

# Python per principianti

Niccolò Nannucci

1a Edizione



# Prefazione e Ringraziamenti

## Chi sono?

Ciao!

Permettimi di presentarmi: mi chiamo Niccolò e sono uno studente di Ingegneria Gestionale, attualmente in procinto di completare la mia laurea triennale. Fin da bambino, ho sempre avuto una grande curiosità per tutto ciò che riguarda l'informatica. Ho iniziato cercando di capire come funzionano i programmi sul computer, poi sperimentando e mettendo alla prova la mia creatività, fino ad arrivare a sviluppare progetti nel campo della programmazione.

Nonostante il mio percorso accademico non sia specificatamente focalizzato sull'informatica, ho potuto applicare le metodologie ingegneristiche anche al mondo della programmazione. Questo mi ha permesso di sviluppare un approccio analitico alla risoluzione dei problemi, che, combinato con la mia passione, ha contribuito alla nascita di questo libro. Sono consapevole che la mia esperienza potrebbe non essere quella di un professionista affermato, ma il mio impegno nel migliorarmi continuamente e nel rimanere aggiornato su un campo in rapida evoluzione mi motiva a condividere le conoscenze che ho acquisito. Questo libro è pensato per coloro che si avvicinano per la prima volta a Python o alla programmazione in generale, con l'obiettivo di rendere questo mondo accessibile a tutti.

## Come è nato questo libro?

Come già accennato, studio Ingegneria Gestionale, una facoltà che, soprattutto nei primi anni, offre una panoramica di concetti provenienti da diversi ambiti. Questo significa che è piuttosto comune incontrare corsi che non si allineano perfettamente con i propri interessi o competenze.

Questo libro nasce proprio da quelle persone che, conoscendo la mia passione per l'informatica e il coding, hanno chiesto il mio aiuto per superare un esame universitario o per soddisfare le loro curiosità, preferendo un contatto diretto con me piuttosto che una semplice ricerca online. Nell'Ottobre del 2023, ho deciso di creare un testo struc-

turato e metodico che racchiudesse i concetti principali della programmazione di base in Python, come supporto futuro per chi si troverà di fronte a questi “ostacoli”, che possono trasformarsi in grandi opportunità per scoprire nuove passioni e talenti, nonché dare una svolta alla propria vita.

## A chi si rivolge?

*Python per principianti*, come suggerisce il titolo stesso, si rivolge a chiunque si avvicini a Python per la prima volta, o addirittura a chi non ha mai programmato in nessun linguaggio. I contenuti sono proposti in modo che chiunque possa comprendere anche i concetti più astratti, con l’obiettivo di coinvolgere tutti.

Non importa se sei uno studente che deve superare un esame universitario, un hobbyista (come me) che vuole sperimentare con Python, o un professionista che desidera ampliare le proprie competenze partendo dalle basi: questo libro è per te.

## Come è strutturato il libro?

Il libro è diviso principalmente in due parti. Escludendo il capitolo 0, dedicato a una panoramica del linguaggio “per rompere il ghiaccio” e fare le dovute presentazioni, la prima parte (capitoli da 1 a 9) si concentra sui concetti fondamentali, con un “punto di riflessione” nel capitolo Intermezzo. La seconda parte (capitoli da 10 a 14) esplora concetti più avanzati e raffina ulteriormente alcuni argomenti trattati in precedenza, in modo più formale.

Ogni capitolo è suddiviso in sezioni per separare i vari concetti, e ho utilizzato formattazioni specifiche (grassetto, corsivo, sottolineato...) per evidenziare le definizioni importanti e i concetti chiave. Durante la lettura, troverai numerose immagini relative a codici di esempio (corredati dal loro output e da una spiegazione passo-passo), oltre a tabelle e grafiche che presentano i concetti in modo chiaro e accattivante.

Il linguaggio cerca di essere il più possibile colloquiale, come se fossi io in persona a spiegarti i concetti a seguito di una tua domanda, nella speranza che questo stile rimuova la classica “barriera” del libro di testo freddo e impersonale. Talvolta, utilizzo analogie non necessariamente rigorose o formali per spiegare contenuti complessi, mantenendo così la semplicità anche quando i temi sono più avanzati.

Alla fine di ogni capitolo troverai un riassunto con i concetti principali, utile per consolidare la teoria appresa.

## Come seguire il libro?

Data la struttura lineare e progressiva del libro, ti consiglio di leggerlo in ordine, anche se sono disponibili vari indici per ritrovare velocemente un concetto o un'immagine specifici. Durante la lettura, farò spesso riferimento a concetti introdotti nei capitoli precedenti, quindi seguire l'ordine proposto ti garantirà di avere tutti gli strumenti necessari per affrontare i temi successivi.

Come ripeterò anche più avanti, la lettura da sola non basta: serve molta pratica e la voglia di “smanettare” sul proprio compilatore. Se qualcosa non è chiaro, rileggi e sfrutta gli esempi proposti. In sintesi: *trust the process* e lasciati accompagnare da questo libro alla scoperta di Python!

## Ringraziamenti

Questo libro non sarebbe mai nato senza l'influenza delle persone che mi hanno trasmesso l'amore per l'informatica fin da bambino, senza i docenti che mi hanno sprovvisto a fare di più, senza il supporto incondizionato della mia famiglia e senza gli amici che mi hanno sostenuto durante la stesura di questo testo, nato proprio dalle loro esigenze.

A tutti voi, dico un sincero *grazie*, perché grazie a voi ho finalmente realizzato un sogno che avevo fin da piccolo: scrivere un libro. E averlo fatto parlando della mia più grande passione è per me la soddisfazione più grande.

## Per segnalazioni di errori, commenti o suggerimenti...

Sono sempre aperto a feedback, suggerimenti, segnalazioni di errori nel testo e qualsiasi altra informazione utile a me e ai lettori di questo libro.

Se vuoi contattarmi, puoi scrivermi a [nannucci.books@gmail.com](mailto:nannucci.books@gmail.com).

Sarò felice di ascoltarti e migliorarmi!



# Prerequisiti

In questa sezione vengono illustrati i prerequisiti necessari per leggere e sfruttare al massimo questo testo.

*Si tenga presente che queste sono solo delle linee guida generiche e che i passaggi effettivi possono variare a causa di diversi fattori. Fare sempre riferimento alle documentazioni ufficiali dei vari prodotti.*

## Installare Python

Il primo passo è **installare** (o verificare di possedere già) Python sul proprio calcolatore. La procedura è simile per i sistemi operativi più comuni e viene illustrata di seguito. *Per eventuali problemi o informazioni aggiuntive fare riferimento alla documentazione ufficiale di Python.*

### • Windows

1. Visita il sito ufficiale di Python e recati alla pagina dei downloads per l'ultima versione disponibile (<https://www.python.org/downloads/windows/>).
2. Seleziona l'installer per Windows (64-bit oppure 32-bit a seconda dell'architettura del tuo computer) della versione più recente di Python.
3. Apri il file .exe scaricato e, nella schermata di setup, seleziona l'opzione "Add Python to PATH" prima di procedere.
4. Fai clic su "Install now" per installare Python in modo classico. L'operazione può richiedere un po' di tempo.
5. Al termine, verifica che Python sia stato installato correttamente aprendo la app "Prompt dei Comandi" e digitando `python3 --version`. Se l'output è la versione di Python installata allora l'installazione è avvenuta con successo!

### • macOS

1. Verifica se Python è già presente sul tuo Mac. Apri la app "Terminale" e digita `python3 --version`. Se l'output è la versione di Python, assicurati che sia

almeno la versione 3.12.0. In tal caso sei già pronto. Se la versione è obsoleta (o se Python non è installato) procedi al punto successivo.

2. Visita il sito ufficiale di Python e recati alla pagina dei downloads per l'ultima versione disponibile (<https://www.python.org/downloads/macos/>).
3. Seleziona l'installer universale per macOS a 64-bit (indipendentemente dall'architettura del tuo processore Intel o Apple Silicon) della versione più recente di Python.
4. Apri il pacchetto .pkg scaricato e segui le istruzioni dell'installer. L'operazione può richiedere un po' di tempo.
5. Al termine, verifica che Python sia stato installato correttamente aprendo la app "Terminale" e digitando `python3 --version`. Se l'output è la versione di Python installata allora l'installazione è avvenuta con successo!

- **Linux**

1. Solitamente, le maggiori distribuzioni Linux possiedono già Python installato. Puoi verificarlo aprendo il terminale e digitando `python3 --version`. Se l'output è la versione di Python, assicurati che sia almeno la versione 3.12.0. In tal caso sei già pronto. Se la versione è obsoleta (o se Python non è installato) procedi al punto successivo.
2. Utilizza il gestore dei pacchetti da terminale per installare la versione più recente di Python. Sui sistemi Ubuntu (e derivati) puoi utilizzare nell'ordine i comandi: `sudo apt-get update` e poi `sudo apt-get install python3` (devi avere i privilegi di super-utente). L'operazione può richiedere un po' di tempo.
3. Al termine, verifica che Python sia stato installato correttamente aprendo il terminale e digitando `python3 --version`. Se l'output è la versione di Python installata allora l'installazione è avvenuta con successo!

## Installare un IDE

Il secondo passo è **scaricare un buon IDE (= Ambiente di Sviluppo)**, ossia un programma che permette di scrivere, modificare e testare il codice che scriveremo (in questo caso nel linguaggio Python).

Python con l'installazione fornisce già un ambiente chiamato **IDLE**, una sorta di "blocco note" con capacità espanso... ma di certo non si può definire un IDE su cui lavorare in modo professionale.

Un discorso analogo vale per tutti i **siti online** che promettono di scrivere ed eseguire codice senza scaricare niente sul computer... non sono ambienti di sviluppo in senso stretto e spesso peccano di strumenti e aiuti che un buon IDE, invece, possiede.

Il lettore può autonomamente scegliere quale sia l'IDE che preferisce utilizzare, tuttavia il caldo consiglio che mi sento di dare è quello di scaricare **PyCharm Community**

**Edition di JetBrains**<sup>1</sup> (gratis con limitazioni, ma più che sufficiente per sperimentare con i concetti proposti in questo libro). (*Alcuni aspetti minori del libro sono stati esposti secondo le logiche e gli strumenti di questo IDE*).

Alternativamente, è un valido strumento **Visual Studio Code di Microsoft**<sup>2</sup>, completamente gratuito.

*Qualsiasi sia la tua scelta, seguì le istruzioni disponibili sui rispettivi siti degli IDE riguardo il completamento dell'installazione del software e successivamente l'impostazione di Python nell'ambiente di sviluppo.*

## Breve panoramica dell'IDE

Sia PyCharm che VSCode condividono una **struttura dell'area di lavoro** simile.

Al centro, nel riquadro più grande, vi è uno spazio con righe numerate: è dove andremo a **scrivere** i codici in Python.

In basso, solitamente, vi è un pannello che contiene la lista dei **problem**i nell'area di lavoro: qui appaiono gli errori gravi (il codice non funziona), gli avvisi (il codice può funzionare ma c'è comunque un problema da risolvere) e gli avvisi di lieve entità (raffinatezze del codice che in genere non influiscono sul funzionamento ma che andrebbero tenute in considerazione nell'ottica di una programmazione efficiente e conforme alle buone pratiche).

Lo stesso pannello, una volta **avviato** il codice (pulsante "Play" generalmente in alto a destra), mostra la **console** nella quale possiamo leggere gli output prodotti dal programma e talvolta inserire manualmente delle informazioni dinamiche che faremo elaborare al nostro codice.

Sulla sinistra, infine, troviamo la colonna dell'**esplorazione risorse** che mostra tutti i files e le cartelle del nostro progetto Python.

*Per una panoramica più dettagliata fai riferimento alle documentazioni ufficiali dei rispettivi IDE.*

## Come orientarsi nel libro

Le parti che compongono la schermata di un IDE, appena introdotte, puoi **ritrovarle nel libro** in forma stilizzata e semplificata attraverso le numerose immagini che raffigurano i codici e gli output degli esempi proposti.

Le figure che contengono codice e hanno le righe numerate sono la rappresentazione dell'area centrale dell'IDE dove scriviamo il **codice**.

---

<sup>1</sup><https://www.jetbrains.com/pycharm/download/>

<sup>2</sup><https://code.visualstudio.com/Download>

Le figure che contengono solo testo (senza righe numerate e senza cornice) rappresentano la **console** dell'IDE, con il suo output (ed eventuali input o errori d'esecuzione).

## Invito alla pratica

Ora che è tutto pronto, non resta che iniziare la lettura e imparare questo bellissimo linguaggio di programmazione!

Tuttavia, *one more thing*: seguire il libro è fondamentale, ma **mettere in pratica** i concetti appena visti, comprendere a fondo il codice, sperimentare e inventare nuovi programmi e metodi risolutivi è ciò che veramente contribuisce a padroneggiare i concetti e il linguaggio in generale. Non bisogna pensare a questo libro come un tutorial o uno strumento che possiede tutte le risposte (e non aspira a esserlo): sono invece la curiosità e la dedizione del lettore che potranno permettergli di comprendere questo linguaggio con la pratica e con il tempo... .

# Indice

<b>0 Introduzione</b>	<b>1</b>
0.1 Perché Python? . . . . .	1
0.2 Breve storia di Python . . . . .	4
0.3 Un salto avanti: 5 aspetti di Python . . . . .	4
0.3.1 Facilità d'uso . . . . .	4
0.3.2 Dichiarazioni di Flusso . . . . .	6
0.3.3 Interpretazione intuitiva . . . . .	8
0.3.4 Dati Composti . . . . .	11
0.3.5 Definizione di Funzioni . . . . .	14
<b>1 Fondamentali di Python</b>	<b>17</b>
1.1 Commenti . . . . .	17
1.2 Indentazione . . . . .	18
1.3 Funzione <code>print()</code> . . . . .	21
1.4 Funzione <code>input()</code> . . . . .	23
1.5 Variabili e Parole Riservate . . . . .	24
1.6 Operatori e Regole di Precedenza . . . . .	27
R1 Riassunto <i>Fondamentali di Python</i> . . . . .	28
<b>2 Logica e Condizioni</b>	<b>31</b>
2.1 Introduzione alla Logica e alle Condizioni . . . . .	31
2.2 Espressioni Booleane e Operatori di Confronto . . . . .	31
2.3 Operatori Logici e Algebra di Boole . . . . .	32
2.4 Condizioni . . . . .	35
2.4.1 Costrutto condizionale <code>if</code> . . . . .	35
2.4.2 Costrutto condizionale <code>if-else</code> . . . . .	37
2.4.3 Costrutto condizionale <code>if-elif-else</code> . . . . .	38
2.5 Prime nozioni sulle Guardie . . . . .	41
2.6 Condizioni Nidificate . . . . .	42
R2 Riassunto <i>Logica e Condizioni</i> . . . . .	47

<b>3 Iterazioni</b>	<b>53</b>
3.1 Introduzione alle Iterazioni . . . . .	53
3.2 Premesse . . . . .	53
3.2.1 Variabili Locali . . . . .	53
3.2.2 Concetto di Generalizzazione . . . . .	55
3.3 Costrutti Ciclici . . . . .	56
3.3.1 Ciclo <b>for</b> . . . . .	56
3.3.2 Ciclo <b>while</b> . . . . .	67
3.4 Cicli nidificati . . . . .	72
R3 Riassunto <i>Iterazioni</i> . . . . .	74
<b>4 Funzioni</b>	<b>77</b>
4.1 Introduzione alle Funzioni . . . . .	77
4.2 Sintassi . . . . .	77
4.2.1 Definizione e Parametri . . . . .	81
4.2.2 Corpo . . . . .	82
4.2.3 Istruzione <b>return</b> . . . . .	83
4.3 Chiamate a funzione e Diagrammi . . . . .	84
4.3.1 Diagrammi di flusso . . . . .	85
4.3.2 Diagrammi di stack . . . . .	86
4.4 Funzioni nidificate . . . . .	87
4.5 Ricorsione . . . . .	91
R4 Riassunto <i>Funzioni</i> . . . . .	96
<b>5 Stringhe</b>	<b>99</b>
5.1 Introduzione alle Stringhe . . . . .	99
5.2 Anatomia e tratti distintivi . . . . .	100
5.3 Operazioni sulle Stringhe . . . . .	100
5.3.1 Lunghezza . . . . .	100
5.3.2 Indexing . . . . .	101
5.3.3 Slicing . . . . .	102
5.3.4 Confronti . . . . .	103
5.3.5 Concatenamento . . . . .	105
5.3.6 Ripetizione . . . . .	105
5.3.7 Ricerca . . . . .	106
5.3.8 Formattazione . . . . .	107
5.3.9 Metodi comuni . . . . .	109
5.4 Esempi . . . . .	114
R5 Riassunto <i>Stringhe</i> . . . . .	123

<b>6 Liste</b>	<b>125</b>
6.1 Introduzione alle Liste . . . . .	125
6.2 Anatomia e tratti distintivi . . . . .	126
6.3 Operazioni sulle Liste . . . . .	126
6.3.1 Definizione di una lista . . . . .	126
6.3.2 List Comprehension . . . . .	128
6.3.3 Lunghezza . . . . .	129
6.3.4 Indexing . . . . .	130
6.3.5 Slicing, Clonazione e Alias . . . . .	133
6.3.6 Cancellazione . . . . .	134
6.3.7 Ricerca . . . . .	136
6.3.8 Concatenamento . . . . .	137
6.3.9 Ripetizione . . . . .	138
6.3.10 Metodi comuni . . . . .	139
6.3.11 Liste nidificate . . . . .	142
6.4 Matrici . . . . .	143
6.4.1 Cos'è una matrice? . . . . .	144
6.4.2 Matrici e Vettori . . . . .	147
6.4.3 Operazioni tra matrici . . . . .	147
6.4.4 Operazioni sulle matrici . . . . .	158
6.4.5 Matrici e Guardie . . . . .	163
6.5 Esempi . . . . .	164
R6 Riassunto <i>Liste</i> . . . . .	175
<b>7 Tuple</b>	<b>177</b>
7.1 Introduzione alle Tuple . . . . .	177
7.2 Anatomia e tratti distintivi . . . . .	178
7.2.1 Casi d'uso e ottimizzazione . . . . .	179
7.3 Operazioni sulle Tuple . . . . .	180
7.3.1 Definizione di una tupla . . . . .	180
7.3.2 Swapping, Packing e Unpacking . . . . .	182
7.3.3 Tuple VS Liste . . . . .	186
7.3.4 Metodi comuni . . . . .	186
7.4 <b>namedtuples</b> . . . . .	187
7.5 Esempi . . . . .	189
R7 Riassunto <i>Tuple</i> . . . . .	196
<b>8 Set</b>	<b>197</b>
8.1 Introduzione ai Set . . . . .	197
8.2 Anatomia e tratti distintivi . . . . .	197
8.3 Operazioni sui set . . . . .	199

8.3.1	Definizione di un set . . . . .	199
8.3.2	Operazioni tipiche . . . . .	201
8.3.3	Operazioni non consentite . . . . .	201
8.3.4	Operazioni tra insiemi . . . . .	202
8.3.5	Metodi comuni . . . . .	203
8.4	Esempi . . . . .	205
R8	Riassunto <i>Set</i> . . . . .	209
<b>9</b>	<b>Dizionari</b>	<b>211</b>
9.1	Introduzione ai Dizionari . . . . .	211
9.2	Anatomia e tratti distintivi . . . . .	212
9.2.1	Chiavi e Valori . . . . .	212
9.2.2	Casi d'uso . . . . .	214
9.3	Operazioni sui Dizionari . . . . .	216
9.3.1	Definizione di un dizionario . . . . .	216
9.3.2	Lunghezza . . . . .	216
9.3.3	Indexing . . . . .	216
9.3.4	Cancellazione . . . . .	218
9.3.5	Ottener le Chiavi . . . . .	219
9.3.6	Ottener i Valori . . . . .	220
9.3.7	Clonazione e Alias . . . . .	221
9.3.8	Operazioni non supportate . . . . .	221
9.3.9	Metodi comuni . . . . .	221
9.4	Matrici sparse . . . . .	223
9.5	Esempi . . . . .	225
R9	Riassunto <i>Dizionari</i> . . . . .	235
<b>Intermezzo</b>		<b>237</b>
Introduzione . . . . .	237	
Punto sulle Strutture Dati . . . . .	238	
Cos'è la OOP? . . . . .	239	
Anteprima dei concetti avanzati . . . . .	239	
<b>10</b>	<b>Classi e Oggetti</b>	<b>241</b>
10.1	Introduzione alle Classi e agli Oggetti . . . . .	241
10.2	Oggetti . . . . .	241
10.3	Classi . . . . .	242
10.3.1	Anatomia e tratti distintivi . . . . .	242
10.3.2	Attributi . . . . .	243
10.3.3	Costruttore . . . . .	244
10.3.4	Rappresentazione . . . . .	246

10.3.5 Metodi . . . . .	248
10.3.6 Encapsulation . . . . .	249
10.3.7 Confronto tra Oggetti . . . . .	253
10.4 Ereditarietà . . . . .	256
10.5 Esempi . . . . .	260
R10 Riassunto <i>Classi e Oggetti</i> . . . . .	269
<b>11 Eccezioni</b> . . . . .	<b>271</b>
11.1 Introduzione alle Eccezioni . . . . .	271
11.2 Concetti di Raising e Handling . . . . .	271
11.3 Costrutto <b>try-except</b> . . . . .	272
11.3.1 Blocco <b>try</b> . . . . .	273
11.3.2 Blocco <b>except</b> . . . . .	273
11.3.3 Blocco <b>else</b> . . . . .	274
11.3.4 Blocco <b>finally</b> . . . . .	275
11.4 Pratiche comuni . . . . .	275
11.4.1 Specificità delle Eccezioni . . . . .	275
11.4.2 Utilizzo nei Cicli <b>while</b> . . . . .	275
11.4.3 Utilizzo nella lettura/scrittura dei Files . . . . .	276
11.5 Esempi . . . . .	277
R11 Riassunto <i>Eccezioni</i> . . . . .	282
<b>12 Files</b> . . . . .	<b>283</b>
12.1 Introduzione ai Files . . . . .	283
12.2 Directory . . . . .	283
12.3 Main . . . . .	286
12.4 <b>import</b> . . . . .	288
12.5 Files di testo . . . . .	289
12.6 Pickling . . . . .	291
R12 Riassunto <i>Files</i> . . . . .	294
<b>13 Alberi</b> . . . . .	<b>295</b>
13.1 Introduzione agli Alberi . . . . .	295
13.2 Anatomia e tratti distintivi . . . . .	296
13.3 Alberi Binari di Ricerca ("BST") . . . . .	297
13.4 Classe <b>Nodo</b> . . . . .	298
13.5 Operazioni sugli Alberi . . . . .	299
13.5.1 Visita Ricorsiva . . . . .	299
13.5.2 Ricerca . . . . .	300
13.5.3 Inserimento . . . . .	301
13.5.4 Rimozione . . . . .	302

13.5.5 Visita Iterativa . . . . .	303
13.6 Unire i concetti . . . . .	305
R13 Riassunto <i>Alberi</i> . . . . .	306
<b>14 Algoritmi</b> . . . . .	<b>307</b>
14.1 Introduzione agli Algoritmi . . . . .	307
14.2 Cos'è un Algoritmo? . . . . .	308
14.3 Concetto di Complessità . . . . .	308
14.3.1 Complessità Temporale . . . . .	308
14.3.2 Complessità Spaziale . . . . .	308
14.3.3 Notazione O-Grande . . . . .	309
14.4 Concetto di Costo . . . . .	309
14.4.1 Modello di Costo semplificato . . . . .	310
14.4.2 Gerarchia dei Costi . . . . .	310
14.4.3 Ottimizzazione . . . . .	311
14.5 Esempi di Algoritmi . . . . .	312
14.5.1 Algoritmi Ricorsivi . . . . .	313
14.5.2 Algoritmi di Ricerca . . . . .	314
14.5.3 Algoritmi di Ordinamento . . . . .	316
R14 Riassunto <i>Algoritmi</i> . . . . .	319
<b>A Moduli e Librerie utili</b> . . . . .	<b>321</b>
A.1 Introduzione . . . . .	321
A.2 Elenco . . . . .	322
A.2.1 Modulo <code>collections</code> . . . . .	322
A.2.2 Modulo <code>math</code> . . . . .	322
A.2.3 Libreria <code>numpy</code> . . . . .	322
A.2.4 Modulo <code>operator</code> . . . . .	322
A.2.5 Modulo <code>random</code> . . . . .	322
A.2.6 Modulo <code>time</code> . . . . .	322
<b>B Debugging e Testing</b> . . . . .	<b>325</b>
B.1 Introduzione . . . . .	325
B.2 Debugging . . . . .	326
B.3 Testing . . . . .	327
B.4 Eccezioni comuni e Gerarchia delle Eccezioni . . . . .	327
<b>C Indici delle tabelle e delle figure</b> . . . . .	<b>329</b>

# Capitolo 0

## Introduzione

### 0.1 Perché Python?



Figura 0.1: Un programma *Hello World!* in Python

Molto probabilmente questa è la prima domanda che viene in mente a chi si approccia per la prima volta a questo linguaggio di programmazione. Che tu sia un appassionato che sta leggendo questo libro per hobby o uno studente che deve superare un esame universitario... è importante sapere perché questo linguaggio è divenuto così popolare in (relativamente) pochi anni.

Non preoccuparti se, per adesso, alcuni concetti verranno trattati velocemente e magari potresti non comprendere tutto fin da subito: in questa sezione ci divertiremo un po' a fare le presentazioni del caso e a familiarizzare con il modo in cui verranno proposti i contenuti del libro. Più avanti vedremo nel dettaglio tutti gli aspetti principali del linguaggio, con approfondimenti ed esempi, nei vari capitoli dedicati. Torniamo a noi... .

**Python** (giunto ormai alla versione 3.13 nell’Ottobre del 2024) è un linguaggio di programmazione ad alto livello, vale a dire che è molto più simile al linguaggio umano che al linguaggio prettamente tecnico della macchina che deve eseguire le istruzioni. È spesso scelto come “punto di partenza” in quanto è caratterizzato da un’elevata semplicità, una curva di apprendimento veloce ed è utilissimo in molteplici campi d’applicazione fra cui l’analisi dei dati, lo scripting, fino alla programmazione videoludica.

La sua semplicità spesso fa sì che Python venga paragonato a dello “pseudocodice”, ossia una sintassi che non trova riscontro in nessun linguaggio di programmazione reale ma che, nella pratica, permette di capire fin da subito quali sono i passaggi logici di un programma, di una funzione o di un algoritmo.

Per convincerci di ciò, però, vediamo subito un esempio pratico.

Storicamente, il primo “programma” che un programmatore si accinge a creare è il cosiddetto **“Hello World!”** ossia il codice necessario a far comparire in console la scritta di saluto “Ciao Mondo!” in inglese. Il codice necessario in Python per fare questo è presente nella figura 0.1 in apertura a questo sottocapitolo... come si può vedere basta una sola riga di codice contenente una funzione (la funzione *print*) alla quale viene passata la stringa di parole **“Hello World!”** e il gioco è fatto! Se adesso paragoniamo questo primo programma al suo equivalente in altri linguaggi di programmazione odierni e di maggiore rilevanza ci accorgiamo di come effettivamente Python risulti essere la soluzione più semplice e anche intuitiva (figure 0.2 e 0.3).



```
● ○ ●

1 using System;
2 class App
3 {
4     static void Main() {
5         Console.WriteLine("Hello World!");
6     }
7 }
```

Figura 0.2: Un programma *Hello World!* in C-Sharp



```
● ○ ●

1 class HelloWorld
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Hello World!");
6     }
7 }
```

Figura 0.3: Un programma *Hello World!* in Java

Difatti, anche per stampare due semplici parole in console abbiamo bisogno di molti più “ingredienti” nel nostro codice. Generalmente (nei linguaggi degli esempi ossia C-Sharp e Java) si inizializzano delle *Classi* che contengono una funzione vuota (il *Main*) nella quale si passa l’istruzione che effettivamente va a effettuare la stampa. Questo non vuol dire che in Python non esistano le Classi (le vedremo molto più avanti quando parleremo di programmazione orientata agli Oggetti) ma tuttavia la sua flessibilità permette di non essere “vincolato” a esse per le operazioni più elementari. Un esempio che invece sembra essere identico è quello della figura 0.4:

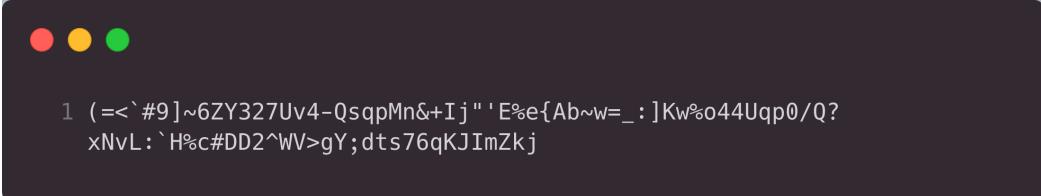


```
1 print("Hello World!")
```

Figura 0.4: Un programma *Hello World!* in Swift

Non confondiamoci (difatti sono stati usati colori diversi nelle immagini): il codice che vedi qui è scritto in Swift, non in Python. Sorge quindi spontanea la domanda: «E allora perché Python e non Swift?». Swift è il linguaggio di programmazione proprietario di Apple, perciò anche se è molto semplice non trova lo stesso livello di applicazione di Python, il quale invece è multipiattaforma. Il nostro obiettivo è imparare un linguaggio che, nella sua semplicità, permette di far funzionare i programmi su più dispositivi possibile e cioè raggiungere un numero più alto possibile di utenti.

Tuttavia, se ancora non dovessi essere convinto/a della semplicità del linguaggio Python ti suggerisco di andare ad approfondire su internet che cos’è un *Linguaggio di programmazione esoterico*, come il Malbolge ad esempio, la cui complessità ha fatto sì che il primo programma “Hello World!” vedesse la luce ben due anni dopo il suo rilascio (figura 0.5).



```
1 (= < #9 ] ~6ZY327Uv4-QsqpMn&+Ij '' E%e{Ab~w=_ : ]Kw%o44Uqp0/Q?
xNvL: `H%c#DD2^WV>gY; dts76qKJImZkj
```

Figura 0.5: Un programma *Hello World!* in Malbolge

Ovviamente, linguaggi come quest’ultimo sono creati a scopo puramente ludico e non trovano applicazione nel lavoro quotidiano, ma spero di averti convinto ad approcciarti a Python senza paure...!

## 0.2 Breve storia di Python

Python è stato creato da questo signore: Guido Van Rossum, informatico olandese, classe 1956. Laureatosi nel 1982 in Matematica e Informatica, prima di Python aveva già partecipato allo sviluppo di un altro linguaggio di programmazione (“ABC”) il quale ha influito molto nella creazione di Python.

Difatti, Python (il cui nome è un tributo al *Monty Python's Flying Circus*) nasce come un nuovo linguaggio contenente molteplici migliorie ad ABC e la sua creazione risale al 1989. Tuttavia, solo nel Febbraio 1991 il suo codice (la versione 0.9.0) fu caricato online da Van Rossum, in forma pubblica.

Dopo il 2000, Python è già giunto alla versione 2.0 la quale migliora notevolmente alcuni aspetti fondamentali come quello delle *List Comprehension* (un costrutto sintattico che permette di generare liste basandosi su altre liste).

Il linguaggio continua a evolversi negli anni fino al Dicembre 2008 quando avviene il passaggio da Python 2 a **Python 3.0**: una vera e propria rivoluzione. Difatti, Python 3 non era granché diverso dal suo predecessore, ma fu rivista la sintassi di molti elementi e semplificato il loro utilizzo a tal punto che il codice non era compatibile con la versione precedente.

Ad oggi, Python ha raggiunto la versione 3.13.



Figura 0.6: Guido Van Rossum  
Immagine di Daniel Stroud,  
sotto licenza CC BY-SA 4.0

## 0.3 Un salto avanti: 5 aspetti di Python

Per gustarci un piccolo “assaggio” di ciò che impareremo nei capitoli a venire, vediamo adesso qualche semplice esempio di codice ispirandoci liberamente alla panoramica che viene proposta sul sito ufficiale di Python, ossia <https://www.python.org/>, e vedremo perciò **5 aspetti** cardine che rendono Python... Python!

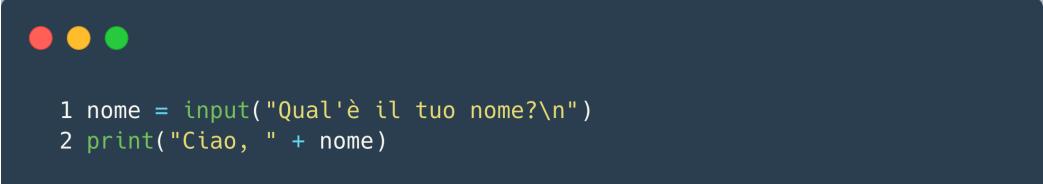
### 0.3.1 Facilità d’uso

Il primo aspetto che vogliamo approfondire con un ulteriore esempio (dopo l’*Hello World!* in figura 0.1) è quello relativo alla facilità d’uso di Python.

Vediamo perciò in breve come è possibile, con poche righe di codice aggiuntive, fare in modo che invece di salutare il mondo si possa salutare una qualsiasi persona che specificheremo a Python in modo dinamico volta per volta.

Siccome in questo primo capitolo ho promesso di non scendere troppo in dettagli, ve-

diamo subito il codice che serve per fare quanto detto. Ti propongo un gioco a questo punto: leggi il codice e prova a immaginare quali azioni svolge. Poi —solo dopo— leggi la spiegazione di seguito... se la tua ipotesi non si discosta di molto abbiamo dimostrato la facilità d'uso e soprattutto la **facile interpretazione** del linguaggio Python!



```

1 nome = input("Qual'è il tuo nome?\n")
2 print("Ciao, " + nome)

```

Figura 0.7: *Input e Print*

Non barare... pensaci un po'. Allora: il codice proposto nella figura 0.7 permette —come accennato in precedenza— di stampare in console un messaggio, come avevamo fatto per *Hello, World!*. Tuttavia, stavolta possiamo stampare al posto di **World** il nome della persona che sceglieremo noi. Difatti, la riga 1 del codice permette di “spiegare” a Python il nome che vogliamo stampare sullo schermo, e salviamo questa informazione in una Variabile che si chiama **nome**. Per farlo, usiamo una sintassi che è:

`nome_della_variabile = valore`

ossia attribuiamo (anzi, per essere precisi **assegnamo**) un certo valore (in questo caso il nome da stampare) alla variabile che ha un certo nome (in questo caso proprio **nome** perché deve contenere questa informazione). In più, siccome vogliamo assegnare una quantità variabile (ogni volta posso scrivere un nome diverso da salvare) ho bisogno di Input che permette di leggere dalla console il testo che inserirò rispondendo alla domanda “Qual’è il tuo nome?” appena essa comparirà sullo schermo. Il **\n** è un formattatore di testo... serve per far sì che il punto in cui dovrò scrivere la risposta sia a capo e non in linea.

Una volta che Python “sa” tutto, non resta che **stampare** la frase componendola grazie alla **concatenazione** (per mezzo del simbolo **+**) secondo la sintassi seguente:

`"testo" + variabile`

nel nostro caso **"Ciao, "** (con lo spazio) tra virgolette perché è un testo ed è fisso, al quale concateniamo il nome della variabile nella quale è salvato il nome da stampare (che guarda un po' si chiama proprio **nome** ma contiene l'*informazione* che abbiamo passato in **input**... e può essere quello che vogliamo).

Se eseguiamo il codice otteniamo in console qualcosa di simile a ciò che puoi vedere in figura 0.8:

```
Qual'è il tuo nome?  
Niccolò  
Ciao, Niccolò
```

Figura 0.8: Risultato in console dell'esempio *Input e Print*

Ricapitolando, il codice ha stabilito le seguenti azioni da svolgere:

1. Ha chiesto in **input** la risposta alla domanda “Qual’è il tuo nome?”.
2. Ha **assegnato** il valore ottenuto alla variabile **nome**.
3. Ha **stampato** una stringa di testo ottenuta **concatenando** “**Ciao,** ” e l’informazione contenuta in **nome**

... producendo così un risultato in console simile a quello mostrato in figura 0.8.

Per cercare di scriverlo il più possibile in modo simile a come potresti averlo pensato si potrebbe dire: «*Il nome è uguale all'input di “Qual’è il tuo nome?”.* Poi stampo “Ciao, ” più *il nome*». In effetti il codice ha permesso di fare proprio questo!

Ora chiediti: quanto ci sei andato vicino? Probabilmente non ti sei allontanato molto dal significato della frase scritta in corsivo, anche se magari potresti non aver colto fin da subito i termini più tecnici o altri aspetti più sottili... ma per questo c’è tempo!

### 0.3.2 Dichiarazioni di Flusso

Vediamo ora rapidamente le **Dichiarazioni di Flusso** (modo molto forzato di dare un nome italiano ai **Flow Statements**). I Flow Statements sono costrutti che il linguaggio Python interpreta in vari modi per creare Condizioni, Cicli, Intervalli... utilizzando le parole dell’inglese comunemente parlato.

Facciamo un esempio solo per la prima tipologia: quando si vuole imporre una condizione si utilizza «se» (in inglese *if*) e quindi «se» qualcosa è in un certo modo allora agiamo di conseguenza, ma «altrimenti» (= *else*) ci comportiamo diversamente. Questo è un tipico esempio di **condizione**, che in programmazione si chiama **If-Else** (= “Se/Altrimenti”) e permette di fare proprio quello che si presume faccia: se una condizione è rispettata succede qualcosa, *altrimenti* succede qualcos’altro. Il codice conterrà quindi dei **controlli** sulle condizioni e in base all’esito prenderà una strada piuttosto che un’altra.

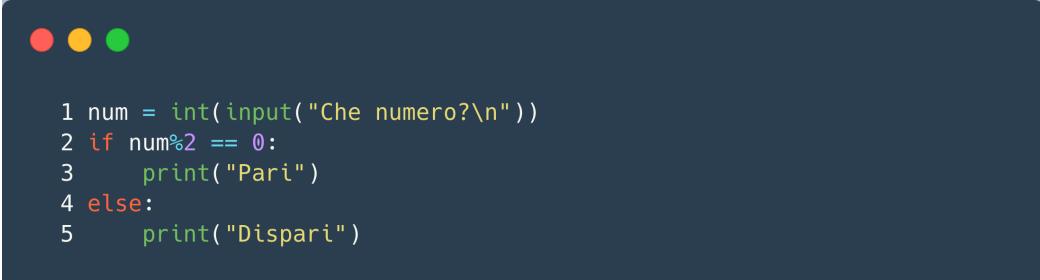
La cosa bella in tutto ciò è che non è difficile intuire come si scriverà il codice che permette di fare tutto ciò:

```
if
elif*
else
```

sono proprio le parole di cui abbiamo bisogno e che possiamo utilizzare nel nostro codice!

\* Nota: *Elif* permette di inserire  $n$  possibili altre condizioni quando ci troviamo in casi che non sono composti solo da “A oppure B”... ma lo capiremo meglio in seguito.

Facciamo quindi nuovamente il gioco di prima: leggiamo un piccolo codice Python e immaginiamo quali azioni sta descrivendo (figura 0.9). *Solo un piccolo indizio: il simbolo % che vedrai nel codice significa «Il resto della divisione», ad esempio:  $3\%2 = 1$ ,  $4\%2 = 0$ ... .*



```

1 num = int(input("Che numero?\n"))
2 if num%2 == 0:
3     print("Pari")
4 else:
5     print("Dispari")

```

Figura 0.9: Pari o Dispari

Il codice in figura svolge le seguenti azioni:

1. **Assegna** alla variabile **num** un **numero intero** (*int*) preso in **input**.
2. **Effettua un controllo** con un **costrutto condizionale If-Else**:
  - a. **Se** il numero diviso per 2 dà resto zero (definizione matematica di “Numero Pari”) **allora stampa "Pari"**.
  - b. **Altrimenti** (quindi se il numero diviso per 2 dà resto diverso da zero, ossia la definizione matematica di “Numero Dispari”) **allora stampa "Dispari"**.

Detto in parole:

«Il “num” è uguale al numero intero dell’input “Che numero?”. Se questo numero è pari stampa “Pari”, altrimenti stampa “Dispari”».

È molto importante soffermarsi non solo sulla scaletta numerata che dimostra i passaggi logici del codice, ma anche sulla scrittura più informale come questa in corsivo. Il perché è presto detto: quando dovrai affrontare esercizi (o magari quando un tuo cliente ti chiederà un software con determinati **requisiti**) la richiesta essenzialmente è esposta in modo non tecnico, eppure devi essere in grado di “tradurre” un bisogno

(il testo in corsivo) in una serie di passaggi logici (la scaletta) per scrivere il codice (la figura).

Non serve di certo farlo vedere, ma il codice della figura 0.9 porta a un risultato in console molto simile al seguente:

```
Che numero?  
3  
Dispari
```

```
Che numero?  
4  
Pari
```

Figura 0.10: Vari esiti in console dell'esempio *Pari o Dispari*

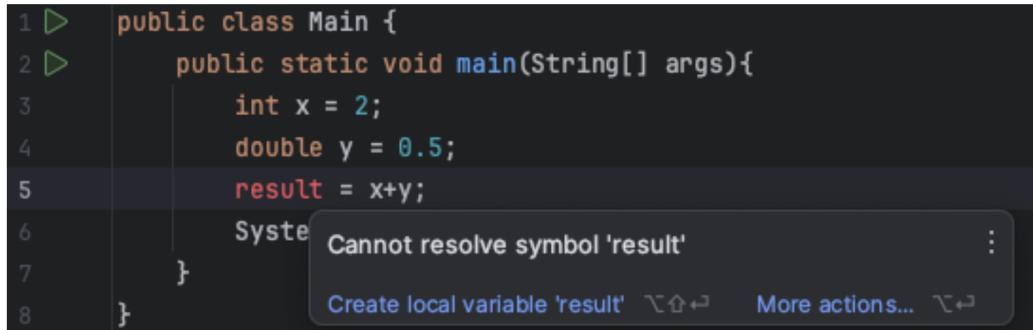
### 0.3.3 Interpretazione intuitiva

Quando parliamo di interpretazione intuitiva in Python non ci riferiamo alla già citata e sviscerata facilità di lettura del codice, ma al ragionamento logico che Python svolge per interpretare il codice che abbiamo scritto. Detto in parole semplici: non solo noi capiamo bene cosa dice il codice, ma anche Python capisce in modo logico e intuitivo quello che noi gli abbiamo chiesto di fare.

Si potrebbe pensare che questa è una banalità, ma non è così. Difatti, questa “intuitività” dipende dal livello di *tipizzazione* del linguaggio di programmazione che utilizziamo. La **tipizzazione** di un linguaggio è il livello di regole sintattiche imposte da esso, in modo che:

- Un linguaggio **fortemente tipizzato** impone molte regole rigide e impedisce usi incoerenti dei dati e delle variabili.
- Un linguaggio **debolmente tipizzato** ha meno regole e lavora con dati formalmente incoerenti in modo fluido e intuitivo.

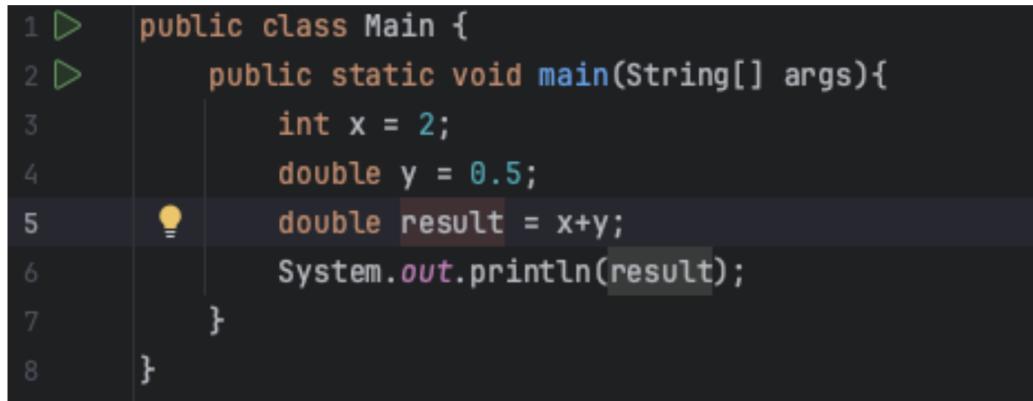
Un esempio del primo tipo è Java, nel quale di ogni variabile deve essere specificato il Tipo (un numero intero piuttosto che uno con la virgola); un esempio del secondo tipo è... proprio Python, che manipola le variabili in modo corretto senza che le informazioni sulla tipologia delle stesse vengano rese esplicite dal programmatore.



```
1 ▶  public class Main {  
2 ▶      public static void main(String[] args){  
3          int x = 2;  
4          double y = 0.5;  
5          result = x+y;  
6          System.out.println(result);  
7      }  
8  }
```

A screenshot of a Java code editor showing an error. The code is identical to Figure 0.11. A tooltip appears over the line 'result = x+y;' with the text 'Cannot resolve symbol 'result''. Below the tooltip are three buttons: 'Create local variable 'result'', 'More actions...', and a vertical ellipsis '...'. The code editor has a dark theme.

Figura 0.11: Somma di un numero intero e di un numero con virgola in Java (Errore)



```
1 ▶  public class Main {  
2 ▶      public static void main(String[] args){  
3          int x = 2;  
4          double y = 0.5;  
5          double result = x+y;  
6          System.out.println(result);  
7      }  
8  }
```

A screenshot of a Java code editor showing a corrected version of the code. The line 'double result = x+y;' now includes a small yellow lightbulb icon indicating a suggestion or code completion was used. The rest of the code remains the same as in Figure 0.11. The code editor has a dark theme.

Figura 0.12: Somma di un numero intero e di un numero con virgola in Java (Corretto)



```

1 x = 2
2 y = 0.5
3 result = x+y
4 print(result)

```

Figura 0.13: Somma di un numero intero e di un numero con virgola in Python

Come si può vedere dalle immagini precedenti (figure 0.11, 0.12, 0.13) in Java la tipizzazione fa sì che di tutte le variabili venga esplicitata la loro tipologia (*int* e *double* alle righe 3, 4 delle figure 0.11 e 0.12) e che la mancata esplicitazione del tipo anche della variabile che è data dalla somma delle due (ossia *result*, riga 5) è un errore che impedisce al codice di funzionare perché «non può essere risolto» (figura 0.11). Invece, se si esplicita la sua tipologia non ci sono problemi e il codice funziona correttamente (figura 0.12).

Invece, in Python, l'interpretazione intuitiva del codice fa sì che senza istruzioni esplicite il programma si comporti nel modo —appunto— più intuitivo possibile, senza farsi troppe domande. Difatti (figura 0.13) Python dal codice comprende *autonomamente* che:

1. La variabile *x* è un numero intero poiché le è stato assegnato un valore intero (pari a *2*).
2. La variabile *y* è un numero con virgola poiché le è stato assegnato un valore con la virgola (pari a *0.5*; la virgola in Python si esprime con un punto *.*).
3. La variabile *result* è un numero con la virgola poiché deriva dalla somma di un numero intero e di un numero con la virgola e perciò, per non perdere informazioni\*, *result* è un numero con la virgola (in Python si dice *float*, non *double*).

\* Nota: Se si volesse fare un arrotondamento si può fare il *Casting* della Variabile, ossia imporre che —nel nostro esempio— *result* sia un *int*, ma ovviamente stiamo perdendo informazioni. Si noti anche che, nonostante l'interpretazione intuitiva, a volte (per imporre condizioni o evitare errori di esecuzione, ad esempio) è necessario e sempre possibile specificare in modo esplicito la tipologia delle variabili anche in linguaggio Python.

Questa “intuitività” rende il codice più semplice ma, come dicono molti programmati amanti dei linguaggi più tipizzati come Java, anche «*più pericoloso*» perché soggetto a possibili errori, interpretazioni erronee o non intenzionali.

In sostanza, si dice che Python ha una scrittura **“Duck Typing”** ossia fa un ragionamento (da cui il nome) simile a: «Se ha le sembianze di una papera, cammina come

una papera, starnazza come una papera... *allora è una papera*», vale a dire che interpreta il codice in modo deduttivo a seconda dell'azione più logica che verrebbe in mente di fare.

È presto per dirlo ma ci proviamo lo stesso: come per le variabili, anche per le funzioni non è necessario specificare la tipologia del risultato che viene *ritornato* (= *returned*, prodotto in uscita) da esse... la riga 2 delle immagini 0.11 e 0.12, in Java, esprime una funzione *void* ossia “vuota”, “che non ritorna niente”... cosa che in Python non è necessario esplicitare.

### 0.3.4 Dati Composti

Oltre alla semplicità di definizione delle Variabili in modo implicito e logico, Python guadagna la possibilità di trattare i **Compound Data Types** (ossia i **“Dati Composti”**). In sostanza, si tratta di strutture che fungono da “contenitori” per altre Variabili il cui Tipo può essere diverso in base ai contesti di utilizzo. Sono esempi tra i più famosi (ma non esaustivi) le **Liste** e i **Dizionari**, che permettono di *immagazzinare* dati e *manipolarli* attraverso delle Funzioni *built-in* (vale a dire “pre-inserite”: sono funzioni che non devono essere definite dal programmatore ma che sono già predisposte nel linguaggio Python; le vedremo meglio in seguito).

Vediamo, come primo esempio, un semplice codice che riguarda le **Liste**... che cosa succede nel codice in figura 0.14 ?



```
1 L = [25]
2 a = 10
3 L.append(a)
4 print(L)
5 L.sort()
6 print(L)
```

Figura 0.14: Manipolazione di una lista

Il codice in figura mostra i seguenti passaggi logici:

1. **Definisce** una lista chiamata **L** (poiché le liste si indicano con le parentesi quadre **[]**) nella quale c'è un elemento: **25**.
2. **Assegna** alla variabile **a** un valore intero: **10**.
3. **Inserisce** la variabile **a** nella lista **L** grazie alla funzione built-in **append()**. La lista ora contiene anche **il valore** di **a**.
4. **Stampa** la lista con **print()**. Il primo risultato in console è:

```
[25, 10]
```

Figura 0.15: Esito in console dell'esempio *Manipolazione di una lista*, giunti alla riga 4

5. Ordina la lista in modo crescente con `sort()`.
6. **Stampa** la lista con `print()`. Il secondo risultato in console è:

```
[10, 25]
```

Figura 0.16: Esito in console dell'esempio *Manipolazione di una lista*, giunti alla riga 6

Come si può vedere, la lista permette di racchiudere dentro di sé degli elementi, che possono essere aggiunti da subito, inseriti successivamente, ordinati... e molto altro! Le liste sono una delle strutture più utilizzate nel linguaggio Python.

Senza spaventarci, cerchiamo di capire che significato potrebbe avere una riga di codice come la seguente:

```
● ● ●
```

```
1 M = [[1,2],[3,4]]
```

Figura 0.17: Una lista che contiene altre liste

Avremo modo di approfondire in seguito, ma quella che vediamo nella figura 0.17 è una lista che contiene come elementi altre liste più interne... anche se a vederla così non sembra (e stampandola in console non cambia) questa scrittura in codice Python rappresenta una **Matrice**, dove ogni elemento della lista più esterna sono le righe, mentre gli elementi delle sotto-liste sono i valori che, colonna per colonna, popolano la matrice fino alla sua larghezza massima.

In sostanza:

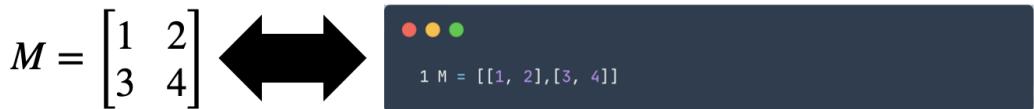


Figura 0.18: Significato algebrico della "lista di liste"

Infine, per completare la brevissima panoramica delle Strutture Dati Composte, vediamo un esempio di **Dizionario** in Python (figura 0.19):

```

1 D = {3:'albero', 4:'casa'}
2 print(D[3])

```

Figura 0.19: Stampare attraverso la Chiave

Il codice sopra esegue i seguenti passaggi:

1. **Definisce** un dizionario **D** (poiché i dizionari si indicano con le parentesi graffe **{}**) nel quale ci sono 2 elementi. Ogni elemento è composto da:
  - a. Una **Chiave** (le chiavi qui sono **3** per il primo elemento del dizionario e **4** per il secondo elemento).
  - b. Un **Valore** collegato a essa (i valori qui sono '**albero**' per il primo elemento del dizionario e '**casa**' per il secondo elemento).
2. **Stampa il valore** dell'elemento **la cui chiave** è **3**, ossia '**albero**'.

In console, cioè, avremo:

```
albero
```

Figura 0.20: Esito in console dell'esempio *Stampare attraverso la Chiave*

I dizionari, come si può intuire, sono utili quando è possibile stabilire una *relazione univoca* tra la Chiave e il suo Valore. Più in generale, ogni struttura dati composta ha le sue caratteristiche, i suoi pregi e i suoi difetti, ma soprattutto dei casi d'uso dove è preferibile rispetto a un'altra... e noi impareremo proprio a capire *quando* va usata una tipologia di struttura piuttosto che un'altra, non temere!

### 0.3.5 Definizione di Funzioni

L'ultima delle cinque caratteristiche che vediamo in introduzione (ma non per importanza, anzi!) è la definizione delle **Funzioni**.

Come abbiamo avuto modo di iniziare a vedere dai primi esempi affrontati, oltre al codice che definisce le variabili, i set di dati composti ecc... ci sono anche delle parti in cui vengono richiamati dei nomi che permettono di “fare” qualcosa (ad esempio `.append()` della figura 0.14 o il `print()` visto in svariate circostanze). Quando “invochiamo” un nome stiamo facendo una **Chiamata a Funzione**, ossia stiamo richiamando una porzione di codice che ha un nome ben preciso. Alcune funzioni sono *built-in*, come abbiamo già visto, ossia sappiamo a priori cosa fanno (`print()` stampa, `.append()` aggiunge un elemento a una lista, e così via...) e le chiamiamo quando servono. Ma la bellezza di Python ci riserva un’ulteriore sorpresa: possiamo scrivere le *nostre* funzioni, per poi richiamarle quando ci serviranno.

Perché si fa tutto ciò? Essenzialmente per tre motivi:

- a. Il codice è **estendibile** perché se ogni nuova funzione può essere scritta da una parte senza toccare il resto del codice possiamo aggiungere funzioni (e funzionalità!) in un secondo momento.
- b. Il codice è **facilmente rifattorizzabile**, ossia può essere modificato o sistemato in caso di errori con più facilità.
- c. Il codice **non è ridondante**, ossia non contiene ripetizioni: quando una certa parte di codice serve più volte, essa diventa una funzione da chiamare ogni qualvolta serva.

In pratica, le Funzioni sono come degli attrezzi sempre disponibili nella nostra cassetta e le utilizziamo per performare varie azioni, chiamandole dal nostro codice.

Supponiamo, ad esempio, di voler fare cinque somme a partire dai numeri immagazzinati in cinque variabili differenti: `a`, `b`, `c`, `d`, `e` e ogni volta stampare in console il risultato. Un codice senza l'utilizzo delle funzioni si può scrivere, simile a quello di figura 0.21.

Così, il codice è inutilmente lungo, altamente ripetitivo e può incorrere in molti più errori rispetto all'utilizzo di una funzione che permetta di fare l'azione ripetuta definendola, però, una volta sola. È logico: se c'è un errore nella funzione si corregge il codice della funzione una volta e così facendo abbiamo risolto il problema, ma in un codice dove l'azione è stata scritta in modo ridondante per cinque volte chissà dove potrebbe essere l'errore (o peggio ancora: potrebbero esserci cinque errori, anche di diversa natura!).

In questo caso è meglio definire allora una funzione che permetta di fare la somma di due termini (che “passeremo” alla funzione come **parametri**) e poi stampi il risultato chiamando `print()` ogni volta (= funzione nella funzione, ossia “**Funzioni nidificate**”).

Il codice allora diventa come quello mostrato in figura 0.22.

```
● ○ ●  
1 a = 1  
2 b = 2  
3 c = 3  
4 d = 4  
5 e = 5  
6  
7 somma1 = a+b  
8 print(somma1)  
9 somma2 = b+c  
10 print(somma2)  
11 somma3 = c+d  
12 print(somma3)  
13 somma4 = d+e  
14 print(somma4)  
15 somma5 = e+a  
16 print(somma5)
```

Figura 0.21: Cinque somme (senza definire funzioni)

```
● ○ ●  
1 def somma(addendo1, addendo2):  
2     risultato = addendo1 + addendo2  
3     print(risultato)  
4  
5  
6 a = 1  
7 b = 2  
8 c = 3  
9 d = 4  
10 e = 5  
11  
12 somma(a, b)  
13 somma(b, c)  
14 somma(c, d)  
15 somma(d, e)  
16 somma(e, a)
```

Figura 0.22: Cinque somme (definendo funzioni)

Magari da questo esempio non è apprezzabile la riduzione di lunghezza del codice (non abbiamo “guadagnato” alcuna riga...) ma la potenza di quello che abbiamo scritto è subito intuitibile: i nostri “attrezzi” (le funzioni) sono un’astrazione il più possibile generica di un’azione che prima aveva degli attori stabiliti in modo preciso. Inoltre, guadagnamo la possibilità di dare dei nomi esplicativi ai parametri (capiremo meglio in seguito perché e come) permettendo anche una migliore comprensione di "cosa fa" il codice.

Un discorso simile si potrebbe fare per le **Classi** che definiscono gli Oggetti... ma lasciamo questa trattazione a tempo debito.

# Capitolo 1

## Fondamentali di Python

### 1.1 Commenti

Iniziamo il nostro viaggio vero e proprio alla scoperta di Python soffermandoci su un aspetto che, a una prima analisi, può sembrare non troppo rilevante: i commenti. I **Commenti** sono porzioni di testo che non hanno valore di codice.

Tuttavia, commentare il codice è una buona pratica che consente di essere più organizzati, efficienti e soprattutto migliora notevolmente la lettura del codice stesso attraverso le spiegazioni, cosicché se riprendi in mano un progetto dopo molto tempo o se devi leggere un codice scritto da terzi sarà più facile capirne i meccanismi e le condizioni di funzionamento.

Inoltre, approfondiamo fin da subito come si commenta il codice in Python anche per una questione prettamente didattica: d'ora in poi, nelle immagini che corredano ogni capitolo di questo libro, potresti trovare dei commenti all'interno del codice per comprendere meglio alcuni passaggi più delicati.

In Python, i commenti sfruttano le seguenti sintassi:

```
# commento in linea
```

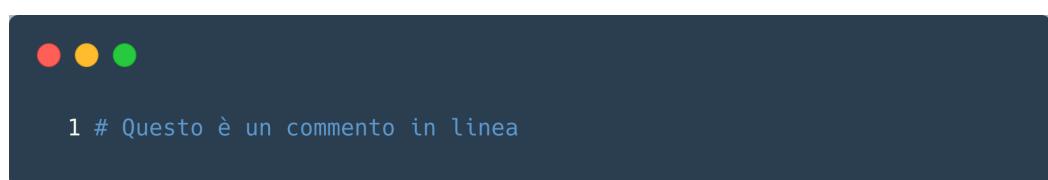
A screenshot of a dark-themed code editor window. In the top-left corner, there are three small circular icons: red, yellow, and green. The main area contains a single line of Python code: "1 # Questo è un commento in linea". The number "1" is likely a line number from the editor's interface.

Figura 1.1: Commento in linea

oppure:

```
"""commento
multilinea"""
```



```
1 """Questo è un commento
2 che si estende su più righe"""

```

Figura 1.2: Commento multilinea

La prima tipologia di commenti può trovarsi sia su righe vuote che quindi conterranno solo il commento stesso, sia su righe di codice. Tieni presente che tutto ciò che si trova dopo il simbolo `#` è considerato commento e perciò non ha alcun valore di programmazione.

La seconda tipologia di commenti, invece, può trovarsi solo su righe che non contengono altre porzioni di codice da eseguire.

Mettendo tutto insieme (figura 1.3):



```
1 a = 3 # Inizializzo una variabile intera con valore "3"
2 ris = 2 + a
3 """Il risultato è sempre il valore della variabile "a"
4 a cui viene sommato il valore 2"""

```

Figura 1.3: Esempi di commenti

Alla riga 1 abbiamo un'assegnazione di variabile e poi una parte commentata in linea, preceduta dal simbolo di commento `#`.

La riga 2 assegna alla variabile `ris` il valore pari alla somma di 2 e del valore contenuto nella variabile `a`. Questa informazione è spiegata nel codice per mezzo di un commento multilinea che si estende nelle righe 3-4.

## 1.2 Indentazione

Vediamo adesso un altro aspetto fondamentale di Python: l'**Indentazione**. “Indentare” il codice significa fare in modo che esso sia strutturato come una sorta di scala

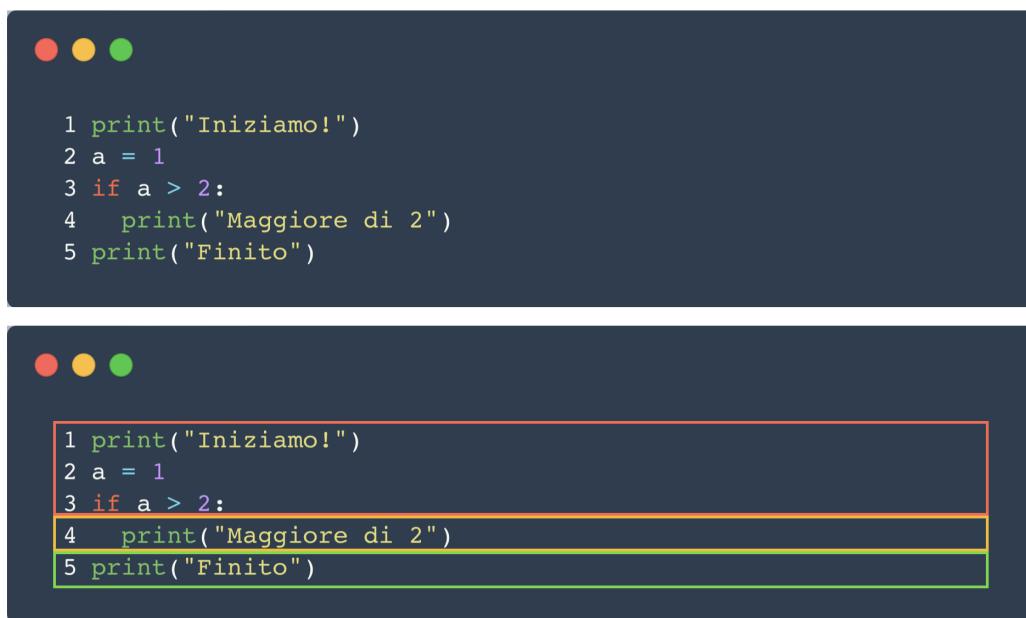
(come dei “denti” più o meno pronunciati) all’inizio di ciascuna riga.

In molti linguaggi si fa uso dell’indentazione (e di altri stratagemmi come l’uso delle parentesi graffe per i blocchi di codice in C-Sharp o Java), solitamente con i seguenti scopi in mente:

- **Funzionale:** Una diversa impostazione del codice può avere un significato diverso per l’interprete (lo strumento integrato che elabora il codice).
- **Supporto alla lettura:** Un codice graficamente ben strutturato (=*Prettified*) è più comodo e ordinato da leggere.

In Python, l’uso dell’Indentazione racchiude in sé entrambi gli scopi appena citati: non solo permette al codice di essere letto facilmente, ma ha una sua valenza per interpretarne il significato!

Sbagliare indentazione, cioè, porta a errori che potrebbero non far funzionare un programma, perché l’interprete non riesce a capire il blocco sul quale stiamo lavorando. Prendiamo un esempio come il seguente:



The screenshot shows two terminal windows. The top window displays a Python script with standard white-on-black syntax highlighting. The bottom window shows the same script, but the code block starting at line 3 is highlighted with three nested levels of colored boxes: a red box for the entire block, an orange box for the first level of indentation, and a green box for the second level of indentation. The first four lines of the script are outside these boxes.

```

1 print("Iniziamo!")
2 a = 1
3 if a > 2:
4     print("Maggiore di 2")
5 print("Finito")

```

Figura 1.4: Indentazione, con livelli diversi evidenziati

Il codice in **rosso** viene sempre eseguito, ed è al livello base di indentazione. Siccome (come vedremo più avanti) alla riga 3 c’è una condizione, in base a se essa è vera oppure no viene eseguita o meno una parte di codice (lo studieremo meglio più avanti, ma per ora basta intuirne il funzionamento «a grandi linee»). Questa parte è distinta grazie a un livello di indentazione, ed è la riga in **arancione**. Poi, il codice in **verde** è tornato al livello di indentazione base: anche questo, quindi, indipendentemente dalle

condizioni o da altri fattori viene sempre eseguito.

Adesso che abbiamo capito *quando* indentare il codice, dobbiamo capire *come* indentarlo. Per indentare il codice si devono seguire delle buone pratiche, raccolte nel **PEP** (=“**Python Enhancement Proposal**”) ossia una serie di documenti che indicano norme e linee guida più o meno dettagliate e stringenti che si dovrebbero seguire per scrivere codice Python il più possibile correttamente (in gergo si dice “*Pythonico*”). Nello specifico, le regole riguardanti l’indentazione rientrano nel documento “**PEP-8**”, il quale stabilisce che:

- I. Per indentare, si devono **usare 4 spazi**.
- II. Si dovrebbe **evitare l’uso di TAB per indentare**.
- III. **Non mischiare spazi e tabulazioni**.

Come detto, il PEP è un insieme di buone pratiche che andrebbero seguite. Tuttavia, Python è in grado di adattarsi anche a sistemi di indentazione leggermente diversi da quelli proposti dal PEP-8: ad esempio è possibile usare comunque la tabulazione per evidenziare i vari livelli di indentazione, oppure è possibile usare 3 spazi o piuttosto 6 spazi anziché i canonici 4. Sono pratiche sconsigliate, ma non vietate; finché lo stile si mantiene coerente per tutto il codice, Python riesce a distinguere i blocchi indentati senza particolari problemi.

Inoltre, quando lavorando coi dati avremo a che fare con le strutture dati composte o con le funzioni, è buona norma usare spazi e tabulazioni per **allineare** gli elementi e i parametri, rispettivamente. Questo migliorerà la lettura del codice (vedi figura 1.5).



```
 1 L = [ 2     1, 2, 3, 4, 5, 3     6, 7, 8, 9, 10 4 ]
```

Figura 1.5: Lista con elementi allineati

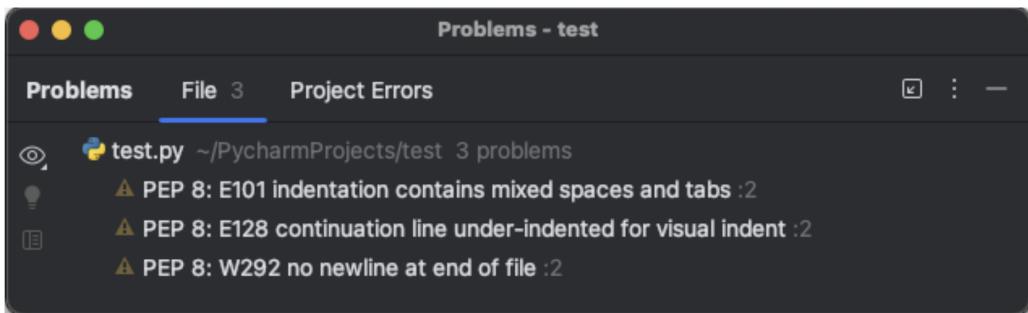


Figura 1.6: Alcuni avvisi di mancato rispetto delle regole del PEP-8 nell’IDE PyCharm

La figura 1.6 dimostra come questi “errori” in realtà sono segnalati come “Avvertimenti di basso livello” che sono perlopiù formalità e non errori che compromettono il funzionamento del programma.

## 1.3 Funzione `print()`

Abbiamo già visto nelle pagine precedenti la funzione `print()`, ma adesso la formalizziamo.

La **Funzione `print()`** permette di stampare in console o altrove i dati (con eventualmente dei separatori) che le vengono passati come parametri (siccome vedremo più avanti cosa sono le funzioni, per ora limitiamoci a dire «gli elementi tra parentesi tonde»). I dati che passiamo alla funzione `print()` sono molteplici e possono essere anche di tipologie diverse. Questo non è (e a volte è, invece) un problema:

- **Non è un problema** perché la funzione `print()` si occupa di trasformare questi dati sempre in formato testuale (stringhe).
- **È un problema** perché se non teniamo presente questa trasformazione potremmo incorrere in errori di visualizzazione (più avanti affronteremo questo discorso parlando di Classi...).

Inoltre, è bene ricordare che oltre a passare dati è possibile passare alla funzione `print()` anche degli *statements* ossia delle “affermazioni” (operazioni da eseguire) che portano comunque a un dato finale. Ad esempio, per passare il valore 9 si possono usare varie strade:

- **Come valore puro** scrivendo `9` come parametro della funzione.
- **Come variabile** scrivendo `a` come parametro della funzione, supponendo che sia stato assegnato alla variabile `a` il valore 9 in precedenza.
- **Come statement** scrivendo `12-3` come parametro della funzione.
- **Come ibrido delle precedenti** scrivendo, ad esempio, `12-b` come parametro della funzione, supponendo che sia stato assegnato alla variabile `b` il valore 3.

Tutti questi esempi trovano riscontro nella figura 1.7 qui sotto.

```
● ● ●  
1 a = 9  
2 b = 3  
3  
4 print(9) # Come valore puro  
5 print(a) # Come variabile  
6 print(12-3) # Come statement  
7 print(12-b) # Come ibrido
```

Figura 1.7: Possibili modi di usare `print()` per ottenere uno stesso risultato

Si tenga presente che questa lista è solo a titolo esemplificativo. Difatti, come abbiamo già visto, nulla vieta di stampare un'intera lista (figure 0.14, 0.15, 0.16) o solo un valore appartenente a essa, oppure un dizionario (figure 0.19, 0.20) e potremmo continuare con gli esempi... .

Infine, formalizziamo anche il concetto della concatenazione e del modo in cui la funzione `print()` stampa i nostri dati a schermo.

Di base, `print()` accetta un numero variabile di parametri in ingresso e —come già detto— li converte in stringhe (testo). I parametri si separano tra di loro con una virgola , e la funzione `print()` stampa gli elementi trasformati in stringhe, separandoli tra di loro con una virgola e concludendo la riga con un隐式的 \n, ossia il **formattatore** di testo che permette di andare “a capo”.

È possibile, però, **concatenare** gli elementi per ottenere un *output* (= risultato) più complesso e strutturato, attraverso l’uso del carattere di concatenazione +, secondo la sintassi già vista in precedenza.

Infine, c’è un modo molto più elegante, comodo e intuitivo per avere lo stesso risultato della concatenazione: il **print formattato** (chiamato anche **printf**). In questo modo, possiamo stampare una stringa inserendo anche variabili al suo interno proprio come nella concatenazione, ma senza dover concatenare niente, bensì specificando il nome della variabile tra parentesi graffe senza interrompere la stringa virgolettata. Per poter fare ciò basta aggiungere la lettera f prima del virgolettato in questione. Per capire meglio, formalizziamo la sintassi e vediamo un esempio.

```
print(f"testo con {variabile}")
```