Errori frequenti

Matteo Ceccarello

Questo documento contiene una collezione di errori frequenti nell'appello del 26 gennaio 2024, con l'auspicio che non si ripetano nei prossimi appelli.

> Eseguite il codice!
>
> Sfruttate il fatto che il testo d'esame ha un programma di collaudo già scritto. Ogni volta che realizzate un metodo, eseguite il programma e leggete attentamente gli eventuali messaggi d'errore.

Table of contents

Confusione tra enunciato di assegnazione e uguaglianza logica

Ricordate che

```
a == 3 # Il risultato è True oppure False
```

è un enunciato che *verifica* se la variabile `a` è uguale a 3. Il risultato è quindi un valore booleano.

Invece

```
a = 3
```

è un enunciato che *assegna* il valore `3` alla variabile `a`. La differenza tra i due enunciati è enorme.

In particolare, nel compito un esercizio chiedeva di implementare un metodo `switchPlayer`. Astraendo i dettagli del compito la situazione è la seguente. Abbiamo tre variabili

- `p1`, una stringa, ad esempio `"O"`
- `p2`, una stringa, ad esempio `"X"`
- `player`, una stringa che assume alternativamente il valore di `p1` e di `p2`.

L'esercizio chiede di cambiare il valore di `player`: se il valore è `p1` allora bisogna *assegnare* il valore `p2` a `player`, altrimenti bisogna assegnare `p1`.

Questo si traduce nel seguente codice, che correttamente stampa `"X"`, essendo `player` inizializzata come `p1`.

Giusto

```
p1 = "O"
p2 = "X"
player = p1

if player == p1: # qui si usa == perchè è un confronto
    player = p2 # qui si usa = perchè è un'assegnazione
else:
    player = p1 # anche qui un singolo = perchè è un'assegnazione

print(player)
```

```
X
```

Il codice seguente invece *non fa nulla*: come vedete il valore di `player` rimane quello iniziale.

Sbagliato!

```
p1 = "O"
p2 = "X"
player = p1

if player == p1: # qui si usa == perchè è un confronto
    player == p2 # fare un confronto qui non ha senso!
else:
    player == p1 # anche qui non ha senso

print(player)
```

```
O
```

Attenzione quindi: anche se la differenza è di un singolo carattere la versione sbagliata del codice non fa quello che viene chiesto!

Visitare tutti gli elementi di una tabella

Due esercizi del compito richiedevano di visitare tutte le posizioni di una tabella rappresentata come una lista di liste (`self._board` nel testo del compito). Astraendoci nuovamente dai dettagli del compito, immaginiamo di avere una tabella `board` rappresentata come lista di lista di liste

```
board = [
    ["X", "O", " "],
    [" ", "X", "X"],
    ["O", "O", " "]
]
```

Di seguito riporto alcuni modi per visitare tutti gli elementi. In ogni caso servono due cicli innestati: uno per visitare tutte e uno per visitare tutti gli elementi di una riga.

Tip

- Quando dovete lavorare con tutti gli elementi di una lista probabilmente vi servirà un ciclo for;
- Quando vedete che dovete lavorare con tutti gli elementi di una tabella probabilmente vi serviranno due cicli for, uno dentro l'altro.

```
# Primo modo, ciclo for sugli elementi delle liste
for row_list in board:                                                    ①
    for element in row_list:                                              ②
        # Qui fate quello che dovete con `element`
        pass
```

① `row_list` è una lista che di volta in volta corrisponde alle righe della tabella `board`.
② `elementi` è una variabile che di volta in volta corrisponde a ciascun elemento della riga `row_list`.

```
# Secondo modo, usando gli indici
for row_index in range(len(board)):                                       ①
    for column_index in range(len(board[row_index])):                     ②
        element = board[row_index][column_index]                          ③
        # Qui fate quello che dovete con `element`
        pass
```

① Iterazione sul range di indici che vanno da 0 (incluso) a `len(board)`, ovvero il numero di righe della tabella (escluso)
② Iterazione sul range di indice che vanno da 0 (incluso) a `len(board[row_index])`, ovvero il numero di elementi nella riga corrente (escluso)
③ Accesso all'elemento della tabella alla riga `row_index` e colonna `column_index`. Attenzione all'uso delle parentesi quadrate per accedere all'elemento: usiamo una coppia di parentesi quadrate per ogni indice.

```
# Terzo modo, usando una variabile per la dimensione del lato
# della tabella e assumendo che sia quadrata
side = len(board)

for row_index in range(side):                                             ①
    for column_index in range(side):                                      ②
        element = board[row_index][column_index]
        # Qui fate quello che dovete con `element`
        pass
```

① Iterazione sugli indici da `0` (incluso) a `side` (escluso)
② Iterazione sugli indici da `0` (incluso) a `side` (escluso)

Accesso ai metodi e alle variabili di istanza di una classe

Quando interagiamo con le istanze di una classe la sintassi corretta per accedere alle variabili è la seguente, supponendo che `istanza` sia il nome della variabile che si riferisce all'instanza cui siamo interessati

```
istanza.nome_variabile
```

Similmente, per utilizzare un metodo dobbiamo usare questa sintassi

```
istanza.nome_metodo(argomento1, argomento2, eccetera)
```

In particolare, se stiamo scrivendo un metodo di una classe (come ad esempio negli esercizi del compito) spesso l'istanza di interesse è quella della stessa classe su cui stiamo lavorando, ovvero `self`. Quindi per accedere a una variabile di istanza della classe su cui si sta lavorando facciamo

```
self._nome_variabile # Ricordate che di solito le variabili di
                     # istanza iniziano con _ per segnalare che
                     # sono private.
```

e per i metodi utilizziamo la seguente sintassi

```
self.nome_metodo(argomento1, argomento2, eccetera)
```

ATTENZIONE: ricordate che anche se i metodi sono definiti così

```
class Esempio:
    def metodo(self, param1, param2):
        # Codice del metodo
        pass
```

quando invochiamo `metodo` da un `altro_metodo` la sintassi è questa

Giusto

```
def altro_metodo(self):
    # ...
    # altro codice che definisce arg1 e arg2
    # ...
    # Invochiamo il metodo che ci interessa, passando
    self.metodo(arg1, arg2)
```

ovvero `self` **non** fa parte degli argomenti passati esplicitamente al `metodo`, ma precede l'invocazione.

Il seguente codice è **sbagliato**

Sbagliato

```
def altro_metodo(self):
    # ...
    # altro codice che definisce arg1 e arg2
    # ...
    # Questa invocazione non funziona!
    metodo(self, arg1, arg2)
```

Non dobbiamo passare `self` come primo parametro dell'invocazione.

Errori di battitura

Come abbiamo avuto modo di dire più volte i computer sono estremamente pignoli. Per questo motivo Python considera tutti diversi i seguenti nomi

```
_nextPlayer
nextPlayer
nextplayer
nextPlaier
__nextPlayer
player
Player
```

Controllate sempre scrupolosamente di non aver fatto errori di battitura e riferitevi sempre

con il nome preciso delle variabili! In particolare, se incontrate errori come il seguente probabilmente avete fatto un errore di battitura.

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

Utilizzo di variabili non definite

Un altro motivo frequente dell'errore `NameError` menzionato sopra è il tentativo di accedere a variabili che non sono state definite da nessuna parte, o che non sono nell'ambito di visibilità del codice che cerca di accedervi.

! Nomi non definiti

In questo caso `z` non è definita, inutile cercare di usarla!

```
x = 2
print( x + z ) # z non è definito!
```

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

! Nomi non visibili

In questo caso `count` è stata definita in un altro metodo, non possiamo accedervi direttamente

```
class Example:
    def count_things(self):
        count = 10

    def print_count(self):
        self.count_things() # Invochiamo count_things
        print(count) # la variabile count non è visibile qui!!

ex = Example()
ex.print_count()
```

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

affinchè il codice funzioni dobbiamo fare in modo che `count_things` *restituisca* il conteggio.

```
class Example:
    def count_things(self):
        count = 10
        return count # restituiamo il conteggio

    def print_count(self):
        the_count = self.count_things() # Invochiamo count_things
        print(the_count) # qui ci stiamo riferendo a una variabile
                        # locale con lo stesso valore di count

ex = Example()
ex.print_count()
```

```
10
```