

# APPUNTI DI INFORMATICA (PYTHON)

## 1. NUMERI E STRINGHE

I valori in Python vengono memorizzati in **VARIABILI** (= zone di memoria, dotate di nome, dove archiviare informazioni); per memorizzare una variabile, si usa l'enunciato di assegnazione [es. `cansPerPack = 6`]. Una variabile infatti viene inizializzata quando le si assegna il 1° valore; se assegno un valore ad una variabile già esistente, il vecchio valore viene sostituito con il nuovo [es. `cansPerPack = cansPerPack + 2` (o solo `+ = 2`)].

**NUMERI**: si dividono in interi (**INT**, es. `±2 - 0...`) e in virgola mobile (**FLOAT**, `0.5 - 1.0 - 1E-6` [ovvero  $1 \cdot 10^{-6}$ ])

**NOMI DELLE VARIABILI**: non si possono usare simboli (? - %...) né spazi; non si possono usare alcuni termini riservati a funzioni/metodi particolari (if – class – int...); per facilitare chi legge il codice, meglio essere più descrittivi possibile. Per le variabili si usa l'iniziale minuscola e convenzionalmente la scrittura "**CAMEL CASE**" (maiuscole separano le varie parole, es. `cansPerPack`); per le costanti, invece, si scrive tutto maiuscolo con la "**SNAKE CASE**" (es. `CAN_VOLUME`). Per i dati definiti dall'utente, invece, si tende ad usare l'iniziale maiuscola.

⚠ Quando si scrive un programma, se è destinato ad essere modificato da altri programmatore, si può aiutare a spiegare meglio i propri passaggi attraverso i **COMMENTI** (spiegazioni/suggerimenti con un `#` davanti, es. `# Do it better!`).

### ARITMETICA:

- **Operatori elementari** ( `+`, `-`, `*`, `/` ): possono essere usati con variabili numeriche per formare espressioni; un'espressione mista (ovvero con valori int e float) dà come risultato un float
- **Potenza** (`**`):  $a^2 \rightarrow a^{** 2}$
- **Divisioni**:
  - o **Normale** (`/`) [es. `7 / 4 = 1.75`]
  - o **Intera** (`//`): dà solo l'intero senza la virgola (ma non approssima, toglie solo la parte decimale) [es. `7 // 4 = 1`]
  - o **Solo il resto della divisione** (`%`) [es. `7 % 4 = 3`]
- **Funzioni matematiche predefinite** (non vanno importate per essere usate, ma sono di default):
  - o **Valore assoluto**  $\rightarrow \text{abs}(\pm 1) = 1$
  - o **Arrotondamento**  $\rightarrow \text{round}(7.625) = 8$  [con argomento solo il numero, dà il numero arrotondato all'intero];  $\text{round}(7.625, 2) = 7.63$  [il secondo argomento dice a quante cifre decimali voglio il numero arrotondato]
  - o **Massimo e Minimo**  $\rightarrow$  con n° di argomenti arbitrario, in base a quanti valori voglio confrontare [es.  $\text{max}(7.25, 5.5, 7, 10.0) = 10.0$  /  $\text{min}(7.25, 5.5, 7, 10.0) = 5.5$ ]
- **Funzioni del modulo "math"**:
  - o **Radice quadrata**  $\rightarrow \text{sqrt}(x)$  con  $x \geq 0$
  - o **Troncare la virgola**  $\rightarrow \text{trunc}(x)$
  - o **Coseno**  $\rightarrow \cos(x)$
  - o **Seno**  $\rightarrow \sin(x)$
  - o **Tangente**  $\rightarrow \tan(x)$
  - o **Esponenziale**  $\rightarrow \exp(x)$
  - o **Da radianti a gradi**  $\rightarrow \text{degrees}(x)$
  - o **Da gradi a radianti**  $\rightarrow \text{radians}(x)$
  - o **Logaritmo**  $\rightarrow \log(x)$  [ovvero  $\ln(x)$ ] oppure  $\log(x, \text{base})$  [con base scelta da noi]
  - o **Pigreco**  $\rightarrow \pi$

Posso anche unire 2 righe di testo che contengono un'espressione con un **backslash** (`\`).

STRINGHE (`str`) = sequenze di caratteri (parole o frasi), memorizzabili in delle variabili per poi riaccedervi, scritte tra virgolette (singoli apici o doppie); OPERAZIONI CON STRINGHE:

- Lunghezza stringa → `len(stringa)` conta la lunghezza della stringa, anche gli spazi vuoti; se `len(stringa)=0` allora si parla di "stringa vuota"
- Concatenare stringhe (+) → "Massimi" + "lano" → "Massimiliano"
- Ripetere stringhe (\*) → "ciao..." \* 3 → "ciao...ciao...ciao..."
- Conversioni numeri-stringhe (necessarie per far interagire numeri e stringhe):
  - o Da numero a stringa → `name = "Agent" + str(007) → "Agent007"`
  - o Da stringa a intero → `idNumber = "105" → idNumber = int(idNumber) → idNumber = 105`
  - o Da stringa a float → `price = "15.25" → price = float(price) → price = 15.25`
- Singoli caratteri di una stringa (risalire tramite la posizione che occupano, ovvero il loro indice): `name = "Mattia"`
  - o Primo carattere → `name[0] → "M"`
  - o Ultimo carattere → `name[len(name) - 1] = "a"`  
→ Per non andare "out of range", l'indice deve essere contenuto  $0 < i < (len(string) - 1)$
- Porzioni di stringhe ("slice") = `stringa[inizio : fine(esclusa) : step(di quanto salta)]`  
`stringa = "informatica" → stringa[3 : 7] → "orma" / stringa[ : 7] → "informa" / stringa[7 : ] → "tica"`  
`stringa[2 : 9 : 2] → "frai" / stringa[9 : 2 : -2] → "ctmo" / stringa[-1 : -5 : -2] → "acit"`
- Metodi per stringhe ("metodo" = istruzioni per compito specifico): `name = "Mattia Domizio"`
  - o `name.upper() → "MATTIA DOMIZIO"`
  - o `name.lower() → "mattia domizio"`
  - o `name.replace("Mattia", "Mat") → "Mat Domizio"`
  - o `name.capitalize() → trasforma in maiuscola la 1^ lettera della stringa`
  - o `name.lstrip() → elimina gli spazi vuoti prima della stringa`
  - o `name.rstrip() → elimina gli spazi vuoti dopo la stringa`
  - o `name.strip() → elimina gli spazi vuoti dopo la stringa`

→ UNICODE (noi però abbiamo il CODICE ASCII, ovvero solo i 127 caratteri dell'alfabeto latino) = schema di codifica omogeneo per codificare in tutte le lingue del mondo; i caratteri sono memorizzati come numeri interi e abbiamo 2 funzioni che ci fanno passare dal carattere al suo relativo numero intero (e viceversa):

- Da carattere a numero → `ord("H") → 72`
- Da numero a carattere → `chr(97) → "a"`

⚠ Sequenze di escape = usate per scrivere i caratteri speciali nelle stringhe, attraverso un "backslash":

- " " → `print("You are \"Welcome\"") → You are "Welcome"`
- \ → `print("Yes \\ Not") → Yes\Not`
- "a capo" → `print("ciao\nnciao\nnciao") → fa andare a capo`

DATI IN INPUT: la funzione `input()` visualizza un prompt (attende dei dati che l'utente deve immettere) e poi legge i dati immessi; esempio:

```
# Calcolare le iniziali
first = input("Enter your first name: ")
second = input("Enter your surname: ")
initials = first[0] + second[0]
print(initials)
```

La funzione `input()` ottiene solo dati sotto forma di stringhe (anche se digitiamo numeri, vengono memorizzati come stringhe); per avere dati immessi sotto forma di int o float, li converti: `int(input())` o `float(input())`.

FORMATTAZIONE OUTPUT → gli operatori di formato fanno in modo che i dati visualizzati (quindi l'output) siano ben ordinati (impaginati come lo si vuole):

- Per INT (es. int = 24):
  - o "%d" → 24
  - o "%5d" → 5 posti, di cui gli ultimi due sono occupati dal 24, mentre i rimanenti da spazi vuoti
  - o "%05d" → 5 posti, di cui gli ultimi due sono occupati dal 24, mentre i rimanenti da zeri
  - o "Numero: %5d" → visualizza Numero: 24 (ovvero la parola Numero + l'esempio sopra)
  - o "%d%%" → 24%
- Per FLOAT (es. float = 1.2197):
  - o "%f" → 1.2197
  - o "%.2f" → 1.22 (.n → approssima a n cifre decimali)
  - o "%6.2f" → 6 spazi: 2 spazi vuoti e gli ultimi 4 occupati da 1.22
- Per STR (es. str = "Hello"):
  - o "%s" → Hello
  - o "%9s" → 9 spazi: 4 vuoti e gli ultimi 5 la stringa Hello
  - o "%-9s" → 9 spazi: nei primi 5 la stringa Hello e poi gli ultimi 4 vuoti

Altro metodo di formattazione è la “**f-string**” (permette di inserire dati int/float in una stringa in output) [es. `print(f"The area is {88} m**2")` ], che è un’alternativa alla **concatenazione di stringhe** [stringa + str(numero)] e alla struttura “**.format**” [es. `print("The area is {r}".format(r = r))` ].

### Esercizio di riepilogo:

```
# Programma di un distributore automatico
# Definisco le costanti
PENNIES_PER_DOLLAR = 100
PENNIES_PER_QUARTER = 25
# Acquisiamo i dati dall'utente
moneyInsert = int(input("Enter how much dollars you insert: "))
itemPrice = int(input("Enter the price in pennies: "))
# Calcolo il resto
changeDue = PENNIES_PER_DOLLAR * moneyInsert - itemPrice
dollars = changeDue // PENNIES_PER_DOLLAR
pennies = changeDue % PENNIES_PER_DOLLAR
quarters = pennies // PENNIES_PER_QUARTER
# Visualizzo il resto dovuto
print("Dollars: %7d" % dollars)
print("Quarters: %6d" % quarters)
```

## 2. DECISIONI

**ENUNCIATO IF:** compie azioni diverse a seconda dei dati; esempio: # Programma ascensore che salta il 13° piano (al posto di 13 passa al 14)

```
floor = int(input("Enter the floor: "))
if floor > 13:
    actualFloor = floor - 1
else:
    actualFloor = floor
print("The elevator will go to floor", actualFloor)
```

Negli enunciati if abbiamo le intestazioni ("header", ovvero le condizioni) e gli enunciati ("statement block", ovvero ciò che succede in quella condizione → sebbene nell'esempio ci sia solo un enunciato semplice, ci possono essere anche più enunciati [enunciati composti]). Nel caso avessimo solo enunciati semplici (ovvero 1 solo enunciato per condizione), posso usare un "espressione condizionale" (enunciato if più compatto):

```
floor = int(input("Enter the floor: "))
actualFloor = floor - 1 if floor > 13 else floor
print("The elevator will go to floor", actualFloor)
```

I confronti negli enunciati if si fanno con gli **operatori relazionali**: **>** [ $>$ ],  **$\geq$**  [ $\geq$ ], **<** [ $<$ ],  **$\leq$**  [ $\leq$ ],  **$=$**  [=, in senso di confronto] e  **$\neq$**  [ $\neq$ ].

⚠ Ricorda che se  $r = \sqrt{2.0} \rightarrow r * r \neq 2.0$  (bisogna specificare a quanto di deve arrotondare in quanto il numero è un float); infatti possiamo definire una  $\epsilon$  molto piccola (come per gli intorni matematici) per approssimare, oppure usare la funzione **isclose(a, b, rel, abs)** del metodo math [if **isclose(r \* r, 2.0)**].

⚠ Inoltre gli operatori relazionali confrontano anche le stringhe, secondo l'**ordinamento lessicografico** (ordine alfabetico, ma con alcune aggiunte: numeri < lettere, maiuscole < minuscole e spazio vuoto < tutti i caratteri).

**Enunciati annidati** ("nested") = enunciati if dentro altri enunciati if; per visualizzarli meglio, è consigliato l'uso dei flowchart. Esempio:

```
sposato = input("Enter s for single and m for married: ")
reddito = float(input("Enter your income: "))
if sposato == "s":
    if reddito <= 32000:
        tasse = 10000
    else:
        tasse = 15000
else:
    if reddito <= 64000:
        tasse = 10000
    else:
        tasse = 15000
print("Your taxes: %.2f$" % tasse)
```

Alternative multiple (**ELIF** = if + else, ovvero delle altre alternative prima di arrivare all'else):

```
if (condizione 1):
    enunciato 1
elif (condizione 2): [Occorre verificare le condizioni (ovvero i vari elif) dalla più specifica alla più generale!]
    enunciato 2
elif (condizione 3):
    enunciato 3 ... [posso usare quanti elif voglio]
else:
    enunciato finale
```

⚠ Se voglio che una condizione non mi dia nulla, ovvero venga passata, posso usare l'enunciato **pass**; se invece voglio terminare il programma dopo una specifica condizione, posso farlo con la funzione **exit** (del modulo sys); se uso **exit("argomento")**, verrà stampato l'argomento della funzione prima di terminare il programma.

**BOOLEANI** (**bool**): una variabile booleana può assumere 2 valori, **True** e **False**. Per combinare valori booleani si usano gli **operatori booleani and** e **or**, mentre per invertire una condizione si usa il **not**.

A	B	A and B	A or B
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

## DOMIZIO MATTIA

A	not A
T	F
F	T

**Legge di De Morgan:** con l'operatore not, gli operatori and e or vengono scambiati

$$\text{not } (\text{A and B}) = \text{not A or not B}$$

$$\text{not } (\text{A or B}) = \text{not A and not B}$$

## ANALISI DI STRINGHE:

### 1. OPERAZIONI DI VERIFICA PER SOTTOSTRINGHE:

- sottostringa in stringa → True se stringa contiene sottostringa
- stringa.count(sottostringa) → n° di volte che troviamo la sottostringa in stringa
- stringa.endswith(sottostringa) → True se stringa termina con sottostringa
- stringa.startswith(sottostringa) → True se stringa inizia con sottostringa
- stringa.find(sottostringa) → dà l'indice dove troviamo la sottostringa per la 1^ volta (o -1 se non c'è)

### 2. VERIFICARE ALCUNE CARATTERISTICHE DI UNA STRINGA:

- stringa.isalnum() → True se stringa è costituita da sole lettere/cifre
- stringa.isalpha() → True se stringa è costituita da sole lettere
- stringa.isdigit() → True se stringa è costituita da sole cifre
- stringa.islower() → True se le lettere di stringa sono tutte minuscole
- stringa.isupper() → True se le lettere di stringa sono tutte maiuscole
- stringa.isspace() → True se stringa è costituita da soli spazi vuoti

## 3. CICLI

**CICLO "WHILE"** = esegue ripetutamente istruzioni finché una condizione = True. Esempio:

```
# Tempo richiesto per raddoppiare un investimento
RATE = 5.0
INITIAL_BALANCE = 10000
TARGET = 2 * INITIAL_BALANCE
balance = INITIAL_BALANCE
year = 0 # il nostro contatore del ciclo [counter]
while balance < TARGET:
    interest = balance * RATE / 100
    balance += interest # la variabile che controlla il ciclo va aggiornata ad ogni iterazione per non avere un ciclo infinito
    year += 1 # il counter ci dirà quanti giri sono passati, dunque va aggiornato ad ogni ciclo
print("The investment double after", year, "years")
```

⚠ Esistono in Python dei formati speciali per visualizzare più argomenti della funzione print:

- **sep=** → definisce il separatore da usare se si stampano diversi argomenti; print(hour, min, sec, sep=":")
- **end=** → definisce cosa stampare alla fine della riga; end="\n" va a capo; end="" attacca la print dopo

**Valore sentinella** = termina il ciclo, ma non è contenuto in esso. Esempio (valore sentinella = -1 in questo caso):

```
total = 0
salary = float(input("Enter a salary or -1 to finish: ")) # LETTURA DI PREPARAZIONE
while salary >= 0:
    total += salary
    salary = float(input("Enter a salary or -1 to finish: ")) # LETTURA DI AGGIORNAMENTO
```

⚠ Posso interrompere un ciclo anche con la funzione `exit("argomento")` vista prima oppure con l'enunciato `break` (opposto a lui l'enunciato `continue`).

CICLO "FOR" = per eseguire istruzioni su un contenitore. Esempio:

```
name = "Gaia"
for letter in name:
    print(letter) # printa G a i a (ogni lettera su una riga)
```

Il ciclo FOR può essere usato con la funzione `range()` per usare come valori una sequenza di numeri interi; gli argomenti del range sono `range(inizio, fine [esclusa], step [di quanto salta i valori])` e l'unico argomento obbligatorio è la fine (gli altri si possono eventualmente omettere):

- `for i in range(6)` → da 0 a 5
- `for i in range(3, 9)` → da 3 a 8
- `for i in range(0, 7, 2)` → 0, 2, 4, 6
- `for i in range(5, 0, -1)` → 5, 4, 3, 2, 1

Esempio di ciclo con range:

```
name = "Gaia"
for i in range(0, len(name)):
    print(name[i])
```

⚠ Per printare indice e valore insieme, si usa la funzione `enumerate()`, che restituisce, ad ogni iterazione, una coppia di valori (indice, valore):

```
for (i, lettera) in enumerate(name):
    print(i, lettera) # printa 0 G - 1 a - 2 i - 3 a (tutti su righe diverse; per averli insieme si potrà creare una lista)
```

Ciclo annidato ("nested loop") = quando il corpo di un ciclo contiene un altro ciclo, questi si dicono **annidati** (per esempio li vedremo nella visualizzazione di una tabella). Esempio:

```
NMAX = 4 # ESPONENTE
XMAX = 10 # BASE
for n in range(1, NMAX + 1):
    print("%10d" % n, end="")
    print() # Scrive gli esponenti
for n in range(1, NMAX + 1):
    print("%10s" % "x", end="")
    print("\n", " ", "-" * 35) # Mette i trattini separatori
for x in range(1, XMAX + 1):
    for n in range(1, NMAX + 1):
        print("%10.0f" % x ** n, end="")
    print("\n")
```

### ELABORARE STRINGHE TRAMITE CICLI:

- Contare le corrispondenze in una stringa (e le loro posizioni):

```
string = input("Enter a string: ")
vowels = 0
for (i, lettera) in enumerate(string):
```

```

if lettera.lower() in "aeiou":
    vowels += 1
    print(f"At position {i} vowel {lettera}")
print("N° vowels:", vowels)
-
```

**Trovare la prima corrispondenza:**

```

string = input("Enter a string: ")
found = False
i = 0
while not found and i < len(string):
    if string[i].isdigit():
        found = True
    else:
        i += 1
if found:
    print("First digit occurs at position", i)
else:
    print("No digits")
-
```

**Determinare se una stringa è valida:**

```

# Phone number
phoneNumber = input("Enter the phone number: ")
valid = len(phoneNumber) == 13
i = 0
while valid and i < len(phoneNumber):
    valid = ((i == 0 and phoneNumber[i] == "(")
              or (i == 4 and phoneNumber[i] == ")")
              or (i == 8 and phoneNumber[i] == "-"))
              or (i != 0 and i != 4 and i != 8 and phoneNumber[i].isdigit()))
    i += 1
print("Valid?", valid)
-
```

**Costruire una nuova stringa:**

```

# credit card number without "-"
userInput = input("Enter a credit card number: ")
creditCardNumber = ""
for char in userInput:
    if char != " " and char != "-":
        creditCardNumber = creditCardNumber + char
print(creditCardNumber)

```

### NUMERI CASUALI (pseudocasuali → `from random import *`):

- La funzione `random()` dà un float casuale compreso tra  $0 \leq f < 1$
- La funzione `randint(a, b)` dà un casuale intero tra a e b [es. `randint(1,6)` lancio di un dado simulato]
- La funzione `choice(string/list1)` dà un elemento casuale del suo contenitore
- La funzione `shuffle(list1)` rimescola in ordine casuale gli elementi della lista

## 4. FUNZIONI

**FUNZIONI** = sequenze di istruzioni dotate di un nome, invocate per eseguire queste istruzioni; alcune vanno invocate con argomenti, altre senza [come `random()`]. Sono delle “scatole nere”, cioè vengono usate ma senza sapere come vengono realizzate; conviene definire funzioni riutilizzabili. Esempio (definire una funzione):

```

def cubeVolume(sidelength): # Intestazione (header) -> def funzione(variabili parametro):
    volume = sidelength ** 3 # Corpo (body) della funzione
    return volume # Termina la funzione e "restituisce" il risultato

```

⚠ Una funzione va definita prima di essere usata; collaudare una funzione = definirla + usarla

⚠ Funzioni con n° variabile di argomenti → def sum(\*values): # può ricevere quanti argomenti si vuole per \*

### Esempio di programma che usa funzioni:

```
##  
# Questo programma calcola i volumi di due cubi  
#  
def main(): # La funzione main() è il programma completo, dove vengono usate tutte le funzioni che definisco  
    result1 = cubeVolume(2)  
    result2 = cubeVolume(10)  
    print("A cube with sidelength 2 has volume", result1)  
    print("A cube with sidelength 10 has volume", result2)  
## Calcola il volume di un cubo  
# @param sideLength la lunghezza di un lato del cubo  
# @return il volume del cubo  
def cubeVolume(sideLength):  
    volume = sideLength ** 3  
    return volume  
main() # Attiva la funzione main (ovvero il programma stesso)
```

⚠ Quelle in giallo sono le **intestazioni**, utili per gli altri programmatore che visualizzeranno il programma e utili se si vuole esportare il programma in formato html.

⚠ Forma compatta della definizione di una funzione:

```
def getDescription(richter):  
    if richter >= 8.0: return "Most structures fall"  
    if richter >= 6.0: return "Many structures fall"  
    return "No destruction"
```

**FUNZIONI CHE NON RESTITUISCONO UN VALORE** = non c'è un “return (valore)”. Esempio:

```
def boxString(contents): # Dà un rettangolo contenente la nostra parola, ma non un valore  
    n = len(contents)  
    if n == 0:  
        return # Termina subito senza portare un risultato  
    print("-" * (n + 2))  
    print("!" + contents + "!")  
    print("-" * (n + 2))
```

⚠ Spesso serve invocare una funzione senza averla ancora definita, per poi definirla in seguito; in questi casi si può usare lo “**stub**” (mozzicone), ovvero una parola che ci ricordiamo di dover poi tornare a definire. Esempio:

```
def cubeVolume(sideLength):  
    return "ciao"
```

### VARIABILI IN FUNZIONI:

- **Variabili locali** = definite in una funzione/blocco di codice;
- **Variabili globali** = definite al di fuori delle funzioni/blocchi di codice; Esempio:  

```
balance = 10000  
def function(amount):  
    global balance #dichiarazione “global” per usare variabili globali in funzioni  
    if balance > amount:  
        etc...
```

TOOLKIT = raccolta di funzioni o classi tra loro correlate, aventi l'obiettivo di risolvere un problema specifico; le funzioni di un toolkit possono essere inserite in un file a sé, per poi importarle in qualsiasi programma. Esempio di toolkit è "imgtools" (per la modifica delle immagini).

## 5. LISTE

LISTA = contenitore che memorizza valori. Esempio:

```
values = [1, 2, 3, 7, 10]
print(values[3]) # Printa il numero 7 (indice = 3)
```

Dunque le liste le troviamo nella forma `list = [n1, n2, n3...]`; elemento della lista = `list[i]` con  $0 \leq i < \text{len}(\text{list})$  (altrimenti dà errore di intervallo ["out of range"]). Con  $-1 \leq i \leq -\text{len}(\text{list})$ , si parte a contare gli elementi dal fondo (-1 ultimo, -2 penultimo e così via).

**SCANSIONE DI UNA LISTA:**

```
for i in range(len(list1)):
    print(i, list1[i]) # Ottengo l'indice e l'elemento relativo della lista
```

Questa scrittura è uguale a quella della funzione `enumerate()`:

```
for (i, element) in enumerate(list1):
    print(i, element)
```

Riferimenti a liste:

```
list1 = [1, 2, 3, 4, 5]
list2 = list1
list1[3] = 10
print(list2[3]) # Printa il numero 10
```

**OPERAZIONI CON LISTE:**

- Aggiungere elementi a liste → `list1.append(element)`
- Aggiungere elemento in posizione specifica → `list1.insert(i, element)` [ad ogni insert cambiano indici]
- Aggiungere tutti gli elementi di una lista ad un'altra:
  - o Concatenazione → `list1 + list2`
  - o `list1.extend(list2)`
- Cercare elemento in lista → `list1.index(element)` # Dà l'indice della 1<sup>a</sup> occorrenza dell'elemento nella lista oppure, se non c'è l'elemento nella lista, da "ValueError"
- Eliminare un elemento tramite il suo indice → `list1.pop(2)` [con solo `list1.pop()` si elimina l'ultimo]
- Eliminare un elemento → `list1.remove(element)`
- Eliminare tutti gli elementi della lista → `list1.clear()`
- Replicare elementi della lista → `list1 * n`
- Verificare se due liste sono uguali → `list1 == list2`
- Sommare tutti i valori della lista → `sum(list1)`
- Massimo e minimo di una lista → `max(list1) / min(list1)`
- Ordinare una lista:
  - o secondo l'ordinamento lessicografico crescente → `list1.sort() / list1.sort(reverse=False)`
  - o secondo l'ordinamento lessicografico decrescente → `list1.sort(reverse=True)`
  - o secondo l'ordinamento crescente, ma creando una nuova lista → `sorted(list1)`
- Copiare elementi lista in una nuova lista → `list2 = list(list1) / list1.copy()`
- Rovesciare l'ordine degli elementi → `list1.reverse()`
- Porzioni di lista (slice) → `list[from : to (escluso) : step]`; se scrivo `list1[1 : 4] = [1, 2]`, sostituisco i valori della list1 con indice da 1 a 4 (escluso) con i valori 1 e 2 (riducendo anche in questo caso la `len(list1)`).

- Scambio di elementi della lista:
  - o element = list1[i]  
list1[i] = list1[j]  
list1[j] = element
  - o tramite le **TUPLE** → (list1[i], list1[j]) = (list1[j], list1[i]) # Tupla = (n1, n2, etc...) → sono come le liste, ma con parentesi tonde e non possono essere modificate (utili nel return delle funzioni)

### DA STRINGHE A LISTE (e viceversa):

- **stringa.split(sep, maxsplit=n)**: restituisce una lista di sottostringhe ottenute suddividendo la stringa ad ogni occorrenza del separatore (sep; se sep è omesso, per default è spazio vuoto). Se maxsplit è specificato, saranno fatte al massimo n separazioni partendo da sinistra;
- **stringa.rsplit(sep, maxsplit=n)**: come split, ma suddivide la stringa partendo da destra;
- **stringa.splitlines()**: come split, ma usa come separatore il '\n', suddivide quindi la stringa in una lista contenente le singole righe di testo presenti nella stringa;
- **sep.join(list1)**: restituisce un'unica stringa contenente tutti gli elementi della list1, separati da sep.

**TABELLA** (o **matrice**) = disposizione dei valori costituita da righe e colonne; accedo agli elementi della tabella usando 2 indici (**tabella[i][j]**); gli elementi adiacenti si trovano ponendo gli indici con un +1 o un -1. Esempio di tabella:

```
for i in range(len(tabella)):
    for j in range(len(tabella[i])):
        print(tabella[i][j], end=" ")
print()
```

## 6. FILE E ECCEZIONI

### FILE:

Aprire e chiudere file:

```
infile = open("input.txt", "r") # Memorizza in variabili i file "input.txt" e "output.txt"
outfile = open("output.txt", "w") # Modalità di apertura: "r" = reading, "w" = writing
# Leggi dati da infile
# Scrivi dati in outfile
infile.close() # Chiudi i file dopo l'elaborazione
outfile.close()
```

Tutte le **modalità di apertura di un file** →

Mode	Description
r	Opens a file for reading. (default)
w	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Opens a file for exclusive creation. If the file already exists, the operation fails.
a	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Opens in text mode. (default)
b	Opens in binary mode.
+	Opens a file for updating (reading and writing)

**Leggere un file:** string = file.readline() → restituisce in una stringa la successiva riga di testo dal file (seguita da "\n" se dopo c'è un'altra riga); se è stata raggiunta la fine del file, viene restituita una stringa vuota (""); se il file contiene una riga vuota, readline restituisce "\n" (new line) [vengono restituite solo stringhe, quindi per avere dei numeri vanno convertiti]

```
line = infile.readline() # Scansione di un file riga per riga
while line != "":
    line = infile.readline()
```

**Scrivere un file:** quando un file è stato aperto in modalità di scrittura ("w"), vi si può scrivere del testo con il metodo `write("stringa (con eventualmente operatori di formato per i numeri)")`:

```
outfile.write("Hello, World!\n")
```

⚠ In alternativa, si può scrivere testo in un file con la funzione `print`:

```
print("Hello, World!", file=outfile) oppure, senza andare a capo, print("Hello, World!", end="", file=outfile)
```

⚠ Si possono anche scrivere più righe contemporaneamente con il metodo `outfile.writelines(righe)`, che ha come parametro/argomento una lista di stringhe, ovvero le righe, ciascuna delle quali deve avere già scritto il carattere new line ("\n").

⚠ Per aprire un file non nella stessa directory in cui stiamo lavorando, bisogna conoscerne il percorso file ("file path"); inoltre, per scrivere le backslash del percorso, dobbiamo inserirne 2 al posto di 1:

```
infile = open("c:\\homework\\input.txt", "r")
```

**Modificare le righe di un file:**

- `line.lstrip()` → elimina i caratteri di spaziatura (spazi vuoti, caratteri di tabulazione e new line) partendo da sinistra (se si usa `line.lstrip(caratteri)`, vengono eliminati da sinistra i caratteri)
- `line.rstrip()` → uguale a `line.lstrip()` ma da destra
- `line.strip()` → uguale ma sia da destra che da sinistra

**Suddividere le righe di un file in liste:**

- `line.split()` → prende le parole separate da spazi nella riga e le mette in una lista
- `line.split(separatore)` → uguale ma con un separatore definito (e non lo spazio vuoto)
- `line.split(separatore, maxsplit)` → maxsplit = massimo di suddivisioni eseguite, generando una lista di `len = maxsplit + 1`
- `line.rstrip(separatore, maxsplit)` → come `split()` ma partendo da destra (dalla fine)
- `line.splitlines()` → suddivisa usando come separatore della riga new line ("\n")

Questi metodi di suddivisione delle stringhe sono specialmente utili nei testi strutturati, con sequenze di dati composti (**record di dati**) contenenti ciascuna più dati diversi sulla stessa riga (es. massa-volume-superficie)

**Leggere i caratteri di una stringa:**

`stringa = file.read()` → legge l'intero contenuto del file, restituendolo come una singola stringa; se invece uso il metodo `char = file.read(n°)` → legge i successivi n° caratteri dal file e li restituisce in una stringa (dà una stringa vuota ("") se abbiamo raggiunto la fine del file).

⚠ Per leggere caratteri con `file.read()` o per leggere righe con `file.readline()`, posso anche basarmi sul numero di bytes ponendo come argomento del modulo (`size=numero`); se non pongo un argomento o pongo (`size=-1`), allora tutto il file viene letto.

⚠ Per leggere l'intero file come una lista di stringhe (una stringa per ogni riga del file), posso usare il metodo `inFile.readlines()`, equivalente al ciclo:

```
lines = []
for line in inFile:
    lines.append(line)
```

**Espressioni canoniche** (regular expression): in Python, il **modulo re** contiene una versione speciale della funzione `split` che accetta un'espressione canonica per descrivere i separatori (nella riga) e la stringa da suddividere come secondo parametro:

```
from re import split
line = "http://python.org"
```

```
regex = "[^A-Za-z]+"
tokens = split(regex, line) # ["http", "python", "org"]
```

**Carattere** = valore compreso tra 0 e 255; ci sono vari modi per codificarli, uno è l'**UTF-8** e codifica ciascun carattere come una sequenza di byte (da 1 a 4); se dobbiamo elaborare file con caratteri speciali (cinesi, caratteri accentati o simboli speciali per esempio), dobbiamo usare la codifica UTF-8:

```
infile = open("input.txt", "r", encoding="utf-8")
outfile = open("output.txt", "w", encoding="utf-8")
```

**Elaborare file in formato CSV**: la maggior parte delle applicazioni che elabora fogli di calcolo archivia i dati in un file con un proprio formato, ma per fortuna sono anche in grado di copiare i dati in un formato portabile, chiamato **CSV** (Comma-Separated Values). Esempio di programma che legge dati da un file csv che contiene informazioni su film (nome, anno, regia, produzione, attori), ignora i dati che non gli interessano e produce un nuovo file csv:

```
from csv import reader, writer # Libreria csv
infile = open("movies.csv")
csvReader = reader(infile) # Stessa funzione del open(file, "r") ma per file csv
outfile = open("filteredMovies.csv", "w") # Stessa funzione del open(file, "w") ma per file csv
csvWriter = writer(outfile)
headers = ["Name", "Year", "Actors"] # Scrivo nel file le intestazioni delle colonne
csvWriter.writerow(headers) → questo modulo aggiunge nel file una lista scritta su una riga
next(csvReader) # Nella lettura ignora la riga delle intestazioni
for row in csvReader: # Esamina le righe con i dati
    year = int(row[1])
    if 1990 <= year <= 1999:
        newRow = [row[0], row[1], row[4]] # Tiene solo i dati che a noi interessano (nome, anno, attori)
        csvWriter.writerow(newRow) # Scrive i dati tenuti come lista nel file csv su una riga
infile.close()
outfile.close()
```

**Cartelle del nostro pc** (**modulo os** [operating system], **os.path** e **shutil**):

- **os.chdir(nomeCartella)**: modifica la cartella di lavoro attuale
- **os.getcwd()**: dà il nome della cartella di lavoro attuale
- **os.listdir()**: dà una lista con i nomi delle cartelle e dei file presenti nella cartella di lavoro attuale (o cartella specificata in parentesi come argomento → **os.listdir(nomeCartella)**)
- **os.rename(sorgente, destinazione)**: modifica il nome di un file (da sorgente a destinazione)
- **os.remove(nomeFile)**: elimina il file nomeFile
- **os.path.exists(nome)**: dà un valore booleano per dire se esiste il file o la cartella di nome "nome"
- **os.path.isdir(nome)**: dà un valore booleano per dire se esiste la cartella di nome "nome"
- **os.path.isfile(nome)**: dà un valore booleano per dire se esiste il file di nome "nome"
- **os.path.join(percorso, nome)**: dà una stringa ottenuta anteponendo il percorso al nome, usando il separatore specifico per il sistema operativo in uso
- **shutil.copy(sorgente, destinazione)**: copia il file (sorgente) nella cartella (destinazione) o nel nuovo file "destinazione" se la cartella non esiste

**FILE BINARI**: i file che contengono immagini e suoni vengono solitamente archiviati in formato binario, perché i file binari risparmiano spazio; per aprire un file binario si usa la modalità "rb", mentre per aprirlo in scrittura si usa "wb" [**inFile = open(filename, "rb")** e **outFile = open(filename, "wb")**].

**ACCESSO CASUALE** (**random access**) = accedere ad un dato specifico nel file, senza dover leggere tutti i dati precedenti (opposto di accesso sequenziale). Ad ogni file viene associato un cursore che indica la posizione attuale all'interno del file (per quindi fare l'accesso casuale); per spostare il cursore, si usa il metodo **seek**:

- inFile.seek(position) → spostare il cursore ad una posizione misurata a partire dall'inizio del file
- inFile.seek(4, SEEK\_CUR) → spostare il cursore avanti di 4 rispetto alla sua posizione attuale [la costante SEEK\_CUR è definita nel modulo io]
- inFile.seek(-3, SEEK\_CUR) → spostare il cursore indietro di 3

Per conoscere la posizione attuale del cursore (sempre misurata rispetto l'inizio del file), si usa inFile.tell()

⚠ Il formato BMP per file immagine, rispetto ai formati più conosciuti, è più semplice ma anche più grande in quanto i file non sono soggetti a compressione dei dati. In questo formato, ogni pixel viene rappresentato da una sequenza di tre byte (blu, verde, rosso).

```
BaseException
  +-+ SystemExit
  +-+ KeyboardInterrupt
  +-+ GeneratorExit
  +-+ Exception
    +-+ StopIteration
    +-+ StopAsyncIteration
    +-+ ArithmeticError
      +-+ FloatingPointError
      +-+ OverflowError
      +-+ ZeroDivisionError
    +-+ AssertionError
    +-+ AttributeError
    +-+ BufferError
    +-+ EOFError
    +-+ ImportError
      +-+ ModuleNotFoundError
    +-+ LookupError
      +-+ IndexError
      +-+ KeyError
    +-+ MemoryError
    +-+ NameError
      +-+ UnboundLocalError
  +-+ OSError
    +-+ BlockingIOError
    +-+ ChildProcessError
    +-+ ConnectionError
      +-+ BrokenPipeError
      +-+ ConnectionAbortedError
      +-+ ConnectionRefusedError
      +-+ ConnectionResetError
    +-+ FileExistsError
    +-+ FileNotFoundError
    +-+ InterruptedError
    +-+ IsADirectoryError
    +-+ NotADirectoryError
    +-+ PermissionError
    +-+ ProcessLookupError
    +-+ TimeoutError
  +-+ ReferenceError
  +-+ RuntimeError
    +-+ NotImplementedError
    +-+ RecursionError
  +-+ SyntaxError
    +-+ IndentationError
      +-+ TabError
  +-+ SystemError
  +-+ TypeError
  +-+ ValueError
    +-+ UnicodeError
      +-+ UnicodeDecodeError
      +-+ UnicodeEncodeError
      +-+ UnicodeTranslateError
  +-+ Warning
    +-+ DeprecationWarning
    +-+ PendingDeprecationWarning
    +-+ RuntimeWarning
    +-+ SyntaxWarning
    +-+ UserWarning
    +-+ FutureWarning
    +-+ ImportWarning
    +-+ UnicodeWarning
    +-+ BytesWarning
    +-+ ResourceWarning
```

**ECCEZIONI:** in Python, la gestione delle eccezioni è un meccanismo per trasferire il controllo dell'esecuzione dal punto in cui è stato rilevato l'errore al punto in cui l'errore viene gestito; infatti quando si rileva una condizione d'errore basta solo sollevare un'eccezione appropriata (a sinistra, GERARCHIA DELLE ECCEZIONI).

Sollevare un'eccezione [`raise eccezione(desrizione dell'eccezione)`]:

`if amount > balance:`

`raise ValueError("Amount exceeds balance")`

`balance = balance - amount`

→ Quando viene sollevata l'eccezione sopra descritta, l'ultima riga non viene eseguita!

Se un'eccezione non ha un GESTORE appropriato, viene visualizzato un messaggio d'errore durante l'esecuzione e il programma termina; gli enunciati che possono provocare delle eccezioni vanno inseriti all'interno di un **blocco TRY**, con il relativo gestore in un **blocco EXCEPT** successivo:

```
try:
    infile = open("input.txt", "r") # Può sollevare un IOError
    line = infile.readline()
    process(line)
except IOError: # Gestione possibile IOError
    print("Could not open input file!")
except Exception as exceptObj: # exceptObj=possibile altra eccezione
    print("Error:", str(exceptObj))
```

→ Quando viene sollevata un'eccezione di tipo IOError, l'esecuzione del programma riprende dal 1° enunciato except

Le eccezioni vanno elencate dalla più specifica alla più generica.

**Clausola FINALLY** = codice eseguito, anche se nel blocco try è stata sollevata un'eccezione:

```
outfile = open(filename, "w")
try:
    writeData(outfile)
finally:
    outfile.close()
```

⚠ Conviene sempre gestire un'eccezione soltanto dove il problema può essere risolto (non subito!).

⚠ Non conviene mai usare `except` e `finally` in uno stesso enunciato `try`; conviene invece usare 2 enunciati:

- `try/finally` per liberare le risorse attive (es. chiudere i file aperti)
- `try/except` per gestire gli errori

```
try:
    outfile = open(filename, "w")
    try:
        writeData(outfile)
    finally:
        outfile.close()
except IOError:
    ...

```

**Enunciato with** (= `try/finally`): apre il file, assegna alla variabile `outfile` l'oggetto che lo rappresenta e lo chiude quando viene raggiunta la fine dell'enunciato oppure quando viene sollevata un'eccezione:

```
with open(filename, "w") as outfile:
    writeData(outfile)
```

⚠ Quando si scrive un programma, bisogna pensare bene a quali tipi di eccezioni si possano verificare; per ogni eccezione, bisogna decidere quale parte del programma sia in grado di gestirla in modo completo.

**Misure statistiche di base** (`modulo statistics` della libreria standard):

- Media → `mean = mean(dati)`
- Mediana → `median = median(dati)`
- Deviazione standard → `stdev = stdev(dati)`

## 7. INSIEMI E DIZIONARI

**INSIEME (SET)** = raccolta di valori univoci (senza duplicati); diversamente da una lista, gli elementi vengono memorizzati senza un ordine specifico (non si può accedere agli elementi usando un indice); le operazioni che si possono fare sugli insiemi sono le stesse degli insiemi matematici.

**FUNZIONI / OPERAZIONI INSIEMI** (indichiamo insieme con "s" [set]):

- `s = set()` → crea insieme vuoto (non si può creare con due graffe (come liste) perché creeremmo un dizionario)
- `s = set(sequenza)` → crea un insieme che contiene una copia della sequenza / lista
- `s = {e1, e2, e3, e4, ..., eN}` → crea un insieme che contiene gli elementi indicati (no liste, file o dizionari)
- `len(s)` → restituisce il numero di elementi dell'insieme
- `e1 in s / e1 not in s` → determina se l'elemento e1 appartiene o non appartiene all'insieme
- `s.add(e1)` → aggiunge un nuovo elemento e1 all'insieme (se è già presente, non succede nulla)
- `s.discard(e1)` → elimina e1 dall'insieme (se non c'è e1, non succede nulla)
- `s.remove(e1)` → elimina e1 dall'insieme (se non c'è e1, solleva un'eccezione / errore)
- `s.clear()` → elimina tutti gli elementi dall'insieme (svuota l'insieme)
- `s == t / s != t` → determina se l'insieme s è uguale / diverso dall'insieme t
- `s.issubset(t)` → determina se s è un sottoinsieme di t [o anche  $s \leq t / s < t$ ]
- `s.issuperset(t)` → determina se s contiene il sottoinsieme t [o anche  $s \geq t / s > t$ ]
- `s.union(t)` → restituisce l'unione di s e t, ovvero un nuovo insieme che contiene gli elementi di s e t, ma senza duplicati [o anche  $s | t$ ]
- `s.intersection(t)` → restituisce l'intersezione di s e t, ovvero un nuovo insieme con gli elementi sia di s sia di t [o anche  $s \& t$ ]
- `s.isdisjoint(t)` → True se hanno intersezione vuota
- `s.difference(t)` → restituisce un nuovo insieme che contiene gli elementi di s, tranne quelli in comune con t [o anche  $s - t$ ]
- `max(s) / min(s)` → dà massimo / minimo dell'insieme
- `list1 = sorted(s)` → dà una lista con elementi dell'insieme ordinati
- `sum(s)` → somma tutti gli elementi dell'insieme
- `s1 = s.copy()` → dà una copia dell'insieme

⚠ Quando dobbiamo scrivere una raccolta di dati non duplicati, l'insiemi sono molto utili (meglio di liste)!

⚠ Per verificare se e1 appartiene all'insieme, si calcola il codice `hash(e1)` e si confronta e1 con tutti gli elementi dell'insieme con lo stesso codice.

**DIZIONARIO (DICTIONARY or map)** = memorizza associazioni tra chiavi (**KEYS**, univoche) e valori (**VALUES**); esempio di dizionario:

```
colors = {"Red", "Blue", "Green"} #Insieme
```

```
favoriteColors = {"Romeo": "Green", "Adam": "Red"} #Dizionario → dict = {key1: value1, key2: value2 ... , keyn: valuen}
```

**FUNZIONI / OPERAZIONI DIZIONARI** (indichiamo insieme con "d" [dictionary]):

- `d = dict() / d = {}` → crea un dizionario vuoto
- `d = {k1:v1, k2:v2, ..., kN:vN}` → crea un dizionario con gli elementi specificati
- `d2 = dict(d1)` → crea un dizionario copia del dizionario d1
- `len(d)` → dà il numero di coppie key:value del dizionario
- `key in d / key not in d` → determina se la chiave appartiene o non appartiene al dizionario
- `d[key] = value`
  - o se la chiave non è presente nel dizionario, aggiunge una nuova coppia key:value al dizionario
  - o se la chiave è già presente, ne modifica il valore associato

- `x = d[key]` → restituisce il valore associato alla chiave; se key non appartiene al dizionario, solleva un'eccezione
- `d.get(key, default)`
  - o se key è presente nel dizionario, restituisce il valore associato alla chiave
  - o se key non è presente, restituisce il valore di default
- `d.pop(key)`
  - o se key è presente nel dizionario, elimina la chiave e il valore ad essa associato (il valore associato si può riusare invocando nuovamente il pop e associandolo ad una variabile)
  - o se key non è presente, solleva un'eccezione
- `d.values()` → restituisce una lista contenente tutti i valori presenti nel dizionario
- `d.keys()` → restituisce una lista con tutte le chiavi nel dizionario
- `d.items()` → restituisce una lista con le tuple (key, value)
- `list1 = sorted(d)` → restituisce una lista ordinata con le chiavi del dizionario
- `list1 = sorted(d.items())` → restituisce una lista ordinata per chiavi delle tuple (key, value)

## STRUTTURE COMPLESSE

**RECORD DI DATI** = se abbiamo file contenenti record di dati (come stato – popolazione – densità stato), questi vanno suddivisi per categorie; esempio:

```
Afghanistan:32738376
Akrotiri:15700
Albania:3619778
Algeria:33769669
American Samoa:57496
Andorra:72413
Angola:12531357
Anguilla:14108
```

```
def extract_record(infile):
    record = {}
    line = infile.readline()
    if line != "":
        fields = line.split(":")
        record["country"] = fields[0]
        record["population"] = int(fields[1])
    return record
```

```
infile = open("populations.txt", "r")
record = extract_record(infile)
while len(record) > 0:
    print(f"{record['country'][:20]} {record['population'][:10]}")
    record = extract_record(infile)
```

Quando leggiamo i file in formato **CSV**, possiamo leggere le varie righe con l'ausilio di:

- `csv.reader(csvfile)` → restituisce i campi di una riga sotto forma di lista
- `csv.DictReader(csvfile)` → restituisce i campi di una riga sotto forma di dizionario

**DA CSV A LISTE DI DIZIONARI** (e poi ordinare queste liste di dizionari):

```
import csv

file = open(FILENAME, 'r')
reader = csv.DictReader(file)

studenti = []
for record in reader:
    studenti.append(record)
file.close()

from operator import itemgetter

# ordine di lettura dal file
print(studenti[0], studenti[1], studenti[2])

# ordina per matricola
studenti.sort(key=itemgetter('matricola'))
print(studenti[0], studenti[1], studenti[2])

# ordina per nome
studenti.sort(key=itemgetter('nome'))
print(studenti[0], studenti[1], studenti[2])

# ordina per cognome
studenti.sort(key=itemgetter('cognome'))
print(studenti[0], studenti[1], studenti[2])
```

L' attributo `key=` è presente anche per le funzioni `max` e `min`:

- `max(esami, key=itemgetter("voto"))`
- `min(caselle, key=itemgetter(0)) #tramite l'indice`

⚠ Si possono combinare `enumerate` e `itemgetter` per trovare in un solo passaggio il valore del massimo e la sua posizione / indice.

**MODULI** = dividere il codice in file sorgenti separati (se programma troppo grande o si lavora in gruppo); il modulo principale contiene la funzione `main()`, mentre i moduli supplementari contengono le funzioni, variabili e costanti di supporto. Come per i moduli della libreria standard, per importare un modulo si fa:

```
import pygame #Importiamo tutto il modulo, ma poi dobbiamo scrivere le funzioni con esso (es. pygame.set())
from pygame import mixer, cdrom #Importo soltanto i sottomoduli / funzioni che mi interessano
from pygame import * #Importo tutto e non devo anteporre il modulo quando invoco funzioni
```

## T1. RAPPRESENTAZIONE DATI

$$252 = 2 \cdot 10^2 + 5 \cdot 10^1 + 2 \cdot 10^0$$

Posso esprimere dunque i numeri come  $A = \sum_{i=0}^{N-1} a_i \cdot B^i$  con **B = base** (che nel nostro sistema decimale = 10, ma nel binario = 2 e nell'esadecimale = 16)

### SISTEMA BINARIO:

- Base [B] = 2
- Cifre = {0, 1}
- **BIT** = cifra binaria (BInary DigiT); il BIT è contenuto in una cella della RAM. Nel computer si usa il sistema binario perché ci basiamo sui condensatori (se cella del condensatore è carica abbiamo 1, se scarica abbiamo 0).

### CONVERSIONE BINARIO-DECIMALE:

$$1111_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 2 + 1 = 15_{10}$$

4 BIT = 1 NIBBLE / 8 BIT = 1 BYTE

- Se l'ultima cifra di un numero in binario è = 1, il numero è dispari
- Se è = 0, il numero è pari

**CONVERSIONE DECIMALE-BINARIO:** si divide il numero per 2 più volte e si considerano i resti (presi però al contrario, ovvero dalla MSB [most significant bit] alla LSB [least significant bit]; esempio (vedi a destra).

### BINARI PURI (senza segno):

- se abbiamo N bit (es. pc a 64 bit o 32 bit), possiamo generare un valore compreso tra:

$$0 \leq x \leq 2^N - 1 \rightarrow 2^N - 1 = MAX \text{ valore}$$

DEC → BIN

252		1	0	← LSB
126		0	0	
63		1	1	
31		1	1	
15		1	1	
7		1	1	
3		1	1	
1		1	1	← MSB
0		0	0	

$$\begin{aligned} 252 &= 11111100_2 \\ 10 &\quad 7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1 \ 0 \\ 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 &= \\ 128 + 64 + 32 + 16 + 8 + 4 &= 252 \end{aligned}$$

**WORD** = una riga di memoria (che in un pc a 64 bit, WORD lunga 64) [ricordiamo che una RAM è costituita varie righe, quindi varie WORD, lunghe N bit, con N = numero di bit della nostra architettura {es. 32 bit o 64 bit}].

**SOMMA IN BINARIO:** quando sommo  $1 + 1$  ho un CARRY (in figura è in verde), ovvero un riporto; se finisco i BIT utilizzabili (come se in figura avessi un ulteriore riporto oltre ai 4 BIT), ho un OVERFLOW / INTERRUPT (cioè un errore, che dovrà essere gestito).

1	1	1	0	+
0	1	1	1	=
0	1	1	1	=
1	1	0	1	-
1	0	0	1	=
0	1	1	0	=
0	0	1	1	=

**SOTTRAZIONE IN BINARIO:** ho un BORROW (prestito dalla cifra dopo)

**SISTEMA OTTALE (0-7):** si converte nella stessa maniera, ma devo ricordare che la massima cifra ottale, ovvero il 7, è rappresentato in binario come 111 (utile per scrivere i numeri binari di 3 bit → 1 cifra ottale = 3 BIT).

**SISTEMA ESADECIMALE** = {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}; {A,B,C,D,E,F} = {11,12,13,14,15,16}: è utile per scrivere in modo compatto i numeri binari di 4 bit (1 cifra esadecimale = 4 BIT).

## NUMERI RELATIVI (con segno):

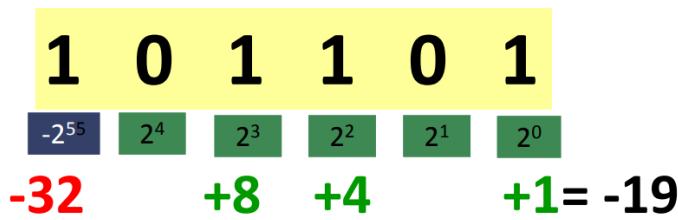
- **CODIFICA MODULO E SEGNO:** il bit MSB è riservato al segno (0 = segno + / 1 = segno - ); esempio:  
 $+3_{10} \rightarrow 0011_{M\&S}$  (*M&S = modulo e segno*)  
 $-3_{10} \rightarrow 1011_{M\&S}$  (*M&S = modulo e segno*)

Svantaggi: doppio zero [+0, -0] e operazioni complesse

Con la rappresentazione M&S su N bit, rappresento numeri che stanno tra:

$$-(2^{N-1} - 1) \leq x \leq +(2^{N-1} - 1)$$

- CODIFICA IN COMPLEMENTO A 2 (CONSIGLIATA): il bit MSB ha peso negativo, dunque:
    - se MSB = 0, positivo [+]
    - se MSB = 1, negativo [-]
    - DA CA2 A DECIMALE:



- DA DECIMALE A CA2:
    - se numero positivo → solita conversione
    - se numero negativo → solita conversione, poi trasformo i bit 0 in 1 (e viceversa) [ovvero COMPLEMENTO A 1] e infine sommo 1 sui bit:

$$+15 \text{ su 5 bit in c.a.2} \Rightarrow +15 = 01111_2 \Rightarrow \mathbf{01111}$$

$$\begin{array}{rcl} -12 \text{ su 5 bit in c.a.2} & \Leftrightarrow +12 = 01100_2 \\ \text{complementiamo i bit} & \Rightarrow & 10011 \\ \text{sommiamo } +1 \text{ (su bit)} & \Rightarrow & 10011 + \\ & & 00001 = \hline \end{array}$$

- #### • OPERAZIONI COMPLEMENTO A 2:

- SOMMA in CA2:
    - Discordi ( $P+N$  o  $N+P$ ) → non consideriamo il carry (il riporto ad un bit dopo)
    - Concordi ( $P+P$  o  $N+N$ ) → se cambia il segno del numero (per esempio sommando  $P+P$  ottengo un numero  $N$ ), ho OVERFLOW (se il sistema ha raggiunto il massimo numero di bit), altrimenti prendo anche il carry.

Con la CA2 posso rappresentare numeri che stanno tra:

$$-(2^{N-1}) \leq x \leq +(2^{N-1} - 1)$$

## RAPPRESENTAZIONE NUMERI REALI

- **VIRGOLA FISSA** = dati N bit, riservarne M per la parte frazionaria e  $(N - M)$  per la parte intera
  - **VIRGOLA MOBILE** = implementare negli N bit la notazione esponenziale (“scientifica”)  $[x = \pm M \times 2^E]$

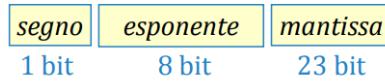
Nella memoria del calcolatore si memorizzano:

- Segno
  - Esponente (con il suo segno)
  - Mantissa

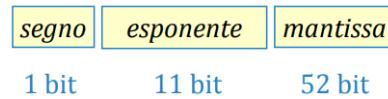


## FORMATO IEEE-754:

- IEEE 754 SP: (**float**)



- IEEE 754 DP: (**double**)



Dato che il calcolatore è in grado di manipolare solo numeri, per gestire dati non numerici deve creare una corrispondenza tra oggetti e numeri, assegnando agli oggetti un codice univoco.

CODICE ASCII = 8 bit per rappresentare:

- 52 caratteri alfabetici (a...z, A...Z)
- 10 cifre (0...9)
- Punteggiatura
- Caratteri di controllo (NL = new line)

UNICODE = 21 bit per rappresentare tutti i caratteri e le emoji

UTF-8 (codifica di Unicode su file più usata):

- 1 byte per caratteri ASCII (MSB = 0)
- 2 byte per caratteri latini, greci, ebraici, arabi...
- 3 byte per altre lingue di uso comune
- 4 byte per caratteri rarissimi

Un testo può essere memorizzato come:

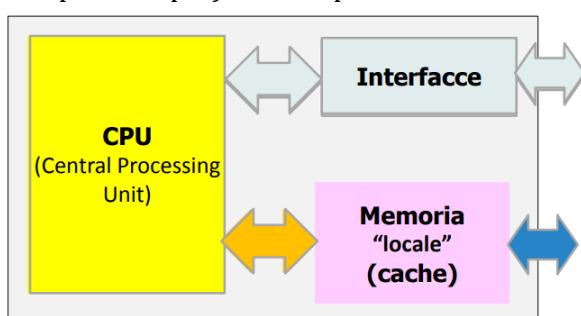
- Formattato = sequenze di byte che gestiscono l'aspetto del testo (font, spaziatura...)
- Non formattato = memorizzati solo i caratteri del testo

## T2. ARCHITETTURA DEGLI ELABORATORI

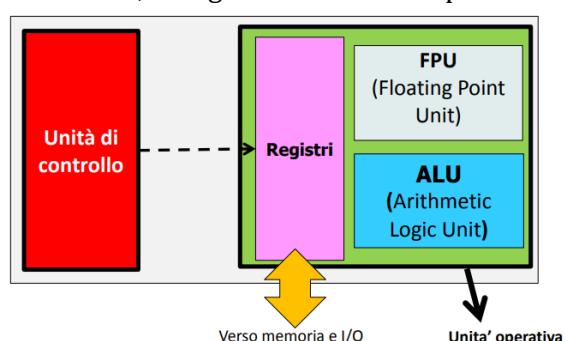
### COMPONENTI FONDAMENTALI:

- Unità di Input/Output (porte, display, speaker...)
- Unità di elaborazione (microprocessore e circuiti per l'esecuzione delle istruzioni)
- Memoria (memorizza dati e programmi ed è necessaria per l'elaborazione)

MICROPROCESSORE = è il circuito che esegue tutte le istruzioni (operazioni sui numeri, operazioni logiche, coordinamento delle istruzioni e degli errori, interfacce per spostare dati da/verso la memoria o da/verso unità di input e output); il microprocessore ha limitate capacità di memoria.



**CPU** (Central Processing Unit): nel FPU si usa la IEEE-754, nell'ALU invece la CA2. La Control Unit controlla tutte le unità della CPU, cioè gestisce le sue componenti.



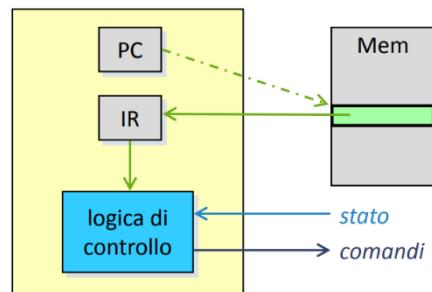
PARTI DELLA CPU:– **UNITÀ OPERATIVA [UO]:**

- **REGISTRI [Reg]** (da 8 a 128): elementi di memoria locale (RAM) usati per conservare temporaneamente dati e istruzioni; ogni trasferimento da processore a memoria (e viceversa) avviene tramite i registri. Ce ne sono 2 tipi:
    - Registro istruzioni e dati
    - Registro dei Flag = contiene un insieme di bit che segnalano i risultati dell'ultima operazione eseguita (usati anche per gli overflow nel carry che supera il numero di bit)
  - **ALU** = fa tutti i calcoli sui numeri interi / **FPU** = fa i calcoli sui numeri reali (virgola)
- Un'operazione FPU è molto più lenta di un'operazione ALU

– **UNITÀ DI CONTROLLO [UC]** = composta da:

- **PC** (Program Counter) = indica l'indirizzo della cella di memoria che contiene la prossima istruzione
- **IR** (Instruction Register) = memorizza temporaneamente l'operazione corrente da eseguire
- **Logica di controllo** = emette gli ordini per le varie unità

→ Esecuzione di un'istruzione: preleva dalla memoria l'istruzione nella posizione indicata da PC, incrementa il valore di PC (in modo da contenere la prossima istruzione).

PARTE USATA – OPERAZIONE FATTA:

Categoria	Operazioni specifiche
Operazioni aritmetiche (ALU)	+ , - , * , /, resto tra numeri interi
Operazioni aritmetiche (FPU)	+ , - , * , /, resto tra numeri reali
Operazioni logiche (ALU)	Operazioni su quantità logiche (unione, intersezione, negazione)
Confronti (ALU)	<, >, =, <=, >=, !=
CPU + Memoria	Trasferimento dati dalla/verso memoria
CPU + unità di I/O (+ Memoria)	Lettura/scrittura da/su dispositivo

**CLOCK** = elemento di temporizzazione che genera un riferimento temporale comune per tutto l'elaboratore:

- T = periodo di clock [s]
- f = frequenza di clock (1/T) [Hz]

→ **CICLO-MACCHINA** = tempo in cui viene svolta un'operazione elementare = multiplo intero di T di clock

**MEMORIA** = memorizza i dati e le istruzioni necessarie all'elaboratore per operare; è organizzata in CELLE (minima unità accessibile direttamente) e ad ogni cella viene associato un indirizzo numerico per identificarla. Ogni cella di memoria contiene una quantità fissa di bit (word = multiplo del byte), identica per tutte le celle e accessibile con un'unica istruzione [tipicamente la dimensione della cella di memoria coincide con quella dei registri]. Le memorie veloci costano molto e sono volatili; le memorie non volatili costano poco, ma sono lente. Dunque è necessaria una **GERARCHIA DI MEMORIA** per ottimizzare costi e tempi:

- Memorie più veloci, costose e volatili → più vicine al processore
- Memorie più lente, economiche e non volatili → più lontane dal processore

TIPI DI MEMORIE:

Metrica	SRAM (cache)	DRAM	FLASH	Disco
Dimensione	MB (1-16)	GB (4-16)	GB (16-512)	TB (>1)
Velocità	1-5 ns	50-150 ns	~10ms*	~10ms
Costo	10-5/byte	10-8/byte	10-9/byte	10-10 \$/byte
Persistenza	Volatile	Volatile	Non volatile	Non volatile

**INTERCONNESSIONI (BUS)** = sistema circolatorio del PC; un'unica linea di connessione per connettere tutti i componenti dell'elaboratore (fisicamente agganciati ad essa).

$$N_{Connessioni} = (N_{Unità\ da\ connettere})^2$$

Un sola linea di connessione?

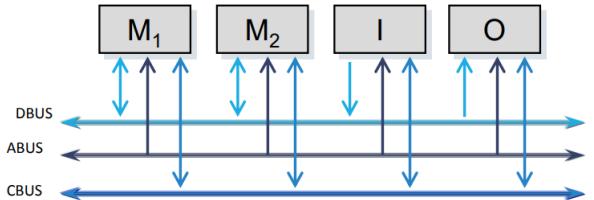
- Vantaggi:
  - Costi ridotti di produzione
  - Estendibilità e facili aggiunte di nuovi dispositivi
- Svantaggi
  - Lentezza
  - Sovraccarico del processore (CPU), che fa da master sul controllo del bus

Caratteristiche di un bus:

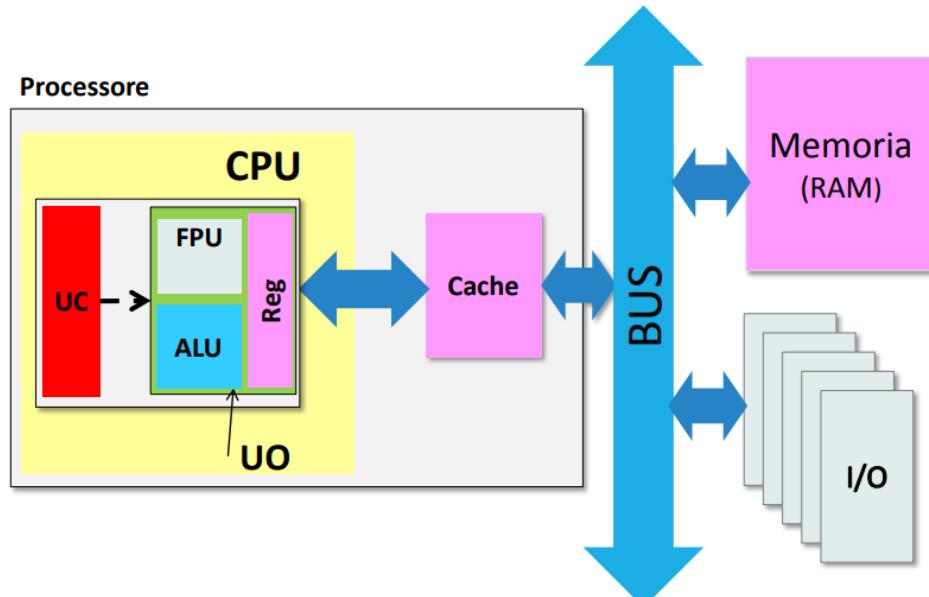
- Trasporta un solo dato per volta
- Frequenza = N° di bit trasportati al secondo
- Ampiezza = N° di bit di cui è costituito un singolo dato

Un singolo bus è suddiviso in 3 “**SOTTO-BUS**”:

- Bus dati → **DBus**
- Bus degli indirizzi → **ABus** (la sua dimensione indica il max n° di celle di memoria indirizzabili)
- Bus di controllo → **CBus** (la sua dimensione indica la dimensione di una cella di memoria)



→ Massima memoria interna fisicamente presente:  $Max\ memoria = 2^{|Abus|} \times |Dbus| \text{ bit}$



⚠ Parallelismo di memoria = dimensione della singola cella di memoria (= ampiezza DBus = dimensione registri) → Definizione utile al fine dell'esame!

#### FONTI:

- “Concetti di Informatica e Fondamenti di Python” [Apogeo, Horstmann & Necaise]
- Slides Professor Bernardi