List Arguments Summarize

Questo PDF presenta un sunto di tutti gli argomenti che saranno coperti durante le lezioni riguardanti le liste.

- **List Structure**
  - Sintassi `[1, 2, 3, 4, 5, ...]`
  - Contenuto di ogni tipo `[1, "uno", 2, [1, 2], "tre"]`
  - Differenze con le stringhe
    - Le stringhe sono *iterables* delimitati dai **doppi apici**
    - Le liste sono *iterables* delimitati dalle **parentesi quadre**
    - Le liste possono contenere ogni tipo di*object* (str, list, int), mentre le stringhe no. Da notare che se io avessi una stringa come questa `"ciao1[2, 2, 3]"`, dovrei considerare `1` o `[2, 2, 3]` come delle sottostringhe, quindi *str object* e non *list* o *int*.
    - Le stringhe e le liste differiscono anche per i**built-in methods**
- **List Properties**
  - Ordinamento non **in-structure**. Posso creare una lista non ordinata come questa `[3, 1, 2]`
  - **Indexing**. Posso accedere ad un elemento della lista tramite il suo indice
  - **Extensibility**. Posso aumentare o decrementare la sua dimensione
- **Loop on lists**
  - Scansionare una lista utilizzando un **For Loop**

```
l = [1, 2, 3, 4, 5, "uno", "due", "tre", "quattro", "cinque"]
for x in l:
    print(x)
```

```
output:
1
2
3
4
5
uno
due
tre
quattro
cinque
```

  - Scansionare una lista utilizzando un **While Loop**

```
l = [1, 2, 3, 4, 5, "uno", "due", "tre", "quattro", "cinque"]
index=0
while index <= len(l) - 1:
    print(l[index])
    index = index + 1
```

```
output:
1
2
3
4
5
uno
due
tre
quattro
cinque
```

> Considera la lista seguente come esempio di partenza `l = [1, 2, 3, 4]`

- **Lists Methods**
  - **Append method**. Appende (inserisce alla fine) un elemento dato in input.

```
l.append(5)
```

```
output: None
modified list: [1, 2, 3, 4, 5]
```

  - **Clear**. Rimuove tutti gli elementi da una lista.

```
l.clear()
```

```
output: None
modified list: []
```

  - **Copy**. Ritorna una copia della lista.

```
l.copy()
```

```
output: [1, 2, 3, 4]
modified list: None
```

  - **Extend**. Inserisce gli elementi di una lista data in input in quella originale.

```
l.extend([5, 6, 7, 8])
```

```
output: None
modified list: [1, 2, 3, 4, 5, 6, 7, 8]
```

  - **Count**. Ritorna il numero di occorrenza, nella lista, di un elemento dato in input.

```
l.count(3)
```

```
output: 1
modified list: None
```

  - **Index**. Ritorna la posizione di un dato elemento nella lista, errore se l'elemento non vi appartiene.

```
l.index(5)
l.index(24)
```

```
output 1: 4
modified list 1: None

output 2: "ValueError: 24 is not in list"
modified list 2: None
```

- **Insert**. Inserisce un dato elemento nella lista, nella posizione specificata.

```
l.insert(0, "I'm at the beginning")
l.insert(len(l)//2 + 1, "I'm at the middle")
l.insert(len(l), "I'm at the end")
```

```
output 1: None
modified list 1: ["I'm at the beginning", 1, 2, 3, 4]

output 2: None
modified list 2: ["I'm at the beginning", 1, 2, "I'm at the middle", 3, 4]

output 3: None
modified list 3: ["I'm at the beginning", 1, 2, "I'm at the middle", 3, 4, "I'm at the end"]
```

- **Pop**. Rimuove e ritorna un elemento dalla lista. Senza input ritorna l'ultimo, oppure quello nella posizione indicata.

```
l.pop()
l.pop(2)
```

```
output 1: 4
modified list: [1, 2, 3]

output 2: 3
modified list: [1, 2]
```

- **Remove**. Rimuove un dato elemento dalla lista, solo se questo vi appartiene.

```
l.remove(4)
l.remove(4)
```

```
output 1: None
modified list: [1, 2, 3]

output 2: "ValueError: list.remove(x): x is not in list"
modified list: [1, 2, 3]
```

- **Reverse**. Fa il reverse (scrive al contrario) della lista.

```
l.reverse()
```

```
output: None
modified list: [4, 3, 2, 1]
```

- **Sort**. Ordina la lista.

```
l = [4, 2, 3, 1]
l.sort()
```

```
output: None
modified list: [1, 2, 3, 4]
```

- **From the strings to the lists**

- Il metodo *Split* delle stringhe. Data una stringa in input, divide l'input per un separatore sep dato anch'esso in input e ritorna una lista con tutte le parole che erano prima o dopo il separatore.

```
stringa = "ciao come stai"
lista = stringa.split() # separatore è lo spazio
print(lista)
```

```
output: ["ciao", "come", "stai"]
```

- Formattazione di stringhe tramite il metodo*Join*. Essenzialmente, data una stringa che contiene uno o più separatori, formatta la stringa mettendo a sinistra o a destra del separatore le parole che stanno all'interno della lista data in input.

```
stringa = "ciao come stai"
lista = stringa.split("o") # ["cia", " c", "me stai"]
print("/".join(lista))

print("-".join([21, 11, 2019]))
```

```
output 1: "ciao/ c/me stai"
output 2: "21-11-2019"
```

- **List Unpacking**

  - **Assegnamento**. Possiamo creare delle variabili alle quali assegnare un singolo contenuto della lista con la seguente sintassi `var1, var2, var3, ..., varN = lista`, da notare che la lista deve essere di lunghezza pari ad N altrimenti da errore. Questo assegnamento spacchetta la lista `lista` ed associa alla variabile `var1 = lista[0]` e così via fino a `varN = lista[len(lista) - 1]`.

```
lista = ["a", "b", "c", "d", "e"] # La lunghezza è 5
a, b, c, d, e = lista             # devo avere 5 variabili alle quali assegnare il contenuto
print(a, b, c, d, e, sep="-")

stringa = "22 Novembre 2019"
data, mese, anno = stringa.split() # Posso utilizzare lo split, tanto mi ritorna una lista
print(data, mese, anno, sep="/")
```

```
output 1: a-b-c-d-e
output 2: 22/Novembre/2019
```

  - **Slicing**. Praticamente identico all'assegnamento ma posso decidere anche quante variabili voglio assegnare, e quindi quali valori voglio "spacchettare". Vi ricordo che lo slicing è quell'operazione per la quale utilizzando l'indexing posso andare a prendere un qualasiasi *sub-iterables* tramite la sequente sintassi `iter[index1:index2:index3]` dove idealmente -len(iter) <= index1 <= index2 <= len(iter). Sappiamo che ci sono casi in cui posso fare una cosa di questo genere `iter[1:-1]` dove il risultato sarà una sequenza composta dal sottoiterabile `iter[1], iter[2], ..., iter[len(iter) - 2]`.

> *Nota*: per altre spiegazioni sullo slicing consultare la lezione 1 sulle stringhe

```
lista = [1, 2, 3, 4, 5]
a, b, c, d, e = lista[:]
print(a, b, c, d, e)

a, b, c = lista[0:3]
print(a, b, c)

b, c, d = lista[1:-1] # O anche lista[1:4]
print(b, c, d)

b, d = lista[1:-1:2] # il due sta per: salta di due
print(b, d)

a, c, e = lista[0::2]
print(a, c, e)
```

```
output 1: 1 2 3 4 5
output 2: 1 2 3
output 3: 2 3 4
output 4: 2 4
output 5: 1 3 5
```

- Il carattere **Underscore** (_). Abbiamo visto che lo slicing è una forma potenziata dell'assegnamento, in qanto è possibile ottenere sotto-iterabili anche di caratteri non contigui (Ex. `iter[0::2] = iter[0], iter[2], ..., iter[-3], iter[-1]`). Però, per esempio, se noi volessimo prendere il primo, il secondo e poi gli utlimi due elementi, non potremmo farlo in quanto un singolo slicing non ci permetterebbe di eseguire una tale operazione. Un modo sarebbe unire due slicing `a, b, d, e = iter[:2] + iter[3:]`, ma è un metodo che a vederlo non mi piace tanto ... Il carattere "_" server essenzialmente, per ignorare l'assegnamento corrispettivo alla posizione in cui si trova il carattere stesso. Possiamo riscrivere la stessa cosa di prima, in modo molto più bello, utilizzando tale carattere `a, b, _, d, e = iter[:]`.

```
timestamp = "22/Novembre/2019 15:16:43"
data, ora = timestamp.split()
data_ora = data.split("/") + ora.split(":")

# Prendo il giorno mese e anno
giorno, mese, anno, _, _, _ = data_ora
print(giorno, mese, anno)

# Prendo l'ora minuti e secondi
_, _, _, ora, minuti, secondi = data_ora
print(ora, minuti, secondi)
```

```
output 1: 22 Novembre 2019
output 2: 15 16 43
```

- Il carattere **Star** (*). Può essere utilizzato sia per fare unpacking delle liste in contesti speciali, come nelle funzioni che prendono in input diversi argomenti i quali stanno in una lista precedentemente creata, oppure negli assegnamenti. Il primo caso lo vedremo più nel dettaglio quando faremo le funzioni, anche se ne daremo qualche esempio con delle funzioni built-in, come per esempio il print. Per quanto riguarda gli assegnamenti il carattere *star* viene applicato alle variabili alle quali si vuole dare un insieme di valori presenti in una lista. Se prendiamo per esempio la lista `iter = [a1, a2, ..., aN]` e facciamo il seguente assegnamento `*var1, var2, var3 = iter` otteniamo che `var1 = [iter[0], ..., iter[-3]]` in quanto `var2 = iter[-2]` mentre `var3 = iter[-1]`. Cosa molto importante è che il carattere*star* può essere applicato solo ad una variabile dell'assegnamento, altrimenti viene restituito un errore. Notiamo che tale carattere non è un "potenziamento" dei precedenti modi di fare unpacking in quanto ha una funzione diversa, almeno nel caso dell'assegnamento, in quanto andrà ad assegnare un sottoiterabile, in questo caso sotto-lista, alla variabile che sarà preceduta dal carattere *star*, a differenza dei metodi precedenti con i quali andavamo ad assegnare degli *elementi* della lista *iter*. Per quanto riguarda l'applicabilità nelle funzioni, per esempio il print, possiamo pensare che se partiamo da una lista alla quale appartenengono degli elementi che vogliamo stampare a schermo il processo da fare sarebbe, o direttamente `print(iter[0], iter[1],`

`..., iter[-1])` oppure assegnare `var1, var2, ..., varN = iter` e poi `print(var1, var2, ..., varN)`. Se noi applichiamo la seguente sintassi `func(*iter)` ottenendo `print(*iter)` quello che andiamo a fare semplicemente è quello di spacchettare la lista e di ottenere lo stesso risultato che avremmo ottenuto utilizzando i metodi precedenti.

```
lista = ["Mario", "Rossi", "25", "Roma", "mario.rossi@email.com", "oRN6s_?[6"]

# Assegnamento
*bio, email, passwd = lista
print(bio, email, passwd)

# Unpacking sulla funzione print
print(*lista)

# Unpacking sul metodo built-in delle stringhe .format
info = "Nome: {}\nCognome: {}\nEtà: {}\nCittà: {}\nEmail: {}\nPassword: {}".format(*lista)
print(info)
```

```
output 1: [Mario, Rossi, 25, Roma] mario.rossi@email.com oRN6s_?[6
output 2: Mario Rossi 25 Roma mario.rossi@email.com oRN6s_?[6
output 3:
Nome: Mario
Cognome: Rossi
Età: 25
Città: Roma
Email: mario.rossi@email.com
Password: oRN6s_?[6
```

- **List built-in function**
  - **Sorted**. Come accadeva per il metodo .sort delle liste, questa funzione built-in ordina la lista in ordine crescente (sia numerico che lessicografico). Applicando l'*help* su *Sorted* otteniamo la seguente struttura della funzione `sorted(iterable, /, *, key=None, reverse=False)`. Senza soffermarci su /, sul carattere * e sul reverse (per adesso), guardiamo all'argomento *opzionale* key. Esso è impostato di default come None, ma è possibile andargli a cambiare valore. A key dovremmo andare ad associare una funzione che ritorni un certo valore rispetto agli elementi dell'iterabile dato in input al sorted. Se per esempio volessimo ordinare gli elementi secondo la loro lunghezza (quindi se sono stringhe oppure iterabili di altro tipo) dovremmo assegnare `key=len`, che è per adesso l'unico assegnamento che possiamo fare in quanto non abbiamo ancora fatto le funzioni. Quello che posso dire, per scopo informativo, è che se per esempio abbiamo una lista di stringhe e vogliamo ordinare le parole in ordine crescente non secondo il primo carattere, ma in base al secondo dovremmo creare una funzione che presa in input una stringa ne ritorni solo il carattere in posizione 1 (secondo l'ordine degli indici).

> **Nota 1**: Se gli elementi della lista che vogliamo ordinare sono a loro volta degli iterabili, per effettuare l'ordinamento vengono confrontati gli elementi in posizione 0, mentre se tra due elementi i primi sono uguali si andranno a confrontare i secondi e così via.

> **Nota 2**: Il *sorted* non modifica in 'loco' l'iterabile, ma bensì ne ritorna uno nuovo.

> **Osservazione**: è ovvio che la lista data in input al*sorted* deve contentenere elementi dello stesso tipo altrimenti, senza passare nessuna funzione al parametro key, darà errore. La funzione che potremo passare verrà ripresa in una lezione più avanzata nel corso.

```
numeri = [5, 2, 3, 1, 8, 4]
print(sorted(numeri))

stringhe = ["ciao", "come", "ao", "ole", "sono"]
print(sorted(stringhe))

liste = [[1, 5, 3, 6], [2, 3, 7, 8]]
print(sorted(liste))

print(sorted(stringhe, key=len))
```

```
output 1: [1, 2, 3, 4, 5, 8]
output 2: ["ao", "ciao", "come", "ole", "sono"]
output 3: [[1, 5, 3, 6], [2, 3, 7, 8]]
output 4: ["ao", "ole", "ciao", "come", "sono"]
```

- **Reversed**. Essenzialmente fa il contario del *Sorted*. La grande differenza rispetto al precedente è che l'unica funzionalità è quella di ritornare l'iterabile in input al contario.

> **Nota**: Il revesed ritorna un'oggetto di tipo `<list_reverseiterator object at 0x7fad9994c128>`, dove il valore esadecimale indica la locazione di memoria in cui è situato l'oggetto. Quindi per assegnarla come lista è molto importante farci il cast.

```python
l = [1, 2, 4, "a", "b", "c", [1, 2, 3], [3, 2, 1]]
print(reversed(l))

print(l) # La lista in l non cambia

new_l = list(reversed(l)) # Fare il cast per tornare una lista
print("Lista: {}, Tipo: {}".format(str(new_l), type(new_l)))
```

```
output 1: <list_reverseiterator object at 0x....>
output 2: [1, 2, 4, "a", "b", "c", [1, 2, 3], [3, 2, 1]]
output 3: Lista: [[3, 2, 1], [1, 2, 3], 'c', 'b', 'a', 4, 2, 1], Tipo: <class 'list'>
```

- Ordinare in ordine **decrescente**. Utilizzare il sorted unito al reversed. Oppure utilizzare l'attributo reverse in sorted.

```python
lista = [9, 5, 4, 7, 10, 11, 3, 2]
lista1 = list(reversed(sorted(lista)))
print(lista1)

# Oppure
lista2 = sorted(lista, reverse=True)
print(lista2)

# Verifica
print(lista1 == lista2)
```

```
output 1: [11, 10, 9, 7, 5, 4, 3, 2]
output 2: [11, 10, 9, 7, 5, 4, 3, 2]
output 3: True
```

- **Enumerate**. Funzione built-in che prende in input una lista e ritorna una lista di coppie (x, y). Nella coppia la x corrisponde ad un valore numerico sempre crescente, mentre la y ad un valore nella lista in input. L'argomento che decide il valore iniziale della x è *start*, che di default è condierato 0. Quindi se nella lista in input ci sono N elementi, il risultato dell'enumerate sarà `[(start, iter[0]), ..., (start + N - 1, iter[N - 1])]`.

> **Nota 1**: La funzione *enumerate* è molto utile per ottenere una lista di coppie `(index, seq[index])` a partire dallo 0.

> **Nota 2**: L'enumerate ritorna un oggetto di tipo enumerate, quindi bisogna fare il cast a list come succedeva per il reversed (non per i for cicle).

```python
lista = [1, 2, 3, "a", "b", "c"]
print(list(enumerate(lista, start=5)))

print(list(enumerate(lista)))

stringa = "ciao"
print(list(enumerate(stringa))) # Funziona con altri iterabili
```

```
output 1: [(5, 1), (6, 2), (7, 3), (8, 'a'), (9, 'b'), (10, 'c')]
output 2: [(0, 1), (1, 2), (2, 3), (3, 'a'), (4, 'b'), (5, 'c')]
output 3: [(0, 'c'), (1, 'i'), (2, 'a'), (3, 'o')]
```

- **Range**. Funzione built-in la cui sintassi è la seguente `range(start, stop[, step])`. Semplicemente ritorna un lista di interi che iniziano da start (0 se start=None) e finiscono a stop (stop non incluso). E' possibile far saltare la conta di step, Ex. `range(0, N, 2) = [2x | 0 <= x <= ceil(N/2) - 1]`.

> **Osservazione**: Può essere molto veloce per creare liste di numeri crescenti, e per fare cicli for.

> **Nota**: Ritorna un range object, quindi fare il cast a list (non per i for cicle).

```python
print(list(range(10)))

print(list(range(5, 15)))

print(list(range(0, 20, 2)))

print(list(range(0, 21, 2)))

for x in range(0, 10):
    print(x)
```

```
output 1: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
output 2: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
output 3: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
output 4: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
output 5:
1
2
3
4
5
6
7
8
9
```

- **Zip**. E' molto simile alla funzione *enumerate*, con la differenza che ritorna una lista di n-ple, dipendentemente da quanti iterabili gli stiamo dando in input. Se gli diamo N iterabili allora la lista risulterà `[(iter1[0], ..., iterN[0]), ..., (iter1[min-1], ..., iterN[min-1])]` dove min è la minima lunghezza tra tutti gli iterabili che gli abbiamo dato in input.

> **Nota 1**: Possiamo considerare l'*enumerate* come un caso particolare dello *zip*, in quanto è possibile ricrearlo tramite questa funzione.

> **Nota 2**: Come le precedenti serve il cast (Non per i for cicle).

```python
iter1 = list(range(10))
iter2 = list(range(10, 20))
iter3 = list(range(20, 31)) # Il range arriva a 30 ma lo zip lo farà arrivare a 29.

for x in zip(iter1, iter2, iter3):
    print(x)

stringa = "ciao"

# Simulare l'enumerate
for x in zip(range(len(stringa)), stringa):
    print(x)

print(list(zip(range(len(stringa)), stringa)) == list(enumerate(stringa)))
```

```
output 1:
(0, 10, 20)
(1, 11, 21)
(2, 12, 22)
(3, 13, 23)
(4, 14, 24)
(5, 15, 25)
(6, 16, 26)
(7, 17, 27)
(8, 18, 28)
(9, 19, 29)
output 2:
(0, 'c')
(1, 'i')
(2, 'a')
(3, '0')
output 3: True
```

- **Map**. Funzione built-in che applica una funzione ad ogni elemento della lista passata in input. Sarebbe possibile definire delle funzioni nostre, oppure passare più liste in input, ma tutto ciò lo vedremo quando faremo le funzioni. Adesso facciamo uso di funzioni quali str, oppure int oppure anche map, enumerate o zip.

> **Nota**: Come le altre non ritorna una lista ma un map object, quindi CAST! (non per for).

```
print(map(str, [1, 2, 3, [1, 2, 3]]))

print(map(int, "123"))

print(list(map(list, list(map(enumerate, [range(0, 10)])))))) # Questo perchè altrimenti avremo [enum

stringa = "pythoncourse"
wow = list(map(list, list(map(zip, [list(range(len(stringa)))], [stringa]))))
for x in wow[0]:
    print(x)
```

```
output 1: ['1', '2','3', '[1, 2, 3]']
output 2: [1, 2, 3]
output 3: [[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8), (9, 9)]]
output 4:
(0, 'p')
(1, 'y')
(2, 't')
(3, 'h')
(4, 'o')
(5, 'n')
(6, 'c')
(7, 'o')
(8, 'u')
(9, 'r')
(10, 's')
(11, 'e')
```

> **Osservazione**: una funzione simile è la funzione*filter*, ma come funzione va una funzione che controlla una qualche condizione, cosa che noi non possiamo fare con quelle built-in che abbiamo visto. La trattazione di questa funzione sarà rimandata quando tratteremo delle funzioni vere e proprie.

- **Altre strutture con le liste**:
  - **Matrici**.