



Python

Gestione del flusso di esecuzione

Argomenti della lezione

- Algebra booleana
- Indentazione del codice
- Passi condizionali
- Stringhe
- Liste
- Tuple
- Set, frozenset e dizionari
- Cicli iterativi
- Esercitazioni

Algebra booleana

L'*algebra di Boole*, o algebra booleana, è un sistema sviluppato nel 1800 dal matematico inglese George Boole per “catturare” la logica del ragionamento umano (anche i computer si sono poi basati su questo sistema).

Gli unici valori possibili sono **vero** e **falso** (in inglese true e false).

AND

Il risultato dell'operatore e (**and**) è vero se e solo se entrambi i suoi operandi sono veri.

Per esempio, considerando la logica proposizionale (cioè delle proposizioni o frasi):

“vado al cinema solo se piove e ho i soldi” significa che vado al cinema solo se entrambi gli eventi si verificano.

OR

Il risultato dell'operatore o (or) è vero se almeno uno dei suoi operandi è vero. Per esempio “metto il cappotto se fa freddo o se nevica” significa che metto il cappotto se si verifica almeno uno dei due eventi, o anche se si verificano tutti e due. L'operatore non (not) non fa altro che invertire il valore del suo operando. Per esempio: “mi chiamo Mario” diventa “non mi chiamo Mario”.

Quindi se è vero che “mi chiamo Mario”, allora è falso che “non mi chiamo Mario” e viceversa.

Siamo in una logica binaria, cioè a due soli valori. Le operazioni logiche sono facilmente rappresentate dalla seguente “tabella di verità” (si dice “di verità” proprio perché quello che interessa di un’espressione è solo il fatto che sia vera o falsa), che riassume tutti i casi possibili, cioè tutte le combinazioni possibili dei valori degli operandi. A e B rappresentano due espressioni o due frasi che possono essere combinate assieme con gli operatori binari (cioè che si applicano a due operandi) and e or. Un valore A può essere negato con l’operatore unario (cioè che si applica a un solo operando) not.

Di seguito è riportata la tabella di verità:

A	B	A and B	A or B	not A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Vediamo alcuni esempi d'uso dei valori e degli operatori booleani:

```
>>> condizione = True
```

```
>>> type(condizione)
```

```
<class 'bool'>
```

```
>>> 2 > 2 or 2 == 2 # → False or True → True
```

```
True
```

```
>>> 2 > 2 or 2 != 2 # → False or False → False
```

```
False
```

```
>>> not condizione # → not True → False
```

```
False
```

```
>>> not 2 > 3 and condizione # → not False and True → True and True → True
```

```
True
```

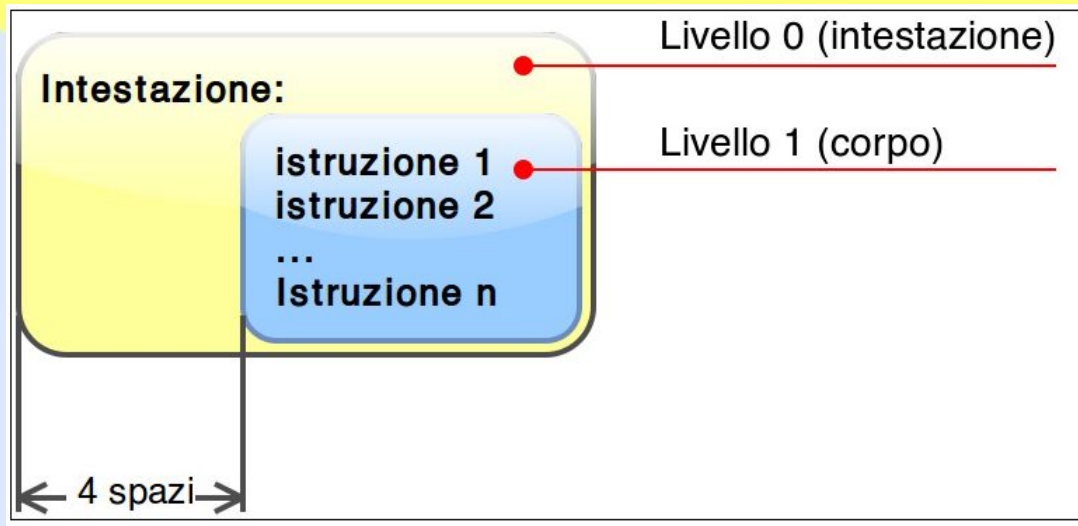
Indentazione del codice

Nel codice visto fino ad ora non abbiamo avuto la necessità di creare “blocchi” di codice.

Un blocco di codice è una sequenza di istruzioni raggruppate assieme che vengono trattate come se fossero un'unica macro-istruzione. In Python le istruzioni di uno stesso blocco hanno la stessa ***indentazione***, cioè sono precedute dallo stesso numero di spazi dall'inizio della riga.

Ogni qual volta indenti un blocco di codice stabilisci un nuovo livello di esecuzione dipendente dalla riga che precede il blocco e a cui quest'ultimo risulta associato.

Nell'immagine seguente il blocco giallo contiene al suo interno un altro blocco, azzurro, costituito da una o più istruzioni.



Quando in IDLE, alla fine dell'intestazione, inserisci i due punti (:) e premi invio, il cursore di inserimento si sposta automaticamente di 4 spazi, proprio perché Python si aspetta il corpo associato all'intestazione.

Questo vale per il costrutto if, che vedremo a breve, e anche per cicli, funzioni, costrutto try-except e classi.

```
File Edit Format Run Options Window Help
"""indentazione codice"""
value = 15
if value > 15:
    print("Maggiore di 15")

    if value > 30:
        print("maggiore di 30")

        if value > 45:
            print("maggiore di 45")
else:
    if value == 15:
        print("Uguale a 15")
    else:
        print("Minore di 15")
```

Passi condizionali

Il codice scritto finora procede in modo sequenziale, leggendo ed eseguendo ogni istruzione dall'alto al basso. La struttura di controllo sequenza elabora le istruzioni passo passo e rappresenta il comportamento di default di un programma.

In effetti, con solo questa struttura di controllo quello che possiamo fare è abbastanza limitato in quanto il programma procede come su un binario unico: non ci sono varianti e quindi si comporta sempre allo stesso modo. Anche nell'esempio precedente, se è pur vero che l'input da parte dell'utente modifica il risultato, il procedimento operativo è sempre lo stesso.

Le istruzioni condizionali vengono utilizzate quando vogliamo eseguire un blocco di codice solo nel caso in cui una condizione sia vera o falsa.

La struttura completa del costrutto **if** prevede, oltre alla clausola iniziale **if**, due clausole opzionali, che quindi ci possono essere oppure no. La clausola **elif** (la parola è la contrazione di else if, ovvero “altrimenti se”) e può comparire zero o più volte e l'altra è la clausola **else** (se è presente, ce n'è una sola).

```
if <condizione 0>:  
    <blocco if>  
elif <condizione 1>:  
    <blocco elif 1>  
elif <condizione 2>:  
    <blocco elif 2>  
...  
else  
    <blocco else>
```

È molto importante capire che nel codice precedente quando una condizione è verificata viene eseguito solo il blocco di codice associato senza che le altre condizioni vengano prese in considerazione. Inoltre la valutazione delle condizioni è sequenziale.

A volte è conveniente ricorrere all'operatore ternario (si chiama così perché opera su 3 operandi) che esprime in forma concisa una situazione **if-else**. Nell'esempio seguente vediamo come assegnare alla variabile maggiore il valore massimo tra le variabili a e b, con il modo **if-else** classico:

```
if a >= b:  
    maggiore = a  
else  
    maggiore = b
```

Un modo equivalente con l'operatore ternario è questo:

```
maggiore = a if a >= b else b
```

1. Proviamo a scrivere un'applicazione che, dato un numero in input, determini se si tratta di un numero pari o dispari (allo scopo possiamo utilizzare l'operatore modulo % che restituisce il resto della divisione intera)

L'esecuzione del programma inserendo il testo 9 da tastiera è la seguente:

Numero: 9

Il numero 9 è dispari

2. Crea un programma che stabilisce qual è il maggiore tra tre numeri (naturalmente è vietato l'utilizzo della funzione **max()**)

Ecco un'esecuzione del programma:

Primo numero: 3

Secondo numero: 9

Terzo numero: -12

Il massimo è 9

3. Scrivere il codice per verificare se tre numeri possono essere le misure dei lati di un triangolo (la verifica consiste nel determinare che siano tutti positivi e che ciascuno sia inferiore alla somma degli altri due).
4. Scrivi un programma che chiede all'utente di inserire un numero e stampa se è positivo, zero o negativo.
5. Scrivi un programma che controlla se un anno inserito dall'utente è bisestile oppure no. Sono bisestili gli anni multipli di 4 a eccezione degli anni secolari (100, 200,...) che sono bisestili solo ogni 400 anni (400, 800,...). Per esempio, gli anni 1968 e 2000 sono bisestili, mentre il 1969, il 2100 e il 2200 non lo sono.

Stringhe

Una stringa è un dato composto da una sequenza di caratteri. per dichiarare una stringa è sufficiente assegnare ad una nuova variabile un testo racchiuso tra virgolette: è possibile racchiudere il suo valore indifferentemente tra apici (carattere ') o doppi apici (carattere ").

Questo permette di superare facilmente il problema dell'utilizzo di questi caratteri all'interno della stringa stessa. È anche possibile usare il carattere di escape \ prima di ' o ".

Le stringhe sono un tipo particolare di sequenze e perciò supportano tutte le operazioni comuni alle sequenze.

Le stringhe, così come le liste o le tuple, sono un tipo particolare di sequenze e perciò supportano tutte le operazioni comuni alle sequenze.

Indexing

Per fare riferimento a un carattere si usa un indice numerico da 0 a $n - 1$, dove n è la lunghezza della stringa. Un altro modo per fare riferimento alle posizioni nella stringa è quello di utilizzare indici negativi. In questo modo, si va all'indietro dalla fine all'inizio della stringa: -1 indica l'ultimo carattere, -2 il penultimo, eccetera.

```
>>> s = "Ciao"  # Definisci una stringa di 4 caratteri (indice da 13)
```

```
>>> len(s)      # Ritorna la lunghezza (length) della stringa
```

```
4
```

```
>>> s[0] # Accedi e recupera il primo carattere
```

```
'C'
```

```
>>> s[-1] # Accedi e recupera l'ultimo carattere
```

```
'o'
```

```
>>> s[0] = "x" # Una stringa è immutabile, non è possibile modificare un suo elemento
```

Slicing

Puoi ricavare una parte di una stringa, o sottostringa, con un'operazione chiamata slicing (to slice, affettare). La forma completa di questo operatore, simile a quella della funzione range(), prevede da 0 a 3 parametri ed è [start:stop:step]. Nel caso generale Python ritorna la sottostringa da start a stop - 1 con passo step, che di default vale 1.

```
>>> s = "Ciao, mondo!"
```

```
>>> s[:3]      # Se start viene omesso la sottostringa parte dall'inizio
```

```
'Cia'
```

```
>>> s[6:]      # Se stop viene omesso la sottostringa arriva fino alla fine
```

```
'mondo!'
```

```
>>> s[2:4]     # Dal terzo al quarto carattere: indici 2 e 3 (= 4 - 1)
```

```
'ao'
```

```
>>> s[:]       # Dall'inizio alla fine
```

```
'Ciao, mondo!'
```

```
>>> s[::2]     # Dall'inizio alla fine con passo 2: indici 0, 2, 4, 6, 8, 10
```

```
'Ca,mno'
```

Contenimento

Gli operatori **in** e **not in** possono essere usati per verificare se un elemento fa parte di una sequenza o no. Nel caso delle stringhe, è anche possibile verificare se una sottostringa è contenuta in una stringa:

```
>>> name = Danilo
```

```
>>> 'D' in name # controlla se il carattere 'D' è contenuto nella stringa name
```

```
True
```

```
>>> 'x' in name # il carattere 'x' non è in name, quindi ritorna False
```

```
False
```

```
>>> 'x' not in name # "not in" esegue l'operazione inversa
```

```
True
```

```
>>> 'Da' in s # controlla se la sottostringa 'Da' è contenuto nella stringa name
```

```
True
```

```
>>> 'Da' in s # il controllo è case-sensitive, quindi ritorna False
```

```
False
```

Concatenamento e ripetizione

È possibile usare l'operatore **+** per concatenare sequenze, e ***** per ripetere sequenze:

```
>>> "Hello, " + "World!" # Concatena tra loro due stringhe  
'Hello, World!'
```

```
>>> "Hello, " * 5 # Ripeti la stringa "Hello, " per 5 volte  
'Hello, Hello, Hello, Hello, Hello, '
```

Concatenamento e ripetizione

È possibile usare l'operatore **+** per concatenare sequenze, e ***** per ripetere sequenze:

```
>>> "Hello, " + "World!" # Concatena tra loro due stringhe  
'Hello, World!'
```

```
>>> "Hello, " * 5 # Ripeti la stringa "Hello, " per 5 volte  
'Hello, Hello, Hello, Hello, Hello, '
```

Formattazione

Spesso c'è la necessità di creare una stringa a partire da valori di vario tipo: stringhe, numeri, eccetera. Un modo per farlo prevede l'uso della funzione `str()` per la conversione in stringa dei valori, che non sono già di tipo stringa, e la successiva operazione di concatenazione. Ecco un esempio:

```
>>> "Raffaele ha "+ str(4) +" anni"  
'Raffaele ha 4 anni'
```

Ci sono anche altre modalità più sofisticate di formattazione delle stringhe che permettono di inserire valori di tipo diverso all'interno di una stringa. L'idea base è quella di aggiungere dei segnaposto con un formato specifico all'interno della stringa e di elencare i valori che andranno a sostituirli in fase di esecuzione.

Una soluzione di formattazione prevede l'uso del metodo **`format()`** con segnaposto che possono essere di tre tipi: anonimi, indicizzati o con nome.

Nel caso di segnaposto anonimi (`{}`) la sintassi da utilizzare è la seguente:

```
>>> text = "{} è nato nel mese di {} del {}"  
>>> name = "Leonardo"  
>>> month = "Ottobre"  
>>> year = 2020  
>>> text.format(name, month, year)  
"Leonardo è nato nel mese di Ottobre del 2020"
```

In questo caso ogni valore passato alla funzione **format()** sostituirà ordinatamente il corrispondente segnaposto, è possibile slegarsi da questa corrispondenza utilizzando dei segnaposto indicizzati (`{n}`)

```
>>> n = 2  
>>> n2 = n * n  
>>> "{1} è il risultato di {0} x {0}".format(n, n2) # 3 segnaposto e 2 valori  
'9 è il risultato di 3 x 3'
```

Un altro modo per formattare una stringa fa uso di segnaposto per specifiche tipologie di valori. Questa metodologia di formattazione è ancora utilizzabile ma è stata sostituita dalla funzione **format()** vista in precedenza che risulta essere più flessibile.

Per completezza aggiungiamo un esempio di formattazione della stringhe con la vecchia notazione:

```
>>> "Lorenzo ha %d anni ed è alto %.2f m." % (9, 1.3825)  
'Lorenzo ha 9 anni ed è alto 1.38 m.'
```

Nell'esempio precedente possiamo notare, all'interno della stringa, i segnaposto **%d** e **%.2f** che identificano, rispettivamente, un segnaposto per un valore int e uno per un valore float che verrà troncato a 2 cifre decimali.

Funzioni di libreria

Python mette a disposizione anche oltre 40 funzioni predefinite per operare sulle stringhe. Vediamone alcune, tenendo sempre conto che ritornano una nuova stringa e non modificano quella originale (dato che il tipo **str** è **immutabile**). Infatti, per “modificare” una stringa è necessario assegnarle un nuovo valore:

```
>>> s = "Ciao, mondo!"
```

```
>>> s.lower() # Converte tutti i caratteri in minuscolo
```

```
'ciao, mondo!'
```

```
>>> s.upper() # Converte tutti i caratteri in maiuscolo
```

```
'CIAO, MONDO!'
```

```
>>> s.replace("Ciao", "Hello") # Sostituisce "Ciao" in s con "Hello"
```

```
'Hello, mondo!'
```

```
>>> s.replace("o", "x") # Sostituisce "o" in s con "x"
```

```
'Ciax, mxndx!'
```

```
>>> "hello, world!".title() # Converte in maiuscolo le iniziali in hello, world!"
```

```
'Hello, World!'
```

```
>>> s.find("ao") # Ritorna l'indice dell'inizio della sottostringa "ao"  
# all'interno di s
```

```
2
```

```
>>> s.find("hello") # Se non trova la stringa specificata in s find() ritorna -1
```

```
-1
```

```
>>> s.count("o") # Conta quante volte "o" appare in s
```

```
3
```

```
>>> "  Senza spazi ".strip() # Rimuove gli spazi iniziali e finali
```

```
'Senza spazi'
```

```
>>> "22810293".isnumeric() # Ritorna True se la stringa specificata contiene  
# solo cifre
```

```
True
```

La lista completa di metodi per operare con le stringhe è presente al seguente indirizzo <https://docs.python.org/3/library/stdtypes.html#string-methods>

Liste

Un dato di tipo list rappresenta una sequenza di valori separati da virgole delimitata da parentesi quadre. In genere viene utilizzato per rappresentare una sequenza mutabile di oggetti, in genere omogenei.

```
>>> capitali = ["Roma", "Oslo", "Lisbona", "Reykjavik"]  
>>> temperature = [25.3, 17.4, 18.2, 34.0]  
>>> lista_vuota = []
```

Gli elementi di una lista possono essere anche di tipo diverso (per esempio, cognome, nome, età, peso, residenza):

```
>>> archimede = ["Pitagorico", "Archimede", 48, 81.3, "Paperopoli"]  
>>> len(archimede) # La lunghezza è pari al numero di elementi della lista
```

Le liste, come le stringhe, sono un tipo di sequenza e supportano le funzionalità di *indexing*, *slicing*, *contenimento*, *concatenazione* e *ripetizione*.

Le liste, a differenza delle stringhe, sono mutabili quindi è possibile modificarne gli elementi utilizzando la funzionalità di *indexing*

```
>>> numeri = [21, 14, 37]
```

```
>>> numeri[1] = 55 # Modifica il valore del secondo elemento
```

```
>>> numeri
```

```
[21, 55, 37]
```

Inoltre è possibile aggiungere in coda o rimuovere elementi dalla lista con nel seguente modo:

```
>>> numeri.append(8)
```

```
>>> numeri
```

```
[21, 55, 37, 8]
```

```
>>> del numeri[2] # Rimuovi il terzo elemento
```

```
>>> numeri
```

```
[21, 55, 8]
```

Vediamo alcune funzioni avanzate che Python mette disposizione per operare sulle liste:

```
>>> numeri = [8, 21, 14, 37, 62, 8, 93, 15, 14]
```

```
>>> max(numeri)
```

```
93
```

```
>>> min(numeri)
```

```
8
```

```
>>> sum(numeri)
```

```
272
```

```
>>> numeri.count(8) # Conta quante volte l'8 appare nella lista numeri
```

```
2
```

```
>>> numeri.reverse() # Modifica la lista, invertendo l'ordine dei suoi elementi
```

```
>>> numeri
```

```
[14, 15, 93, 8, 62, 37, 14, 21, 8]
```

```
>>> numeri.sort() # Modifica la lista, ordinando i suoi elementi in modo crescente
>>> numeri
[8, 8, 14, 14, 15, 21, 37, 62, 93]
>>> numeri.remove(14) # Rimuove la prima occorrenza di 14 nella lista
>>> numeri
[8, 8, 14, 15, 21, 37, 62, 93]
```

Una lista può essere nidificata, cioè i suoi elementi possono essere liste a loro volta. Nel caso di liste multidimensionali si individuano gli elementi procedendo dall'esterno all'interno con una sequenza di indici.

Una matrice è una struttura bidimensionale che può essere rappresentata da una tabella e nella quale ogni elemento è individuato da un indice di riga e un indice di colonna. Le matrici sono fondamentali nell'algebra lineare e sono utili in tutte quelle situazioni nelle quali i dati possono essere rappresentati in formato tabellare.

```
>>> matrice = [[4, 1, 9], [6, 7, 2], [3, 8, 5]]
```

```
>>> matrice[1]
```

```
[6, 7, 2]
```

```
>>> matrice[1][1]
```

```
7
```

Il listato precedente dichiara una variabile *matrice* che contiene una lista bidimensionale che potrebbe essere rappresentato in forma tabellare nel seguente modo:

Indici	0	1	2
0	4	1	9
1	6	7	2
2	3	8	5

Tuple

Una tupla è un elenco immutabile di valori normalmente delimitati da parentesi tonde. Quando non c'è ambiguità di interpretazione, come nell'assegnazione multipla, “a, b = 3, 4”, le parentesi si possono omettere.

Sulle tuple possiamo effettuare le stesse operazioni viste per le liste, tranne quelle che modificano la lista o i suoi valori in quanto anche le tuple, come le stringhe, sono **immutabili** quindi una volta create non è possibile aggiungere, rimuovere, o modificare gli elementi.


```
>>> t = ('Danilo', 35, 1.70)
>>> t[0] # le tuple supportano indexing
'Danilo'
>>> t[:2] # slicing
('Danilo', 35)
>>> 35 in t # gli operatori di contenimento "in" e "not in"
True
>>> t + ('R-DeV', 'CTO') # concatenazione (ritorna una nuova tupla)
('Danilo', 35, 1.70, 'R-DeV', 'CTO')
>>> t * 2 # ripetizione (ritorna una nuova tupla)
('Danilo', 35, 1.70, 'Danilo', 35, 1.70)
```

Anche le tuple possono essere impiegate per costruire strutture multi-dimesionali:

```
>>> t = ('Danilo', 'Di Nuzzo', 'R-DeV', ('CTO', 'PM'))
>>> t[3]
('CTO', 'PM')
```

Set, frozenset e dizionari

Python mette a disposizione il tipo set per rappresentare gli insiemi (set). A differenza delle liste, negli insiemi gli elementi non sono ordinati né ripetuti. Per creare un insieme possiamo partire da una lista, una tupla, una stringa oppure un elenco di valori separati da virgole e racchiusi da parentesi graffe:

```
>>> insieme_vuoto = set()
>>> type(insieme_vuoto)
<class 'set'>
>>> len(insieme_vuoto)
0
>>> discendenti = {'Elia', 'Agnese', 'Lorenzo'}
>>> type(figli)
<class 'set'>
```

Gli elementi dei set devono essere unici: se non lo sono Python rimuoverà automaticamente i duplicati, un modo per rimuovere i duplicati di una lista è quello di convertirla in set e quindi riconvertirla in lista:

```
>>> numeri_estratti = [40, 50, 20, 40, 60, 30, 20, 20, 10, 90]
>>> set(numeri_estratti)
{40, 10, 50, 20, 90, 60, 30}
>>> list(set(numeri_estratti)) # Composizione delle due operazioni di conversione
[40, 10, 50, 20, 90, 60, 30]
```

La differenza tra set e frozenset, oltre ai metodi utilizzati per la loro creazione (**set()** e **frozenset()**), è che nel primo caso parliamo di una sequenza mutabile mentre la seconda, frozenset, non lo è.

Set e frozenset supportano alcune operazioni di base delle sequenze (**len()**, **min()**, **max()**, **in**, **not in**).

I Set mettono a disposizione anche i seguenti metodi:

```
>>> insieme = {10, 20, 30, 40}
```

```
>>> insieme.add(50) # Aggiunge 50 all'insieme
```

```
>>> insieme.remove(15) # Rimuove 15 dall'insieme se non presente restituisce  
                        # KeyError
```

```
>>> insieme.discard(10) # Rimuove 10 dall'insieme se presente
```

```
>>> insieme.pop() # Restituisce e rimuove un elemento arbitrario
```

```
>>> insieme.clear() # Rimuove tutti gli elementi
```

```
>>> insieme.copy() # Crea una copia del set e la restituisce
```

Naturalmente questi metodi, dato che modificano l'insieme, si applicano solo ai set. L'unico metodo supportato dal frozenset è copy().

I set, inoltre, supportano anche una serie di operazioni tipiche degli insiemi.

Operatore	Metodo	Descrizione
	<code>s1.isdisjoint(s2)</code>	Restituisce True se i due set non hanno elementi in comune
<code>s1 <= s2</code>	<code>s1.issubset(s2)</code>	Restituisce True se s1 è un sottoinsieme di s2
<code>s1 < s2</code>		Restituisce True se s1 è un sottoinsieme proprio di s2
<code>s1 >= s2</code>	<code>s1.issuperset(s2)</code>	Restituisce True se s2 è un sottoinsieme di s1
<code>s1 > s2</code>		Restituisce True se s2 è un sottoinsieme proprio di s1
<code>s1 s2 ...</code>	<code>s1.union(s2, ...)</code>	Restituisce l'unione degli insiemi (tutti gli elementi)
<code>s1 & s2 & ...</code>	<code>s1.intersection(s2, ...)</code>	Restituisce l'intersezione degli insieme (elementi in comune a tutti i set)

Operatore	Metodo	Descrizione
$s1 - s2 - \dots$	<code>s1.difference(s2, ...)</code>	Restituisce la differenza tra gli insiemi (elementi di $s1$ che non sono negli altri set)
$s1 \wedge s2$	<code>s1.symmetric_difference(s2)</code>	Restituisce gli elementi dei due set senza quelli comuni a entrambi
$s1 = s2 \dots$	<code>s1.update(s2, ...)</code>	Aggiunge a $s1$ gli elementi degli altri insiemi
$s1 \&= s2 \& \dots$	<code>s1.intersection_update(s2, ...)</code>	Aggiorna $s1$ in modo che contenga solo gli elementi comuni a tutti gli insiemi
$s1 -= s2 \dots$	<code>s1.difference_update(s2, ...)</code>	Rimuove da $s1$ tutti gli elementi degli altri insiemi
$s1 \wedge= s2$	<code>s1.symmetric_difference_update(s2)</code>	Aggiorna $s1$ in modo che contenga solo gli elementi non comuni ai due insiemi

Un dizionario, in Python **dict** da dictionary, è un tipo di dato formato da coppie *chiave:valore* separate da virgole e delimitate da parentesi graffe. Una chiave può essere un valore immutabile qualsiasi (numero, booleano, stringa, tupla) ed è associata a un altro valore qualsiasi. In alcuni linguaggi questa struttura dati si chiama array associativo o mappa, perché associa (o collega) un valore a un altro valore.

I dizionari vengono definiti elencando tra parentesi graffe (**{}**) una serie di elementi separati da virgole (,), dove ogni elemento è formato da una chiave e un valore separati dai due punti (:). È possibile creare un dizionario vuoto usando le parentesi graffe senza nessun elemento all'interno.

```
>>> dizionario_vuoto = {}
```

```
>>> supereroi = {21:"Flash", 7:"Tempesta", 14:"Hulk"}
```

```
>>> lista_spesa = {"Pere":2, "Mele":4, "Ananas":1}
```

Nell'esempio precedente possiamo notare come le chiavi non devono essere necessariamente numeriche

Una volta creato un dizionario, è possibile ottenere il valore associato a una chiave usando la sintassi *dizionario[chiave]*:

```
>>> lista_spesa["Mele"]
```

```
4
```

```
>>> lista_spesa["Ananas"] += 2 # Incrementa di 2 il valore associato alla key Ananas
```

Inoltre si possono passare in rassegna le chiavi, i valori o gli elementi, cioè le coppie chiave:valore del dizionario:

```
>>> for voce in lista_spesa: # Scorre le chiavi, equivale a: lista_spesa.keys()
```

```
>>> for valore in lista_spesa.values(): # Scorre i valori del dizionario
```

```
>>> for chiave, valore in lista_spesa.items(): # Scorre gli elementi (item)
```

for è una parola chiave che vedremo nelle prossime slides.

Per aggiungere un elemento si usa la seguente modalità:

```
>>> lista_spesa["Arance"] = 8 # Aggiunge l'elemento con chiave "Arance" e valore 8
```


Selezionando una chiave che non esiste nel dizionario Python dà un errore. Il metodo **get()** permette di specificare un valore da ritornare in caso di chiave non presente:

```
>>> lista_spesa["Kiwi"] # Accedere a un elemento con una chiave inesistente
```

Traceback (most recent call last):

File "<pyshell#35>", line 1, in <module>

```
    lista_spesa["Kiwi"] # Accedere a un elemento con una chiave inesistente
```

KeyError: 'Kiwi'

```
>>> lista_spesa.get("Kiwi", 0) # Se la chiave non è presente ritorna il valore 0
```

```
0
```

Nella prossima slide sono riportati i principali metodi dei dizionari e la relativa descrizione.

Metodo	Descrizione
d.items()	Restituisce gli elementi di d come un insieme di tuple
d.keys()	Restituisce le chiavi di d
d.values()	Restituisce i valori di d
d.get(chiave, default)	Restituisce il valore corrispondente a chiave se presente, altrimenti il valore di default (None se non specificato)
d.pop(chiave, default)	Rimuove e restituisce il valore corrispondente a chiave se presente, altrimenti il valore di default (dà KeyError se non specificato)
d.popitem()	Rimuove e restituisce un elemento arbitrario da d
d.update(d2)	Aggiunge gli elementi del dizionario d2 a quelli di d
d.copy()	Crea e restituisce una copia di d
d.clear()	Rimuove tutti gli elementi di d

Cicli iterativi

In questo capitolo introduciamo i cicli e la geometria della tartaruga (in inglese *turtle*). Il ciclo permette la ripetizione di un blocco di istruzioni, mentre l'estensione ***turtle*** di Python permette di scrivere programmi con un output grafico basato sulla geometria e sul piano cartesiano.

Python mette a disposizione due tipologie di cicli:

- **for**: generalmente utilizzato per “scorrere” una lista di oggetti. Il numero di iterazioni è legato al numero di elementi della lista usata;
- **while**: che ripete le sue iterazioni finché una certa condizione è vera.

Dopo aver dato un descrizione di “ciclo” vediamo nella pratica quali problematiche risolvono.

Per farlo utilizzeremo la libreria **turtle** che, come anticipato, permette di scrivere programmi con un output grafico basato sulla geometria e sul piano cartesiano. L'idea base è disegnare su un piano grazie a robot virtuali (o tartarughe digitali) istruiti a muoversi lasciando un'impronta o una traccia del percorso compiuto.

Proviamo ad inserire all'interno di IDLE le seguenti istruzioni:

```
>>>import turtle  
>>>t = turtle.Turtle()
```

Dopo aver eseguito la seconda riga, appare una nuova finestra intitolata “Python Turtle Graphics”. Al centro di uno sfondo bianco, chiamato canvas, o tela, che rappresenta l'area di disegno, appare una tartaruga rappresentata dall'immagine di una punta di una freccia.

Posizioniamo le due finestre una a fianco all'altra, in questo modo possiamo scrivere istruzioni nella finestra dell'interprete e contemporaneamente vedere il loro effetto nell'altra finestra.

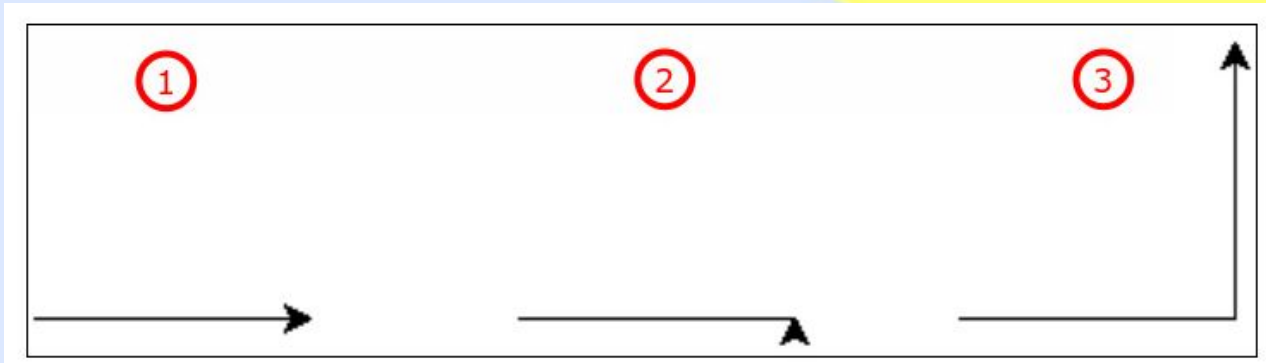
Ora proviamo a disegnare sul nostro canvas con le seguenti istruzioni:

t.forward(100) # Procedi in avanti di 100 passi (o pixel)

t.left(90) # Ruota a sinistra, cioè in senso antiorario, di 90 gradi

t.forward(100) # Procedi in avanti di 100 passi (o pixel)

Nell'immagine seguente possiamo osservare il risultato dell'esecuzione dei singoli passaggi



Turtle usa un sistema di riferimento ortogonale cartesiano simile a quello della battaglia navale. Le celle sono i singoli pixel. Gli assi x e y dividono l'area di disegno in 4 quadranti: il centro, da dove parte il cursore, ha coordinate (0, 0), l'ascissa x (coordinata orizzontale) cresce verso destra e l'ordinata y (coordinata verticale) cresce verso l'alto.

La direzione (heading) indica dove punta, o guarda, la tartaruga e determina il senso di marcia quando questa procede in avanti.

Per andare all'indietro (backward) è possibile utilizzare il corrispondente comando oppure procedere come il gambero, andando in avanti di una distanza negativa, cioè camminando all'indietro. Quindi, le due istruzioni seguenti sono equivalenti:

t.forward(-100) # Vai all'indietro di 100 pixel

t.backward(100) # idem (come sopra)

Per le rotazioni, di default gli angoli si misurano in gradi relativamente alla direzione corrente della tartaruga, che all'inizio punta verso destra. Ruotando verso sinistra (left) la tartaruga gira in senso antiorario, ruotando verso destra (right) gira in senso orario.

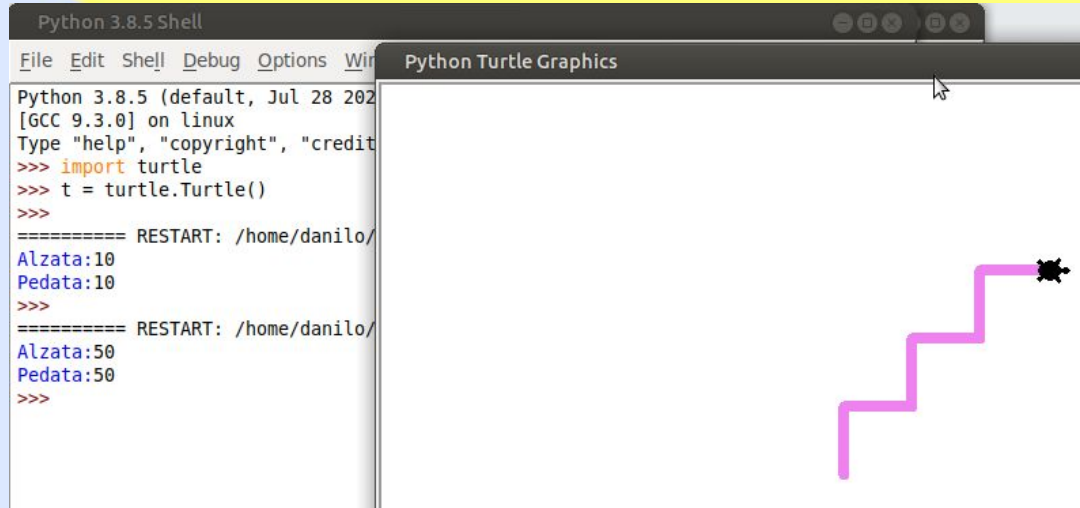
È possibile anche utilizzare valori negativi per ruotare nel verso opposto. Quindi, le due istruzioni seguenti sono equivalenti:

ninja.left(-90) # Ruota in senso orario di 90°

ninja.right(90) # idem

Ora che abbiamo le nozioni di base per poterlo fare proviamo a disegnare una scala colorata chiedendo all'utente le misure del “segmento verticale” (o alzata) e del “segmento orizzontale” (o pedata).

Per ottenere il risultato presente nell'immagine precedente dobbiamo utilizzare due nuovi comandi: ***pensize(int)*** per impostare la larghezza del tratto della penna e ***pencolor(string)*** per definire il colore.



```
Python 3.8.5 Shell
File Edit Shell Debug Options Window Help
Python 3.8.5 (default, Jul 28 2020) on linux
Type "help", "copyright", "credits()" or "quit()" for more
>>> import turtle
>>> t = turtle.Turtle()
>>>
===== RESTART: /home/danilo/Python3.8.5 Shell
Alzata:10
Pedata:10
>>>
===== RESTART: /home/danilo/Python3.8.5 Shell
Alzata:50
Pedata:50
>>>
```

The image shows a Python 3.8.5 Shell window and a Python Turtle Graphics window. The shell window displays the execution of a script that imports the turtle module, creates a turtle object, and then uses the `pensize` and `pencolor` methods to draw a pink staircase. The Turtle Graphics window shows the resulting drawing: a pink staircase with three steps, each with a width of 50 and a height of 10. The drawing is completed with a small black star at the end of the final step.

Durante la stesura del codice vi renderete conto che dovrete necessariamente ripetere 3 volte le istruzioni per la creazione di un gradino. Se i gradini fossero 40 il programma sarebbe decisamente lungo e soprattutto ripetitivo!

Turtle mette a disposizione molti più comandi (metodi) di quelli che abbiamo visto fino ad ora, per esempio, il comando ***shape()*** gli dice di cambiare la forma, o il costume, con cui appare; ***penup()*** di sollevare la penna in modo che i suoi spostamenti successivi non lascino traccia sul piano; ***pendown()*** di riprendere a scrivere quando si muove.

Alcuni metodi di Turtle hanno più nomi: possiamo usare ***penup*** oppure ***up***, ***pendown*** o ***down***, ***forward*** oppure ***fd***, ***left*** o ***lt***, eccetera.

Per un elenco completo potete consultare la documentazione ufficiale all'indirizzo <https://docs.python.org/3.9/library/turtle.html>.

Il ciclo for

Il ciclo for ci permette di iterare su tutti gli elementi di un iterabile ed eseguire un determinato blocco di codice. Un iterabile è un qualsiasi oggetto in grado di restituire tutti gli elementi uno dopo l'altro, come ad esempio liste, tuple, set, dizionari (restituiscono le chiavi), che vedremo più avanti nel dettaglio.

Per disegnare la scala con turtle abbiamo dovuto ripetere le istruzioni per il numero di gradini che la compongono. Un modo alternativo, più conciso ed informatico, è quello di utilizzare un ciclo for, che serve proprio a ripetere un blocco di istruzioni un determinato numero di volte.

```
for i in range(4): # Ripeti per 4 volte...
```

```
    istruzione 1
```

```
    istruzione 2
```

La funzione ***range(4)*** ritorna un elenco di 4 valori, costituito da numeri da 0 a 3. Il ciclo **for** passa in rassegna, uno alla volta, tutti gli elementi di un elenco.

A ogni passaggio:

- la variabile di ciclo assume il valore dell'elemento corrente nell'elenco (spesso, se i valori sono numerici, è chiamata *i*, che sta per index, indice);
- vengono eseguite le istruzioni del corpo del ciclo **for**.

Puoi leggere l'istruzione **for** anche in questo modo: per *i* che assume i valori da 0 a 3 esegui il blocco successivo. *i* è chiamata anche contatore perché di fatto conta quante volte viene ripetuto il ciclo (o loop).

In questo caso, puoi interpretare il ciclo anche così: ripeti per 4 volte il blocco successivo.

La forma generale della funzione predefinita **range()** è la seguente:

```
range(1, 10, 2)
```

```
range(stop) -> range object
```

```
range(start, stop[, step]) -> range object
```

Vediamo il significato dei 3 argomenti: **start** indica il valore, incluso, da cui partire e vale 0 di default; **stop** è l'unico argomento obbligatorio e indica il termine ultimo, non incluso, dell'elenco, quindi l'ultimo valore è (stop - 1); **step** (passo) è il valore dell'incremento, di default è 1.

```
>>> for i in range(3): # Ripete il ciclo per 3 volte; "i" assume i valori 0, 1 e 2
```

```
    print(i) # Stampa 0 1 2
```

```
>>> for i in range(1, 10, 2): # "i" va da 1, incluso, a 10, escluso, con passo 2
```

```
    print(i, end=", ") # Stampa 1, 3, 5, 7, 9
```

Oltre all'utilizzo con la funzione **range()**, quindi per effettuare un determinato numero di iterazioni, il ciclo **for** può essere utilizzato per “scorrere” una lista di elementi (o più in generale un qualsiasi dato riconducibile ad una sequenza di elementi), ad esempio è possibile:

```
>>> seq = ['nero', 'verde', 'rosso', 'giallo']
```

```
>>> for s in seq:
```

```
    print(s)
```

```
nero
```

```
verde
```

```
rosso
```

```
giallo
```

In Python puoi realizzare un ciclo anche col costrutto **while** (fintanto che). Vediamo, per esempio, come stampare i numeri da 1 a 10:

```
>>> i = 1
>>> while i <= 10:
    print(i, end=" ")
    i = i + 1
```

1 2 3 4 5 6 7 8 9 10

Python mette a disposizione i costrutti **break** e **continue** che possono venirci in aiuto in particolari esigenze in cui abbiamo la necessità di avere più controllo sui cicli e/o per ottimizzarli. Il primo interrompe il ciclo mentre il secondo interrompe l'iterazione corrente per passare direttamente alla successiva. Altra interessante possibilità messa a disposizione da Python è data dalla presenza delle clausole **for-else** e **while-else**. In pratica è possibile aggiungere una clausola else ai due cicli, come già visto per il costrutto **if**, che verrà eseguito quando il ciclo termina tutte le iterazione. Se invece il ciclo è interrotto da un **break**, l'else non viene eseguito.

Esercitazioni

1. Creare un programma che utilizza un ciclo for per effettuare un conto alla rovescia della durata di 10 secondi e produce il seguente output:
10...9...8...7...6...5...4...3...2...1...vai!
per far sì che tra una stampa e l'altra il tempo trascorso sia di circa 1 secondo si può utilizzare il metodo `time.sleep()` del modulo `time`.
2. Creare un programma che, con il supporto di `turtle`, disegni sul canvas un quadrato utilizzando un ciclo for.
3. Creare un programma che utilizza un ciclo while per stampare tutti i numeri minori di `n` e divisibili per 3 e 5, con `n` inserito dall'utente. Ecco un esempio di esecuzione:
Numero: 100
15 30 45 60 75 90

4. Creare un programma che utilizza un ciclo for per stampare i quadrati dei numeri da 1 a n (con n inserito dall'utente) e la loro somma:

Numero: 10

1 4 9 16 25 36 49 64 81 100

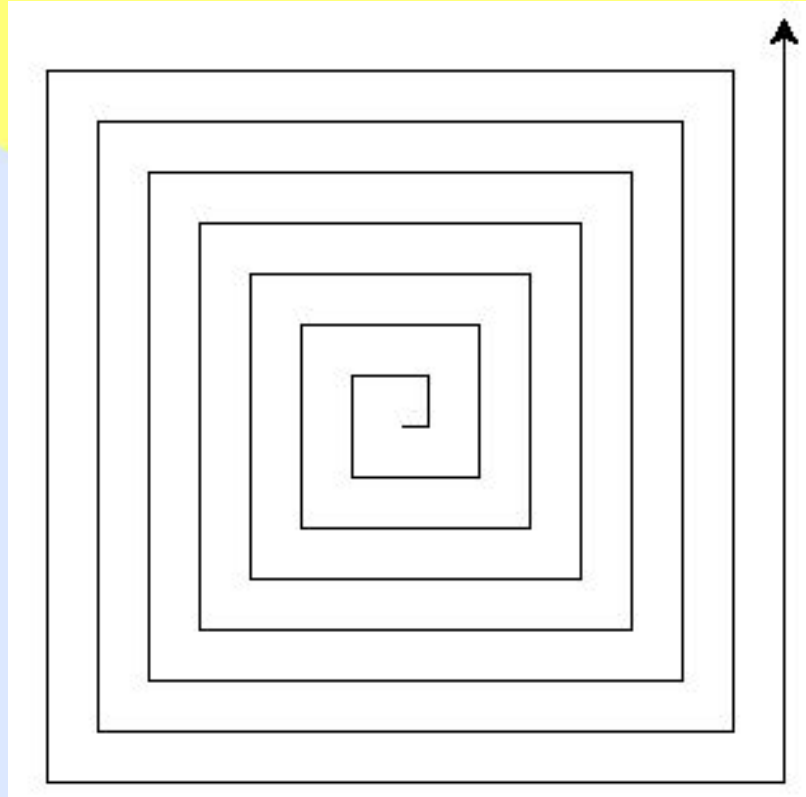
Somma quadrati: 385

5. Creare il programma stella.py per disegnare una stella a 5 punte come nella figura seguente



suggerimento: una possibile soluzione tiene conto del fatto che tutti gli angoli hanno un'apertura di 36°

6. Creare un programma per ottenere il seguente output (spirale quadratica):



6. Creare un programma che genera una griglia come nella figura seguente con un certo numero di righe e un numero di colonne pari al numero di colori definiti in un elenco.

La lista di colori da usare per ottenere questo risultato è la seguente:

"green", "red", "blue", "yellow", "purple", "lightblue"

ps. scegliere la struttura dati che si ritiene più opportuna.

