# Dispensa · Modulo 1

## 1. Che cosa facciamo in questo corso

I linguaggi di programmazione sono tutti equivalenti dal punto di vista di quello che possono esprimere: qualsiasi programma scritto in Python potrebbe essere riscritto in Java, in C, in Ruby o in qualsiasi altro linguaggio general-purpose. La scelta del linguaggio cambia la sintassi, non le idee sottostanti. Il corso usa Python come strumento per imparare a codificare problemi in modo chiaro e testabile, senza concentrarsi sulle sue specificità. Le strutture affrontate — variabili, condizioni, cicli, funzioni — esistono in quasi tutti i linguaggi di programmazione con forme diverse ma significato analogo.

Il corso lavora su tre livelli contemporaneamente. Il primo è la **sintassi** di Python: le regole formali che stabiliscono come si scrive un'istruzione valida. Il secondo è la **semantica** di base, cioè che cosa significa davvero quello che si scrive — un livello che vale per tutti i linguaggi, non solo per Python. Il terzo è un **metodo** di lavoro: come affrontare un problema, come leggere un errore, come correggere un programma che non funziona come previsto.

Non rientrano negli obiettivi del corso, almeno in questa fase, lo stile pythonic avanzato, la programmazione a oggetti, il paradigma funzionale o l'ottimizzazione algoritmica. L'obiettivo è più semplice e più importante: prima si impara a leggere, scrivere e controllare programmi semplici; poi si usano queste basi per lavorare su dati e problemi reali.

## 2. Python in quattro aggettivi

Python è un linguaggio **multiparadigma**, **dinamicamente tipizzato**, **interpretato** e **ad alto livello**. Questi quattro aggettivi descrivono scelte progettuali precise, ognuna con conseguenze concrete sul modo di lavorare.

| Aggettivo | Significato |
|---|---|
| **ad alto livello** | il codice è vicino al modo in cui ragiona un essere umano, lontano dalle istruzioni della macchina |
| **interpretato** | il programma viene eseguito riga per riga da un interprete, non tradotto tutto in anticipo |
| **dinamicamente tipizzato** | i tipi non si dichiarano: vengono determinati durante l'esecuzione, e operazioni tra tipi incompatibili producono un errore |
| **multiparadigma** | supporta stili diversi di programmazione (imperativo, funzionale, a oggetti) |

Dire che Python è **ad alto livello** significa che è possibile scrivere `len("ciao")` per contare i caratteri di una parola senza preoccuparsi di come la CPU esegua concretamente quell'operazione in memoria. Più un linguaggio è ad alto livello, più il codice assomiglia al ragionamento umano e meno al funzionamento interno della macchina.

Dire che è **interpretato** significa che non esiste una fase separata di traduzione prima dell'esecuzione: l'interprete legge il programma e lo esegue riga per riga, direttamente. Questa caratteristica rende possibile la REPL, la modalità interattiva in cui si scrive un'istruzione e se ne vede subito il risultato.

Dire che è **dinamicamente tipizzato** significa che non bisogna dichiarare il tipo di una variabile prima di usarla: Python lo determina automaticamente a partire dal valore assegnato. Se si tenta

un'operazione tra tipi incompatibili, come `3 + "ciao"`, l'errore non viene segnalato in anticipo ma emerge durante l'esecuzione, nel momento in cui quella riga viene raggiunta.

Infine, **multiparadigma** significa che Python non impone un unico stile di programmazione: supporta lo stile imperativo (istruzioni in sequenza), quello funzionale e quello a oggetti. Nel corso si usa quasi sempre lo stile imperativo.

## 3. Python come linguaggio formale

Un **linguaggio formale** è un linguaggio in cui ogni simbolo ha un significato preciso e le regole di combinazione sono rigide. A differenza dell'italiano, non ammette ambiguità: l'interprete non interpreta "più o meno", non corregge, non suppone. O l'istruzione è scritta esattamente come previsto, o il programma non funziona.

La differenza con il linguaggio naturale è immediata. In italiano, la frase "Ho visto l'uomo con il cannocchiale" ha due strutture grammaticali distinte, entrambe valide: ho visto, *usando il cannocchiale*, un uomo; oppure ho visto un uomo *che aveva* il cannocchiale. Un parlante umano sceglie l'interpretazione giusta dal contesto della conversazione. Un interprete Python non ha accesso a nessun contesto: ogni istruzione deve avere un solo significato possibile, determinato unicamente dalla sua forma.

Un linguaggio formale si analizza su tre livelli distinti.

| Componente | Domanda | Esempi in Python |
|---|---|---|
| **Vocabolario** | Quali simboli sono ammessi? | `3`, `"ciao"`, `+`, `if`, `def` |
| **Sintassi** | Come si combinano correttamente? | `3 + 2` — `3 + * 2` — `3 +` |
| **Semantica** | Che cosa significa quello che si scrive? | `3 + 2` produce `5`; `"a" + 2` è un TypeError |

Il **vocabolario** è l'insieme dei simboli che il linguaggio riconosce: numeri, stringhe, operatori, parole chiave. La **sintassi** stabilisce come quei simboli possono essere combinati: `3 + 2` è valido, `3 +` no, perché la struttura è incompleta. La **semantica** riguarda il significato dell'operazione: `3 + 2` è sintatticamente corretto e semanticamente valido, `"a" + 2` è sintatticamente corretto ma semanticamente non ha senso in Python, perché non si possono sommare una stringa e un numero.

### Errori di sintassi ed errori di tipo

Questa distinzione è utile fin da subito per leggere i messaggi di errore, che sono tra le prime cose che si incontrano quando si programma. Un **errore di sintassi** (`SyntaxError`) significa che l'interprete non riesce nemmeno a leggere l'istruzione: è come una frase con le parole in un ordine impossibile. Un **errore di tipo** (`TypeError`) significa invece che la sintassi era corretta, ma l'operazione non ha senso per i valori usati.

```
3 +       # SyntaxError: struttura incompleta
3 + "ciao" # TypeError: non si può sommare un intero e una stringa
```

Imparare a leggere i messaggi di errore è parte integrante del lavoro. L'errore non è solo un ostacolo: è spesso la spiegazione più diretta e precisa di che cosa non funziona.

## 4. Che cosa si installa davvero

Quando si dice "installare Python" si sta usando una scorciatoia linguistica. Python, in quanto linguaggio, è una specifica formale: un insieme di regole che definiscono vocabolario, sintassi e semantica. In quanto tale non può essere installato — è solo un documento, un insieme di definizioni. Quello che si installa concretamente è l'**interprete**, cioè un programma capace di leggere il codice sorgente scritto in Python, verificarne la sintassi ed eseguire le istruzioni una alla volta.

La distinzione tra linguaggio e interprete non è solo teorica. Esistono più implementazioni dell'interprete Python: CPython è quella di riferimento, ma ne esistono altre. Tutte eseguono lo stesso linguaggio, con caratteristiche tecniche diverse.

### Linguaggi ad alto e basso livello

Non tutti i linguaggi di programmazione stanno alla stessa distanza dalla macchina.

| Livello | Idea generale | Esempi |
|---|---|---|
| **Alto livello** | più vicino al modo in cui ragiona un essere umano | Python, Java |
| **Basso livello** | più vicino alle istruzioni effettive della macchina | C, Assembly |

Per rendere concreta la differenza, si consideri il calcolo `a = 3 + 5` che salva il valore `8` in una posizione di memoria disponibile per Python.

In Python si scrive una sola riga leggibile:

```
a = 3 + 5
```

In Assembly, la stessa operazione richiede istruzioni esplicite per ogni singolo passo che la CPU deve compiere:

```
mov eax, 3    ; carica il valore 3 nel registro eax
add eax, 5    ; somma 5 al contenuto di eax
mov [a], eax  ; salva il risultato nella cella di memoria chiamata a
```

La CPU non conosce il concetto di "somma di due numeri e salvataggio in una variabile": conosce solo operazioni elementari su registri e celle di memoria. In Python tutto questo è nascosto dietro una singola riga; l'interprete si occupa di tradurla nelle istruzioni di basso livello necessarie.

### Interprete e compilatore

Interprete e compilatore sono due strategie diverse per tradurre un programma scritto in linguaggio ad alto livello nelle istruzioni eseguibili dalla macchina.

| Strumento | Che cosa fa | Esempio tipico |
|---|---|---|
| **Interprete** | legge ed esegue il programma riga per riga | Python, Javascript |
| **Compilatore** | traduce tutto il programma in anticipo, poi produce un eseguibile | C, Java |

**Il compilatore** prende il codice sorgente e produce un file in linguaggio macchina che la CPU può eseguire direttamente — il cosiddetto **eseguibile**. Questo processo si chiama **compilazione** e avviene una volta sola: l'eseguibile può girare su qualunque macchina compatibile senza che il codice sorgente sia presente. La traduzione è già avvenuta.

**L'interprete**, al contrario, non produce nessun eseguibile permanente. Ogni volta che il programma viene avviato, l'interprete rilegge il codice sorgente e lo esegue istruzione per istruzione. Il codice sorgente deve essere sempre disponibile.

Per rendere intuitiva la differenza, si può immaginare un libro scritto in una lingua sconosciuta. Con un compilatore, un traduttore traduce l'intero libro e consegna il testo nella lingua di destinazione: se c'è un errore a pagina 325, il traduttore se ne accorge *prima* che il libro venga aperto. Con un interprete, il traduttore legge e traduce riga per riga durante la lettura: se c'è un errore a pagina 325, non emerge finché non si arriva a quella pagina.

| | Interprete | Compilatore |
|---|---|---|
| **Quando si scoprono gli errori** | durante l'esecuzione, alla riga sbagliata | prima dell'esecuzione, in fase di traduzione |
| **Velocità di esecuzione** | più lenta — la traduzione avviene a runtime | più veloce — la traduzione è già fatta |
| **Ciclo di sviluppo** | rapido — modifica ed esegui subito | più lento — bisogna ricompilare ad ogni modifica |
| **Eseguibile prodotto** | no — il codice viene riletto ogni volta | sì — gira senza rifare la traduzione |

**Una nota su Python.** Python non è un interprete puro. Quando si esegue un file `.py`, Python lo compila internamente in un formato intermedio chiamato **bytecode** (i file `.pyc` che compaiono nella cartella `__pycache__`). Questo bytecode non è linguaggio macchina diretto, ma viene eseguito da una macchina virtuale Python. Si tratta di un approccio ibrido, comune a molti linguaggi moderni, che combina la flessibilità dell'interpretazione con una parte dell'efficienza della compilazione.

## 5. Architettura di Von Neumann

Tutti i computer moderni seguono un'architettura descritta da Von Neumann negli anni '40, che rimane il modello di riferimento ancora oggi. Prima di Von Neumann, i computer venivano "programmati" ricablando fisicamente i circuiti per ogni nuovo compito. L'intuizione fondamentale che porta il suo nome è più semplice e più potente: le istruzioni possono essere trattate come dati. Possono essere scritte in memoria, lette, modificate, spostate — esattamente come i valori su cui il programma opera. Da questa idea discende il computer a programma memorizzato: **un programma è una sequenza di istruzioni memorizzate insieme ai dati**.

### La CPU

Il cuore del sistema è la **CPU** (Central Processing Unit). Il suo compito è leggere le istruzioni una alla volta ed eseguirle. Non ha una visione globale del programma: sa solo qual è l'istruzione corrente, la esegue, e passa alla successiva. Al suo interno la CPU contiene tre componenti principali.

La **Control Unit** è il direttore d'orchestra: legge l'istruzione corrente dalla memoria, la decodifica (cioè capisce di che tipo di operazione si tratta) e coordina gli altri componenti affinché la eseguano.

È la Control Unit che decide quale istruzione eseguire dopo — normalmente la successiva in memoria, a meno che l'istruzione corrente non sia un salto condizionale (come accade con un `if`).

L'**ALU** (Arithmetic Logic Unit) è l'unità che esegue i calcoli veri e propri: somme, sottrazioni, moltiplicazioni, confronti. Quando Python valuta `3 + 5` o controlla se `x > 0`, è l'ALU che produce il risultato. Non sa nulla del programma nel suo complesso: riceve due valori, esegue un'operazione, restituisce un risultato.

I **registri** sono celle di memoria minuscole fisicamente all'interno della CPU, ed è lì che i valori vengono tenuti mentre la CPU ci lavora. Sono pochissimi ma estremamente veloci. Quando l'ALU deve sommare due numeri, li carica prima nei registri, esegue l'operazione, e deposita il risultato in un altro registro.

### Memoria e disco

La distinzione tra RAM e disco è cruciale e spesso fonte di confusione, perché entrambe "conservano" informazioni — ma in modi radicalmente diversi.

La **RAM** (Random Access Memory) è la memoria di lavoro del computer. È rapida — la CPU può accedervi in tempi dell'ordine dei nanosecondi — ma **volatile**: si svuota completamente quando il computer viene spento. Tutto ciò che un programma in esecuzione usa — variabili, valori intermedi, lo stesso codice del programma — risiede in RAM mentre il programma gira.

Il **disco** (hard disk o SSD) è la memoria permanente. È molto più lento della RAM, ma **persistente**: i dati rimangono anche senza corrente. I file `.py`, i documenti, le immagini vivono sul disco. Quando si spegne il computer, il disco conserva tutto; la RAM dimentica tutto.

| Componente | Velocità | Persistenza | Cosa ci vive |
|---|---|---|---|
| **RAM** | veloce | volatile (si svuota a spegnimento) | variabili, istruzioni del programma in esecuzione |
| **Disco** | lento | persistente (resta senza corrente) | file `.py`, documenti, immagini |

### Cosa succede quando un programma viene eseguito

Quando si lancia `python3 programma.py`, si mette in moto una sequenza precisa di eventi. Il **sistema operativo** individua il file sul disco e ne copia il contenuto in RAM: solo a quel punto il programma è "in memoria" e può essere eseguito. La **Control Unit** prende la prima istruzione dalla RAM, la decodifica e attiva l'ALU se è necessario un calcolo. L'**ALU** esegue l'operazione usando i registri come appoggio temporaneo, e il risultato viene scritto di nuovo in RAM — ad esempio nella cella corrispondente a una variabile. Poi la Control Unit passa all'istruzione successiva, e il ciclo ricomincia. Questo ciclo — fetch, decode, execute — si ripete per ogni singola istruzione del programma, centinaia di milioni di volte al secondo.

Alla fine, se il programma deve mostrare qualcosa sullo schermo o scrivere un file, il risultato viene inviato ai **dispositivi di output**. Tutto ciò che il programma produce e usa durante l'esecuzione vive in RAM: quando il programma termina, quella memoria viene liberata e i dati scompaiono. Per conservare un risultato bisogna scriverlo esplicitamente su disco — ad esempio con `print()` verso lo schermo, o aprendo un file in scrittura.

La macchina non capisce l'intenzione generale del programma. Esegue istruzioni. Il significato è nel programma, non nella macchina.

## 6. Cos'è un programma

Nel linguaggio comune la parola *programma* indica un'applicazione completa: Word, Excel, un browser. In queste dispense il termine è usato in un senso più specifico e più ristretto: **una sequenza di azioni che prende dati in ingresso, li elabora e produce un risultato**. Ogni programma, per quanto piccolo, risponde a tre domande: quali dati riceve in ingresso? Quali regole applica per trasformarli? Che risultato restituisce?

Questo schema — input, elaborazione, output — vale già per i programmi più semplici. Un programma che calcola la somma di due numeri riceve `a` e `b`, applica l'addizione, e restituisce `a + b`. Un programma che conta i caratteri di una stringa riceve la stringa, ne calcola la lunghezza, e restituisce un numero intero. La complessità cresce, ma lo schema rimane.

## 7. REPL: che cos'è e a che cosa serve

La **REPL** è la modalità interattiva di Python. L'acronimo — Read, Eval, Print, Loop — descrive esattamente il ciclo che si ripete: l'interprete legge l'istruzione digitata, la valuta, stampa il risultato, e poi ricomincia ad aspettare la successiva. Si avvia dal terminale con:

```
python3
```

Il prompt `>>>` indica che l'interprete è pronto a ricevere un'istruzione. Si può scrivere qualsiasi espressione Python e vederne subito il risultato:

```
>>> 3 * (2 + 4)
18
```

La REPL è particolarmente utile nelle prime fasi: permette di fare prove rapide, controllare il comportamento di un'operazione, osservare immediatamente il risultato di un calcolo, e leggere i messaggi di errore senza dover creare un file. È uno strumento di esplorazione, non di sviluppo: i programmi veri si scrivono in file e si eseguono dal terminale.

## 8. Tipi fondamentali e operazioni

Ogni valore in Python appartiene a un **tipo**, che determina quali operazioni hanno senso su di esso. Nel modulo 1 si lavora con tre tipi fondamentali: `int` per i numeri interi (come `0`, `7`, `-12`), `float` per i numeri con la virgola (come `3.14`, `0.5`, `-2.0`), e `str` per le stringhe di caratteri (come `"ciao"` o `'Python'`).

### Operazioni sui numeri

| Operazione | Sintassi | Esempio | Risultato |
|---|---|---|---|
| Somma | `N + N` | `3 + 15` | `18` |
| Sottrazione | `N - N` | `10 - 4` | `6` |
| Moltiplicazione | `N * N` | `3 * 7` | `21` |
| Divisione | `N / N` | `7 / 2` | `3.5` |
| Potenza | `N ** N` | `3 ** 2` | `9` |
| Divisione intera | `N // N` | `17 // 5` | `3` |

| Operazione | Sintassi | Esempio | Risultato |
|---|---|---|---|
| Modulo (resto) | `N % N` | `17 % 5` | `2` |

Vale la pena soffermarsi su due operatori meno familiari. L'operatore `//` calcola il **quoziente** della divisione intera: `17 // 5` dà `3`, perché 5 entra in 17 tre volte. L'operatore `%` calcola il **resto**: `17 % 5` dà `2`, perché dopo aver sottratto tre volte 5 da 17 rimane 2. Quanto alla divisione ordinaria `/`, restituisce **sempre** un `float`, anche quando il risultato è intero: `4 / 2` dà `2.0`, non `2`.

### Operazioni sulle stringhe

| Operazione | Sintassi | Esempio | Risultato |
|---|---|---|---|
| Concatenazione | `S + S` | `"ab" + "cd"` | `"abcd"` |
| Ripetizione | `S * N` | `"ciao" * 3` | `"ciaociaociao"` |
| Selezione | `S[i]` | `"ciao"[1]` | `"i"` |
| Lunghezza | `len(S)` | `len("ciao")` | `4` |
| Maiuscolo | `S.upper()` | `"ciao".upper()` | `"CIAO"` |
| Minuscolo | `S.lower()` | `"CIAO".lower()` | `"ciao"` |
| Conteggio | `S.count(S2)` | `"aaa".count("a")` | `3` |

Una caratteristica importante dell'operatore `+` è che il suo significato dipende dal tipo degli operandi: tra numeri significa somma, tra stringhe significa concatenazione. Scrivere `3 + 15` produce `18`; scrivere `"3" + "15"` produce `"315"`. Non si tratta di un errore, ma di un comportamento preciso: le due stringhe vengono accostate, non sommate aritmeticamente.

### Operazioni che cambiano tipo

Alcune espressioni restituiscono un valore di tipo diverso rispetto agli operandi. Questo può sorprendere all'inizio, ma segue regole precise.

| Espressione | Tipo in ingresso | Tipo in uscita | Cambia tipo? |
|---|---|---|---|
| `3 + 5` | `int` + `int` | `int` | no |
| `"ab" + "cd"` | `str` + `str` | `str` | no |
| `"ciao".upper()` | `str` | `str` | no |
| `7 / 2` | `int` / `int` | `float` | **sì** |
| `len("ciao")` | `str` | `int` | **sì** |
| `"ciao" * 3` | `str` × `int` | `str` | — |

Nel modulo 2 si incontreranno anche `int()` e `str()`, funzioni che cambiano tipo in modo esplicito: si chiamano **cast** e permettono, ad esempio, di trasformare la stringa `"42"` nel numero intero `42`.

## 9. Selezione di caratteri e slicing

Le stringhe in Python sono sequenze di caratteri, e ogni carattere ha una posizione numerata. La convenzione fondamentale dell'informatica è che si conta a partire da zero, non da uno. Il primo

carattere di `"informatica"` è in posizione `0`, il secondo in posizione `1`, e così via fino all'undicesimo in posizione `10`.

```
 i   n   f   o   r   m   a   t   i   c   a
 0   1   2   3   4   5   6   7   8   9  10
-11 -10  -9  -8  -7  -6  -5  -4  -3  -2  -1
```

Python permette anche di contare dal fondo usando indici negativi: `-1` indica l'ultimo carattere, `-2` il penultimo, e così via. Questa simmetria è comoda quando si vuole accedere alla fine di una stringa senza conoscerne la lunghezza in anticipo.

```
>>> "informatica"[0]
'i'
>>> "informatica"[-1]
'a'
>>> "informatica"[-3]
'i'
```

Oltre alla selezione di un singolo carattere, è possibile estrarre sottostringhe con l'operazione di **slicing**, specificando un intervallo di indici separati da `:`. Il carattere iniziale è sempre **incluso**, quello finale è sempre **escluso**. Omettere il primo indice significa "dall'inizio della stringa"; omettere il secondo significa "fino alla fine".

```
>>> "informatica"[0:3]
'inf'
>>> "informatica"[:3]
'inf'
>>> "informatica"[-3:]
'ica'
>>> "informatica"[2:]
'formatica'
```

## 10. REPL e script si comportano diversamente

Una delle distinzioni più importanti del modulo riguarda il comportamento della REPL rispetto a quello di uno script. Nella REPL, ogni espressione valutata produce automaticamente un output visibile: scrivere `3 + 5` e premere Invio mostra `8` sullo schermo. Questo avviene per comodità interattiva, non perché il programma stia "stampando" qualcosa.

In un file `.py`, le cose funzionano diversamente. Se si scrive `3 + 5` in uno script e lo si esegue, il calcolo viene effettuato — ma il risultato non viene mostrato da nessuna parte, perché nessuno ha dato l'istruzione di farlo. Per produrre output visibile in uno script è necessario usare esplicitamente `print(...)`:

```
# questo non produce alcun output:
3 + 5

# questo sì:
print(3 + 5)
```

La distinzione è importante perché la REPL crea l'abitudine di vedere sempre un risultato. Passando agli script, bisogna ricordare che l'output va richiesto esplicitamente.

Vale anche notare che l'estensione `.py` è una convenzione per gli esseri umani — aiuta l'editor a riconoscere il tipo di file e a colorare il codice di conseguenza — ma l'interprete Python non la richiede. Un file con istruzioni Python valide viene eseguito correttamente indipendentemente dall'estensione.

## 11. Ambiente di lavoro

Per programmare servono tre strumenti distinti, ognuno con un ruolo preciso. L'**interprete** esegue il codice Python: è il programma che legge le istruzioni e le traduce in operazioni della macchina. L'**editor** è lo strumento con cui si scrivono e salvano i file di testo che contengono il codice: esempi comuni sono VS Code e VSCodium. Il **terminale** è l'interfaccia da cui si lancia l'interprete e si gestiscono file e cartelle.

È importante non confonderli. In particolare, Word non è un editor di testo nel senso rilevante per la programmazione: aggiunge al file formattazione, font e metadati che non fanno parte del testo puro. Python non può leggere un file `.docx` come se fosse uno script, perché il contenuto del file non è solo il testo visibile ma una struttura XML con molte informazioni aggiuntive. Un editor di testo salva invece esattamente e solo i caratteri digitati.

La sequenza tipica di lavoro è: scrivere il file nell'editor, aprire il terminale nella cartella giusta, eseguire `python3 nome_file.py`, osservare l'output e correggere se necessario. Questa sequenza si ripete molte volte durante ogni sessione di lavoro.

## 12. Shell e sistema operativo

Quando un programma accede a file, cartelle o risorse del computer, non interagisce direttamente con l'hardware: ogni operazione passa attraverso il **sistema operativo**, che gestisce il file system, la memoria, i processi e l'accesso alle periferiche. Il **terminale** è l'interfaccia a testo del sistema operativo, e il programma che interpreta i comandi digitati si chiama **shell**.

Sistemi operativi diversi usano shell diverse, con sintassi leggermente differenti.

| Sistema operativo | Shell predefinita | Prompt tipico |
|---|---|---|
| **macOS** | `zsh` | `%` |
| **Linux** | `bash` | `$` |
| **Windows** | PowerShell | `>` |

I comandi presentati nel corso funzionano allo stesso modo in bash e zsh. Su Windows si può usare **Git Bash**, installato insieme a Git, per seguire gli stessi esempi.

I comandi essenziali per muoversi nel file system sono pochi ma indispensabili: `pwd` mostra la cartella corrente, `ls` elenca il contenuto della cartella, `cd nome-cartella` entra in una sottocartella, `cd ..` risale alla cartella padre, `mkdir nome` crea una nuova cartella, e `python3 file.py` lancia l'esecuzione di uno script.

## 13. Percorsi assoluti e relativi

Quando un comando "non trova il file", il problema è quasi sempre il percorso. Capire come funzionano i percorsi è una competenza pratica fondamentale.

Un **percorso assoluto** descrive la posizione di un file a partire dalla radice del file system: su Unix comincia con `/`, su Windows con `C:\`. Un **percorso relativo** descrive invece la posizione di un file a partire dalla cartella in cui ci si trova al momento del comando. Lo stesso file può essere raggiunto con entrambi i tipi di percorso, a seconda del punto di partenza.

Si consideri questa struttura di esempio:

```
/home/studente/
├── corso/
│   ├── script.py
│   └── dati/
│       └── input.txt
└── documenti/
    └── appunti.txt
```

Partendo da `corso/`, questi percorsi sono equivalenti:

| Percorso | Tipo | Raggiunge |
|---|---|---|
| `/home/studente/corso/script.py` | assoluto | `script.py` |
| `script.py` oppure `./script.py` | relativo | `script.py` |
| `dati/input.txt` | relativo | `input.txt` |
| `../documenti/appunti.txt` | relativo (salendo di uno) | `appunti.txt` |

Tre scorciatoie facilitano la navigazione: `.` indica la cartella corrente, `..` indica la cartella superiore, e `~` indica la cartella home dell'utente (`/Users/tuonome` su macOS, `/home/tuonome` su Linux). Se un comando non trova un file, il primo controllo da fare è sempre la cartella corrente: molto spesso il problema è che si sta cercando il file nel posto sbagliato.

## 14. Da ricordare a fine modulo

Al termine del modulo si dovrebbe essere in grado di:

- distinguere interprete, editor e terminale
- aprire e usare la REPL di Python
- riconoscere `int`, `float` e `str` ed eseguire operazioni fondamentali su di essi
- spiegare la differenza tra errore di sintassi ed errore di tipo
- descrivere il modello input → elaborazione → output
- spiegare in modo intuitivo il ruolo di CPU, RAM e disco nell'esecuzione di un programma
- eseguire comandi base nel terminale e capire la differenza tra percorso assoluto e relativo
- capire perché uno script ha bisogno di `print(...)` per mostrare un risultato
- sapere che l'estensione `.py` è una convenzione per gli esseri umani (editor, sistema operativo), non un requisito dell'interprete