SuperDuper Py 03

Ciao youtubers e benvenuti in SuperDuper PY una serie di tutorial pensati per chi è all'inizio con Python e sta cercando un qualcosa che non sia il classico tutorial tipo addominali in 7 minuti o che spieghi ogni singolo comando senza dare esempi o idee.

Oggi vedremo come funzionano i cicli **For** e come possiamo usarli nei nostri codici. Abbiamo visto cos'è una lista, e abbiamo accennato cos'è un dizionario che in pratica è una lista solo che ad ogni elemento non è associato solo un indice, ma è assegnato un nome e questo ci può aiutare quando ad esempio chiediamo all'utente di inserire una memoria nella calcolatrice: x = 2 o pi= 3.14.

Ha una struttura che può essere vista come:

memories = {"x":2, "pi":3.14}
e scrivendo print(memories[x]) ottengo 2 e così via.

Ora può capitare di avere liste molto lunghe e se ad esempio ho bisogno di sapere se un qualcosa è nella lista come abbiamo visto posso usare l'operatore "in". if "m" in alphabet, dove alfabet è una lista di caratteri ottengo true o false. E' ovviamente un operatore comodo "in", ma prima di questo si usava ad esempio un altro metodo che ci permetteva di controllare uno a uno tutti gli elementi della lista.

```
alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

charToFind = "m"
for i in range(len(alphabet)):
   if findChar == alphabet[i]:
       print(f"ho trovato {charToFind} alla posizione {i}")
       break
```

For si può usare in due modi, usando dei numeri che aumentano di volta in volta come in questo caso e quello che succede è che i ogni volta aumenta di 1, oppure può essere usato direttamente sulla lista e in questo ogni volta assume il valore dell'elemento.

```
findChar = "m"
for i in alphabet:
   if findChar == i:
       print(f"ho trovato {findChar}")
       break
```

Per avere la posizione della lista in cui ho trovato il carattere che stavo cercando posso usare altri modi, come ad esempio creare una variabile index = 0 che aumenta di 1 ad ogni ciclo.

```
    findChar = "m"
    index = 0
    for i in alphabet:
        if findChar == i:
            print(f"ho trovato {findChar} alla posizione {index}")
            break
        index += 1
```

In alternativa posso usare il metodo index della lista passandogli l'elemento.

```
    findChar = "m"
    for i in alphabet:
        if findChar == i:
                        print(f"ho trovato {findChar} alla posizione
{alphabet.index(i)}")
            break
```

Proviamo ora a creare l'agoritmo per cifrare una frase usando il cifrario di cesare.

Il cifrario di Cesare è una delle tecniche di cifratura più antiche e semplici. Deve il suo nome a Giulio Cesare, che, secondo Svetonio, lo usava nelle sue lettere private, specialmente in quelle inviate a Gaio Oppio, uno dei suoi più fidati luogotenenti. Nonostante fosse facilmente decifrabile, la sua efficacia era garantita dal fatto che, in quel periodo, la maggior parte delle persone non sapeva né leggere né scrivere, e quelle che potevano leggere non erano familiari con il concetto di cifratura.

Il cifrario di Cesare si basa su un sistema di sostituzione in cui ogni lettera del testo originale viene sostituita con una lettera posta a una certa distanza fissa nel normale ordine alfabetico. Ad esempio, con uno spostamento di 3 posizioni, la lettera A sarebbe sostituita dalla lettera D, la B dalla E, e così via. Una volta raggiunta la fine dell'alfabeto, lo spostamento riparte dall'inizio: ad esempio, con uno spostamento di 3, la lettera X sarebbe sostituita dalla A, la Y dalla B, e la Z dalla C.

Il valore dello spostamento, noto anche come "chiave", determina la sostituzione. Nel caso delle lettere inviate da Cesare a Oppio, lo spostamento era di 3 posizioni.

Quello che dobbiamo fare quindi è per ogni carattere del testo, prendere prendere la cifra che ha l'indice+3. Ovviamente l'alfabeto ha una sua lunghezza quindi dobbiamo fare in modo che l'indice + shift delle lettere che sono più lunghe devono ricominciare da zero.

```
    def caesar_cipher(text, shift):
        """
        Cifra un testo usando il cifrario di Cesare.

        :param text: testo da cifrare
        :param shift: numero di posizioni da shiftare ogni lettera
        :return: testo cifrato
        """

        encryptedText = ""
        alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',
                    'v', 'w', 'x', 'y', 'z']

        for char in text:
            # Se il carattere fa parte dell'alfabeto calcola il suo indice
            # altrimenti lo lascia invariato
            if char in alphabet:
                index = alphabet.index(char.lower())
                   # Se lo shift fa andare fuori dalla lunghezza alfabeto,
ricomincia dall'inizio
                if index + shift >= len(alphabet):
                    index = (index + shift) - len(alphabet)
                else:
                    index += shift
                encryptedText += alphabet[index]

            else:
                encryptedText += char

        return encryptedText


    # Test della funzione
    text = input("Inserisci il testo da cifrare: ")
    shift = int(input("Inserisci lo shift (numero di posizioni da
spostare): "))
    encrypted = caesar_cipher(text, shift)
    print(f"Testo cifrato: {encrypted}")
```

Pronti per la Sfida? Create la funzione che può decifrare il testo! Mettete il video in pausa e prendetevi il tempo che vi serve.

```
    def descript(text, shift):
        decryptedText = ""
        alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',
                    'v', 'w', 'x', 'y', 'z']
        for char in text:
            if char in alphabet:
                index = alphabet.index(char.lower())
                if index - shift < 0:
                    index = (index - shift) + len(alphabet)
                else:
                    index -= shift
                decryptedText += alphabet[index]
```

```
            else:
                decryptedText += char
        return decryptedText
```

Ovviamente quello che dovete fare adesso è rendere il codice interattivo, dando la possibilità all'utente di scrivere **decifra** in modo da decifrare il codice. Insomma, il solito esercizio per abituarsi a rendere il programma user friendly.

Un altro modo per usare i cicli for è con i numeri. Supponiamo ad esempio che l'utente inserisca un numero e che il programma possa indovinare se quel numero è il cubo di un numero.

Per capirlo ad esempio posso fare un ciclo for che prova tutti i numeri da zero fino al numero che ho inserito e elevare al cubo ogni numero fino a che non ho trovato la sua radice cuba se esiste, oppure avverto l'utente che quel numero non è un cubo perfetto.

```
# Trova se un numero è un cubo perfetto
cube = 27
for guess in range(cube + 1):
    print(f"guess = {guess} cube of guess = {guess ** 3}")
    if guess ** 3 == cube:
        print(f"Cube root of {cube} is {guess}")
        break
    elif guess ** 3 > cube:
        print(f"{cube} is not a perfect cube")
        break
```

Ovviamente quello che interessa a noi è dare una sorta di interattività al nostro codice in modo che non si limiti a durare solo un shot e per farlo, come abbiamo fatto nel video precedente, inseriamo il codice in una funzione.

```
def findPerfectCube(cube):
    """
    Trova se un numero è un cubo perfetto elevando al cubo
    tutti i numeri da 0 al numero passato nei parametri.
    :param cube: numero da controllare
    :return: nothing
    """
    if cube.isdigit():
        cube = int(cube)
        for guess in range(cube + 1):
            print(f"guess = {guess} cube of guess = {guess ** 3}")
            if guess ** 3 == cube:
                print(f"{guess}: Cube root of {cube} is {guess}")
                break
            elif guess ** 3 > cube:
                print(f"{cube} is not a perfect cube")
                break
```

```
        else:
            print(f"sei sicuro che {cube} sia un numero?")
```

A questo punto creiamo la nostra App, scrivendo:

```
def mainApp():
    isWannaContinue = True
    while isWannaContinue:
        cube = input("Enter an integer: ")
        if cube.lower() == "exit":
            isWannaContinue = False
            break
        else:
            findPerfectCube(cube)


if __name__ == "__main__":
    mainApp()
```

if ___name___ == “___main___” è una dicitura speciale che serve a far sapere a python durante il run del file cosa deve fare.

```
Enter an integer: 3
guess = 0 cube of guess = 0
guess = 1 cube of guess = 1
guess = 2 cube of guess = 8
3 is not a perfect cube
Enter an integer: 8
guess = 0 cube of guess = 0
guess = 1 cube of guess = 1
guess = 2 cube of guess = 8
2: Cube root of 8 is 2
```

Come vedete il programma va avanti, però non considera ad esempio che 15.625 è il cubo di 2.5! Questo lo limita a trovare solo i numeri interi.

Posso provare a creare un loop che aumenta guess di una cifra più piccola di 1, come 0.5 o ancora meglio 0.01, in questo modo provo tutti i numeri fino a quando non trovo un numero che sia vicino abbastanza al numero al cubo che ho inserito. Per capire se un numero è abbastanza vicino mi basta fare la sottrazione di (cube - (guess**)) e se dico che è inferiore a 0.1 significa che beh non è proprio quello il numero, però grosso modo mi ci sono avvicinato.

Questo approccio si chiama “brute force” e in pratica funziona provando tutti i numeri fino a quando non trovo il risultato che mi serve.

In questo caso, non posso usare il ciclo for, perchè mi permette di inserire solo interi nel suo range e sommare 0.01 può essere un'operazione più lunga del range.

Per trovare la radice cubica di un numero, possiamo fare ad esempio che invece di andare avanti con un intero, usiamo ad esempio 0.01 in questo modo, probabilmente non troverò la cifra ma la cifra che più gli si avvicina.

```
    def findApprox(cube):
        try:
            cube = float(cube)
            guess = 1.0
            for _ in range(int(cube)):
                guess += 0.01
                print(f"guess = {guess} cube of guess = {guess ** 3}")
                if guess ** 3 == cube:
                    print(f"{guess}: Cube root of {cube} is {guess}")
                    break
                elif guess ** 3 > cube:
                    print(f"{cube} is not a perfect cube")
                    break
        except ValueError:
            print(f"sei sicuro che {cube} sia un numero?")
```

```
Enter an integer: 15.625
guess = 1.01 cube of guess = 1.0303010000000001
guess = 1.02 cube of guess = 1.0612080000000002
guess = 1.03 cube of guess = 1.092727
guess = 1.04 cube of guess = 1.124864
guess = 1.05 cube of guess = 1.1576250000000001
guess = 1.06 cube of guess = 1.191016
guess = 1.07 cube of guess = 1.225043
guess = 1.08 cube of guess = 1.2597120000000002
guess = 1.09 cube of guess = 1.2950290000000002
guess = 1.1 cube of guess = 1.3310000000000004
guess = 1.11 cube of guess = 1.3676310000000003
guess = 1.12 cube of guess = 1.4049280000000004
guess = 1.1300000000000001 cube of guess = 1.4428970000000005
guess = 1.1400000000000001 cube of guess = 1.4815440000000004
guess = 1.1500000000000001 cube of guess = 1.5208750000000004
```

In questo caso ho bisogno che il loop vada avanti, come nel loop che ho creato per il main, quindi devo usare un While.

```
def findApproxCube(cube):
    """
    Trova se un numero è un cubo perfetto elevando al cubo
    tutti i numeri da 0 al numero passato nei parametri.
    :param cube: numero da controllare
    :return: nothing
```

```
    """
    try:
    # trasformo la stringa in un numero
    cube = float(cube)
    isContinue = True
    # definisco la tolleranza
    acceptableError = 0.1
    # definisco il valore di partenza
    guess = 0.0
    # definisco l'incremento
    increment = 0.01
    # definisco il numero di tentativi
    num_guesses = 0

    # cerca una risposta abbastanza vicina e si assicura
    # di non aver saltato accidentalmente il limite abbastanza vicino
    while abs(guess**3 - cube) >= acceptableError and guess <= cube:
        print(f"{abs(guess ** 3 - cube)} >= {acceptableError} and {guess} <=
{cube} = {abs(guess**3 - cube) >= acceptableError and guess <= cube}")
        print(f"guess: {guess} += {increment} = {guess + increment}")
        guess += increment
        num_guesses += 1
    print('num_guesses =', num_guesses)
    if abs(guess**3 - cube) >= acceptableError:
        print('Failed on cube root of', cube, "with these parameters.")
    else:
        print(guess, 'is close to the cube root of', cube)
except ValueError:
    print(f"sei sicuro che {cube} sia un numero?")
```

Ho inserito nel codice dei print in modo da farvi vedere come il programma riesce a calcolare se è vicino o meno al numero al cubo.

```
Enter an integer: 15.625
15.625 >= 0.1 and 0.0 <= 15.625 = True
guess: 0.0 += 0.01 = 0.01
15.624999 >= 0.1 and 0.01 <= 15.625 = True
guess: 0.01 += 0.01 = 0.02
15.624992 >= 0.1 and 0.02 <= 15.625 = True
guess: 0.02 += 0.01 = 0.03
```

Come vedete calcolo quanto sono vicino alla soluzione e siccome guess è 0.0 - il numero al cubo che ho inserito è maggiore della tolleranza e è uguale al numero di partenza quindi vado a sommare a guess l'incremento 0.01 e continuo ad andare avanti fino a che non ho trovato un risultato.

```
0.18675100000016975 >= 0.1 and 2.489999999999991 <= 15.625 = True
guess: 2.489999999999991 += 0.01 = 2.4999999999999907
num_guesses = 250
2.4999999999999907 is close to the cube root of 15.625
```

Quello che posso notare è che in effetti ci è arrivato molto vicino però ha fatto 250 tentativi. Se gli chiedo di calcolare se 10.000 è un cubo perfetto usa almeno 1.000.000 di tentativi prima di arrivare alla conclusione che non c'è una risposta.

Se pensate a come cercate un nome sul dizionario, o sull'elenco del telefono, non vi mettete a sfogliare tutte le pagine fino a che non avete trovate quello che cercate. Di solito, aprite l'elenco o il libro a metà e a quel punto decidete se quello che state cercando è prima o dopo, se è prima aprirete di nuovo a metà solamente la parte del dizionario e così via.

Piuttosto che scorrere tutte le pagine quello che fate è creare due margini. Inizialmente il margine basso è A e il margine alto è Z. Aprendolo a metà arrivate alla N, se la parola è fra la A e la N, N diventa il nuovo magine alto e continuerete la ricerca dividendo in due ovvero E e se la parola comincia con E, continuate a fare la stessa cosa spostando il margine basse alla E e il margine alto alla F e così via fino a che non avete trovato la parola.

Ovviamente richiede un certo numero di tentativi, ma saranno sicuramente, entro una certa cifra, molti meno che non sfogliare tutte le pagine.

```
    def bisection_sqrt(S, tolerance=1e-10):
        """Calcola la radice cuba di S usando il metodo della
bisezione."""

        # Definisci gli estremi dell'intervallo
        lowBoundary = 0
        highBoundary = max(1, S)  # Considera il caso in cui S < 1
        num_guesses = 0
        while (highBoundary - lowBoundary) > tolerance:
            guess = (lowBoundary + highBoundary) / 2  # Punto medio
dell'intervallo

            if guess ** 3 < S:
                lowBoundary = guess
            else:
                highBoundary = guess
            num_guesses += 1
        return (lowBoundary + highBoundary) / 2, num_guesses


    # Test della funzione
    S = 15.625
    root, num_guesses = bisection_sqrt(S)
    print(f"Radice quadrata di {S} approssimata con bisezione: {root} in
{num_guesses}")
```

Se provo a cercare se 15.625 è il cubo perfetto di un certo numero ad esempio lo trovo in 38 tentativi.

Sfida! Come vi sarete accorti l'esempio che vi ho fatto vedere non dice se un certo numero è il cubo perfetto di qualche altro numero. Si può modificare il codice in modo che usi l'algoritmo di bisection per farlo?

Bene siamo giunti al termine di questo tutorial. Oltre a ricordarvi che in descrizione trovate il link dove trovare i codici che abbiamo usato nel tutorial, se non lo avete fatto vi invito a iscrivervi al canale e a mettere un like.

GoodByte! Alla prossima