



Corso di Apprendistato Python

Mod. Python
Docente:
Tonino Petrulli



Operatori di stringa – introduzione

È ora di tornare a questi due operatori aritmetici: + e *.

Vogliamo mostrarvi che hanno una seconda funzione. Sono in grado di fare qualcosa di più di una semplice **addizione** e **moltiplicazione**.

Li abbiamo visti in azione quando i loro argomenti sono numeri (interi o float, non importa).

Ora vi mostreremo che possono gestire anche le stringhe, anche se in un modo molto specifico.

Concatenazione

Il segno + (più), applicato a due stringhe, diventa un **operatore di concatenazione**:

stringa + stringa

Si tratta semplicemente di **concatenare** (incollare) due stringhe in una sola. Naturalmente, come il suo fratello aritmetico, può essere usato più di una volta in un'espressione e in tale contesto si comporta secondo il binding sinistro.

A differenza del suo fratello aritmetico, l'operatore di concatenazione **non è commutativo**, cioè "ab" + "ba" non è uguale a "ba" + "ab".

Non dimenticate che se volete che il segno + sia un **concatenatore** e non un sommatore, dovete assicurarvi che **entrambi i suoi argomenti siano stringhe**.

Non è possibile mischiare i tipi di prodotto.

Operatori di stringa – introduzione

Questo semplice programma mostra il segno + nel suo secondo utilizzo:

```
fnam = input("May I have your first name, please? ")
```

```
lnam = input("May I have your last name, please? ")
```

```
print("Thank you.")
```

```
print("\nYour name is " + fnam + " " + lnam + ".")
```

Nota: l'uso di + per concatenare le stringhe consente di costruire l'output in modo più preciso rispetto alla funzione print() pura, anche se arricchita con gli argomenti delle parole chiave end= e sep=.

.

Replicazione

Il segno * (asterisco), se applicato a una stringa e a un numero (o a un numero e a una stringa, poiché in questa posizione rimane commutativo), diventa un **operatore di replica**:

stringa * numero

numero * stringa

Replica la stringa lo stesso numero di volte specificato dal numero.

Ad esempio:

- "James" * 3 gives "JamesJamesJames"
- 3 * "an" gives "ananan"
- 5 * "2" (or "2" * 5) gives "22222" (not 10!)

Un numero minore o uguale a zero produce una **stringa vuota**.

Replicazione

Questo semplice programma "disegna" un rettangolo, utilizzando un vecchio operatore (+) in un nuovo ruolo:

```
print("+ " + 10 * "-" + "+")
```

```
print(("|" + " " * 10 + "|\\n") * 5, end="")
```

```
print("+ " + 10 * "-" + "+")
```

Provate a esercitarvi per creare altre forme

Conversione di tipo: str()

Sapete già come utilizzare le funzioni int() e float() per convertire una stringa in un numero.

Questo tipo di conversione non è a senso unico. È anche possibile **convertire un numero in una stringa**, il che è molto più semplice e sicuro; questo tipo di operazione è sempre possibile.

Una funzione in grado di farlo si chiama str():

```
str(numero)
```

Ancora il "triangolo rettangolo"

Ecco di nuovo il nostro programma "triangolo rettangolo":

```
leg_a = float(input("Input first leg length: "))  
leg_b = float(input("Input second leg length: "))  
print("Hypotenuse length is " + str((leg_a**2 + leg_b**2) **.5))
```

L'abbiamo modificata un po' per mostrarvi come funziona la funzione str(). Grazie a questa funzione, possiamo **passare l'intero risultato alla funzione print() come un'unica stringa**.

LAB-08

Tempo stimato: 5-10 minuti

Livello di difficoltà: Facile

Obiettivi

acquisire familiarità con l'immissione e l'emissione di dati in Python;
valutare semplici espressioni.

Scenario

Il vostro compito è quello di completare il codice per valutare i risultati di quattro operazioni aritmetiche di base. I risultati devono essere stampati nella console.

Testate il vostro codice: produce i risultati che vi aspettate?

Codice:

```
# inserire qui un valore per la variabile a
```

```
# inserire qui un valore per la variabile b
```

```
# produrre qui il risultato dell'addizione
```

```
# produrre qui il risultato della sottrazione
```

```
# emettere qui il risultato della moltiplicazione
```

```
# emettere qui il risultato della divisione
```

```
print("\nE' tutto, gente!")
```

LAB-09

Tempo stimato: 15-20 minuti

Livello di difficoltà: facile

Obiettivi

migliorare la capacità di utilizzare numeri, operatori e operazioni aritmetiche in Python;
utilizzando le capacità di formattazione della funzione print();
imparare a esprimere i fenomeni della vita quotidiana in termini di linguaggio di programmazione.

Scenario

Il vostro compito è preparare un semplice codice in grado di valutare l'**ora di fine** di un periodo di tempo, dato come numero di minuti (potrebbe essere arbitrariamente grande). L'ora di inizio è data come coppia di ore (0..23) e minuti (0..59). Il risultato deve essere stampato nella console.

Ad esempio, se un evento inizia alle **12:17** e dura **59 minuti**, terminerà alle **13:16**.

Non preoccupatevi di eventuali imperfezioni nel codice - va bene se accetta un tempo non valido - la cosa più importante è che il codice produca risultati validi per dati di input validi.

Verificate attentamente il vostro codice. Suggerimento: l'uso dell'operatore % può essere la chiave del successo.

Continua...

LAB-09

Codice:

```
hour = int(input("Starting time (hours): "))  
mins = int(input("Starting time (minutes): "))  
dura = int(input("Event duration (minutes): "))
```

Write your code here.

Dati del test

Ingresso campione: 12 17 59 Risultato atteso: 13:16	Ingresso campione: 23 58 642 Uscita prevista: 10:40	Ingresso campione: 0 1 2939 Uscita prevista: 1:0
---	---	--

Esercizio 1

Qual è l'output del seguente snippet?

```
x = int(input("Inserisci un numero: ")) # L'utente inserisce 2
```

```
print(x * "5")
```

Esercizio 2

Qual è l'output atteso del seguente snippet?

```
x = input("Inserisci un numero: ") # L'utente inserisce 2
```

```
print(type(x))
```

Confronto: operatore di uguaglianza

Domanda: due valori sono uguali?

Per porre questa domanda, si utilizza l'operatore ==

Non dimenticate questa importante distinzione:

= è un operatore di assegnazione, ad esempio $a = b$ assegna ad a il valore di b;

== è la domanda: *questi valori sono uguali?*; $a == b$ confronta a e b.

È un operatore binario con legame a sinistra. Ha bisogno di due argomenti e verifica se sono uguali

Confronto: operatore di uguaglianza

Esercizi

Ora poniamo alcune domande. Provate a indovinare le risposte.

Domanda n. 1: Qual è il risultato del seguente confronto?

`2 == 2`

Vero - ovviamente, 2 è uguale a 2. Python risponderà Vero

Domanda n. 2: Qual è il risultato del seguente confronto?

`2 == 2.`

Questa domanda non è facile come la prima. Fortunatamente, Python è in grado di convertire il valore intero nel suo equivalente reale e, di conseguenza, la risposta è Vero.

Domanda n. 3: Qual è il risultato del seguente confronto?

`1 == 2`

Dovrebbe essere facile. La risposta sarà (o meglio, è sempre) Falso.

Confronto: operatore di uguaglianza

L'operatore `==` (uguale a) confronta i valori di due operandi. Se sono uguali, il risultato del confronto è `True`. Se non sono uguali, il risultato del confronto è `False`.

Osservate il confronto di uguaglianza qui sotto: qual è il risultato di questa operazione?

```
var == 0
```

Si noti che non è possibile trovare la risposta se non si conosce il valore attualmente memorizzato nella variabile `var`.

Se la variabile è stata modificata più volte durante l'esecuzione del programma, o il suo valore iniziale è stato inserito dalla console, la risposta a questa domanda può essere data solo da Python e solo in fase di esecuzione.

Immaginate ora un programmatore che soffre di insonnia e che deve contare separatamente le pecore nere e quelle bianche, a patto che il numero di pecore nere sia esattamente il doppio di quelle bianche. La domanda sarà la seguente:

```
black_sheep == 2 * white_sheep
```

A causa della bassa priorità dell'operatore `==`, la domanda sarà trattata come equivalente a questa:

```
black_sheep == (2 * white_sheep)
```

Disuguaglianza: l'operatore *non uguale a* (!=)

Anche l'operatore != (non uguale a) confronta i valori di due operandi. La differenza è che se sono uguali, il risultato del confronto è False. Se non sono uguali, il risultato del confronto è True. Ora guardate il confronto tra disuguaglianze qui sotto: riuscite a indovinare il risultato di questa operazione?

```
var = 0 # Assegnazione di 0 a var  
print(var != 0)
```

```
var = 1 # Assegnare 1 a var  
print(var != 0)
```

Eseguite il codice e verificate se avevate ragione

Operatori di confronto: maggiore di

È anche possibile porre una domanda di confronto utilizzando l'operatore `>` (maggiore di).

Se si vuole sapere se ci sono più pecore nere che bianche, si può scrivere come segue:

```
black_sheep > white_sheep # Greater than
```

Il Vero lo conferma; il Falso lo nega.

Operatori di confronto: maggiore o uguale a

L'operatore *maggiore di* ha un'altra variante speciale e non rigida, ma è indicato in modo diverso rispetto alla notazione aritmetica classica: `>=` (maggiore di o uguale a).

Ci sono due segni successivi, non uno.

Entrambi questi operatori (rigoroso e non rigoroso), così come gli altri due discussi nella prossima sezione, sono operatori binari con vincolo a sinistra e la **loro priorità è maggiore** di quella indicata da `==` e `!=`.

Se vogliamo sapere se dobbiamo indossare o meno un cappello caldo, poniamo la seguente domanda:

```
centigrade_outside ≥ 0,0 # Maggiore o uguale a
```

Operatori di confronto: minore o uguale a

Come probabilmente avrete già intuito, gli operatori utilizzati in questo caso sono: l'operatore < (meno di) e il suo fratello non rigoroso: <= (meno di o uguale a).

Guardate questo semplice esempio:

velocità_attuale_macchina_veloce < 85 # Inferiore a

current_velocity_mph ≤ 85 # Inferiore o uguale a

Verificheremo se c'è il rischio di essere multati dalla polizia stradale (la prima domanda è rigorosa, la seconda no).

Tabella delle priorità

Priorità	Operatore	
1	+ , -	unario
2	**	
3	* , / , // , %	
4	+ , -	binario
5	< , <= , > , >=	
6	== , !=	

Condizioni ed esecuzione condizionale

Per prendere decisioni, Python offre un'istruzione speciale. Per la sua natura e la sua applicazione, è chiamata istruzione condizionale (o dichiarazione condizionale).

La prima forma di dichiarazione condizionale, che potete vedere qui sotto, è scritta in modo molto informale ma figurato:

if true_or_not:

do_this_if_true

Questa dichiarazione condizionale è composta dai seguenti elementi, strettamente necessari, solo in questo ordine:

- la parola chiave if;
- uno o più spazi bianchi;
- un'espressione (una domanda o una risposta) il cui valore sarà interpretato esclusivamente in termini di Vero (quando il suo valore è diverso da zero) e Falso (quando è uguale a zero);
- i due punti seguiti da una nuova linea;
- un'istruzione o un insieme di istruzioni rientrate (almeno un'istruzione è assolutamente necessaria); l'indentazione può essere ottenuta in due modi: inserendo un determinato numero di spazi (il consiglio è di utilizzare quattro spazi di indentazione), oppure utilizzando il carattere *tabulazione*;

nota: se c'è più di un'istruzione nella parte rientrata, l'indentazione deve essere la stessa in tutte le righe; anche se può sembrare uguale se si usano tabulazioni mescolate a spazi, è importante che tutte le indentazioni siano esattamente uguali - **Python 3 non permette di mescolare spazi e tabulazioni per l'indentazione.**

Esecuzione condizionale: l'istruzione if

Esempio:

```
if the_weather_is_good:  
    go_for_a_walk()  
have_lunch()
```

L'istruzione if-else:

```
if true_or_false_condition:  
    perform_if_condition_true  
else:  
    perform_if_condition_false
```

Esempio:

```
if the_weather_is_good:  
    go_for_a_walk()  
else:  
    go_to_a_theater()  
    have_lunch()
```

L'indentazione funziona allo stesso modo all'interno del ramo *else*:

Dichiarazioni annidate if-else

Discutiamo ora due casi speciali dell'enunciato condizionale.

Innanzitutto, si consideri il caso in cui l'istruzione posta dopo l'if sia un altro if.

Leggete cosa abbiamo in programma per questa domenica. Se il tempo è bello, andremo a fare una passeggiata. Se troviamo un bel ristorante, pranzeremo lì. Altrimenti mangeremo un panino. Se il tempo è brutto, andremo a teatro. Se non ci sono biglietti, andremo a fare shopping nel centro commerciale più vicino.

Scriviamo la stessa cosa in Python. Considerate attentamente il codice qui riportato:

```
if the_weather_is_good:
    if nice_restaurant_is_found:
        have_lunch()
    else:
        eat_a_sandwich()
else:
    if tickets_are_available:
        go_to_the_theater()
    else:
        go_shopping()
```

Dichiarazioni annidate if-else

Ecco due punti importanti:

Questo uso dell'istruzione if è noto come annidamento; ricordate che ogni else si riferisce all'if che si trova allo stesso livello di indentazione; è necessario saperlo per determinare come si accoppiano gli *if* e gli *else*;

considerare come l'indentazione migliori la leggibilità e renda il codice più facile da capire e da tracciare.

L'istruzione elif

Il secondo caso speciale introduce un'altra nuova parola chiave di Python: elif. Come probabilmente si sospetta, è una forma più breve di else if.

elif viene utilizzato per controllare più di una condizione e per fermarsi quando viene trovata la prima affermazione vera.

Il prossimo esempio assomiglia al nesting, ma le somiglianze sono minime. Anche in questo caso, cambieremo i nostri piani e li esprimeremo come segue: Se il tempo è bello, andremo a fare una passeggiata, altrimenti se avremo i biglietti, andremo a teatro, altrimenti se ci sono tavoli liberi al ristorante, andremo a pranzo; se tutto il resto fallisce, torneremo a casa e giocheremo a scacchi.

```
if the_weather_is_good:
    go_for_a_walk()
elif tickets_are_available:
    go_to_the_theater()
elif table_is_available:
    go_for_lunch()
else:
    play_chess_at_home()
```

L'istruzione elif

Il modo di assemblare le dichiarazioni *if-elif-else* successive è talvolta chiamato cascata.

Si noti ancora una volta come l'indentazione migliori la leggibilità del codice.

In questo caso è necessario prestare un'attenzione supplementare:

- non si deve usare else senza un if precedente;
- else è sempre l'ultimo ramo della cascata, indipendentemente dal fatto che si sia usato elif o meno;
- else è una parte opzionale della cascata e può essere omessa;
- se c'è un ramo else nella cascata, viene eseguito solo uno di tutti i rami;
- se non c'è un ramo else, è possibile che nessuno dei rami disponibili venga eseguito.

Esempi di istruzione condizionale

```
# Read two numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))
# Choose the larger number
```

```
if number1 > number2:
    larger_number = number1
else:
    larger_number = number2
```

Se nel ramo if o else c'è solo un'istruzione possiamo anche scriverla sulla stessa riga

```
if number1 > number2: larger_number = number1
else: larger_number = number2
```

```
# Print the result
```

```
print("The larger number is:", larger_number)
```

LAB-10

Tempo stimato: 5-15 minuti

Livello di difficoltà: Facile

Obiettivi

familiarizzare con la funzione `input()`;

familiarizzare con gli operatori di confronto in Python;

familiarizzare con il concetto di esecuzione condizionale.

Scenario

Lo [Spathiphyllum](#), più comunemente noto come giglio della pace o pianta della vela bianca, è una delle piante d'appartamento più popolari che filtra le tossine nocive dall'aria. Alcune delle tossine che neutralizza sono il benzene, la formaldeide e l'ammoniaca.

Immaginate che il vostro programma informatico ami queste piante. Ogni volta che riceve un input sotto forma della parola `Spathiphyllum`, urla involontariamente alla console la seguente stringa: "Lo `Spathiphyllum` è la pianta migliore in assoluto!".

Scrivere un programma che utilizzi il concetto di esecuzione condizionale, che prenda in input una stringa e che sia in grado di gestire il programma:

stampa sullo schermo la frase "Sì, lo `Spathiphyllum` è la pianta migliore in assoluto!" se la stringa inserita è "`Spathiphyllum`" (maiuscola)

stampa "No, voglio un grande `Spathiphyllum`!" se la stringa inserita è "`spathiphyllum`" (minuscolo)

stampa "`Spathiphyllum`! Non `[input]`!" altrimenti. Nota: `[input]` è la stringa presa come input.

Testate il vostro codice utilizzando i dati che vi abbiamo fornito.

LAB-11

Tempo stimato: 10-20 minuti

Livello di difficoltà: Facile/Medio

Obiettivi

Familiarizzare lo studente con:

utilizzando l'istruzione *if-else* per ramificare il percorso di controllo;
costruire un programma completo che risolva semplici problemi reali.

Scenario

C'era una volta una terra - una terra di latte e miele, abitata da gente felice e prospera. La gente pagava le tasse, naturalmente - la loro felicità aveva dei limiti. L'imposta più importante, chiamata *Imposta sul Reddito delle Persone Fisiche* (in breve *PIT*), doveva essere pagata una volta all'anno e veniva valutata in base alla seguente regola:

- se il reddito del cittadino non superava gli 85.528 talleri, l'imposta era pari al 18% del reddito meno 556 talleri e 2 centesimi (questo era il cosiddetto *sgravio fiscale*)
- se il reddito era superiore a questo importo, l'imposta era pari a 14.839 talleri e 2 centesimi, più il 32% dell'eccedenza rispetto agli 85.528 talleri.

Il vostro compito è quello di scrivere un calcolatore fiscale.

- Deve accettare un solo valore in virgola mobile: il reddito.
- Poi, dovrebbe stampare l'imposta calcolata, arrotondata ai talleri interi. C'è una funzione chiamata `round()` che si occuperà dell'arrotondamento; la troverete nel codice seguente

Continua...

LAB-11

```
income = float(input("Enter the annual income: "))
```

```
#
```

```
# Write your code here.
```

```
#
```

```
tax = round(tax, 0)
```

```
print("The tax is:", tax, "thalers")
```

Nota: questo Paese felice non restituisce mai denaro ai suoi cittadini. Se l'imposta calcolata è inferiore a zero, significa che non c'è nessuna imposta (l'imposta è uguale a zero). Tenetene conto durante i vostri calcoli.

Testate il vostro codice utilizzando i dati forniti in tabella

Ingresso campione: 10000 Risultato atteso: La tassa è: 1244,0 talleri	Ingresso campione: 100000 Risultato atteso: L'imposta è: 19470,0 talleri	Ingresso campione: 1000 Risultato atteso: La tassa è: 0,0 talleri	Ingresso campione: -100 Risultato atteso: La tassa è: 0,0 talleri
--	---	--	--

LAB-12

Tempo stimato:10-25 minuti

Livello di difficoltà: Facile/Medio

Obiettivi

Familiarizzare lo studente con:
utilizzando l'istruzione if-elif-else;
trovare la corretta applicazione di regole definite verbalmente;
codice di prova utilizzando input e output di esempio.

Scenario

Come sicuramente saprete, per motivi astronomici gli anni possono essere *bisestili* o *comuni*. I primi hanno una durata di 366 giorni, mentre i secondi hanno una durata di 365 giorni.

Dall'introduzione del calendario gregoriano (nel 1582), per determinare il tipo di anno si utilizza la seguente regola:

- se il numero dell'anno non è divisibile per quattro, si tratta di un *anno comune*;
- altrimenti, se il numero dell'anno non è divisibile per 100, si tratta di un *anno bisestile*;
- altrimenti, se il numero dell'anno non è divisibile per 400, si tratta di un *anno comune*;
- altrimenti è un *anno bisestile*.

Testate il vostro codice utilizzando i dati che vi abbiamo fornito.

Continua...

LAB-12

Osservate il codice :

```
year = int(input("Enter a year: "))
```

```
#
```

```
# Write your code here.
```

```
#
```

legge solo un numero di anno e deve essere completato con le istruzioni che implementano il test appena descritto.

Il codice deve produrre uno dei due messaggi possibili, ovvero Anno bisestile o Anno comune, a seconda del valore inserito.

Sarebbe opportuno verificare se l'anno inserito rientra nell'era gregoriana e, in caso contrario, emettere un avviso: Non rientra nel periodo del calendario gregoriano. Suggerimento: utilizzare gli operatori != e %.

Dati di test

Ingresso campione: 2000 Uscita prevista: Anno bisestile	Esempio di input: 2015 Risultato previsto: Anno comune	Ingresso campione: 1999 Risultato previsto: Anno comune	Ingresso campione: 1996 Uscita prevista: Anno bisestile	Ingresso campione: 1580 Risultato atteso: Non rientra nel periodo del calendario gregoriano
--	---	--	--	---

Esercizio 2

Qual è l'output del seguente snippet?

`x, y, z = 5, 10, 8`

`print(x > z)`

`print((y - 5) == x)`

Esercizio 3

Qual è l'output del seguente snippet?

`x, y, z = 5, 10, 8`

`x, y, z = z, y, x`

`print(x > z)`

`print((y - 5) == x)`

Esercizio 4

Qual è l'output del seguente snippet?

```
x = 10
```

```
if x == 10:
```

```
    print(x == 10)
```

```
if x > 5:
```

```
    print(x > 5)
```

```
if x < 10:
```

```
    print(x < 10)
```

```
else:
```

```
    print("else")
```


Esercizio 5

Qual è l'output del seguente snippet?

```
x = "1"
```

```
if x == 1:  
    print("one")  
elif x == "1":  
    if int(x) > 1:  
        print("two")  
    elif int(x) < 1:  
        print("three")  
    else:  
        print("four")  
if int(x) == 1:  
    print("five")  
else:  
    print("six")
```

Esercizio 6

Qual è l'output del seguente snippet?

```
x = 1  
y = 1.0  
z = "1"
```

```
if x == y:  
    print("one")  
if y == int(z):  
    print("two")  
elif x == y:  
    print("three")  
else:  
    print("four")
```

I cicli in Python | while

while there is something to do
do it

In generale, in Python, un ciclo può essere rappresentato come segue:

while conditional_expression:

instruction_one

instruction_two

instruction_three

:

:

instruction_n

I cicli in Python | while

- se si vuole eseguire più di un'istruzione all'interno di un while, è necessario (come per if) indentare tutte le istruzioni allo stesso modo;
- Un'istruzione o un insieme di istruzioni eseguite all'interno di un ciclo while è chiamata corpo del ciclo;
- se la condizione è False (uguale a zero) già quando viene testata per la prima volta, il corpo non viene eseguito nemmeno una volta (si noti l'analogia di non dover fare nulla se non c'è nulla da fare);
- il corpo dovrebbe essere in grado di cambiare il valore della condizione, perché se la condizione è Vera all'inizio, il corpo potrebbe essere eseguito continuamente all'infinito

Un ciclo infinito

Un ciclo infinito, chiamato anche loop infinito, è una sequenza di istruzioni in un programma che si ripete indefinitamente (loop infinito).

Ecco un esempio di ciclo che non riesce a terminare la sua esecuzione: mentre è vero:

while True:

```
    print("I'm stuck inside a loop.")
```

Questo ciclo stamperà all'infinito sullo schermo **"I'm stuck inside a loop."**

Per terminare il programma, basta premere *Ctrl-C* (o *Ctrl-Break* su alcuni computer). Questo causerà la cosiddetta eccezione `KeyboardInterrupt` e permetterà al programma di uscire dal loop.

Il ciclo while: altri esempi

Analizzate attentamente il programma. Osservate dove inizia il ciclo. Individuare il corpo del ciclo e scoprire come si esce da esso:

```
# Store the current largest number here.  
largest_number = -999999999  
# Input the first value.  
number = int(input("Enter a number or type -1 to stop: "))  
# If the number is not equal to -1, continue.  
while number != -1:  
    # Is number larger than largest_number?  
    if number > largest_number:  
        # Yes, update largest_number.  
        largest_number = number  
    # Input the next number.  
    number = int(input("Enter a number or type -1 to stop: "))  
# Print the largest number.  
print("The largest number is:", largest_number)
```

Il ciclo while: altri esempi

Vediamo un altro esempio che utilizza il ciclo while

A program that reads a sequence of numbers and counts how many numbers are even and how many are odd. The program terminates when zero is entered.

```
odd_numbers = 0
```

```
even_numbers = 0
```

```
number = int(input("Enter a number or type 0 to stop: "))
```

```
while number != 0:
```

```
    # Check if the number is odd.
```

```
    if number % 2 == 1:
```

```
        odd_numbers += 1
```

```
    else:
```

```
        even_numbers += 1.
```

```
    number = int(input("Enter a number or type 0 to stop: "))
```

```
# Print results.
```

```
print("Odd numbers count:", odd_numbers)
```

```
print("Even numbers count:", even_numbers)
```

Il ciclo while: altri esempi

Alcune espressioni possono essere semplificate senza modificare il comportamento del programma. Cercate di ricordare come Python interpreta la verità di una condizione e notate che queste due forme sono equivalenti:

while number != 0: e **while number:**

Anche la condizione che verifica se un numero è dispari può essere codificata in queste forme equivalenti:

if number % 2 == 1: e **if number % 2:**

LAB-13

Tempo stimato: 15 minuti

Livello di difficoltà: Facile

Obiettivi

Familiarizzare lo studente con:
utilizzando il ciclo while;
riflettere situazioni di vita reale nel codice del computer.

Scenario

Un mago junior ha scelto un numero segreto. Lo ha nascosto in una variabile chiamata `numero_segreto`. Vuole che tutti coloro che eseguono il suo programma giochino a *Indovina il numero segreto* e indovinino il numero che ha scelto per loro. Chi non indovina il numero rimarrà bloccato in un ciclo infinito per sempre! Purtroppo non sa come completare il codice.

Il vostro compito è quello di aiutare il mago a completare il codice nell'editor in modo tale che il codice

- chiederà all'utente di inserire un numero intero;
- utilizzerà un ciclo while;
- controllerà se il numero inserito dall'utente è uguale a quello scelto dal mago. Se il numero scelto dall'utente è diverso dal numero segreto del mago, l'utente dovrebbe vedere il messaggio "Ha ha! Sei bloccato nel mio loop!" e gli verrà richiesto di inserire nuovamente un numero. Se il numero inserito dall'utente corrisponde a quello scelto dal mago, il numero viene stampato sullo schermo e il mago pronuncia le seguenti parole: "Ben fatto, babbano! Ora sei libero".

Il mago conta su di voi! Non deludetelo.

LAB-13

Codice:

```
secret_number = 777
```

```
print(
    """
+=====+
| Welcome to my game, muggle!    |
| Enter an integer number        |
| and guess what number I've     |
| picked for you.                |
| So, what is the secret number? |
+=====+
    """)
```

guardate la funzione `print()`. Il modo in cui l'abbiamo usata qui è chiamato *stampa multilinea*. È possibile utilizzare le triple virgolette per stampare le stringhe su più righe, in modo da facilitare la lettura del testo o creare un design speciale basato sul testo. Sperimentate.

I cicli in Python | for

```
for i in range(100):  
    # do_something()  
    pass
```

- la parola chiave *for* apre il ciclo for; si noti che non c'è alcuna condizione dopo di essa; non è necessario pensare alle condizioni, poiché vengono verificate internamente, senza alcun intervento;
- qualsiasi variabile dopo la parola chiave *for* è la variabile di controllo del ciclo; conta i giri del ciclo e lo fa automaticamente;
- la parola chiave *in* introduce un elemento di sintassi che descrive l'intervallo di valori possibili da assegnare alla variabile di controllo;
- la funzione `range()` è responsabile della generazione di tutti i valori desiderati della variabile di controllo; nel nostro esempio, la funzione creerà (possiamo anche dire che alimenterà il ciclo) valori successivi dal seguente insieme: 0, 1, 2 ... 97, 98, 99; nota: in questo caso, la funzione `range()` inizia il suo lavoro da 0 e lo termina un passo (un numero intero) prima del valore del suo argomento;
- Si noti la parola chiave *pass* all'interno del corpo del ciclo: non fa nulla, è un'istruzione vuota - la mettiamo qui perché la sintassi del ciclo for richiede almeno un'istruzione all'interno del corpo (tra l'altro, *if*, *elif*, *else* e *while* esprimono la stessa cosa).

I cicli in Python | for

```
for i in range(10):
```

```
    print("The value of i is currently", i)
```

- il loop è stato eseguito dieci volte (è l'argomento della funzione range())
- il valore dell'ultima variabile di controllo è 9 (non 10, perché parte da 0 e non da 1)

La funzione range() può essere invocata con due argomenti e non con uno solo:

```
for i in range(2, 8):
```

```
    print("The value of i is currently", i)
```

In questo caso, il primo argomento determina il valore iniziale (primo) della variabile di controllo. L'ultimo argomento indica il primo valore che non verrà assegnato alla variabile di controllo.

Nota: la funzione range() accetta come argomenti solo numeri interi e genera sequenze di numeri interi. Riuscite a indovinare l'output del programma?

Ulteriori informazioni sul ciclo for e sulla funzione range() con tre argomenti

La funzione range() può accettare anche tre argomenti: date un'occhiata al codice:

```
for i in range(2, 8, 3):  
    print("The value of i is currently", i)
```

Il terzo argomento è un incremento: si tratta di un valore aggiunto per controllare la variabile a ogni giro del ciclo (come si può sospettare, il valore predefinito dell'incremento è 1).

Quante righe appariranno nella console e quali valori conterranno?

LAB-14

Tempo stimato: 5-15 minuti

Livello di difficoltà: Molto facile

Obiettivi

Familiarizzare lo studente con:
utilizzando il ciclo for;
riflettere situazioni di vita reale nel codice del computer.

Scenario

Sapete cos'è il Mississippi? È il nome di uno degli Stati e dei fiumi degli Stati Uniti. Il fiume Mississippi è lungo circa 2.340 miglia, il che lo rende il secondo fiume più lungo degli Stati Uniti (il più lungo è il fiume Missouri). È così lungo che una singola goccia d'acqua ha bisogno di 90 giorni per percorrere la sua intera lunghezza!

La parola *Mississippi* è usata anche per uno scopo leggermente diverso: *contare mississippi*.

Se non conoscete questa frase, vi spieghiamo cosa significa: si usa per contare i secondi.

L'idea alla base è che l'aggiunta della parola *Mississippi* a un numero quando si contano i secondi ad alta voce li fa sembrare più vicini all'ora dell'orologio, e quindi "un Mississippi, due Mississippi, tre Mississippi" impiegherà circa tre secondi effettivi di tempo! Viene spesso usato dai bambini che giocano a nascondino per assicurarsi che chi cerca faccia un conteggio onesto.

Il vostro compito è molto semplice: scrivete un programma che utilizzi un ciclo for per "contare a vuoto" fino a cinque. Dopo aver contato fino a cinque, il programma deve stampare sullo schermo il messaggio finale "Pronto o no, arrivo!".

LAB-14

```
import time
```

```
# Write a for loop that counts to five.
```

```
    # Body of the loop - print the loop iteration number and the word "Mississippi".
```

```
    # Body of the loop - use: time.sleep(1)
```

```
# Write a print function with the final message.
```

Si noti che il codice contiene due elementi che potrebbero non essere del tutto chiari in questo momento: la dichiarazione `import time` e il metodo `sleep()`. Ne parleremo presto.

Per il momento, ci basta sapere che abbiamo importato il modulo `time` e usato il metodo `sleep()` per sospendere per un secondo l'esecuzione di ogni successiva funzione `print()` all'interno del ciclo `for`, in modo che il messaggio inviato alla console assomigli a un vero e proprio conteggio.

Le istruzioni break e continue

Finora abbiamo trattato il corpo del ciclo come una sequenza indivisibile e inseparabile di istruzioni che vengono eseguite completamente a ogni giro del ciclo. Tuttavia, come sviluppatore, potreste trovarvi di fronte alle seguenti scelte:

- sembra che non sia necessario continuare il ciclo nel suo complesso; si dovrebbe astenersi dall'eseguire ulteriormente il corpo del ciclo e andare oltre;
- sembra che sia necessario avviare il giro successivo del ciclo senza completare l'esecuzione del giro corrente.

Python fornisce due istruzioni speciali per l'implementazione di entrambi i compiti. Per amor di precisione, diciamo che la loro presenza nel linguaggio non è necessaria: un programmatore esperto è in grado di codificare qualsiasi algoritmo senza queste istruzioni. Tali aggiunte non migliorano la potenza espressiva del linguaggio ma semplificano solo il lavoro dello sviluppatore. Queste due istruzioni sono:

- `break` - esce immediatamente dal ciclo e termina incondizionatamente l'operazione del ciclo; il programma inizia a eseguire l'istruzione più vicina dopo il corpo del ciclo;
- `continue` - si comporta come se il programma avesse improvvisamente raggiunto la fine del corpo; viene avviato il turno successivo e l'espressione di condizione viene testata immediatamente.

Entrambe le parole sono parole chiave.

.

Le istruzioni break e continue: altri esempi

Variante con break

```
largest_number = -99999999
counter = 0
while True:
    number = int(input("Enter a number or type -1 to end program: "))
    if number == -1:
        break
    counter += 1
    if number > largest_number:
        largest_number = number
if counter != 0:
    print("The largest number is", largest_number)
else:
    print("You haven't entered any number.")
```

.

Le istruzioni break e continue: altri esempi

Variante con continue

```
largest_number = -99999999
counter = 0
number = int(input("Enter a number or type -1 to end program: "))
while number != -1:
    if number == -1:
        continue
    counter += 1
    if number > largest_number:
        largest_number = number
    number = int(input("Enter a number or type -1 to end program: "))
if counter:
    print("The largest number is", largest_number)
else:
    print("You haven't entered any number.")
```

.

LAB-15

Tempo stimato: 10-20 minuti

Livello di difficoltà: Facile

Obiettivi

Familiarizzare lo studente con:
utilizzare l'istruzione break nei cicli;
riflettere situazioni di vita reale nel codice del computer.

Scenario

L'istruzione break viene utilizzata per uscire/terminare un ciclo.

Progettate un programma che utilizzi un ciclo while e chieda continuamente all'utente di inserire una parola, a meno che l'utente non inserisca "chupacabra" come parola segreta di uscita, nel qual caso dovrebbe essere stampato sullo schermo il messaggio "Sei uscito con successo dal ciclo" e il ciclo dovrebbe terminare.

Non stampare nessuna delle parole inserite dall'utente. Utilizzate il concetto di esecuzione condizionale e l'istruzione break.

.

LAB-16

Tempo stimato: 10-20 minuti

Livello di difficoltà: Facile

Obiettivi

Familiarizzare lo studente con:

utilizzare l'istruzione continue nei cicli;

riflettere situazioni di vita reale nel codice del computer.

Scenario

L'istruzione continue viene utilizzata per saltare il blocco corrente e passare all'iterazione successiva, senza eseguire le istruzioni all'interno del ciclo. Può essere utilizzato sia con il ciclo while che con il ciclo for. Il vostro compito qui è molto speciale: dovete progettare un mangiatore di vocali! Scrivete un programma che utilizzi:

- un ciclo for;
- il concetto di esecuzione condizionale (*if-elif-else*)
- L'istruzione continue

Il vostro programma deve:

- chiedere all'utente di inserire una parola;
- utilizzare `parola_utente = parola_utente.upper()` per convertire la parola inserita dall'utente in maiuscola;
- utilizzare l'esecuzione condizionale e l'istruzione continue per "mangiare" le seguenti vocali *A, E, I, O, U* dalla parola inserita;
- stampa sullo schermo le lettere non consumate, ciascuna su una riga separata.

LAB-16

```
# Prompt the user to enter a word
# and assign it to the user_word variable.
```

```
for letter in user_word:
    # Complete the body of the for loop.
```

Testate il vostro programma con i seguenti dati

Esempio di input: Gregory Risultato previsto: G R G R Y	Esempio di input: astemio Risultato previsto: B S T M S	Esempio di ingresso: IOUEA Risultato previsto:
---	---	---

LAB-17

Tempo stimato: 5-15 minuti

Livello di difficoltà: Facile

Obiettivi

Familiarizzare lo studente con:

utilizzare l'istruzione continue nei cicli;

modificare e aggiornare il codice esistente;

riflettere situazioni di vita reale nel codice del computer.

Scenario

Il vostro compito qui è ancora più speciale di prima: dovete riprogettare il (brutto) mangiatore di vocali del laboratorio precedente e creare un mangiatore di vocali migliore e aggiornato (bello)! Scrivete un programma che utilizzi:

- un ciclo for;
- il concetto di esecuzione condizionale (*if-elif-else*)
- L'istruzione continue.

Il vostro programma deve:

- chiedere all'utente di inserire una parola;
- utilizzare `parola_utente = parola_utente.upper()` per convertire la parola inserita dall'utente in maiuscola; parleremo molto presto dei cosiddetti metodi stringa e del metodo `upper()`, non preoccupatevi;

LAB-17

- utilizzare l'esecuzione condizionale e l'istruzione continue per "mangiare" le seguenti vocali *A, E, I, O, U* dalla parola inserita;
- assegnare le lettere non mangiate alla variabile `word_without_vowels` e stampare la variabile sullo schermo.

Osservate il codice seguente:

```
word_without_vowels = ""
```

```
# Prompt the user to enter a word
```

```
# and assign it to the user_word variable.
```

```
for letter in user_word:
```

```
    # Complete the body of the loop.
```

```
# Print the word assigned to word_without_vowels.
```

Abbiamo creato `word_without_vowels` e le abbiamo assegnato una stringa vuota. Usate l'operazione di concatenazione per chiedere a Python di combinare le lettere selezionate in una stringa più lunga durante i cicli successivi e assegnatela alla variabile `word_without_vowels`.

Testate il vostro programma

Il ciclo while e il ramo else

Entrambi i cicli, while e for, hanno una caratteristica interessante, il ramo else (viene eseguito quando la condizione è falsa)

```
i = 5
```

```
while i < 5:
```

```
    print(i)
```

```
    i += 1
```

```
else:
```

```
    print("else:", i)
```

Il ramo else del ciclo viene sempre eseguito una volta, indipendentemente dal fatto che il ciclo sia entrato o meno nel suo corpo.

Prova ad eseguire il codice anche con i=1

Il ciclo for e il ramo else

I cicli for si comportano in modo leggermente diverso: date un'occhiata al codice seguente ed eseguitelo:

```
for i in range(5):  
    print(i)  
else:  
    print("else:", i)
```

Il risultato può essere un po' sorprendente.
La variabile i mantiene il suo ultimo valore.

Il ciclo for e il ramo else

Altro esempio

```
i = 111
```

```
for i in range(2, 1):
```

```
    print(i)
```

```
else:
```

```
    print("else:", i)
```

Il corpo del ciclo non verrà eseguito in alcun modo.

Nota: abbiamo assegnato la variabile `i` prima del ciclo.

Eseguire il programma e verificarne l'output.

Quando il corpo del ciclo non viene eseguito, la variabile di controllo mantiene il valore che aveva prima del ciclo.

Nota: se la variabile di controllo non esiste prima dell'inizio del ciclo, non esisterà quando l'esecuzione raggiungerà il ramo `else`. Prova a commentare la prima riga.

.

LAB-18

Tempo stimato: 20-30 minuti

Livello di difficoltà: Medio

Obiettivi

Familiarizzare lo studente con:

utilizzando il ciclo while;

trovare la corretta applicazione di regole definite verbalmente;

riflettere situazioni di vita reale nel codice del computer.

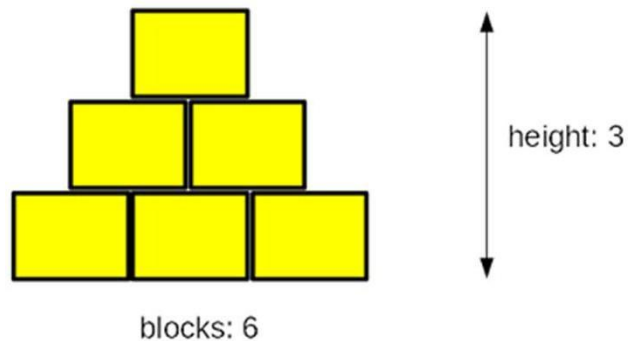
Scenario

Ascoltate questa storia: un bambino e suo padre, un programmatore di computer, stanno giocando con dei blocchi di legno. Stanno costruendo una piramide.

La loro piramide è un po' strana, perché in realtà è un muro a forma di piramide - è piatto. La piramide è impilata secondo un semplice principio: ogni strato inferiore contiene un blocco in più rispetto allo strato superiore.

La figura illustra la regola utilizzata dai costruttori:

LAB-18



Il vostro compito è quello di scrivere un programma che legga il numero di blocchi che i costruttori hanno e fornisca l'altezza della piramide che può essere costruita con questi blocchi.

Nota: l'altezza si misura in base al numero di strati completamente completati - se i costruttori non hanno un numero sufficiente di blocchi e non possono completare lo strato successivo, terminano immediatamente il loro lavoro.

Testate il vostro codice utilizzando i dati che vi abbiamo fornito.

LAB-18

```
blocks = int(input("Enter the number of blocks: "))
```

```
#
```

```
# Write your code here.
```

```
#
```

```
print("The height of the pyramid:", height)
```

Ingresso campione: 6 Risultato atteso: L'altezza della piramide: 3	Ingresso campione: 20 Risultato atteso: Altezza della piramide: 5	Ingresso campione: 1000 Risultato atteso: L'altezza della piramide: 44	Ingresso campione: 2 Risultato atteso: Altezza della piramide: 1
--	---	--	--

LAB-19

Tempo stimato: 20 minuti

Livello di difficoltà: Medio

Obiettivi

Familiarizzare lo studente con:

utilizzando il ciclo while;

convertire i loop definiti verbalmente in codice Python vero e proprio.

Scenario

Nel 1937, un matematico tedesco di nome Lothar Collatz formulò un'ipotesi intrigante (tuttora non dimostrata) che può essere descritta nel modo seguente:

- prendere un qualsiasi numero intero non negativo e non nullo e chiamarlo $c0$;
- se è pari, valutare un nuovo $c0$ come $c0 \div 2$;
- altrimenti, se è dispari, valuta un nuovo $c0$ come $3 \times c0 + 1$;
- se $c0 \neq 1$, passare al punto 2.

L'ipotesi dice che, a prescindere dal valore iniziale di $c0$, esso sarà sempre pari a 1.

Naturalmente, utilizzare un computer per dimostrare l'ipotesi di qualsiasi numero naturale è estremamente complesso (potrebbe persino richiedere un'intelligenza artificiale), ma è possibile utilizzare Python per verificare alcuni singoli numeri. Forse riuscirete a trovare quello che smentisce l'ipotesi.

LAB-19

Scrivete un programma che legga un numero naturale ed esegua i passaggi sopra descritti finché c_0 rimane diverso da 1. Vogliamo anche che contiate i passaggi necessari per raggiungere l'obiettivo. Il vostro codice deve mostrare anche tutti i valori intermedi di c_0 .

Suggerimento: la parte più importante del problema è come trasformare l'idea di Collatz in un ciclo while: questa è la chiave del successo.

Testate il vostro codice utilizzando i seguenti dati

Ingresso campione: 15

passi = 17

Ingresso campione: 16

Risultato previsto:

8

4

2

1

passi = 4

Ingresso campione: 1023

passi = 62

Esercizio 1

Create un ciclo for che conta da 0 a 10 e stampa sullo schermo i numeri dispari. Utilizzare lo scheletro sottostante:

```
for i in range(1, 11):  
    # Line of code.  
    # Line of code.
```

Esercizio 2

Create un ciclo while che conta da 0 a 10 e stampa sullo schermo i numeri dispari. Utilizzare lo scheletro sottostante:

```
x = 1  
while x < 11:  
    # Line of code.  
    # Line of code.  
    # Line of code.
```


Esercizio 3

Creare un programma con un ciclo for e un'istruzione break. Il programma deve iterare i caratteri di un indirizzo e-mail, uscire dal ciclo quando raggiunge il simbolo @ e printre la parte prima di @ su una riga. Utilizzate lo scheletro sottostante:

```
for ch in "john.smith@pythoninstitute.org":  
    if ch == "@":  
        # Line of code.  
        # Line of code.
```

Esercizio 4

Creare un programma con un ciclo for e un'istruzione continue. Il programma deve iterare su una stringa di cifre, sostituire ogni 0 con una x e stampare sullo schermo la stringa modificata. Utilizzare lo scheletro riportato di seguito:

```
for digit in "0165031806510":  
    if digit == "0":  
        # Line of code.  
        # Line of code.  
        # Line of code.
```

Esercizio 5

Qual è l'output del seguente codice?

```
n = 3
```

```
while n > 0:  
    print(n + 1)  
    n -= 1  
else:  
    print(n)
```

Esercizio 6

Qual è l'output del seguente codice?

```
n = range(4)
```

```
for num in n:  
    print(num - 1)  
else:  
    print(num)
```

Esercizio 7

Qual è l'output del seguente codice?

```
for i in range(0, 6, 3):  
    print(i)
```

Operatori logici

or

and

not

Example 1:

print(var > 0)

print(not (var <= 0))

Example 2:

print(var != 0)

print(not (var == 0))

Operatori logici - leggi di De Morgan

$\text{not } (p \text{ and } q) == (\text{not } p) \text{ or } (\text{not } q)$

$\text{not } (p \text{ or } q) == (\text{not } p) \text{ and } (\text{not } q)$

Operatori bitwise

Esistono quattro operatori che consentono di manipolare singoli bit di dati. Sono chiamati operatori bitwise.

Coprono tutte le operazioni menzionate in precedenza nel contesto logico e un operatore aggiuntivo. Si tratta dell'operatore xor (come in exclusive or), indicato con ^ (caret).

Eccoli tutti:

- & (ampersand) - congiunzione bitwise;
- | (bar) - disgiunzione bitwise;
- ~ (tilde) - negazione bitwise;
- ^ (caret) - bitwise exclusive or (xor).

& richiede esattamente due 1 per fornire 1 come risultato;

| richiede almeno un 1 per fornire 1 come risultato;

^ richiede esattamente un 1 per fornire 1 come risultato.

gli argomenti di questi operatori devono essere numeri interi; non si possono usare i float.

Operatori bitwise

La differenza nel funzionamento degli operatori logici e degli operatori di bit è importante: gli operatori logici non penetrano nel livello di bit del loro argomento. Sono interessati solo al valore intero finale. Gli operatori bitwise sono più severi: trattano ogni bit separatamente. Se assumiamo che la variabile intera occupi 64 bit (cosa comune nei moderni sistemi informatici), si può immaginare l'operazione bitwise come una valutazione 64 volte dell'operatore logico per ogni coppia di bit degli argomenti.

Operazioni logiche vs operazioni in bit: continua

Mostreremo ora un esempio della differenza di funzionamento tra le operazioni logiche e quelle con i bit. Supponiamo che siano state eseguite le seguenti assegnazioni:

i = 15

j = 22

Se assumiamo che i numeri interi siano memorizzati con 32 bit, l'immagine bitwise delle due variabili sarà la seguente:

i: 000000000000000000000000000000001111

j: 0000000000000000000000000000000010110

log = i and j

Si tratta di una congiunzione logica. Tracciamo il corso dei calcoli. Entrambe le variabili i e j non sono zeri, quindi saranno considerate vere. Consultando la tabella di verità per l'operatore and, possiamo vedere che il risultato sarà Vero. Non vengono eseguite altre operazioni.

log: Vero

Operazioni logiche vs operazioni in bit: continua

Ora l'operazione bitwise: eccola:

bit = i & j

L'operatore & agisce separatamente su ogni coppia di bit corrispondenti, producendo i valori dei relativi bit del risultato. Pertanto, il risultato sarà il seguente

<code>i</code>	000000000000000000000000000000001111
<code>j</code>	0000000000000000000000000000000010110
<code>bit = i & j</code>	00000000000000000000000000000000110

Questi bit corrispondono al valore intero di sei.

Operazioni logiche vs operazioni in bit: continua

Vediamo ora gli operatori di negazione. Prima quello logico:

logneg = non i

La variabile logneg sarà impostata su False e non sarà necessario fare altro.

La negazione bitwise si svolge in questo modo:

$$\text{bitneg} = \sim i$$

Può essere un po' sorprendente: il valore della variabile `bitneg` è -16. Può sembrare strano, ma non lo è affatto. Per saperne di più, si consiglia di consultare il sistema numerico binario e le regole che governano i numeri a complemento a due.

i	000000000000000000000000000000001111
bitneg = ~i	111111111111111111111111111111110000

Operatori bitwise

Ognuno di questi operatori a due argomenti può essere utilizzato in forma abbreviata. Questi sono gli esempi delle loro notazioni equivalenti:

<code>x = x & y</code>	<code>x &= y</code>
<code>x = x y</code>	<code>x = y</code>
<code>x = x ^ y</code>	<code>x ^= y</code>

Spostamento binario a sinistra e spostamento binario a destra

Python offre un'altra operazione relativa ai singoli bit: lo shifting. Questa operazione si applica solo ai valori interi e non si possono usare i float come argomenti.

Applicate già questa operazione molto spesso e in modo del tutto inconsapevole. Come si fa a moltiplicare un numero qualsiasi per dieci? Date un'occhiata:

$$12345 \times 10 = 123450$$

Come si può notare, la moltiplicazione per dieci è in realtà uno spostamento di tutte le cifre verso sinistra e il riempimento dello spazio risultante con lo zero.

Divisione per dieci? Date un'occhiata:

$$12340 \div 10 = 1234$$

Dividere per dieci non è altro che spostare le cifre a destra.

Spostamento binario a sinistra e spostamento binario a destra

Lo stesso tipo di operazione viene eseguita dal computer, ma con una differenza: poiché la base dei numeri binari è due (e non 10), spostare un valore di un bit a sinistra corrisponde a moltiplicarlo per due; rispettivamente, spostare un bit a destra equivale a dividere per due (si noti che il bit più a destra viene perso).

Gli operatori di spostamento in Python sono `<<` e `>>`, che suggeriscono chiaramente in quale direzione agirà lo spostamento.

valore `<<` bit

valore `>>` bit

L'argomento sinistro di questi operatori è un valore intero i cui bit vengono spostati. L'argomento destro determina la dimensione dello spostamento.

Questa operazione non è certamente commutativa.

La priorità di questi operatori è molto alta. Li vedrete nella tabella aggiornata delle priorità.

Spostamento binario a sinistra e spostamento binario a destra

Osservate il seguente codice

```
var = 17  
var_right = var >> 1  
var_left = var << 2  
print(var, var_left, var_right)
```

L'invocazione finale di print() produce il seguente risultato:

17 68 8

Nota:

- $17 \gg 1 \rightarrow 17 // 2$ (17 diviso per 2 alla potenza di 1) $\rightarrow 8$ (lo spostamento a destra di un bit equivale alla divisione intera per due)
- $17 \ll 2 \rightarrow 17 * 4$ (17 moltiplicato per 2 alla potenza di 2) $\rightarrow 68$ (lo spostamento a sinistra di due bit equivale alla moltiplicazione intera per quattro)

.

Tabella delle priorità aggiornata

Priorità	Operatore	
1	~, +, -	unario
2	**	
3	*, /, //, %	
4	+, -	binario
5	<<, >>	
6	<, <=, >, >=	
7	==, !=	
8	&	
9		
10	=, +=, -=, *=, /=, %=, &=, ^=, =, >>=, <<=	

Esercizio 1

Qual è l'output del seguente snippet?

```
x = 1
```

```
y = 0
```

```
z = ((x == y) e (x == y)) o not(x == y)
```

```
print(not(z))
```

Esercizio 2

Qual è l'output del seguente snippet?

```
x = 4
```

```
y = 1
```

```
a = x & y
```

```
b = x | y
```

```
c = ~x # tricky!
```

```
d = x ^ 5
```

```
e = x >> 2
```

```
f = x << 2
```

```
print(a, b, c, d, e, f)
```