Laboratorio di Programmazione I

Con il linguaggio Python

ARTIFICIAL INTELLIGENCE
& DATA ANALYTICS

UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

Laura Nenzi, Giulio Caravagna, Stefano Alberto Russo

Corso di Laurea Triennale in Intelligenza Artificiale e Data Analytics (AIDA)

Dipartimento di Matematica, Informatica e Geoscienze
Università di Trieste

Dispense trascritte dalle lezioni grazie ai contributi cumulativi di: Gianluca Guglielmo (stesura iniziale), Michele Rispoli, Pietro Morichetti, Nicolas Solomita, Andrea Mecchina, Elena Buscaroli, Federico Pigozzi, Lucrezia Valeriani e Valentina Blasone.

Versione 1.0.1 (26 febbraio 2026)

[1] La licenza dice grossomodo che, a condizione di citare l'autore e di utilizzare la stessa tipologia di licenza, questi materiali possono essere liberamente copiati, modificati, usati per creare opere derivate e ridistribuiti.

Indice

Preambolo. Questa dispensa costituisce il materiale didattico per il corso di *Laboratorio di Programmazione Modulo 1*, relativo all'anno accademico di riferimento. Il corso è un modulo del corso *Introduzione alla Programmazione e Laboratorio*.

A seconda delle annualità e del corso di laurea, il corso prevede 9, 12 o 15 CFU e si articola in due o tre moduli: uno dedicato alla programmazione (oggetto della dispensa del prof. Caravagna) e uno (o due) al laboratorio (oggetto della presente dispensa).

Questo modulo di laboratorio si concentra interamente su `Python`. Il primo modulo è propedeutico a questo quindi viene assunto che lo studente sia già familiare con i costrutti e gli operatori di base di un linguaggio di programmazione. Verranno inoltre affrontati i fondamenti della programmazione a oggetti e alcuni concetti di programmazione funzionale, entrambi trattati in `Python`.

Gli obiettivi formativi e i materiali didattici possono variare di anno in anno: non sempre l'intera dispensa sarà utilizzata, e talvolta potranno essere forniti contenuti integrativi. In linea generale, il modulo di programmazione mira a introdurre i concetti essenziali del *computational thinking*, con particolare attenzione ai principi di ragionamento e progettazione che restano validi indipendentemente dal linguaggio utilizzato, mentre il laboratorio approfondisce l'uso di `Python`, affrontando tematiche complementari e fornendo competenze pratiche che si affiancano alla solida base concettuale costruita nel modulo di programmazione, utile per affrontare con agilità diversi linguaggi.

L'intelligenza artificiale (IA) scrive codice, ma non capisce il vostro problema. L'IA produce testo plausibile anche se non sapete formalizzare il problema, ed il codice plausibile è pericoloso: compila, ma fa la cosa sbagliata. Senza esercizio non sviluppate il "debug mentale", cioè la capacità di leggere un algoritmo e chiedervi: ha senso? copre i casi limite? cosa succede se l'input è sporco? Programmare non è scrivere righe, è pensare in modo strutturato. Bisogna decidere cosa fare prima di decidere come farlo. Gli esercizi allenano la decomposizione del problema, potete vederla come una ginnastica cognitiva. La verità è che oggi programmare è ancora più importante, non meno. Perché ora dovete saper valutare, correggere, integrare codice generato automaticamente. È un ruolo più maturo: da scrittori a ingegneri del ragionamento. Se vogliamo dirla in modo quasi filosofico: l'IA è un acceleratore di pensiero. Ma se non c'è pensiero, accelera il vuoto. E il vuoto, quando lo acceleri, resta vuoto. Per cui vi consiglio vivamente di provare a fare gli esercizi usando l'IA in maniera limitata.

Libri di testo. Durante la prima lezione e nella pagina del corso sono forniti dei riferimenti testuali aggiornati ogni anno. Gli studenti che desiderino consultare testi di riferimento più strutturati possono rivolgersi al docente per ricevere indicazioni sui titoli più appropriati.

Guida alla dispensa. Questa dispensa si "evolve" di anno in anno, e pertanto a volte potrebbe contenere errori. Dubbi, chiarimenti ed eventuali errori possono essere segnalati via mail a `lnenzi@units.it`. È prevista una forma di ricompensa per la segnalazione di errori (non banali).

Nel testo le spiegazioni sono colloquiali, sacrificando il rigore per la chiarezza espositiva per fornire un accesso semplice e diretto ai concetti base. Esempi di codice sono colorati e discussi. In generale una variabile indicata come x, y, etc. denota una quantità matematica, mentre `x` o `y` una variabile scritta nel programma, quindi codice. Spesso le due coincidono e la notazione non è necessariamente consistente tra le varie sezioni.

PART

I

Strumenti di lavoro

Per affrontare il *Laboratorio di Programmazione in Python* è necessario usare alcuni strumenti, che vediamo brevemente.

File manager Il *File Manager* è un software che consente di gestire e organizzare i file e le cartelle presenti in un sistema operativo. In pratica, è quello strumento che permette all'utente di vedere, modificare, copiare ed eliminare file e cartelle. Esempi sono Windows Explorer (Esplora file) su Windows, il Finder su MacOS e Nautilus o Dolphin sui sistemi Linux.

Shell La *shell* permette di eseguire programmi e navigare il file system attraverso una linea di comando, *Command-Line Interface (CLI)*, ovvero di interagire direttamente con la macchina. Nei sistemi Windows le shell a disposizione sono il *Prompt dei Comandi (CMD)* e la *PowerShell*, mentre nei sistemi Unix si ha *BASH* (in genere accessibile tramite l'applicazione *Terminale*).

Editor del codice L'*Editor del Codice* può essere un qualsiasi programma per la scrittura del testo *semplice*, ovvero che non aggiunga della formattazione come il grassetto, corsivo, paragrafi etc. È possibile usare un editor che interpreti il linguaggio utilizzato per colorare le diverse sintassi e facilitarne la lettura. Per il corso è necessario impostare l'editor in modo che utilizzi 4 spazi al posto dei tab, che serviranno per indentare il codice. Python segue la convenzione PEP 8 (le linee guida ufficiali di stile) che raccomanda 4 spazi per livello di indentazione, un tab può essere interpretato in modo diverso da sistemi diversi.

IDE L'*Integrated Development Environment* è un editor che supporta lo sviluppatore attraverso l'integrazione con sistemi di debugging, File Mangaer, Shell, talvolta Git e svariate altre agevolazioni, le quali rendono la programmazione più scorrevole. Gli IDE più diffusi supportano diversi linguaggi e possono adattare i propri suggerimenti a seconda della sintassi utilizzata. Tra i più utilizzati troviamo Sublime e Code::Blocks.

Visual Studio Code *Visual Studio Code* (VS Code) è un editor di codice estensibile sviluppato da Microsoft. Non è propriamente un IDE tradizionale, bensì un editor leggero che può essere arricchito tramite estensioni per supportare numerosi linguaggi di programmazione. Integra un terminale,

strumenti di debugging e il supporto al versionamento tramite *Git*. La sua architettura modulare consente di aggiungere funzionalità come completamento automatico, analisi statica del codice e gestione di ambienti virtuali. Grazie al sistema di estensioni e al *Language Server Protocol*, VS Code può assumere il comportamento di un vero e proprio ambiente di sviluppo integrato.

Git Il software per il versionamento *Git* è uno strumento per tenere traccia delle modifiche fatte al codice, può essere sincronizzato con dei collaboratori ed è decentralizzato. Le *Repository* (repo) sono le cartelle dei progetti e possono contenere sia codice che file di altro tipo. L'operazione base è il *commit*, che crea un "salvataggio" del codice in Git permettendo di avere dei checkpoint, univocamente identificati da un cosiddetto *hash*. È possibile trovare qui una guida su Git scritta da Michele Rispoli.

GitHub *GitHub* è una piattaforma online che ospita repository *Git* e facilita la collaborazione tra sviluppatori. Permette di condividere progetti, gestire modifiche tramite *pull request*, discutere problemi attraverso le *issue* e integrare sistemi di automazione per test e distribuzione del software. Oltre all'hosting del codice, offre strumenti per la documentazione, la revisione collaborativa e la gestione di progetti.

GitHub Codespaces *GitHub Codespaces* è un servizio cloud che consente di creare ambienti di sviluppo remoti direttamente collegati a una repository GitHub. L'ambiente viene eseguito su una macchina virtuale nel cloud e può essere utilizzato tramite browser oppure collegandosi con *Visual Studio Code*. In questo modo lo sviluppo non dipende dalle risorse locali del computer, ma da un ambiente configurato e riproducibile, accessibile da qualsiasi dispositivo.

Ambiente virtuale Un *ambiente virtuale* è uno spazio isolato che contiene una specifica versione di Python e le librerie necessarie a un determinato progetto. Ogni progetto può avere il proprio ambiente, indipendente dagli altri e dal sistema operativo. Questo isolamento è fondamentale per evitare conflitti tra versioni diverse delle stesse librerie o tra progetti che richiedono configurazioni differenti.

Nel corso utilizzeremo *Miniconda* per creare e gestire ambienti virtuali. In particolare, verrà creato un ambiente dedicato al laboratorio, all'interno del quale verranno installate solo le librerie necessarie. Quando un ambiente è attivo, il terminale mostra il suo nome tra parentesi all'inizio della riga di comando: ciò indica che i comandi Python e le installazioni di librerie verranno eseguiti all'interno di quell'ambiente.

È buona pratica non lavorare mai nell'ambiente `base`, ma creare un ambiente specifico per ciascun progetto.

Pseudocodice Un ultimo strumento concettuale è costituito dal cosiddetto *pseudocodice*. Prima di scrivere il codice vero e proprio, in qualsiasi linguaggio di programmazione, è utile scriverne una bozza, anche a carta e penna, focalizzandosi non sul come, ma sul *cosa* fare. Non c'è uno standard con cui scrivere questa bozza: lo pseducodice è del tutto inventato, usando parole anche in linguaggio naturale (italiane o inglesi) che permettano di trasporre il problema che si cerca di risolvere in logica di programmazione imperativa ma senza, appunto, preoccuparsi di dettagli come la sintassi, i tipi dati etc. per non interrompere il flusso di ragionamento logico.

PART

II

Introduzione a Python

Cos'è Python *Python* è un linguaggio di programmazione interpretato di alto livello, cioè "distante" dal linguaggio macchina. Durante il corso useremo la versione 3, e in particolare ≥ 3.6. Python nasce nel 1991 come linguaggio di programmazione ad oggetti. Il suo utilizzo è in costante crescita, grazie anche alla sua semplicità e intuitività. È un linguaggio *interpretato*, quindi non ha bisogno di compilazione e può essere anche usato in modalità interattiva dall'interprete Python, molto utile ogniqualvolta si voglia testare delle cose in rapidità.

Python è anche il linguaggio "de facto" standard per la Data Science, grazie ad un enorme ecosistema di strumenti per il calcolo numerico, scientifico e statistico.

In questo corso si assume che si abbia già familiarità con i costrutti e gli operatori base di un linguaggio di programmazione, come ad esempio:

- Assegnazione di variabili e stampa a schermo (`=, print`)
- Operatori condizionali (`if, else`)
- Operatori aritmetici (`+,-,*,` etc.)
- Operatori logici (`and, or`)
- Operatori di confronto (`==, <, >,` etc.)
- Cicli (`for, while,` etc.)

Nei prossimi capitoli verrano comunque presentati questi costrutti e operatori nel linguaggio Python.

Ciao, mondo! Come ogni buona trattazione di un linguaggio di programmazione, iniziamo con un classico "Ciao mondo".

Funzione 1 (Print)**.** La funzione `print` permette di stampare una stringa a schermo. La sintassi è la seguente:

```
print('Ciao, Mondo!') # Stampa "Ciao, Mondo!"
```

È anche possibile inserire un valore personalizzato nella stringa usando la sintassi `.format`:

```
x = 25
print('Ho {} anni.'.format(x)) # Stampa 'Ho 25 anni."
```

1 Tipi di dato ed operatori

In Python non è necessario definire il tipo di dato delle variabili durante la dichiarazione. Infatti, Python deduce automaticamente il tipo durante l'inizializzazione grazie al valore che le viene assegnato. E' infatti un linguaggio *non tipizzato*. Al contrario dei linguaggi *tipizzati* come ad esempio il C, in Python una variabile può cambiare tipo dati in continuazione, basta assegnarle un nuovo valore di tipo diverso.

In Python sono presenti i seguenti tipi dati principali (ma ce ne sono anche altri più complessi, come liste e dizionari, che vedremo in seguito):

```
mia_var = 1       # variabile di tipo intero (int)
mia_var = 1.1     # variabile di tipo floating point (float)
mia_var = 'ciao'  # variabile di tipo stringa (str)
mia_var = True    # variabile di tipo booleano (bool)
my_var = [1, 2, 3]           # variabile di tipo lista (list)
my_var = (1, 2, 3)           # variabile di tipo tupla (tuple)
my_var = {1, 2, 3}           # variabile di tipo insieme (set)
my_var = {"a": 1, "b": 2}.   # variabile di tipo dizionario (dict)
mia_var = None    # "niente" si rappresenta con il None
```

Nel codice sopra, i cancelletti indicano un commento: ogni carattere successivo ad un cancelletto viene ignorato dall'interprete Python, fino a nuova riga.

In Python il tipo dati `None` viene usato, oltre al valore nullo, anche per semplicemente indicare qualcosa di non rilevante, come ad esempio una variabile che è necessario inizializzare per motivi sintattici ma che non è ancora stata utilizzata (p.es. `somma_totale=None`).

Il tipo dato di una variabile può essere controllato usando la funzione `type`. La chiamata `type(var)` ritorna il tipo della variabile.

Esempio 1. Esempi di utilizzo della funzione `type`:

```
x = 10
print(type(x))      # <class 'int'>

y = 3.14
print(type(y))      # <class 'float'>

z = 'ciao'
print(type(z))      # <class 'str'>
```

Spesso è necessario convertire un valore da un tipo ad un altro. Questa operazione prende il nome di *casting* e si effettua richiamando il tipo di destinazione

come fosse una funzione.

Esempio 2. Esempi di casting tra tipi diversi:

```
a = '25'
b = int(a)          # da stringa a intero

c = 3.7
d = int(c)          # da float a intero (troncamento)

e = 10
f = float(e)        # da intero a float

g = str(e)          # da intero a stringa
```

Se il valore non è compatibile con il tipo di destinazione, viene generato un errore.

1.1 Operatori

Python dispone di diversi operatori built-in (cioè predefiniti e disponibili senza dover importare niente) per effettuare operazioni su valori e variabili. per effettuare operazioni su valori e variabili. Di seguito vediamo gli operatori aritmetici, di confronto e logici ma ce ne sono anche altri che vedremo in seguito.

Sintassi 1 (Operatori aritmetici)**.** Gli operatori aritmetici effettuano le semplici operazioni matematiche indicate dal simbolo.

Operatore	Nome	Esempio
`+`	`Addizione`	`x+y`
`-`	`Sottrazione`	`x-y`
`*`	`Moltiplicazione`	`x*y`
`/`	`Divisione`	`x/y`
`%`	`Modulo` [a]	`x%y`
`**`	`Esponenziale`	`x**y`

[a] L'operazione modulo ritorna il resto della divisione x/y.

Sintassi 2 (Operatori di confronto)**.** Gli operatori di confronto ritornano `True` o `False` a seconda che la condizione indicata sia rispettata o no. Sono usati soprattutto nei costrutti `if`.

Operatore	Nome	Esempio
==	Uguale	x==y
!=	Diverso	x!=y
>	Maggiore	x>y
<	Minore	x<y
>=	Maggiore o uguale	x>=y
<=	Minore o uguale	x<=y

Sintassi 3 (Operatori logici)**.** Gli operatori logici permettono di collegare due (o più) condizioni attraverso le logiche `and`, `or` e `not`.

Operatore	Descrizione	Esempio
and	True se entrambe True	x<5 and x<10
or	True se almeno una True	x<5 or x<4
not	Inverte il risultato	not(x<5 and x<10)

1.2 Stringhe

Una *stringa* (`str`) è una sequenza *immutabile* di caratteri. In Python le stringhe si possono delimitare sia con apici singoli `'...'` sia con doppi apici `"..."` (scegline uno stile e usalo in modo coerente). Essendo iterabili, supportano gli operatori di appartenenza, l'accesso per indice e lo *slicing*, esattamente come le liste che vedrete subito dopo.

Esempio 3. Alcune stringhe in Python:

```
s1 = 'Ciao'
s2 = "Mondo"
s3 = 'Python 3.12'

print(s1 + ' ' + s2)    # concatenazione: 'Ciao Mondo'
print(len(s1))          # lunghezza: 4
```

Sintassi 4 (Accesso e slicing su stringhe)**.** Come per le liste, posso accedere ai caratteri per posizione e fare slicing:

```
s = 'Programmazione'

s[0]    # 'P'  (primo carattere)
s[-1]   # 'e'  (ultimo carattere)
s[0:7]  # 'Program'
s[7:]   # 'mazione'
```

Sintassi 5 (Immutabilità)**.** Le stringhe sono immutabili: non posso modificare un carattere "in place".

```
s = 'ciao'
# s[0] = 'C'      # ERRORE: le stringhe sono immutabili
s = 'C' + s[1:]   # OK: creo una nuova stringa
```

Sintassi 6 (Formattazione di stringhe)**.** Per inserire valori nelle stringhe si può usare `.format()` oppure le *f-string* (consigliate per leggibilità):

```
x = 25
print('Ho {} anni.'.format(x))
print(f'Ho {x} anni.')
```

Input dell'utente In Python esiste una funzione built-in chiamata `input` che sospende l'esecuzione del programma e attende che l'utente inserisca un valore da tastiera. Il valore inserito viene sempre ritornato come *stringa* (`str`), quindi se serve un numero è necessario effettuare un casting.

Funzione 2 (Input)**.** La funzione `input` legge una riga da tastiera. La sintassi base è la seguente:

```
testo = input()          # attende l'input dell'utente
print(testo)             # stampa ciò che l'utente ha scritto
```

Prima dell'inserimento dei dati è buona norma visualizzare un messaggio (*prompt*) che informi l'utente di cosa deve inserire:

```
nome = input('Come ti chiami?\n')
print('Ciao, {}!'.format(nome))
```

Poiché `input` ritorna sempre una stringa, per ottenere valori numerici bisogna convertire il risultato:

```
eta = int(input('Quanti anni hai?\n'))  # casting da str a int
print('Tra un anno avrai {} anni'.format(eta + 1))
```

1.3 Liste

La *lista* è un tipo dati built-in che permette di rappresentare una sequenza mutabile di oggetti. Gli oggetti possono essere di tipo eterogeneo, anche se questo tipo di lista non serve in quasi nessun caso. Una lista si definisce usando le parentesi quadre.

Esempio 4. Alcune liste in Python:

```
mia_lista = [1,2,3] # Lista di numeri
mia_lista = ['Marco', 'Irene', 'Paolo'] # Lista di stringhe
```

```
mia_lista = [15, 32, 'ambo'] # Lista eterogenea
```

Le liste ci introducono agli operatori di appartenenza, all'accesso per posizione e allo *slicing*.

Sintassi 7 (Operatori di appartenenza)**.** Gli operatori di appartenenza servono a controllare se un valore si trova in una lista, stringa o tupla[a].

Operatore	Descrizione	Esempio
`in`	`True se x è nella lista y`	`x in y`
`not in`	`Inverte risultato di in`	`x not in y`

[a]Una tupla è una lista *immutabile* definita usando le parentesi tonde.

Sintassi 8 (Accesso per posizione)**.** Posso accedere agli elementi di una lista per posizione usando la notazione con parentesi quadre:

```
mia_lista[0] # Primo elemnto
mia_lista[1] # Secondo elemento
mia_lista[-1] # Ultimo  elemento
```

Con la stessa sintassi si può accedere agli elementi delle stringhe, che si possono immaginare come una serie (lista) di caratteri.

Sintassi 9 (Slicing)**.** È possibile effettuare lo *slicing* delle liste, ovvero tagliarne una fetta, usando la seguente sintassi[a]:

```
mia_stringa[0:50]  # Dal 1° al 50° carattere
mia_stringa[30:50] # Dal 30° al 50° carattere
mia_stringa[0:-1]  # Dal 1° al penultimo carattere
```

Si può applicare lo slicing anche alle stringhe, che sono immaginabili come una serie di caratteri.

[a]In questa sintassi, l'estremo sinistro dell'intervallo è compreso, l'estremo destro è escluso.

1.4 Dizionari

Il *dizionario* è un tipo dati built-in che permette di associare un valore `value` ad una chiave `key`. Si definisce usando le parentesi graffe e separando i `value` e le `key` con i due punti. Si accede al valore associato ad una chiave utilizzando le parentesi quadre.

Esempio 5. Nel seguente esempio, alcuni dizionari in Python:

```
mio_diz = {'Trieste': 34100, 'Padova': 35100} # Diz. di numeri
mio_diz = {'Trieste': 'TS', 'Padova': 'PD'} # Diz. di stringhe
```

```
3
4 sigla_ts = mio_diz['Trieste'] # A sigla_ts sarà assegnato il
5                               # valore legato alla chiave
6                               # 'Trieste' nel dizionario
                                ↪  my_dict
```

Le chiavi devono essere tipi dati che possono essere univocamente identificati ("hashabili"). Non approfondiamo ora questo concetto, come prima approssimazione diciamo che possiamo usare come chiave tutti i tipi dati "base" non composti, ovvero interi, stringhe, floating point ma non le liste. I valori, invece, possono essere di un tipo dati qualsiasi, incluso liste e dizionari: possiamo infatti tranquillamente creare un dizionario di liste o un dizionario di altri dizionari, a patto che le chiavi siano impostate come descritto sopra. Come per le liste è possibile usare gli operatori di appartenenza per controllare se una chiave è contenuta nel dizionario. Non è possibile utilizzare gli operatori di appartenenza per i valori.

1.5 Tuple

La *tupla* è un tipo dati built-in che rappresenta una sequenza *immutabile* di elementi. Si definisce usando le parentesi tonde. A differenza delle liste, una volta creata non può essere modificata.

Esempio 6. Alcune tuple in Python:

```
1 mia_tupla = (1, 2, 3)
2 mia_tupla = ('Marco', 'Irene', 'Paolo')
3 mia_tupla = (10, 'ciao', 3.14)
```

Come per le liste, è possibile accedere agli elementi per posizione e usare lo slicing:

```
1 mia_tupla[0]    # Primo elemento
2 mia_tupla[-1]   # Ultimo elemento
3 mia_tupla[0:2]  # Slice
```

Le tuple vengono spesso utilizzate per rappresentare dati che non devono essere modificati, oppure per restituire più valori da una funzione.

```
1 def esempio():
2     return (1, 2)
3
4 a, b = esempio()  # unpacking della tupla
```

1.6 Insiemi (Set)

Il *set* è un tipo dati built-in che rappresenta un insieme di elementi *non ordinati* e *senza duplicati*. Si definisce usando le parentesi graffe oppure la funzione `set()`.

Esempio 7. Esempi di insiemi:

```
mio_set = {1, 2, 3}
mio_set = {1, 2, 2, 3}  # diventa {1, 2, 3}
```

I set sono utili per verificare appartenenza e per effettuare operazioni insiemistiche:

```
A = {1, 2, 3}
B = {3, 4, 5}

A | B   # unione
A & B   # intersezione
A - B   # differenza
```

Non è possibile accedere agli elementi di un set per indice, poiché non sono ordinati.

2 Indentazione, istruzioni condizionali e cicli

2.1 Indentazione

Nella sintassi di C e C++ un blocco di codice contenente le istruzioni di un costrutto `if` o di un ciclo è delimitato da due parentesi graffe. In Python invece, l'inizio e la fine di un blocco di codice è definito dall'*indentazione* del blocco stesso.

Indentare significa "far rientrare" alcune parti del listato di codice, in genere tramite degli spazi. È una pratica comune in tutti i linguaggi di programmazione, ma in Python è strettamente legata al funzionamento del linguaggio stesso ed è da considerarsi parte della sintassi. Di norma, vengono usati 4 spazi o un carattere `tab` (l'importante è essere coerenti in tutto il codice), ma gli spazi sono da preferirsi. Vanno indentanti anche i commenti allo stesso livello del blocco di codice cui si riferiscono, per semplificare la lettura.

2.2 Istruzioni condizionali

Un'*istruzione condizionale* (il classico `if-else`) permette di inserire una condizione, che deve essere verificata per poter eseguire le istruzioni contenute nelle successive righe di codice.

Nota bene: la riga contenente l'istruzione condizionale termina con due punti, che la separano dal blocco di codice che contiene le istruzioni da eseguire nel caso in cui la condizione sia verificata e che deve essere correttamente indentato.

Esempio 8. Un semplice costrutto `if`:

```
if (mia_var > tua_var):
    # Indentazione per segnalare che queste
    # istruzioni fanno parte del blocco if
    print("La mia var è più grande della tua")
    if (mia_var-tua_var) <= 1:
        # Ulteriore indentazione per il secondo if
        print("...Non così tanto...")
```

Note bene: in Python è stata introdotta la parola chiave `elif` per comprimere la scrittura di `else if`.

Esempio 9. Un costrutto `if-elif-else`:

```
if (mia_var > tua_var):
    print("La mia var è più grande della tua")
    if (mia_var-tua_var) <= 1:
        print("...Non così tanto...")
    elif (mia_var-tua_var) <= 5:
        print("...Abbastanza")
    else:
        # L'ultima condizione ha bisogno solo dell'else, che
        # gestisce tutti i casi che non sono stati considerati
        print("...Di molto")
```

2.3 Cicli

Anche in Python abbiamo i classici cicli `for` e `while`. Anche in questo caso la prima riga termina con i due punti ed è seguita da un blocco indentato, contenente le istruzioni da eseguire per ogni iterazione del ciclo.

Esempio 10. Un ciclo `for`:

```
for i in range(10):
    # Indentazione per identificare il blocco del for
    print(i)
```

Esempio 11. Un ciclo `while`:

```
i = 0
while i<10:
    # Indentazione per identificare il blocco del while
```

```
    print(i)
    i = i + 1
```

Funzione 3 (Range)**.** La funzione built-in `range`, introdotta nel `for` dell'esempio precedente, crea "al volo" una lista di numeri su cui poi si può effettuare un ciclo, nel seguente modo[a]:

```
range(10)       # Lista di numeri da 0 a 9
range(3, 10)    # Lista di numeri da 3 a 9
range(3, 10, 2) # Lista di numeri da 3 a 9
                # con passo 2: [3, 5, 7, 9]
```

[a] Anche in questo caso si ha l'estremo destro escluso.

Esempio 12. Un modo più comodo per ciclare sui tipi dati iterabili (come liste, stringhe e tuple), senza usare `range`, è il seguente:

```
for elemento in mia_lista:
    # ogni elemento è preso in modo ordinato dalla lista
    ↪ mylist
    print(elemento)
```

Questo modo di ciclare è detto "pythonico", perché sfrutta appieno le potenzialità del linguaggio. Essere "pythonici" vuol dire proprio questo: scrivere codice che sfrutta quello che ci mette a disposizione il linguaggio evitando di fare le cose a mano quando non serve.

Funzione 4 (Enumerate)**.** La funzione built-in `enumerate` ritorna sia l'elemento che il suo indice nella lista, sotto forma di *tupla*:

```
for (i, elemento) in enumerate(mia_lista):
    print("Posizione {}: elemento {}".format(i, elemento))
```

3 Funzioni

Le *funzioni* in Python sono dichiarate usando la parola chiave `def` e indicando eventuali paramenti di input tra parentesi. Dopo i due punti che terminano la prima riga della definizione, si passa poi a un blocco indentato, come già visto sia per le istruzioni condizionali che per i cicli. La parola chiave `return`, alla fine del blocco di istruzioni, precede i valori da ritornare al chiamante. Non è necessario, al contrario di C e C++, indicare il tipo dati degli input, ne tanto mento quello dei valori da ritornare.

Esempio 13. Funzione con due argomenti e nessun ritorno. Se eseguo il codice verrà stampato a schermo "Argomenti: Pippo e Pluto":

```
def mia_funzione(argomento1, argomento2):
    print(f"Argomenti: {argomento1} e {argomento2}")

print(mia_funzione('Pippo', 'Pluto'))  # Chiamata della
↪ funzione
```

Esempio 14. Funzione con un argomento e un ritorno. Se eseguo il codice verrà stampato a schermo "Risultato: 16":

```
def eleva_al_quadrato(numero):
    return numero * numero

risultato = eleva_al_quadrato(4)
print(f"Risultato: {risultato}")
```

Python dispone di molte funzioni *built-in*, ovvero disponibili senza dover importare nessuna libreria. È possibile consultarne un elenco qui.

3.1 Scope

Si può pensare ad uno *scope* come ad una regione di codice. In Python si stabilisce una gerarchia tra diversi scope. Una variabile, infatti, è disponibile solo all'interno dello scope in cui è stata creata e in tutti gli scope di livello più basso nella gerarchia. È importante saper prevedere quali variabili apparterranno a quali scope, in modo da non far confusione con il loro utilizzo. Python segue la regola *LEGB*, che indica la seguente gerarchia:

- Local: scope locale relativo a ciascuna classe e funzione (ogni classe e funzione ha il suo local scope). È il livello più basso nella gerarchia.
- Enclosed: scope che racchiude la regione di codice attuale, come ad esempio la funzione esterna nel caso di due funzioni annidate, o più semplicemente anche il "corpo" esterno del programma in cui sto definendo una funzione.
- Global: scope più esterno, contiene i nomi visibili in tutto il codice. È sconsigliato l'utilizzo diretto di variabili globali all'interno di funzioni, in quanto si può perdere in interpretabilità e creare bug indesiderati.
- Built-in: scope che contiene i nomi riservati ai moduli built-in di Python. È importante non sovrascrivere i nomi built-in per evitare malfunzionamenti nel codice [1].

[1] Ad esempio, mai chiamare una variabile `sum`, perchè in Python esiste una funzione built-in `sum` che verrrebbe in questo modo sovrascritta.

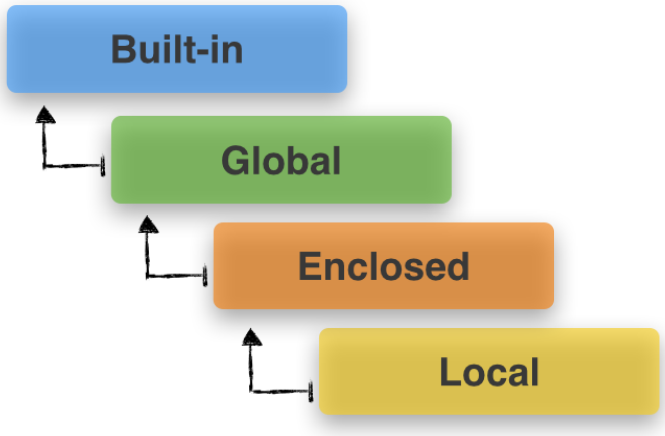

Figura 1. Struttura LEGB

È buona norma, in una funzione, utilizzare variabili locali. Si noti che un argomento in input è considerato locale dalla funzione, in quanto il suo scope è la funzione stessa a cui viene passato. Per salvare un risultato, è prassi passarlo ad una variabile dello scope superiore usando `return`, mentre è sconsigliato modificare variabili dello scope superiore direttamente dall'interno della funzione. Una funzione dovrebbe essere vista come un elemento isolato, che lavora solo sulle variabili locali e comunica con lo scope esterno solo tramite l'input (i suoi argomenti) e l'output (ciò che segue il return).

3.2 Moduli

Un *modulo* è un file che funge da libreria di funzioni. Le funzioni presenti in un modulo possono essere *importate*, usando la parola chiave `import`, ossia `import modulo`, nel file principale. Di norma, per utilizzare una funzione di un determinato modulo, è necessario precedere il nome della funzione con il nome del modulo importato. Alternativamente, si può importare direttamente una singola funzione con la sintassi `from modulo import funzione`. In quest'ultimo caso, il nome della funzione non deve essere preceduto da quello del modulo. I moduli non verranno usati durante il corso.

Esempio 15. Due modalità diverse per usare la stessa funzione `sqrt` del modulo `math`:

```
import math
var = math.sqrt(4) # var = 2

from math import sqrt
var = sqrt(4)      # var = 2
```

4 Essere pythonici

Quando si scrive codice Python, in particolare se si arriva da altri linguaggi di programmazione, si potrebbe essere portati ad adottare una filosofia "classica" nella scrittura del codice.

Per esempio, per iterare su di una lista si potrebbe essere portati ad usare gli indici:

Esempio 16. Iterazione su di una lista usando un indice di supporto

```
mia_lista=['a','b','c']
i=0
while (i < len(mia_lista)):
    print(mia_lista[i])
    i = i + 1
```

Il codice qui sopra riportato funziona, ma non è *pythonico*. Essere Pythonici vuol dire sfruttare appieno le potenzialità del linguaggio, che porta ad una sintassi più semplice e un codice più compatto ed efficiente.

Con Python si può infatti quasi rasentare lo pseudocodice, e programmare in modo quasi discorsivo. Vediamo come possiamo iterare su di una lista in un modo più pythonico, riprendendo quanto già esposto nelle sezioni precedenti:

Esempio 17. Iterazione su di una lista in modo pythonico

```
mia_lista=['a','b','c']
for elemento in mia_lista:
    print(elemento)
```

Ci sono molti trucchetti per scrivere codice in modalità più pythonica, e in generale laddove ci si ritrova ad usare degli indici o del codice poco facilmente comprensibile, questo è segno che c'è qualcosa di ampiamente migliorabile.

È una capacità che si fa propria con gli anni, ma che è bene inquadrare fin da subito, poichè facilita notevolmente nella programmazione con Python. Su https://coady.github.io si può trovare un blog con vari spunti su come essere più pythonici (in inglese).