Corso ITS:

PROGETTISTA E SVILUPPATORE SOFTWARE:

*FULL STACK DEVELOPER E CLOUD SPECIALIST*

Modulo: **Programmazione in Python**

Docente: *Andrea Ribuoli*

---

# Martedì **8 Aprile** 2025

# 09:00 - 13:00

# 13:30 - 16:30

---

## programma per oggi e domani:

- **liste**: creazione, accesso, scansione e riferimento;
- operazioni: aggiungere, inserire, cercare, eliminare;
- concatenazione e replica;
- ordinamento;
- algoritmi che operano sulle liste;
- **insiemi**: creazione, aggiunta/rimozione elementi
- sottoinsiemi;
- unione, intersezione e differenza;
- **dizionari**: creazione; accesso ai valori;
- aggiunta e modifica coppie, eliminazione, scansione

---

# list

- per memorizzare raccolte di valori Python usa le **liste**
- trattandosi di contenitori verranno scandite tramite `for`
- supponiamo di leggere (con un ciclo) una serie di numeri
- vogliamo stamparli evidenziando il **valore massimo**
- non possiamo farlo prima di aver acquisito l'intero elenco
- una lista si inizializza con le **parentesi quadre** `[]`
- i valori iniziali sono separati da virgole
- accedo agli elementi con l'**indice** ( `lista[indice]` )
- per accedere tramite indice la posizione **deve già esistere**
- `esempio = []` crea un **lista vuota**
- ha senso? Sì, perchè posso **aggiungere** elementi
- `esempio.append(valore)` metodo **append()**
- il metodo *append()* accoda alla fine della lista
- col metodo **insert(posizione, valore)** si aggiunge ovunque
- gli elementi successivi vengono spostati
- non posso indicare come posizione un indice fuori dal range
- `if 30 in esempio :` è vero se uno dei valori presenti è 30
- `esempio.index(30)` ci dice la prima posizione in cui è presente

In [1]:
```
esempio = []
esempio.append(24)
esempio.append(27)
esempio.append(30)
esempio.append(22)
esempio.append(28)
esempio.append(30)
n = esempio.index(30)
n2 = esempio.index(30, n + 1)
print("Ho preso 30 negli esami", n, "e", n2)
```

```
Ho preso 30 negli esami 2 e 5
```

- se non presente la *index* genera una eccezione
- nell'incertezza devo verificare con `if 30 in esempio :`
- è possibile eliminare (estrarre) elementi con `pop(indice)`
- i successivi *"slittano"*
- se ometto l'indice, `pop()`, estraggo l'ultimo elemento
- il metodo `remove(valore)` elimina in base al valore
- il valore deve esistere

In [2]:
```
a = esempio.remove(30)
```

In [3]:
```
a = esempio.remove(30)
```

In [4]:
```
a = esempio.remove(30)
```

```
---------------------------------------------------------------------------
-------
ValueError                                Traceback (most recent cal
l last)
Input In [4], in <cell line: 1>()
----> 1 a = esempio.remove(30)

ValueError: list.remove(x): x not in list
```

- il confronto tra due liste considera l'ordine dei valori

### Prima di rimuovere i due 30:

In [ ]:
```
print(sum(esempio)/len(esempio))
```

### dopo:

In [ ]:
```
print(sum(esempio)/len(esempio))
```

- il metodo **list()** crea *"veramente"* una nuova lista

---

- l'operatore **porzione**(*slice*) si indica coi **due punti**( `:` )
- `esempio[a : b]`
- **a**: indice elemento da includere
- **b**: indice elemento da escludere
- **entrambi facoltativi**
- `esempio[ : b]` : tutti fino alla posizione `b` **esclusa**
- `esempio[a : ]` : tutti dalla posizione `a` **inclusa**

```
valori = []
print('Inserisci i voti degli esami separati da Invio')
print('Termina con Q:')
inputUtente = input("")
while inputUtente.upper() != "Q" :
    valori.append(int(inputUtente))
    inputUtente = input("")
```

- esiste un metodo per l'ordinamento: **sort()**
- in Python non esistono **tabelle**
- possiamo però creare **liste di liste**

In [6]:
```
scacchiera = []
RIGHE   = 8
COLONNE = 8

for i in range(RIGHE) :
    riga = [" "] * COLONNE
```

```
    scacchiera.append(riga)
```

In [ ]:
```
lista = [ "Mi", "piace", "molto", "programmare", "in", "Python" ]
lista2 = lista
lista3 = list(lista)
lista[5] = "Ruby"
msg = ""
for elem in lista2 :
    msg += elem + " "
print(msg)
msg = ""
for elem in lista3 :
    msg += elem + " "
print(msg)
```

# Esercizio

Un negozio di animali domestici vuole fare uno sconto ai clienti che acquistano un animale (o più) e almeno cinque altri articoli. Lo sconto è il 20% del costo degli altri articoli, mentre gli animali sono esclusi.

Scrivete la funzione `discount(prices, isPet, nItems)` che calcoli lo sconto sulla base delle informazioni ricevute sulla vendita in esame, costituita da `nItems` articoli: per l'articolo i-esimo, `prices[i]` è il prezzo prima dell'eventuale sconto e `isPet[i]` è *True* se l'articolo i-esimo è un animale.

Scrivete un programma che chieda al cassiere di digitare tutti i prezzi, ciascuno seguito da una *Y* se si tratta di un animale e da una *N* per tutti gli altri articoli, usando il prezzo *-1* come sentinella. Memorizzate in una lista i dati acquisiti, poi invocate la funzione che avete progettato e visualizzate lo sconto.

## Traccia di sviluppo

Partendo da un file di input così strutturato:

- 2.49 N
- 22.49 N
- 12.49 N
- 32.49 N
- 102.49 Y
- 2.49 N
- -1

vogliamo acquisire in memoria due liste Python:

- la prima di nome **prices** con i valori numerici flottanti [2.49, 22.49, 12.49,

32.49, 102.49, 2,49]
- la seconda di nome **isPet** con i valori booleani [False, False, False, False, True, False ]

---

- il **contenitore** di tipo *lista* è **solo uno** dei tipi di contenitore
- Python offre anche **insiemi** e **dizionari**
- potremo **combinare** più contenitori per ottenere strutture più complesse

---

# set

- un **insieme**(*set*) memorizza valori **univoci**
- ossia non possono esistere elementi duplicati
- contrariamente alle *liste*, non esiste un ordine degli elementi
- quindi, non vi si può accedere con un indice
- le **operazioni** sugli insiemi sono quella della matematica
- *maggiore velocità* rispetto alle liste (non occorre ordinamento)

In [8]:
```
CUORI  = chr(9829)
QUADRI = chr(9830)
FIORI  = chr(9827)
PICCHE = chr(9824)
semi = { CUORI, QUADRI, FIORI, PICCHE }
print(semi)
```

```
{'♦', '♠', '♣', '♥'}
```

- l'**ordine** di visita dipende dalla modalità di memorizzazione (dettagli tecnici)

In [9]:
```
for seme in sorted(semi) :
    print(seme)
```

```
♠
♣
♥
♦
```

- non si può inizializzare un **insieme vuoto** in modo simile alle liste
- è necessario scrivere: `insieme_vuoto = set()`

In [10]:
```
print(len(semi))
```

```
4
```

- operatore di appartenenza è **in**

In [11]:
```
if PICCHE in semi :
    print("Ci sono anche le PICCHE")
```

```
Ci sono anche le PICCHE
```

- si possono *aggiungere* ( `add` ) e *rimuovere* ( `discard` .vs. `remove` ) elementi
- **discard**, se non trova l'elemento, lascia l'insieme invariato ma non segnala errore
- **remove**, se non trova l'elemento, solleva una *eccezione*
- il metodo **clear** svuota l'insieme di tutti gli elementi presenti
- concetto di **sotto-insieme**
- metodo **issubset**: *True* o *False*
- insiemi uguali? `==`
- insiemi diversi? `!=`
- concetto di **unione**
- metodo **union** (l'operazione elimina eventuali duplicati)
- concetto di **intersezione**
- metodo **intersection**
- concetto di **differenza**
- metodo **difference**

---

# dict

- un **dizionario**(*dictionary*) è un contenitore che memorizza associazioni **chiave-valore** (*key-value*)
- ogni chiave è associata ad un valore (univoche)
- un singolo valore può però essere associato a più chiavi

In [12]:
```
famiglia = { "Andrea": 62,
             "Laura": 59,
             "Roberto": 34,
             "Giovanni": 29,
             "Francesca": 23 }
```

- dizionario **vuoto**: `nipoti = {}`
- modifica valore con `famiglia["Francesca"] += 1`
- eliminazione coppie con `pop(key)` (eccezione se key assente)
- scansione delle chiavi con `for key in famiglia :` (ordine in base a ottimizzazione)

- alternativamente `for key in sorted(famiglia) :` (se chiave alfanumerica, ordine lessicografico)
- scansione dei valori con `for eta in famiglia.values() :`
- in un colpo solo: `for item in contact.items() :` (**items** restituisce sequenza di tuple)

In [13]:
```
for item in famiglia.items() :
    print(item)
```

```
('Andrea', 62)
('Laura', 59)
('Roberto', 34)
('Giovanni', 29)
('Francesca', 23)
```

- coppie chiave-valore separate dai **due punti** ( `:` )
- elenco raccolto tra **parentesi graffe** ( `{}` )
- coppie separate da **virgole** ( `,` )
- come con **list()** per le liste con **dict()** si copia un dizionario
- l'operatore di indicizzazione ( `[key]` ) si usa per accedere al valore associato

In [14]:
```
print("Giovanni ha già compiuto", famiglia["Giovanni"], "anni")
```

```
Giovanni ha già compiuto 29 anni
```

- dizionario **vuoto**: `nipoti = {}`
- modifica valore con `famiglia["Francesca"] += 1`
- eliminazione coppie con `pop(key)` (eccezione se key assente)
- scansione delle chiavi con `for key in famiglia :` (ordine in base a ottimizzazione)
- alternativamente `for key in sorted(famiglia) :` (se chiave alfanumerica, ordine lessicografico)
- scansione dei valori con `for eta in famiglia.values() :`
- in un colpo solo: `for item in contact.items() :` (**items** restituisce sequenza di tuple)

In [15]:
```
for item in famiglia.items() :
    print(item)
```

```
('Andrea', 62)
('Laura', 59)
('Roberto', 34)
('Giovanni', 29)
('Francesca', 23)
```

In [16]:
```
allievi = { 1: ["Mirco", "Azzolini"],
            2: ["Wallace", "Bezerra Beretta"]}
print(allievi)
```

```
{1: ['Mirco', 'Azzolini'], 2: ['Wallace', 'Bezerra Beretta']}
```