Lezione 1: basi di Python e strutture dati

![ready-rumble](assets/ready-rumble.gif)

Il nostro primissimo programma

In [1]:
```
# questo è un commento
print("Hello World!")
```

```
Hello World!
```

```
Adesso potete dire a tutti i vostri amichetti che sapete programmare in Python!
Il corso è finito, andate in pace. Congratulazioni!
```

![congratulazioni](assets/congratulations.gif)

Come creare e assegnare una variabile

In [2]:
```
# cambiamo la frase
frase = "Hello All!"
print(frase)
```

```
Hello All!
```

```
Parliamo un attimo di nomenclatura e nomi di variabili:
- non possono iniziare con un numero
- non possono contenere spazi
- non possono contenere caratteri speciali come !, $, %, ecc.
- non possono essere uguali a parole riservate del linguaggio (come if, else, ecc.).
- quando il nome è composto da più parole, si usa la notazione snake_case (cioè con gli
underscore tra le parole) e non il camelCase (cioè con la prima lettera di ogni parola
maiuscola).

Per vedere una lista completa delle parole riservate, digitare in una cella di codice:
```

In [3]:
```
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in',
'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yiel
d']
```

In [4]:
```
#pass = 59
```

![danger](assets/lm-in-danger.gif)

```
**NOTA BENE**: ci sono altre parole speciali, per esempio i nomi dei tipi
di dati built-in, come int, float, str, list, dict, set, tuple, ecc. o di funzioni di base come
print, len, ecc.  che però il compilatore non riconosce come parole riservate.
Questi sono comunque nomi
che non possono essere usati come nomi di variabili o funzioni, altrimenti
andranno a sovrascrivere il significato originale e possono causare errori.

Per esempio:
```python
```
```

```
print = 'Ciao'
print(print)
```
Adesso ho perso per tutta l'esecuzione la funzione `print` e non la recupererò finché non riavvio il kernel.
In altre parole: So cazzi!

Assegnazione multipla (la rivedremo poi)

In [5]:
```
a, b = 1, 2
print(a, b)
# scambio di variabili
a, b = b, a
print(a, b)
```

```
1 2
2 1
```

Variabili di base

Python è un linguaggi 'dynamically typed' cioè non è necessario dichiarare il tipo di variabile che si sta creando. Il tipo viene assegnato automaticamente in base al valore che viene assegnato alla variabile. Per vedere il tipo di una variabile, usiamo la funzione type().

In [6]:
```
# interi
a = 1
print(type(a))
# float
a = 1.5
print(type(a))
# booleani
a = True
print(type(a))
# stringhe, no worries le rivedremo dopo!
a = "ciao"
print(type(a))
a = 'ciao'
print(type(a))
a = """Multas per gentes et multa per aequora vectus,
advenio has miseras, frater, ad inferias,
ut te postremo donarem munere mortis
et mutam nequiquam alloquerer cinerem.
Quandoquidem fortuna mihi tete abstulit ipsum,
hey miser indigne frater adempte mihi!"""
print(type(a))
```

```
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
<class 'str'>
<class 'str'>
```

come vedete le stringhe possono essere dichiarate sia con le virgolette singole che con quelle doppie. Questo è utile se abbiamo bisogno di usare le virgolette all'interno della stringa stessa. Con le triple virgolette possiamo creare stringhe multilinea.
In Python non esiste il tipo char, ma una stringa di un solo carattere è comunque una stringa.

Ma soprattuto non ci siamo mai dovuti porre il problema di dichiarare il tipo di una variabile.

![no-problem](assets/no-problem.gif)

Castare una variabile in un altro tipo

In [7]:
```
p = 3.14
print(type(p))
p = int(p)
print(p) # sempre arrontondato per difetto
print(type(p))
```

```
<class 'float'>
3
<class 'int'>
```

Che poi questo è il vero valore di π. Lo stesso processo si può fare con tutti gli altri tipi.

Fate però attenzione, non tutte le conversioni sono possibili.
Per esempio, non è possibile convertire una stringa in un numero se la stringa non rappresenta un
numero. La conversione deve avere senso da un punto di vista logico.

In particolare sono molto importanti le conversioni a booleano, che vedremo più avanti.

Operatori

Bene, abbiamo visto un po' di tipi di dati, ma come possiamo manipolarli?
Python ci mette a disposizione una serie di operatori aritmetici e di confronto che ci permettono di fare operazioni matematiche e di confrontare i valori delle variabili.

Operatori aritmetici (int e float)

In [8]:
```
x = 2
y = 3
print(x + y)
print(x - y)
print(x * y)
print(x / y) # divisione reale
print(x // y) # divisione intera
print(x % y) # resto
print(x ** y) # potenza
```

```
5
-1
6
0.6666666666666666
0
2
8
```

Piccola nota a margine: state attenti a non usare mai l'operatore `^` per fare le potenze. Questo operatore in Python è usato per fare l'operazione di XOR bitwise, che non ci interessa. Per fare le potenze usiamo l'operatore `**`.

In [9]:
```
print(x ^ y) # xor
```

```
1
```

![da-fuq](assets/da-fuq.gif)

Per la gioia di tutti voi esistono anche le versioni "cortocircuitate" di questi operatori, che permettono di modificare il valore di una variabile in un solo passaggio.

In [10]:
```
x *= 2
print(x)
```

```
4
```

![bet-you-like-that](assets/bet-you-like-that.gif)

Operatori di confronto (int e float)

Il valore restituito da un operatore di confronto è un booleano, cioè un valore che può essere vero o falso.

In [11]:
```
print(x > y)
print(x < y)
print(x >= y)
print(x <= y)
print(x == y)
print(x != y)
y = 2
print(x is y)
```

```
True
False
True
False
False
True
False
```

L'operatore `is` è un operatore di confronto che verifica se due variabili puntano allo stesso oggetto. In altre parole controlla se due oggetti occupano la stessa posizione in memoria.
Questo torna molto utile per fare il confronto fra due oggetti.

In [12]:
```
y = 1.5
x = 1.5
print(x is y)
```

```
False
```

Python alloca gli interi sempre nello stesso spazio di memoria, quindi due interi uguali sono sempre lo stesso oggetto. Questo non succede per i float e per le stringhe.

Operatori logici (bool)

Ci sono poi gli operatori logici, che ci permettono di combinare più condizioni booleane. Gli operatori logici sono `and`, `or` e `not`.

In [13]:
```
x, y = 2, 3
print(x > 0 and y > 0)
print(x > 2 or y > 2)
print(not (x > 0)) # per sicurezza, wrappare con parentesi
print(not (x > 0 and y > 0))
```

```
True
True
False
False
```

Quando si usano questi operatori con valori non-booleani, Python è abbastanza flessibile e considera
`0` come `False`, mentre tutti gli altri numeri sono considerati `True`.

In [14]:
```
print((x>0) and x) # True viene convertito in 1
print(x>0 or 0)
```

```
2
True
```

Ultima osservazione, Python valuta le espessioni logiche da sinistra a destra
e si ferma appena può. Per esempio, se abbiamo un `and` e il primo valore è
`False`, Python non valuta il secondo valore, perché sa già che il risultato
dell'operazione sarà `False`. Questo fenomeno si chiama *short-circuiting*.

In [15]:
```
x = 2
y = 0
print(x < 0 and x/y > 0)
#print(x > 0 and x/y > 0)
```

```
False
```

![ah-shit](assets/ah-shit.gif)

Liste e tuple

Liste

Una lista è semplicemente una sequenza di valori.
In python le liste possono contenere oggetti di tipo diverso e non hanno una lunghezza predefinita.
Possiamo accedere e modificare gli elementi di una lista usando l'indice, che inizia da 0.

In [16]:
```
l = list() # lista vuota
l = [] # lista vuota
print(type(l))
l = [0, 2, 3]
l = [1, "ciao", 3.14, True]
l[0] = 1
print(l)
```

```
<class 'list'>
[1, 'ciao', 3.14, True]
```

![spongebob-list](assets/spongebob-list.gif)

Per modificare una lista possiamo usare i seguenti metodi:

In [17]:
```
l = [1, 3, 2]
# aggiungo un elemento alla fine della lista
l.append(5)
# aggiungo un elemento in posizione
l.insert(1, 2)
print(l)
# rimuovo un elemento
l.remove(2)
# rimuovo un elemento (l'ultimo di default) e lo ritorno
print(l.pop())
print(l.pop(0))

# ordino la lista
l.sort()
print(l)
# oppure sorted(l) che ritorna una nuova lista
l.reverse()
print(l)

# lista finale
print(l)
```

```
[1, 2, 3, 2, 5]
5
1
[2, 3]
[3, 2]
[3, 2]
```

Ci sono inoltre alcune utili funzioni per lavorare con le liste:

In [18]:
```
print("Lunghezza:", len(l))
print("Minimo:", min(l))
print("Massimo:", max(l))
print("Somma:", sum(l))
```

```
Lunghezza: 2
Minimo: 2
Massimo: 3
Somma: 5
```

Si può castare un oggetto non lista in una lista usando la funzione `list`

In [19]:
```
print(list(range(10)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Operatori di liste

Per le liste possiamo usare l'operatore `+` per concatenare (unire) due liste e `*` per ripetere una lista.
Possiamo anche usare l'operatore `==` per verificare l'uguaglianza.

In [20]:
```
l1 = [1, 2, 3]
l2 = [4, 5, 6]

# concatenazione
l3 = l1 + l2
print(l3)

# ripetizione
l4 = l1 * 3
print(l4)

# uguaglianza
print(l1 == [1, 2, 3])
# maggioranza
print(l1 > [1, 2, 2]) # da evitare
# identità
print(l1 is [1, 2, 3])
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
True
True
False
```

Infine possiamo verificare se un elemento (o più) è presente nella lista usando l'operatore `in`

In [21]:
```
print(1 in l1)
print(4 not in l1)
```

```
True
True
```

Copie e raddoppiamento

Attenzione, quando in python assegniamo una lista ad un'altra stiamo assegnando lo stesso spazio in memoria a entrambi i nomi.
Abbiamo due riferimenti allo stesso spazio!

In [22]:
```
l2 = l1
print(l1 is l2)
```

```
True
```

Attenzione: Se modifico `l2` modifico anche `l1`

In [23]:
```
print(l1)
l2[0] = 0
print(l1)
```

```
[1, 2, 3]
[0, 2, 3]
```

![dont-like-that](assets/dont-like-that.gif)

Serve usare la funzione `copy`, o `deepcopy` se la lista è annidata.

In [24]:
```
print(l1)
l2 = l1.copy()
l2[0] = 1
print(l1)
```

```
[0, 2, 3]
[0, 2, 3]
```

Slicing e indici

Non solo posso accedere agli elementi della lista tramite gli indici, ma usando lo slicing posso accedere anche a particolari sottoinsiemi della lista stessa.
Posso anche usare indici negativi, che mi permettono di accedere agli elementi della lista partendo dalla fine. -1 è l'ultimo elemento, -2 il penultimo, ecc.
La sintassi dello slicing è:

```python
lista[start:stop:step]
```

In [25]:
```
# primo elemento
print(l[0])
# ultimo elemento
print(l[-1])
# dal secondo elemento in poi
print(l[1:])
# dal primo elemento al terzo
print(l[:3])
# al contrario
print(l[::-1])
# ogni due elementi
print(l[::2])
```

```
3
2
[2]
[3, 2]
[2, 3]
[3]
```

Questa sintassi è uno dei punti forti di Python e permette un sacco di libertà nello scrivere codice.
Poter usare indici negativi, vi assicuro che è particolarmente comodo.

E con questo possiamo togliere le liste dalla nostra lista (per ora...)

![lists-done](assets/lists-done.gif)

Tuple

Una tupla è una sequenza di valori separati da virgole. La tupla è immutabile, cioè non posso modificare i valori che contiene. Per accedere agli elementi della tupla si usa la stessa sintassi delle liste.

In [26]:
```
t = tuple() # tupla vuota
t = () # tupla vuota
print(type(t))
t = (0, 2, 3)
#t[0] = 1 # errore
print(t)
```

```
<class 'tuple'>
(0, 2, 3)
```

Probabilmente creerete raramente delle tuple da soli, ma spesso le funzioni restituiscono tuple, quindi è importante saperle usare.
Inoltre vedremo che l'immutabilità è una caratteristica importantissima per le tuple.

![cool-beans](assets/cool-beans.gif)

Dizionari

I dizionari sono strutture dati della seguente forma:

```python
dizionario = {
    chiave1: valore1,
    chiave2: valore2,
    chiave3: valore3,
    ...
}
```

In altre parole sono delle hashmap, cioè delle strutture dati che permettono di associare ad una chiave un valore.

In [27]:
```python
d = dict() # dizionario vuoto
d = {} # dizionario vuoto

d = {
    "nome": "Mario",
    "cognome": "Rossi",
    "eta": 30
}
print(type(d))
```

```
<class 'dict'>
```

Molti dei metodi che abbiamo visto applicati alle liste si possono applicare anche ai dizionari.

In [28]:
```python
# accesso
print(d["nome"])
# aggiunta
d["sesso"] = "M"
print(d)
# modifica
d["eta"] = 31
print(d)
# rimozione (funziona anche con pop)
del d["eta"]
print(d)
# accesso sicuro
print(d.get("eta"))
# accesso sicuro con default
print(d.get("eta", 0))
```

```
Mario
{'nome': 'Mario', 'cognome': 'Rossi', 'eta': 30, 'sesso': 'M'}
{'nome': 'Mario', 'cognome': 'Rossi', 'eta': 31, 'sesso': 'M'}
{'nome': 'Mario', 'cognome': 'Rossi', 'sesso': 'M'}
None
0
```

Come per le liste possiamo usare l'operatore `in` per verificare se una chiave è presente nel dizionario e la funzione `len` per verificare la lunghezza del dizionario.
Anche l'operatore `==` funziona allo stesso modo.

Diversamente dalle liste, i dizionari non sono ordinati, quindi non possiamo usare indici per accedere agli elementi. Possiamo però usare le chiavi e possiamo manipolare direttamente le chiavi e i valori usando i metodi `keys`, `values` e `items`.

In [29]:
```
print("Chiavi:", d.keys())
print("Valori:", d.values())
print("Elementi:", d.items())
```

```
Chiavi: dict_keys(['nome', 'cognome', 'sesso'])
Valori: dict_values(['Mario', 'Rossi', 'M'])
Elementi: dict_items([('nome', 'Mario'), ('cognome', 'Rossi'), ('sesso', 'M')])
```

Come vedete gli elementi vengono restituiti sotto forma di tuple.

![dictionaries-done](assets/dictionaries-done.gif)

Tipi mutabili e immutabili

Adesso facciamo un piccolo escursus su come Python gestisce i vari tipi di dati in memoria.

![gimme-break](assets/gimme-a-break.gif)

Sebbene Python sia un linguaggio dinamicamente tipato, ogni oggetto ha un tipo e questo tipo può essere mutabile o immutabile.

Un tipo si dice mutabile se è possibile modificarlo dopo averlo creato, mentre si dice immutabile se non è possibile modificarlo.
Nonostante int e stringhe sembrino immutabili, in realtà non lo sono. Quando modifichiamo un int Python crea un nuovo oggetto e lo assegna alla variabile (cambiando il suo id).
I tipi immutabili sono:
- int
- string
- tuple
- float
- bool (sottoclasse di int)
- ecc ecc

I tipi mutabili al contrario possono essere liberamente modificati e possono avere più di un riferimento, come nell'esempio delle liste che abbiamo visto prima.
I tipi mutabili sono:
- list
- dict
- ecc ecc

Questa distinzione è molto importante per quando si danno riferimenti a nuovi oggetti, come abbiamo visto sopra, ma soprattutto un dizionario può avere come chiave solo oggetti immutabili, mentre come valore può avere oggetti di qualsiasi tipo.

Cicli e conditional statements
Ciclo for
Adesso che abbiamo introdotto le liste, possiamo parlare dei cicli for. I cicli for sono un modo per iterare su una sequenza di valori. La sintassi è la seguente:
```python
for elemento in lista:
    # fai qualcosa
```
A differenza di altri linguaggi, itero direttamente sugli elementi della sequenza, non sugli indici.
Se voglio iterare su una sequenza di numeri, posso usare la funzione range() ma di solito è più lento e scomodo.
Se proprio ci serve iterare sugli indici, consiglio di usare la funzione enumerate(), che oltre a restituire l'elemento della lista, restituisce anche l'indice corrispondente (sotto forma di tupla).

In [30]:
```
l = [1, 2, 3, 4]
print("Ranged for loop:")
for i in l:
    print(i)
print("Index for loop:")
for i in range(len(l)):
    print(l[i])
print("Enumerate for loop:")
for i, num in enumerate(l):
    print(i, num)
```

```
Ranged for loop:
1
2
3
4
Index for loop:
1
2
3
4
Enumerate for loop:
0 1
1 2
2 3
3 4
```

Conditional statements

Inoltre posso controllare le azioni che faccio in base a condizioni booleane usando gli statement if, elif e else.

In [31]:
```
x, y = 2, 3
if x > 0:
    print("x è positivo")
elif x < 0:
    print("x è negativo")
else:
    print("x è zero")
```

```
x è positivo
```

Prestate tantissima attenzione all'indentazione, perché in Python è fondamentale.
Se il codice non è indentato correttamente, Python restituisce un errore e voi ci perdete la testa.
Quando create degli if-elif-else, state sempre attenti a quali casi state considerando e a quali no.
L'else statement cattura tutti i casi non considerati nei primi due.

Ovviamente gli if-statements possono essere annidati, ma cercate di evitarlo, per varie ragioni:
- il codice diventa più difficile da leggere
- il codice diventa più difficile da debuggare

E finirete così:

![head-scratch](assets/head-scratch.gif)

Come regola di base, se avete più di 4 livelli di indentazione, avete scazzato.
Questo è particolarmente vero in Python dove l'indentazione è fondamentale.
Se volete saperne di più guardate questo [video](https://youtu.be/CFRhGnuXG-4?si=36O3fJJjUKRhgSLf).

![ok-norris](assets/ok-norris.gif)

Operatore ternario

L'operatore ternario è un modo per scrivere un if statement in una sola riga.
La sintassi è la seguente:

```python
risultato = valore1 if condizione else valore2
```

notiamo che all'interno dell'espressione non si possono fare assegnazioni, perchè dobbiamo restituire un valore.

In [32]:
```python
x = 10
y = 20
minimum = x if x < y else y
print(minimum)
```

```
10
```

L'operatore ternario si può anche estendere per considerare più di due casi, ma è sconsigliato perché diventa illeggibile.

In [33]:
```python
print(x, "è positivo") if x > 0 else print(x, "è negativo")
```

```
10 è positivo
```

Funzione zip e unpacking

Se dobbiamo iterare su più liste contemporaneamente, possiamo usare la funzione zip invece di usare direttamente gli indici.

La funzione zip prende in input un numero arbitrario di liste e restituisce una lista di tuple, dove ogni tupla contiene gli elementi delle liste in input.

In [34]:
```python
l1 = [1, 2, 3]
l2 = [4, 5, 6]
l3 = [7, 8, 9]

for i, j, k in zip(l1, l2, l3):
    print(i, j, k)
```

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

l'unpacking operator `*` ci permette di fare l'operazione inversa, cioè di prendere una lista e dividerla in elementi singoli.

In [35]:
```python
l = [1, 2, 3, 4, 5, 6, 7, 8, 9]
a = l[0::]
# primo, secondo, resto (assegnamento multiplo)
a, b, *c = l
print(a, b, c)
# tutto, copia senza copy
*c, = l
print(c, l)
# unire due liste
l = [*l1, *l2, *l3]
print(l)
```

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

Inoltre esiste anche l'unpacking operator `**` che ci permette di fare l'unpacking di un dizionario.

Ciclo while

Il ciclo while è un altro modo per iterare su una sequenza di valori. La sintassi è la seguente:

```python
while condizione:
    # fai qualcosa
```

In [36]:
```python
x = 0
while x < 10:
    print(x)
    x += 1
```

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

In [37]:
```python
import time

# ciclo while
N = 10_000_000
start = time.time()
x = 0
while x < N:
    x += 1
end = time.time()
print("Tempo ciclo while:", end - start)

start = time.time()
for x in range(N):
    pass
end = time.time()
print("Tempo ciclo for:", end - start)
```

```
Tempo ciclo while: 0.7178292274475098
Tempo ciclo for: 0.256763219833374
```

Questa differenza sostanziale di tempo di esecuzione è dovuta al fatto che il ciclo for è molto più ottimizzato del ciclo while. In particolare il ciclo for è implementato in C, mentre il ciclo while è implementato in Python puro.

Loop comprehensions

Le loop comprehensions sono un modo molto potente per creare liste, dizionari o tuple in modo molto compatto.
Sono uno strumento molto potente e molto usato in Python.
Sono di solito più veloci di altre sintassi più tradizionali come usare un ciclo for per riempire una lista.

In [38]:
```
l = [i for i in range(10)]
```

Questo tipo di sintassi (e altre che vedremo) sono tipiche di Python e sono spesso più ottimizzate e veloci di altre sintassi più tradizionali.

In [39]:
```
print("Ciclo tradizionale")
start = time.time()
l = []
for i in range(N):
    l.append(i)
end = time.time()
print(end - start)
print("List comprehension")
start = time.time()
l = [i for i in range(N)]
end = time.time()
print(end-start)
```

```
Ciclo tradizionale
0.8226494789123535
List comprehension
0.5786433219909668
```

![am-speed](assets/am-speed.gif)

Ovviamente posso usare anche degli if per filtrare gli elementi che voglio inserire nella lista.

In [40]:
```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# voglio una lista con i quadrati dei numeri pari
# solo if
l2 = [x**2 for x in l if x % 2 == 0]
print(l2)

# if else
l2 = [x**2 if x % 2 == 0 else x for x in l]
print(l2)
```

```
[4, 16, 36, 64, 100]
[1, 4, 3, 16, 5, 36, 7, 64, 9, 100]
```

Per creare una dict comprehension, la sintassi è molto simile, ma devo specificare sia la chiave che il valore e devo usare le parentesi graffe.

In [41]:
```
l = [1, 2, 3, 4, 5]
# dizionario con chiave il numero e valore il quadrato
d = {x: x**2 for x in l}
print(d)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Un tipo, anzi un topo, molto particolare: il tipo None

![stilton](assets/stilton.gif)

Rappresenta l'assenza di un valore, come il valore 'NULL' in SQL.

In [42]:
```
x = None
print(type(x))

if x is None: # identità
  print("Variabile Nulla")

if x == None: # uguaglianza
  print("Stessa cosa")

print(x == False)
print( x == "")
```

```
<class 'NoneType'>
Variabile Nulla
Stessa cosa
False
False
```

A differenza di altri linguaggi, in Python None è un tipo a sé stante, non è un valore nullo o vuoto. Per esempio, None è diverso da 0, da False, da '', da [], da {}, da (), ecc.

In [43]:
```
print("" == None)
print(False == None)
print([] == None)
```

```
False
False
False
```

Di solito il valore none, proprio per questa sua caratteristica di non convertirsi in altri tipi, viene usato come valore di default per le variabili.

Stringhe

Finalmente, siamo arrivati al tipo di dato che ci interessa di più, visto tutto il debugging che farete.

Le stringhe sono sequenze di caratteri, e si possono definire in diversi modi come abbiamo visto.

Come abbiamo visto all'inizio della lezione le stringhe possono essere dichiarate usando le virgolette singole `''`, doppie `""` o triple `"""""""` (o `'''''''`).

Attenzione, se uso le virgolette singole poi posso usare quelle doppie all'interno della stringa e viceversa.

In [44]:
```
s = "Ciao a tutti!"
type(s)
```

Out[44]:
```
str
```

Le stringhe possono essere trattate esattamente come delle tuple di caratteri, quindi posso usare gli indici e lo slicing per accedere ai caratteri.

In [45]:
```
print(s[0])
print(s[0:4])
print(s[::-1])
# non posso modificare una stringa
#s[0] = "c"
```

```
C
Ciao
!ittut a oaiC
```

Possiamo anche usare l'operatore `in` per verificare se un carattere, o una sottosequenza, è presente nella stringa e la funzione `len` per verificare la lunghezza della stringa.

In [46]:
```
print(len(s))
print('C' in s)
print("Ciao" in s)
```

```
13
True
True
```

Inoltre si possono usare alcuni degli operatori per modificare le stringhe, come `+`, `*` e `==` o combinazioni di questi.

In [47]:
```
s += "Come va?"
print(s)

t = "Dai " * 3 + "!"
print(t)
```

```
Ciao a tutti!Come va?
Dai Dai Dai !
```

Metodi per le stringhe

I metodi, sono le funzioni che possiamo usare per manipolare le stringhe.

Per essere precisi, le stringhe, come tutti gli altri tipi di dati, sono degli oggetti e i metodi sono delle funzioni interne a questi oggetti che ci permettono di manipolarli.

In [48]:
```
# elenco di metodi per stringhe
print(dir(s))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format
__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getstate__', '__gt__', '__
hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul_
_', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'cente
r', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnu
m', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprin
table', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partit
ion', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rspli
t', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'up
per', 'zfill']
```

Notiamo che come visto sopra esistono alcuni metodi speciali come `add` e `hash` che conferiscono alle stringhe le loro proprietà particolari. Nello specifico `add` permette di concatenare due stringhe e `hash` permette di calcolare l'hash di una stringa (per usarla come chiave di un dizionario, per esempio).

In [49]:
```
# maiuscolo
print(s.upper())
# la stringa è tutta maiuscola?
print(s)
# minuscolo
print(s.lower())
# titolo (prima lettera di ogni parola maiuscola)
print(s.title())
```

```
CIAO A TUTTI!COME VA?
Ciao a tutti!Come va?
ciao a tutti!come va?
Ciao A Tutti!Come Va?
```

In [50]:
```
print(s.startswith('Ciao'))
print(s.endswith('?'))
print(s.lower().startswith('ciao'))
```

```
True
True
True
```

Spesso ci serve eliminare gli spazi bianchi all'inizio e alla fine di una stringa. Per farlo possiamo usare i metodi `strip`, `lstrip` e `rstrip`.

In [51]:
```
s = "   Ciao a tutti!   "
print(s)
print(s.strip())
print(s.lstrip())
print(s.rstrip())
```

```
   Ciao a tutti!
Ciao a tutti!
Ciao a tutti!
   Ciao a tutti!
```

Il metodo `replace` permette di sostituire una sottostringa con un'altra.

In [52]:
```
s = "Ciao a tutti!"
s = s.replace("Ciao", "Hello")
print(s)
```

```
Hello a tutti!
```

Posso anche usare il metodo `split` per dividere una stringa in una lista di stringhe, usando come separatore un carattere o una sottosequenza di caratteri.

In [53]:
```
s = "Ciao a tutti!"
s = s.split("a") # default " "
print(s)
```

```
['Ci', 'o ', ' tutti!']
```

In generale usare l'operatore `+` per concatenare stringhe è sconsigliato, perché è molto lento. Invece si consiglia di usare il metodo `join` che è molto più veloce e accetta in ingresso una lista di stringhe (o un qualsiasi iterabile di stringhe).

In [54]:
```
s = "a".join(s)
print(s)
# una stringa è immutabile ma anche iterabile
print(' '.join(s))
```

```
Ciao a tutti!
C i a o   a   t u t t i !
```

Formattazione delle stringhe

La formattazione delle stringhe è un argomento molto importante, perché è un'operazione che facciamo spesso e che può essere fatta in molti modi diversi.

Inoltre è fondamentale per la sicurezza del codice, perché se non viene fatta correttamente, può portare a vulnerabilità di sicurezza.
Per esempio se prendo un input da un utente e lo concateno ad una stringa, posso creare un'apertura per un attacco di tipo SQL injection.

Infine la usiamo spesso per stampare a schermo dei messaggi di errore o di log o per controllare che il codice stia funzionando correttamente.

In [55]:
```
l = [1, 2, 3]
# usando la concatenazione
s = 'La lista è: ' + str(l)
print(s)
# formattazione con % (vecchio)
s = 'La lista è: %s' % l
print(s)
# formattazione con .format (nuovo)
s = 'La lista è: {}'.format(l)
print(s)
# formattazione con f-string (nuovissimo)
s = f'La lista è: {l}'
print(s)
```

```
La lista è: [1, 2, 3]
La lista è: [1, 2, 3]
La lista è: [1, 2, 3]
La lista è: [1, 2, 3]
```

Il primo metodo (la concatenazione diretta) è sconsigliato perché è molto lento e poco sicuro.

Il secondo metodo, cioè la formattazione vecchia usando l'operatore `%` è ancora molto usato, e offre varie opzioni di formattazione con una sintassi simile a quella di C.

Il terzo metodo, cioè la formattazione nuova usando il metodo `format` è più veloce e guadagna di leggibilità e in generale ha più flessibilità nel formattare anche oggetti diversi dalle stringhe come liste, dizionari, tuple, ecc.

Il quarto metodo ci permette di formattare le stringhe usando le f-string, che sono molto più veloci e leggibili delle altre sintassi.
Inoltre, oltre a permetterci di formattare le variabili come negli altri metodi, ci permettono di inserire direttamente delle espressioni all'interno della stringa.

In [56]:
```
# f-string
x = 2
y = 3
s = f'{x} + {y} = {x+y}'
print(s)

# if else embedded
print(f'Il numero {x} è {"pari" if x%2==0 else "dispari"}')
# list comprehension embedded
print(f'I numeri da 0 a 9 sono {[i for i in range(10)]}')
```

```
2 + 3 = 5
Il numero 2 è pari
I numeri da 0 a 9 sono [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Vediamo alcuni dei comandi per formattare numeri e stringhe.

In [57]:
```
# mostrare il nome della variabile (debugging mode)
print(f'{x=}')
# aggiungere spazi
y = 123456789
print(f"""{x=: 10}
{y=: 10}""")
# aggiungere caratteri di riempimento
print(f"""{x:=^10}
{y:=^10}""")
# arrontondamento
x = 3.141592653589793
print(f'{x=:.3f}')
```

```
x=2
x=         2
y= 123456789
====2=====
123456789=
x=3.142
```

Per una guida completa alla formattazione delle stringhe, potete consultare [questo link](https://cheatography.com/brianallan/cheat-sheets/python-f-strings-basics/).

Proviamo a fare qualcosa di utile (ma anche divertente)
Calcolare il fattoriale di un numero

Il fattoriale di un numero è il prodotto di tutti i numeri interi positivi minori o uguali a quel numero.
Per esempio il fattoriale di 5 è 5! = 5 * 4 * 3 * 2 * 1 = 120.

In [58]:
```
N = 10
print('Numero Quadrato Cubo Fattoriale')
fattoriale = 1
for i in range(1, N+1):
    fattoriale *= i
    print(f'{i:6} {i**2:7} {i**3:4} {fattoriale:10}')
```

```
Numero Quadrato Cubo Fattoriale
     1       1    1          1
     2       4    8          2
     3       9   27          6
     4      16   64         24
     5      25  125        120
     6      36  216        720
     7      49  343       5040
     8      64  512      40320
     9      81  729     362880
    10     100 1000    3628800
```

Codice Morse

L'alfabeto del codice morse è composto da una serie di punti e linee, che rappresentano le lettere dell'alfabeto.
In particolare, l'alfabeto è così composto:

![morse](assets/morse.jpg)

Possiamo trasformare una stringa in codice morse usando un dizionario che associa ad ogni lettera la sua rappresentazione in codice morse.
Per creare la stringa in codice morse, iteriamo sulla stringa e per ogni carattere prendiamo il suo valore nel dizionario.
Attenzione, dobbiamo prima convertire il carattere in maiuscolo, perché il dizionario contiene solo le lettere maiuscole.
Inoltre, dobbiamo usare la funzione `get` per evitare che il programma restituisca un errore se la lettera non è presente nel dizionario, in particolare come caso base usiamo `""`.

In [59]:
```
# alfabeto morse
morse_code = {
    'A': '.-', 'B': '-...', 'C': '-.-.', 'D': '-..', 'E': '.',
    'F': '..-.', 'G': '--.', 'H': '....', 'I': '..', 'J': '.---',
    'K': '-.-', 'L': '.-..', 'M': '--', 'N': '-.', 'O': '---',
    'X' : '-..-', 'Y' : '-.--', 'Z' : '--..'
}

# prendiamo una stringa
s = 'Ciao come va? Tutto bene? Anche a te e parenti!'

# creo la stringa in morse
s = [morse_code.get(c, "") for c in s.upper()]

# unisco la lista in una stringa (in morse non ci sono spazi)
s = ''.join(s)

print(s)
```

```
-.-....-----.-.-----..-----....-...---.-.-.......-...-.-...
```

Usando questo stesso principio (e un po' di sintassi in più) possiamo anche creare un programma che oltre a tradurre qualsiasi testo in codice morse ne generi anche il suono.

Per esempio come questo sito qui: [MorsePy](https://www.tetralark.com/MorsePy/)

Che guarda un po' è stato scritto in Python! Link al [codice](https://github.com/Elucidation/MorsePy/blob/master/morse.py).