notebook1

March 2, 2023

1 Benvenuti in Jupyter Notebook

2 A cura di Mauro Maria Baldi

3 mauromaria.baldi@unimc.it

Il titolo qua sopra è stato scritto selezionando la cella in modalità "markdown" e anteponendo un hashtag piú uno spazio al titolo: # Benvenuti in Jupyter Notebook. Poi si preme SHIFT+INVIO

4 Titolo con un hashtag

4.1 Titolo con due hashtag

4.1.1 ...con tre

e infine con quattro La bellezza di Jupyter Notebook è che, suddividendo il lavoro in piú celle, si può documentare il proprio codice in una maniera molto elegante e professionale. Fino adesso abbiamo considerato celle di markdown (testuali). Vediamo adesso una cella di codice. È tradizione in ogni linguaggio di programmazione realizzare un primo programma stampando a video la scritta "Hello, world!".

[1]:
```
print("Hello, world!")
```

```
Hello, world!
```

Anche qui valgono tutte le regole per l'utilizzo di Python come calcolatrice che abbiamo già visto. Vediamo qualche esempio per rinfrescarci la memoria...

[2]:
```
2 + 3
```

[2]: 5

[3]:
```
2**3
```

[3]: 8

[4]:
```
import math
r = 2
area_circ = math.pi*pow(r, 2)
```

```
area_circ
```

[4]:
```
12.566370614359172
```

Ancora un esempio:

[5]:
```
x = eval(input("Inserisci un numero: "))
exp = math.e**x
auxString = "e^{:.3f} = {:.3f}".format(x, exp)
print(auxString)
```

```
Inserisci un numero: 3
e^3.000 = 20.086
```

In modalità markdown posso realizzare un elenco mettendo * davanti a ogni punto. Vediamo un esempio subito dopo la seguente cella di codice:

[6]:
```
a = 1
b = 2.0
c = "sono la solita stringa"
d = True
print(type(a), a)
print(type(b), b)
print(type(c), c)
print(type(d), d)
```

```
<class 'int'> 1
<class 'float'> 2.0
<class 'str'> sono la solita stringa
<class 'bool'> True
```

Nella cella precedente abbiamo utilizzato quattro variabili: * la variabile a è di tipo intero e vale 1 * la variabile b è di tipo float e vale 2.0 * la variabile c è ...la solita stringa! * la variabile d ...dice il vero!

Un'altra grande potenzialità di Jupyter Notebook è la possibilità di integrare nelle celle di markdown le formule latex. Vediamo questa splendida funzionalità con un altro esempio

[7]:
```
x = float(input("Dammi un numero: "))
y = math.sin(x)
print("sin(%.3f) = %.3f" % (x, y))
```

```
Dammi un numero: 3
sin(3.000) = 0.141
```

La cella precedente riguarda la funzione $y = \sin(x)$. Invece in qualche cella precedente abbiamo utilizzato la formula dell'area del cerchio $A = \pi r^2$.

Possiamo anche creare delle tabelle:

Regione	Capoluogo
Sicilia	Palermo
Marche	Ancona

Possiamo anche scrivere in corsivo e grassetto frapponendo la porzione di testo interessata rispettivamente tra uno e due asterischi

[8]:
```
print(type(a))
print(type(b))
```

```
<class 'int'>
<class 'float'>
```

La variabile **a** è di tipo *int* mentre la varibile **b** è di tipo *float*

notebook2

March 2, 2023

1 La selezione

1.1 A cura di Mauro Maria Baldi

1.2 mauromaria.baldi@unimc.it

Introduciamo una nuova istruzione di Python: la selezione. Per vedere questa nuova istruzione, proviamo a svolgere il seguente esercizio: si inserisca in una variabile il costo di un prodotto e si chieda all'utente quanti soldi ha. Dire se l'utente può comprare o meno il prodotto. Nel caso che i soldi a disposizione siano insufficienti, dire quanti soldi mancano per completare l'acquisto.

[2]:
```
costo = 50
soldi = eval(input("Quanti soldi hai? "))
if soldi < costo:
    diff = costo - soldi
    print("Non puoi comprare il prodotto, ti mancano", diff, "€")
else:
    print("Puoi comprare il prodotto")
```

```
Quanti soldi hai? 30
Non puoi comprare il prodotto, ti mancano 20 €
```

Nell'esempio qui sopra abbiamo gestito due soli casi: o si hanno i soldi necessarî all'acquisto o non si può comprare il prodotto. Python ci permette di gestire anche di operare con piú scelte. Vediamolo con il seguente esercizio: si inserisca in una variabile il costo di un prodotto. Si chieda all'utente quanti prodotti intende acquistare. Se il numero di prodotti è inferiore a 4 il prezzo è pieno, se si acquistano da 5 a 7 prodotti si ha uno sconto del 10%, se si acquistano da 8 a 10 prodotti si ha uno sconto del 20%, se invece si acquistano piú di 10 prodotti si ha uno sconto del 30%. Si stampi a video l'importo totale da pagare da parte dell'utente.

[1]:
```
costoProdotto = 10
n = int(input("Quanti prodotti vuoi acquistare? "))
if n <= 3:
    prezzo = n*costoProdotto
elif n <= 7:
    prezzo = n*costoProdotto*(1 - 10/100)
elif n <= 10:
    prezzo = n*costoProdotto*(1 - 20/100)
else:
    prezzo = n*costoProdotto*(1 - 30/100)
```

```
print("Totale:", prezzo)
```

```
Quanti prodotti vuoi acquistare? 4
Totale: 36.0
```

notebook3

March 2, 2023

1 La funzione range e il ciclo *for*

1.1 A cura di Mauro Maria Baldi

1.2 mauromaria.baldi@unimc.it

La funzione di built-in *range(n)*, ove n è un intero, tiene traccia di tutti gli interi da 0 a $n-1$.

[2]:
```
range(10)
```

[2]:
```
range(0, 10)
```

Osserviamo che il precedente comando non ci permette di vedere tutti gli interi da 0 a 9. Per fare questo, ci serviamo di un ciclo for:

[3]:
```
for k in range(10):
    print(k, end = ' ')
```

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

Nella cella precedente ci accorgiamo di due novità. La prima riguarda il ciclo for stesso. La scrittura *for k in range(10)* vuol dire che k assume di volta in volta tutti i valori in range(10) e dunque prima assume il valore 0, poi il valore 1 e così via fino al valore 9.

La seconda novità riguarda l'aggiunta dell'opzione *end = ' '* dentro alla funzione di built-in print(). Senza specifincare nulla, la print() va a capo. In questo modo, invece, invece che andare a capo si aggiunge uno spazio (per non avere tutti i numeri attaccati). Per convincerci, proviamo a eseguire lo stesso ciclo for ma senza quest'ultima opzione.

[4]:
```
for k in range(10):
    print(k)
```

```
0
1
2
3
4
5
6
7
```

```
8
9
```

Lo stesso principio del ciclo for vale anche per le stringhe. Vediamo un altro esempio.

[6]:
```
s = "Ciao, sono una stringa"
print(s)
for c in s:
    print(c, end = '|')
```

```
Ciao, sono una stringa
C|i|a|o|,| |s|o|n|o| |u|n|a| |s|t|r|i|n|g|a|
```

La sintassi completa della funzione di built-in range è:

range(start, stop[, step]),

ove: * *start* è il numero di partenza incluso che nell'istruzione *range(10)* abbiamo omesso. In altre parole, scrivere *range(10)* è come scrivere *range(0, 10)*, che è proprio la scritta che è comparsa alla prima cella di output * *stop* è il numero di arrivo escluso. Infatti con *range(0, 10)* partiamo sí da 0, ma arriviamo fino a 9. Si tenga bene a mente questa convenzione in quanto la incontreremo anche piú avanti. * una convenzione informatica afferma che mettere un parametro fra parentesi quadre, vuol dire che il suddetto parametro è opzionale. Pertanto *step* è un parametro opzionale e che possiamo inserire se vogliamo avere degli incrementi non necessariamente unitarî.

Vediamo alcuni esempî per chiarire meglio la teoria appena esposta. Ad esempio, ecco cosa si può scrivere se si vogliono scrivere tutti i numeri da 2 a 10:

[7]:
```
for k in range(2, 11):
    print(k, end = ' ')
```

```
2 3 4 5 6 7 8 9 10
```

Vogliamo adesso stampare tutti i numeri pari da 2 a 20. In questo caso basterà partire da 2 e imporre un incremento di 2 invece che unitario:

[9]:
```
for k in range(2, 21, 2):
    print(k, end = " ")
```

```
2 4 6 8 10 12 14 16 18 20
```

Finora abbiamo considerato numeri crescenti. È anche possibile considerare dei decrementi impostando uno *step* negativo. Vediamo, ad esempio, come poter scrivere tutti i numeri in ordine inverso da 10 a 3:

[10]:
```
for k in range(10, 2, -1):
    print(k, end = ' ')
```

```
10 9 8 7 6 5 4 3
```

Anche in questo caso è possibile impostare dei decrementi non necessariamente unitarî. Ad esempio possiamo volere stampare in ordine decrescente tutti i multipli di 4 positivi minori o uguali a 20:

[11]:
```
for k in range(20, 3, -4):
    print(k, end = ' ')
```

```
20 16 12 8 4
```

2 Cicli for annidati

Finora abbiamo visto cicli *for* semplici. Possiamo inserire degli altri cicli *for* all'interno di un ciclo *for*. Si parla in questo caso di cicli *for* annidati. A titolo di esempio, facciamo un esercizio con l'obiettivo di stampare a video la tavola pitagorica. Avremo un ciclo *for* esterno (indice i) relativo alle righe della tabella. Fissato un valore di i, partirà un ciclo *for* interno che lavorerà sulle colonne e ivi scriverà i primi dieci multipli di i. Una possibile soluzione è la seguente:

[1]:
```
print("TAVOLA PITAGORIGA\n") # Il carattere speciale (di escape) \n ci fa␣
 ↪andare ulteriormente a capo
for i in range(1, 11):
    for j in range(1, 11):
        print(i*j, end = ' ')
    print()
```

```
TAVOLA PITAGORIGA

1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Osserviamo che il programma è corretto ma non è molto elegante. Infatti non appena i multipli iniziano ad avere due cifre, si perde l'allineamento. Per ovviare a questo problema invece che uno spazio, inseriamo un cosiddetto tab: un carattere speciale (di escape) che di solito consiste in quattro spazî e che in questo caso ci garantisce un buon allineamento.

[2]:
```
print("TAVOLA PITAGORIGA\n") # Il carattere speciale (di escape) \n ci fa␣
 ↪andare ulteriormente a capo
for i in range(1, 11):
    for j in range(1, 11):
        print(i*j, end = '\t')
    print()
```

```
TAVOLA PITAGORIGA

1       2       3       4       5       6       7       8       9       10
```

2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

notebook4

March 2, 2023

1 I cicli *while*

1.1 A cura di Mauro Maria Baldi

1.2 mauromaria.baldi@unimc.it

Un'alternativa ai cicli for sono i cicli while. La logica è che si sta dentro al ciclo *mentre* viene soddisfatta una certa condizione. Proviamo a rivisitare alcuni esercizî svolti in precedenza, stavolta utilizzando il ciclo while. Uno di questi è quello della simulazione del lancio del razzo, di cui riportiamo qui sia il codice vecchio col ciclo for che il codice nuovo col ciclo while.

[1]:
```
# Codice vecchio (ciclo for)
n = int(input("Quanto manca al lancio? "))
for k in range(n, 0, -1):
    print(k, end = "...")
print("via!")

# Codice nuovo (ciclo while)
k = n
while k >= 1:
    print(k, end = "...")
    k = k - 1
print("via!")
```

```
Quanto manca al lancio? 6
6...5...4...3...2...1...via!
6...5...4...3...2...1...via!
```

Approfittiamo del codice qui sopra per introdurre una "scorciatoria" offerta da Python. L'istruzione $k = k-1$ si può condensare in $k- = 1$. Simili considerazioni valgono per le altre operazioni. Vediamo alcuni esempî.

[4]:
```
m = 2
m *= 3 # è come dire m = m*3, quindi ora m vale 6
print(m)
m /= 2 # è come dire m = m/2, quindi ora m vale 3
print(m)
m -= 1 # è come dire m = m - 1, quindi ora m vale 2
print(m)
m **= -1 # è come dire m = m**-1, quindi ora m vale 0.5
```

```
print(m)
```

```
6
3.0
2.0
0.5
```

Per convincerci, possiamo riscrivere il ciclo while precedente nel seguente modo:

[5]:
```
k = n
while k >= 1:
    print(k, end = "...")
    k -= 1
print("via!")
```

```
6...5...4...3...2...1...via!
```

Rivisitiamo anche l'esercizio della tavola periodica, aggiungendo al codice vecchio con i cicli for annidati il codice nuovo con i cicli while annidati.

[6]:
```
# Codice vecchio
print("TAVOLE PITAGORIGA\n") # Il carattere speciale (di escape) \n ci fa␣
 ↪andare ulteriormente a capo
for i in range(1, 11):
    for j in range(1, 11):
        print(i*j, end = '\t')
    print()

# Codice nuovo
print("\nUN'ALTRA TAVOLA PITAGORICA\n")
i = 1
while i <= 10:
    j = 1
    while j <= 10:
        print(i*j, end = '\t')
        j+= 1
    print()
    i+= 1
```

```
TAVOLE PITAGORIGA

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
```

```
9       18      27      36      45      54      63      72      81      90
10      20      30      40      50      60      70      80      90      100

UN'ALTRA TAVOLA PITAGORICA

1       2       3       4       5       6       7       8       9       10
2       4       6       8       10      12      14      16      18      20
3       6       9       12      15      18      21      24      27      30
4       8       12      16      20      24      28      32      36      40
5       10      15      20      25      30      35      40      45      50
6       12      18      24      30      36      42      48      54      60
7       14      21      28      35      42      49      56      63      70
8       16      24      32      40      48      56      64      72      80
9       18      27      36      45      54      63      72      81      90
10      20      30      40      50      60      70      80      90      100
```

2 Una differenza importante

Negli esercizî affrontati finora abbiamo visto che possiamo indifferentemente usare il ciclo for o il ciclo while? Ma quando è meglio usare uno piuttosto che l'altro? In linea di massima il ciclo for si usa quando si conosce a priori il numero di iterazioni, mentre il ciclo while quando tale numero non si conosce a priori. Si tratta di una regola di massima e che può trovare delle eccezioni in altri linguaggi di programmazione (si pensi ad esempio ai cicli for infiniti nel C o nel C++ con uscita forzata), ma per restare in Python vediamo un esempio dove la scelta del ciclo while è d'obbligo. Chiediamo all'utente di inserire dei valori. Ne calcoliamo la somma finché l'utente non inserisce la stringa "STOP".

[1]:
```
somma = 0.0
txt = '0.0'
while txt != "STOP":
    somma+= eval(txt)
    txt = input("Inserisci un numero (STOP per terminare): ")
print("La somma dei numeri inseriti vale", somma)
```

```
Inserisci un numero (STOP per terminare): 1
Inserisci un numero (STOP per terminare): 2
Inserisci un numero (STOP per terminare): 3
Inserisci un numero (STOP per terminare): 5.5
Inserisci un numero (STOP per terminare): STOP
La somma dei numeri inseriti vale 11.5
```

3 Cicli "infiniti"

Si rimane nel ciclo while finché viene soddisfatta una certa condizione. Pertanto intestazioni di cicli while come

while True:

while $2 > 1$:

while 1:

generano dei cicli while infiniti. A proposito dell'ultima intestazione, è bene precisare che in Python (come in altri linguaggi di programmazione) i valori diversi da 0 sono considerati come True e quindi scrivere while 1 è come scrivere while True, ovvero una condizione sempre vera!

Per uscire dal ciclo potenzialmente infinito, si effettua un'uscita forzata grazie alla parola riservata break. A titolo di esempio, rivisitiamo l'esercizio precedente con una variante.

[3]:
```
somma = 0.0
while 1:
    txt = input("Inserisci un numero (STOP per terminare): ")
    if txt != "STOP":
        somma+= eval(txt)
    else:
        break
print("La somma dei numeri inseriti vale", somma)
```

```
Inserisci un numero (STOP per terminare): 1
Inserisci un numero (STOP per terminare): 2
Inserisci un numero (STOP per terminare): 3
Inserisci un numero (STOP per terminare): 5.5
Inserisci un numero (STOP per terminare): STOP
La somma dei numeri inseriti vale 11.5
```

notebook5

March 2, 2023

1 Funzioni

1.1 A cura di Mauro Maria Baldi

1.2 mauromaria.baldi@unimc.it

Finora abbiamo parlato delle funzioni di buil-in: ovvero di quelle funzioni fornite da Python come *print()* o *pow()* che ci permettono di svolgere comodamente delle determinate azioni. Possiamo anche noi creare delle funzioni artigianali che potremo richiamare ogni volta che vorremo compiere una determinata azione. Riprendiamo, ad esempio, un esercizio svolto in precedenza:

Esercizio 1: scrivere un programma che assegna alla variabile n il valore 7 e stampa il suo quadrato e il suo cubo.

Il codice relativo a questo esercizio era il seguente:

[1]:
```
n = 7
print("Il quadrato di", n, "è", n**2)
print("Il cubo di", n, "è", n**3)
```

```
Il quadrato di 7 è 49
Il cubo di 7 è 343
```

Se volessimo ripetere l'esercizio, oltre che con n = 7 anche per valori di n pari a 2, -1, 3 e -4, dovremmo scrivere quanto segue:

[2]:
```
# Codice per n = 7
n = 7
print("Il quadrato di", n, "è", n**2)
print("Il cubo di", n, "è", n**3)

# Codice per n = 2
n = 2
print("Il quadrato di", n, "è", n**2)
print("Il cubo di", n, "è", n**3)

# Codice per n = -1
n = -1
print("Il quadrato di", n, "è", n**2)
print("Il cubo di", n, "è", n**3)
```

```
# Codice per n = 3
n = 3
print("Il quadrato di", n, "è", n**2)
print("Il cubo di", n, "è", n**3)

# Codice per n = 4
n = 4
print("Il quadrato di", n, "è", n**2)
print("Il cubo di", n, "è", n**3)
```

```
Il quadrato di 7 è 49
Il cubo di 7 è 343
Il quadrato di 2 è 4
Il cubo di 2 è 8
Il quadrato di -1 è 1
Il cubo di -1 è -1
Il quadrato di 3 è 9
Il cubo di 3 è 27
Il quadrato di 4 è 16
Il cubo di 4 è 64
```

Il codice è corretto e funziona, ma è ripetitivo. Se volessimo riscrivere del codice simile ogni volta che vogliamo ripetere qualcosa, oltre a perdere tempo rischieremmo di commettere degli errori. Le funzioni servono proprio a evitare questa evenienza: si scrive il codice una sola volta e si richiama (eventualmente con parametri diversi) ogni volta che serve. Proviamo allora a ripetere l'esercizio precedente, stavolta definendo la funzione **stampa_quadrato_e_cubo()**.

[16]:
```
def stampa_quadrato_e_cubo(n):
    print("Il quadrato di", n, "è", n**2)
    print("Il cubo di", n, "è", n**3)
```

Nella definizione qui sopra n è un parametro. Questo vuol dire che se si passa il valore 1, verrà stampato il quadrato ed il cubo di 1. Se si passa il valore 2, verrà stampato il quadrato e il cubo di 2. Passiamo, dunque, alla conclusione del codice:

[4]:
```
stampa_quadrato_e_cubo(7)
stampa_quadrato_e_cubo(2)
stampa_quadrato_e_cubo(-1)
stampa_quadrato_e_cubo(3)
stampa_quadrato_e_cubo(-4)
```

```
Il quadrato di 7 è 49
Il cubo di 7 è 343
Il quadrato di 2 è 4
Il cubo di 2 è 8
Il quadrato di -1 è 1
Il cubo di -1 è -1
Il quadrato di 3 è 9
```

```
Il cubo di 3 è 27
Il quadrato di -4 è 16
Il cubo di -4 è -64
```

Come vedremo quando tratteremo le liste, esiste una maniera ancora piú comoda di realizzare il codice, ma sempre utilizzando la funzione **stampa_quadrato_e_cubo()** appena definita.

1.3 Il *docstring*

Per una buona programmazione, è bene commentare il piú possibile il proprio codice. Python offre l'interessante possibilità di commentare con una stringa (eventualmente andando a capo) cosa fa una funzione subito dopo l'intestazione. Cosí facendo, invocando la funzione di built-in **help()**, verrà stampata a video la suddetta stringa di documentazione, che prende proprio il nome di *docstring*. Al momento la docstring di **stampa_quadrato_e_cubo()** è vuota.

[5]:
```
help(stampa_quadrato_e_cubo)
```

```
Help on function stampa_quadrato_e_cubo in module __main__:

stampa_quadrato_e_cubo(n)
```

Proviamo allora a migliorare la nostra funzione **stampa_quadrato_e_cubo()** aggiungendo una docstring descrittiva.

[17]:
```
def stampa_quadrato_e_cubo(n):
    """
        Funzione che stampa a video il quadrato e il cubo di n.
    """
    print("Il quadrato di", n, "è", n**2)
    print("Il cubo di", n, "è", n**3)
```

[7]:
```
stampa_quadrato_e_cubo(-3)
help(stampa_quadrato_e_cubo)
```

```
Il quadrato di -3 è 9
Il cubo di -3 è -27
Help on function stampa_quadrato_e_cubo in module __main__:

stampa_quadrato_e_cubo(n)
    Funzione che stampa a video il quadrato e il cubo di n.
```

Osserviamo che, a livello di codice, è come se avessimo ridefinito nello stesso file *.py* la stessa funzione. Questa è un'operazione lecita in Python che considererà soltanto l'ultima definizione.

2 Variabili locali

Consideriamo il seguente esempio per meglio capire un argomento insidioso e che, se non ben compreso, può dare luogo a errori importanti. Riprendiamo la nostra funzione

stampa_quadrato_e_cubo() applicata al seguente codice:

[8]:
```
a = -1
stampa_quadrato_e_cubo(a)
```

```
Il quadrato di -1 è 1
Il cubo di -1 è -1
```

La funzione **stampa_quadrato_e_cubo()** è stata definita nel seguente modo:

def stampa_quadrato_e_cubo(n): “ “ ” Funzione che stampa a video il quadrato e il cubo di n. “ “ ” print(“Il quadrato di”, n, “è”, n**2) print(“Il cubo di”, n, “è”,** n3)

Ci aspettiamo, dunque, che il parametro n sia la variabile a. In realtà le cose non stanno cosí. Il parametro **a** che si passa alla funzione viene detto *parametro attuale*, mentre il parametro **n** nell'intestazione viene detto parametro attuali. I parametri attuali sono una **copia** dei parametri attuali. Rappresentano, in altre parole, delle **variabili locali** che copiano il valore dei corrispondenti parametri attuali e hanno vita solo all'interno della funzione. Per chiarire meglio questo concetto, vediamo un ulteriore esempio con una funzione che aumenta di 2 il valore del parametro che riceve.

[9]:
```
def aumenta_di_due(x):
    x+= 2
    print("Valore aumentato di 2:", x)

a = 1
aumenta_di_due(a)
print("Ma la variabile a continua a valere", a)
```

```
Valore aumentato di 2: 3
Ma la variabile a continua a valere 1
```

Come si vede dall'esempio precedente, una volta che la funzione è terminata, il valore della variabile a continua a valere 1. In altre parole la variabile a è sempre stata pari a 1 mentre il parametro formale x ha “copiato” il valore di a per poi venire incrementato di 2. Come appena detto, il parametro x è una variabile locale e che ha vita solo dentro la funzione. Se infatti si provasse a chiamare dal main (quindi fuori la funzione), Python darebbe errore.

[10]:
```
dir()
```

[10]:
```
['In',
 'Out',
 '_',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
```

```
 '__package__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_i10',
 '_i2',
 '_i3',
 '_i4',
 '_i5',
 '_i6',
 '_i7',
 '_i8',
 '_i9',
 '_ih',
 '_ii',
 '_iii',
 '_oh',
 'a',
 'aumenta_di_due',
 'exit',
 'get_ipython',
 'n',
 'quit',
 'stampa_quadrato_e_cubo']
```

La **x** non compare nella lista di variabili che si ottengono con la funzione di built-in **dir()**.

[11]:
```
# Provando a chiamare x dal main si ottiene un messaggio di errore
print(x)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_6668\648499532.py in <module>
      1 # Provando a chiamare x dal main si ottiene un messaggio di errore
----> 2 print(x)

NameError: name 'x' is not defined
```

3 Ritorno di valori

Le funzioni possono anche ritornare dei valori. Ad esempio, la funzione math.sqrt(4) ritorna 2. Possiamo rifare l'esercizio guida di questa unità definendo le funzioni **quadrato()** e **cubo()** che restituiscono rispettivamente il quadrato e il cubo di un numero.

[18]:
```
def quadrato(x):
    """Funzione che restituisce il quadrato di un numero"""
    return x**2

def cubo(x):
    """Funzione che restituisce il cubo di un numero"""
    return x**3

a = -1
a2 = quadrato(a)
print("Il quadrato di", a, "è", a2)
a3 = cubo(a)
print("Il cubo di", a, "è", a3)
```

```
Il quadrato di -1 è 1
Il cubo di -1 è -1
```

Non deve sorprendere che abbiamo potuto risolvere lo stesso problema in due modi diversi. Il bello della programmazione è che possiamo usare la fantasia avendo molta libertà di scelta: dal nome delle variabili alle tecniche di implementazione. A volte possiamo stabilire a priori quale strada sia migliore di un'altra, altre volte lo si capisce strada facenda e in altri casi risulta difficile discernere a priori quale sia la strada migliore. A proposito di piú strade possibili, vediamo un'ulteriore implementazione del nostro problema, stavolta scrivendo la funzione **quadrato_e_cubo()** che ritorna sia il quadrato che il cubo. Python, infatti, a differenza di altri linguaggi di programmazione, permette di ritornare piú di un valore.

[19]:
```
def quadrato_e_cubo(x):
    """Ritorna il quadrato e il cubo di x."""
    x2 = x**2
    x3 = x**3
    return x2, x3

b = 2
b2, b3 = quadrato_e_cubo(b)
print("Il quadrato di", b, "è", b2)
print("Il cubo di", b, "è", b3)
help(quadrato_e_cubo)
```

```
Il quadrato di 2 è 4
Il cubo di 2 è 8
Help on function quadrato_e_cubo in module __main__:

quadrato_e_cubo(x)
    Ritorna il quadrato e il cubo di x.
```

Prima di concludere, osserviamo che una funzione che non ritorna alcunché, in realtà ritorna il valore indefinito None.

[20]:
```
prova = stampa_quadrato_e_cubo(-2)
print("La variabile di prova vale", prova)
type(prova)
```

```
Il quadrato di -2 è 4
Il cubo di -2 è -8
La variabile di prova vale None
```

[20]:
```
NoneType
```

3.1 Funzioni con parametri opzionali... e qualcosa in piú sulle stringhe

Possiamo creare delle funzioni con parametri opzionali. Per capire questo argomento, partiamo con un esempio definendo la funzione **quadrato_opzionale()**.

[21]:
```
def quadrato_opzionale(x = 2):
    return x**2

y = quadrato_opzionale(3)
print(y)
y = quadrato_opzionale()
print(y)
```

```
9
4
```

Nel primo caso abbiamo stampato il quadrato di 3, ovvero 9. Ma nel secondo caso abbiamo invocato la funzione **quadrato_opzionale()** senza alcun parametro attuale. Questo vuol dire che il parametro formale x assume il valore di default 2 espresso nell'intestazione e infatti viene stampato il valore 4, il quadrato di 2. Si tenga bene a mente che i parametri opzionali devono essere messi in coda agli altri parametri, onde evitare spiacevoli errori.

Per capire meglio l'argomento, vediamo un altro esempio. Iniziamo a scrivere la funzione **stampa_quadrato(n)** che stampa un quadrato cavo di asterischi di lato n. Prima di partire, però, abbiamo bisogno di aggiungere qualche informazione sulle stringhe. Abbiamo detto che le stringhe sono sequenze di caratteri racchiusi tra apici, doppi apici o tripli apici se vogliamo andare a capo. Abbiamo anche detto che in Python usiamo l'operatore * per effettuare la moltiplicazione. Anche se siamo soliti a pensare a una moltiplicazione fra numeri, Python offre la possibilità di moltiplicare le stringhe! Vediamo questo concetto con il seguente esempio:

[22]:
```
num1 = 2.0
num2 = 5/2
prod = num1*num2    # una normale moltiplicazione
print(prod)
s1 = "ciao "
print(2*s1)    # la stringa s1 viene moltiplicata per 2
s2 = 'sì '
print(s2*3)    # mentre la stringa s2 viene moltiplicata per 3
```

```
5.0
ciao ciao
sì sì sì
```

Prima di passare alla nostra funzione **stampa_quadrato()** abbiamo bisogno di un ultimo ingrediente: la concatenazione fra stringhe. È possibile concatenare due stringhe grazie all'operatore + che, come per la precedente moltiplicazione, possiamo pensare a una generalizzazione dell'addizione. Ma vediamo subito qualche esempio.

[23]:
```
s3 = "Buon"
s4 = "Natale"
print(s3)
print(s4)
s5 = s3 + s4
print(s5)
```

```
Buon
Natale
BuonNatale
```

La stringa *s5* è effettivamente la concatenazione della stringa *s3* ed *s4* ma non è un proprio bello spettacolo. Per avere la stringa "Buon Natale" dovremmo inserire uno spazio tra *s3* ed *s4*. Ma allora possiamo realizzare una concatenazione multipla dove tra le stringhe *s3* ed *s4* inseriamo una stringa contenente il carattere di spazio: ' ' oppure " "

[24]:
```
s5 = s3 + ' ' + s4
print(s5)
```

```
Buon Natale
```

Anche per le stringhe possiamo usare l'operatore +=

[25]:
```
s5 += " e felice anno nuovo!"
print(s5)
```

```
Buon Natale e felice anno nuovo!
```

A questo punto possiamo pensare al cuore della nostra funzione **stampa_quadrato()**: per $n > 1$ avremo il lato superiore e inferiore che consisteranno in una stringa di n asterischi che si ottiene stampando la stringa $n *' *'$. La parte centrale sarà una stringa formata da un asterisco piú $n-2$ spazî seguita da un ultimo spazio, il che si ottiene come $' *' + (n-2) *' ' + ' *'$. Una tale stringa si dovrà ripetere per $n-2$ volte e ciò si realizza con un ciclo *for*. Bisogna anche considerare i casi particolari in cui $n = 0$ (non si stampa alcunché) ed $n = 1$ (si stampa soltanto un asterisco.

[26]:
```
def stampa_quadrato(n):
    """Stampa un quadrato cavo di asterischi di lato n."""
    if n == 0:
        return
    elif n == 1:
        print('*')
```

```
    else:
        print(n*'*')
        for k in range(n - 2):
            print('*' + (n - 2)*' ' + '*')
        print(n*'*')

help(stampa_quadrato)
stampa_quadrato(0)
print()
stampa_quadrato(1)
print()
stampa_quadrato(2)
print()
stampa_quadrato(3)
print()
stampa_quadrato(4)
print()
```

```
Help on function stampa_quadrato in module __main__:

stampa_quadrato(n)
    Stampa un quadrato cavo di asterischi di lato n.


*

**
**

***
* *
***

****
*  *
*  *
****
```

Infine, osserviamo che se $n = 0$ si esce subito dalla funzione grazie alla parola chiave *return*.

Ma volendo possiamo fare ancora meglio stampando a video un rettangolo cavo di dimensioni m ed n. Possiamo inizialmente definire una funzione **stampa_rettangolo(m, n)** che si comporterà come la funzione **stampa_quadrato(n)** quando $m = n$. Vediamo una prima implementazione.

[27]:
```
def stampa_rettangolo(m, n):
    """Stampa un rettangolo cavo di asterischi di dimensioni m ed n."""
    if m == 0 or n == 0:
```

```
        return
    elif m == 1:
        print('*'*n)
    elif m != 1 and n == 1:
        print('*\n'*m)
    else:
        print(n*'*')
        for k in range(m - 2):
            print('*' + (n - 2)*' ' + '*')
        print(n*'*')


help(stampa_rettangolo)
print("stampa_rettangolo(0, 0)")
stampa_rettangolo(0, 0)
print()
print("stampa_rettangolo(3, 0)")
stampa_rettangolo(3, 0)
print()
print("stampa_rettangolo(0, 3)")
stampa_rettangolo(0, 3)
print()
print("stampa_rettangolo(1, 1)")
stampa_rettangolo(1, 1)
print()
print("stampa_rettangolo(1, 3)")
stampa_rettangolo(1, 3)
print()
print("stampa_rettangolo(3, 1)")
stampa_rettangolo(3, 1)
print("stampa_rettangolo(2, 2)")
stampa_rettangolo(2, 2)
print()
print("stampa_rettangolo(3, 5)")
stampa_rettangolo(3, 5)
print()
print("stampa_rettangolo(5, 5)")
stampa_rettangolo(5, 5)
print()
```

```
Help on function stampa_rettangolo in module __main__:

stampa_rettangolo(m, n)
    Stampa un rettangolo cavo di asterischi di dimensioni m ed n.

stampa_rettangolo(0, 0)
```

```
stampa_rettangolo(3, 0)

stampa_rettangolo(0, 3)

stampa_rettangolo(1, 1)
*

stampa_rettangolo(1, 3)
***

stampa_rettangolo(3, 1)
*
*
*

stampa_rettangolo(2, 2)
**
**

stampa_rettangolo(3, 5)
*****
*   *
*****

stampa_rettangolo(5, 5)
*****
*   *
*   *
*   *
*****
```

Nel caso in cui m sia diverso da 1 ed n sia uguale a 1 dobbiamo stampare m asterischi in colonna. Avremmo potuto usare un ciclo for, mentre invece, grazie alla moltiplicazione di stringhe, ripetiamo per m volte il carattere '*' seguito dal carattere di escape che serve per andare a capo, che è proprio quello che vogliamo! L'unica cosa è che andremo a capo una volta in piú del dovuto (terminata la riga m) ed è per questo che, a differenza degli altri casi di esempio, non abbiamo aggiunto il comando print(). Vediamo nell'ultimo esempio che l'invocazione di **stampa_rettangolo(5, 5)** effettivamente genera un quadrato di lato 5.

Come abbiamo già detto, non esiste una maniera univoca di programmare. Infatti una volta che abbiamo a disposizione la funzione **stampa_rettangolo(m, n)** potremmo riscrivere la funzione **stampa_quadrato(n)** nel seguente modo:

[28]:
```
def stampa_quadrato(n):
    """Stampa un quadrato cavo di lato n usando la funzione␣
 ↪stampa_rettangolo(m, n)."""
    stampa_rettangolo(n, n)
```

```
help(stampa_quadrato)
stampa_quadrato(0)
print()
stampa_quadrato(1)
print()
stampa_quadrato(2)
print()
stampa_quadrato(3)
print()
stampa_quadrato(4)
print()
```

```
Help on function stampa_quadrato in module __main__:

stampa_quadrato(n)
    Stampa un quadrato cavo di lato n usando la funzione stampa_rettangolo(m,
n).


*

**
**

***
* *
***

****
*  *
*  *
****
```

Osserviamo alcune cose. La prima è che ridefinendo **stampa_quadrato(n)** perdiamo la vecchia funzione, appunto perché è stata sovrascritta. Ci accorgiamo di questo dal docstring che è cambiato in base alla funzione piú recente. Una seconda cosa è che la nuova funzione **stampa_quadrato(n)** chiama al suo interno la funzione **stampa_rettangolo(m, n)** con $m = n$. Indipendentemente dal valore di m, si tratta di una soluzione interessante perché mette in luce un fatto molto importante, ovvero che possiamo invocare delle funzioni all'interno di altre funzioni. L'ultima cosa da osservare è il fatto che avremmo potuto semplificare il codice qui sopra con un ciclo, per esempio un for:

[29]:
```
for k in range(5):
    stampa_quadrato(k)
    print()
```

```
*

**
**

***
* *
***

****
*  *
*  *
****
```

Un'altra possibilità è quella di scrivere la funzione **stampa_rettangolo(m, n)** con il parametro n opzionale. Se omesso, si intende $m = n$ e siamo nel caso della stampa di un quadrato.

[30]:
```
def stampa_rettangolo(m, n = None):
    """Stampa un rettangolo cavo di asterischi di dimensioni m ed n.
       Se il parametro n viene omesso, si intende m = n e dunque stampa
       un quadrato cavo di asterischi di lato m"""
    if n is None:
        n = m

    if m == 0 or n == 0:
        return
    elif m == 1:
        print('*'*n)
    elif m != 1 and n == 1:
        print('*\n'*m)
    else:
        print(n*'*')
        for k in range(m - 2):
            print('*' + (n - 2)*' ' + '*')
        print(n*'*')

help(stampa_rettangolo)
for k in range(5):
    stampa_rettangolo(k)
    print()
stampa_rettangolo(4, 7)
```

```
Help on function stampa_rettangolo in module __main__:

stampa_rettangolo(m, n=None)
    Stampa un rettangolo cavo di asterischi di dimensioni m ed n.
    Se il parametro n viene omesso, si intende m = n e dunque stampa
```

```
    un quadrato cavo di asterischi di lato m


*

**
**

***
* *
***

****
*  *
*  *
****

*******
*     *
*     *
*******
```

Abbiamo assegnato a n il valore di default None. Per verificare se n è pari a None usiamo l'istruzione:

if n is None:

3.2 Ricorsione

La ricorsione è quando una funzione chiama sé stessa. Due classici esempî sono il calcolo del fattoriare ed i numeri di Fibonacci. Vediamo ambo i casi sia nel caso ricorsivo che non ricorsivo.

[31]:
```
def fattoriale_non_ricorsivo(n):
    if n == 0:
        return 1
    else:
        fatt = 1
        for k in range(1, n + 1):
            fatt*= k
        return fatt

def fattoriale_ricorsivo(n):
    if n == 0:
        return 1
    else:
        return n*fattoriale_ricorsivo(n - 1)

n = 4
fatt = fattoriale_non_ricorsivo(n)
print(n, "! = ", fatt, sep = '')
```

```
fatt = fattoriale_ricorsivo(n)
print(n, "! = ", fatt, sep = '')
```

```
4! = 24
4! = 24
```

Osserviamo che abbiamo sfruttato la seguente proprietà del fattoriale:

$$n! = n \cdot (n-1) \cdot ... \cdot 1 = n \cdot (n-1)!$$

Osserviamo altresí che se avessimo scritto print(n, "! =", fatt), avremmo visualizzato quando segue:

[32]:
```
print(n, "! =", fatt)
```

```
4 ! = 24
```

Si viene cioè a creare uno spazio tra il 4 e il punto esclamativo. Un modo per togliere lo spazio è quello di annullare lo spazio separatore tra i vari argomenti della funzione di *built-in* print(), che di default è appunto uno spazio. Agendo sul parametro (opzionale) della funzione di *built-in* print(), possiamo modificare il carattere separatore, come si vede dai seguenti esempî:

[33]:
```
print("Ciao,", "come", "ti", "chiami?")
print("Ciao,", "come", "ti", "chiami?", sep = '*')
print("Ciao,", "come", "ti", "chiami?", sep = '')
```

```
Ciao, come ti chiami?
Ciao,*come*ti*chiami?
Ciao,cometichiami?
```

Come si vede dall'ultima print(), ponendo sep = '', si annulla lo spazio separatore, ma allora nell'istruzione print(n, "! =", fatt, sep =") bisogna aggiungere uno spazio dopo il segno di uguale.

Passiamo ai numeri di Fibonacci. Qui vale la seguente regola ricorsiva:

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$.

[34]:
```
def fibonacci_non_ricorsivo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_prec_2 = 0
        fib_prec_1 = 1
        for k in range(2, n + 1):
            fib = fib_prec_1 + fib_prec_2
            fib_prec_2 = fib_prec_1
            fib_prec_1 = fib
        return fib
```

```
def fibonacci_ricorsivo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_ricorsivo(n - 1) + fibonacci_ricorsivo(n - 2)

for n in range(12):
    print("F(", n, ") = ", fibonacci_non_ricorsivo(n), sep = '')
    print("F(", n, ") = ", fibonacci_ricorsivo(n), sep = '')
```

```
F(0) = 0
F(0) = 0
F(1) = 1
F(1) = 1
F(2) = 1
F(2) = 1
F(3) = 2
F(3) = 2
F(4) = 3
F(4) = 3
F(5) = 5
F(5) = 5
F(6) = 8
F(6) = 8
F(7) = 13
F(7) = 13
F(8) = 21
F(8) = 21
F(9) = 34
F(9) = 34
F(10) = 55
F(10) = 55
F(11) = 89
F(11) = 89
```

Seguono tre esempî matematici: la risoluzione dell'equazione lineare $ax + b = 0$, la risoluzione dell'equazione di secondo grado $ax^2 + bx + c = 0$ e infine un'approssimazione della funzione esponenziale. Ma procediamo con ordine.

[35]:
```
def equazione_primo_grado(a, b):
    if a == 0.0 and b == 0.0:
        print("Equazione indeterminata")
    elif a == 0.0 and b!= 0.0:
        print("Equazione impossibile")
    else:
```

```
        x = -b/a
        print("x =", x)


equazione_primo_grado(0.0, 0.0)
equazione_primo_grado(0.0, 1.0)
equazione_primo_grado(2.0, -2.0)
```

```
Equazione indeterminata
Equazione impossibile
x = 1.0
```

Per quanto riguarda la risoluzione dell'equazione di secondo grado $ax^2 + bx + c = 0$, possiamo scrivere la funzione **equazione_secondo_grado()**. Nella trattazione teorica di tale argomento si esclude a priori il caso $a = 0$ in quanto ci si riconduce a un'equazione lineare la cui risoluzione è stata appena trattata. D'altra parte, nulla impedisce all'utente di chiamare la funzione con $a = 0$. Pertanto, possiamo prevedere tale eventualità all'interno della funzione chiamando, se necessario, la funzione **equazione_primo_grado()**. In altre parole siamo di nuovo in un caso di funzione che al suo interno chiama un'altra funzione.

[36]:
```
def equazione_secondo_grado(a, b, c):
    if a == 0.0:
        equazione_primo_grado(b, c)
    else:
        D = b**2 - 4*a*c
        if D > 0.0:
            print("Due soluzioni reali e distinte")
            x1 = (-b + D**.5)/(2*a)
            print("x1 =", x1)
            x2 = (-b - D**.5)/(2*a)
            print("x2 =", x2)
        elif D == 0.0:
            print("Due soluzioni reali e coincidenti")
            x = -b/(2*a)
            print("x1 =", x)
            print("x2 =", x)
        else:
            print("Due soluzioni complesse coniugate")
            A = -b/(2*a)
            B = (-D)**.5/(2*a)
            print("x1 =", A, "+ i", B)
            print("x1 =", A, "- i", B)


equazione_secondo_grado(0, 0, 0)
print()
equazione_secondo_grado(0, 0, 1)
print()
```

```
equazione_secondo_grado(0, 2, 3)
print()
equazione_secondo_grado(1, -5, 6)
print()
equazione_secondo_grado(1, -2, 1)
print()
equazione_secondo_grado(1, -2, 10)
```

```
Equazione indeterminata

Equazione impossibile

x = -1.5

Due soluzioni reali e distinte
x1 = 3.0
x2 = 2.0

Due soluzioni reali e coincidenti
x1 = 1.0
x2 = 1.0

Due soluzioni complesse coniugate
x1 = 1.0 + i 3.0
x1 = 1.0 - i 3.0
```

Facciamo notare che per calcolare la radice quadrata del discriminante **D** abbiamo usato il comando $D**.5$ che è una maniera abbreviata di $D**0.5$, ovvero $D^{\frac{1}{2}}$, ovvero $\sqrt{D}$. In alternativa avremmo potuto usare la funzione di built-in come **pow(D, .5)** oppure importare il pacchetto math per poi scrivere **math.sqrt(D)**.

Nel nostro ultimo esempio sulle funzioni vogliamo proporre un'approssimazione per il calcolo della funzione esponenziale $y = e^x$. A tale scopo sfruttiamo lo sviluppo di Mac-Laurin

$$e^x = \sum_{k=0}^{+\infty} \frac{x^k}{k!}$$

Possiamo riscrivere lo sviluppo come un polinomio di Mac-Laurin di ordine n piú un resto:

$$e^x = \sum_{k=0}^{n} \frac{x^k}{k!} + o\left(x^n\right).$$

Indicando con $P_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!}$ il polinomio di Mac-Laurin di ordine n, possiamo calcolare un'approssimazione della funzione esponenziale restituendo tale polinomio valutato in x e passando n come parametro:

[14]:
```
def exp(x, n):
    p = 1.0
    if n == 0:
        return p
    termine = 1
    for k in range(1, n + 1):
        termine*= x/k
        p+= termine
    return p

import math
print(math.e**0, exp(0, 0))
print(math.e**2, exp(2, 8))
```

```
1.0 1.0
7.3890560989306495 7.387301587301587
```

Come abbiamo imparato, possiamo rendere il parametro n opzionale:

[13]:
```
def exp(x, n = 8):
    p = 1.0
    if n == 0:
        return p
    termine = 1
    for k in range(1, n + 1):
        termine*= x/k
        p+= termine
    return p

import math
print(math.e**0, exp(0))
print(math.e**2, exp(2))
print(math.e**2, exp(2, 14))
```

```
1.0 1.0
7.3890560989306495 7.387301587301587
7.3890560989306495 7.3890560703259105
```

Un'altra possibilità è quella di fermarsi quando $|P_n(x) - P_{n-1}(x)| < \epsilon$, ove $\epsilon > 0$ è una tolleranza che si può passare come parametro opzionale.

[12]:
```
def exp(x, tol = 1e-3):
    p_prec = -1
    p = 1.0
    termine = 1
    k = 1
    while abs(p - p_prec) >= tol:
        p_prec = p
        termine*= x/k
```

```
        p+= termine
        k+= 1
    return p

import math
print(math.e**0, exp(0))
print(math.e**2, exp(2))
print(math.e**2, exp(2, tol = 1e-6))
```

```
1.0 1.0
7.3890560989306495 7.388994708994708
7.3890560989306495 7.3890560703259105
```

notebook6

March 4, 2023

1 Liste e stringhe

1.1 A cura di Mauro Maria Baldi

1.2 mauromaria.baldi@unimc.it

Introduciamo il concetto di lista, ovvero un elenco di elementi (ordinati) di qualsiasi tipo. Vediamo un esempio di lista che contiene gli elementi 1, 2 e 8:

[4]:
```
lista1 = [1, 2, 8]
print(type(lista1))
print(lista1)
```

```
<class 'list'>
[1, 2, 8]
```

In altri linguaggi di programmazione, il concetto di lista è associabile a quello di array. Ma a differenza di altri linguaggi di programmazione dove gli elementi dell'array debbono essere dello stesso tipo (si pensi al C, al C++ o al Java), Python ci permette di avere liste con elementi eterogenei; proprio come nel seguente esempio, dove inseriamo in una lista il numero 3, il valore True e la stringa "treno":

[5]:
```
lista2 = [3, True, "treno"]
print(lista2)
```

```
[3, True, 'treno']
```

La funzione append() permette di aggiungere in coda un elemento alla lista:

[6]:
```
lista2.append("aereo")
lista2
```

[6]:
```
[3, True, 'treno', 'aereo']
```

Si tratta di un risultato notevole dal momento che in altri linguaggi di programmazione (come ad esempio il C) le cose non sono cosí semplici e scontate. Lo stesso risultato si può ottenere anche in un altro modo, ovvero usando la concatenazione di stringhe. Supponiamo di voler aggiungere alla lista2 un altro mezzo di trasporto, ad esempio il tram. Possiamo procedere cosí:

[7]:
```
lista2+= ["tram"]
lista2
```

[7]:
```
[3, True, 'treno', 'aereo', 'tram']
```

Si ricorda che l'istruzione lista2+= ["tram"] è la contrazione dell'istruzione completa lista2 = lista2 + ["tram"]. In altre parole si concatena alla lista2 una lista costituita da un solo elemento: la stringa "tram".

Per esercizio, chiediamo all'utente di inserire un prezzo soglia e un insieme di costi di prodotti. Vogliamo inserire in una lista i costi inferiori al prezzo soglia e in un'altra lista i costi maggiori o uguali al prezzo soglia. Il programma termina non appena si inserisce un prezzo negativo. Per fare questo, usiamo due liste inizialmente vuote. In una aggiungeremo volta per volta con il comando append() i prezzi inferiori al prezzo soglia e nell'altra i prezzi magiori o uguali alla soglia. Per dichiarare una stringa vuota, si usano due parentesi quadre appiccicate, come nel seguente spezzone di codice che illustra una possibile realizzazione del problema.

[3]:
```
prezzoSoglia = eval(input("Inserisci il prezzo soglia: "))
listaInf = []
listaSup = []
prezzoAttuale = eval(input("Inserisci il prezzo di un prodotto: "))
while prezzoAttuale >= 0.0:
    if prezzoAttuale < prezzoSoglia:
        listaInf.append(prezzoAttuale)
    else:
        listaSup.append(prezzoAttuale)
    prezzoAttuale = eval(input("Inserisci il prezzo di un prodotto: "))
print("Prezzi inferiori alla soglia:", listaInf)
print("Prezzi maggiori o uguali alla soglia:", listaSup)
```

```
Inserisci il prezzo soglia: 10
Inserisci il prezzo di un prodotto: 8
Inserisci il prezzo di un prodotto: 5
Inserisci il prezzo di un prodotto: 12
Inserisci il prezzo di un prodotto: 20
Inserisci il prezzo di un prodotto: 4.5
Inserisci il prezzo di un prodotto: 10
Inserisci il prezzo di un prodotto: -3
Prezzi inferiori alla soglia: [8, 5, 4.5]
Prezzi maggiori o uguali alla soglia: [12, 20, 10]
```

Si noti che per inizializzare le due liste inizialmente vuote abbiamo usato le istruzioni listaInf = [] e listaSup = []. In alternativa, avremmo potuto usare la funzione di built in list() senza alcun argomento:

[1]:
```
listaVuota = list()
print(listaVuota)
```

```
[]
```

L'aggettivo "ordinati" tra parentesi nella definizione (informale) data a inizio capitolo significa che a ogni elemento è associato un indice che denota la posizione di quell'elemento nella lista. La convenzione di Python (come nel C, nel C++ e in Java) è che il primo elemento sia in posizione

0. Viceversa, in altri linguaggi di programmazione (come in Matlab), il primo elemento viene considerato in posizione 1. In altre parole, rimanendo nell'esempio della lista2, abbiamo che in posizione 0 c'è il numero 3, in posizione 1 c'è il valore booleano True e in posizione 2 c'è la stringa "treno".

Possiamo visualizzare un elemento in una particolare posizione postponendo al nome della lista l'indice della posizione desiderata fra parentesi quadre. Ad esempio, se vogliamo richiamare l'elemento in posizione 1 della lista1 scriviamo:

[8]:
```
lista1[1]
```

[8]:
```
2
```

Possiamo anche determinare la lunghezza di una stringa mediante la funzione di built-in len():

[9]:
```
len(lista2)
```

[9]:
```
5
```

A titolo di esempio ulteriore, creiamo una lista di articoli sciistici:

[10]:
```
articoli = ["sci", "scarponi", "guanti", "casco", "maschera", "tavola",␣
 ↪"giaccone", "pantaloni", "giaccone"]
print(articoli)
print(len(articoli))
```

```
['sci', 'scarponi', 'guanti', 'casco', 'maschera', 'tavola', 'giaccone',
'pantaloni', 'giaccone']
9
```

Per esercizio, stampiamo a video in riga i suddetti prodotti. Potremmo fare un ciclo for dove l'indice k varia tra 0 e 8, ovvero in range(9), ma 9 è proprio la lunghezza della stringa e dunque avremo range(len(articoli)). Il codice è il seguente:

[11]:
```
for k in range(len(articoli)):
    print(articoli[k])
```

```
sci
scarponi
guanti
casco
maschera
tavola
giaccone
pantaloni
giaccone
```

Python offre una soluzione piú elegante di quella appena presentata e può essere la seguente:

[12]:
```
for art in articoli:
    print(art)
```

```
sci
scarponi
guanti
casco
maschera
tavola
giaccone
pantaloni
giaccone
```

In altre parole, nella cella di codice qua sopra art **non** è un indice ma è l'articolo corrente. Vengono cioè visitati tutti gli elementi della lista senza passare dagli indici.

È possibile verificare se un elemento è nella lista grazie alla parola riservata *in*. Vediamo alcuni esempî:

[13]:
```
"guanti" in articoli
```

[13]:
```
True
```

Poiché la stringa "guanti" è nella lista articoli, l'istruzione dà True. In caso contrario viene restituito False, come nel seguente esempio:

[14]:
```
"crema solare" in articoli
```

[14]:
```
False
```

Per esercizio, creiamo un'altra lista di articoli e vediamo quali di questi articoli sono in comune con la lista di partenza.

[15]:
```
articoli2 = ["crema solare", "giaccone", "macchina fotografica", "tavola"]

for art in articoli2:
    if art in articoli:
        print(art, "è fra gli articoli iniziali")
    else:
        print(art, "non è fra gli articoli iniziali")
```

```
crema solare non è fra gli articoli iniziali
giaccone è fra gli articoli iniziali
macchina fotografica non è fra gli articoli iniziali
tavola è fra gli articoli iniziali
```

Un'altra interessante potenzialità offerta da Python è la concatenazione di liste, che opera in maniera simile alla concatenazione di stringhe che abbiamo visto in precedenza e che pure ricordiamo nel prossimo esempio.

[16]:
```
# Concatenazione di stringhe (ripasso)
s1 = "Porto"
print(s1)
s2 = "fino"
print(s2)
s3 = s1 + s2
print(s3)
l1 = ["Tizio", "Caio"]
print(l1)
l2 = ["e", "Sempronio"]
print(l2)
l3 = l1 + l2
print(l3)
```

```
Porto
fino
Portofino
['Tizio', 'Caio']
['e', 'Sempronio']
['Tizio', 'Caio', 'e', 'Sempronio']
```

Se volessimo unire gli elementi dell'ultima lista per ottenere la famosa frase "Tizio Caio e Sempronio", possiamo utilizzare la funzione join delle stringhe:

[17]:
```
' '.join(l3)
```

[17]:
```
'Tizio Caio e Sempronio'
```

In altre parole si uniscono in un'unica stringa gli elementi della lista separati dalla lista che precede il . prima del join. Si veda anche quest'altro esempio:

[18]:
```
s4 = '*'.join(l3)
print(s4)
s5 = ' <|> '.join(l3)
print(s5)
```

```
Tizio*Caio*e*Sempronio
Tizio <|> Caio <|> e <|> Sempronio
```

L'inverso della funzione join() è la funzione split(). Tale funzione si applica a una stringa e restituisce in una lista le parti della lista separate dall'argomento nella funzione split. Ecco alcuni esempî per chiarire meglio il concetto:

[19]:
```
s6 = "Sono una frase con spazî"
l4 = s6.split()
print(l4)
s7 = "Adesso*le*parole*sono*separate*da*asterischi"
l5 = s7.split('*')
print(l5)
```

```
['Sono', 'una', 'frase', 'con', 'spazî']
['Adesso', 'le', 'parole', 'sono', 'separate', 'da', 'asterischi']
```

Possiamo pensare alle stringhe come a degli array di caratteri che, come per le liste, partono dalla posizione 0. Prendiamo la stringa s6. Ci aspettiamo che in posizione 0 ci sia il carattere 'S', in posizione 1 il carattere 'o', eccetera. Verifichiamolo:

[20]:
```
print(s6[0])
print(s6[1])
```

```
S
o
```

Anche in questo caso possiamo ciclare sui caratteri in almeno due modi diversi:

[21]:
```
for k in range(len(s6)):
    print(s6[k])
print("\nOppure:\n")
for car in s6:
    print(car)
```

```
S
o
n
o

u
n
a

f
r
a
s
e

c
o
n

s
p
a
z
î

Oppure:

S
```

```
o
n
o

u
n
a

f
r
a
s
e

c
o
n

s
p
a
z
î
```

Sappiamo che per calcolare la lunghezza di una stringa o di una lista si utilizza la funzione di built-in len():

[22]:
```
len(s6)
```

[22]:
```
24
```

La stringa s6 ha 24 caratteri e dunque se volessimo accedere all'ultimo carattere senza contarli tutti a mano, dovremmo scrivere:

[23]:
```
s6[len(s6) - 1]
```

[23]:
```
'î'
```

Si faccia molta attenzione al -1. Infatti, dal momento che il primo carattere è in posizione 0, l'ultimo carattere sarà proprio in posizione len(s6) - 1 e non in posizione len(s6)! Una scrittura di questo tipo non è molto comoda. Fortunatamente Python ci offre un modo veloce e sicuro di accedere all'ultimo elemento della lista, semplicemente inserendo l'indice -1:

[24]:
```
s6[-1]
```

[24]:
```
'î'
```

In tal modo il penultimo carattere è quello in posizione -2, il terzultimo in posizione -3 e cosí via:

[25]:
```
print(s6[-1])
print(s6[-2])
print(s6[-3])
```

```
î
z
a
```

Per esercizio, realizziamo una funzione che stampa al contrario una stringa passata come argomento. Osserviamo che se la stringa è lunga, ad esempio, 4 caratteri, si dovrà stampare prima il carattere in posizione -1, poi quello in posizione -2, e infine quelli rispettivamente in posizione -3 e -4. Possiamo allora realizzare un ciclo for all'interno della funzione che stampi ogni carattere con l'opzione end = '' nella funzione print().

[26]:
```
def stampa_stringa_al_contrario(s):
    l = len(s)
    for k in range(-1, -(l + 1), -1):
        print(s[k], end = '')
    print()

stampa_stringa_al_contrario(s6)
stampa_stringa_al_contrario("ciao")
stampa_stringa_al_contrario("anna")    # Questa è una stringa palindroma!
```

```
îzaps noc esarf anu onoS
oaic
anna
```

In realtà potremmo fare ancora meglio: invece che stampare direttamente a video, potremmo creare una funzione che, data una stringa, restituisce una stringa invertita che, in un secondo momento, possiamo stampare a video. Per fare questo, partiamo da una stringa di servizio vuota e, visitando la stringa s come nella funzione precedente, a mano a mano che procediamo nel ciclo for, andiamo ad aggiungere un carattere alla stringa di servizio vuota. Alla fine del ciclo for, la stringa di servizio conterrà la stringa invertita e verrà restituita al programma principale.

[27]:
```
def inverti_stringa(s):
    auxString = ''
    l = len(s)
    for k in range(-1, -(l + 1), -1):
        auxString+= s[k]
    return auxString

auxString2 = inverti_stringa(s6)
print(auxString2)
auxString2 = inverti_stringa("ciao")
print(auxString2)
auxString2 = inverti_stringa("anna")
print(auxString2)
```

```
îzaps noc esarf anu onoS
oaic
anna
```

Quanto appena visto ci permette di introdurre lo slicing. Se, ad esempio, volessimo selezionare il secondo, terzo e quarto carattere della stringa s6, potremmo procedere come segue:

[28]:
```
s6[1:4]
```

[28]:
```
'ono'
```

La sintassi dentro le quadre è la seguente:

posizione di partenza inclusa : posizione di arrivo esclusa Pertanto in s6[1:4] si considerano i caratteri dalla posizione 1 inclusa (che per noi è la seconda perché, come abbiamo detto, Python parte dalla posizione 0) fino alla posizione 4 esclusa, ovvero fino alla posizione 3.

Proviamo a considerare i primo otto caratteri:

[29]:
```
s6[0:8]
```

[29]:
```
'Sono una'
```

Poiché, in questo caso, partiamo dall'inizio della stringa, possiamo omettere la posizione di partenza:

[30]:
```
s6[:8]
```

[30]:
```
'Sono una'
```

Possiamo anche considerare i caratteri con un certo passo. Per esempio, se volessimo selezionare i primi otto caratteri in posizione dispari (quindi per Python in posizione pari per la solita questione che lí si parte dalla posizione 0), potremmo scrivere:

[31]:
```
s6[0:8:2]
```

[31]:
```
'Sn n'
```

oppure

[32]:
```
s6[:8:2]
```

[32]:
```
'Sn n'
```

Se invece volessimo selezionare i caratteri da una certa posizione in poi, possiamo procedere, ad esempio, come segue:

[33]:
```
s6[15:]
```

[33]:
```
'con spazî'
```

Si badi bene al fatto che stavolta l'istruzione appena scritta è diversa dallo scrivere s6[15:-1]. In tal caso, infatti, per la convenzione precedentemente illustrata, la fine della stringa (posizione -1) verrebbe esclusa, come dimostra il seguente comando:

[34]:
```
s6[15:-1]
```

[34]:
```
'con spaz'
```

Per selezionare tutti i caratteri dalla posizione 15 (inclusa) alla fine (inclusa) con un passo di 2, possiamo scrivere:

[35]:
```
s6[15::2]
```

[35]:
```
'cnsaî'
```

1.3 Sottoliste e matrici

Come abbiamo detto, le liste possono contenere tipi di dati eterogenei. Ma allora possiamo inserire una lista dentro una lista. Proviamo, a titolo di esempio, ad aggiungere la sottolista ["linea", 4, "barrato", True] alla lista lista2:

[36]:
```
lista2.append(["linea", 4, "barrato", True])
lista2
```

[36]:
```
[3, True, 'treno', 'aereo', 'tram', ['linea', 4, 'barrato', True]]
```

Come sappiamo, in lista2[0] abbiamo 3, in lista2[1] abbiamo True e cosí via. Ma come facciamo ad accedere alla sottolista? Infatti, in lista2[5] abbiamo la sottolista:

[37]:
```
lista2[5]
```

[37]:
```
['linea', 4, 'barrato', True]
```

Possiamo accedere agli elementi della sottolista usando un altro livello di parentesi quadre:

[38]:
```
print(lista2[5][0])
print(lista2[5][1])    # eccetera
```

```
linea
4
```

Ma allora in questo modo possiamo realizzare delle matrici pensate come liste di liste. Ad esempio, possiamo creare una matrice identità 3×3 nel seguente modo:

[39]:
```
I3 = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
print(I3[0][0])
print(I3[1][0])
print(I3)
```

```
1
0
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

Come si vede, la stampa non è molto elegante. Un modo per ovviare a questo problema è quello di crearsi una funzione che stampi la matrice allineando opportunamente le righe e le colonne. Possiamo pensare di passare alla funzione non solo la matrice (o meglio, la lista di liste), ma anche le dimensioni che ci serviranno in un due cicli for annidati.

[41]:
```
import math
def stampaMatrice(M, numRighe, numColonne):
    for ii in range(numRighe):
        for jj in range(numColonne):
            print(M[ii][jj], end = "\t")
        print()
    print()

A = [[1, 2, 3], [4, 5, 6]]
B = [[math.sqrt(2), math.e], [math.pi, math.sin(1)]]

stampaMatrice(I3, 3, 3)
stampaMatrice(A, 2, 3)
stampaMatrice(B, 2, 2)
```

```
1       0       0
0       1       0
0       0       1

1       2       3
4       5       6

1.4142135623730951      2.718281828459045
3.141592653589793       0.8414709848078965
```

La soluzione appena proposta è piuttosto artigianale. Come vedremo, Python offre delle soluzioni piú rapide e comode per la gestione degli array e delle matrici (ovvero gli array bidimensionali) grazie al pacchetto *numpy*.

notebook7

March 6, 2023

1 Le tuple

1.1 Note a cura di Mauro Maria Baldi

1.2 mauromaria.baldi@unimc.it

Le tuple sono un tipo di dato offerto da Python che, a differenza delle liste, una volta inizializzato non può essere modificato. Si tratta, in altre parole, di ennuple non modificabili. Vediamo qualche esempio.

[8]:
```
t1 = (3, 2, 1, "via!") # L'inizializzazione avviene tra parentesi tonde
print(type(t1))
print(t1)
for k in range(len(t1)):
    print(t1[k], end = ' ')
print()
for el in t1:
    print(el, end = ' ')
print()
```

```
<class 'tuple'>
(3, 2, 1, 'via!')
3 2 1 via!
3 2 1 via!
```

Come si vede dall'esempio, possiamo accedere agli elementi della tupla come facevamo per le liste e in una tupla ci possono essere elementi di vario tipo, anche altre tuple o liste:

[7]:
```
t2 = (1, [2, 3], (4, 5, 6))
print(t2[0])
print(t2[1])
print(t2[2][2])
```

```
1
[2, 3]
6
```

Addirittura è possibile omettere le parentesi tonde per definire una tupla

[11]:
```
t3 = "Sono", "una", "tupla"
print(type(t3))
print(t3)
```

```
<class 'tuple'>
('Sono', 'una', 'tupla')
```

Questo meccanismo ci permette di effettuare lo scambio di due variabili in una maniera geniale che consiste in un'unica istruzione, mentre in altri linguaggi di programmazione si dovrebbe utilizzare una variabile di appoggio, come si può apprezzare nel seguente spezzone di codice:

[15]:
```
# Scambio "classico" tra due variabili a e b
a = 2
b = 4
print("Situazione prima dello scambio")
print("a = %d, b = %d" % (a, b))
tmp = a
a = b
b = tmp
print("Situazione dopo lo scambio")
print("a = %d, b = %d\n" % (a, b))

# Scambio rapido in stile Python
c = 3
d = 5
print("Situazione prima dello scambio")
print("c = %d, d = %d" % (c, d))
c, d = d, c
print("Situazione dopo lo scambio")
print("c = %d, d = %d" % (c, d))
```

```
Situazione prima dello scambio
a = 2, b = 4
Situazione dopo lo scambio
a = 4, b = 2

Situazione prima dello scambio
c = 3, d = 5
Situazione dopo lo scambio
c = 5, d = 3
```

Sempre secondo questa logica, per definire una tupla di un solo elemento, possiamo agire nei seguenti due modi:

[20]:
```
t4 = (1,)
print(type(t4))
print(t4)
t5 = 1,
print(type(t5))
```

```
print(t5)
```

```
<class 'tuple'>
(1,)
<class 'tuple'>
(1,)
```

L'istruzione t6 = (1) non sarebbe sbagliata ma non darebbe luogo a una tupla bensí a un intero in quanto (1) viene assimilato a una mini espressione con un uno tra due parentesi tonde, come si può facilmente verificare:

[21]:
```
prova = (1)
print(type(prova))
prova
```

```
<class 'int'>
```

[21]:
```
1
```

Possiamo altresí convertire una lista in una tupla grazie alla funzioni di built in *tuple()*. Ad esempio:

[23]:
```
l1 = ["Torino", "Milano", "Palermo"]
print(type(l1))
print(l1)
l1 = tuple(l1)
print(type(l1))
print(l1)
```

```
<class 'list'>
['Torino', 'Milano', 'Palermo']
<class 'tuple'>
('Torino', 'Milano', 'Palermo')
```

Come abbiamo detto, una tupla non è modificabile. Torniamo alla tupla t1 e proviamo a modificare l'ultimo elemento:

[24]:
```
print(t1)
t1[-1] = "pronti partenza via!"
```

```
(3, 2, 1, 'via!')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_6348\3688210191.py in <module>
      1 print(t1)
----> 2 t1[-1] = "pronti partenza via!"

TypeError: 'tuple' object does not support item assignment
```

Come ci aspettavamo, genera un messaggio di errore. In realtà esiste una maniera indiretta di modificare una tupla: come poco fa abbiamo convertito una lista in una tupla tramite la funzione di built-in *tuple()*, allo stesso modo possiamo convertire una tupla in una lista tramite la funzione di built-in *list()* precedentemente già incontrata. Ma allora possiamo dapprima trasformare la tupla t1 in una lista, a quel punto diventa possibile modificarne gli elementi e poi effettuare la conversione a tupla.

[25]:
```
print(t1)
t1 = list(t1)
t1[-1] = "pronti partenza via!"
print(t1)
t1 = list(t1)
t1 = tuple(t1)
print(t1)
t1 = list(t1)
```

```
(3, 2, 1, 'via!')
[3, 2, 1, 'pronti partenza via!']
(3, 2, 1, 'pronti partenza via!')
```

Come per le liste, possiamo creare delle tuple vuote in due modi:

[3]:
```
t6 = ()
print(type(t6))
print(t6)
t7 = tuple()
print(type(t7))
print(t7)
```

```
<class 'tuple'>
()
<class 'tuple'>
()
```

1.3 Retta passante per due punti

Per esercizio, realizziamo la funzione retta_per_due_punti(p1, p2) che riceve in ingresso due tuple p1 e p2 contenenti le coordinate di due punti (distinti) del piano cartesiano e restituisce la tupla (a, b, c) con i coefficienti della retta in forma implicita $ax + by + c = 0$.

Siano $P_1(x_1, y_1)$ e $P_2(x_2, y_2)$ le coordinate dei due punti. Se $x_1 = x_2 \wedge y_1 = y_2$ allora i due punti non sono coincidenti e restituiamo una tupla vuota, oltre a stampare a video un messaggio di errore che avvisi l'utente dell'inserimento sbagliato. Se $x_1 = x_2 \wedge y_1 \neq y_2$ allora la retta è parallela all'asse y ed ha equazione $x = x_1$ che in forma implicita diventa $x - x_1 = 0$ e dunque restituiamo la tupla $(1, 0, -x_1)$. Se $x_1 \neq x_2 \wedge y_1 = y_2$ allora la retta è parallela all'asse x ed ha equazione $y = y_1$ che in forma implicita diventa $y - y_1 = 0$ e dunque restituiamo la tupla $(0, 1, -y_1)$. In tutti gli altri casi, la retta non è parallela agli assi e quindi possiamo usare l'equazione

$$\frac{y - y_1}{y_2 - y_1} = \frac{x - x_1}{x_2 - x_1}.$$

Per portarla in forma implicita, iniziamo a moltiplicare membro a membro per la quantità (per ipotesi non nulla) $(y_2 - y_1)(x_2 - x_1)$ in modo tale da ottenere

$$(x_2 - x_1)(y - y_1) = (y_2 - y_1)(x - x_1).$$

Portando tutto a un unico membro otteniamo

$$(y_2 - y_1)(x - x_1) - (x_2 - x_1)(y - y_1) = 0,$$

ovvero

$$(y_2 - y_1)x - (x_2 - x_1)y - x_1(y_2 - y_1) + y_1(x_2 - x_1) = 0,$$

che a sua volta diviene

$$(y_2 - y_1)x + (x_1 - x_2)y - x_1y_2 + x_2y_1 = 0.$$

Possiamo quindi restituire la tupla $(y_2 - y_1,\ x_1 - x_2,\ -x_1y_2 + x_2y_1)$.

[7]:
```
def retta_per_due_punti(p1, p2):
    x1 = p1[0]; y1 = p1[1]
    x2 = p2[0]; y2 = p2[1]
    if x1 == x2 and y1 == y2:
        print("Attenzione: i due punti sono coincidenti!")
        return tuple()
    elif x1 == x2:
        return (1, 0, -x1)
    elif y1 == y2:
        return (0, 1, -y1)
    else:
        return (y2 - y1, x1 - x2, -x1*y2 + x2*y1)
```

Osserviamo che al primo elif avremmo potuto scrivere elif x1 == x2 and y1!= y2: Tuttavia, non è stato necessario specificare y1!= y2 in quanto abbiamo già superato il primo ramo della selezione (quello dove $x_1 = x_2$ e $y_1 = y_2$) e quindi sicuramente o $x_1 \neq x_2$ o $y_1 \neq y_2$.

A questo punto possiamo realizzare una funzione che stampi a video l'equazione della retta passante per P_1 e per P_2. Purtroppo, se t è la tupla di ritorno dalla funzione, non basta scrivere qualcosa del tipo "{}x + {}y + {} = 0".format(t[0], t[1], t[2]). Infatti se il coefficiente della y (e/o il termine noto) è negativo (per esempio 2), si leggerà la sgradevole (e tecnicamente sbagliata) scritta +-2y invece che -2y. Se qualche coefficiente è nullo, si rischia di leggere qualcosa del tipo 0x oppure + 0. Tali scritture, al contrario di prima, non sono sbagliate, ma non sono molto eleganti da vedere. Possiamo costruirci una stringa inizialmente vuota che, mano a mano che visitiamo i coefficienti conservati nella tupla di ritorno dalla precedente funzione, aggiungiamo dei pezzi opportunamente adattati a seconda dei valori. Un possibile modo è il seguente:

[13]:
```
def stampa_equazione_retta(p1, p2):
    t = retta_per_due_punti(p1, p2)
    if t == ():
```

```
        return
    a = t[0]
    b = t[1]
    c = t[2]
    aNullo = False
    strEq = ''
    if a == 0.0:
        auxString = ''
        aNullo = True
    elif a == 1.0:
        auxString = "x "
    elif a == -1.0:
        auxString = "-x "
    else:
        auxString = str(a) + "x "
    strEq+= auxString
    if b == 0.0:
        auxString = ''
    elif b == 1.0:
        if aNullo:
            auxString = "y "
        else:
            auxString = '+ y '
    elif b == -1.0:
        auxString = "- y "
    elif b > 0:
        if aNullo:
            auxString = str(b) + 'y '
        else:
            auxString = "+ " + str(b) + 'y '
    else:
        auxString = str(b) + 'y '
    strEq+= auxString
    if c == 0.0:
        strEq+= "= 0"
    elif c > 0:
        strEq+= "+ " + str(c) + " = 0"
    else:
        strEq+= str(c) + " = 0"
    auxString = "Equazione retta per P1({}, {}) e P2({}, {}):".format(p1[0],␣
↪p1[1], p2[0], p2[1])
    print(auxString)
    print(strEq)
```

Facciamo qualche prova per testare la nostra funzione. Iniziamo a considerare il caso banale di due punti allineati. Potremmo, ad esempio, scrivere p1 = (1, 1); p2 = (1, 1). Ma Python offre un'altra grande comodità, ovvero quella di fare due assegnazioni in un colpo solo grazie all'istruzione pi =

p2 = (1, 1).

[19]:
```
p1 = p2 = (1, 1)
stampa_equazione_retta(p1, p2)
```

```
Attenzione: i due punti sono coincidenti!
```

Proviamo con una retta parallela all'asse x:

[20]:
```
p1 = (1, 2)
p2 = (2, 2)
stampa_equazione_retta(p1, p2)
```

```
Equazione retta per P1(1, 2) e P2(2, 2):
y -2 = 0
```

Ora proviamo con una retta parallela all'asse y:

[21]:
```
p1 = (-2, -2)
p2 = (-2, 3)
stampa_equazione_retta(p1, p2)
```

```
Equazione retta per P1(-2, -2) e P2(-2, 3):
x + 2 = 0
```

Altri esempî:

[22]:
```
p1 = (0, 0)
p2 = (1, 1)
stampa_equazione_retta(p1, p2)
p1 = (1, 2)
p2 = (3, 4)
stampa_equazione_retta(p1, p2)
p1 = (-1, -2)
p2 = (-3, -4)
stampa_equazione_retta(p1, p2)
```

```
Equazione retta per P1(0, 0) e P2(1, 1):
x - y = 0
Equazione retta per P1(1, 2) e P2(3, 4):
2x -2y + 2 = 0
Equazione retta per P1(-1, -2) e P2(-3, -4):
-2x + 2y + 2 = 0
```

1.4 Sistemi lineari 2×2

Quanto appena fatto ci permette di poter risolvere con un piccolo sforzo ulteriore un problema importante: la risoluzione dei sistemi 2×2. Partendo dal sistema

$$\begin{cases} a_1x + b_1y = c_1 \\ a_2x + b_2y = c_2, \end{cases}$$

il metodo di Cramer ci garantisce che il sistema è senz'altro risolubile se

$$\Delta := \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \neq 0.$$

In tal caso, posto

$$\Delta_x := \begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix} \quad \text{e} \quad \Delta_y := \begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix},$$

si ha che

$$x = \frac{\Delta_x}{\Delta} \quad \text{e} \quad y = \frac{\Delta_y}{\Delta}.$$

Se invece $\Delta = 0$, si possono verificare le seguenti due possibilità: - se $\Delta_x \neq 0$ oppure $\Delta_y \neq 0$ il sistema è impossibile; - altrimenti se $\Delta_x \neq 0$ e $\Delta_y \neq 0$ il sistema è indeterminato.

Ragionando in termini di rette, il caso di un sistema con una sola soluzione corrisponde al punto di intersezione di due rette incidenti; il caso di un sistema impossibile corrisponde a due rette parallele e infine il caso di un sistema indeterminato corrisponde a due rette parallele e coincidenti. Possiamo allora scrivere la funzione *risolvi_Cramer_2(t1, t2)* che riceve come parametri due tuple contenente i coefficienti (a, b, c) di un'equazione nella forma $ax + by = c$ del sistema. Dopodiché, possiamo testare la nostra funzione su tre semplici sistemi che contemplino i tre casi appena descritti. Infine, possiamo mettere insieme i varî pezzi risolvendo un problema di geometria analitica di trovare il punto di intersezione di due rette, ciascuna passante per due punti dati. Si faccia attenzione in questo caso che, poiché una retta in forma implicità è nella forma $ax + by + c = 0$ e, come abbiamo visto nella sezione precedente, a questa associata la tupla (a, b, c), la stessa equazione nel sistema assume la forma $ax + by = -c$ e dunque bisognerà passare alla funzione *risolvi_Cramer_2()* delle tuple nella forma $(a, b, -c)$.

[1]:
```
def risolvi_Cramer_2(t1, t2):
    a1 = t1[0]; b1 = t1[1]; c1 = t1[2]
    a2 = t2[0]; b2 = t2[1]; c2 = t2[2]
    D = a1*b2 - a2*b1
    Dx = c1*b2 - c2*b1
    Dy = a1*c2 - a2*c1
    if D!= 0:
        x = Dx/D
        y = Dy/D
        print("Sistema determinato")
        print("x =", x, "y =", y)
    elif Dx!= 0 or Dy!= 0:
        print("Sistema impossibile")
    else:
        print("Sistema indeterminato")
```

Iniziamo a testare la nostra funzione con il sistema

$$\begin{cases} 2x - 3y = 3 \\ x - y = 5, \end{cases}$$

per il quale ci aspettiamo la soluzione (12, 7).

[2]:
```
risolvi_Cramer_2((2, -3, 3), (1, -1, 5))
```

```
Sistema determinato
x = 12.0 y = 7.0
```

Passiamo adesso al sistema

$$\begin{cases} 2x - 4y = 8 \\ x - 2y = 3, \end{cases}$$

che è impossibile.

[3]:
```
risolvi_Cramer_2((2, -4, 8), (1, -2, 3))
```

```
Sistema impossibile
```

Proviamo anche con il sistema

$$\begin{cases} 2x - 4y = 8 \\ x - 2y = 4, \end{cases}$$

che è chiaramente indeterminato in quanto la prima equazione si ottiene moltiplicando per due ciascun membro della seconda.

[5]:
```
risolvi_Cramer_2((2, -4, 8), (1, -2, 4))
```

```
Sistema indeterminato
```

Concludiamo con l'ultimo esercizio: quello di trovare le coordinate del punto di intersezione della retta passante per i punti (0, 0) e (4, 2) con la retta passante per i punti (0, 2) e (4, 0). Ci aspettiamo di trovare il punto (2, 1).

[9]:
```
t1 = retta_per_due_punti((0, 0), (4, 2))
t2 = retta_per_due_punti((0, 2), (4, 0))

# Cambio del segno ai termini noto
t1 = list(t1)
t1[-1] = -t1[-1]
t2 = list(t2)
t2[-1] = -t2[-1]
risolvi_Cramer_2(t1, t2)
```

```
Sistema determinato
x = 2.0 y = 1.0
```

Si osservi che è stato necessario trasformare le tuple in liste per poter modificare il segno dei termini noti. Si osservi anche che, nonostante abbiamo passato alla funzione *risolvi_Cramer_2()* due liste invece che due tuple, il programma non ha dato errori perché all'interno andiamo semplicemente a leggere gli elementi in posizione 0, 1 e 2. Anche questo dimostra la grande flessibilità di Python: altri linguaggi di programmazione avrebbero giustamente dato degli errori.

Infine, ci preme sottolineare che gli esempi matematici riportati in questo capitolo hanno la limitata pretesa di essere meramente didattici. Sistemi piú complessi non possono senz'altro essere risolti con le funzioni qui presentati. Come vedremo, però, Python offre dei potenti pacchetti che permette la risoluzione di sistemi piú complessi nonché di problemi tipici di argomenti piú avanzati.

notebook8

March 14, 2023

1 Gli insiemi

1.1 Note a cura di Mauro Maria Baldi

1.2 mauromaria.baldi@unimc.it

Il prossimo tipo di dato presentato è quello degli insiemi, per il quale valgono le stesse regole e operazioni della teoria matematica degli insiemi, a cominciare dal fatto che non ci possono essere duplicati.

[4]:
```
sA = set([1, 2, 3])
print(type(sA))
print(sA)
sB = {4, 5}
print(type(sB))
print(sB)
sC = set((5, 6, 5))
print(type(sC))
print(sC)
```

```
<class 'set'>
{1, 2, 3}
<class 'set'>
{4, 5}
<class 'set'>
{5, 6}
```

Si osservi che sono stati definiti tre insiemi (sA, sB ed sC) in tre modi diversi. sA è stato definito partendo da una lista, sB è stato definito tramite un elenco tra parentesi graffe ed sC è stato definito partendo da una tupla. Si noti che in quest'ultimo caso è stato rimosso il duplicato 5 dal momento che, come è ben noto e come è appena stato detto, negli insiemi i duplicati non contano.

Ma allora un modo per togliere i duplicati da una lista (tupla) è quello di trasformare la lista (tupla) in un insieme per poi ritrasformarlo in una lista (tupla).

[9]:
```
l1 = [1, 2, 1, 3, 2, 4, 5, 4]
print("l1 prima dell'eliminazione dei duplicati:", l1)
l1 = list(set(l1))
print("l1 dopo l'eliminazione dei duplicati:", l1)
t1 = (1, 2, 1, 3, 2, 4, 5, 4)
```

```
print("t1 prima dell'eliminazione dei duplicati:", t1)
t1 = tuple(set(t1))
print("t1 dopo l'eliminazione dei duplicati:", t1)
```

```
l1 prima dell'eliminazione dei duplicati: [1, 2, 1, 3, 2, 4, 5, 4]
l1 dopo l'eliminazione dei duplicati: [1, 2, 3, 4, 5]
t1 prima dell'eliminazione dei duplicati: (1, 2, 1, 3, 2, 4, 5, 4)
t1 dopo l'eliminazione dei duplicati: (1, 2, 3, 4, 5)
```

È possibile effettuare varie operazioni sugli insiemi. Iniziamo con l'unione.

[14]:
```
print("sA =", sA)
print("sB =", sB)
print("sA U sB =", sA.union(sB))
```

```
sA = {1, 2, 3}
sB = {4, 5}
sA U sB = {1, 2, 3, 4, 5}
```

Vediamo adesso l'intersezione:

[16]:
```
sB.intersection(sC)
```

[16]:
```
{5}
```

Differenza:

[19]:
```
print(sC.difference(sB))
print(sB.difference(sC))
```

```
{6}
{4}
```

Non è possibile accedere agli elementi di un insieme come per le liste o le tuple perché non sono indicizzati. Un tentativo di accesso genera un messaggio di errore.

[20]:
```
sB[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_868\1879679326.py in <module>
----> 1 sB[0]

TypeError: 'set' object is not subscriptable
```

È però possibile verificare se un elemento appartiene all'insieme:

[21]:
```
risultato = 5 in {1, 2, 3}
print(risultato)
```

```
risultato = 5 not in {1, 2, 3}
print(risultato)
risultato = 2 in {1, 2, 3}
print(risultato)
risultato = 2 not in {1, 2, 3}
print(risultato)
```

```
False
True
True
False
```