



**University
of Ferrara**

Python

Cicli, liste, dizionari, sets, arrays

Cicli

- Ciclo while

```
while condizione:  
    # istruzione
```

- Esiste anche il ciclo **for**, ma è piuttosto diverso da quanto si vede in altri linguaggi. Vediamo prima array e liste.

Liste

- Le liste vengono identificate da:

```
list_name = [e10, e11, ..., e1n]
```

- L'indicizzazione funziona come per le stringhe:

- Gli indici vanno da 0 fino alla lunghezza della lista -1

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

- Le liste sono **mutabili**

Liste

- E' possibile avere delle liste vuote : `[]`.
- Si possono usare indici a partire dalla fine della lista.
 - `list_name[-i]` ritorna l'i-esimo elemento a partire dalla fine.
- Le liste sono **eterogenee**:
 - Gli elementi in una lista possono essere di differenti tipi, possono avere interi e stringhe.
 - E' possibile anche avere liste di liste.

Liste: Funzioni

- Le liste hanno numerose funzioni e metodi.
 - **`len(list_name)`**, come per le stringhe, restituisce la lunghezza di una lista.
 - **`min(list_name)`** e **`max(list_name)`** restituisce min e max se gli operatori `>` e `<` sono definiti per gli elementi.
 - **`sum(list_name)`** restituisce la somma degli elementi se questi sono numeri.
 - **Non** è definita per le liste di stringhe.

Liste: Metodi

- **append(value)** – aggiunge il valore **value** alla fine della lista.
- **sort()** – ordina le liste sempre che per gli elementi siano definiti gli operatori **>** e **==**
- **insert(index, value)** – inserisce l'elemento **value** in corrispondenza dell'indice, **index**, specificato.
- **remove(value)** – rimuove la prima istanza dell'elemento **value**.
- **count(value)** – conta il numero di istanze dell'elemento **value** nella lista.
- **index(value)** – restituisce l'indice a partire da 0, della posizione dove per primo compare l'elemento **value** all'interno della lista. Genera un **ValueError** se tale elemento non è presente all'interno della lista.

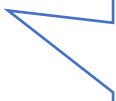
Cilci sulle Liste: ciclo for

- Vogliamo ripetere una stessa operazione su ogni singolo elemento della lista.
- Python permette di utilizzare il ciclo for come segue

```
for item in list:  
    # statements
```

- Questo è equivalente a:

```
item = list[0]  
block  
item = list [1]  
block  
...
```



Ricordate il
**for (type element :
list)** di Java?

Ciclo For e liste

. ATTENZIONE!

```
for item in list:
```

```
    # statements
```

- Qui, `item` è **un riferimento ad un oggetto**, quindi se assegniamo un valore ad `item`, esso non apparirà in `list`
- Se vogliamo modificare gli elementi della lista dobbiamo utilizzare gli **indici** della stessa.

Cicli for con indici

- Per fare ciò utilizziamo la funzione `range()`.
- `range(i)` restituisce un oggetto che rappresenta una lista ordinata di interi che vanno da 0 a i-1.
- `range(i, j)` restituisce un oggetto che rappresenta una lista ordinata di interi che vanno da i a j-1 incluso.
- `range(i, j, k)` restituisce un oggetto che rappresenta una lista ordinata di interi che vanno da i a j-1 con un passo k tra gli interi.
 - Quindi `range(i, k) == range(i, k, 1)`
- Per modificare una lista un elemento alla volta possiamo usare:

```
for i in range(len(list)):  
    # istruzione
```

Aggiunta su range()

- La funzione range restituisce un oggetto, un'**immutabile** sequenza di numeri.
- Può essere usato per inizializzare una lista, ma non direttamente: **lst = range(10)** non crea una lista e quindi non è possibile assegnare un valore a **lst**
 - **lst[2] = 2** genera un errore
- Dobbiamo usare il costruttore

```
lst = list(range(10))
```

Cicli for

```
a=list(range(4))  
for i in a:  
    i=10  
print(a) # prints [0,1,2,3]  
for i in range(4):  
    a[i]=10  
print(a) # prints [10,10,10,10]
```

Partizionamento (slicing) di una lista

- Il partizionamento funziona come per le stringhe.
- **$y = x[i:j]$** genera una lista **y** con gli elementi della lista **x** da **i** a **$j-1$** inclusi.
 - **$x[:]$** crea una lista che contiene tutti gli elementi dell'originale → **copia della lista!**
 - **$x[i:]$** crea una lista che contiene tutti gli elementi della lista **x** dall'elemento **i** fino alla fine.
 - **$x[:j]$** crea una lista che contiene tutti gli elementi della lista **x** dall'inizio fino all'elemento **$j-1$** .
- **y** è una **nuova lista**

Liste innestate

- Siccome le liste sono eterogenee, è possibile avere liste di liste.
- Questo è utile se vogliamo creare una matrice, o rappresentare una griglia o spazi multidimensionali.
- Possiamo poi referenziare la lista `list_name[i][j]` se vogliamo il *j-esimo* elemento dell'*i-esima* lista.
- Se ne deduce che se si vuole ciclare su tutti gli elementi, sarà necessario utilizzare cicli innestati:

```
for item in list:  
    for item2 in item:  
        # statements
```

Liste

Aliasing descrive una situazione in cui una stessa area di [memoria](#) può essere acceduta attraverso differenti nomi nel programma. Perciò, se i dati vengono modificati attraverso un nome implicitamente cambierà il valore anche di tutte le altre variabili che si riferiscono alla stessa cella di memoria (aliased names).

- Siccome le liste sono mutabili, le liste innestate possono causare situazioni in cui non è banale predire l'aliasing.
- Fai attenzione al partizionamento di liste che contengono elementi mutabili.
- Mentre il partizionamento crea una nuova lista, gli elementi della lista sono degli alias degli elementi della lista originale.
- Per questo motivo, anche pensando di andare a modificare solo una lista, in realtà si modificano entrambe: l'originale e la partizionata.

Liste

- **+** e ***** vengono utilizzate come per le stringhe.
 - **+** per concatenare le liste.
 - ***** usato con un intero crea altrettante copie della lista
 - Attenzione alle liste innestate ed ai problem di mutabilità!
- **x in list** è un'operazione booleana che verifica se x sia presente o meno nella lista.
 - Utile perchè non genera errori come il metodo `index()` nel caso in cui non sia presente.
- **list.pop()** rimuove l'ultimo elemento della lista e lo restituisce come parametro di ritorno.

Tuple

- Se abbiamo bisogno di **liste immutabili** possiamo affidarci alle **tuple**

`tuple_name=(item0,item1,item2,...)`

- Da notare l'utilizzo di parentesi differenti!
- Gli elementi possono essere referenziati come in una lista

`tuple_name[0], tuple_name[1],...`

- Le tuple con un solo elemento devono essere definite utilizzando la virgola per evitare ambiguità:

`(8+3)` vs. `(8+3,)`

Stringhe

- Le stringhe possono essere considerati tuple di singoli caratteri in quanto esse sono immutabili.
- Abbiamo:
 - Indicizzazione e partizionamento
 - Le stringhe non sono eterogenee, possono contenere solo caratteri.
 - Funzioni **min()** e **max()** sono definite per le stringhe, ma non **sum()**.

List comprehensions

- Un modo conciso per creare liste.
- Usualmente le applicazioni creano nuove liste in cui ogni elemento è il risultato di un'operazione su ogni membro di un'altra sequenza o un'iterazione, oppure sono sottosequenze di quegli elementi che soddisfano specifiche condizioni.
- Per esempio, assumiamo di volere una lista di quadrati:

```
>>> squares = []  
>>> for x in range(10):  
...     squares.append(x**2)  
...  
>>> squares  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehensions

- Notiamo che in questo modo stiamo creando o sovrascrivendo una variabile detta **x** che continuerà ad esistere anche una volta che il ciclo sarà concluso.
- Possiamo, invece, calcolare una lista di quadrati senza effetti collaterali, usando:

```
squares = [x**2 for x in range(10)]
```

List comprehensions

- Una list comprehension si realizza con parentesi quadrate contenenti un'espressione seguita da una clausola for, eventualmente seguita da clausole if o for.
- Il risultato sarà una nuova lista creata partendo dalla valutazione delle clausole for ed if. Ad esempio la lista che segue combina in una list comprehension gli elementi di 2 liste che non sono uguali tra loro:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]  
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

List comprehensions

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

- È equivalente a

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

List comprehensions

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # creare una nuova lista con i valori raddoppiati
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filtrare la lista per escludere i numeri negativi
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # applicare la funzione a tutti gli elementi
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # chiamare un metodo su ciascun elemento
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [fruit.strip() for fruit in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # creare una tupla con coppie di valori (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

List comprehensions

```
>>> # le tuple devono essere messe dentro alle parentesi,  
altrimenti viene generato un nuovo errore
```

```
>>> [x, x**2 for x in range(6)]
```

```
File "<stdin>", line 1, in <module>
```

```
[x, x**2 for x in range(6)]
```

```
^
```

```
SyntaxError: invalid syntax
```

```
>>> # flattening di una lista usando una listcomp con 2 'for'
```

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

```
>>> [num for elem in vec for num in elem]
```

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

List Comprehension innestate

- L'espressione iniziale di una list comprehension può essere un'espressione arbitraria, che può anche includere un'altra list comprehension.
- Consideriamo il seguente esempio di una mtrice 3x4 implementata come 3 liste di lunghezza 4:

```
>>> matrix = [  
...     [1, 2, 3, 4],  
...     [5, 6, 7, 8],  
...     [9, 10, 11, 12],  
... ]
```


List Comprehension innestate

- La seguente list comprehension traspone righe e colonne:

```
>>> [[row[i] for row in matrix] for i in range(4)]  
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

L'istruzione del

- Esiste un modo per rimuovere un elemento dalla lista piuttosto che solo il suo valore: la direttiva [del](#).
- La differenza rispetto al metodo `pop()` è che quest'ultimo ritorna un valore, `del`, invece, no.
- La direttiva `del` può anche essere usata per rimuovere parti di una lista o per ripulire una lista dal suo contenuto

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
```

```
>>> del a[0]
```

```
>>> a
```

```
[1, 66.25, 333, 333, 1234.5]
```

```
>>> del a[2:4]
```

```
>>> a
```

```
[1, 66.25, 1234.5]
```

```
>>> del a[:]
```

```
>>> a
```

```
[]
```

L'istruzione del

- del può essere utilizzata per eliminare anche variabili:

```
>>> del a
```

- A questo punto, referenziare il nome `a` sarebbe un errore (almeno fino a che non venga assegnato ad essa un nuovo valore).

Esercizio 2

1. Creare una lista di interi contenente numeri da 2 a 7
2. Ciclare la lista per sommare i valori in essa contenuti, stampare questo valore (non usare la funzione `sum()`)
3. Stampare il contenuto della lista
4. Ciclare sulla lista e modificare i suoi valori sommando ad ogni elemento il risultato della somma ottenuta precedentemente.
5. Stampare il contenuto della lista

Array

- Python non definisce gli array come un tipo integrato.
- Esistono diversi moduli che definiscono gli array.
- Di norma un array è visto come un contenitore **mutabile** di oggetti **omogenei**.
- Eccetto che per il punto precedente, gli array si comportano come liste.

Array

- Ci sono due moduli principali per definire gli array:

- `array`
- `numpy`

- Vediamo l'inizializzazione:

```
from array import array
```

```
from numpy import array
```

```
arr = array('I', range(3))
```

```
arr = array(range(3))
```

- Array creati con **`array`** consistono di elementi di uno stesso tipo dato (sono liste con elementi tra loro omogenei)
- Array creati con **`numpy`** sono ottimizzati per operazioni aritmetiche e si comportano in maniera diversa rispetto alla liste.

Array

```
from numpy import array
```

```
arr = array(range(3))
```

```
arr = arr + 10
```

Somma elemento per elemento.

```
print(arr)
```

Output: [10 11 12]

Il modulo numpy deve essere installato utilizzando il comando:

```
pip install numpy
```

oppure

```
conda install numpy
```

Array

Il modulo numpy usa dei tipi propri che possono essere automaticamente tradotti in tipi Python. Il cast avviene in maniera completamente trasparente.

```
from numpy import array
```

```
nmp = array(range(3))
```

```
# nmp contiene elementi di tipo numpy.int64
```

```
sum = 10 # somma di tipo intero
```

```
nmp = nmp + sum
```

```
print(nmp)
```

Output: [10 11 12]

Esercizio 3

1. Ripetere l'esercizio 2 ma usando gli array di numpy
2. Creare un array di interi contenente numeri da 2 a 7
3. Ciclare sull'array per sommare i valori in esso contenuti (non usare la funzione `sum()`)
4. Stampare il contenuto dell'array
5. Cambiare il valore degli elementi dell'array sommando a ciascun elemento il risultato della somma eseguita prima
6. Stampare il contenuto dell'array

Dizionari

- Si può accedere alle liste solo utilizzando gli indici
- Talvolta può essere utile accedere ad un dato indicizzandolo tramite informazioni significative piuttosto che tramite un indice
 - Ricorda le mappe in Java
- Riassumendo, i dizionari sono coppie (chiave,valore). A volte, come in Java, essi sono chiamati mappe.

`{key0 : val0, key1 : val1, ..., keyn : valn}`

- I dizionari sono di tipo **dict**. Siccome hanno un tipo possono essere assegnati ad una variabile.
- Per riferirsi ad un valore associato ad una chiave in un dizionario, usiamo **dictionary_name[key]**

Dizionari

- I dizionari sono non ordinati.
- Le chiavi dei dizionari devono essere immutabili, ma il loro valore può essere qualsiasi cosa.
- Non può, però, essere **None**.
- Una volta creato un dizionario si può aggiungere una coppia chiave-valore assegnando un valore ad una chiave.

dictionary_name[key] = value

- Le chiavi sono uniche: se nell'istruzione sopra key esisteva già, viene sostituito il valore

Metodi dei dizionari

- `len(dict_name)` come per stringhe e liste ritorna la lunghezza.
- `+` e `*` **non** sono definiti per i dizionari.
- `dict.keys()` – restituisce le chiavi secondo un qualche ordine.
- `dict.values()` – restituisce i valori secondo un qualche ordine.
- `dict.items()` – restituisce le coppie (chiave, valore) secondo un qualche ordine.

Metodi dei dizionari

- **key in dict** – restituisce **True** se e solo se il dizionario ha key al suo interno.
- **dict.get(key)** – restituisce il valore associato alla chiave key, oppure **None** se la chiave non esiste.
- **dict.clear()** – rimuove tutte le coppie chiave valore presenti all'interno del dizionario.

Metodi dei dizionari

- **`dict.copy()`** – copia l'intero dizionario.
 - Attenzione che i dizionari sono oggetti mutabili.
 - Si potrebbero presentare le stesse problematiche delle liste innestate.
- **`dict.update(dict_name)`** - aggiunge le coppie chiave-valore in **`dict_name`** a **`dict`**.
- **`dict.pop(key)`** – rimuove la coppia chiave valore indicizzata da **`key`** e la restituisce come valore di ritorno.
- Per rimuovere elementi da un dizionario è possibile utilizzare la **funzione `del`** oppure la **direttiva `del`**
`del(dict_name[key])`
`del dict_name[key]`
Attenzione: **`del dict_name`** elimina la variabile **`dict_name`**. Un successivo riferimento a questa variabile è un errore (almeno fino a quando un nuovo valore non è ad essa assegnato).

Cicli sui dizionari

```
for key in d:  
    print(key, d[key])
```

```
for val in d.values():  
    print(val)
```

```
for key, val in d.items():  
    print(key, val)
```

- Comunque, l'ordine è arbitrario.
- Come possiamo fare dei cicli ordinati?

Cicli sui dizionari

```
dict_keys = dict_name.keys()
```

```
dict_keys.sort()
```

```
for key in dict_keys:  
    print(key, dict_name[key])
```


Insiemi - Sets

- Python include un tipo di dato specifico per gli insiemi.
- Un insieme è una collezione non ordinata senza elementi ripetuti.
- Gli usi di set includono la verifica di appartenenza di un elemento e l'eliminazione dei doppi tra gli elementi in ingresso.
- Gli oggetti set supportano anche molte operazioni matematiche, come l'unione, l'intersezione, la differenza e la differenza simmetrica.
- Le parentesi graffe o la funzione [set\(\)](#) possono essere utilizzate per creare gli oggetti di tipo set.
- Nota: per creare un insieme vuoto, è necessario usare `set()`, e non `{}`; quest'ultimo creerebbe un dizionario vuoto.

Insiemi - Sets

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange',  
'banana'}  
  
>>> print(basket)                # mostra che i duplicati  
sono stati rimossi  
{'orange', 'banana', 'pear', 'apple'}  
  
>>> 'orange' in basket           # una veloce verifica di  
appartenenza  
True  
  
>>> 'crabgrass' in basket  
False
```

Insiemi - Sets

```
>>> # Dimostrazione delle operazioni sull'unicità delle lettere in due parole...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                     # lettere uniche in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                                 # lettere in a ma non in b
{'r', 'd', 'b'}
>>> a | b                                 # lettere in a o in b oppure in entrambi
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                                 # lettere sia in a che in b
{'a', 'c'}
>>> a ^ b                                 # lettere in a o in b ma non in entrambi
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Set comprehension

- Come per la [list comprehensions](#), anche la set comprehensions è supportata:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc' }
```

```
>>> a
```

```
{'r', 'd' }
```

Esercizio 4

- Talvolta vogliamo capire a quale chiave corrisponde un dato valore.
 - Questo è impossibile da fare in maniera naïve.
- Abbiamo bisogno di costruire un dizionario dove i valori sono usati come chiavi.
- Problemi?

Esercizio 4

- Talvolta vogliamo capire a quale chiave corrisponde un dato valore.
 - Questo è impossibile da fare in maniera naïve.
- Abbiamo bisogno di costruire un dizionario dove i valori sono usati come chiavi.
- Problemi?
 - Mentre le chiavi in un dizionario devono essere uniche, non abbiamo alcuna restrizione sui valori.
 - Quindi è possibile che più chiavi possano avere lo stesso valore.
- Quindi?

Esercizio 4

- I dizionari possono prendere qualsiasi tipo di valore (oggetti immutabili o mutabili – ricorda, invece, le chiavi sono immutabili)
- Possiamo allora usare una lista
- Prova creando un dizionario che contiene le seguenti coppie

<code>'Boss Nass': 'Star Wars'</code>	<code>.Dizionario invertito:</code>
<code>'Tom Bombadil': 'The Lord of the Rings'</code>	<code>'Star Wars': ['Boss Nass', 'Yoda']</code>
<code>'Hari Seldon': 'Foundation series'</code>	<code>'The Lord of the Rings': ['Tom Bombadil']</code>
<code>'Polliver': 'Game of Thrones'</code>	<code>'Foundation series': ['Hari Seldon', 'The Mule']</code>
<code>'Jules Winnfield': 'Pulp Fiction'</code>	<code>'Game of Thrones': ['Polliver']</code>
<code>'The Mule': 'Foundation series'</code>	<code>'Pulp Fiction': ['Jules Winnfield', 'Vince Vega']</code>
<code>'Flynn Rider': 'Rapunzel'</code>	<code>'Rapunzel': ['Flynn Rider']</code>
<code>'Yoda': 'Star Wars'</code>	
<code>'Vince Vega': 'Pulp Fiction'</code>	

Esercizio 4

- Stampa il dizionario creato.
- Inverti il dizionario.
- Stampa un nuovo dizionario.

Basato su slide del Prof. Riccardo Zese
riccardo.zese@unife.it