



**UNIVERSITÀ  
DEGLI STUDI  
DI TRIESTE**



**Corso di Laurea in Ingegneria Clinica e Biomedica  
Informatica Medica I**

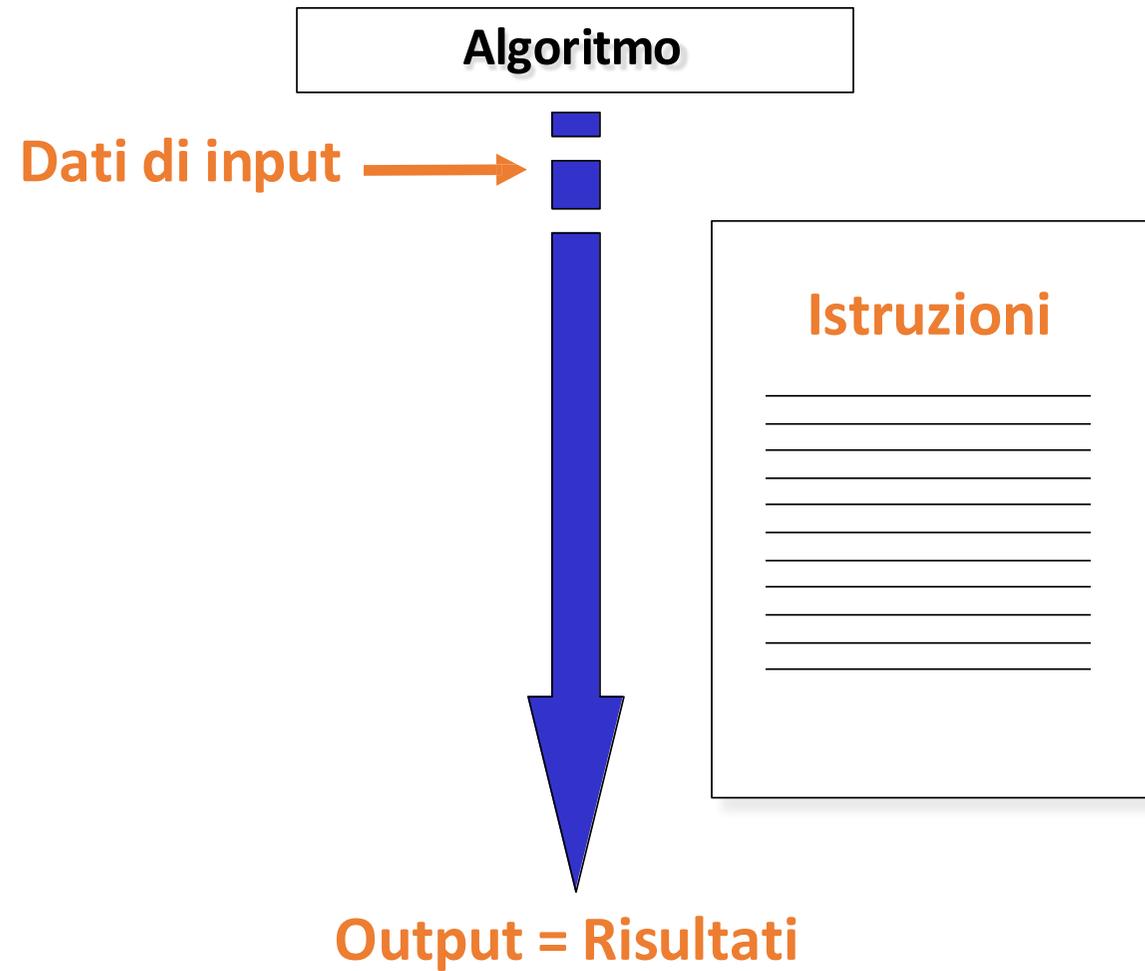
**IL LINGUAGGIO PYTHON**

***Prof. Aleksandar Miladinović***

# Il linguaggio Python

- Linguaggio di scripting a oggetti
- Supporta la programmazione strutturata
- Caratteristiche principali:
  - Semplicità di codifica
  - Leggibilità del codice (fa uso di indentazione)
  - Permette anche la creazione di software molto complessi

# PROGRAMMAZIONE ALGORITMICA



# PROGRAMMAZIONE ALGORITMICA (2)

**DATO UN VETTORE IN INGRESSO**  
[2 77 1 935 11 19 773 15 3]



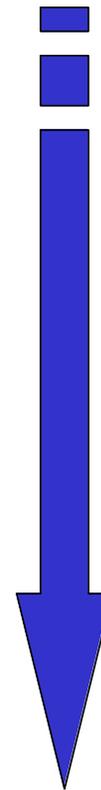
**VOGLIO IN USCITA IL VETTORE ORDINATO**



Cerco il minimo  
Lo metto da parte  
Lo elimino dal vettore  
Cerco il minimo nel vettore rimanente



**VETTORE IN USCITA**  
[1 2 3 11 15 19 77 773 935]

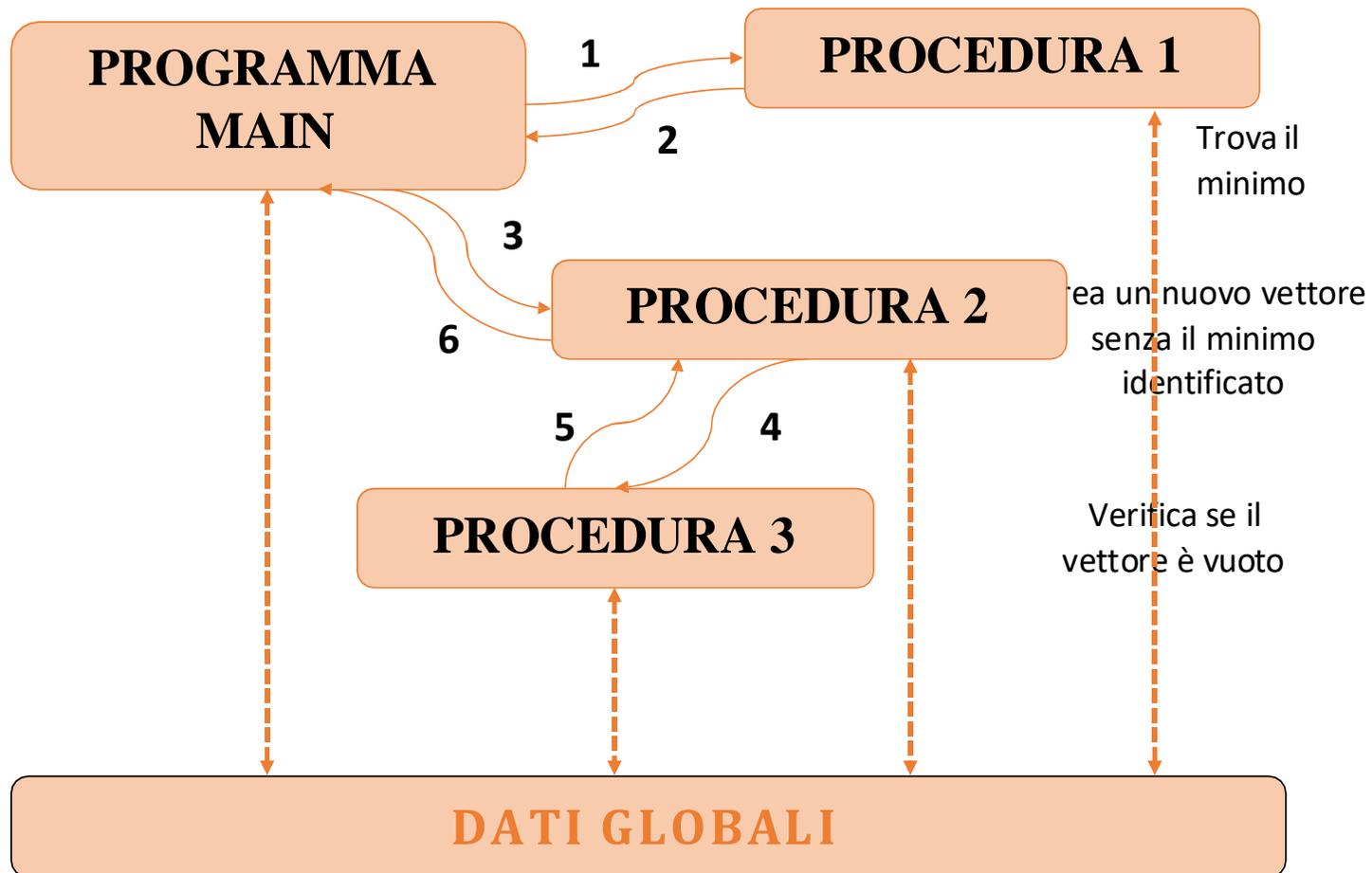


# PROGRAMMAZIONE NON STRUTTURATA

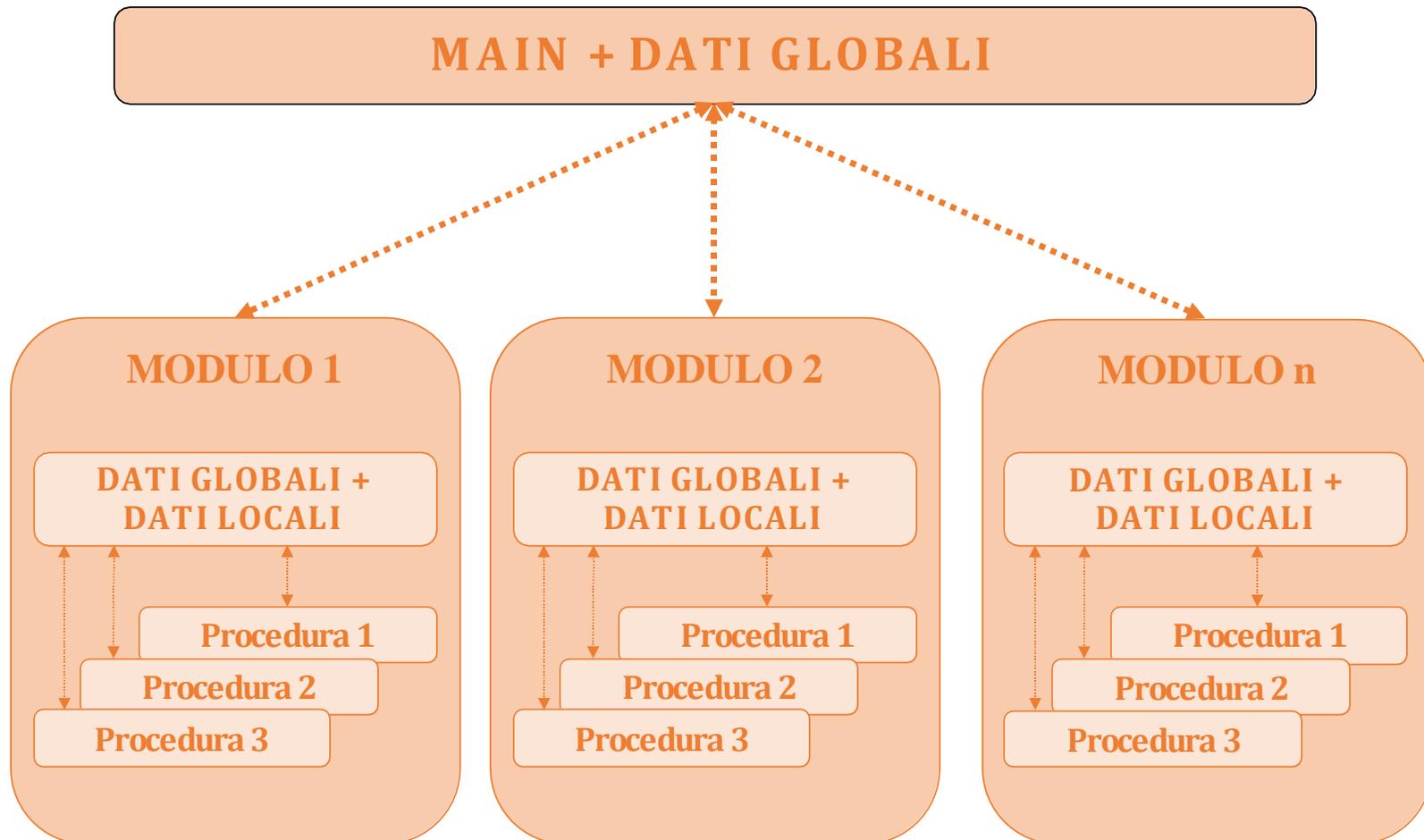
**STRUTTURE  
DATI**

**PROGRAMMA  
MAIN**

# PROGRAMMAZIONE STRUTTURATA PROCEDURALE



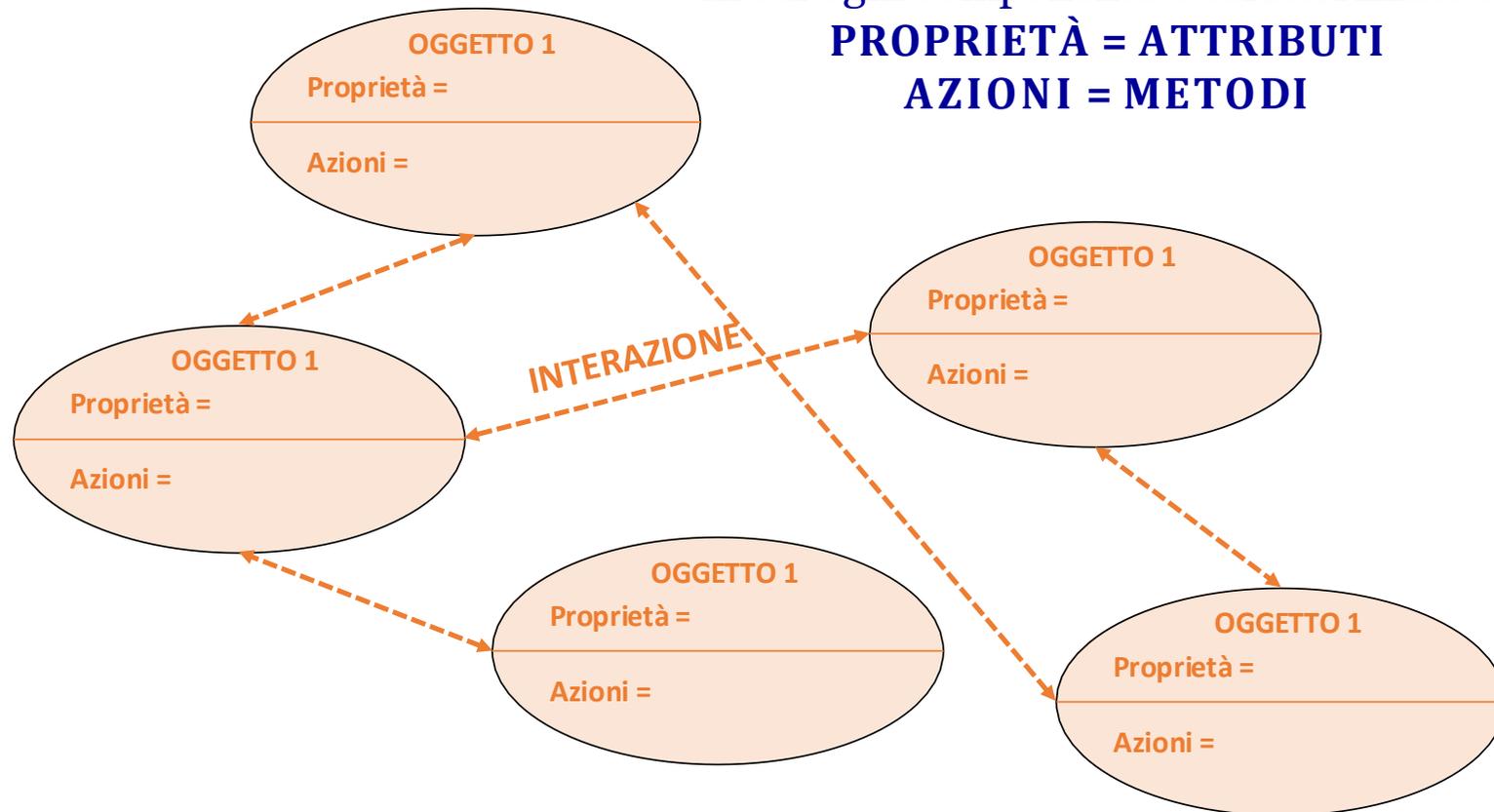
# PROGRAMMAZIONE STRUTTURATA MODULARE



# PROGRAMMAZIONE A OGGETTI

## SISTEMA

Un sistema è un insieme di  
entità interagenti = **OGGETTI**  
in cui ogni componente è caratterizzato da  
**PROPRIETÀ = ATTRIBUTI**  
**AZIONI = METODI**



# ALGORITMI vs OGGETTI

## PROGRAMMAZIONE ALGORITMICA

- Sequenza di azioni
- Basato su DATI e FUNZIONI = PROGRAMMI
- Obiettivo: risolvere un PROBLEMA

## PROGRAMMAZIONE A OGGETTI

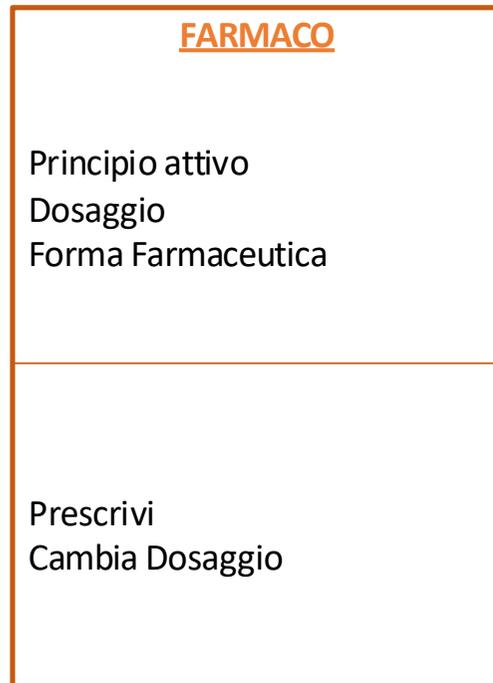
- Sistema = Insieme di oggetti
- Basato su OGGETTI fatti da AZIONI e ATTRIBUTI
- Obiettivo: gestire un SISTEMA

# CLASSI E OGGETTI

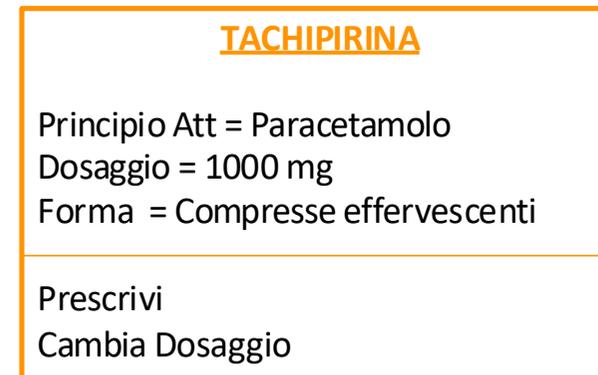
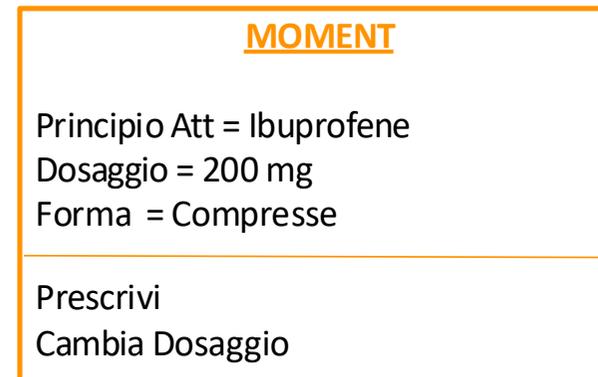
CLASSE = definizione

ATTRIBUTI =  
Caratteristiche

METODI =  
Comportamenti



OGGETTI = istanze di una classe



## ESEMPIO: OSSERVAZIONI

### MOMENT

Principio Att = Ibuprofene

Dosaggio = 200 mg

Forma = Compresse

Prescrivi

Cambia Dosaggio

I VALORI DEGLI ATTRIBUTI  
SONO SPECIFICI  
DELL'OGGETTO ISTANZIATO  
(ogni oggetto ha il suo insieme di  
valori)

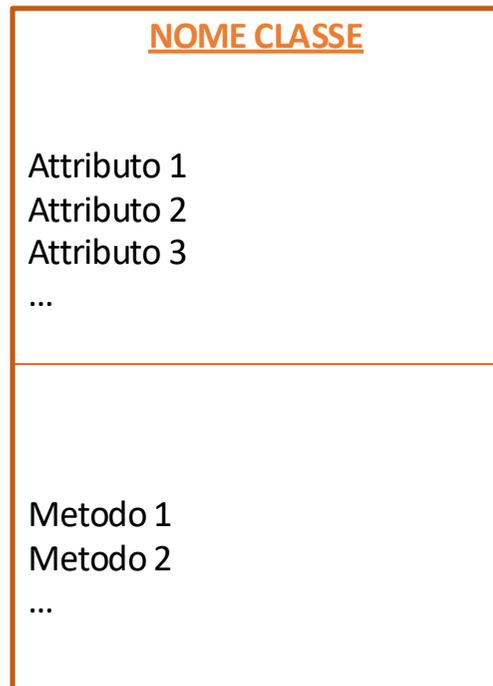
LE AZIONI SONO COMUNI A  
TUTTE LE ISTANZE DELLA  
CLASSE

# CLASSI E OGGETTI: DEFINIZIONE

## CLASSE

ATTRIBUTI =  
Caratteristiche

METODI =  
Comportamenti



## OGGETTO 1

Att 1 = Val 1  
Att 2 = Val 2  
Att 3 = Val 3

Metodo 1  
Metodo 2

## OGGETTI

## OGGETTO 2

Att 1 = Val 4  
Att 2 = Val 5  
Att 3 = Val 6

Metodo 1  
Metodo 2

# ATTRIBUTI E METODI

## ATTRIBUTI

- Descrivono le proprietà **statiche** dell'oggetto
- Nella programmazione gli attributi vengono realizzati attraverso l'uso delle **variabili** utilizzate dall'oggetto per memorizzare i dati

## METODI

- Descrivono le proprietà **dinamiche** dell'oggetto
- Nella programmazione i metodi vengono realizzati attraverso la scrittura di codice (**procedure** e **funzioni**) che implementano le operazioni dell'oggetto

# PROPRIETÀ DELLE CLASSI: EREDITARIETÀ

## *Classe genitrice*

farmaco
Principio attivo
Dosaggio
Forma Farmaceutica
Costo SSN
Prescrivi
Cambia Dosaggio

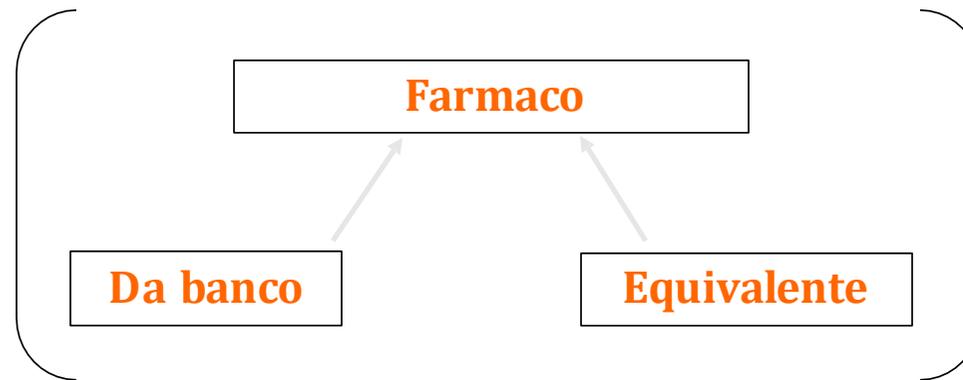


## *Nuova Classe*

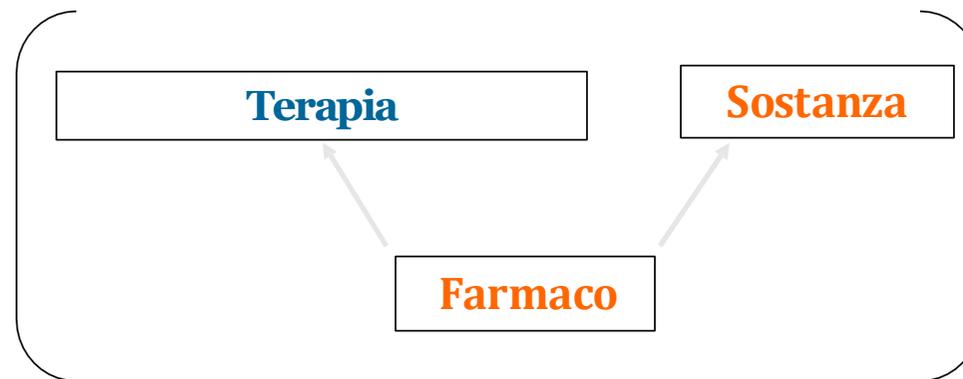
Farmaco di marca
Principio attivo
Dosaggio
Forma Farmaceutica
<i>Nome Commerciale</i>
Costo SSN
<i>Prezzo Pubblico</i>
Prescrivi
Cambia Dosaggio
<i>Calcola costo paziente</i>

# TIPI DI EREDITARETÀ

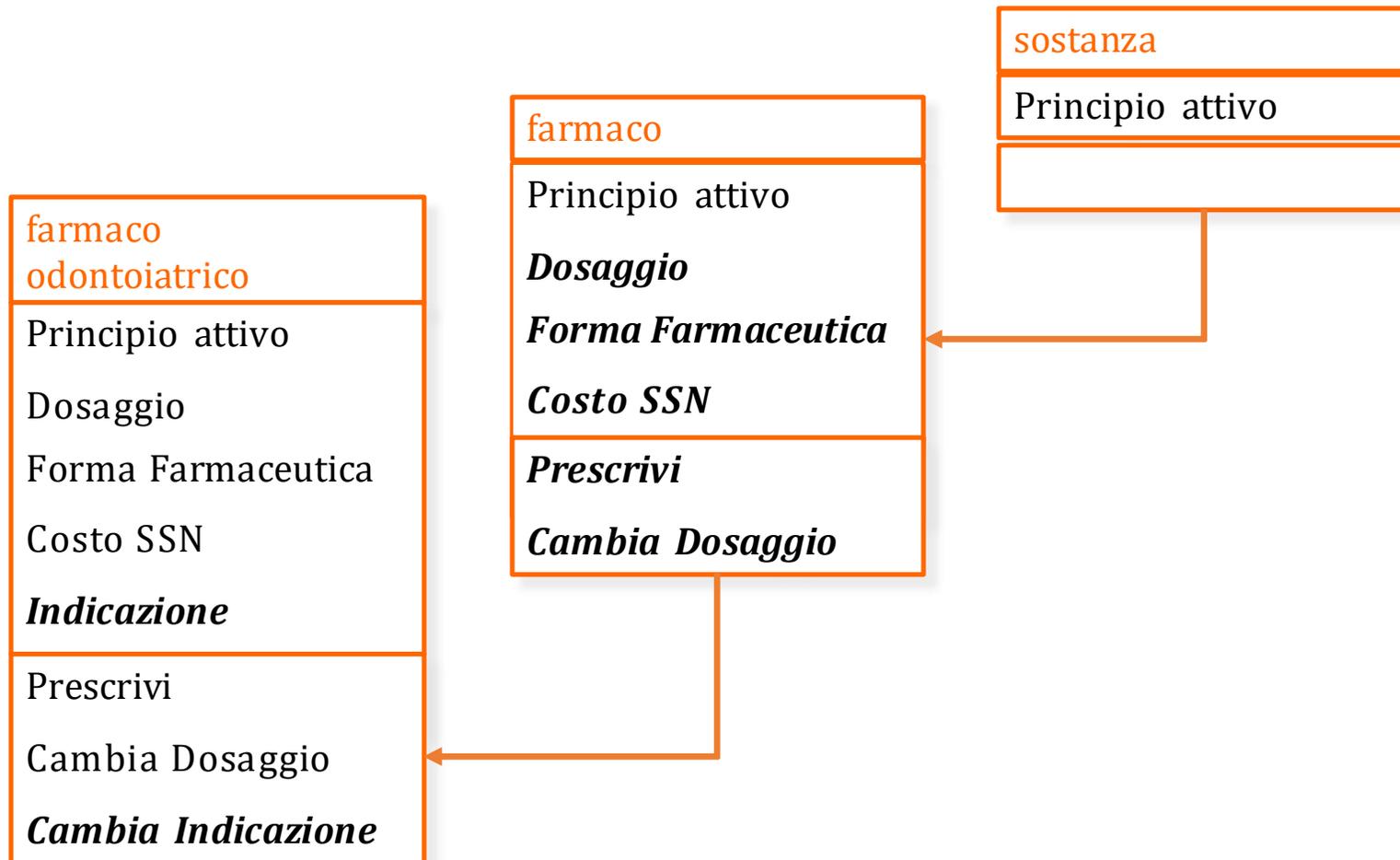
**Ereditarietà  
singola**



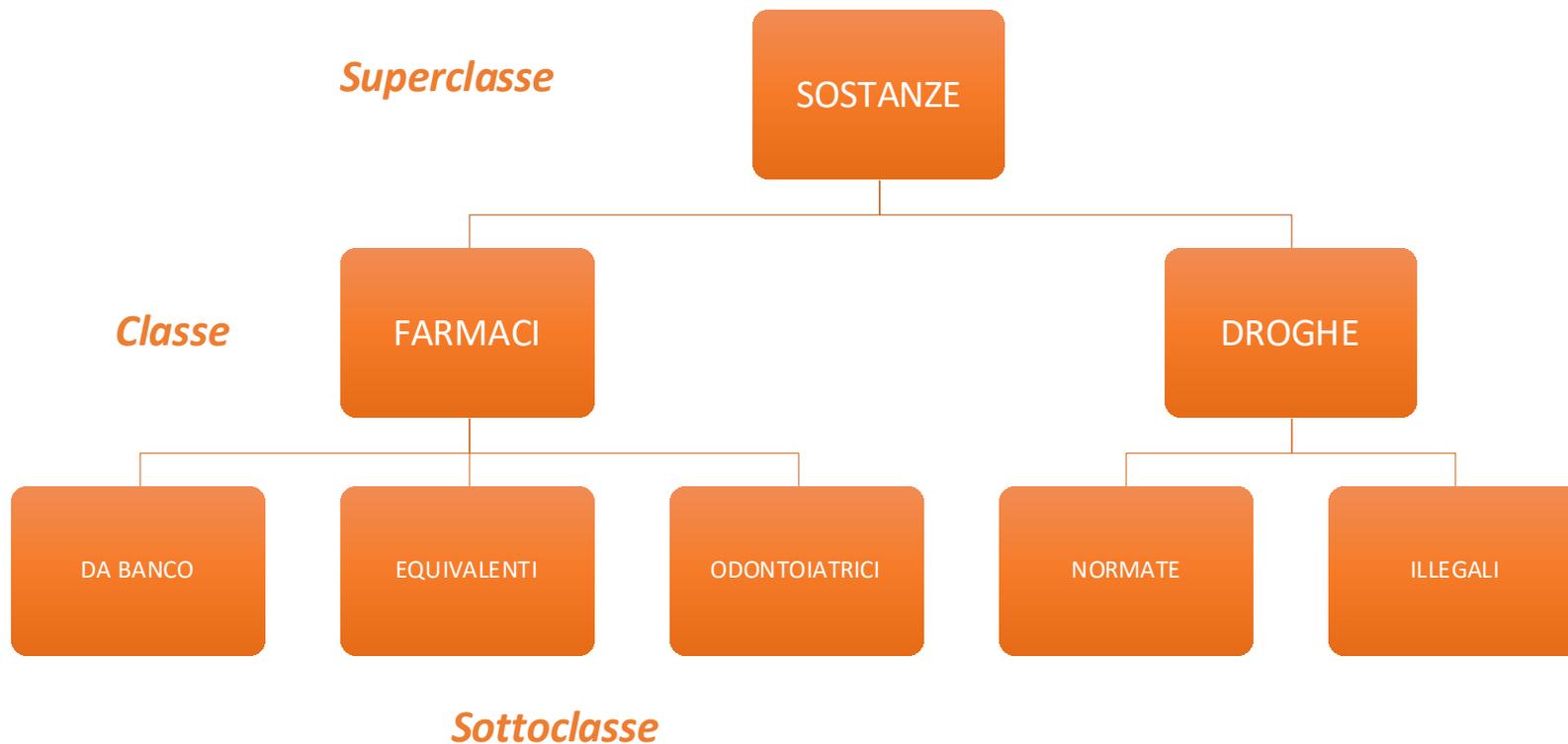
**Ereditarietà  
multipla**



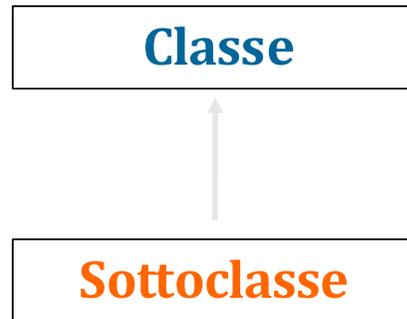
# ESEMPIO



# PROPRIETÀ DELLE CLASSI: GERARCHIE



# DEFINIZIONE DELL'EREDITARIETÀ



## ESTENSIONE

la sottoclasse  
AGGIUNGE NUOVI  
METODI/ATTRIBUTI

## RIDEFINIZIONE

la sottoclasse  
RIDEFINISCE I  
METODI

OVERRIDING = riscrittura del  
codice del metodo

# PROPRIETÀ DELLE CLASSI: POLIMORFISMO

**OVERRIDING** = i **metodi** possono **assumere forme diverse** (cioè *implementazioni diverse*) all'interno della gerarchia delle classi

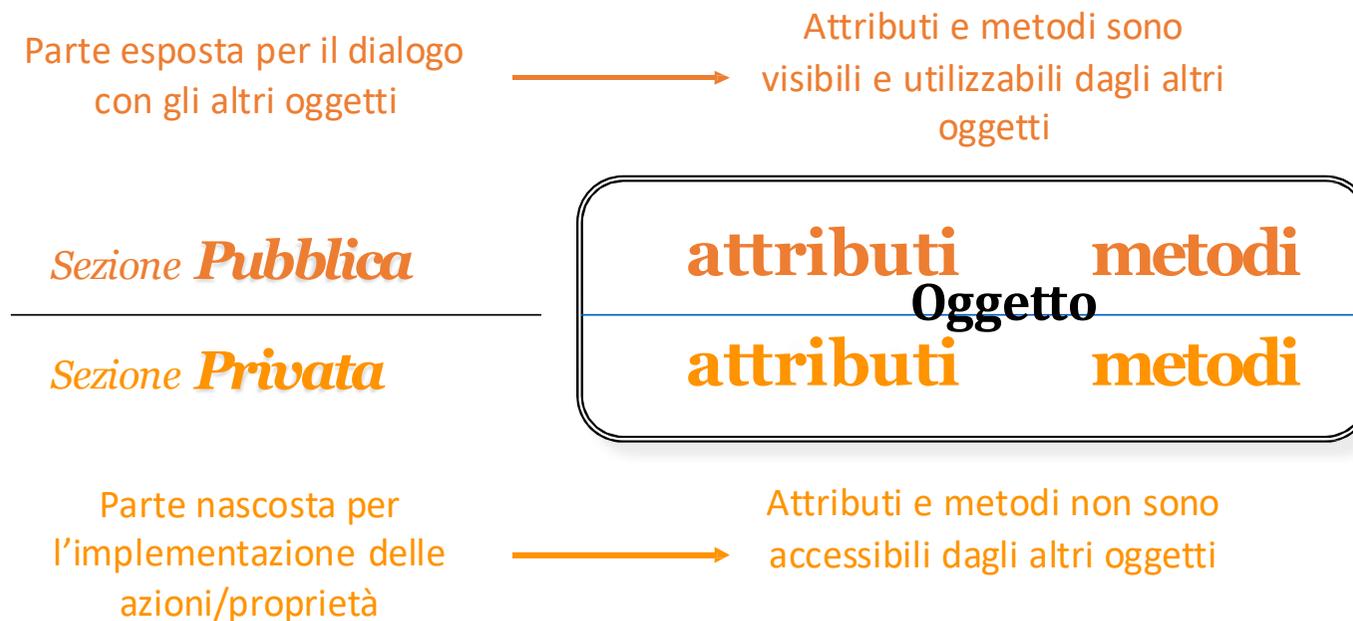
ES. IL METODO "PRESCRIVI" SARÀ IMPLEMENTATO DIVERSAMENTE NEL FARMACO DA BANCO E NEL FARMACO GENERICO

**OVERLOADING** = i **metodi** possono **assumere forme diverse** (cioè *implementazioni diverse*) all'interno della stessa classe

ES. IL METODO "CAMBIA DOSAGGIO" PUÒ RICHIEDERE O DI CAMBIARE IL NUMERO DI ASSUNZIONI O DI CAMBIARE IL NUMERO DI DOSI PER ASSUNZIONE

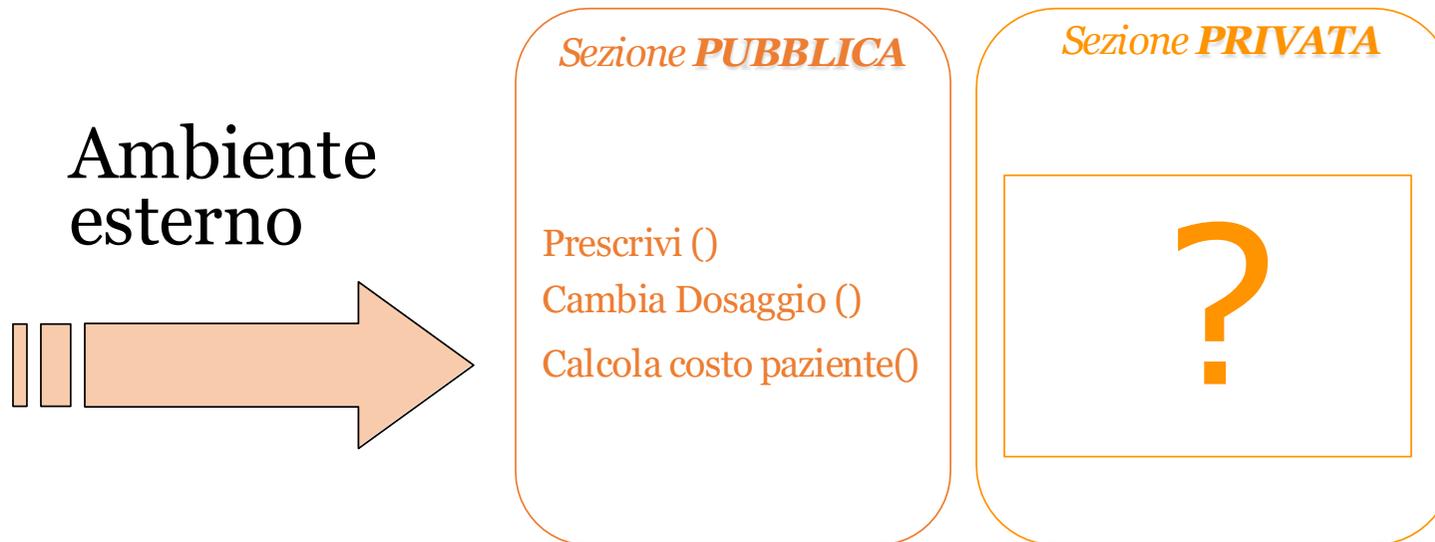
# INCAPSULAMENTO

- **Incapsulamento** = Proprietà dell'oggetto di incorporare al suo interno attributi e metodi
- **Information hiding** = mascheramento dell'informazione all'interno dell'oggetto  
→ espone solo metodi e attributi della sezione pubblica



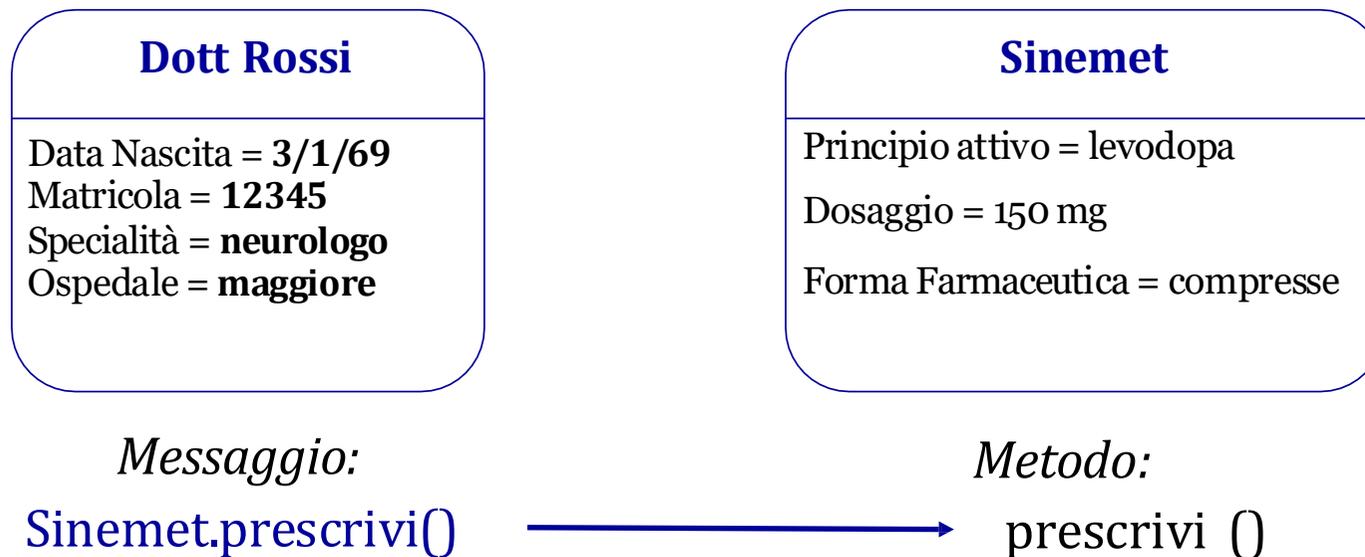
# INTERFACCIA

- **INTERFACCIA** = insieme dei messaggi inviabili all'oggetto/ricevibili dall'oggetto
- L'interfaccia **non consente di vedere come sono implementati i metodi**, ma ne permette il loro utilizzo e l'accesso agli attributi pubblici



# INTERAZIONE TRA OGGETTI: MESSAGGI

- Un programma ad oggetti è caratterizzato dalla presenza di tanti oggetti che **interagiscono fra loro attraverso il meccanismo dello scambio di messaggi**
- I messaggi possono:
  - Richiedere un'informazione su un oggetto
  - Modificare lo stato di un oggetto



# PYTHON

## Programmazione strutturata

- Blocchi di istruzioni delimitati da indentazione
- Terminatore di istruzione: andata a capo
- Nidificazione di blocchi di istruzioni: mediante il carattere “:”

## Linguaggio a oggetti

- Le istruzioni sono soltanto quelle necessarie per definire le strutture algoritmiche di iterazione e condizione e per definire funzioni
- Il resto è gestito da **chiamate a metodi di oggetti** definiti in **moduli** in cui sono definite le classi specifiche

# INSTALLAZIONE - PYTHON

- <https://www.python.org>
- Downloads → versione 3.x (attuale 3.9)
- Versione dell'interprete:
  - Python: versione 2.x (precedente)
  - Python3: versione 3.x (attuale)
- Shell di python:
  - Permette l'esecuzione in modalità interattiva (prompt dei comandi)
  - Permette di eseguire programmi scritti in Python (estensione .py)

## INSTALLAZIONE IDE

- Ci sono vari ambienti di sviluppo
- Usiamo Anaconda e Spyder
- Download Anaconda:  
<https://www.anaconda.com/products/individual>
- Tutte le opzioni sono di Default
- Chiede se installare PyCharm:
  - è una buona IDE per lo sviluppo di app per Python (a pagamento, ha un free trial)
  - A noi non serve perché useremo Spyder.

# SINTASSI

- **Indentazione:**

- Necessaria perché Python NON USA PARENTESI graffe
- Il livello di annidamento del codice è definito dal livello di indentazione

```
a= int(input("Inserisci un numero: "))
b= int(input("Inserisci un altro numero: "))
if a > b:
    print("Il numero più piccolo è ",b)
else:
    c=int(input("Inserisci un altro numero: "))
    if a+b<c:
        print(a+b)
print("Ciao e grazie!")
```

Primo livello

Primo livello

Secondo livello

Primo livello

Secondo livello

Secondo livello

Terzo livello

Primo livello

## SINTASSI: BLOCCHI DI ISTRUZIONI

- Ogni blocco di istruzioni è iniziato dal simbolo ":" e viene svolto in un livello di indentazione successivo
- Il blocco di istruzioni è "chiuso" dall'andata a capo e dal rientro dell'indentazione

```
a= int(input("Inserisci un numero: "))
b= int(input("Inserisci un altro numero: "))
if a > b: ←
→ print("Il numero più piccolo è ",b)
else: ←
→ c=int(input("Inserisci un altro numero: "))
    if a+b<c: ←
→ print(a+b)
print("Ciao e grazie!")
```

## SINTASSI: COMMENTI

- Singola linea di commento: #  
codice  
#comment  
codice
- Linee di commento multiple: """"  
codice  
"""" comment  
commento  
comment""""  
codice

# VARIABILI

- Python supporta diverse tipologie di variabili
- **NON** è necessario dichiarare il tipo di variabile a priori
- L'operatore di assegnazione è il simbolo "="

```
a=9          INTERO  
b=9/2       FLOAT  
c="Sara"     STRINGA
```

```
print(a," ", b," ",c)
```

# TIPI DI DATO

Nome	Tipo
Int	Intero
Long	Intero lungo
Float	Virgola mobile
Complex	Numero complesso
Bool	Boolean
Str	Stringhe

La funzione ***type(nomeVar)*** ritorna il tipo della variabile

```
>>>type(c) ← Richiesta  
<class 'str'> ← Risposta
```

# OPERAZIONI SU VARIABILI

Operazione	Operatore
Specificare il tipo di variabile	Parola chiave che identifica il tipo <code>&gt;&gt;&gt;a=str(9)</code>
Modificare il tipo di variabile	Parola chiave che identifica il tipo <code>&gt;&gt;&gt;a=9</code> <code>&gt;&gt;&gt;str(a)</code> <code>'g'</code>
Operazioni aritmetiche	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> <code>a**n</code> → a elevato alla potenza n <code>%</code> → resto della divisione intera (modulo) <code>//</code> → divisione intera
Contare la lunghezza di una stringa	<code>len(nomeVar)</code>
Composizione di stringhe	<code>format(...)</code> <code>&gt;&gt;&gt;a="Sara"</code> <code>&gt;&gt;&gt;b=50</code> <code>&gt;&gt;&gt;s="La signora {0} ha vinto {1} euro".format(a,b)</code> <code>&gt;&gt;&gt;print(s)</code> La signora Sara ha vinto 50 euro

# COMPARATORI

Operatore	Significato
<	Minore
<=	Minore o uguale
>	Maggiore
>=	Maggiore o uguale
==	Uguale
!=	Diverso
is	Uguaglianza tra oggetti (NB: si riferisce all'indirizzo)
is not	Disuguaglianza tra oggetti

# COMPARATORI: DIFFERENZA TRA == E IS

```
>>> a=[4,6,7]
>>> b=a
>>> b
[4, 6, 7]
>>> a.append('Danza')
>>> a
[4, 6, 7, 'Danza']
>>> b
[4, 6, 7, 'Danza']
>>> a is b
True
>>> c=a.copy()
>>> c
[4, 6, 7, 'Danza']
>>> c is a
False
>>> c==a
True
```

L'assegnazione mediante "=" assegna la variabile allo stesso contenuto di memoria

La copia genera un oggetto diverso, ma con lo stesso contenuto

`c == a` (comparazione di contenuti) è Vero  
`c is a` (comparazione di oggetti) è Falso

# STANDARD INPUT E STANDARD OUTPUT

- Input da tastiera (standard input): **input()**

- In generale, `input()` restituisce una stringa, anche se a terminale viene passato un numero

```
>>>a= input("Inserisci un numero: ")
>>>type(a)
<class 'str'>
```

- Per avere un input di tipo specifico, la funzione `input()` deve essere preceduta dal tipo di variabile desiderato

```
>>>a= int(input("Inserisci un numero: "))
>>>type(a)
<class 'int'>
```

- Output su terminale: **print()**

- Aggiunge automaticamente l'andata a capo al termine della stringa
- Se vengono passati più parametri, viene automaticamente visualizzato uno spazio tra di essi
- Se non si vuole lo spazio, deve essere indicato il tipo di separatore mediante il parametro `sep=«...»`

```
>>>c="Sara»
>>>print("Ciao",c,"come stai?")
Ciao Sara come stai?
>>>print("Ciao",c,"come stai?",sep="_")
Ciao_Sara_come stai?
```

# STRUTTURE DATI

Nome	Tipologia
<b>list</b>	Lista <ul style="list-style-type: none"><li>- Sequenze di dati di qualsiasi tipo</li><li>- Includono anche gli array</li><li>- Gli elementi sono indirizzabili mediante la loro posizione nella lista (partendo da 0): nomeLista[i]</li></ul>
<b>tuple</b>	Tupla <ul style="list-style-type: none"><li>- Lista non modificabile</li></ul>
<b>set</b>	Insieme <ul style="list-style-type: none"><li>- Insiemi di dati di qualsiasi tipo</li><li>- Non può contenere valori duplicati</li><li>- Possono essere applicati operatori insiemistici: unione ( ), intersezione (&amp;), differenza (-), differenza simmetrica (^)</li><li>- Non supportano indicizzazione</li></ul>
<b>frozenset</b>	Insieme non modificabile
<b>dict</b>	Dizionario <ul style="list-style-type: none"><li>- Rappresentano array associativi, ossia liste composte da coppie di elementi (chiave, valore)</li></ul>
<b>file</b>	File persistente

# LISTE

- Collezioni di dati (anche di diverso tipo)
  - ORDINATI → supportano l'indicizzazione
  - MUTABILI → posso operare sui dati contenuti (tuple sono la versione immutabile)
  - POSSONO contenere DUPLICATI
- Si definiscono:
  - Mediante assegnazione inserendo gli elementi tra [...]  
`A = [1,2,3,'tavolo']`
  - Usando il costruttore della classe list  
`A = list ()`  
`A = list ([1,2,3,'tavolo'])`
- L'accesso agli elementi della lista è effettuato utilizzando l'indice (partendo da 0)  
`A[3] → 'tavolo'`
- Se la lista è N-dimensionale, l'accesso si effettua inserendo ogni indice dimensionale all'interno di []  
`MultiA[i][j]...[N]`
- La lista può essere manipolata usando metodi e operatori
- Gli "array" sono liste

# OPERAZIONI SU LISTE

Operation	Result
<code>s[i] = x</code>	i-esimo elemento è sostituito da x
<code>s[i:j] = t</code>	Gli elementi da i a j sono sostituiti da t (deve essere un iterable – lista, tupla, stringa)
<code>del s[i:j]</code>	Cancella gli elementi da i a j (escluso j)
<code>s.append(x)</code>	Aggiunge x in fondo alla lista
<code>s.clear()</code>	Cancella tutti gli elementi della lista
<code>s.copy()</code>	Crea una copia di tutti gli elementi della lista
<code>s.extend(t)</code> or <code>s += t</code>	Aggiunge a s il contenuto di t
<code>s *= n</code>	Modifica s ripetendone il contenuto n volte
<code>s.insert(i, x)</code> or <code>s[i:i] = [x]</code>	Inserisce x nella posizione i-esima (spostando gli elementi in avanti)
<code>s.pop(i)</code>	Estrae e elimina l'i-esimo elemento (se <code>s.pop()</code> → elimina l'ultimo elemento)
<code>s.remove(x)</code>	Rimuove il primo elemento =x
<code>s.reverse()</code>	Inverte l'indicizzazione di s
<code>s.sort()</code>	Ordina in modo crescente gli elementi in s (devono essere ordinabili)
<code>len(s)</code>	Ritorna la lunghezza della lista

# SET

- Collezioni di dati (anche di diverso tipo)
  - NON ORDINATI → non supportano l'indicizzazione
  - MUTABILI → posso operare sui dati contenuti (frozenset sono la versione immutabile)
  - NON POSSONO contenere DUPLICATI
- Si definiscono:
  - Mediante assegnazione inserendo gli elementi tra {...}  
`A = {'Cane', 'Gatto', 'Topo', 'Casa'}`
  - Usando il costruttore della classe set  
`A = set()`  
`A = set({'Cane', 'Gatto', 'Topo', 'Casa'})`
- Supportano le operazioni insiemistiche (unione, intersezione, differenza)
- Poichè non contengono duplicati, è possibile facilmente cercare valori all'interno dell'insieme (funzione "in")

# OPERAZIONI SU SET

Operation	Result
<code>x in s</code>	Verifica se <code>x</code> è un elemento di <code>s</code>
<code>x not in s</code>	Verifica se <code>x</code> non è un elemento di <code>s</code>
<code>len(s)</code>	Ritorna il numero di elementi di <code>s</code>
<code>s.isdisjoint(s1)</code>	Verifica se <code>s</code> e <code>s1</code> sono insiemi disgiunti (=intersezione nulla)
<code>s.issubset(s1)</code> or <code>s&lt;s1</code>	Verifica se <code>s</code> è sottoinsieme di <code>s1</code>
<code>s.issuperset(s1)</code> or <code>s&gt;s1</code>	Verifica se <code>s1</code> è sottoinsieme di <code>s</code>
<code>s.clear</code>	Cancella tutti gli elementi dell'insieme
<code>s.copy()</code>	Crea una copia di tutti gli elementi dell'insieme
<code>s.add(x)</code>	Aggiunge <code>x</code> all'insieme
<code>s.pop()</code>	Estrae e elimina un qualunque elemento dall'insieme
<code>s.discard(x)</code>	Elimina <code>x</code> dall'insieme se esiste
<code>s.remove(x)</code>	Elimina <code>x</code> dall'insieme e ritorna errore se <code>x</code> non esiste
<code>s.union(s1,..)</code> or <code>s s1 ...</code>	Unione ( <code>s.update(s1,..)</code> or <code>s =s1 ...</code> fa l'unione e aggiorna <code>s</code> con il risultato)
<code>s.intersection(s1,..)</code> or <code>s&amp;s1&amp;...</code>	Intersezione ( <code>s.intersection_update(s1,..)</code> or <code>s&amp;=s1&amp;...</code> fa l'intersezione e aggiorna <code>s</code> con il risultato)
<code>s.difference(s1,..)</code> or <code>s-s1-...</code>	Differenza ( <code>s.difference_update(s1,..)</code> or <code>s-=s1-...</code> fa la differenza e aggiorna <code>s</code> con il risultato)
<code>s.symmetric_difference(s1)</code> or <code>s^s1</code>	Differenza simmetrica ( <code>s.symmetric_difference_update(s1)</code> or <code>s^=s1</code> fa la differenza simmetrica e aggiorna <code>s</code> con il risultato)

# DICT

- Array associativi (coppie chiave-valore)
- Servono per fare mapping
- Si definiscono mediante parentesi graffe all'interno delle quali si elencano le coppie di "chiave-valore"
- Ogni chiave può avere un solo valore (sovrascritto se ripetuto)

Dizionario\_VAS={'Per niente':0, 'Poco':3, 'Abbastanza':5, 'Molto':7, 'Moltissimo':10}

Dizionario\_VAS.keys() or list(Dizionario\_VAS) → ritorna tutte le chiavi

Dizionario\_VAS.values() → ritorna tutti i valori

# OPERAZIONI SUI DICT

Operation	Result
len(s)	Ritorna la lunghezza del dict
list(s)	Ritorna la lista di tutte le chiavi
del s[k]	Cancella l'elemento con chiave k
s[k]	Ritorna il valore dell'elemento associato alla chiave k
s.clear()	Cancella tutti gli elementi della lista
s.copy()	Crea una copia di tutti gli elementi della lista
k in s	Verifica se la chiave k si trova in s
k not in s	Verifica se la chiave k si trova in s
s[k]=x	Aggiorna il valore dell'elemento con chiave k a x
s.pop(k)	Estrae e elimina la coppia chiave k-valore

# ALGORITMI: STRUTTURA CONDIZIONALE

- Istruzione if-else

**if** *condizione {AND/OR/NOT condizione-i}*:

istruzione 1

istruzione 2

**elif** *{condizione}*:

istruzione 1

istruzione 2

**else:**

istruzione 1

istruzione 2

## ALGORITMI: CICLO WHILE

- Ciclo iterativo in cui il blocco di istruzioni al di sotto del while viene eseguito fintanto che la condizione rimane vera

**while** *condizione:*

istruzione 1

istruzione 2

...

## ALGORITMI: CICLO FOR

- Ciclo iterativo per un numero finito di iterazioni

```
for i in range (m,n):
```

```
    Istruzione 1
```

```
    Istruzione 2
```

```
    ...
```

- Tipo range:

***class range(stop)*** - sequenza di interi da 0 a stop-1

***class range(start, stop{, step})*** – sequenza di interi da start a stop-1 spostandosi di step (default 1)

# FUNZIONI

```
def nomeFunzione(parametro1,parametro2,...):           dichiarazione  
    istruzione 1  
    istruzione 2  
    ...  
    return(valore)                                     terminatore
```

Chiamata della funzione:

- nomeFunzione (valoreParametro1,valoreParametro2,...) – **notazione POSIZIONALE**
- nomeFunzione (parametro1 = valore, parametro2 = valore, ...) – **indipendente dalla posizione dei parametri**

## FUNZIONI: OSSERVAZIONI

- Si possono definire funzioni che non ritornano nessun valore
- È buona pratica **NON USARE** variabili GLOBALI all'interno delle funzioni ma incapsulare tutta l'esecuzione all'interno.
- È possibile definire funzioni **RICORSIVE** (funzioni che richiamano se stesse). Il numero di ricorsioni dipende dalla memoria disponibile

# FILE

- La persistenza dei dati è garantita dall'uso di file
- Funzioni principali:
  - `p = Open(nomeFile, modalità _apertura)`
    - “w” = write (aperto in scrittura, se esiste, viene sovrascritto, se non esiste, viene creato)
    - “a” = append (aperto in scrittura, se esiste, viene aggiornato, se non esiste, viene creato)
    - “r” = read
  - `p.write(dati_da_scrivere)` → consente di scrivere su file (vengono scritte delle stringhe, quindi va tutto trasformato in stringa prima di scrivere)
  - `p.read()` → legge l'intero contenuto del file (se sono numeri, devono essere trasformati dopo la lettura)
  - `p.readline()` → legge una riga per volta
  - `p.close()` → chiude il file dopo averlo utilizzato

# DEFINIZIONE DI CLASSI

- La classe si definisce mediante la parola chiave Class

```
class NomeClasse:
```

```
...
```

- Convenzioni:

- nome della classe maiuscolo (ciascuna parola che lo compone deve essere maiuscola)
- nome dei campi in minuscolo (se composti da più parole, separate da \_)

- Metodo `__init__` (costruttore)

```
def __init__(self, att1, att2, ...):
```

```
    self.att1 = att1
```

```
    self.att2 = att2
```

```
...
```

Il parametro "self" indica l'istanza e deve essere sempre indicato esplicitamente come primo in ogni metodo

## DEFINIZIONE DI METODI

- I metodi sono definiti come funzioni

```
def nomeMetodo (self, param1, param2, ...):
```

```
    istruzione 1
```

```
    istruzione 2
```

```
    ...
```

```
    return
```

Il parametro "self" indica l'istanza  
e deve essere sempre indicato  
esplicitamente come primo in ogni  
metodo



- Se si vuole lasciare un metodo vuoto, si usa la parola chiave "pass".  
Altrimenti ritorna errore

```
def nomeMetodo (self)
```

```
    pass
```

## DEFINIZIONE DI ISTANZE

nomeVar = nomeClasse() #chiama il costruttore vuoto

nomeVar = nomeClasse(att1,att2, ...) #chiama il costruttore definito nella funzione `__init__`

- Attenzione: l'istanza della classe è sempre passata automaticamente come primo parametro in ogni chiamata di funzione
- È possibile definire attributi di istanza semplicemente definendoli mediante dot notation

nomeIstanza.nome\_Nuovo\_Attributo= valore\_nuovo\_attributo

- Chiamata di metodi:

nomeIstanza.nomeMetodo()

nomeIstanza.nomeMetodo(param1, param2,..) automaticamente

La chiamata del metodo passa l'istanza come primo parametro

automaticamente

# SETTER, GETTER E DELETER

- Per definire setter, getter e delete si utilizza il decorator

## @property

- Esempio

**class Paziente:**

```
def __init__(self, ID, age, smoke):  
    self.patID = ID  
    self.age = age  
    self._smoke = smoke
```

**Proprietà  
privata**

@property

**Getter**

```
def smoke(self):  
    if self._smoke==0:  
        return ('Not smoker')  
    else:  
        return ('Smoker')
```

@smoke.setter     **Setter**

```
def smoke(self,newSmoke):  
    if newSmoke == 'Smoker':  
        self._smoke=1  
    elif newSmoke == 'Not smoker':  
        self._smoke=0  
    else:  
        print('Inserisci Smoker/Not smoker')
```

@smoke.deleter

**Deleter**  
def smoke(self):  
 print('Smoke status deletd')  
 self.\_smoke = None

**L'accesso alla  
proprietà sembra  
diretto ma in realtà  
è controllato**



```
pat=Paziente('GAL-11',70,1)  
print(pat.smoke)  
pat.smoke = 'Not smoker'  
del pat.smoke
```

# SQLITE E PYTHON

- SQLITE3 è disponibile in Python come modulo
- Per utilizzare le funzionalità di sqlite3 è sufficiente importare il modulo

```
import sqlite3
```

- La connessione ad un sqlite DB si effettua mediante la creazione di un oggetto “connettore”

```
conn = sqlite3.connect("nomeDB")
```

- Per connettersi ad un database esistente si passa la stringa contenente il path “pathfile/nomeDB”
  - Per creare un nuovo database, si dichiara il nome del nuovo DB
  - Per lavorare con un database runtime (si crea ogni volta che viene lanciato il programma e si cancella alla fine dell’esecuzione) si passa la stringa “:memory:”
- Per chiudere la connessione si usa il metodo **close()** → conn.close()

## ESEGUIRE ISTRUZIONI SQL

- Creare un oggetto cursore mediante il metodo `cursor()` della classe `connector`

```
c = conn.cursor()
```

- Il metodo `execute()` della classe `cursor` permette di eseguire le query SQL

```
c.execute("CREATE TABLE Paziente (PatID integer primary key autoincrement,  
Nome text, Cognome Text,...")
```

```
c.execute("INSERT INTO Paziente(Nome,Cognome,...) VALUES ('Carlo',  
'Rossi',...)")
```

```
c.execute("SELECT * FROM Paziente")
```

## ESEGUIRE ISTRUZIONI SQL

- Le istruzioni di creazione, inserimento valori, update, delete, etc richiedono un esplicito “commit” per essere eseguite, mediante il metodo `commit()` della classe `connector`

**`conn.commit()`**

- I metodi di fetch della classe `cursor` consentono di ottenere i risultati delle query come liste bidimensionali (Righe: tuple, Colonne: attributi)
  - **`c.fetchone()`** – ritorna solo la prima riga
  - **`c.fetchmany(N)`** – ritorna le prime N righe
  - **`c.fetchall()`** – ritorna tutte le righe
- L'accesso alle liste si effettua mediante indicizzazione
  - `nomeLista[i]` – ritorna la i-esima tuple
  - `nomeLista[i][j]` – ritorna l'attributo j-esimo della i-esima tupla

# PARAMETRI NELLE QUERY SQL

```
c.execute("SELECT * FROM Patient WHERE Pat_ID='GAL-12'")
```



- Metodo 1: utilizzo di “?” e tuple

```
c.execute("SELECT * FROM Patient WHERE Pat_ID=?", ('GAL-12', ))
```



Definizione della tuple →  
necessita la “,” anche se è  
costituita da un solo valore

- Metodo 2: utilizzo di dict

```
c.execute("SELECT * FROM Patient WHERE Pat_ID=:ID", {'ID': 'GAL-12'})
```



Definizione del dict  
(dizionario)