Corso Pratico Python

[Modulo 1- 2026]

2. Linguaggio Python
2.1. Variabili
2.1.1. Visibilità, Interprete
2.1.2. Classi, Moduli
2.1.3. Liste, Array, Dizionari, Tuple.
2.1.4. Argomenti, Kwargs

Richiedente: Dott. Roberto Felici [CNR-ISM] • Insegnante: Dott. Scigè J. Liu [INAF-IAPS]

2.1. Variabili

Regole del Nome (Syntax)

- **Inizio:** Deve iniziare con una **lettera** o un trattino basso _.
- **Caratteri:** Può contenere lettere, numeri e underscore (A-z, 0-9, _).
- **Case-sensitive:** Nome e nome sono due variabili diverse.
- **Vietato:** Non puoi usare parole riservate (if, else, class, def, ecc.) o iniziare con un numero.

Convenzioni di Stile (PEP 8)

Sebbene Python non obblighi, la comunità segue regole per la leggibilità:

- **Snake Case:** Per variabili e funzioni → tutto minuscolo con underscore (es. valore_totale).
- **Camel Case:** Si usa solo per le **Classi** (es. UtentePremium).
- **Costanti:** Si scrivono in tutto MAIUSCOLO (es. PI_GRECO = 3.14).

2.1.1. Visibilità, Interprete

Visibilità (Scope) e la regola LEGB

L'interprete cerca le variabili seguendo un ordine gerarchico chiamato **LEGB**.

Quando si richiama una variabile, l'interprete controlla in questi quattro livelli:

- **L (Local):** Dentro la funzione corrente (o decoratore). Se la trovi qui, finisce la ricerca.
- **E (Enclosing):** Nelle funzioni esterne (importante per decoratori e funzioni annidate).
- **G (Global):** Al livello più alto dello script o del modulo (il file .py).
- **B (Built-in):** Funzioni predefinite di Python (come print, len, range).

Convenzioni Lettura vs Scrittura

- ***Lettura:*** *In lettora di una variabile, Python risale la gerarchia LEGB, o segnala eccezione.*
- ***Scrittura:*** *In assegnazione, Python crea (o sovrascrive) la variabile* ***sempre nel livello Local*** *[SHADOWING]*

Convenzioni `global` e `nonlocal` – per variabili di un livello superiore:

- **`global`**: accesso a una variabile definita a livello di modulo (G) dentro una funzione.
- **`nonlocal`**: Permette di modificare variabili definite nella funzione esterna (E).

[Fondamentale nei decoratori!]

2.1.1. Visibilità, Interprete

Variabili Globali

→**accedibili ovunque non «ombreggiate»**

→perdurano per tutta l'esecuzione

→Built-in: globals() --> dict()

```
In [15]: globals()
Out[15]:
{'__name__': '__main__',
 '__doc__': '\nCreated on Sun Feb 22 23:34:21 2026\n\n@author: sc
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': {'__name__': 'builtins',
  '__doc__': "Built-in functions, types, exceptions, and other ob
in'\nidentifiers of Python; for example, builtins.len is\nthe ful
normally accessed explicitly by most\napplications, but can be us
built-in value, but in\nwhich the built-in of that name is also n
  '__package__': '',
  '__loader__': _frozen_importlib.BuiltinImporter,
  '__spec__': ModuleSpec(name='builtins', loader=<class '_frozen_
  '__build_class__': <function __build_class__>,
  '__import__': <function __import__(name, globals=None, locals=N
  'abs': <function abs(x, /)>,
  'all': <function all(iterable, /)>,
  'any': <function any(iterable, /)>,
  'ascii': <function ascii(obj, /)>,
  'bin': <function bin(number, /)>,
  'breakpoint': <function breakpoint(*args, **kws)>,
  'callable': <function callable(obj, /)>,
```

2.1.2. Classi, Moduli [e packages]

Variabili di Modulo
definite al livello di un file .py sono chiamate **variabili di modulo**.
In Python significa «globale all'interno di questo file».

- **Visibilità:** Accessibili da ogni funzione o classe all'interno dello stesso file.
- **Ciclo di vita:** Esistono finché il modulo è caricato in memoria.

```python
# modulo_meteo.py
UNITA_MISURA = "Celsius"  # Variabile di modulo

def cambia_unita():
    global UNITA_MISURA
    UNITA_MISURA = "Fahrenheit"
```

Variabili di Classe (Statiche)
definite all'interno di una class, fuori dai metodi, condivise da **tutte** le istanze di quella classe. Ogni cambiamento è visibile da ogni oggetto creato da essa.

- **Accesso:** Si usa NomeClasse.nome_variabile.
- **Uso tipico:** Contatori di istanze, costanti specifiche per quel tipo di oggetto.

Non esiste ombreggiamento a livello di classe, ma ombreggiamento a livello di oggetto.

```python
class StampaFormato:
    # Variabile di CLASSE (Statica)
    formato = "Semplice"
    def __init__(self,nuovoFormato=None):
        if nuovoFormato is not None:
            self.offuscaFormato(nuovoFormato)

    def offuscaFormato(self, nuovoFormato):
        self.formato = nuovoFormato

    @staticmethod
    def offuscaFormati1(nuovoFormato):
        StampaFormato.formato = nuovoFormato
    @classmethod
    def offuscaFormati2(cls,nuovoFormato):
        cls.formato = nuovoFormato

    def __str__(self):
        return str(f"formato:{self.formato}")
```

2.1.2. Classi, Moduli [e packages]

Variabili di Modulo
definite al livello di un file .py sono chiamate **variabili di modulo**.
In Python significa «globale all'interno di questo file».

- **Visibilità:** Accessibili da ogni funzione o classe all'interno dello stesso file.
- **Ciclo di vita:** Esistono finché il modulo è caricato in memoria.

```python
# modulo_meteo.py
UNITA_MISURA = "Celsius"  # Variabile di modulo

def cambia_unita():
    global UNITA_MISURA
    UNITA_MISURA = "Fahrenheit"
```

Variabili di Classe (Statiche)
defi
clas
- A
- U

Nor

```python
class StampaFormato:
    # Variabile di CLASSE (Statica)
    formato = "Semplice"
    def __init__(self,nuovoFormato=None):
```

```python
if __name__=='__main__':
    print('-'*40)
    sf1=StampaFormato()
    print(sf1)
    sf2=StampaFormato("Avanzato")
    print(sf2)
    print('-'*40)
    StampaFormato.offuscaFormati1("Intermedio1")
    print(sf1)
    print(sf2)
    print('-'*40)
    sf2.offuscaFormati2("Intermedio2")
    print(sf1)
    print(sf2)
    print('-'*40)
```

```
In [34]: %runfile C:/P_Python/prova/StampaFormato.py --wdir
----------------------------------------
formato:Semplice
formato:Avanzato
----------------------------------------
formato:Intermedio1
formato:Avanzato
----------------------------------------
formato:Intermedio2
formato:Avanzato
----------------------------------------

In [35]:
```

2.1.2. Classi, Moduli [e packages]

Variabili di Modulo
definite al livello di un file .py sono chiamate **variabili di modulo**.
In Python significa «globale all'interno di questo file».

- **Visibilità:** Accessibili da ogni funzione o classe all'interno dello stesso file.
- **Ciclo di vita:** Esistono finché il modulo è caricato in memoria.

Variabili di Classe (Statiche)
definite all'interno di una class, fuori dai metodi, condivise da **tutte** le istanze di quella classe. Ogni cambiamento è visibile da ogni oggetto creato da essa.

- **Accesso:** Si usa NomeClasse.nome_variabile.
- **Uso tipico:** Contatori di istanze, costanti specifiche per quel tipo di oggetto.

Non esiste ombreggiamento a livello di classe, ma ombreggiamento a livello di oggetto.

Variabili di Packages (in __init__.py)
I package [cartelle] con moduli, trai quali il file __init__.py : definisce variabili.

- **Uso:** Si usano per esporre costanti comuni a tutto il package o per definire i metadati.
- **Esempio:** in package1/__init__.py : DEBUG_MODE = True

```
import mio_package;
mio_package.DEBUG_MODE.
```

2.1.2. Classi, Moduli [e packages]

Variabili di Modulo
definite al livello di un file .py sono chiamate **variabili di modulo**.
In Python significa «globale all'interno di questo file».

- **Visibilità:** Accessibili da ogni funzione o classe all'interno dello stesso file.
- **Ciclo di vita:** Esistono finché il modulo è caricato in memoria.

Variabili di Classe (Statiche)
definite all'interno di una class, fuori dai metodi, condivise da **tutte** le istanze di quella classe. Ogni cambiamento è visibile da ogni oggetto creato da essa.

- **Accesso:** Si usa NomeClasse.nome_variabile.
- **Uso tipico:** Contatori di istanze, costanti specifiche per quel tipo di oggetto.

Non esiste ombreggiamento a livello di classe, ma ombreggiamento a livello di oggetto.

Variabili di Packages (in __init__.py)
I package [cartelle] con moduli, trai quali il file __init__.py : definisce variabili.
•**Uso:** Si usano per esporre costanti comuni a tutto il package o per definire i metadati.
•**Esempio:** in package1/__init__.py : DEBUG_MODE = True
import mio_package;
mio_package.DEBUG_MODE.

EFFETTI DEVASTANTI
Sovrascrittura di :

- **Costanti ASSOLUTE**
  - **np.pi = 3.1 [!AAAH!]**

```
import importlib
import numpy as np
importlib.reload(np)

print(np.pi) # Torna al valore originale
```

2.1.2. Classi, Moduli

Classi + ereditarietà

```python
class Cane:
    # Il metodo __init__ è il "costruttore"
    # Serve a inizializzare le caratteristiche dell'oggetto
    def __init__(self, nome, razza):
        self.nome = nome      # Attributo di istanza
        self.razza = razza    # Attributo di istanza

    # Un metodo: definisce cosa può FARE l'oggetto
    def abbaia(self):
        return f"{self.nome} dice: Bau Bau!"

    def descriviti(self):
        return f"Sono un {self.razza} e mi chiamo {self.nome}."

# --- Utilizzo della classe ---

# Creiamo un "oggetto" (istanza) della classe Cane
mio_cane = Cane(nome="Buddy", razza="Golden Retriever")

print(mio_cane.descriviti()) # Output: Sono un Golden Retriever e mi chiamo Buddy
print(mio_cane.abbaia())     # Output: Buddy dice: Bau Bau!
```

```python
# Poliziotto eredita da Cane
class CanePoliziotto(Cane):
    def arresta_ladro(self):
        return f"{self.nome} ha fermato il sospetto!"

rex = CanePoliziotto("Rex", "Pastore Tedesco")
print(rex.abbaia())          # Funziona perché ereditato da Cane
print(rex.arresta_ladro())   # Metodo specifico di CanePoliziotto
```

2.1.2. Classi, Moduli

Moduli + MONKEY PATCHING

```python
# file: utility.py
def somma(a, b):
    return a + b

def sottrai(a, b):
    return a - b
```

```python
# file: mario.py
def hello():
    print("Ciao da Mario!")
```

```python
import mario

# 1. Definiamo la nuova funzione "sostituta"
def ciao():
    print("Questa è la nuova funzione ciao!")

# 2. SOVRASCRITTURA (Monkey Patching)
# Assegniamo l'oggetto funzione 'ciao' all'attributo 'hello' del modulo
mario.hello = ciao

# 3. Testiamo il risultato
mario.hello()
# Output: Questa è la nuova funzione ciao!
```

COSA SUCCEDE
IN CASO DI FUNZIONI CON ARGOMENTI?

Corso Pratico Python

[Modulo 1- 2026]

2. Linguaggio Python
2.1. Variabili
2.1.1. Visibilità, Interprete
2.1.2. Classi, Moduli
2.1.3. Liste, Array, Dizionari, Tuple.
2.1.4. Argomenti, Kwargs

Richiedente: Dott. Roberto Felici [CNR-ISM] • Insegnante: Dott. Scigè J. Liu [INAF-IAPS]

2.1.3. Liste, Array, Dizionari, Tuple.

1. Liste (List) → Aggregato dinamico, anche eterogeneo

Ordinata: Gli elementi mantengono l'ordine di inserimento.

Mutabile: Puoi cambiare, aggiungere o rimuovere elementi.

Versatile: Può contenere tipi di dati diversi.

```
>>> ciccio=[]
>>> ciccio+="asdfoi"
>>> ciccio
['a', 's', 'd', 'f', 'o', 'i']
>>> ciccio+=["asdfoi"]
>>> ciccio
['a', 's', 'd', 'f', 'o', 'i', 'asdfoi']
```

2. Array → Aggregato omogeneo, ottimizzato per operazioni di massa

Come altri linguaggi (come C o Java), «dimensione plastica» e tipo fisso.

Omogeneo: Tutti gli elementi devono essere dello stesso tipo.

Efficienza: Sono molto più veloci delle liste per calcoli matematici.

```
import array
# Creiamo un array di numeri interi ('i' sta per signed integer)
numeri = array.array('i', [10, 20, 30, 40])
# 1. Aggiungere un elemento alla fine
numeri.append(50)
# 2. Modificare un elemento esistente
numeri[1] = 25  # Cambia il 20 in 25
# 3. Stampare l'array
print(numeri)
# Output: array('i', [10, 25, 30, 40, 50])
```

2.1.3. Liste, Array, Dizionari, Tuple.

3. Dizionari (Dictionary / Map) → Aggregato di associazioni CHIAVE-VALORE

Coppie Chiave-Valore: Ogni elemento è identificato da una chiave univoca.

Accesso rapido: Trovi un valore istantaneamente se conosci la chiave.

Mutabile: Puoi aggiornare i valori o aggiungere nuove coppie.

```python
# 1. Creazione del dizionario
giocatore = { "nickname": "ProGamer99", "livello": 15,
 "punti_vita": 100, "inventario": ["spada", "scudo"] }
# 2. Accedere a un valore (usando la chiave)
print(giocatore["nickname"])  # Output: ProGamer99
# 3. Aggiungere o aggiornare un dato
giocatore["livello"] = 16       # Aggiorna il valore
giocatore["online"] = True      # Aggiunge chiave-valore
# 4. Rimuovere un dato
del giocatore["punti_vita"]
# 5. Stampare il dizionario aggiornato
print(giocatore)
```

4. Tuple → Aggregato immutabile → È una lista "congelata".

Immutabile: Una volta creata, non puoi modificarla, aggiungervi o togliervi nulla.

Sicurezza: Si usa per proteggere i dati che non devono cambiare durante l'esecuzione del programma.

Performance: Leggermente più veloce di una lista proprio perché è statica

Sintassi: (10, 20, 30)

PRO : è sintatticamente il sostituto di restituzione/assegnazione immediato di aggregati, visibile a tempo di compilazione

```python
# --- 4. TUPLA (Ordinata e Immutabile) ---
# Ideale per dati fissi che non devono essere protetti da modifiche
punto_gps = (45.4642, 9.1900) # Latitudine e Longitudine di Milano
# punto_gps[0] = 46.0 <-- Questo darebbe ERRORE (le tuple non si cambiano!)
# Unpacking della tupla
lat, lon = punto_gps
print(f"Tupla GPS: {punto_gps} -> Lat: {lat}, Lon: {lon}")
```