



## **Corso di Apprendistato Python**

Mod. Python  
Docente:  
Tonino Petrulli



## Tipi di sequenza e mutabilità

Prima di iniziare a parlare di **tuple** e **dizionari**, dobbiamo introdurre due concetti importanti: i **tipi di sequenza** e la **mutabilità**.

Un **tipo di sequenza** è un tipo di dati in Python in grado di memorizzare più di un valore (o meno di uno, dato che una sequenza può essere vuota), e questi valori possono essere sfogliati in sequenza (da qui il nome), elemento per elemento.

Poiché il ciclo `for` è uno strumento appositamente progettato per iterare attraverso le sequenze, possiamo esprimere la definizione come: **una sequenza è un dato che può essere analizzato dal ciclo `for`**.

Finora avete incontrato una sequenza Python: l'elenco. La lista è un classico esempio di sequenza Python, anche se esistono altre sequenze che meritano di essere menzionate e che ora vi presenteremo.

La seconda nozione - la **mutabilità** - è una proprietà di qualsiasi dato di Python che descrive la sua disponibilità a essere modificato liberamente durante l'esecuzione del programma. Esistono due tipi di dati in Python: **mutabili** e **immutabili**.

**I dati mutabili possono essere aggiornati liberamente in qualsiasi momento** - chiamiamo questa operazione in situ.

*In situ* è una locuzione latina che si traduce letteralmente in *posizione*. Ad esempio, la seguente istruzione modifica i dati in situ:

```
list.append(1)
```

## Tipi di sequenza e mutabilità

**I dati immutabili non possono essere modificati in questo modo.**

Immaginate che un elenco possa essere solo assegnato e letto. Non è possibile né aggiungervi un elemento, né rimuoverlo. Ciò significa che l'aggiunta di un elemento alla fine dell'elenco richiederebbe la ricreazione dell'elenco da zero.

Si dovrebbe costruire un elenco completamente nuovo, composto da tutti gli elementi dell'elenco già esistente, più il nuovo elemento.

Il tipo di dati di cui vogliamo parlare ora è una **tupla**. **Una tupla è un tipo di sequenza immutabile**. Può comportarsi come una lista, ma non deve essere modificata sul posto.

## Che cos'è una tupla?

La prima e più chiara distinzione tra liste e tuple è la sintassi usata per crearle: le **tuple preferiscono usare le parentesi**, mentre le liste preferiscono le parentesi, anche se è **possibile creare una tupla solo da un insieme di valori separati da virgole**.

Guardate l'esempio:

```
tuple_1 = (1, 2, 4, 8)
tuple_2 = 1., .5, .25, .125
```

Ci sono due tuple, entrambe contenenti **quattro elementi**.

Stampiamoli:

```
tuple_1 = (1, 2, 4, 8)
tuple_2 = 1., .5, .25, .125
print(tuple_1)
print(tuple_2)
```

Ecco cosa dovrebbe apparire nella console:

```
(1, 2, 4, 8)
(1.0, 0.5, 0.25, 0.125)
```

Nota: **ogni elemento della tupla può essere di tipo diverso** (a virgola mobile, intero o qualsiasi altro tipo di dato non ancora introdotto).

### Come creare una tupla?

È possibile creare una tupla vuota: in questo caso sono necessarie le parentesi:

```
empty_tuple = ()
```

Se si vuole creare una **tupla a un elemento**, bisogna tenere conto del fatto che, per motivi di sintassi (una tupla deve essere distinguibile da un normale valore singolo), è necessario terminare il valore con una virgola:

```
one_element_tuple_1 = (1, )
```

```
one_element_tuple_2 = 1.,
```

Rimuovendo le virgole non si otterrà un errore sintattico, ma si otterranno due variabili singole, non tuple.

### **Come si usa una tupla?**

Se si vogliono ottenere gli elementi di una tupla per leggerli, si possono usare le stesse convenzioni a cui si è abituati quando si usano gli elenchi.

Date un'occhiata al codice.

```
my_tuple = (1, 10, 100, 1000)  
print(my_tuple[0])  
print(my_tuple[-1])  
print(my_tuple[1:])  
print(my_tuple[:-2])  
for elem in my_tuple:  
    print(elem)
```

Il programma dovrebbe produrre il seguente output: eseguitelo e verificate:

```
1  
1000  
(10, 100, 1000)  
(1, 10)  
1  
10  
100  
1000
```

## Come si usa una tupla?

Le somiglianze possono essere fuorvianti: **non cercate di modificare il contenuto di una tupla!** Non è un elenco! Tutte queste istruzioni (tranne quella più in alto) causano un errore di esecuzione:

```
my_tuple = (1, 10, 100, 1000)
```

```
my_tuple.append(10000)
```

```
del my_tuple[0]
```

```
my_tuple[1] = -10
```

Questo è il messaggio che Python fornisce nella finestra della console:

```
AttributeError: 'tuple' object has no attribute 'append'
```

## Come utilizzare una tupla: continua

Cos'altro possono fare le tuple per voi?

- la funzione `len()` accetta tuple e restituisce il numero di elementi contenuti all'interno;
- l'operatore `+` può unire tuple tra loro (lo abbiamo già mostrato)
- l'operatore `*` può moltiplicare le tuple, proprio come le liste;
- Gli operatori `in` e `not in` funzionano come negli elenchi.

Lo snippet seguente li presenta tutti.

```
my_tuple = (1, 10, 100)  
t1 = my_tuple + (1000, 10000)  
t2 = my_tuple * 3  
print(len(t2))  
print(t1)  
print(t2)  
print(10 in my_tuple)  
print(-10 not in my_tuple)
```

L'output dovrebbe essere il seguente:

```
9  
(1, 10, 100, 1000, 10000)  
(1, 10, 100, 1, 10, 100, 1, 10, 100)  
True  
True
```



## Come utilizzare una tupla: continua

Una delle proprietà più utili delle tuple è la loro capacità di **apparire sul lato sinistro dell'operatore di assegnazione**. Avete visto questo fenomeno qualche tempo fa, quando è stato necessario trovare uno strumento elegante per scambiare i valori di due variabili.

Date un'occhiata allo snippet qui sotto:

```
var = 123
t1 = (1, )
t2 = (2, )
t3 = (3, var)
t1, t2, t3 = t2, t3, t1
print(t1, t2, t3)
```

Mostra tre tuple che interagiscono - in effetti, i valori in esse memorizzati "circolano" - t1 diventa t2, t2 diventa t3 e t3 diventa t1.

Nota: l'esempio presenta un altro fatto importante: **gli elementi di una tupla possono essere variabili**, non solo letterali. Inoltre, possono essere espressioni se si trovano sul lato destro dell'operatore di assegnazione.

## Che cos'è un dizionario?

Il **dizionario** è un'altra struttura dati di Python. **Non è un** tipo di **sequenza** (ma può essere facilmente adattato all'elaborazione di sequenze) ed è **mutabile**.

Per spiegare cos'è effettivamente il dizionario Python, è importante capire che si tratta letteralmente di un dizionario.

Il dizionario Python funziona come un **dizionario bilingue**. Ad esempio, si ha una parola inglese (ad esempio, cat) e si ha bisogno del suo equivalente francese. Si sfoglia il dizionario per trovare la parola (si possono usare diverse tecniche per farlo, non importa) e alla fine la si ottiene. Poi si controlla la controparte francese, che è (molto probabilmente) la parola "chat".



## Che cos'è un dizionario?

Nel mondo di Python, la parola che si cerca si chiama chiave. La parola che si ottiene dal dizionario si chiama valore.

Ciò significa che un dizionario è un insieme di coppie **chiave-valore**. Nota:

- ogni chiave deve essere **unica**: non è possibile avere più di una chiave con lo stesso valore;
- una chiave può essere **qualsiasi tipo di oggetto immutabile**: può essere un numero (intero o float), o anche una stringa, ma non un elenco;
- un dizionario non è un elenco: un elenco contiene un insieme di valori numerati, mentre un **dizionario contiene coppie di valori**;
- la funzione len() funziona anche per i dizionari: restituisce il numero di elementi chiave-valore nel dizionario;
- un dizionario è **uno strumento unidirezionale**: se avete un dizionario inglese-francese, potete cercare gli equivalenti francesi dei termini inglesi, ma non viceversa.

## Come creare un dizionario?

Se si desidera assegnare alcune coppie iniziali a un dizionario, si deve usare la seguente sintassi:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}  
phone_numbers = {'boss': 5551234567, 'Suzy': 22657854310}  
empty_dictionary = {}  
print(dictionary)  
print(phone_numbers)  
print(empty_dictionary)
```

Nel primo esempio, il dizionario utilizza chiavi e valori che sono entrambi stringhe. Nel secondo, le chiavi sono stringhe, ma i valori sono numeri interi. È possibile anche la disposizione inversa (chiavi → numeri, valori → stringhe), così come la combinazione numero-numero.

L'elenco delle coppie è **circondato da parentesi graffe**, mentre le coppie stesse sono **separate da virgole** e le **chiavi e i valori da punti**.

Il primo dei nostri dizionari è un dizionario inglese-francese molto semplice. Il secondo è un piccolissimo elenco telefonico.

I dizionari vuoti sono costruiti da una **coppia di parentesi graffe vuote**: niente di strano.

Il dizionario nel suo complesso può essere stampato con una singola invocazione di `print()`.

## Come creare un dizionario?

L'output del codice precedente:

```
{'dog': 'chien', 'horse': 'cheval', 'cat': 'chat'}  
{'Suzy': 5557654321, 'boss': 5551234567}  
{}
```

Avete notato qualcosa di sorprendente? L'ordine delle coppie stampate è diverso da quello dell'assegnazione iniziale. Che cosa significa?

Prima di tutto, è una conferma che **i dizionari non sono elenchi**: non conservano l'ordine dei loro dati, poiché l'ordine è completamente privo di significato (a differenza dei veri dizionari cartacei). L'ordine in cui un dizionario **memorizza i suoi dati è completamente fuori dal vostro controllo** e dalle vostre aspettative. È normale. (\*)

NOTA

(\*) In Python 3.6x i dizionari sono diventati collezioni **ordinate** per impostazione predefinita. I risultati possono variare a seconda della versione di Python in uso.

## Come si usa un dizionario?

Per ottenere uno qualsiasi dei valori, è necessario fornire un valore di chiave valido:

```
print(dictionary['cat'])  
print(phone_numbers['Suzy'])
```

Ottenere il valore di un dizionario assomiglia all'indicizzazione, soprattutto grazie alle parentesi che circondano il valore della chiave.

Nota:

- se la chiave è una stringa, è necessario specificarla come stringa;
- **Le chiavi sono sensibili alle maiuscole e alle minuscole:** "Suzy" è qualcosa di diverso da "suzy".

Lo snippet produce due righe di testo:

```
chat  
22657854310
```

E ora la notizia più importante: **non si deve usare una chiave inesistente**. Provate a fare qualcosa del genere:

```
print(phone_numbers['president'])
```

causerà un errore di runtime. Provate a farlo.

## **Come si usa un dizionario?**

Fortunatamente, esiste un modo semplice per evitare questa situazione. L'operatore `in`, insieme al suo compagno, `not in`, può salvare questa situazione.

Il codice seguente cerca in modo sicuro alcune parole francesi:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}  
words = ['cat', 'lion', 'horse']  
for word in words:  
    if word in dictionary:  
        print(word, "->", dictionary[word])  
    else:  
        print(word, "is not in dictionary")
```

L'output del codice è il seguente:

cat -> chat

lion is not in dictionary

horse -> cheval

## Come si usa un dizionario?

**Nota:** Quando si scrive un'espressione grande o lunga, può essere una buona idea mantenerla allineata verticalmente. In questo modo è possibile rendere il codice più leggibile e più facile da leggere per i programmatori, ad es:

# Example 1:

```
dictionary = {  
    "cat": "chat",  
    "dog": "chien",  
    "horse": "cheval"  
}
```

# Example 2:

```
phone_numbers = {'boss': 5551234567,  
    'Suzy': 22657854310  
}
```

Questo tipo di formattazione si chiama **hanging indent**.



## Come utilizzare un dizionario: il metodo keys()

I dizionari possono essere **sfogliati** utilizzando il ciclo for, come le liste o le tuple?

No e sì.

No, perché un dizionario **non è un tipo di sequenza**: il ciclo for è inutile con esso.

Sì, perché esistono strumenti semplici e molto efficaci che possono **adattare qualsiasi dizionario ai requisiti del ciclo for** (in altre parole, creando un collegamento intermedio tra il dizionario e un'entità di sequenza temporanea).

Il primo di essi è un metodo chiamato keys(), posseduto da ogni dizionario. Il metodo **restituisce un oggetto iterabile costituito da tutte le chiavi raccolte nel dizionario**. Avere un gruppo di chiavi consente di accedere all'intero dizionario in modo semplice e pratico.

Proprio come qui:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
```

```
for key in dictionary.keys():
```

```
    print(key, "->", dictionary[key])
```

L'output del codice è il seguente:

horse -> cheval

dog -> chien

cat -> chat

## La funzione sorted()

Si vuole che sia **ordinato**? Basta arricchire il ciclo for:

**for key in sorted(dictionary.keys()):**

La funzione sorted() farà del suo meglio: l'output sarà simile a questo:

cat -> chat

dog -> chien

horse -> cheval

## Come utilizzare un dizionario: I metodi `items()` e `values()`

Un altro modo si basa sull'uso del metodo di un dizionario chiamato `items()`. Il metodo **restituisce tuple** (questo è il primo esempio in cui le tuple sono qualcosa di più di un semplice esempio di se stesse) **in cui ogni tupla è una coppia chiave-valore**.

Ecco come funziona:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
```

```
for english, french in dictionary.items():
```

```
    print(english, "->", french)
```

Si noti il modo in cui la tupla è stata utilizzata come variabile del ciclo `for`.

L'esempio stampa:

```
cat -> chat
```

```
dog -> chien
```

```
horse -> cheval
```

## Come utilizzare un dizionario: I metodi `items()` e `values()`

Esiste anche un metodo chiamato `values()`, che funziona in modo simile a `keys()`, ma **restituisce dei valori**.

Ecco un semplice esempio:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
```

```
for french in dictionary.values():
```

```
    print(french)
```

Poiché il dizionario non è in grado di trovare automaticamente una chiave per un dato valore, il ruolo di questo metodo è piuttosto limitato.

Ecco il risultato atteso:

```
cheval
```

```
chien
```

```
chat
```

## Come utilizzare un dizionario: modifica e aggiunta di valori

Assegnare un nuovo valore a una chiave esistente è semplice: poiché i dizionari sono completamente **mutabili**, non ci sono ostacoli alla loro modifica.

Sostituiremo il valore "chat" con "minou", che non è molto preciso, ma funziona bene con il nostro esempio. Guarda:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}  
dictionary['cat'] = 'minou'  
print(dictionary)
```

L'output è il seguente:

```
{'cat': 'minou', 'dog': 'chien', 'horse': 'cheval'}
```

## Aggiunta di una nuova chiave

Aggiungere una nuova coppia chiave-valore a un dizionario è semplice come cambiare un valore: basta assegnare un valore a una nuova **chiave, prima inesistente**.

Nota: si tratta di un comportamento molto diverso rispetto alle liste, che non consentono di assegnare valori a indici inesistenti.

Aggiungiamo al dizionario una nuova coppia di parole, un po' strana, ma comunque valida:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}  
dictionary['swan'] = 'cygne'  
print(dictionary)
```

L'esempio produce output:

```
{'cat': 'chat', 'dog': 'chien', 'horse': 'cheval', 'swan': 'cygne'}
```

È anche possibile inserire un elemento in un dizionario utilizzando il metodo `update()`, ad es:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}  
dictionary.update({"duck": "canard"})  
print(dictionary)
```

## Rimozione di una chiave

Riuscite a indovinare come si rimuove una chiave da un dizionario?

Nota: la rimozione di una chiave provoca sempre la **rimozione del valore associato**. I valori non possono esistere senza le loro chiavi.

Ciò avviene con l'istruzione del.

Ecco un esempio:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}  
del dictionary['dog']  
print(dictionary)
```

Nota: la **rimozione di una chiave non esistente causa un errore**.

L'esempio produce output:

```
{'cat': 'chat', 'horse': 'cheval'}
```

Per rimuovere l'ultimo elemento di un dizionario, si può usare il metodo popitem():

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}  
dictionary.popitem()  
print(dictionary)    # outputs: {'cat': 'chat', 'dog': 'chien'}
```

Nelle vecchie versioni di Python, cioè prima della 3.6.7, il metodo popitem() rimuove un elemento casuale da un dizionario.

## **Tuple e dizionari possono lavorare insieme**

Abbiamo preparato un semplice esempio che mostra come tuple e dizionari possano lavorare insieme. Immaginiamo il seguente problema:

- è necessario un programma per valutare i punteggi medi degli studenti;
- il programma deve chiedere il nome dello studente, seguito dal suo punteggio singolo;
- i nomi possono essere inseriti in qualsiasi ordine;
- l'inserimento di un nome vuoto termina l'inserimento dei dati (nota 1: l'inserimento di un punteggio vuoto solleverà l'eccezione `ValueError`, ma non preoccupatevi di questo ora, vedrete come gestire questi casi quando parleremo di eccezioni nella seconda parte della serie di corsi Python Essentials)
- viene emesso un elenco di tutti i nomi, insieme

Guardate il codice



## **Tuple e dizionari possono lavorare insieme**

```
school_class = {}  
while True:  
    name = input("Enter the student's name: ")  
    if name == "":  
        break  
    score = int(input("Enter the student's score (0-10): "))  
    if score not in range(0, 11):  
        break  
    if name in school_class:  
        school_class[name] += (score,)   
    else:  
        school_class[name] = (score,)   
  
for name in sorted(school_class.keys()):  
    adding = 0  
    counter = 0  
    for score in school_class[name]:  
        adding += score  
        counter += 1  
    print(name, ":", adding / counter)
```

## Tuple e dizionari possono lavorare insieme

Ora analizziamo riga per riga:

**riga 1:** creare un dizionario vuoto per i dati di input; il nome dello studente è usato come chiave, mentre tutti i punteggi associati sono memorizzati in una tupla (la tupla può essere un valore del dizionario, non è un problema)

**riga 3:** inserire un ciclo "infinito" (non preoccupatevi, si interromperà al momento giusto)

**riga 4:** leggere qui il nome dello studente;

**riga 5-6:** se il nome è una stringa vuota (), lascia il ciclo;

**riga 8:** richiedere uno dei punteggi dello studente (un numero intero compreso nell'intervallo 0-10)

**righe 9-10:** se il punteggio inserito non rientra nell'intervallo da 0 a 10, lascia il ciclo;

**riga 12-13:** se il nome dello studente è già presente nel dizionario, allunga la tupla associata con il nuovo punteggio (notare l'operatore +=)

**righe 14-15:** se si tratta di un nuovo studente (sconosciuto al dizionario), crea una nuova voce - il suo valore è una tupla a un elemento contenente il punteggio inserito;

**riga 17:** iterare i nomi degli studenti ordinati;

**righe 18-19:** inizializzazione dei dati necessari per valutare la media (somma e contatore)

**righe 20-22:** si itera attraverso la tupla, prendendo tutti i punteggi successivi e aggiornando la somma, insieme al contatore;

**riga 23:** valutare e stampare il nome dello studente e il punteggio medio.

## **Tuple e dizionari possono lavorare insieme**

Questo è un resoconto della conversazione avuta con il nostro programma:

Enter the student's name: Bob

Enter the student's score (0-10): 7

Enter the student's name: Andy

Enter the student's score (0-10): 3

Enter the student's name: Bob

Enter the student's score (0-10): 2

Enter the student's name: Andy

Enter the student's score (0-10): 10

Enter the student's name: Andy

Enter the student's score (0-10): 3

Enter the student's name: Bob

Enter the student's score (0-10): 9

Enter the student's name:

Andy : 5.333333333333333

Bob : 6.0

## Tuple

È anche possibile creare una tupla utilizzando una funzione integrata in Python chiamata `tuple()`. Questa funzione è particolarmente utile quando si vuole convertire un determinato iterabile (ad esempio, un elenco, un intervallo, una stringa, ecc.) in una tupla:

```
my_tuple = tuple((1, 2, "string"))
print(my_tuple)
my_list = [2, 4, 6]
print(my_list)    # outputs: [2, 4, 6]
print(type(my_list))    # outputs: <class 'list'>
tup = tuple(my_list)
print(tup)    # outputs: (2, 4, 6)
print(type(tup))    # outputs: <class 'tuple'>
```

## Dizionari

Se si desidera scorrere le chiavi e i valori di un dizionario, è possibile utilizzare il metodo `items()`, ad es:

```
pol_eng_dictionary = {  
    "zamek": "castle",  
    "woda": "water",  
    "gleba": "soil"  
}
```

```
for key, value in pol_eng_dictionary.items():  
    print("Pol/Eng ->", key, ":", value)
```

È possibile utilizzare la parola chiave `del` per rimuovere un elemento specifico o cancellare un dizionario. Per rimuovere tutti gli elementi del dizionario, è necessario utilizzare il metodo `clear()`:

```
pol_eng_dictionary = {  
    "zamek": "castle",  
    "woda": "water",  
    "gleba": "soil"  
}
```

```
print(len(pol_eng_dictionary))    # outputs: 3  
del pol_eng_dictionary["zamek"]   # remove an item  
print(len(pol_eng_dictionary))    # outputs: 2  
pol_eng_dictionary.clear()        # removes all the items  
print(len(pol_eng_dictionary))    # outputs: 0  
del pol_eng_dictionary            # removes the dictionary
```

## Dizionari

. Per copiare un dizionario, utilizzare il metodo `copy()`:

```
pol_eng_dictionary = {  
    "zamek": "castle",  
    "woda": "water",  
    "gleba": "soil"  
}
```

```
copy_dictionary = pol_eng_dictionary.copy()
```

## Esercizio 1

Cosa succede quando si tenta di eseguire il seguente snippet?

```
my_tup = (1, 2, 3)  
print(my_tup[2])
```

## Esercizio 2

Qual è l'output del seguente snippet?

```
tup = 1, 2, 3  
a, b, c = tup
```

```
print(a * b * c)
```

### Esercizio 3

Completate il codice per utilizzare correttamente il metodo count() per trovare il numero di duplicati di 2 nella seguente tupla.

```
tup = 1, 2, 3, 2, 4, 5, 6, 2, 7, 2, 8, 9  
duplicates = # Write your code here.
```

```
print(duplicates) # outputs: 4
```

**Soluzione:**

```
duplicates = tup.count(2)
```

### Esercizio 4

Scrivete un programma che "incolli" i due dizionari (d1 e d2) e ne crei uno nuovo (d3).

```
d1 = {'Adam Smith': 'A', 'Judy Paxton': 'B+'}
```

```
d2 = {'Mary Louis': 'A', 'Patrick White': 'C'}
```

```
d3 = {}
```

```
for item in (d1, d2):
```

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

```
print(d3)
```

**Soluzione:**

```
d3.update(item)
```

### **Esercizio 5**

Scrivere un programma che converta l'elenco my\_list in una tupla.

```
my_list = ["car", "Ford", "flower", "Tulip"]
```

```
t = # Write your code here.  
print(t)
```

**Soluzione:** t = tuple(my\_list)

### **Esercizio 6**

Scrivere un programma che converta la tupla dei colori in un dizionario.

```
colors = (("green", "#008000"), ("blue", "#0000FF"))
```

```
# Write your code here.  
print(colors_dictionary)
```

**Soluzione:**

```
colors = (("green", "#008000"), ("blue", "#0000FF"))  
colors_dictionary = dict(colors)  
print(colors_dictionary)
```



### **Esercizio 7**

Cosa succede quando si esegue il seguente codice?

```
my_dictionary = {"A": 1, "B": 2}  
copy_my_dictionary = my_dictionary.copy()  
my_dictionary.clear()  
print(copy_my_dictionary)
```

### **Esercizio 8**

Qual è l'output del seguente programma?

```
colors = {  
    "white": (255, 255, 255),  
    "grey": (128, 128, 128),  
    "red": (255, 0, 0),  
    "green": (0, 128, 0)  
}
```

```
for col, rgb in colors.items():  
    print(col, ":", rgb)
```

## Che cos'è un modulo?

Il codice informatico ha la tendenza a crescere. Possiamo dire che il codice che non cresce è probabilmente completamente inutilizzabile o abbandonato. Un codice reale, ricercato e ampiamente utilizzato si sviluppa continuamente, in quanto le richieste e le aspettative degli utenti si sviluppano secondo i propri ritmi.

Un codice che non è in grado di rispondere alle esigenze degli utenti verrà dimenticato rapidamente e sostituito all'istante con un codice nuovo, migliore e più flessibile. Preparatevi a questo, e non pensate mai che uno dei vostri programmi sia completato. Il completamento è uno stato di transizione e di solito passa rapidamente, dopo la prima segnalazione di bug. Python stesso è un buon esempio di come agisce questa regola.

La crescita del codice è di fatto un problema crescente. Un codice più grande significa sempre una manutenzione più difficile. La ricerca di bug è sempre più facile quando il codice è più piccolo (proprio come trovare una rottura meccanica è più semplice quando il macchinario è più semplice e più piccolo).

Inoltre, quando si prevede che il codice da creare sia molto grande (si può usare il numero totale di righe di sorgente come misura utile, ma non molto accurata, della dimensione di un codice), si può desiderare (o meglio, si è costretti) a dividerlo in molte parti, implementate in parallelo da pochi, una dozzina, diverse decine o addirittura diverse centinaia di singoli sviluppatori.

Naturalmente, questo non può essere fatto utilizzando un unico grande file sorgente, che viene modificato da tutti i programmatori contemporaneamente. Questo porterebbe sicuramente a un disastro spettacolare.

## Che cos'è un modulo?

Se volete che un progetto software di questo tipo venga portato a termine con successo, dovete disporre dei mezzi che vi consentano di farlo:

- dividere tutti i compiti tra gli sviluppatori;
- unire tutte le parti create in un unico insieme funzionante.

Ad esempio, un determinato progetto può essere suddiviso in due parti principali:

- l'interfaccia utente (la parte che comunica con l'utente utilizzando widget e una schermata grafica)
- la logica (la parte che elabora i dati e produce risultati)

Ciascuna di queste parti può essere (molto probabilmente) suddivisa in altre più piccole, e così via. Questo processo è spesso chiamato **decomposizione**.

Ad esempio, se vi chiedessero di organizzare un matrimonio, non fareste tutto da soli, ma trovereste una serie di professionisti e dividereste il compito tra tutti.

Come si fa a dividere un software in parti separate ma collaboranti? Questa è la domanda. **I moduli** sono la risposta.

## Come utilizzare un modulo?

Che cos'è un modulo? Il [Tutorial Python](#) lo definisce come **un file contenente definizioni e dichiarazioni Python**, che può essere importato e utilizzato in seguito quando necessario.

La gestione dei moduli consiste in due aspetti diversi:

- il primo (probabilmente il più comune) si verifica quando si vuole utilizzare un modulo già esistente, scritto da qualcun altro o creato da voi stessi durante il lavoro su qualche progetto complesso - in questo caso siete l'**utente** del modulo;
- la seconda si verifica quando si vuole creare un modulo nuovo di zecca, per uso personale o per semplificare la vita di altri programmatori: si è il **fornitore** del modulo.

Discutiamone separatamente.

Innanzitutto, un modulo è identificato dal suo **nome**. Se si vuole utilizzare un modulo, è necessario conoscerne il nome. Un numero (piuttosto elevato) di moduli viene fornito insieme a Python stesso. Si può pensare a questi moduli come a una sorta di "equipaggiamento supplementare di Python".

Tutti questi moduli, insieme alle funzioni integrate, formano la **libreria standard di Python**, una sorta di biblioteca speciale in cui i moduli svolgono il ruolo di libri (possiamo anche dire che le cartelle svolgono il ruolo di scaffali).

Se volete dare un'occhiata all'elenco completo di tutti i "volumi" raccolti in quella biblioteca, potete trovarlo qui:

<https://docs.python.org/3/library/index.html>.

Ogni modulo è composto da entità (come un libro è composto da capitoli). Queste entità possono essere funzioni, variabili, costanti, classi e oggetti. Se si sa come accedere a un particolare modulo, è possibile utilizzare tutte le entità in esso memorizzate.

## Importare un modulo

Iniziamo la discussione con uno dei moduli più utilizzati, chiamato `math`. Il suo nome parla da sé: il modulo contiene una ricca collezione di entità (non solo funzioni) che consentono al programmatore di implementare efficacemente i calcoli che richiedono l'uso di funzioni matematiche, come `sin()` o `log()`.

Per rendere utilizzabile un modulo, è necessario **importarlo** (come se si trattasse di prendere un libro dallo scaffale). L'importazione di un modulo avviene tramite un'istruzione chiamata `import`. Nota: `import` è anche una parola chiave (con tutte le conseguenze del caso).

Supponiamo di voler utilizzare due entità fornite dal modulo `math`:

- un simbolo (costante) che rappresenta un valore preciso (il più preciso possibile usando l'aritmetica a virgola mobile doppia) di  $\pi$  (sebbene l'uso di una lettera greca per nominare una variabile sia pienamente possibile in Python, il simbolo è chiamato **pi greco** - è una soluzione più comoda, soprattutto per quella parte del mondo che non ha né intende usare una tastiera greca)
- una funzione denominata `sin()` (l'equivalente informatico della funzione matematica *seno*)

Entrambe queste entità sono disponibili attraverso il modulo matematico, ma il modo in cui è possibile utilizzarle dipende fortemente da come è stata effettuata l'importazione.

Il modo più semplice per importare un particolare modulo è utilizzare l'istruzione `import` come segue:

```
import math
```

## Importare un modulo

La clausola contiene:

- la parola chiave `import`;
- il **nome del modulo** da importare.

L'istruzione può essere collocata in qualsiasi punto del codice, ma deve essere posta **prima del primo utilizzo di una qualsiasi entità del modulo**.

Se si vuole (o si deve) importare più di un modulo, lo si può fare ripetendo la clausola `import` (preferibile):

```
import math  
import sys
```

oppure elencando i moduli dopo la parola chiave `import`, come in questo caso:

```
import math, sys
```

L'istruzione importa due moduli, prima quello denominato `math` e poi il secondo denominato `sys`.  
L'elenco dei moduli può essere arbitrariamente lungo.

## Importazione di un modulo: continua

Per continuare, è necessario familiarizzare con un termine importante: lo **spazio dei nomi**. Non preoccupatevi, non entreremo nei dettagli: la spiegazione sarà la più breve possibile.

Uno **spazio dei nomi** è uno spazio (inteso in un contesto non fisico) in cui esistono alcuni nomi e i nomi non sono in conflitto tra loro (cioè non esistono due oggetti diversi con lo stesso nome). Possiamo dire che ogni gruppo sociale è uno spazio dei nomi: il gruppo tende a dare un nome unico a ciascuno dei suoi membri (ad esempio, i genitori non daranno ai loro figli lo stesso nome).



## Importazione di un modulo: continua

L'unicità può essere ottenuta in molti modi, ad esempio utilizzando dei soprannomi insieme ai nomi di battesimo (funzionerà all'interno di un piccolo gruppo come una classe in una scuola) o assegnando identificatori speciali a tutti i membri del gruppo (il codice fiscale è un buon esempio di questa pratica).

**All'interno di un certo spazio dei nomi, ogni nome deve rimanere unico.** Questo può significare che alcuni nomi possono scomparire quando un'entità con un nome già noto entra nello spazio dei nomi. Vi mostreremo come funziona e come controllarlo, ma prima torniamo alle importazioni.

Se il modulo con il nome specificato **esiste ed è accessibile** (un modulo è infatti un **file sorgente Python**), Python ne importa il contenuto, cioè **tutti i nomi definiti nel modulo diventano noti**, ma non entrano nello spazio dei nomi del codice.

Ciò significa che si possono avere entità di nome `sin` o `pi` e non saranno in alcun modo influenzate dall'importazione.



A questo punto, ci si potrebbe chiedere come accedere al `pi` greco proveniente dal modulo matematico. Per fare ciò, è necessario qualificare il `pi` greco con il nome del suo modulo originale.



## Importazione di un modulo: continua

Guardate lo snippet qui sotto: questo è il modo in cui si qualificano i nomi di pi e sin con il nome del modulo di origine:

**math.pi**  
**math.sin**

È semplice:

- il **nome del modulo** (ad esempio, matematica)
- un **punto** (cioè .)
- il **nome dell'entità** (ad esempio, pi)

Tale forma indica chiaramente lo spazio dei nomi in cui il nome esiste.

Nota: l'uso di questa qualifica è **obbligatorio** se un modulo è stato importato dall'istruzione `import module`. Non importa se i nomi del codice e dello spazio dei nomi del modulo sono in conflitto o meno.

Questo primo esempio non sarà molto avanzato: vogliamo solo stampare il valore di  **$\sin(\frac{1}{2}\pi)$** .

Guardate il codice

```
import math  
print(math.sin(math.pi/2))
```

Questo è il modo in cui lo testiamo.

Il codice restituisce il valore atteso: 1,0.

## Importazione di un modulo: continua

Nel secondo metodo, la sintassi dell'importazione indica con precisione quale entità (o entità) del modulo è accettabile nel codice:

**from math import pi**

L'istruzione è composta dai seguenti elementi:

- la parola chiave **from**;
- il **nome del modulo** da importare (selettivamente);
- la parola chiave **import**;
- il **nome o l'elenco di nomi dell'entità o delle entità** che vengono importate nello spazio dei nomi.

L'istruzione ha questo effetto:

- le entità elencate (e solo quelle) sono **importate dal modulo indicato**;
- i nomi delle entità importate sono **accessibili senza alcuna qualificazione**.

Nota: non vengono importate altre entità. Inoltre, non è possibile importare entità aggiuntive, ovvero una riga come questa:

```
print(math.e)
```

causerà un errore.

## Importazione di un modulo: continua

ATTENZIONE!

Se usiamo nomi di funzioni o costanti definiti nel modulo, copriranno quelli definiti nel modulo

```
from math import sin, pi
```

```
print(sin(pi / 2))
```

```
pi = 3.14
```

```
def sin(x):
```

```
    if 2 * x == pi:
```

```
        return 0.99999999
```

```
    else:
```

```
        return None
```

```
print(sin(pi / 2))
```

La prima print userà la funzione sin e la costante pi definite nel modulo math,  
L'ultima print userà pi=3.14 e la funzione sin definita nel codice.

**Importare un modulo: \***

Nel terzo metodo, la sintassi dell'importazione è una forma più aggressiva di quella presentata in precedenza:

**from module import \***

Come si può vedere, il nome di un'entità (o l'elenco dei nomi delle entità) viene sostituito da un singolo asterisco (\*).

Un'istruzione di questo tipo **importa tutte le entità del modulo indicato**.

È conveniente? Sì, perché vi solleva dal compito di elencare tutti i nomi di cui avete bisogno.

Non è sicuro? Sì, lo è: a meno che non si conoscano tutti i nomi forniti dal modulo, **potrebbe non essere possibile evitare conflitti di nomi**. Consideratela come una soluzione temporanea e cercate di non usarla nel codice normale.

## **Importare un modulo: la parola chiave as**

Se si usa la variante `import module` e non si gradisce il nome di un particolare modulo (per esempio, è lo stesso di una delle entità già definite, quindi la qualificazione diventa problematica), si può dargli un nome a piacere: questo si chiama **aliasing**.

L'aliasing fa sì che il modulo venga identificato con un nome diverso da quello originale. Questo può abbreviare anche i nomi qualificati.

La creazione di un alias avviene insieme all'importazione del modulo e richiede la seguente forma dell'istruzione di importazione:

**`import module as alias`**

Il "modulo" identifica il nome del modulo originale, mentre l'"alias" è il nome che si desidera utilizzare al posto dell'originale.

Nota: `as` è una parola chiave.

Se è necessario modificare la parola matematica, è possibile introdurre il proprio nome, come nell'esempio:

```
import math as m  
print(m.sin(m.pi/2))
```

## **Importare un modulo: la parola chiave as**

Nota: dopo l'esecuzione di un'importazione alias, il **nome del modulo originale diventa inaccessibile** e non deve essere utilizzato.

A sua volta, quando si usa la variante `from module import name` e si vuole cambiare il nome dell'entità, si crea un alias per l'entità. In questo modo, il nome verrà sostituito dall'alias scelto.

Ecco come si può fare:

### **`from module import name as alias`**

Come in precedenza, il nome originale (non equalizzato) diventa inaccessibile.

Il nome della frase come alias può essere ripetuto - utilizzare le virgole per separare le frasi moltiplicate, in questo modo:

### **`from module import n as a, m as b, o as c`**

L'esempio può sembrare un po' strano, ma funziona:

```
from math import pi as PI, sin as sine  
print(sine(PI/2))
```

## Lavorare con i moduli standard

Prima di passare in rassegna alcuni moduli standard di Python, vogliamo presentarvi la funzione `dir()`. Non ha nulla a che vedere con il comando `dir` che conoscete dalle console di Windows e Unix, poiché `dir()` non mostra il contenuto di una directory/cartella del disco, ma non si può negare che faccia qualcosa di molto simile: è in grado di rivelare tutti i nomi forniti attraverso un particolare modulo.

C'è una condizione: il modulo deve essere stato precedentemente importato nella sua interezza (cioè, non basta usare l'istruzione `import module - from module`).

La funzione restituisce un **elenco ordinato alfabeticamente** contenente tutti i nomi delle entità disponibili nel modulo identificati da un nome passato alla funzione come argomento:

`dir(modulo)`

Nota: se il nome del modulo ha un alias, si deve usare l'alias e non il nome originale.

L'uso della funzione all'interno di un normale script non ha molto senso, ma è comunque possibile.

Ad esempio, si può eseguire il seguente codice per stampare i nomi di tutte le entità del modulo matematico:

**for name in dir(math):**

**print(name, end="\t")**

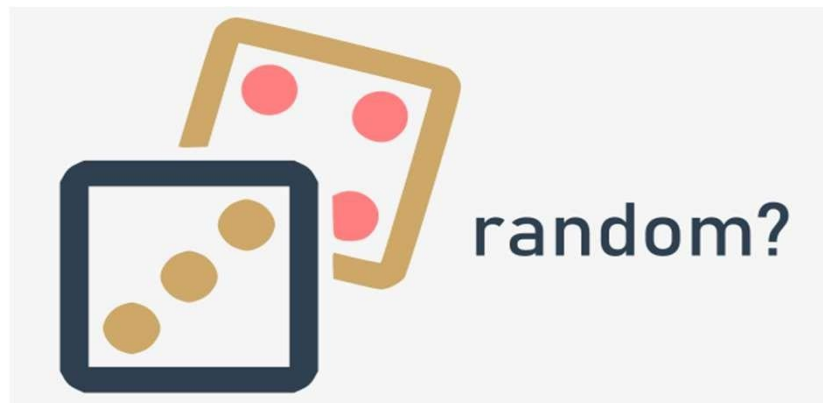
Il codice di esempio dovrebbe produrre il seguente risultato:

__doc__	__loader__	__name__	__package__	__spec__	acos	acosh	asin	asinh		
atan	atan2	atanh	ceil	copysign	cos	cosh	degrees	e	erf	erfc
exp	expm1	fabs	factorial	floor	fmod	frexp	fsum	gamma	hypot	
isfinite	isinf	isnan	ldexp	lgamma	log	log10	log1p	log2	modf	pi
pow	radians	sin	sinh	sqrt	tan	tanh	trunc			

## Esiste una vera casualità nei computer?

Un altro modulo degno di nota è quello denominato random.

Offre alcuni meccanismi che consentono di operare con **numeri pseudorandom**.



Si noti il prefisso **pseudo**: i numeri generati dai moduli possono sembrare casuali, nel senso che non si possono prevedere i loro valori successivi, ma non bisogna dimenticare che sono tutti calcolati con algoritmi molto raffinati.

Gli algoritmi non sono casuali, ma deterministici e prevedibili. Solo i processi fisici che sfuggono completamente al nostro controllo (come l'intensità della radiazione cosmica) possono essere utilizzati come fonte di dati casuali. I dati prodotti da computer deterministici non possono essere in alcun modo casuali.



## Esiste una vera casualità nei computer?

Un generatore di numeri casuali prende un valore chiamato **seme**, lo tratta come un valore di input, calcola un numero "casuale" sulla base di esso (il metodo dipende da un algoritmo scelto) e produce un **nuovo valore seme**. La durata di un ciclo in cui tutti i valori del seme sono unici può essere molto lunga, ma non è infinita: prima o poi i valori del seme inizieranno a ripetersi e anche i valori generatori si ripeteranno. Questo è normale. È una caratteristica, non un errore o un bug.

Il valore iniziale del seme, impostato all'avvio del programma, determina l'ordine in cui appariranno i valori generati.

Il fattore di casualità del processo può essere **incrementato impostando il seme con un numero preso dall'ora corrente**; questo può garantire che ogni lancio del programma parta da un valore di seme diverso (ergo, utilizzerà numeri casuali diversi).

Fortunatamente, tale inizializzazione viene effettuata da Python durante l'importazione del modulo.

## Funzioni selezionate dal modulo random

### La funzione random

La funzione più generale denominata `random()` (da non confondere con il nome del modulo) **produce un numero fluttuante x compreso nell'intervallo**  $(0,0, 1,0)$  - in altre parole:  $(0,0 \leq x < 1,0)$ .

Il programma di esempio qui sotto produrrà cinque valori pseudorandom - poiché i loro valori sono determinati dal valore di seme corrente (piuttosto imprevedibile), non è possibile indovinarli:  
da random importa random

```
for i in range(5):  
    print(random())
```

Eseguire il programma. Questo è il risultato:

0.9535768927411208

0.5312710096244534

0.8737691983477731

0.5896799172452125

0.02116716297022092

## La funzione seed

La funzione seed() permette di **impostare** direttamente **il seme del generatore**. Ne mostreremo due varianti:

- seed() - imposta il seme con l'ora corrente;
- seed(int\_value) - imposta il seme con il valore intero int\_value.

Abbiamo modificato il programma precedente, eliminando di fatto ogni traccia di casualità dal codice:

```
from random import random, seed
```

```
seed(0)
```

```
for i in range(5):  
    print(random())
```

Poiché il seme è sempre impostato con lo stesso valore, la sequenza di valori generati è sempre la stessa.

Eseguire il programma. Questo è il risultato:

```
0.844421851525  
0.75795440294  
0.420571580831  
0.258916750293  
0.511274721369
```

## **Le funzioni randrange e randint**

Se si desiderano valori casuali interi, una delle seguenti funzioni è più adatta:

**randrange(end)**

**randrange(inizio, fine)**

**randrange(inizio, fine, passo)**

**randint(sinistra, destra)**

Le prime tre invocazioni genereranno un numero intero preso (in modo pseudorandom) dall'intervallo (rispettivamente):

- range(end)
- range(beg, end)
- range(beg, end, step)

## Le funzioni randrange e randint

Si noti l'implicita **esclusione a destra!**

L'ultima funzione è un equivalente di `randrange(left, right+1)` - genera il valore intero `i`, che rientra nell'intervallo `[left, right]` (senza esclusione sul lato destro).

Osservate il codice nell'editor. Questo programma di esempio produrrà di conseguenza una riga composta da tre zeri e uno zero o uno al quarto posto.

```
from random import randrange, randint
```

```
print(randrange(1), end=' ')\nprint(randrange(0, 1), end=' ')\nprint(randrange(0, 1, 1), end=' ')\nprint(randint(0, 1))
```

## **Funzioni selezionate dal modulo random: continua**

Le funzioni precedenti hanno un importante svantaggio: possono produrre valori ripetuti anche se il numero di invocazioni successive non è superiore alla larghezza dell'intervallo specificato.

Guardate il codice qui sotto: è molto probabile che il programma produca un insieme di numeri in cui alcuni elementi non sono unici:

```
for i in range(10):  
    print(randint(1, 10), end=',')
```

Questo è ciò che abbiamo ottenuto in uno dei lanci:

9,4,5,4,5,8,9,4,8,4,

## Le funzioni di choice e sample

Come si può vedere, questo non è un buon strumento per generare i numeri di una lotteria. Fortunatamente, esiste una soluzione migliore rispetto alla scrittura del proprio codice per verificare l'unicità dei numeri "estratti". È una funzione che si chiama in modo molto suggestivo "choice":

- `choice(sequence)`
- `sample(sequence, elements_to_choose)`

La prima variante sceglie un elemento "casuale" dalla sequenza di input e lo restituisce.

Il secondo costruisce un elenco (un campione) costituito dagli elementi da scegliere "estratti" dalla sequenza di input.

In altre parole, la funzione sceglie alcuni degli elementi in ingresso, restituendo un elenco con la scelta. Gli elementi del campione sono disposti in ordine casuale. Nota: gli elementi\_da\_scegliere non devono essere maggiori della lunghezza della sequenza di input.

Guardate il codice qui sotto:

```
from random import choice, sample
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(choice(my_list))
print(sample(my_list, 5))
print(sample(my_list, 10))
```

## **Le funzioni di choice e sample**

Anche in questo caso, l'output del programma non è prevedibile. I nostri risultati sono stati questi:

4

[3, 1, 8, 9, 10]

[10, 8, 5, 1, 6, 4, 3, 9, 7, 2]



## **Ecosistema di packaging Python e come utilizzarlo**

Python è uno strumento molto potente. Lo pensano molte persone in tutto il mondo, che usano regolarmente Python per sviluppare le loro capacità in molti campi di attività completamente diversi. Ciò significa che Python è diventato **uno strumento interdisciplinare** impiegato in innumerevoli applicazioni. Non possiamo passare in rassegna tutti gli ambiti in cui Python mostra brillantemente le sue capacità, quindi ci limitiamo a raccontare quelli più impressionanti.

Innanzitutto, Python è diventato **un leader della ricerca sull'intelligenza artificiale**. Anche il data mining, una delle discipline scientifiche moderne più promettenti, utilizza Python. Matematici, psicologi, genetisti, meteorologi, linguisti: tutte queste persone usano già Python, o se non lo fanno ancora, siamo sicuri che lo faranno molto presto. Non si può sfuggire a questa tendenza.

Naturalmente, non ha senso che tutti gli utenti di Python scrivano il loro codice da zero, tenendoli perfettamente isolati dal mondo esterno e dai risultati degli altri programmatori. Sarebbe innaturale e controproducente.

La cosa migliore e più efficiente è consentire a tutti i membri della comunità Python di scambiare liberamente i propri codici e le proprie esperienze. In questo modello, nessuno è costretto a iniziare il lavoro da zero, perché c'è un'alta probabilità che qualcun altro abbia lavorato sullo stesso problema (o su uno molto simile).

Come sapete, Python è stato creato come software open-source e questo è anche un invito per tutti i programmatori a mantenere l'intero ecosistema Python come un ambiente aperto, amichevole e libero. Per far funzionare ed evolvere questo modello, è necessario fornire alcuni strumenti aggiuntivi, che aiutino i creatori a pubblicare, mantenere e curare il proprio codice.

Questi stessi strumenti dovrebbero aiutare gli utenti a utilizzare il codice, sia quello già esistente, sia quello nuovo che appare ogni giorno. Grazie a ciò, scrivere nuovo codice per nuove sfide non è come costruire una nuova casa, partendo dalle fondamenta.

## Ecosistema di packaging Python e come utilizzarlo

Inoltre, il programmatore è libero di modificare il codice di qualcun altro per adattarlo alle proprie esigenze, costruendo di fatto un prodotto completamente nuovo che può essere utilizzato da un altro sviluppatore. Il processo sembra non avere fine. Fortunatamente.

Per far girare questo mondo, è necessario creare e mantenere in movimento due entità fondamentali: **un archivio centralizzato** di tutti i pacchetti software disponibili e uno strumento che consenta agli utenti di **accedere all'archivio**. Entrambe queste entità esistono già e possono essere utilizzate in qualsiasi momento.

Il repository (o *repo* in breve) di cui abbiamo parlato prima si chiama PyPI (abbreviazione di Python Package Index) ed è gestito da un gruppo di lavoro chiamato Packaging Working Group, parte della Python Software Foundation, il cui compito principale è quello di supportare gli sviluppatori Python nella diffusione efficiente del codice.

Il loro sito web è disponibile qui:

<https://wiki.python.org/psf/PackagingWG>.

Il sito web di PyPI (amministrato da PWG) si trova all'indirizzo

: <https://pypi.org/>.

Quando ci abbiamo fatto un salto a marzo 2026, abbiamo scoperto che PyPI ospitava oltre 700.000 progetti, composti da oltre 11 milioni di file gestiti da oltre 1 milione di utenti.

Questi tre numeri dimostrano chiaramente la potenza della comunità Python e l'importanza della cooperazione tra sviluppatori.

## **Ecosistema di packaging Python e come utilizzarlo**

Dobbiamo sottolineare che PyPI non è l'unico repository Python esistente. Al contrario, ce ne sono molti, creati per progetti e guidati da molte comunità Python più o meno grandi. È probabile che un giorno voi e i vostri colleghi vogliate creare i vostri repository.

In ogni caso, PyPI è il più importante repo Python del mondo. Se modifichiamo un po' il detto classico, possiamo affermare che "tutte le strade di Python portano a PyPI", e non è affatto un'esagerazione.

## **Il repo PyPI: il negozio di formaggi**

Il repo di PyPI viene talvolta definito il negozio di formaggi. Davvero.

Vi sembra un po' strano? Non preoccupatevi, è tutto perfettamente innocente.

Ci riferiamo al repo come a un negozio, perché ci si reca lì per gli stessi motivi per cui si va in altri negozi: per soddisfare le proprie esigenze. Se si vuole del formaggio, si va al negozio di formaggi. Se si vuole un software, si va al negozio di software. Fortunatamente, l'analogia finisce qui: non c'è bisogno di denaro per prendere un software dal negozio di repo.

**PyPI è completamente gratuito** e potete semplicemente scegliere un codice e usarlo: non incontrerete né cassieri né guardie di sicurezza. Naturalmente, questo non vi esime dall'essere educati e onesti. Dovete rispettare tutti i termini di licenza, quindi non dimenticate di leggerli.

"Ok", direte voi, "ora il negozio è chiaro, ma cosa c'entra il formaggio con Python?".

The Cheese Shop è uno degli sketch più famosi dei Monty Python. Rappresenta l'avventura surreale di un inglese che cerca di comprare del formaggio. Purtroppo, il negozio che visita (immodestamente chiamato Ye National Cheese Emporium) non ha formaggio in magazzino.

Naturalmente, il messaggio è ironico. Come già sapete, PyPI ha un sacco di software in magazzino ed è disponibile 24 ore su 24, 7 giorni su 7. Ha tutto il diritto di identificarsi come *Ye International Python Software Emporium*.

## Il repo PyPI: il negozio di formaggi (continua)

PyPI è un negozio molto specifico, non solo perché offre tutti i suoi prodotti gratuitamente. Richiede anche uno strumento speciale per utilizzarlo.

Fortunatamente, questo strumento è anche gratuito, quindi se volete creare il vostro cheeseburger digitale utilizzando i prodotti offerti dal PyPI Shop, avrete bisogno di uno strumento gratuito chiamato *pip*.

No, non hai capito male. Solo *pip*. È un altro acronimo, naturalmente, ma la sua natura è più complessa rispetto al già citato PyPI, in quanto è un esempio di acronimo ricorsivo, il che significa che l'acronimo si riferisce a se stesso, il che significa che spiegarlo è un processo infinito.

Perché? Perché *pip* significa "***pip installs packages***", e il *pip* all'interno di "***pip installs packages***" significa "***pip installs packages***" e ...

Fermiamoci qui. Grazie per la collaborazione.

A proposito, esistono altri acronimi ricorsivi molto famosi. Uno di questi è *Linux*, che può essere interpretato come "*Linux is Not Unix*".

## Come installare *pip*

La domanda da porsi ora è: come ottenere un coltello da formaggio adeguato? In altre parole, come assicurarsi che la *pip* sia installata e pronta a lavorare?

La risposta più precisa è "dipende". Davvero.

Alcune installazioni di Python sono dotate di *pip*, altre no. Inoltre, non dipende solo dal sistema operativo utilizzato, anche se questo è un fattore molto importante.

Cominciamo con MS Windows.

### *pip* su MS Windows

Il programma di installazione di MS Windows Python contiene già *pip* e quindi non è necessario eseguire altri passi per installarlo. Purtroppo, se la variabile PATH non è configurata correttamente, *pip* potrebbe non essere disponibile.

Per verificare che non vi abbiamo ingannato, provate a fare così:  
aprire la console di Windows (*CMD* o *PowerShell*, come preferite)  
eseguire il seguente comando:

**pip --version**

## Dipendenze

Ora che siamo sicuri che *pip* è pronto ai nostri comandi, limiteremo la nostra attenzione solo a MS Windows, poiché il suo comportamento è (dovrebbe essere) lo stesso in tutti i sistemi operativi, ma prima di iniziare, dobbiamo spiegare una questione importante e parlare delle **dipendenze**.

Immaginate di aver creato una brillante applicazione Python chiamata *redsuspenders*, in grado di prevedere i tassi di cambio delle azioni con una precisione del 99% (a proposito, se ci riuscite davvero, contattatemi immediatamente).

Naturalmente, per raggiungere questo obiettivo si è utilizzato del codice esistente: ad esempio, la propria applicazione importa un pacchetto chiamato *nyse*, contenente alcune funzioni e classi cruciali. Inoltre, il pacchetto *nyse* importa un altro pacchetto chiamato *wallstreet*, mentre il pacchetto *wallstreet* importa altri due pacchetti essenziali chiamati *bull* e *bear*.

## Dipendenze

Come probabilmente avrete già intuito, le connessioni tra questi pacchetti sono cruciali e se qualcuno decide di usare il vostro codice (ma ricordate, abbiamo già dato la precedenza) dovrà anche assicurarsi che tutti i pacchetti richiesti siano al loro posto.

Per farla breve, possiamo dire che la **dipendenza è un fenomeno che si manifesta ogni volta che si utilizza un software che si basa su un altro software**. Si noti che la dipendenza può includere (e generalmente include) più di un livello di sviluppo del software.

Questo significa che un potenziale utente del pacchetto *nyse* è obbligato a rintracciare tutte le dipendenze e a installare manualmente tutti i pacchetti necessari? Sarebbe orribile, non è vero?

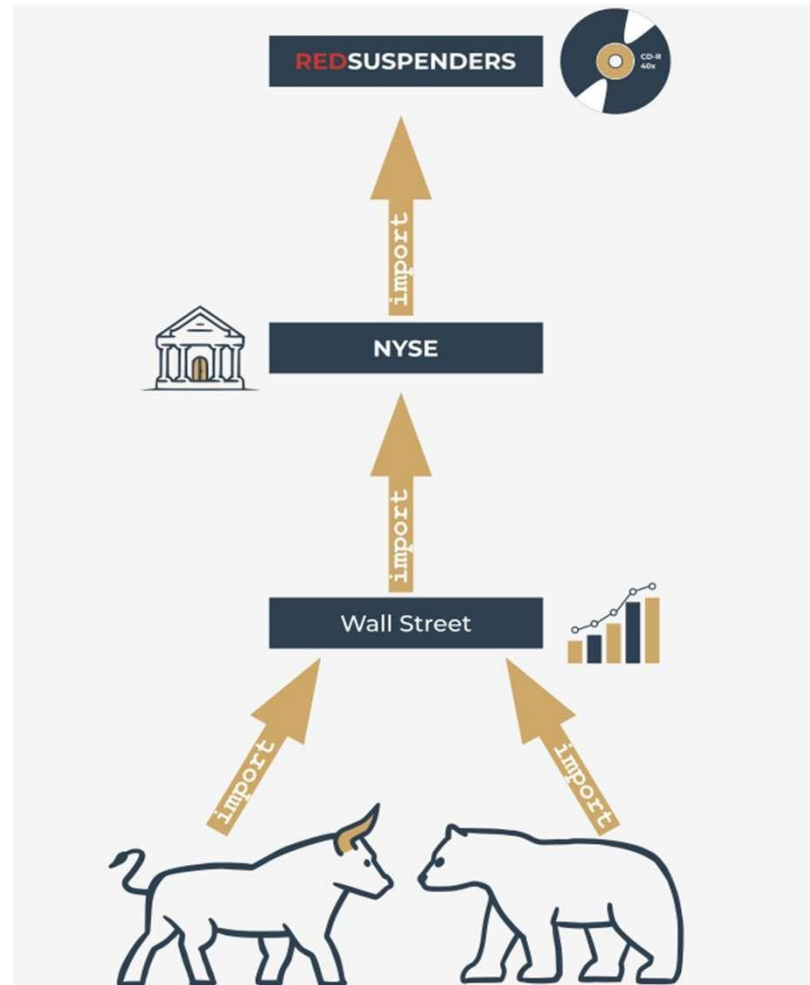
Sì, è decisamente orribile, quindi non dovrete essere sorpresi dal fatto che il processo per soddisfare faticosamente tutti i requisiti successivi abbia un nome proprio, che si chiama *inferno delle dipendenze*.

Come possiamo gestire questa situazione? Ogni utente è condannato a visitare l'inferno per eseguire il codice per la prima volta?

Fortunatamente no: il *pip* può fare tutto questo per voi. Davvero. È in grado di scoprire, identificare e risolvere tutte le dipendenze. Inoltre, lo fa nel modo più intelligente, evitando download e reinstallazioni inutili.



## Dipendenze

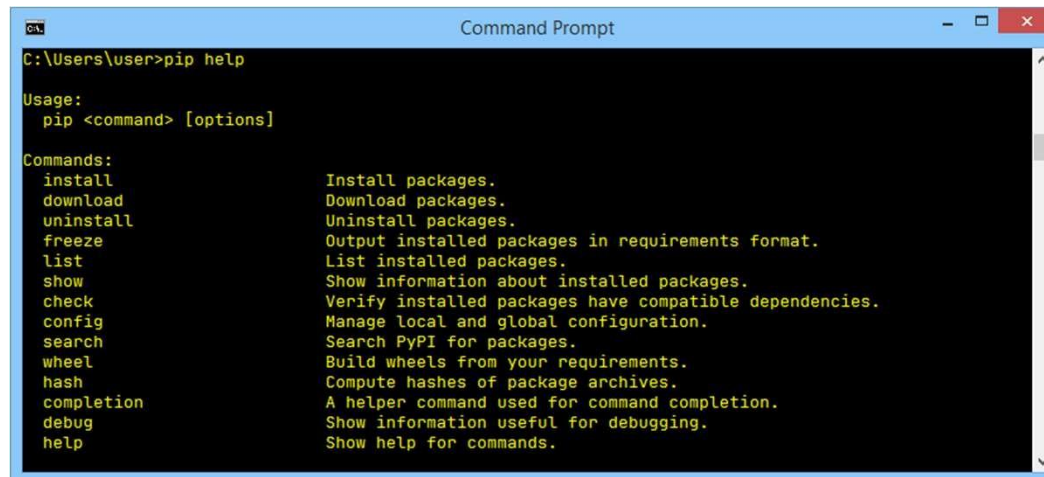


## Come utilizzare *pip*

Ora siamo pronti a chiedere a *pip* cosa può fare per noi. Facciamolo: lanciamo il seguente comando:

**pip help**

e attendere la risposta di *pip*. Ecco come appare:



```
C:\Users\user>pip help

Usage:
  pip <command> [options]

Commands:
  install          Install packages.
  download         Download packages.
  uninstall        Uninstall packages.
  freeze           Output installed packages in requirements format.
  list             List installed packages.
  show             Show information about installed packages.
  check            Verify installed packages have compatible dependencies.
  config           Manage local and global configuration.
  search           Search PyPI for packages.
  wheel            Build wheels from your requirements.
  hash             Compute hashes of package archives.
  completion       A helper command used for command completion.
  debug            Show information useful for debugging.
  help             Show help for commands.
```

Il comando

**pip list**

Mostra i pacchetti installati

## Come utilizzare *pip*

La potenza di *pip* deriva dal fatto che è in realtà una porta d'accesso all'universo del software Python. Grazie a ciò, è possibile sfogliare e installare qualsiasi pacchetto pronto all'uso raccolto nei repository PyPI. Non dimenticate che *pip* non è in grado di memorizzare localmente tutti i contenuti di PyPI (non è necessario e sarebbe antieconomico).

In effetti, *pip* utilizza Internet per interrogare PyPI e scaricare i dati richiesti. Ciò significa che è necessario avere una connessione di rete funzionante ogni volta che si chiede a *pip* qualcosa che possa comportare interazioni dirette con l'infrastruttura di PyPI.

## Come usare *pip*: continua

Supponendo che siate decisi a installare un pacchetto specifico, potete usare pip per installare il pacchetto sul vostro computer.

Due possibili scenari possono essere messi in atto ora:

- volete installare un nuovo pacchetto solo per voi - non sarà disponibile per nessun altro utente (account) esistente sul vostro computer; questa procedura è l'unica disponibile se non potete elevare i vostri permessi e agire come amministratore di sistema;
- avete deciso di installare un nuovo pacchetto a livello di sistema - avete i diritti di amministrazione e non avete paura di usarli.

Per distinguere tra queste due azioni, pip utilizza un'opzione dedicata denominata `--user` (notare il doppio trattino). La presenza di questa opzione indica a pip di agire localmente per conto dell'utente (non amministrativo).

Se non lo si aggiunge, pip presume che siate un amministratore di sistema e non farà nulla per correggervi se non lo siete.

Nel nostro caso, installeremo un pacchetto chiamato pygame: si tratta di una libreria estesa e complessa che consente ai programmatori di sviluppare giochi per computer utilizzando Python.

Il progetto è in fase di sviluppo dal 2000, quindi si tratta di un codice maturo e affidabile. Se volete saperne di più sul progetto e sulla comunità che lo gestisce, visitate il sito <https://www.pygame.org>.

Se siete un amministratore di sistema, potete installare pygame usando il seguente comando:

**pip install pygame-ce** (oppure in caso di problemi di PATH **py -m pip install pygame-ce**)

Se non siete amministratori o non volete ingrassare il vostro sistema operativo installando pygame a livello di sistema, potete installarlo solo per voi:

**pip install --user pygame-ce** (oppure in caso di problemi di PATH **py -m pip install --user pygame-ce**)

## Come usare pip: un semplice programma di prova

Ora che *pygame* è finalmente accessibile, possiamo provare a utilizzarlo in un programma di prova molto semplice.

```
import pygame
```

```
run = True
```

```
width = 400
```

```
height = 100
```

```
pygame.init()
```

```
screen = pygame.display.set_mode((width, height))
```

```
font = pygame.font.SysFont(None, 48)
```

```
text = font.render("Welcome to pygame", True, (255, 255, 255))
```

```
screen.blit(text, ((width - text.get_width()) // 2, (height - text.get_height()) // 2))
```

```
pygame.display.flip()
```

```
while run:
```

```
    for event in pygame.event.get():
```

```
        if event.type == pygame.QUIT\
```

```
        or event.type == pygame.MOUSEBUTTONUP\
```

```
        or event.type == pygame.KEYUP:
```

```
            run = False
```

## Come usare pip: un semplice programma di prova

Commentiamolo brevemente.

riga 1: importare *pygame* e lasciare che ci serva;

riga 3: il programma viene eseguito finché la variabile *run* è *True*;

righe 4 e 5: determinano le dimensioni della finestra;

riga 6: inizializzare l'ambiente *pygame*;

riga 7: preparare la finestra dell'applicazione e impostarne le dimensioni;

riga 8: creare un oggetto che rappresenti il carattere predefinito di 48 punti;

Riga 9: creare un oggetto che rappresenti un testo dato; il testo sarà anti-alias (Vero) e bianco (255,255,255)

riga 10: inserisce il testo nel buffer dello schermo (attualmente invisibile);

riga 11: capovolgere i buffer dello schermo per rendere visibile il testo;

riga 12: il ciclo principale di *pygame* inizia qui;

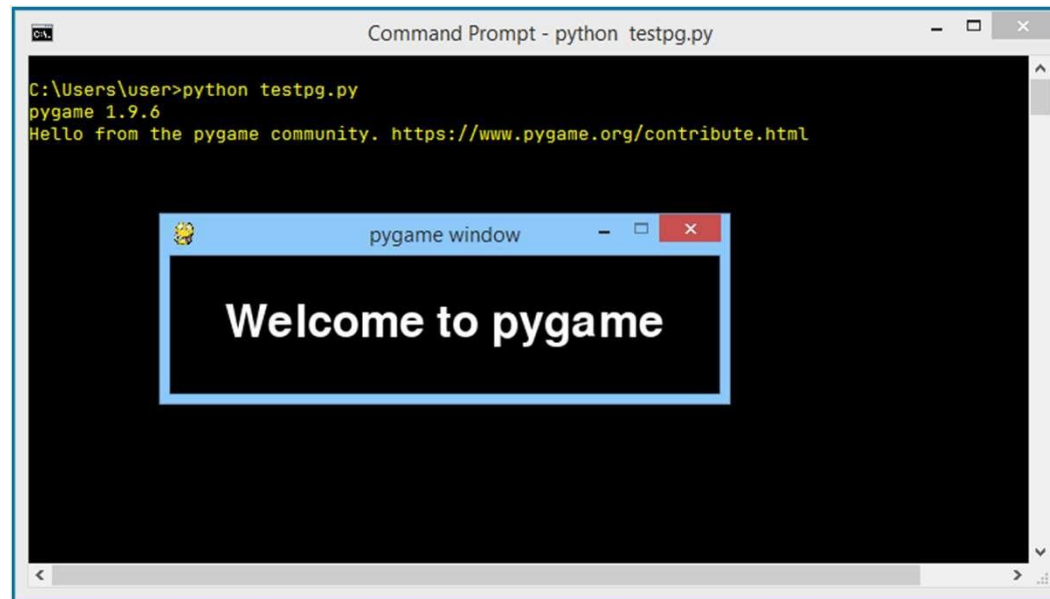
riga 13: ottiene un elenco di tutti gli eventi *pygame* in sospenso;

righe da 14 a 16: controllano se l'utente ha chiuso la finestra o ha fatto clic da qualche parte al suo interno o ha premuto un tasto;

riga 15: se sì, interrompere l'esecuzione del codice.

## Come usare pip: un semplice programma di prova

Questo è ciò che ci aspettiamo dal nostro codice:



The image shows a Windows Command Prompt window titled "Command Prompt - python testpg.py". The prompt shows the execution of the command `C:\Users\user>python testpg.py`. The output of the script is displayed in yellow text: `pygame 1.9.6` followed by `Hello from the pygame community. https://www.pygame.org/contribute.html`. Overlaid on the Command Prompt is a smaller window titled "pygame window" with a blue border. This window has a black background and displays the text "Welcome to pygame" in a large, white, sans-serif font.

## Come usare pip: un semplice programma di prova

L'installazione con pip ha due importanti funzioni aggiuntive:

è in grado di **aggiornare** un pacchetto installato localmente; ad esempio, se si vuole essere sicuri di utilizzare l'ultima versione di un particolare pacchetto, si può eseguire il seguente comando:

**pip install -U nome\_pacchetto**

dove -U significa aggiornamento. Nota: questa forma del comando utilizza l'opzione --user per lo stesso scopo presentato in precedenza;

è in grado di **installare una versione selezionata dall'utente** di un pacchetto (*pip* installa la versione **più recente** disponibile per impostazione predefinita); per raggiungere questo obiettivo si deve usare la seguente sintassi:

**pip install nome\_pacchetto==versione\_pacchetto**

(notare il doppio segno di uguale) ad es,

**pip install pygame==1.9.2**



## Come usare pip: un semplice programma di prova

Se uno dei pacchetti attualmente installati **non è più necessario** e si vuole sbarazzarsene, sarà utile anche *pip*. Il suo comando di disinstallazione eseguirà tutti i passaggi necessari.

La sintassi richiesta è chiara e semplice:

**pip uninstall nome\_pacchetto**

quindi se non si vuole più *pygame* si può eseguire il seguente comando:

**pip uninstall pygame**

## Come usare pip: un semplice programma di prova

L'elenco delle principali attività **del pip** è il seguente:

- `pip help operation` - mostra una breve descrizione di pip;
- `pip list` - mostra l'elenco dei pacchetti attualmente installati;
- `pip show package_name` - mostra le informazioni sul *nome del pacchetto*, comprese le sue dipendenze;
- `pip search anystring` - cerca nelle directory di PyPI i pacchetti il cui nome contiene *una qualsiasi stringa*;
- `pip install name` - installa *name* a livello di sistema (aspettatevi problemi se non avete i diritti di amministrazione);
- `pip install --user name` - installa *il nome* solo per voi; nessun altro utente della vostra piattaforma potrà usarlo;
- `pip install -U name` - aggiorna il pacchetto precedentemente installato;
- `pip uninstall name` - disinstalla il pacchetto precedentemente installato;

## Le stringhe

Facciamo un breve ripasso della natura delle stringhe di Python.

Innanzitutto, le stringhe di Python (o semplicemente le stringhe, dato che non parleremo delle stringhe di altri linguaggi) sono **sequenze immutabili**.

È molto importante notare questo aspetto, perché significa che dovete aspettarvi un comportamento familiare da parte loro.

Analizziamo il codice seguente per capire di cosa stiamo parlando:

**# Esempio 1**

```
word = 'by'
```

```
print(len(word))
```

**# Esempio 2**

```
empty = ''
```

```
print(len(empty))
```

**# Esempio 3**

```
i_am = 'I\'m'
```

```
print(len(i_am))
```

Guardate l'**esempio 1**. La funzione `len()` utilizzata per le stringhe restituisce il numero di caratteri contenuti negli argomenti. Lo snippet restituisce 2.

Qualsiasi stringa può essere vuota. Allora la sua lunghezza è 0, proprio come nell'**Esempio 2**.

Non dimenticate che un backslash (`\`) usato come carattere di escape non è incluso nella lunghezza totale della stringa. Il codice dell'**esempio 3**, quindi, produce 3.

Eseguire i tre codici di esempio e verificare.

## Stringhe multilinea

Questo è un ottimo momento per mostrare un altro modo di specificare le stringhe all'interno del codice sorgente Python. Notate che la sintassi che già conoscete non vi permetterà di utilizzare una stringa che occupi più di una riga di testo.

Per questo motivo, il codice qui riportato è errato:

```
multiline = 'Line #1  
Line #2'  
print(len(multiline))
```

Fortunatamente, per questo tipo di stringhe, Python offre una sintassi separata, comoda e semplice. Guardate il codice

```
multiline = '''Line #1  
Line #2'''  
print(len(multiline))
```

Ecco come appare.

Come si può notare, la stringa inizia con **tre apostrofi** e non con uno. Lo stesso apostrofo triplo viene utilizzato per terminarla.

Il numero di righe di testo inserite in questa stringa è arbitrario.

Lo snippet produce 15.

## Stringhe multilinea

Contate attentamente i caratteri. Il risultato è corretto o no? A prima vista sembra corretto, ma quando si contano i caratteri non lo è affatto.

La riga n. 1 contiene sette caratteri. Due righe di questo tipo comprendono 14 caratteri. Abbiamo perso un carattere? Dove? Come?

No, non l'abbiamo fatto.

**Il carattere mancante è semplicemente invisibile: è uno spazio bianco.** Si trova tra le due righe di testo.

È indicato come: \n.

Anche le stringhe multilinea possono essere delimitate da **tripli apici**, proprio come in questo caso:

```
multilinea = """Linea #1
```

```
Linea #2"""
```

```
print(len(multiline))
```

## Operazioni sulle stringhe

Come altri tipi di dati, anche le stringhe hanno un proprio insieme di operazioni consentite, sebbene siano piuttosto limitate rispetto ai numeri.

In generale, le stringhe possono essere:

- **concatenate** (unite)
- **replicate**

La prima operazione viene eseguita dall'operatore + (attenzione: non è un'addizione), mentre la seconda dall'operatore \* (attenzione: non è una moltiplicazione).

La possibilità di utilizzare lo stesso operatore per tipi di dati completamente diversi (come i numeri o le stringhe) si chiama **overloading** (un operatore viene sovraccaricato con compiti diversi).

Analizzare l'esempio:

```
str1 = 'a'
```

```
str2 = 'b'
```

```
print(str1 + str2)
```

```
print(str2 + str1)
```

```
print(5 * 'a')
```

```
print('b' * 4)
```

## Operazioni sulle stringhe

L'operatore + usato con due o più stringhe produce una nuova stringa contenente tutti i caratteri dei suoi argomenti (nota: l'ordine è importante - questo + sovraccaricato, a differenza della sua versione numerica, **non è commutativo**)

L'operatore \* ha bisogno di una stringa e di un numero come argomenti; in questo caso, l'ordine non ha importanza: si può mettere il numero prima della stringa o viceversa, il risultato sarà lo stesso: una nuova stringa creata dall'ennesima replica della stringa dell'argomento.

Lo snippet produce il seguente risultato:

ab

ba

aaaaa

bbbb

Nota: le varianti di scorciatoia degli operatori precedenti sono applicabili anche alle stringhe (+= e \*=).

## Operazioni sulle stringhe: ord()

Se si vuole **conoscere il valore del codice ASCII/UNICODE di un carattere specifico**, si può usare una funzione chiamata `ord()` (come *ordinale*).

La funzione deve avere **come argomento una stringa di un solo carattere**; la violazione di questo requisito causa un'eccezione `TypeError` e restituisce un numero che rappresenta il codice dell'argomento.

Guardate il codice

### # Dimostrazione della funzione ord()

```
function. char_1 = 'a'
```

```
char_2 = ' ' # spazio
```

```
print(ord(char_1))
```

```
print(ord(char_2))
```

ed eseguitelo. Lo snippet viene visualizzato: 97



## Operazioni sulle stringhe: chr()

Se si conosce il codice (numero) e si vuole ottenere il carattere corrispondente, si può usare una funzione chiamata chr().

La funzione **prende il codice e restituisce il suo carattere**.

L'invocazione con un argomento non valido (ad esempio, un punto di codice negativo o non valido) causa eccezioni ValueError o TypeError.

Eseguire il codice

**# Dimostrazione della funzione chr().**

```
print(chr(97))
```

```
print(chr(945))
```

Lo snippet di esempio viene visualizzato: a

α

Nota: chr(ord(x)) == x

ord(chr(x)) == x

## Stringhe come sequenze: indicizzazione

Vi abbiamo già detto che **le stringhe in Python sono sequenze**. È ora di mostrarvi cosa significa in realtà. Le stringhe non sono elenchi, ma **si possono trattare come tali in molti casi particolari**.

Ad esempio, se si vuole accedere a uno qualsiasi dei caratteri di una stringa, lo si può fare utilizzando l'**indicizzazione**, come nell'esempio seguente. Eseguire il programma:

```
# Indecizzando le stringhe.  
the_string = 'silly walks'  
for ix in range(len(the_string)):  
    print(the_string[ix], end=' ')  
print()
```

Fate attenzione: non provate a passare i confini di una stringa, causerà un'eccezione. L'esempio di output è il seguente:

```
silly walks
```

Anche gli indici negativi si comportano come previsto

## **Stringhe come sequenze: iterazione**

Anche l'**iterazione delle stringhe** funziona. Guardate l'esempio seguente

```
the_string = 'silly walks'
```

```
for character in the_string:  
    print(character, end=' ')
```

```
print()
```

## Slices

Inoltre, tutto ciò che sapete sulle slices è ancora utilizzabile.

Abbiamo raccolto alcuni esempi che mostrano il funzionamento delle fette nel mondo delle stringhe. Guardate il codice analizzatelo ed eseguitelo.

### # Slices

```
alpha = "abdefg"  
print(alpha[1:3])  
print(alpha[3:])  
print(alpha[:3])  
print(alpha[3:-2])  
print(alpha[-3:4])  
print(alpha[:2])
```

Non vedrete nulla di nuovo nell'esempio, ma vogliamo che siate sicuri di poter spiegare tutte le righe del codice.

L'output del codice è:

```
bd  
efg  
abd  
e  
e  
adf
```

## Gli operatori in e non in

### L'operatore in

L'operatore in non dovrebbe sorprendere se applicato alle stringhe: **controlla** semplicemente **se il suo argomento di sinistra (una stringa) si trova in un punto qualsiasi dell'argomento di destra (un'altra stringa)**.

Il risultato del controllo è semplicemente Vero o Falso.

Guardate il programma di esempio qui sotto. Ecco come funziona l'operatore in:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"  
print("f" in alphabet)  
print("F" in alphabet)  
print("1" in alphabet)  
print("ghi" in alphabet)  
print("Xyz" in alphabet)
```

L'esempio di output è il seguente:

True

False

False

True

False

## L'operatore not in

Come probabilmente si sospetta, l'operatore not in è applicabile anche in questo caso.  
Ecco come funziona:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"  
print("f" not in alphabet)  
print("F" not in alphabet)  
print("1" not in alphabet)  
print("ghi" not in alphabet)  
print("Xyz" not in alphabet)
```

L'esempio di output è il seguente:

False

True

True

False

True

## Le stringhe Python sono immutabili

Abbiamo anche detto che le **stringhe** di Python **sono immutabili**. Si tratta di una caratteristica molto importante. Che cosa significa?

Questo significa principalmente che la somiglianza tra stringhe ed elenchi è limitata. Non tutto ciò che si può fare con un elenco può essere fatto con una stringa.

La prima importante differenza **non consente di utilizzare l'istruzione del per rimuovere qualcosa da una stringa**. L'esempio qui riportato non funziona:

```
alfabeto = "abcdefghijklmnopqrstuvwxyz"  
del alfabeto[0]
```

L'unica cosa che si può fare con del e una stringa è **rimuovere la stringa nel suo complesso**. Provate a farlo.

**Le stringhe Python non hanno il metodo** append(): non è possibile espanderle in alcun modo.

L'esempio che segue è errato:

```
alfabeto = "abcdefghijklmnopqrstuvwxyz"  
alfabeto.append("A")
```

con l'assenza del metodo append(), anche **il metodo** insert() **è illegale**:

```
alfabeto = "abcdefghijklmnopqrstuvwxyz"  
alfabeto.inserire(0, "A")
```

## Operazioni sulle stringhe: continua

Non pensate che l'immutabilità di una stringa limiti la vostra capacità di operare con le stringhe.

L'unica conseguenza è che bisogna ricordarsene e implementare il codice in modo leggermente diverso - si veda l'esempio di codice nell'editor.

Questa forma di codice è del tutto accettabile,

```
alphabet = "bcdefghijklmnopqrstuvwxy"  
alphabet = "a" + alphabet  
alphabet = alphabet + "z"  
print(alphabet)
```

funzionerà senza infrangere le regole di Python e porterà sullo schermo l'intero alfabeto latino:

```
abcdefghijklmnopqrstuvwxyz
```

Ci si può chiedere se la **creazione di una nuova copia di una stringa ogni volta che se ne modifica il contenuto peggiori l'efficacia del codice.**

Sì, è così. Un po'. Ma non è affatto un problema.



## Operazioni sulle stringhe: min()

Ora che avete capito che le stringhe sono sequenze, possiamo mostrarvi alcune funzionalità di sequenza meno ovvie. Le presenteremo utilizzando le stringhe, ma non dimenticate che anche le liste possono adottare gli stessi trucchi.

Cominciamo con una funzione chiamata min().

**# Dimostrazione min() - Esempio 1:**

```
print(min("aAbByYzZ"))
```

**# Dimostrazione min() - Esempi 2 & 3:**

```
t = 'The Knights Who Say "Ni!'"
```

```
print('[' + min(t) + ']')
```

```
t = [0, 1, 2]
```

```
print(min(t))
```

La funzione **trova l'elemento minimo della sequenza passata come argomento**. C'è una condizione: la sequenza (stringa, elenco, non importa) **non può essere vuota**, altrimenti si otterrà un'eccezione ValueError.

Il programma dell'**esempio 1** produce le uscite:

A

Nota: è una A maiuscola. Perché? Ricordate la tabella ASCII: quali lettere occupano le prime posizioni - maiuscole o minuscole?

## Operazioni sulle stringhe: min()

Abbiamo preparato altri due esempi da analizzare: **Esempi 2 e 3**.

Come si può vedere, non presentano solo stringhe. L'output previsto è il seguente:

```
[ ]  
0
```

Nota: abbiamo utilizzato le parentesi quadre per evitare che lo spazio venga trascurato sullo schermo.

## Operazioni sulle stringhe: max()

Allo stesso modo, una funzione chiamata max() **trova l'elemento massimo della sequenza**.  
Guardate l'**Esempio 1**

```
# Dimostrazione max() - Esempio 1:  
print(max("aAbByYzZ"))
```

```
# Dimostrazione max() - Esempi 2 & 3:  
t = 'The Knights Who Say "Ni!'"  
print('[' + max(t) + ']')
```

```
t = [0, 1, 2]  
print(max(t))
```

Il programma di esempio produce:

z

Nota: è una z minuscola.

Vediamo ora la funzione max() applicata agli stessi dati di prima. Guardate gli **esempi 2 e 3**.

L'output previsto è:

[y]

2

## Operazioni sulle stringhe: il metodo index()

Il metodo index() (è un metodo, non una funzione) **cerca la sequenza dall'inizio, per trovare il primo elemento del valore specificato nel suo argomento.**

Nota: l'elemento cercato deve essere presente nella sequenza; la **sua assenza causerà un'eccezione ValueError.**

Il metodo restituisce l'**indice della prima occorrenza dell'argomento** (ciò significa che il risultato minimo possibile è 0, mentre il massimo è la lunghezza dell'argomento diminuita di 1).

Pertanto, l'esempio

**# Dimostrazione del metodo index():**

```
print("aAbByYzZaA".index("b"))
```

```
print("aAbByYzZaA".index("Z"))
```

```
print("aAbByYzZaA".index("A"))
```

produce risultati:

2

7

1

## **Operazioni sulle stringhe: la funzione list()**

La funzione list() **prende il suo argomento (una stringa) e crea un nuovo elenco contenente tutti i caratteri della stringa, uno per elemento dell'elenco.**

Nota: non è una funzione strettamente legata alle stringhe: list() è in grado di creare un nuovo elenco da molte altre entità (ad esempio, da tuple e dizionari).

## Operazioni sulle stringhe: il metodo count()

Il metodo count() **conta tutte le occorrenze dell'elemento all'interno della sequenza**. L'assenza di tali elementi non causa alcun problema.

Guardate il secondo esempio

```
# Dimostrazione della funzione list():  
print(list("abcabc"))
```

```
# Dimostrazione del metodo count():  
print("abcabc".count("b"))  
print('abcabc'.count("d"))
```

Riuscite a indovinare l'output?

E':

2

0

**uscita**

Inoltre, le stringhe Python hanno un numero significativo di metodi destinati esclusivamente all'elaborazione dei caratteri. Non aspettatevi che funzionino con altre collezioni. L'elenco completo è presentato qui:

<https://docs.python.org/3.4/library/stdtypes.html#string-methods>.

### Esercizio 1

Qual è la lunghezza della seguente stringa, supponendo che non ci siano spazi bianchi tra le virgolette?

```
""  
""
```

### Esercizio 2

Qual è l'output previsto del seguente codice?

```
print(len("\n\n"))  
print(len("\n\n"))
```

### Esercizio 3

Qual è l'output previsto del seguente codice?

```
for ch in "abc":  
    print(chr(ord(ch) + 1), end="")
```

## Il metodo capitalize()

Esaminiamo alcuni metodi standard di Python per le stringhe. Li esamineremo in ordine alfabetico - a dire il vero, ogni ordine ha tanti svantaggi quanti vantaggi, quindi la scelta può anche essere casuale.

Il metodo capitalize() fa esattamente quello che dice: **crea una nuova stringa con i caratteri presi dalla stringa di partenza**, ma cerca di modificarli nel modo seguente:

- **se il primo carattere all'interno della stringa è una lettera** (nota: il primo carattere è un elemento con indice uguale a 0, non solo il primo carattere visibile), **verrà convertito in maiuscolo**;
- **tutte le lettere rimanenti della stringa saranno convertite in minuscole.**

Non dimenticate che:

- la stringa originale (da cui viene invocato il metodo) non viene modificata in alcun modo (l'immutabilità di una stringa deve essere rispettata senza riserve)
- la stringa modificata (in questo caso con la lettera maiuscola) viene restituita come risultato; se non la si utilizza in alcun modo (assegnandola a una variabile o passandola a una funzione/metodo), sparirà senza lasciare traccia.

Nota: i metodi non devono essere invocati solo da variabili. Possono essere invocati direttamente da letterali di stringa. Utilizzeremo questa convenzione regolarmente: semplificherà gli esempi, in quanto gli aspetti più importanti non scompariranno tra assegnazioni non necessarie.



## Il metodo `capitalize()`

Guardate l'esempio

```
print("Alpha".capitalize())  
print('ALPHA'.capitalize())  
print(' Alpha'.capitalize())  
print('123'.capitalize())  
print("αβγδ".capitalize())
```

Ecco cosa stampa:

Abcd

## Il metodo center()

La variante a un parametro del metodo center() crea una copia della stringa originale, cercando di centrarla all'interno di un campo di larghezza specificata.

La centratura viene effettuata **aggiungendo degli spazi prima e dopo la stringa**.

Non aspettatevi che questo metodo dimostri abilità sofisticate. È piuttosto semplice.

L'esempio utilizza le parentesi per mostrare chiaramente dove inizia e termina la stringa centrata.

**# Dimostrazione del metodo center():**

```
print([' + 'alpha'.center(10) + ''])
```

L'output è il seguente:

```
[ alpha ]
```

Se la lunghezza del campo di destinazione è troppo piccola per adattarsi alla stringa, viene restituita la stringa originale.

È possibile vedere il metodo center() in altri esempi qui:

```
print([' + 'Beta'.center(2) + ''])
```

```
print([' + 'Beta'.center(4) + ''])
```

```
print([' + 'Beta'.center(6) + ''])
```

## Il metodo center()

**La variante a due parametri di center() utilizza il carattere del secondo argomento, invece di uno spazio.**

Analizzate l'esempio seguente:

```
print([' + 'gamma'.center(20, '*') + ''])
```

Per questo motivo l'output appare ora come questo:

```
[*****gamma*****]
```

## **Il metodo endswith()**

Il metodo endswith() **verifica se la stringa data termina con l'argomento specificato e restituisce True o False**, a seconda del risultato del controllo.

Nota: la sottostringa deve aderire all'ultimo carattere della stringa, non può trovarsi semplicemente vicino alla fine della stringa.

Guardate l'esempio, analizzatelo ed eseguitelo.

### **# Dimostrazione del metodo endswith():**

```
if "epsilon".endswith("on"):
```

```
    print("yes")
```

```
else:
```

```
    print("no")
```

L'output è:

yes

Ora dovrete essere in grado di prevedere l'output dello snippet sottostante:

```
t = "zeta"
```

```
print(t.endswith("a"))
```

```
print(t.endswith("A"))
```

```
print(t.endswith("et"))
```

```
print(t.endswith("eta"))
```

## Il metodo find()

Il metodo find() è simile a index(), che già conoscete: **cerca una sottostringa e restituisce l'indice della prima occorrenza di questa sottostringa**, ma:

- è più sicuro: **non genera un errore per un argomento che contiene una sottostringa inesistente** (restituisce -1 in quel caso)
- **funziona solo con le stringhe**: non provate ad applicarlo a qualsiasi altra sequenza.

Guardate il codice

**# Dimostrazione del metodo find():**

```
print("Eta".find("ta"))  
print("Eta".find("mma"))
```

Ecco come si può utilizzare.

L'esempio stampa:

```
1  
-1
```

## Il metodo find()

Nota: non utilizzate find() se volete controllare solo se un singolo carattere si trova all'interno di una stringa; l'operatore in sarà molto più veloce.

Ecco un altro esempio:

```
t = 'theta'
print(t.find('eta'))
print(t.find('et'))
print(t.find('the'))
print(t.find('ha'))
```

Riuscite a prevedere l'output? Eseguitelo e verificate le vostre previsioni.

Se si vuole eseguire la ricerca non dall'inizio della stringa, ma **da qualsiasi posizione**, si può usare una **variante a due parametri** del metodo find(). Guardate l'esempio:

```
print('kappa'.find('a', 2))
```

Il secondo argomento **specifica l'indice dal quale verrà avviata la ricerca** (non deve necessariamente rientrare nella stringa).

Tra le due lettere *a*, verrà trovata solo la seconda. Eseguite lo snippet e verificate.

## Il metodo find()

È possibile utilizzare il metodo find() per cercare tutte le occorrenze della sottostringa, come in questo caso:

```
the_text = """A variation of the ordinary lorem ipsum  
text has been used in typesetting since the 1960s  
or earlier, when it was popularized by advertisements  
for Letraset transfer sheets. It was introduced to  
the Information Age in the mid-1980s by the Aldus Corporation,  
which employed it in graphics and word-processing templates  
for its desktop publishing program PageMaker (from Wikipedia)"""
```

```
fnd = the_text.find('the')  
while fnd != -1:  
    print(fnd)  
    fnd = the_text.find('the', fnd + 1)
```

Il codice stampa gli indici di tutte le occorrenze dell'articolo *the* e il suo risultato è simile a questo:

```
15  
80  
198  
221  
238
```

## Il metodo find()

Esiste anche una versione **a tre parametri del metodo** find(): il terzo parametro **indica il primo indice che non verrà preso in considerazione durante la ricerca** (in realtà è il limite superiore della ricerca).

Guardate il nostro esempio qui sotto:

```
print('kappa'.find('a', 1, 4))  
print('kappa'.find('a', 2, 4))
```

Il secondo argomento specifica l'indice dal quale verrà avviata la ricerca (non deve necessariamente rientrare nella stringa).

Pertanto, l'esempio modificato produce risultati:

```
1  
-1
```

(*a* non può essere trovato entro i limiti di ricerca indicati nella seconda print()).



## Il metodo isalnum()

Il metodo senza parametri denominato isalnum() **verifica se la stringa contiene solo cifre o caratteri alfabetici (lettere) e restituisce True o False** a seconda del risultato.

Guardate l'esempio

**# Dimostrazione del metodo isalnum():**

```
print('lambda30'.isalnum())
```

```
print('lambda'.isalnum())
```

```
print('30'.isalnum())
```

```
print('@'.isalnum())
```

```
print('lambda_30'.isalnum())
```

```
print('').isalnum())
```

ed eseguitelo.

Nota: qualsiasi elemento di stringa che non sia una cifra o una lettera fa sì che il metodo restituisca False. Anche una stringa vuota lo fa.

L'esempio di output è il seguente:

True

True

True

False

False

False

## Il metodo `isalnum()`

Altri tre esempi intriganti sono qui:

```
t = 'Six lambdas'  
print(t.isalnum())
```

```
t = 'AβΓδ'  
print(t.isalnum())
```

```
t = '20E1'  
print(t.isalnum())
```

Eseguiteli e verificate i risultati.

Suggerimento: la causa del primo risultato è uno spazio, non è una cifra né una lettera

## Il metodo isdigit()

A sua volta, il metodo isdigit() esamina **solo** le **cifre**: qualsiasi altro risultato produce False.

**# Esempio 1: Dimostrazione del metodo isalpha():**

```
print("Moooo".isalpha())
```

```
print('Mu40'.isalpha())
```

**# Esempio 2: Dimostrazione del metodo isdigit():**

```
print('2018'.isdigit())
```

```
print("Year2019".isdigit())
```

Guardate l'Esempio 2: il suo output è:

True

False

## Il metodo islower()

Il metodo islower() è una variante pignola di isalpha(): accetta **solo lettere minuscole**.

## Il metodo isspace()

Il metodo isspace() **identifica solo gli spazi bianchi**, ignorando qualsiasi altro carattere (il risultato è quindi False).

## Il metodo isupper()

Il metodo isupper() è la versione maiuscola di islower(): si concentra **solo** sulle **lettere maiuscole**.  
Di nuovo, guardate il codice

**# Esempio 1: Dimostrazione del metodo islower():**

```
print("Moooo".islower())  
print('moooo'.islower())
```

**# Esempio 2: Dimostrazione del metodo isspace():**

```
print(' \n '.isspace())  
print(" ".isspace())  
print("mooo mooo mooo".isspace())
```

**# Esempio 3: Dimostrazione del metodo isupper():**

```
print("Moooo".isupper())  
print('moooo'.isupper())  
print('MOOOO'.isupper())
```

l'esempio 3 produce il seguente risultato

False

False

True

## Il metodo join()

Il metodo join() è piuttosto complicato, quindi vi guideremo passo dopo passo alla sua scoperta:

- Come suggerisce il nome, il metodo **esegue un join**: si aspetta un argomento come una lista; deve essere garantito che tutti gli elementi della lista siano stringhe, altrimenti il metodo solleverà un'eccezione TypeError;
- tutti gli elementi dell'elenco saranno **uniti in una stringa**, ma...
- ...la stringa da cui è stato invocato il metodo viene **usata come separatore**, inserita tra le stringhe;
- la stringa appena creata viene restituita come risultato.

### # Dimostrazione del metodo join():

```
print(",".join(["omicron", "pi", "rho"]))
```

- il metodo join() viene invocato all'interno di una stringa contenente una virgola (la stringa può essere arbitrariamente lunga, oppure vuota)
- l'argomento del join è un elenco contenente tre stringhe;
- il metodo restituisce una nuova stringa.

Eccola qui:

omicron, pi, rho

## Il metodo lower()

Il metodo lower() **esegue una copia di una stringa di origine, sostituisce tutte le lettere maiuscole con le loro controparti minuscole** e restituisce la stringa come risultato. Anche in questo caso, la stringa di partenza rimane inalterata.

Se la stringa non contiene caratteri maiuscoli, il metodo restituisce la stringa originale.

Nota: il metodo lower() non accetta alcun parametro.

**# Dimostrazione del metodo lower():**

```
print("SiGmA=60".lower())
```

## Il metodo lstrip()

Il metodo lstrip(), privo di parametri, **restituisce una stringa creata ex novo, formata a partire da quella originale, rimuovendo tutti gli spazi bianchi iniziali.**

```
# Dimostrazione del metodo lstrip():  
print "[" + " tau ".lstrip() + "]"
```

Le parentesi non fanno parte del risultato, ma ne indicano solo i confini.

L'esempio produce output:

```
[tau]
```

Il metodo lstrip(), **con un solo parametro**, fa la stessa cosa della sua versione senza parametri, ma **rimuove tutti i caratteri presenti nel suo argomento** (una stringa), non solo gli spazi bianchi:

```
print("www.google.com".lstrip("w."))
```

Le parentesi non sono necessarie in questo caso, poiché il risultato è il seguente:

```
cisco.com
```

## Il metodo replace()

Il metodo `replace()`, a due parametri, restituisce una copia della stringa originale in cui tutte le occorrenze del primo argomento sono state sostituite dal secondo.

Guardate il codice

**# Demonstrating the replace() method:**

```
print("www.netacad.com".replace("netacad.com", "pythoninstitute.org"))  
print("This is it!".replace("is", "are"))  
print("Apple juice".replace("juice", ""))
```

L'esempio produce output:

www.pythoninstitute.org

Thare are it!

Apple

Se il secondo argomento è una stringa vuota, la **sostituzione consiste nel rimuovere** la stringa del primo argomento. Che tipo di magia accade se il primo argomento è una stringa vuota?

La variante `replace()` a tre parametri utilizza il terzo parametro (un numero) per **limitare il numero di sostituzioni**.



## Il metodo `replace()`

Guardate l'esempio di codice modificato qui sotto:

```
print("This is it!".replace("is", "are", 1))  
print("This is it!".replace("is", "are", 2))
```

Altri metodi:

## Il metodo `rfind()`

I metodi a uno, due e tre parametri denominati `rfind()` fanno quasi le stesse cose delle loro controparti (quelle prive del prefisso *r*), ma **iniziano la loro ricerca dalla fine della stringa**, non dall'inizio (da qui il prefisso *r*, per *right*).

## Il metodo `rstrip()`

Due varianti del metodo `rstrip()` fanno quasi lo stesso effetto di `lstrip()`, ma **agiscono sul lato opposto della stringa**.

## Il metodo `split()`

Il metodo `split()` fa quello che dice: **divide la stringa e costruisce un elenco di tutte le sottostringhe rilevate**.

Il metodo **presuppone che le sottostringhe siano delimitate da spazi bianchi**: gli spazi non partecipano all'operazione e non vengono copiati nell'elenco risultante.

Se la stringa è vuota, anche l'elenco risultante è vuoto.

### **Il metodo startswith()**

Il metodo startswith() è un riflesso speculare di endswith(): **verifica se una determinata stringa inizia con la sottostringa specificata.**

### **Il metodo strip()**

Il metodo strip() combina gli effetti causati da rstrip() e lstrip(): **crea una nuova stringa priva di tutti gli spazi bianchi iniziali e finali.**

### **Il metodo swapcase()**

Il metodo swapcase() **crea una nuova stringa scambiando le lettere all'interno della stringa di partenza: i caratteri minuscoli diventano maiuscoli e viceversa.**

### **Il metodo title()**

Il metodo title() svolge una funzione in qualche modo simile: **cambia la prima lettera di ogni parola in maiuscola, trasformando tutte le altre in minuscola.**

### **Il metodo upper()**

Infine, il metodo upper() **esegue una copia della stringa di partenza, sostituisce tutte le lettere minuscole con quelle maiuscole e restituisce la stringa come risultato.**

## LAB-28

### Tempo stimato

20-25 minuti

### Livello di difficoltà

Medio

### Obiettivi

migliorare le capacità dello studente di operare con gli archi;  
utilizzando i metodi di stringa integrati in Python.

### Scenario

Sapete già come funziona `split()`. Ora vogliamo che lo dimostriate.

Il vostro compito è quello di **scrivere la vostra funzione, che si comporta quasi esattamente come il metodo `split()` originale**, cioè:

- deve accettare esattamente un argomento, una stringa;
- dovrebbe restituire un elenco di parole create dalla stringa, divise nei punti in cui la stringa contiene spazi bianchi;
- se la stringa è vuota, la funzione deve restituire un elenco vuoto;
- il suo nome dovrebbe essere `mysplit()`

Utilizzare il modello nell'editor. Testate attentamente il codice.

### Risultato previsto

## LAB-28

### Risultato previsto

```
['To', 'be', 'or', 'not', 'to', 'be,', 'that', 'is', 'the', 'question']  
['To', 'be', 'or', 'not', 'to', 'be,that', 'is', 'the', 'question']  
[]  
['abc']  
[]
```

Codice:

```
def mysplit(strng):  
    #  
    # put your code here  
    #  
  
print(mysplit("To be or not to be, that is the question"))  
print(mysplit("To be or not to be,that is the question"))  
print(mysplit(" "))  
print(mysplit(" abc "))  
print(mysplit(""))
```

## Confronto tra stringhe

Le stringhe di Python **possono essere confrontate utilizzando la stessa serie di operatori** utilizzati per i numeri.

Date un'occhiata a questi operatori: tutti possono confrontare anche le stringhe:

==

!=

>

>=

<

<=

C'è un "ma": i risultati di questi confronti possono talvolta essere un po' sorprendenti. Non dimenticate che Python non è a conoscenza (non può esserlo in alcun modo) di sottili questioni linguistiche: si limita a **confrontare i valori dei punti di codice**, carattere per carattere.

I risultati che si ottengono da un'operazione di questo tipo sono talvolta sorprendenti. Cominciamo con i casi più semplici.

Due stringhe sono uguali quando sono composte dagli stessi caratteri nello stesso ordine. Allo stesso modo, due stringhe non sono uguali quando non sono composte dagli stessi caratteri nello stesso ordine.

## Confronto tra stringhe

Entrambi i confronti danno come risultato Vero:

'alpha' == 'alpha'

'alfa' != 'Alfa'

La relazione finale tra le stringhe si determina **confrontando il primo carattere diverso in entrambe le stringhe** (tenere sempre a mente i punti di codice ASCII/UNICODE).

Quando si confrontano due corde di lunghezza diversa e quella più corta è identica all'inizio di quella più lunga, la **corda più lunga è considerata maggiore**.

Proprio come qui:

'alfa' < 'alfabeto'

La relazione è vera.

Il confronto tra stringhe è sempre sensibile alle maiuscole e alle **minuscole (le lettere maiuscole sono considerate minori di quelle minuscole)**.

L'espressione è Vero:

'beta' > 'Beta'

## Ordinamento

Il confronto è strettamente legato all'ordinamento (o meglio, l'ordinamento è in realtà un caso molto sofisticato di confronto).

Questa è una buona occasione per mostrare due modi possibili per **ordinare elenchi contenenti stringhe**. Questa operazione è molto comune nel mondo reale: ogni volta che si vede un elenco di nomi, merci, titoli o città, ci si aspetta che siano ordinati.

Supponiamo di voler ordinare il seguente elenco:

```
greco = ['omega', 'alfa', 'pi', 'gamma'].
```

In generale, Python offre due modi diversi per ordinare gli elenchi.

La prima è implementata come **funzione denominata** `sorted()`.

La funzione prende un argomento (un elenco) e **restituisce un nuovo elenco**, riempito con gli elementi dell'argomento ordinato. (Nota: questa descrizione è un po' semplificata rispetto all'implementazione reale; ne parleremo più avanti).

L'elenco originale rimane intatto.

## Ordinamento

# Dimostrazione della funzione sorted():

```
first_greek = ['omega', 'alpha', 'pi', 'gamma']
```

```
first_greek_2 = sorted(first_greek)
```

```
print(first_greek)
```

```
print(first_greek_2)
```

```
print()
```

# Dimostrazione del metodo sort():

```
second_greek = ['omega', 'alpha', 'pi', 'gamma']
```

```
print(second_greek)
```

```
second_greek.sort()
```

```
print(second_greek)
```

Lo snippet produce il seguente output:

```
['omega', 'alfa', 'pi greco', 'gamma'].
```

```
['alfa', 'gamma', 'omega', 'pi greco'].
```

Il secondo metodo agisce sull'elenco stesso: **non viene creato un nuovo elenco**. L'ordinamento viene eseguito in loco dal metodo sort().

L'output non è cambiato:

```
['omega', 'alfa', 'pi greco', 'gamma'].
```

```
['alfa', 'gamma', 'omega', 'pi greco'].
```



## Stringhe e numeri

Ci sono altre due questioni da discutere: **come convertire un numero (un intero o un float) in una stringa e viceversa**. Potrebbe essere necessario eseguire tale trasformazione. Inoltre, si tratta di un modo di routine per elaborare i dati di input/output.

La conversione numero-stringa è semplice, perché è sempre possibile. Viene effettuata da una funzione chiamata `str()`.

Proprio come qui:

```
itg = 13
```

```
flt = 1.3
```

```
si = str(itg)
```

```
sf = str(flt)
```

```
print(si + ' ' + sf)
```

Il codice viene emesso:

```
13 1.3
```

La trasformazione inversa (stringa-numero) è possibile quando e solo quando la stringa rappresenta un numero valido. Se la condizione non è soddisfatta, ci si aspetta un'eccezione `ValueError`.

## Stringhe e numeri

Utilizzate la funzione `int()` se volete ottenere un numero intero e `float()` se avete bisogno di un valore in virgola mobile

```
si = '13'
```

```
sf = '1.3'
```

```
itg = int(si)
```

```
flt = float(sf)
```

```
print(itg + flt)
```

Questo è ciò che si vedrà nella console:

```
14.3
```

## LAB-29

### Tempo stimato

30 minuti

### Livello di difficoltà

Medio

### Obiettivi

migliorare le capacità dello studente di operare con gli archi;  
utilizzando stringhe per rappresentare dati non testuali.

### Scenario

Avrete sicuramente visto un *display a sette segmenti*.

È un dispositivo (a volte elettronico, a volte meccanico) progettato per presentare una cifra decimale utilizzando un sottoinsieme di sette segmenti. Se ancora non sapete cos'è, consultate il seguente [articolo](#) di Wikipedia.

Il vostro compito è quello di scrivere **un programma in grado di simulare il funzionamento di un dispositivo a sette display**, anche se utilizzerete singoli LED al posto dei segmenti.

Ogni cifra è composta da 13 LED (alcuni illuminati, altri scuri, ovviamente): ecco come la immaginiamo:

```
# ### ### # # ### ### ### ### ### ###  
# # ##### # #####  
# ### ### ### ### ### # ### ### # #  
## # # ##### ###  
# ### ### # ### ### # ### ### ###
```

## LAB-29

Nota: il numero 8 indica l'accensione di tutti i LED.

Il codice deve *visualizzare* qualsiasi numero intero non negativo inserito dall'utente.

Suggerimento: può essere molto utile utilizzare un elenco contenente i modelli di tutte le dieci cifre.

### Dati del test

Ingresso campione:

123

Esempio di uscita:

# ### ###

# # #

# ### ###

# # #

# ### ###

Ingresso campione:

9081726354

Esempio di uscita:

### ### ### # ### ### ### ### ### # #

# # # # # # # # # # # #

### # # ### # # ### ### ### ### ###

# # # # # # # # # # # #

### ### ### # # ### ### ### ### #

## LAB-30

**Tempo stimato:** 30-90 minuti

**Livello di difficoltà:** medio

### Obiettivi

migliorare le capacità dello studente di operare con gli archi;  
conversione dei caratteri in codice ASCII e viceversa.

### Scenario

Conoscete già il cifrario di Cesare ed è per questo che vogliamo che miglioriate il codice che vi abbiamo mostrato di recente.

Il cifrario Cesare originale sposta ogni carattere di uno: *a* diventa *b*, *z* diventa *a* e così via. Rendiamo la cosa un po' più difficile e permettiamo che il valore spostato sia compreso nell'intervallo 1..25 incluso.

Inoltre, il codice deve preservare le lettere maiuscole (le lettere minuscole rimarranno minuscole) e tutti i caratteri non alfabetici devono rimanere intatti.

Il vostro compito è quello di scrivere un programma che:

chiede all'utente una riga di testo da criptare;

chiede all'utente un valore di spostamento (un numero intero compreso nell'intervallo 1..25 - nota: è necessario forzare l'utente a inserire un valore di spostamento valido (non arrendetevi e non lasciatevi ingannare da dati errati!).

stampa il testo codificato.

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

## LAB-30

### Dati del test

Ingresso campione:

abcxyzABCxyz 123

2

Esempio di uscita:

cdezabCDEzab 123

Ingresso campione:

Il dado è tratto

25

Esempio di uscita:

Sgd chd hr bzrs

## LAB-31

### Il validatore IBAN

Il quarto programma implementa (in forma leggermente semplificata) un algoritmo utilizzato dalle banche europee per specificare i numeri di conto. Lo standard denominato **IBAN** (International Bank Account Number) fornisce un metodo semplice e abbastanza affidabile per convalidare i numeri di conto contro i semplici errori di battitura che possono verificarsi durante la riscrittura del numero, ad esempio, da documenti cartacei, come fatture o bollette, a computer.

Per maggiori dettagli, consultare il sito: [https://en.wikipedia.org/wiki/International\\_Bank\\_Account\\_Number](https://en.wikipedia.org/wiki/International_Bank_Account_Number).

Un numero di conto conforme all'IBAN è composto da:

un codice paese di due lettere tratto dallo standard ISO 3166-1 (ad esempio, *FR* per la Francia, *GB* per la Gran Bretagna, *DE* per la Germania e così via)

due cifre di controllo utilizzate per eseguire i controlli di validità - test veloci e semplici, ma non del tutto affidabili, che mostrano se un numero non è valido (falsato da un errore di battitura) o sembra esserlo;

il numero di conto effettivo (fino a 30 caratteri alfanumerici - la lunghezza di questa parte dipende dal Paese)

Secondo lo standard, la convalida richiede i seguenti passaggi (secondo Wikipedia):

(fase 1) Verificare che la lunghezza totale dell'IBAN sia corretta in base al paese (questo programma non lo farà, ma se lo desiderate potete modificare il codice per soddisfare questo requisito; nota: dovete insegnare al codice tutte le lunghezze utilizzate in Europa)

(fase 2) Spostare i quattro caratteri iniziali alla fine della stringa (cioè il codice paese e le cifre di controllo).

(fase 3) Sostituire ogni lettera della stringa con due cifre, espandendo così la stringa, dove  $A = 10$ ,  $B = 11$  ...  $Z = 35$ ;

(fase 4) Interpretare la stringa come un numero intero decimale e calcolare il resto di tale numero modulo-dividendolo per 97; se il resto è 1, il test della cifra di controllo è superato e l'IBAN potrebbe essere valido.

## **LAB-32**

### **Tempo stimato**

30-60 minuti

### **Livello di difficoltà**

Facile

### **Obiettivi**

migliorare le capacità dello studente di operare con gli archi;  
convertire le stringhe in elenchi e viceversa.

### **Scenario**

Un anagramma è una nuova parola formata riordinando le lettere di una parola, utilizzando tutte le lettere originali esattamente una volta. Ad esempio, le frasi "sicurezza ferroviaria" e "favole" sono anagrammi, mentre "io sono" e "tu sei" non lo sono.

Il vostro compito è quello di scrivere un programma che:  
chiede all'utente due testi distinti;  
verifica se i testi inseriti sono anagrammi e stampa il risultato.

Nota:

assumere che due stringhe vuote non siano anagrammi;  
trattare le lettere maiuscole e minuscole come uguali;  
Gli spazi non vengono presi in considerazione durante il controllo: considerarli come inesistenti.  
Testate il vostro codice utilizzando i dati che vi abbiamo fornito.



## **LAB-32**

### **Dati del test**

Ingresso campione:

Ascoltare

Silenzioso

Esempio di uscita:

Anagrammi

Ingresso campione:

moderno

normanno

Esempio di uscita:

Non anagrammi

## LAB-33

### Tempo stimato

15-30 minuti

### Livello di difficoltà

Facile

### Obiettivi

migliorare le capacità dello studente di operare con gli archi;  
convertire i numeri interi in stringhe e viceversa.

### Scenario

Alcuni sostengono che la *Cifra della Vita* sia una cifra valutata utilizzando la data di nascita di qualcuno. È semplice: basta sommare tutte le cifre della data. Se il risultato contiene più di una cifra, bisogna ripetere l'addizione finché non si ottiene esattamente una cifra. Per esempio:

1 gennaio 2017 = 2017 01 01

$2 + 0 + 1 + 7 + 0 + 1 + 0 + 1 = 12$

$1 + 2 = 3$

3 è la cifra che abbiamo cercato e trovato.

Il vostro compito è quello di scrivere un programma che:

chiede all'utente la sua data di nascita (nel formato AAAAMMGG, o AAAAMMGG, o AAAAMMGG - in realtà, l'ordine delle cifre non ha importanza)

produce la *cifra della vita* per la data.

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

## **LAB-33**

### **Dati del test**

Ingresso campione:  
19991229

Esempio di uscita:  
6

Ingresso campione:  
20000101

Esempio di uscita:  
4

## LAB-34

### Tempo stimato

30-45 minuti

### Livello di difficoltà

Medio

### Obiettivi

migliorare le capacità dello studente di operare con gli archi;  
utilizzando il metodo `find()` per la ricerca di stringhe.

### Scenario

Facciamo un gioco. Vi daremo due stringhe: una è una parola (ad esempio, "cane") e la seconda è una combinazione di caratteri qualsiasi.

Il vostro compito è quello di scrivere un programma che risponda alla seguente domanda: **i caratteri che compongono la prima stringa sono nascosti nella seconda stringa?**

Ad esempio:

se la seconda stringa è data come "vcxzxduybfdsobywuefgas", la risposta è sì;

se la seconda stringa è "vcxzxdcybfdstbywuefsas", la risposta è no (poiché non ci sono né le lettere "d", né "o", né "g", in questo ordine)

Suggerimenti:

è necessario utilizzare le varianti a due parametri delle funzioni `pos()` all'interno del codice;  
non preoccupatevi della sensibilità delle maiuscole e delle minuscole.

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

## **LAB-34**

### **Dati del test**

Ingresso campione:

donatore

Nabucodonosor

Esempio di uscita:

Sì

Ingresso campione:

ciambella

Nabucodonosor

Esempio di uscita:

No

## LAB-35

### Tempo stimato

60-90 minuti

### Livello di difficoltà

Duro

### Obiettivi

migliorare le capacità dello studente di operare con stringhe ed elenchi;  
convertire le stringhe in elenchi.

### Scenario

Come probabilmente saprete, il *Sudoku* è un rompicapo di posizionamento dei numeri giocato su una tavola 9x9.

Il giocatore deve riempire il tabellone in un modo molto specifico:

ogni riga del tabellone deve contenere tutte le cifre da 0 a 9 (l'ordine non ha importanza)

ogni colonna della tavola deve contenere tutte le cifre da 0 a 9 (anche in questo caso, l'ordine non ha importanza)

Ciascuna delle nove "tessere" 3x3 (che chiameremo "sottoquadri") della tabella deve contenere tutte le cifre da 0 a 9.

Se avete bisogno di maggiori dettagli, potete trovarli [qui](#).

Il vostro compito è quello di scrivere un programma che:

legge 9 righe del Sudoku, ognuna delle quali contiene 9 cifre (controllare attentamente se i dati inseriti sono validi)

esce Sì se il Sudoku è valido e No altrimenti.

## LAB-35

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

<b>Dati del test</b>	<b>Ingresso campione:</b>
<b>Ingresso campione:</b>	<b>195743862</b>
<b>295743861</b>	<b>431865927</b>
<b>431865927</b>	<b>876192543</b>
<b>876192543</b>	<b>387459216</b>
<b>387459216</b>	<b>612387495</b>
<b>612387495</b>	<b>549216738</b>
<b>549216738</b>	<b>763524189</b>
<b>763524189</b>	<b>928671354</b>
<b>928671354</b>	<b>254938671</b>
<b>154938672</b>	<b>Esempio di uscita:</b>
<b>Esempio di uscita:</b>	<b>No</b>
<b>Sì</b>	