezie in un altro, gli strofinacci in un terzo. Non è che tu abbia più roba, ma sai sempre dove mettere le mani e non rovisti nel caos a ogni ricetta. Un modulo è esattamente questo: un cassetto con dentro le cose che stanno bene insieme.

Importare un file

Immagina di avere un file `forme.leo` con un po' di funzioni geometriche:

```
# forme.leo
costante PI_GRECO_MIO = 3.14159
funzione area_cerchio(raggio)
    restituisci PI_GRECO_MIO * raggio * raggio
fine
funzione area_rettangolo(base, altezza)
    restituisci base * altezza
fine
```

Dal programma principale lo tiri dentro con `importa`, e da quel momento i suoi nomi sono lì, disponibili come se li avessi scritti tu stesso:

```
# principale.leo
importa "forme.leo"
scrivi "Area del cerchio: {area_cerchio(2)}"
scrivi "Area del rettangolo: {area_rettangolo(3, 4)}"
```

```
Area del cerchio: 12.56636
Area del rettangolo: 12
```

`importa "forme.leo"` esegue l'altro file e mette a disposizione le sue funzioni e costanti. Leo cerca `forme.leo` prima nella cartella del file che importa, poi in quella del programma, poi in quella corrente. Così i progetti restano portabili: li sposti e continuano a funzionare.

Importare «con il punto»

Portarsi tutti i nomi di un modulo dentro il programma è comodo, ma se due moduli hanno una funzione con lo stesso nome nasce confusione. La forma `importa ... come` la taglia alla radice: raccoglie i nomi del modulo sotto un'etichetta, e li usi col punto. Funziona anche con i gruppi della libreria standard:

```
importa forme come f
scrivi "Area del cerchio: {f.area_cerchio(2)}"
importa matematica come m
scrivi "Radice: {m.radice(9)}"
```

```
Area del cerchio: 12.56636
Radice: 3.0
```

Con `f.area_cerchio` si capisce al volo da dove arriva quella funzione. Questo stile diventa prezioso nei progetti grandi, dove sapere la provenienza di un nome è mezza mappa per orientarsi.

Le regole del gioco

`importa` è semplice, ma ha qualche regola che conviene conoscere, perché ti para da errori sottili.

- **Ogni file si esegue una volta sola**, anche se lo importi da più punti. Non rischi di ripetere effetti collaterali o di rallentare il programma reimportando lo stesso codice.
- **I giri circolari sono vietati.** Se `a` importa `b` che a sua volta importa `a`, Leo se ne accorge e si ferma subito con un messaggio che mostra il giro (*«Importazione circolare: a → b → a»*). È un invito a ripensare la divisione: di solito la parte comune va estratta in un terzo file che entrambi importano.
- **Costanti e funzioni importate restano tali.** Non puoi riassegnare una costante presa da un modulo; le funzioni restano funzioni. Le variabili del modulo, invece, si possono cambiare.
- Se un nome del modulo **esiste già** nel tuo programma, l'errore te lo dice e ti suggerisce la forma `importa "..." come nome` per tenerli separati.

> 🦁 **Curiosità — Non ripeterti (DRY)**
>
> Uno dei principi più citati di tutta la programmazione si riassume in tre lettere: DRY, «Don't Repeat Yourself», cioè «non ripeterti». L'espressione fu resa celebre dal libro *The Pragmatic Programmer* (1999) di Andrew Hunt e David Thomas. L'idea è semplice: una cosa va scritta in un posto solo e poi riusata, così quando cambia la correggi una volta e non dieci. I moduli servono proprio a questo.

Oltre: la libreria Python, per chi vuole

Leo include già quasi tutto ciò che serve per l'uso normale, e lo scopriremo nel tour del Capitolo 15. Ma se un giorno ti servisse un pacchetto

dell'immenso ecosistema Python, la porta c'è: `importa python "nome"`. È roba da esperti, e non ti servirà per un bel pezzo; la citiamo solo perché tu sappia che il tetto, sopra la tua testa, è altissimo. Quando quel momento arriverà, la Guida al linguaggio e la scheda `PACCHETTI.md` ti prenderanno per mano.

Prova tu

1. Dividi uno dei programmi dei capitoli scorsi in due file: un modulo con le funzioni e un file principale che le importa e le usa.
2. Crea un modulo `conversioni.leo` con funzioni come `euro_in_dollari(euro)` e `km_in_miglia(km)`, poi usalo da un programma principale con `importa conversioni come c`.
3. Prova di proposito a creare un'importazione circolare tra due file e leggi il messaggio d'errore. Poi sistemala estraendo la parte comune in un terzo file.

Capitolo 13 — File e cartelle

I programmi visti finora soffrono di amnesia totale: appena finiscono, dimenticano tutto. I dati vivono in memoria e svaniscono alla chiusura. Per *ricordare* da un'esecuzione all'altra — salvare una lista di cose da fare, leggere un file di configurazione, sfornare un rapporto — bisogna scrivere su disco. Leo rende l'operazione diretta, con qualche attenzione in più che ti toglie un bel po' di grattacapi.

> **🦁 Curiosità — Un disco da una tonnellata per 5 MB**
>
> Il primo disco rigido della storia, l'IBM 350 del 1956, era alto come un armadio e pesava quasi una tonnellata, eppure conteneva soltanto circa 5 MB: oggi una singola foto scattata col telefono può occupare più spazio di così. Per spostarlo serviva un carrello elevatore; per spostare le tue foto basta un dito. La capacità di «ricordare su disco» è diventata enormemente più leggera in meno di settant'anni.

Una nota prima di partire: i percorsi relativi (`"nota.txt"`) partono dalla cartella da cui lanci il programma. Le cartelle si separano con la barra `/`, anche su Windows.

> **💡 L'idea — Il percorso è un indirizzo**
>
> Il percorso di un file (la sua *path*) funziona come l'indirizzo di casa: provincia, città, via, numero civico restringono il campo un pezzo alla volta, finché non arrivi alla porta giusta. Così `documenti/2026/nota.txt` è «cartella documenti, dentro la cartella 2026, il file nota.txt». Una volta che leggi un percorso come un indirizzo, non ti perdi più nell'albero delle cartelle.

Leggere e scrivere testo

Le due operazioni fondamentali sono scrivere un testo in un file e rileggerlo:

```
scrivi_file("nota.txt", "Prima riga\nSeconda riga")
aggiungi_a_file("nota.txt", "Terza riga")
variabile contenuto = leggi_file("nota.txt")
scrivi "--- contenuto ---"
scrivi(contenuto)
scrivi "--- righe ---"
per ogni riga in leggi_righe("nota.txt") fai
    scrivi "> {riga}"
fine
scrivi "Esiste nota.txt? {esiste("nota.txt")}"
scrivi "Esiste manca.txt? {esiste("manca.txt")}"
```

```
--- contenuto ---
Prima riga
Seconda riga
Terza riga

--- righe ---
> Prima riga
> Seconda riga
> Terza riga
Esiste nota.txt? vero
Esiste manca.txt? falso
```

`scrivi_file(percorso, testo)` crea il file, o ne sostituisce il contenuto se esiste già. `aggiungi_a_file(percorso, testo)` invece attacca in fondo senza cancellare nulla — comodo per un registro che cresce nel tempo — e va a capo da sé. `leggi_file(percorso)` ti restituisce tutto il contenuto come un unico testo; `leggi_righe(percorso)` te lo dà come lista di righe, già ripulite dai caratteri di a-capo, perfette da scorrere con `per ogni`. Infine `esiste(percorso)` controlla se un file (o una cartella) c'è, prima che tu ci metta le mani.

Sotto questa semplicità c'è parecchia cura. `leggi_file` riconosce da solo la codifica del testo (UTF-8, i formati Unicode del Blocco note, la codifica di Excel su Windows) e raddrizza gli a-capo di Windows. Provi a leggere come testo un file che testo non è — un'immagine, un programma? Ottieni un `ErroreFile` in italiano, non una sfilza di caratteri illeggibili. E su Windows i nomi di file vietati vengono respinti con una spiegazione, prima ancora di aprire il file.

JSON e CSV: dati strutturati

Il testo libero va bene per le note. Ma per i dati *strutturati* — liste di dizionari, tabelle — ci sono formati fatti apposta, e Leo li legge e scrive con una riga sola. Il **JSON** è lo standard per scambiare dati tra programmi:

```
variabile persone = [
    {"nome": "Ada", "eta": 36},
    {"nome": "Marco", "eta": 29}
]
scrivi_json("persone.json", persone)
variabile lette = leggi_json("persone.json")
per ogni p in lette fai
    scrivi "{p["nome"]} ha {p["eta"]} anni"
fine
```

```
Ada ha 36 anni
Marco ha 29 anni
```

`scrivi_json` trasforma una struttura Leo (liste, dizionari, numeri, testi) in un file JSON ben formattato, e `leggi_json` fa il viaggio inverso, ridandoti liste e dizionari pronti all'uso. Il bello è che il dato che rileggi è *identico* a quello che hai salvato. Niente conversioni a mano.

Per le tabelle in stile foglio di calcolo c'è il **CSV**, quello che Excel e gli altri programmi d'ufficio capiscono al volo:

```
scrivi_tabella("persone.csv", persone)
scrivi(leggi_file("persone.csv"))
```

```
nome,eta
Ada,36
Marco,29
```

`scrivi_tabella` prende una lista di dizionari e ne fa un CSV, mettendo in cima i nomi delle colonne. C'è anche `leggi_tabella`, che fa il contrario e indovina da solo il separatore (virgola, punto e virgola o tabulazione). Vedremo nel tour della libreria che esiste pure un tipo `tabella` più muscoloso, con filtri, ordinamenti e raggruppamenti.

Lavorare con le cartelle

Oltre ai singoli file, puoi creare cartelle ed elencarne il contenuto:

```
crea_cartella("sotto")
scrivi_file("sotto/a.txt", "a")
scrivi_file("sotto/b.txt", "b")
scrivi(elenca_file("sotto"))
```

```
["a.txt", "b.txt"]
```

`crea_cartella` crea la cartella, e anche quelle intermedie se nel percorso ne mancano. `elenca_file(cartella)` restituisce i nomi dei file in ordine alfabetico, e volendo li filtri per estensione. C'è il gemello `elenca_cartelle` per le sottocartelle. Con questi mattoni scrivi programmi che riordinano una cartella, elaborano tutti i file di un certo tipo o costruiscono un indice.

> 🦁 **Curiosità — Nomi presi in prestito dall'ufficio di carta**
>
> Parole come «file» (in inglese, l'archivio di documenti), «cartella» e «cestino» non sono nate col computer: vengono dritte dall'ufficio di carta. Chi progettò le prime interfacce grafiche copiò di proposito la scrivania, le cartelline e il cestino della spazzatura, per rendere familiare una cosa del tutto nuova. Ancora oggi buttiamo i file nel «cestino»: una metafora di carta sopravvissuta dentro lo schermo.

Un esempio che mette tutto insieme

Pensa a un piccolo diario: a ogni esecuzione aggiunge una riga con la data e un pensiero, poi mostra tutto quello che hai scritto finora. Scrivere, aggiungere, rileggere, scorrere: con `aggiungi_a_file` e `leggi_righe` lo tiri su in pochissime righe. È questo il momento in cui i tuoi programmi smettono di essere esercizi e cominciano a *servire a qualcosa*, perché finalmente ricordano.

Prova tu

1. Scrivi un programma «diario» che chiede un pensiero all'utente, lo aggiunge a un file con la data di oggi (usa `oggi()`), poi stampa tutte

le righe scritte finora.
2. Salva in JSON una piccola rubrica (nome, telefono, email per ogni contatto), poi scrivi un secondo programma che la rilegge e la stampa ordinata per nome.
3. Scrivi un programma che crea una cartella `backup`, vi copia dentro il contenuto di un file di testo esistente, e conferma stampando l'elenco dei file nella cartella.

Con questo si chiude la Parte III. Sai organizzare il codice in oggetti e moduli, e far sì che i tuoi programmi ricordino i dati su disco. Nella Parte IV alziamo il tiro: parallelismo, sequenze pigre e il grande catalogo della libreria.

Capitolo 14 — Fare più cose insieme

I computer di oggi hanno più processori, capaci di lavorare nello stesso momento. Un programma «normale» però ne usa uno solo e fa una cosa alla volta; e quando quella cosa è lunga — un calcolo pesante, lo scaricamento di un file — tutto il resto resta fermo ad aspettare. Il **parallelismo** ti permette di avviare un lavoro e intanto tirare avanti, oppure di spezzare un compito su più processori. Leo lo offre con una sintassi così pulita che quasi non sembra parallelismo.

💡 L'idea — Più casse al supermercato

Il parallelismo è come aprire più casse al supermercato. Con una cassa sola la fila avanza lentamente e tutti aspettano; con quattro casse aperte la stessa fila si smaltisce in circa un quarto del tempo, perché quattro clienti vengono serviti insieme. Dividere il lavoro su più processori è esattamente questo: più «casse» che lavorano nello stesso momento invece di una coda unica.

☕ Pausa — Un nanosecondo lungo trenta centimetri

Grace Hopper, per spiegare quanto è veloce un computer, girava con dei fili lunghi circa **30 centimetri**: tanta è la strada che la luce percorre in un **nanosecondo** (un miliardesimo di secondo). Ne teneva uno in mano e diceva: «Ecco, questo è un nanosecondo». Rende l'idea di perché far lavorare più cose insieme — in parallelo — faccia davvero la differenza.

`in parallelo`: avvia e continua

Il mattone fondamentale è il blocco `in parallelo fai ... fine`. Avvia un lavoro che gira per conto suo e ti restituisce un «segnaposto»; il programma, intanto, prosegue senza aspettare. Quando poi ti serve il risultato, lo chiedi con `aspetta(lavoro)`:

```
funzione lavoro_lungo(n)
    variabile totale = 0
    per ogni i in 1..n fai
        totale += i
    fine
    restituisci totale
fine

variabile lavoro = in parallelo fai
    restituisci lavoro_lungo(1000000)
fine
scrivi "Intanto faccio altro..."
scrivi "Risultato: {aspetta(lavoro)}"
```

```
Intanto faccio altro...
Risultato: 500000500000
```

Guarda l'ordine delle stampe: «Intanto faccio altro…» esce *mentre* il calcolo è ancora in corso, non dopo. `aspetta(lavoro)` mette in pausa il programma solo nell'istante in cui gli serve davvero il risultato, e ti ridà quello che il lavoro ha passato a `restituisci`. E se il lavoro aveva incontrato un errore? `aspetta` te lo ripropone, così lo gestisci con `prova` / `cattura` come sempre.

La regola d'oro: lavori isolati

Qui c'è l'idea che rende il parallelismo di Leo *sicuro*, dove in altri linguaggi è un campo minato. **Ogni lavoro riceve la propria copia** delle variabili del programma, scattata nel momento in cui parte. Le può leggere, ma non può

toccare quelle di fuori. Provi, dentro un lavoro, a scrivere `contatore += 1` su un `contatore` del programma? Leo ti ferma *prima ancora di eseguire*, con un messaggio chiaro.

Perché è una buona notizia? Perché la causa numero uno dei bug nel parallelismo è proprio questa: due lavori che pasticciano con lo stesso dato nello stesso momento, e sfornano risultati imprevedibili, diversi a ogni esecuzione. Isolando i lavori, Leo rende quella categoria di errori impossibile per costruzione. Due lavori non si rovinano mai i dati a vicenda. Punto.

Ma allora, come comunicano i lavori tra loro? Attraverso i **canali**.

I canali: messaggi tra lavori

Un canale è un tubo in cui un lavoro infila messaggi (`invia`) e un altro li pesca (`ricevi`), in ordine. Quando il mittente ha finito, `chiude` il canale. Il destinatario scorre i messaggi con `per ogni`, che si ferma da sé quando il canale è chiuso e vuoto:

```
variabile quadrati = canale()
variabile l = in parallelo fai
    per ogni n in 1..3 fai
        quadrati.invia(n * n)
    fine
    quadrati.chiudi()
fine
per ogni q in quadrati fai
    scrivi(q)
fine
```

```
1
4
9
```

Il lavoro `l` calcola i quadrati e li spara nel canale uno a uno; il programma principale li riceve e li stampa man mano che arrivano. Ogni messaggio è

una copia, coerente con la regola dell'isolamento. I canali sono il modo idiomatico per far collaborare lavori senza memoria condivisa.

Aspettare in tanti modi

Spesso avvii più lavori insieme e poi vuoi i loro risultati. Leo ha diverse forme di attesa, una per ogni situazione comune:

```
funzione quadrato(n)
    restituisci n * n
fine
variabile l1 = in parallelo fai
    restituisci quadrato(10)
fine
variabile l2 = in parallelo fai
    restituisci quadrato(20)
fine
variabile risultati = aspetta_tutti([l1, l2])
scrivi(risultati)
```

```
[100, 400]
```

`aspetta_tutti([...])` raccoglie i risultati di tutti i lavori in una lista e, al primo errore, annulla gli altri. Poi ci sono `aspetta([...])` per una lista di lavori, `aspetta_il_primo([...])` per acchiappare solo il primo che taglia il traguardo, e `aspetta(lavoro, 5)` per darsi un tempo massimo di cinque secondi. Un lavoro lo puoi anche fermare con `annulla(lavoro)`, e chiedergli a che punto è con `lavoro.finito()`.

Dividere il lavoro su più processori

Quando devi applicare lo stesso calcolo pesante a tanti elementi, c'è una scorciatoia che fa tutto da sé, spalmando il lavoro sui processori disponibili: `mappa_in_parallelo`.

```
funzione pesante(n)
    restituisci n * n
fine
variabile risultati = mappa_in_parallelo([1, 2, 3, 4], pesante)
scrivi(risultati)
```

```
[1, 4, 9, 16]
```

Si usa come `mappa` (Capitolo 9), ma dietro le quinte sfrutta più processori sul serio. Per sapere quanti ne hai a disposizione c'è `processori()` (il numero dipende dalla macchina). Occhio però a non aspettarti miracoli con calcoli leggeri: avviare e coordinare i lavori ha un costo, e conviene solo quando ogni singolo calcolo è abbastanza pesante da ripagare quel costo. Per i dettagli e le misure c'è la scheda `PARALLELISMO.md`.

> 🦁 **Curiosità — Nove donne non fanno un bambino in un mese**
>
> Non tutto si può accelerare mettendo più «lavoratori» sul problema. Lo ricorda una battuta celebre dell'ingegneria del software, legata alla *legge di Brooks* del libro *The Mythical Man-Month* (Fred Brooks, 1975): una gravidanza dura nove mesi, e non c'è modo di ridurli a uno assegnando nove donne al compito. Alcuni passi vanno fatti in sequenza, per forza, uno dopo l'altro — e nessun numero di processori li renderà simultanei.

Prova tu

1. Avvia due lavori `in parallelo` che calcolano due somme lunghe diverse, poi raccogline i risultati con `aspetta_tutti` e stampane il totale.

2. Scrivi un lavoro che invia su un canale i numeri primi fino a 50, e un programma principale che li riceve e li conta.
3. Prova `mappa_in_parallelo` su una lista di numeri con una funzione che fa un calcolo abbastanza pesante, e confronta — a occhio — il tempo con la versione `mappa` normale.

Capitolo 15 — Le sequenze pigre

Finora ogni lista che abbiamo costruito esisteva tutta in una volta, in memoria. Tutto lì, pronto, pesante. Ma a volte vuoi ragionare su una sequenza *enorme*, o addirittura **infinita**: tutti i numeri naturali, per dire, senza calcolarla per intero. Le **sequenze pigre** fanno esattamente questo. Producono i loro elementi *uno alla volta, solo quando servono*. «Pigre» non è un difetto. È la loro virtù.

Una sequenza senza fine

`numeri_infiniti(1)` è la sequenza 1, 2, 3, 4, … che non finisce mai. Stamparla tutta è fuori discussione. Ma puoi chiederne i primi elementi con `.prendi(n)`, e trasformarla in una lista vera con `.a_lista()`:

```
variabile primi5 = numeri_infiniti(1).prendi(5).a_lista()
scrivi(primi5)
```

```
[1, 2, 3, 4, 5]
```

La chiave è questa: fino al `.prendi(5)` non è stato calcolato *niente*. La sequenza infinita è solo una *ricetta* per produrre numeri. Chiedi i primi cinque, Leo ne genera cinque e si ferma. Ecco il senso della pigrizia: il lavoro si fa all'ultimo momento utile, e solo quel tanto che serve.

> 💡 **L'idea — Come un rubinetto**
>
> Una sequenza pigra è come un rubinetto: l'acqua esce solo quando apri, e nella quantità che ti serve, non tutta insieme ad allagare la stanza. I valori si calcolano uno alla volta, nel momento in cui li chiedi. È per questo che puoi perfino avere sequenze infinite e prenderne tranquillamente solo i primi pezzi.

Trasformare senza materializzare

Il bello arriva quando concateni le trasformazioni. `.mappa(...)` e `.filtra(...)` funzionano come le loro versioni sulle liste (Capitolo 9), ma restano pigre. Non producono una lista intermedia. Descrivono soltanto *come* trasformare ogni elemento quando verrà il suo turno.

```
variabile quadrati = numeri_infiniti(1).mappa(funzione (x) restituisci x
* x fine).prendi(5).a_lista()
scrivi(quadrati)
variabile pari = numeri_infiniti(1).filtra(funzione (x) restituisci x %
2 == 0 fine).prendi(4).a_lista()
scrivi(pari)
```

```
[1, 4, 9, 16, 25]
[2, 4, 6, 8]
```

Guarda cosa succede nella seconda riga: stiamo filtrando i pari *di una sequenza infinita* e ne prendiamo quattro. Con le liste normali sarebbe impossibile. Non puoi filtrare una lista infinita, non finiresti mai. Con le sequenze pigre funziona, perché Leo genera numeri solo finché non ne ha trovati quattro pari, poi si ferma.

> **Curiosità — Più grande della memoria**
>
> Proprio perché ogni elemento si calcola solo nell'istante in cui serve, una sequenza pigra ti permette di lavorare con elenchi più grandi della memoria stessa del computer. Nessuno chiede mai di tenere tutto in mano nello stesso momento: si legge un valore, lo si usa e lo si lascia andare. È la differenza tra leggere un libro pagina per pagina e pretendere di tenerne a mente tutte le pagine in una volta sola.

Fermarsi al momento giusto

Oltre a `.prendi(n)` ci sono modi più espressivi per decidere dove fermarsi, o da dove partire:

```
variabile piccoli = numeri_infiniti(1).finche(funzione (x) restituisci x
< 6 fine).a_lista()
scrivi(piccoli)
variabile saltati = numeri_infiniti(1).salta(2).prendi(3).a_lista()
scrivi(saltati)
```

```
[1, 2, 3, 4, 5]
[3, 4, 5]
```

`.finche(funzione)` prende gli elementi dall'inizio *finché* la funzione dà `vero`, poi si ferma: qui, tutti i numeri finché sono minori di 6. `.salta(n)` scarta i primi `n` elementi e continua da lì. Combina `salta`, `prendi`, `mappa`, `filtra` e `finche` e costruisci pipeline di trasformazione molto espressive. Nessuna materializza dati inutili.

Rendere pigra una sequenza qualunque

Non serve partire da `numeri_infiniti`. Puoi rendere pigra una lista che hai già, con `pigra(...)`. Ti serve quando vuoi applicare una catena di trasformazioni senza creare liste intermedie a ogni passo:

```
variabile s = pigra([1, 2, 3, 4, 5, 6])
variabile r = s.filtra(funzione (x) restituisci x % 2 == 0
fine).mappa(funzione (x) restituisci x * 10 fine).a_lista()
scrivi(r)
per ogni n in numeri_infiniti(10, 5).prendi(4) fai
    scrivi(n)
fine
```

```
[20, 40, 60]
10
15
20
25
```

Nella prima parte, da `[1..6]` teniamo i pari e li moltiplichiamo per dieci, tutto in un flusso solo. Nella seconda, `numeri_infiniti(10, 5)` parte da 10 con passo 5 (10, 15, 20, 25, …) e un `per ogni` ne consuma i primi quattro. Le sequenze pigre si possono scorrere direttamente con `per ogni`, senza passare da `.a_lista()`.

Quando usarle

Le sequenze pigre non sostituiscono le liste. Le liste restano la scelta giusta per i dati di dimensione nota su cui vuoi tornare più volte. Le sequenze pigre brillano in tre situazioni: quando la sequenza è concettualmente infinita (flussi, generatori); quando è enorme e vuoi solo i primi risultati senza calcolare tutto; e quando concateni molte trasformazioni e vuoi evitare le liste intermedie. In tutti questi casi la pigrizia non è un compromesso. È esattamente ciò che serve.

Prova tu

1. Costruisci la sequenza dei multipli di 3 (`numeri_infiniti` con il passo giusto, oppure `.filtra`) e stampane i primi dieci.
2. Usando `.finche`, prendi tutti i quadrati dei numeri naturali finché restano sotto 200, e stampali.
3. Parti da una lista di parole, rendila pigra, tieni solo quelle lunghe più di quattro lettere e trasformale in maiuscolo; stampa il risultato con `.a_lista()`.

Capitolo 16 — Un tour della libreria

Un linguaggio non è solo la sua grammatica. È anche tutto ciò che porta già in dotazione. Leo include **oltre novecento funzioni** di libreria, sempre disponibili senza installare nulla. Questo capitolo non le elenca tutte: per quello ci sono le schede `LIBRERIA_*.md` in `docs/`. Ti accompagno invece, area per area, a vedere cosa c'è e come si usa, con esempi veri. Pensalo come una passeggiata in una grande bottega: non per comprare tutto, ma per sapere cosa c'è sugli scaffali, così da ricordartene quando ti servirà.

Una sola avvertenza: alcune funzioni (quelle che pescano a caso o misurano il tempo) danno un risultato diverso a ogni esecuzione. Dove capita, te lo segnalo.

> 🦁 **Curiosità — Batterie incluse**
>
> Tra i programmatori si dice che un buon linguaggio arrivi «con le batterie incluse» (*batteries included*): tante funzioni già pronte all'uso, senza doverle cercare o installare altrove. L'espressione è stata resa celebre dal mondo di Python ed è diventata un complimento ambito. Con le sue oltre novecento funzioni di libreria, Leo prende la cosa molto sul serio.

Testo e lingua italiana

Qui Leo dà il meglio di sé, perché è un linguaggio italiano fin nelle ossa. Scrive i numeri in lettere, fa plurali e articoli, divide in sillabe:

```
scrivi(numero_in_lettere(123))
scrivi(euro_in_lettere(12.50))
scrivi(plurale("casa", femminile = vero))
scrivi(con_articolo("zaino"))
scrivi(sillabe("programmare", separatore = "-"))
```

```
centoventitré
dodici euro e cinquanta centesimi
case
lo zaino
pro-gram-ma-re
```

`con_articolo` sa che davanti a «zaino» va «lo», non «il». Piccole cose che, in un programma che parla a persone italiane, fanno la differenza tra un robot e un testo scritto bene. Nella stessa area trovi funzioni per Markdown, XML e YAML, e l'inverso `lettere_in_numero`.

Date e tempo

Il tipo `data` conosce il calendario e si stampa all'italiana. Puoi fare aritmetica tra date, sapere in che giorno della settimana cade una certa data, contare i giorni che mancano:

```
variabile natale = data(2026, 12, 25)
variabile oggi_data = data(2026, 10, 5)
scrivi(natale)
scrivi "Giorni al Natale: {giorni_tra(oggi_data, natale)}"
scrivi(nome_giorno(5))
scrivi "Natale cade di {natale.nome_giorno}"
```

```
25/12/2026
Giorni al Natale: 81
venerdì
Natale cade di venerdì
```

Una `data` ha proprietà come `giorno`, `mese`, `anno`, `nome_giorno`, `fine_settimana`, e metodi come `aggiungi_giorni`. Come i testi, le date non si modificano: aggiungere giorni restituisce una data nuova. Per il tempo che scorre ci sono `oggi()`, `ora()`, `adesso()` e `attendi(secondi)`.

Calcolo e statistica

Oltre alle funzioni matematiche di base, Leo porta la statistica pronta all'uso. Niente librerie da installare: media, mediana, deviazione e un riepilogo completo sono a portata di mano:

```
variabile voti = [18, 25, 30, 28, 22]
scrivi "Media: {media(voti)}"
scrivi "Mediana: {mediana(voti)}"
scrivi "Deviazione: {arrotonda(deviazione(voti), 2)}"
variabile r = riepilogo_numeri(voti)
scrivi "Minimo {r["minimo"]}, massimo {r["massimo"]}"
```

```
Media: 24.6
Mediana: 25
Deviazione: 4.27
Minimo 18, massimo 30
```

`riepilogo_numeri` ti dà in un colpo solo conteggio, media, mediana, quartili, minimo, massimo e deviazione. Perfetto per farsi un'idea veloce di un insieme di dati. Nella scheda scientifica trovi anche unità di misura, calcoli finanziari e qualche strumento di intelligenza artificiale elementare.

Tabelle di dati

Quando i dati sono tanti e strutturati, il tipo `tabella` è una piccola meraviglia. Filtra, ordina, raggruppa, come un foglio di calcolo ma scritto in italiano.

```
variabile t = tabella([
    {"nome": "Ada", "voto": 30},
    {"nome": "Marco", "voto": 24},
    {"nome": "Lia", "voto": 28}
])
variabile bravi = t.filtra("voto", ">", 27).ordina("voto", decrescente =
vero)
scrivi(bravi)
```

```
 nome | voto
------+------
 Ada  |   30
 Lia  |   28
(2 righe, 2 colonne)
```

Una tabella si stampa già allineata e leggibile: si legge da un CSV con `tabella_da_file`, si salva, si disegna in un grafico. Per chi lavora con i dati, è lo strumento che ti evita mille cicli scritti a mano.

Formati: JSON, CSV e dintorni

Abbiamo già usato JSON e CSV nel Capitolo 13. La libreria sa anche produrre JSON ben formattato per l'occhio umano, con il rientro:

```
variabile dati = {"nome": "Ada", "voti": [30, 28]}
scrivi(a_json(dati, 2))
```

```
{
  "nome": "Ada",
  "voti": [
    30,
    28
  ]
}
```

Nella stessa famiglia ci sono il lavoro sui dati binari, la compressione, e la lettura del «tipo vero» di un file (capire se un file è davvero un'immagine, indipendentemente dall'estensione).

Modelli di testo

Per cercare e validare schemi dentro un testo (numeri, email, date) Leo offre i **modelli**, un'alternativa leggibile alle espressioni regolari, con scorciatoie come `<cifre>`, `<email>`, `<data>`:

```
scrivi(trova_tutti("3 gatti e 12 pesci", "<cifre>"))
scrivi(corrisponde("mario@esempio.it", "<email>"))
```

```
["3", "12"]
vero
```

`trova_tutti` estrae tutti i pezzi che seguono il modello; `corrisponde` dice se *tutto* il testo lo rispetta. Perfetto per validare un campo inserito da un utente.

Geometria e giochi

C'è un'intera area dedicata alla geometria vettoriale, utile per i giochi, le animazioni e i disegni: punti, vettori, distanze, angoli.

```
variabile a = crea_punto(0, 0)
variabile b = crea_punto(3, 4)
scrivi "Distanza: {distanza_punti(a, b)}"
scrivi(somma_punti(a, b))
```

```
Distanza: 5.0
[3, 4]
```

(Sì, è il triangolo 3-4-5 della scuola.) Su queste fondamenta, Leo Studio costruisce una ricca scheda «Giochi» con sprite, fisica, collisioni e scene, che incontreremo nella Parte V.

Sicurezza

Leo prende sul serio la sicurezza, con un'area pensata per la difesa (ne faremo un capitolo intero nella Parte V). Qui solo un assaggio: l'impronta crittografica di un dato, la valutazione della robustezza di una password, la generazione di password memorizzabili.

```
scrivi(impronta("password"))
variabile f = misura_forza_password("password")
scrivi "Livello: {f["livello"]}, punteggio {f["punteggio"]}"
```

```
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
Livello: pessima, punteggio 0
```

`impronta` calcola l'impronta SHA-256 (la stessa password dà sempre la stessa impronta, ma dall'impronta non si risale alla password). `misura_forza_password` giudica una password senza pietà: `"password"` è, giustamente, «pessima». E `genera_frase_password(4)` crea una frase-password robusta e facile da ricordare, mettendo in fila parole a caso: il risultato cambia a ogni esecuzione (qualcosa come `zucca-latte-tetto-gradino-27`).

Caso e cronometro

Per i giochi, le simulazioni e le misure ci sono le funzioni del caso e del tempo:

```
variabile estratti = scegli_a_caso(1..90, quanti = 6)
scrivi "Estratti {lunghezza(estratti)} numeri tra 1 e 90"
```

```
Estratti 6 numeri tra 1 e 90
```

`scegli_a_caso` pesca elementi a caso (qui sei numeri diversi del lotto); `mescola` rimescola una lista; `casuale(da, a)` dà un intero a caso. E `cronometro()` misura quanto impiega un pezzo di programma: utile per capire dove conviene ottimizzare.

> 💡 **L'idea — La cassetta degli attrezzi**
>
> La libreria è una cassetta degli attrezzi ben fornita: invece di forgiarti il martello ogni volta che ti serve, apri il cassetto giusto e prendi lo strumento già pronto. Il mestiere non sta nel rifare gli attrezzi, ma nel sapere quale c'è e quando usarlo. Ecco perché conviene farsi un giro tra gli scaffali prima di mettersi a scrivere tutto da capo.

E molto altro

Abbiamo toccato le aree più comuni, ma la bottega è grande. Ci sono funzioni per le **immagini** (apri una foto, applica filtri, crea collage e GIF), per il **web** (scarica pagine, leggi feed, parla con API; le funzioni offline come `html_in_testo` e `proteggi_html` lavorano su HTML che hai già), per la **rete** e il **sistema** su ogni piattaforma, per l'**ufficio** (Excel, Word, PDF, stampa unione), per i **database** senza SQL, per i **media** (suoni, voce). Quando ti troverai davanti un problema, è probabile che un pezzo della soluzione sia già sullo scaffale. La prima mossa, prima di scrivere codice, è dare un'occhiata alle schede `LIBRERIA_*.md`.

Prova tu

1. Chiedi all'utente un importo e stampalo in lettere con `euro_in_lettere`, poi anche la sua impronta SHA-256 con `impronta`.
2. Costruisci una `tabella` con i tuoi film preferiti (titolo, anno, voto), filtra quelli con voto alto e ordinali per anno.
3. Scrivi un «estrattore del lotto» che pesca sei numeri con `scegli_a_caso` e li stampa ordinati. Eseguilo due volte e osserva che cambiano.

Finisce qui la Parte IV. Hai visto come far lavorare Leo in parallelo, come ragionare su sequenze infinite e quanto ti regala già la libreria. Nella Parte V usciamo dal terminale: finestre, telefono, sicurezza e distribuzione.

Capitolo 17 — Finestre e Leo Studio

Tutto quello che abbiamo scritto finora vive nel terminale: testo che entra, testo che esce. È il modo giusto per imparare, ed è anche il modo giusto per molti strumenti veri. Ma prima o poi vorrai una finestra con pulsanti e caselle, un'**interfaccia grafica**. Leo la costruisce con un modello collaudato da decenni: disegni la finestra, e scrivi il codice che reagisce a ciò che fa l'utente.

Dal terminale alle finestrelle

Prima ancora delle finestre vere e proprie, due funzioni fanno il ponte: aprono finestrelle quando la grafica è disponibile, e si comportano come testo quando non lo è (per esempio in un terminale puro).

```
messaggio("Benvenuto nel programma!")
variabile ok = domanda("Vuoi continuare?")
scrivi "Hai risposto: {ok}"
```

Eseguito in un terminale senza grafica, rispondendo «s», stampa:

```
[Leo] Benvenuto nel programma!
[Leo] Vuoi continuare? [s/n] Hai risposto: vero
```

`messaggio(testo)` mostra un avviso con un pulsante OK; `domanda(testo)` pone una domanda sì/no e restituisce `vero` o `falso`. Sono il modo più rapido per dare un tocco interattivo a un programma, senza ancora disegnare nulla.

Leo Studio: disegnare invece di descrivere

Le finestre complete, con pulsanti, caselle, elenchi e grafici, non si scrivono a mano: si **disegnano**. Lo strumento per farlo è **Leo Studio**, l'ambiente visuale di Leo, nello spirito dei vecchi Visual Basic e Delphi. Trascini un pulsante dove lo vuoi, imposti le sue proprietà in un pannello, e Leo Studio registra tutto in un file `.maschera`. Tu scrivi solo la parte interessante: cosa succede *quando* l'utente agisce.

Leo Studio richiede PySide6 (la libreria grafica) e si avvia con il comando `leo studio` o dal menu del sistema operativo. Un programma con finestre è una **cartella**, un *progetto*, con un file `progetto.toml` che lo descrive, un file `.maschera` per ogni finestra (i controlli e il loro aspetto) e un file `.leo` per ogni finestra (il codice degli eventi). La cosa bella: un progetto disegnato sul computer gira identico sul telefono, e viceversa, perché il formato è lo stesso.

> 🦁 **Curiosità — La madre di tutte le demo**
>
> Mouse, finestre e puntatore sullo schermo furono mostrati al mondo da Douglas Engelbart il 9 dicembre 1968, in una presentazione passata alla storia come «la madre di tutte le demo». L'idea di costruire interfacce trascinando i componenti si diffuse poi tra il grande pubblico con strumenti come Visual Basic. Quando trascini un pulsante in Leo Studio, stai ripetendo un gesto che ha più di mezzo secolo.

Gli eventi: reagire con `quando`

Il cuore di un programma a finestre è il gestore di eventi, introdotto dalla parola `quando`. Si legge come una frase: «quando il pulsante riceve un clic, fai…». Ecco il codice reale del progetto di esempio «Saluto», che trovi in `esempi/progetto_saluto/`:

```
quando Finestra1.apertura fai
    Etichetta1.testo = "Benvenuto! Come ti chiami?"
fine

quando PulsanteSaluta.clic fai
    se vuota(CasellaNome.testo) allora
        messaggio("Scrivi prima il tuo nome")
    altrimenti
        Etichetta1.testo = "Ciao, {CasellaNome.testo}!"
    fine
fine
```

Due gestori, due momenti. Il primo scatta all'**apertura** della finestra e imposta il testo di un'etichetta. Il secondo scatta al **clic** del pulsante: legge quello che l'utente ha scritto nella casella (`CasellaNome.testo`), e se è vuoto protesta con un `messaggio`, altrimenti saluta. I controlli sono **oggetti**: leggi e cambi le loro proprietà con il punto, come hai fatto con gli oggetti del Capitolo 11.

> 💡 **L'idea — Il campanello**
>
> Un evento come `quando PulsanteSaluta.clic fai` è come il campanello di casa: il programma se ne sta tranquillo, senza fare nulla, finché qualcuno non suona. Solo allora esegue la risposta che hai preparato. Non devi stare tu a controllare di continuo se c'è qualcuno alla porta: ci pensa il campanello ad avvisarti al momento giusto.

La regola d'oro delle finestre

C'è un principio che all'inizio confonde e poi diventa ovvio. Il codice «fuori» dai gestori (il blocco `inizio`, le variabili) gira **una volta sola**, all'apertura. Per reagire a ciò che l'utente fa *dopo*, serve un `quando`. Questo codice, per esempio, **non** funziona come ci si aspetterebbe:

```
inizio
    se CasellaTesto1.testo == "Rosso" allora
        Finestra1.colore = "rosso"
    fine
fine
```

All'apertura la casella è vuota, la condizione è falsa, e non viene mai più ricontrollata. La versione giusta mette il controllo dentro un gestore che scatta *ogni volta che la casella cambia*:

```
quando CasellaTesto1.cambio fai
    se CasellaTesto1.testo == "Rosso" allora
        Finestra1.colore = "rosso"
    fine
fine
```

Leo è così attento a questo errore che, se nel blocco iniziale leggi una proprietà che l'utente può cambiare, ti avvisa all'avvio suggerendoti l'evento giusto. Tienilo a mente: nelle finestre *il tempo passa*, e il codice che reagisce deve stare dentro un `quando`.

Più finestre nello stesso progetto

Un programma può avere più finestre, e ognuna vede i controlli di tutte. Con `apri_finestra("Nome")` e `chiudi_finestra("Nome")` passi dall'una all'altra. L'esempio `esempi/login_programma/` mostra uno schema ricorrente: una schermata di accesso che, se le credenziali sono giuste, apre la finestra del programma vero e proprio:

```
quando PulsanteAccedi.clic fai
    se CasellaUtente.testo == UTENTE e CasellaSegreta.testo ==
PAROLA_SEGRETA allora
        EtichettaErrore.testo = ""
        apri_finestra("Programma")
    altrimenti
        EtichettaErrore.testo = "Utente o password non corretti.
Riprova."
    fine
fine
```

Un'accortezza importante, spiegata nei commenti dell'esempio: non si chiude la finestra iniziale (quella del login), perché chiudere la finestra iniziale termina il programma. La si lascia aperta «sotto» e si apre la nuova sopra; per tornare indietro si chiude quella nuova e la prima riricompare.

Il grande catalogo dei controlli

Leo Studio mette a disposizione oltre centoquaranta controlli, raccolti in schede tematiche: dalle basi (etichette, pulsanti, caselle) ai contenitori, dai dati (tabelle, grafici, alberi) al disegno, dai media (suono, video, fotocamera) ai controlli «Italia» (codice fiscale, IBAN, targa, calendario delle festività), fino a una scheda «Giochi» con sprite, fisica e scene, e una scheda «Telefono» con bussola, GPS e sensori. Non c'è spazio qui per elencarli; l'elenco completo, con le proprietà e gli eventi di ciascuno, è nella Guida al linguaggio e nella scheda `STUDIO.md`. La cosa da ricordare è una sola: quando ti servirà un certo mattone dell'interfaccia, quasi certamente c'è già.

Prova tu

> Per questi esercizi serve Leo Studio installato (con PySide6).

1. Apri Leo Studio, crea un progetto nuovo e ricostruisci il «Saluto»: una casella di testo, un pulsante e un'etichetta che saluta per nome al clic.
2. Aggiungi al tuo «Saluto» un secondo pulsante «Pulisci» che svuota la casella e l'etichetta.
3. Parti dall'esempio `login_programma`, cambia le credenziali e aggiungi alla finestra del programma un contatore di note salvate che cresce a ogni Salva.

Capitolo 18 — App per Android

Leo non vive solo sul computer. Con l'app **Leo per Android** puoi scrivere ed eseguire programmi direttamente sul telefono, disegnare app con le finestre toccando lo schermo e trasformarle in **app vere** da installare. E non è una versione ridotta. È lo stesso interprete, la stessa libreria, lo stesso linguaggio. Un progetto fatto sul PC gira sul telefono, e viceversa.

Lo stesso linguaggio, un nuovo mondo

Tutto quello che hai imparato finora vale identico sul telefono. Un progetto con finestre ha lo stesso formato di Leo Studio (`progetto.toml`, i file `.maschera`, i file `.leo`), quindi puoi cominciare un'app sul computer e rifinirla sul telefono, o il contrario. La differenza non è nel linguaggio. È in ciò che il telefono *ha in più*. Un computer non ha una bussola, un accelerometro, un GPS, una torcia. Un telefono sì, e Leo ti ci fa parlare con la stessa naturalezza con cui, sul PC, parli a un pulsante.

> 💡 **L'idea — Un computer con i sensi**
>
> Uno smartphone è un piccolo computer tascabile dotato di sensi: la fotocamera sono i suoi occhi, il microfono le orecchie, il GPS il senso dell'orientamento, l'accelerometro l'equilibrio dell'orecchio interno. Programmarlo significa dargli istruzioni su come usare questi sensi. Leo ti lascia parlare a ciascuno di essi con la stessa naturalezza con cui, sul computer, parli a un pulsante.

L'hardware a portata di `quando`

I sensori e i dispositivi del telefono sono controlli come gli altri, con le loro proprietà e i loro eventi. Vibrazione, torcia, batteria, bussola, livella,

posizione, condivisione: ognuno è un mattone pronto. Ecco, per esempio, un programma che calcola quanto sei lontano da un punto noto usando il GPS (è codice reale dalla documentazione dell'app):

```
quando PulsanteDove.clic fai
    Posizione1.avvia()
fine

quando Posizione1.cambio fai
    variabile km = arrotonda(Posizione1.distanza_da(41.8902, 12.4922) /
1000, 1)
    Etichetta1.testo = "Sei a {km} km dal Colosseo"
fine

quando Posizione1.errore fai
    Etichetta1.testo = evento["messaggio"]
fine
```

Riconosci la struttura: al clic avvii il GPS; quando la posizione cambia, calcoli la distanza e aggiorni un'etichetta; se qualcosa va storto (permesso negato, segnale assente), l'evento `errore` ti dà un messaggio da mostrare. È lo stesso schema a eventi del capitolo scorso, applicato all'hardware.

Oppure un piccolo «pappagallo» che ascolta e ripete, usando dettatura, voce e vibrazione:

```
quando Microfono.frase_riconosciuta fai
    Stato.testo = "Hai detto: {evento["testo"]}"
    VoceTelefono.leggi(evento["testo"])
fine

quando VoceTelefono.fine_lettura fai
    Vibrazione1.vibra(60)
fine
```

La prima volta che un programma usa la fotocamera, il microfono o la posizione, il telefono chiede il **permesso**. Se l'utente rifiuta, Leo lo scrive in italiano e il programma prosegue senza andare in crisi. I permessi si

chiedono solo quando servono davvero, e l'app non ha account, pubblicità né tracciamento.

Cosa c'è e cosa no

Quasi tutto funziona uguale, ma qualche controllo è legato al mondo del PC e sul telefono non c'è (ancora): i giochi con la fisica (`Scena`, `Sprite`), l'hardware da banco (`SchedaPico`, `PortaSeriale`), il server web. Quando un programma li incontra sul telefono, al loro posto compare un riquadro «non disponibile» e l'esecuzione continua. Viceversa, molte cose funzionano su entrambi: video, mappe disegnate senza Internet, cruscotti, grafici dal vivo e tutti i sensori del telefono. La tabella completa, controllo per controllo, è nella scheda `CONTROLLI.md`; la scheda `TELEFONO.md` è la guida dedicata all'app.

Dall'idea all'APK

Quando l'app è pronta, diventa un file `.apk`, il formato delle applicazioni Android, che puoi installare sul tuo telefono e passare ad altri. L'app Leo per Android stessa è un unico APK che gira su tutti i telefoni dalla versione 7.0 in su. La costruzione dell'APK si fa sul computer; i passi dettagliati sono nelle schede `TELEFONO.md` e `android/LEGGIMI.md`, e torneremo sul tema della distribuzione nel prossimo… anzi, tra due capitoli, quando parleremo di come consegnare i programmi a chi non ha Leo.

> **Curiosità — Cosa c'è dentro un APK**
>
> Un'app Android non è altro che un file con estensione `.apk`: in pratica un pacchetto compresso, parente stretto di uno zip, che contiene tutto il necessario per installarla — codice, immagini, suoni e istruzioni. Quando tocchi «installa», il telefono apre il pacchetto e monta l'app al posto giusto. È un po' come ricevere un mobile in scatola: tutti i pezzi sono lì dentro, pronti per essere montati.

Il punto da portare a casa è questo: l'italiano, la semplicità e la libreria completa di Leo non restano chiusi in un terminale. Lo stesso programma che hai scritto per imparare può diventare un'app che porti in tasca, che usa la bussola e il GPS, e che regali a un amico con un file solo.

Prova tu

> Questi esercizi richiedono l'app Leo per Android installata sul telefono.

1. Crea un'app con un pulsante che, al clic, fa vibrare il telefono e accende la torcia; un secondo pulsante la spegne.
2. Fai un'app «bussola» che mostra in un'etichetta il punto cardinale verso cui sei rivolto, usando l'evento `cambio` del controllo `Bussola`.
3. Riprendi un progetto a finestre che hai fatto sul PC (il «Saluto», per esempio), copialo sul telefono e verificalo: gira senza modifiche?

Capitolo 19 — Cybersecurity difensiva

Qui arriviamo a una parte che amo. Leo porta in dotazione un'arsenale di strumenti di sicurezza, e il taglio è dichiaratamente **difensivo**: proteggere i tuoi dati, i tuoi account, i tuoi file. Non attacchiamo i sistemi di nessuno. Costruiamo programmi che custodiscono bene ciò che gli affidi, e tocchiamo con mano i concetti che reggono la sicurezza di tutti i giorni. Quattro pilastri, uno dopo l'altro: password robuste, integrità, riservatezza e autenticazione.

> 🦁 **Curiosità — Il verme che svegliò tutti**
>
> Nel novembre del 1988 un piccolo programma sperimentale sfuggì di mano e divenne il primo a diffondersi da solo su vasta scala attraverso Internet, rallentando migliaia di computer collegati alla rete. L'episodio mostrò al mondo quanto contasse la sicurezza e portò, pochi mesi dopo, alla nascita dei primi centri di risposta agli incidenti informatici. Da allora la difesa ha smesso di essere un pensiero secondario.

> ☕ **Pausa — Il primo virus era quasi gentile**
>
> Nel **1971**, molto prima dei malware moderni, un programma sperimentale chiamato **Creeper** cominciò a spostarsi da un computer all'altro sulla rete ARPANET, limitandosi a mostrare il messaggio: «Sono il Creeper: prendimi se ci riesci». Per fermarlo fu scritto **Reaper**, che lo inseguiva e lo cancellava: di fatto, il primo antivirus della storia.

Password: misurarle e generarle

La prima linea di difesa di qualunque account è la password. Leo ne giudica la robustezza senza pietà e senza illusioni:

```
variabile f = misura_forza_password("ciao123")
scrivi "Livello: {f["livello"]}, punteggio {f["punteggio"]} su 4"
scrivi "È tra le più comuni? {f["comune"]}"
```

```
Livello: pessima, punteggio 0 su 4
È tra le più comuni? vero
```

`misura_forza_password` restituisce un dizionario con il livello, un punteggio da 0 a 4, i bit di casualità stimati, il tempo che ci vorrebbe a indovinarla e se è fra le password più usate al mondo. C'è anche `spiega_forza_password`, che dà un verdetto in italiano con i problemi puntati uno per uno. Perfetto da mostrare a un utente che sta scegliendo una password.

E le password robuste Leo le sa anche *creare*, in forme pensate per essere davvero usabili. `genera_frase_password(6)` mette in fila parole italiane a caso (come `zucca-latte-tetto-gradino-...`), facili da ricordare e difficilissime da indovinare; `genera_password_pronunciabile()` ne crea una leggibile ad alta voce; `genera_pin()` evita gli schemi deboli come `1234` o gli anni. Una frase di sei parole generata così raggiunge una robustezza «ottima», centinaia di bit di casualità, restando qualcosa che un essere umano può ricordare.

Integrità: accorgersi delle modifiche

Come fai a sapere che un file non è stato alterato? Calcoli la sua **impronta** crittografica: un codice che cambia completamente anche per la più piccola

modifica, ma dal quale non si può risalire al contenuto. L'abbiamo già visto con `impronta`. Qui lo usiamo per il suo scopo tipico, verificare l'integrità:

```
variabile originale = impronta("contratto versione 1")
variabile modificato = impronta("contratto versione 1 ")
scrivi "Uguali? {originale == modificato}"
```

```
Uguali? falso
```

Un solo spazio in fondo, e l'impronta è completamente diversa. Su questo principio si fonda la verifica dei download, il controllo che un file non sia stato manomesso, e, nell'area forense di Leo, funzioni come `impronta_albero`, che fotografa una cartella intera (impronta, dimensione e data di ogni file) e `confronta_alberi`, che dice esattamente cosa è cambiato tra due fotografie. È lo strumento con cui ti accorgi che qualcosa, in una cartella di sistema, è stato toccato.

Riservatezza: cifrare per un destinatario

Per tenere un messaggio segreto agli occhi di tutti tranne il destinatario, si usa la crittografia a **chiave pubblica**. L'idea è elegante, e diciamocelo, un po' magica: ciascuno ha una coppia di chiavi, una *pubblica* (che può dare a tutti) e una *privata* (che tiene per sé). Chiunque può cifrare un messaggio con la tua chiave pubblica, ma solo tu, con la privata, puoi aprirlo.

```
variabile chiavi = crea_chiavi_riservatezza()
variabile busta = cifra_per("Il codice del caveau è 4821",
chiavi["pubblica"])
variabile aperto = decifra_da(busta, chiavi["privata"])
scrivi "Messaggio aperto: {aperto}"
```

```
Messaggio aperto: Il codice del caveau è 4821
```

`crea_chiavi_riservatezza` genera la coppia; `cifra_per(dati, pubblica)` chiude il messaggio in una «busta» leggibile solo con la privata corrispondente; `decifra_da(busta, privata)` la riapre. Ci sono anche le versioni per i file (`cifra_file_per`, `decifra_file_da`) e per più destinatari insieme (`busta_per_molti`). Attenzione, però: nell'esempio generiamo le chiavi sul momento, ma nel mondo reale la chiave privata va custodita con cura. Se la perdi, perdi l'accesso. Se te la rubano, perdi il segreto.

> 💡 **L'idea — La cassaforte e il sigillo**
>
> Cifrare un messaggio è come chiuderlo in una cassaforte di cui solo il destinatario ha la chiave: chiunque può vedere la cassaforte, ma nessun altro può aprirla. L'impronta crittografica, invece, è come il sigillo di ceralacca su una busta: non nasconde il contenuto, ma rivela subito se qualcuno l'ha aperta o modificata. Riservatezza e integrità rispondono a due domande diverse — «chi può leggere?» e «è stato toccato?» — ed è per questo che si usano strumenti diversi.

Autenticazione: i codici a tempo

I codici OTP, quelli che leggi sull'app di autenticazione e che cambiano ogni trenta secondi, sono il secondo fattore che protegge i tuoi account. Leo li sa calcolare. Qui chiediamo il codice valido a un istante *preciso*, così il risultato è riproducibile:

```
variabile segreto = "JBSWY3DPEHPK3PXP"
scrivi "Codice alle 1700000000: {codice_otp_a(segreto, 1700000000)}"
```

```
Codice alle 1700000000: 324550
```

A partire da un segreto condiviso, `codice_otp_a(segreto, istante)` calcola il codice valido in quel momento. Nell'uso reale si usa `codice_otp(segreto)` per il codice *di adesso* e `verifica_codice_otp(segreto, codice)` per controllarlo. Leo sa anche produrre il codice QR (`qr_otp_svg`, `salva_qr_otp`) con cui configurare l'app di autenticazione. Con questi pezzi puoi costruire, per capire come funziona da dentro, il tuo piccolo sistema a due fattori.

Forense di base: il file dice la verità?

Chiudiamo con uno strumento investigativo, ed è il mio preferito: capire il *vero* tipo di un file, indipendentemente dall'estensione. Un malware si traveste spesso cambiando l'estensione. Leo non ci casca, guarda il contenuto.

```
scrivi_file("finto.png", "Questo non è un PNG, è testo")
variabile esito = estensione_coerente("finto.png")
scrivi "Coerente? {esito["coerente"]}"
scrivi(esito["avviso"])
```

Il programma crea un file chiamato `finto.png` che in realtà contiene testo, e `estensione_coerente` lo smaschera: restituisce `coerente: falso` e un avviso che spiega che il file, malgrado l'estensione `.png`, è in realtà testo. Su questo si costruiscono il triage dei file e le regole di scansione in stile YARA della scheda `LIBRERIA_FORENSE.md`.

Una nota di responsabilità

Questi strumenti sono potenti, e servono a proteggere. Usali per quello: le tue password, i tuoi file, i sistemi che amministri o per cui hai un permesso esplicito. La crittografia e l'analisi sono conoscenze preziose proprio perché chi le capisce sa difendersi meglio. Impararle su te stesso, cifrando i tuoi appunti e verificando le tue cartelle, è il modo giusto per farle tue.

Prova tu

1. Scrivi un «controllore di password» che chiede una password all'utente e stampa il verdetto di `spiega_forza_password`, poi gli propone una `genera_frase_password` come alternativa robusta.
2. Cifra un piccolo file con `cifra_file_per` usando una coppia di chiavi appena creata, poi riaprilo con `decifra_file_da` e verifica che il contenuto coincida.
3. Crea un file `relazione.pdf` che in realtà contiene testo e usa `estensione_coerente` per smascherarlo; stampa l'avviso che Leo restituisce.

Capitolo 20 — Distribuire i programmi

Hai scritto un programma utile. Ora vuoi darlo a qualcuno che non ha Leo installato: un collega, un amico, un'intera classe. Non puoi pretendere che chi lo riceve sappia aprire un terminale, installare Python e lanciare `python3 -m leo`. La distribuzione risolve proprio questo. Impacchetta il programma in qualcosa che si avvia con un doppio clic. Su Windows è un eseguibile `.exe`; su Android, l'APK del capitolo scorso.

> 🦁 **Curiosità — «Funziona sulla mia macchina»**
>
> È diventata proverbiale, tra i programmatori, la frase pronunciata davanti a un programma che gira benissimo sul computer di chi l'ha scritto e da nessun'altra parte: «funziona sulla mia macchina». Di solito la si dice con un sorriso rassegnato, perché il problema è proprio quello. Distribuire bene un programma serve esattamente a questo: far sì che funzioni anche sulle macchine di tutti gli altri.

Un eseguibile Windows con `leo compila`

Il comando `leo compila` trasforma un programma `.leo`, o un intero progetto con finestre, in un **eseguibile Windows autonomo**. Chi lo riceve non installa niente. Né Python, né Leo, né la libreria grafica:

```
leo compila saluto.leo
leo compila Rubrica
```

Il primo compila un singolo programma, il secondo un intero progetto (quello disegnato con Leo Studio). Alla fine Leo ti dice dove si trova

l'eseguibile e ti consegna un rapporto dettagliato su cosa ha incluso e cosa ha lasciato fuori.

Una cosa che mi piace molto, ed è tutta nello spirito di Leo: l'eseguibile contiene **solo ciò che il programma usa davvero**. Se il tuo programma non apre finestre, l'eseguibile non si porta dietro la libreria grafica; se non usa i database, non include il motore dei database. Il risultato è più piccolo e più pulito di quanto ci si aspetterebbe, e il rapporto ti mostra nero su bianco cosa è stato escluso.

Le opzioni che contano

La compilazione si rifinisce con qualche opzione, utile quando consegni un programma «vero» a qualcuno:

```
leo compila Rubrica --nome "Rubrica 2026" --icona rubrica.ico
leo compila Rubrica --autore "Anna Rossi" --descrizione "Rubrica della
classe" --versione-programma 1.2
leo compila Rubrica --un-file
```

`--nome` e `--icona` danno all'eseguibile un nome e un'icona riconoscibili; `--autore`, `--descrizione` e `--versione-programma` riempiono le proprietà del file che Windows mostra. L'opzione `--un-file` produce un *unico* `.exe` invece di una cartella: è più comodo da spedire, un file solo, ma parte un po' più lento, perché deve scompattarsi a ogni avvio. Senza `--un-file`, ottieni una cartella con l'eseguibile e le sue librerie, che parte subito ma va distribuita per intero.

Dietro le quinte, `leo compila` si appoggia a uno strumento chiamato PyInstaller, che va installato una volta sola nell'ambiente di Leo. Se manca, Leo te lo dice e ti mostra il comando esatto per installarlo (o lo fa per te con `--installa`).

Firmare il programma

Quando distribuisci un eseguibile su Windows, il sistema operativo e gli antivirus guardano con sospetto i programmi senza **firma digitale**. Se possiedi un certificato, puoi firmare l'eseguibile direttamente dalla compilazione:

```
leo compila Rubrica --firma certificato.pfx
```

La firma non è obbligatoria per i programmi che giri tu o che passi a persone di fiducia, ma diventa importante se vuoi distribuire più ampiamente: dice a chi riceve che il programma viene davvero da te e non è stato manomesso per strada.

L'APK per Android

Sul fronte Android, la storia è quella del capitolo scorso: un progetto a finestre diventa un file `.apk` da installare sul telefono. La costruzione si fa sul computer, e il risultato è un singolo file che gira su tutti i telefoni dalla versione 7.0 in su. Puoi passarlo come qualunque altro file, via messaggio, via e-mail, su una chiavetta, e chi lo riceve lo installa consentendo l'installazione «da origini sconosciute». I passi precisi sono nelle schede `TELEFONO.md`, `android/LEGGIMI.md` e `ANDROID_PLAY.md` (quest'ultima per chi punta al Play Store).

Pensare a chi riceve

La distribuzione è il momento in cui il tuo programma smette di essere tuo e diventa di chi lo usa. Vale la pena curarlo: un nome chiaro, un'icona, una versione, e, se il programma legge o scrive file, istruzioni semplici su dove mette le sue cose. Un buon programma mal consegnato frustra; un programma onesto ben impacchettato si fa ricordare. Le schede

`DISTRIBUZIONE.md` e `COMPILAZIONE.md` raccolgono tutti i dettagli, dalle impronte SHA-256 alle installazioni silenziose.

> 💡 **L'idea — Il prodotto sullo scaffale**
>
> Distribuire un programma è come confezionare un prodotto per lo scaffale di un negozio: non basta che funzioni in laboratorio. Deve arrivare chiuso, completo e pronto all'uso tra le mani di chi lo installa, senza che debba procurarsi i pezzi mancanti. Un'etichetta chiara, una confezione ordinata e il nome ben visibile fanno la differenza tra un prodotto che si prende con fiducia e uno che si lascia sullo scaffale.

Prova tu

> Gli esercizi di compilazione richiedono Windows con l'ambiente di Leo e PyInstaller.

1. Prendi un programma testuale che hai scritto nei primi capitoli e compilalo in un `.exe` con un nome e una descrizione tua. Provalo con un doppio clic.
2. Compila lo stesso programma due volte, una con `--un-file` e una senza, e confronta le dimensioni e la velocità di avvio.
3. Progetta, anche solo sulla carta, come consegneresti la tua app a una classe di trenta studenti: file unico o cartella? Con quali istruzioni? Con quale icona?

Qui si chiude la Parte V, e con essa il percorso principale del libro. Hai cominciato con un `scrivi "Ciao"` e sei arrivato a programmi con finestre, app per il telefono, strumenti di sicurezza e pacchetti da distribuire. Le appendici che seguono sono il tuo scaffale di riferimento: tienile a portata di mano.

Appendice A — Riferimento rapido della sintassi

Una pagina sola, da tenere sottomano. Tutte le forme qui elencate sono trattate per esteso nei capitoli; qui le ritrovi in forma compatta, pronte al volo.

Commenti e struttura

```
# commento di riga
#[ commento
   su più righe ]#
```

I blocchi si aprono con `allora` (dopo `se`, `altrimenti se`, `caso`) o con `fai` (dopo `mentre`, `per ogni`, `quando`) e si chiudono **sempre** con `fine`. A fine riga `allora` / `fai` si possono omettere. `ripeti N volte` non vuole né l'uno né l'altro.

Dati

```
variabile x = 0          # dichiara (poi: x = 5, x += 1)
costante PI = 3.14       # non si può più cambiare
variabile a, b = 1, 2    # più nomi insieme; scambio: a, b = b, a
```

Tipo	Esempio
intero	`42`, `-7`, `1_000_000`, `2 ^ 100`
decimale	`3.14` (punto, non virgola)
testo	`"ciao"`, `"""su più righe"""`
logico	`vero`, `falso`
nulla	`nulla`

Tipo	Esempio
lista	`[1, 2, 3]`
dizionario	`{"nome": "Ada"}`
soldi / frazione	`soldi("1.50")`, `frazione(1, 3)`

Operatori

Aritmetici: `+ - * /` (sempre decimale) `//` (quoziente intero) `%` (resto) `^` (potenza). Confronto: `== != < <= > >=`, `in`, `è` / `non è` (con `nulla`). Logici: `e`, `o`, `non`. Catena: `valore |> funzione`.

Testi

```
"Ciao {nome}, hai {anni + 1} anni"     # interpolazione
"{prezzo:2}"          # due decimali
"{n:migliaia}"        # punti delle migliaia
"{importo:migliaia 2}"   # entrambi (all'italiana)
testo[0]  testo[-1]  testo[0..2]       # indici e fette
```

Condizioni

```
se condizione allora
    ...
altrimenti se altra allora
    ...
altrimenti
    ...
fine

scegli valore
    caso 10 allora ...
    caso 8, 9 allora ...
    caso 6..7 allora ...
    altrimenti ...
fine
```

Cicli

```
mentre condizione fai ... fine
per ogni x in sequenza fai ... fine
per ogni chiave, valore in dizionario fai ... fine
ripeti 3 volte ... fine
ripeti per sempre ... fine        # con interrompi
ogni 1 secondo fai ... fine       # temporizzato
dopo 3 secondi fai ... fine
```

`interrompi` esce dal ciclo; `continua` salta al giro dopo.

Funzioni

```
funzione nome(a, b = 10)
    restituisci a + b
fine

variabile f = funzione (x) restituisci x * 2 fine   # anonima
funzione tutti(valori...) ... fine                  # quanti argomenti
vuoi
nome(voti...)                                       # srotola una lista
```

Senza `restituisci`, una funzione si chiama anche come comando: `saluta Ada`.

Errori

```
prova
    ...
cattura ErroreValore come e
    scrivi(e.messaggio)
cattura come e
    ...
infine
    ...
fine

lancia ErroreValore("messaggio")

verifica "nome" fai          # test, eseguiti con `leo verifica`
    aspettati doppio(21) == 42
    aspettati errore ErroreValore fai ... fine
fine
```

Classi

```
classe Persona
    ha nome
    ha eta = 0
    azione saluta()
        scrivi "Ciao, sono {questo.nome}"
    fine
fine
variabile p = Persona("Ada", 36)      # oppure: nuovo Persona(...)

classe Cane estende Animale
    azione verso() dà testo
        restituisci genitore.verso() + "!"
    fine
fine
```

Moduli

```
importa "forme.leo"            # porta dentro i nomi
importa forme come f           # usa con il punto: f.area(2)
importa matematica come m      # anche i gruppi della libreria
```

Parallelismo e sequenze pigre

```
variabile l = in parallelo fai restituisci calcolo() fine
scrivi(aspetta(l))
variabile c = canale()   # c.invia(x), c.ricevi(), c.chiudi()
aspetta_tutti([l1, l2])  aspetta_il_primo([l1, l2])  annulla(l)

numeri_infiniti(1).mappa(f).filtra(g).prendi(5).a_lista()
pigra(lista).finche(f).a_lista()
```

Esecuzione

```
leo programma.leo           # esegui
leo verifica programma.leo  # esegui i blocchi `verifica`
leo controlla progetto      # controlli statici (senza eseguire)
leo formatta programma.leo  # riscrive nella forma canonica
leo compila programma.leo   # eseguibile Windows
leo                         # REPL interattivo
```

Appendice B — Errori comuni e come risolverli

Gli errori di Leo sono fatti per essere letti: dicono *dove*, *cosa* e spesso *come rimediare*. Questa appendice raccoglie quelli in cui inciampano tutti, all'inizio, con il messaggio reale e la soluzione a fianco. Imparali. Ti faranno risparmiare ore.

«`print` non è definito» — il riflesso di un altro linguaggio

```
print("ciao")
```

```
ErroreNome alla riga 1, colonna 1 di programma.leo

    1 | print("ciao")
      | ^^^^^

`print` non è definito.
Suggerimento: In Leo si scrive `stampa`.

Riga corretta proposta:
    1 | stampa("ciao")
```

Leo riconosce le parole degli altri linguaggi e ti indica la traduzione. Vale per `print` → `stampa`, `if` → `se`, `for` → `per ogni`, `def` → `funzione`, e molte altre. Un riflesso che ti porti dietro da un altro linguaggio, niente di grave. **Soluzione:** usa la parola italiana (qui `stampa` o `scrivi`).

«manca `fine`» — un blocco non chiuso

```
se 3 > 2 allora
    scrivi "si"
```

```
ErroreSintassi ...
Il blocco `se` aperto alla riga 1 non è mai chiuso: manca `fine`.
Suggerimento: Ogni blocco si chiude con `fine`.
```

Ogni `se`, `mentre`, `per ogni`, `funzione`, `classe`, `prova` va chiuso con `fine`. **Soluzione:** aggiungi il `fine` mancante. Un trucco che uso sempre: rientra il codice di quattro spazi, così i blocchi aperti si vedono a colpo d'occhio.

«non si può unire un testo e un numero» — tipi non mescolati

```
scrivi("3" + 4)
```

```
ErroreTipo ...
Con `+` non si può unire un testo e un numero (`4`).
Suggerimento: Metti il valore dentro il testo tra graffe, `{4}`:
`scrivi("3{4}")`.
```

Leo non converte i tipi di nascosto. **Soluzione:** usa l'interpolazione (`"3{4}"`) oppure converti a mano (`"3" + testo(4)`). È la stessa idea vista nel Capitolo 4.

«una variabile nuova si crea con `variabile`» — il refuso protetto

```
pnuti = 3
```

```
ErroreNome ...
`pnuti` non esiste ancora: una variabile nuova si crea con `variabile`.
```

Assegnare a un nome mai dichiarato è un errore apposta, e ti protegge: così un refuso non crea di nascosto una variabile sbagliata. **Soluzione:** se è una variabile nuova, scrivi `variabile pnuti = 3`; se volevi `punti`, correggi il nome.

«stampa la parola invece del valore» — le parentesi che calcolano

Questo non dà errore, e proprio per questo inganna:

```
variabile totale = 12
scrivi totale        # stampa la parola: totale
scrivi(totale)       # stampa il valore: 12
```

`scrivi` senza parentesi tratta ciò che segue come testo letterale. **Soluzione:** usa le parentesi `scrivi(totale)` o l'interpolazione `scrivi "Il totale è {totale}"`.

«indice fuori dalla lista»

```
variabile l = [1, 2, 3]
scrivi(l[5])
```

```
ErroreIndice ...
L'indice 5 è fuori dalla lista: gli indici vanno da 0 a 2.
```

Gli indici partono da 0, quindi una lista di 3 elementi ha indici 0, 1, 2. **Soluzione:** controlla con `lunghezza(l)` prima di indicizzare, o usa un

indice valido. Ricorda che le *fette* (`l[0..100]`) invece non danno errore: si accostano ai limiti.

«la chiave non è nel dizionario»

```
variabile d = {"a": 1}
scrivi(d["b"])
```

```
ErroreChiave ...
La chiave "b" non è nel dizionario.
Suggerimento: Usa `.prendi(chiave, predefinito)` o controlla con
`.ha(chiave)`.
```

Soluzione: usa `d.prendi("b", valore_di_riserva)` quando la chiave potrebbe mancare, oppure controlla con `d.ha("b")`.

«la condizione deve essere vero o falso»

```
variabile l = [1]
se l allora
    scrivi "x"
fine
```

```
ErroreTipo ...
La condizione di `se` deve essere `vero` o `falso`, non un valore di
tipo `lista`.
Suggerimento: Per controllare se non è vuoto scrivi `se non vuota(...)`.
```

In `se` e `mentre` Leo vuole una condizione vera o falsa, non «qualcosa che assomiglia al vero». Niente scorciatoie ambigue. **Soluzione:** scrivi `se non vuota(l)` per «se la lista non è vuota».

«divisione per zero»

```
scrivi(5 / 0)
```

```
ErroreDivisione ...
Divisione per zero.
```

Soluzione: controlla che il divisore non sia zero prima di dividere, oppure racchiudi l'operazione in un `prova` / `cattura ErroreDivisione`.

Una strategia generale

Quando incontri un errore che non capisci:

1. **Leggi il tipo e la riga.** Leo ti dice esattamente dove guardare.
2. **Leggi la spiegazione e il suggerimento.** Molto spesso contengono già la soluzione, o persino la riga corretta.
3. **Guarda la riga *sopra*.** Alcuni errori (parentesi, `fine`) si manifestano una riga dopo la vera causa.
4. **Semplifica.** Se non capisci, togli pezzi finché l'errore non sparisce: l'ultimo pezzo tolto era il colpevole.
5. **Prova in piccolo.** Apri il REPL (`leo`) e ricrea il problema in una riga.

Tieniti vicini `leo controlla` (controlli statici senza eseguire) e `leo formatta` (riscrive nella forma canonica): sono due alleati preziosi per scovare i problemi prima ancora di lanciare il programma.

Appendice C — Un confronto onesto con Python

Leo gira sull'interprete Python, e con Python condivide molto DNA. Viene naturale chiedersi: cosa cambia davvero? Questa appendice risponde senza marketing, nello stesso spirito della scheda `PARITA_PYTHON.md` del progetto, che classifica ogni confronto come *migliore*, *pari*, *indietro* o *mancante*. Se conosci già Python, qui trovi la mappa. Se non lo conosci, questa pagina ti dice dove Leo è più di una semplice traduzione.

La differenza più evidente: la lingua

La cosa che salta all'occhio è che Leo è in italiano: parole chiave, errori, libreria, documentazione. Non è una patina. Cambia l'esperienza di chi impara e di chi rilegge. `if/elif/else` diventano `se/altrimenti se/altrimenti`; `for x in` diventa `per ogni x in`; `True/False/None` diventano `vero/falso/nulla`. Un messaggio d'errore in italiano, con la riga corretta già proposta, è un'altra cosa rispetto a un `Traceback` inglese. Qui Leo è, per il suo pubblico, semplicemente **migliore**.

Dove Leo fa di più (migliore)

Alcune scelte di Leo risolvono di serie problemi che in Python richiedono attenzione o import:

- **Denaro esatto.** In Python servono `from decimal import Decimal` e qualche cautela; in Leo `soldi("0.10").piu(soldi("0.20"))` dà `0,30 €` esatto, già formattato all'italiana, senza import.
- **Frazioni di serie.** `frazione(1, 3)` è nel linguaggio; in Python serve `from fractions import Fraction`.

- **Costanti vere.** `costante IVA = 22` non si può riassegnare, e se ci provi è un errore chiaro; in Python la costante è solo una convenzione (`IVA = 22` maiuscolo) che nulla impedisce di cambiare.
- **Niente conversioni silenziose.** `"3" + 4` in Leo è un errore spiegato; in Python è un `TypeError` più scarno, e altri casi di conversione implicita possono nascondere bug.
- **Errori che insegnano.** Messaggi in italiano con suggerimento e, spesso, la riga già corretta; l'autocorrezione e `leo controlla` colgono molti sbagli prima di eseguire.
- **Libreria inclusa.** Statistica, date all'italiana, numeri in lettere, password, impronte, tabelle, QR: tutto di serie, mentre in Python molte di queste cose sono pacchetti esterni da installare.
- **Parallelismo sicuro.** I lavori `in parallelo` sono isolati per costruzione: non possono pasticciare con i dati altrui, e Leo lo impedisce prima di eseguire. È più difficile farsi male che con i thread di Python.

Dove sono alla pari

Molto del cuore del linguaggio è equivalente, perché sotto c'è lo stesso motore: interi di dimensione illimitata, l'imprecisione dei decimali IEEE-754 (identica a Python), le liste e i dizionari, le comprensioni, le funzioni come valori, le chiusure, le eccezioni con `prova` / `cattura` / `infine`, i moduli. Chi viene da Python ritrova gli stessi concetti con nomi diversi. Lo slicing ha una differenza di stile: in Leo `a..b` include *entrambi* gli estremi, mentre in Python `a:b` esclude il secondo. Una scelta di Leo, per leggersi come l'italiano «da…a».

Dove Leo è indietro, e dove manca qualcosa

Onestà vuole anche questo. Python ha un ecosistema sterminato, centinaia di migliaia di pacchetti, e per i domini specialistici (calcolo scientifico pesante, machine learning, web server su larga scala) resta la scelta di

riferimento. Leo offre un ponte (`importa python "pacchetto"`) proprio per non rinunciarvi, ma è una funzionalità da esperti, non il terreno dove Leo dà il meglio. Alcune funzionalità molto avanzate di Python (metaprogrammazione spinta, certi protocolli del linguaggio) in Leo non hanno un equivalente diretto, e per scelta: la semplicità è un valore, e aggiungere potenza che pochi usano al prezzo della chiarezza non sarebbe nello spirito del progetto.

Come scegliere

La regola pratica è semplice. Se il tuo scopo è **imparare a programmare**, **insegnare**, scrivere **strumenti personali**, programmi per la **scuola**, app con finestre per il PC o il telefono, e farlo in una lingua che capisci senza attrito, Leo è pensato apposta per te, e in molte di queste cose fa *meno fatica* di Python. Se il tuo scopo è integrarti in un grande progetto Python esistente, o sfruttare un pacchetto specialistico dell'ecosistema, Python resta la via maestra. E Leo, se vuoi, ti ci accompagna con `importa python`.

Non è una gara a somma zero: Leo è costruito *su* Python, e questo significa che imparando Leo hai già un piede nel mondo Python, con la gentilezza dell'italiano a farti da guida.

Appendice D — Glossario

Le parole che tornano in tutto il libro, spiegate in breve e senza giri. In *corsivo* i rimandi ad altre voci.

Ambito (o visibilità). La zona del programma in cui una *variabile* esiste ed è utilizzabile. Una variabile dichiarata dentro un *blocco* vive solo lì.

Argomento. Il valore che passi a una *funzione* quando la chiami. Da non confondere con il *parametro*, che è il nome con cui la funzione lo riceve.

Blocco. Un gruppo di istruzioni che stanno insieme, aperto da `allora` o `fai` e chiuso da `fine` (o senza aperture, come in `ripeti N volte`).

Canale. Un «tubo» per scambiare messaggi tra *lavori* che girano *in parallelo*: uno invia, l'altro riceve, in ordine.

Chiave. L'etichetta con cui ritrovi un valore in un *dizionario*.

Chiusura. Una *funzione* che ricorda le *variabili* del posto in cui è nata, anche dopo che quel posto ha terminato l'esecuzione.

Classe. Lo stampo da cui si creano gli *oggetti*: unisce *dati* (campi, dichiarati con `ha`) e *comportamenti* (azioni, con `azione`).

Comprensione. La forma compatta per costruire una *lista* calcolando ogni elemento: `[n * n per ogni n in 1..5]`.

Condizione. Un'espressione che vale `vero` o `falso`, usata da `se` e `mentre` per decidere.

Costante. Un valore con un nome che, una volta dichiarato con `costante`, non si può più cambiare.

Decimale. Un numero con la virgola, scritto in Leo con il **punto** (`3.14`). Soggetto all'imprecisione IEEE-754: per il denaro usa *soldi*.

Dizionario. Un contenitore di coppie *chiave* → valore: `{"nome": "Ada"}`.

Ereditarietà. Il meccanismo per cui una *classe* (`estende` un'altra) ne riceve dati e comportamenti, potendone ridefinire alcuni.

Eccezione (errore). Un evento anomalo durante l'esecuzione (`ErroreValore`, `ErroreIndice`…), che si può gestire con `prova` / `cattura`.

Evento. Qualcosa che accade in un programma a finestre (un clic, l'apertura, un tasto), a cui si reagisce con un gestore `quando`.

Funzione. Un blocco di codice con un nome, che riceve *argomenti* e può restituire un valore con `restituisci`. In Leo è anche un *valore*.

Impronta. Un codice crittografico (SHA-256) che riassume un dato: cambia del tutto per la minima modifica, ma non permette di risalire al dato. Si calcola con `impronta`.

Interpolazione. L'inserimento di un valore dentro un *testo* tramite le graffe: `"Ciao {nome}"`.

Intero. Un numero senza virgola, di dimensione illimitata in Leo.

Intervallo. Una sequenza di numeri da un estremo all'altro, estremi compresi: `1..5`. Tra le quadre indica una *fetta*.

Interprete. Il programma che legge ed esegue il tuo codice. Quello di Leo è costruito su Python.

Lavoro. Un'unità di calcolo avviata *in parallelo*, isolata dagli altri.

Lista. Una sequenza ordinata di valori, numerati da 0: `[1, 2, 3]`. Si può modificare.

Metodo. Una *funzione* che appartiene a un valore e si chiama con il punto: `testo.maiuscolo()`, `lista.aggiungi(x)`.

Modulo. Un file `.leo` importato in un altro con `importa`, per dividere il programma.

Oggetto. Un singolo esemplare creato da una *classe*, con i propri *dati*.

Parametro. Il nome con cui una *funzione* riceve un *argomento*.

Parola chiave. Una parola riservata del linguaggio (`se`, `mentre`, `funzione`…), che non si può usare come nome.

REPL. L'ambiente interattivo (`leo` senza file) in cui provi il codice una riga alla volta.

Ricorsione. Una *funzione* che chiama sé stessa; richiede sempre un *caso base* che la fermi.

Sequenza pigra. Una sequenza che produce i suoi elementi uno alla volta, solo quando servono: può essere anche infinita (`numeri_infiniti`).

Sinonimo. Una parola che Leo interpreta come una *parola chiave* (per esempio `finché` per `mentre`, `azione` per `funzione`).

Slicing (fetta). Prendere un pezzo di una *lista* o di un *testo* con un *intervallo*: `voti[0..2]`.

Soldi. Il tipo per il denaro, che tiene i centesimi esatti: `soldi("1.50")`.

Testo (stringa). Una sequenza di caratteri tra virgolette: `"ciao"`. Non si modifica.

Tipo. La natura di un valore: intero, decimale, testo, logico, lista, dizionario, funzione, e altri. Leo è a *tipizzazione forte*: non mescola i tipi di nascosto.

Variabile. Un nome a cui è associato un valore, che può cambiare. Si dichiara con `variabile`.

Le pagine interattive

Questo libro ha un gemello sul web: ogni grande tema dell'informatica ha una pagina da **toccare ed esplorare** — i bit da accendere, le porte logiche, il processore in 3D, gli algoritmi animati. Sulla carta inquadri il **QR**; nell'ebook tocchi il **link**.

- **Indice di tutte le pagine** — https://leoitalia.com/informatica/
- Tutto è bit — https://leoitalia.com/informatica/bit.html
- La logica — https://leoitalia.com/informatica/logica.html
- Dentro il computer (il processore in 3D) — https://leoitalia.com/informatica/cpu.html
- Gli algoritmi — https://leoitalia.com/informatica/algoritmi.html
- I dati — https://leoitalia.com/informatica/dati.html
- Reti e Internet — https://leoitalia.com/informatica/reti.html
- Sicurezza — https://leoitalia.com/informatica/sicurezza.html
- Scrivere in Leo — https://leoitalia.com/informatica/scrivere-leo.html

Inquadra e prova

Ogni grande tema di questo libro ha una pagina da toccare ed esplorare. Inquadra un QR col telefono e l'argomento prende vita: i bit da accendere, le porte logiche, il processore in 3D, gli algoritmi animati.

Indice

Tutto è bit

La logica

Dentro il computer

Gli algoritmi

I dati

Reti e Internet

Sicurezza

Scrivere in Leo