Introduction to Python programming

Object-Oriented Programming, lambda functions and regex

- Armando Camerlingo

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

Introduzione alla Programmazione in Python

Armando Camerlingo

18/06/2025 - Lezione 5

Programmazione ad oggetti (OOP)

- Python è un linguaggio di programmazione ad **oggetti**, Object-Oriented Programming (OOP);
- Un oggetto è un'entità a cui sono legate proprietà e funzioni, ottimi per modellare anche oggetti reali;
- OOP si basa su 4 concetti base:
  - **Incapsulazione**: attributi e metodi in unità coese, per favorire integrità e modularità tra diversi oggetti;
  - **Ereditarietà**: relazioni tra classi e sottoclassi per ridurre verbosità e duplicazione del codice;
  - **Astrazione**: nascondere il codice e esporre solo le funzionalità dell'oggetto, favorendo la facilità di uso;
  - **Polimorfismo**: oggetti di tipo differente trattati come istanze di uno stesso tipo base, con stessi metodi e attributi;

Classi

ITINERIS

- Le classi sono gli oggetti base di Python;
- Permette di creare strutture dati definite dall'utente;
- Basata sulla definizione di metodi: azioni che possono essere effettuate sui dati contenuti nella classa;
- L'istanza è un oggetto costruito a partire dalla classe contenente dati;

```
class <Nome classe>:
  def <metodo 1>:
    ...
  def <metodo 2>:
    ...
```

Classi: metodo __init__()

ITINERIS

In questo metodo vengono definite tutte le proprietà essenziali di un oggetto della classe. Imposta lo stato iniziale dell'istanza.

```
class Cane:
  def __init__(self, nome, eta):
    self.nome = nome
    self.eta = eta
```

Il primo parametro è sempre `self`, una variabile a cui viene passata l'istanza per definire i nuovi attributi.

Classi: metodo __init__()

ITINERIS

```
class Cane:
  def __init__(self, nome, eta):
    self.nome = nome
    self.eta = eta
```

`nome` e `eta` sono chiamati attributi d'istanza. Hanno valori specifici per ogni istanza. Si possono definire variabili anche a livello di classe (attributi di classe), specificando il loro valore al di fuori di init() e altri metodi:

```
class Cane:
  specie = 'Canis Familiaris'
  def __init__(self, nome, eta):
    self.nome = nome
    self.eta = eta
```

Classi: istanze

ITINERIS

Un’istanza viene creata chiamando semplicemente il nome della class seguito da delle parentesi tonde e valori per gli attributi d’istanza:

```
Cane('nome', 'eta')
```

Se si ripete l’istruzione, verrà creata un’altra istanza diversa (notare la stringa diversa nell’output). Ciò è evidente anche tramite le seguenti istruzioni:

```
a = Cane('nome', 'eta')
b = Cane('nome', 'eta')
a==b
```

ed è falso

Classi: istanze

ITINERIS

Creiamo due istanze con valori diversi:

```
Pongo = Cane('Pongo', 5)
Peggy = Cane('Peggy', 3)
```

Si può accedere alle variabili di istanza:

```
Pongo.nome
Pongo.eta
Peggy.nome
Peggy.eta
```

Allo stesso modo si accede alle variabili di classe:

```
Pongo.specie
```

Classi: istanze

In modo simile si possono modificare gli attributi di un’istanza:

```
Pongo.eta = 10
Pongo.eta
```

```
Pongo.specie = 'gatto'
Pongo.specie
```

Classi: metodi

ITINERIS

Sono funzioni definite all'interno della classe che possono essere chiamate su un'istanza di essa.

```
class Cane:
  specie = 'Canis Familiaris' # attributo di classe

  def __init__(self, nome, eta):
    self.nome = nome # attributo d'istanza
    self.eta = eta # attributo d'istanza

  def descrizione(self):
    return f'{self.nome} ha {self.eta} anni'

  def verso(self, suono):
    return f'{self.nome} dice {suono}'
```

Classi: metodi

ITINERIS

Il primo metodo può essere chiamato senza parametri:

```
Pongo.descrizione()
```

Il secondo metodo ha bisogno di un’altra variabile:

```
Pongo.verso('bau')
```

Il primo metodo può essere utile per dare delle info sulla classe, ma non è la scelta migliore. Possiamo usare i cosiddetti metodi **speciali**, chiamati anche **dunder** da **d**ouble **under**score.

Classi: metodi dunder

Se si usa `print()` dando come input una classe, otteniamo un messaggio di info sull'oggetto.

Possiamo definire un metodo nella classe per modificare questo comportamento

```
Class Cane(...):
    ...
    def __str__(self):
        return f'{self.nome} ha {self.eta} anni'
```

Se adesso rifacciamo il print:

```
Pongo = Cane('Pongo', 5)
print(Pongo)
```

I metodi come __init__ e __str__ (caratterizzati dal doppio underscore) hanno comportamenti speciali e sono fondamentali per usare al massimo le classi.

Esercizio

Creare una classe “macchina”, con più di 2 attributi di istanza, un metodo __str__ che dà una descrizione della macchina e un metodo che prende in input il prezzo in euro e lo restituisce in dollari

Soluzione

```
class Macchina:
  def __init__(self, marca, modello, prezzo):
    self.marca = marca
    self.modello = modello
    self.prezzo = prezzo

  def __str__(self):
    return f"La macchina è una {self.modello} della {self.marca}
e costa {self.prezzo} euro"

  def conver(self):
    return f"Il prezzo in dollari è {self.prezzo * 1.13}"
```

```
Polo = Macchina('Volkswagen', 'Polo', 25000)
print(Polo)

print(Polo.conver())
```

Classi: ereditarietà

ITINERIS

Una nuova classe può “ereditare” gli attributi e i metodi di un’altra classe. Prendono il nome rispettivamente di classe genitore e classe figlio

```
class Genitore:
  colore_capelli = 'marrone'

class Figlio(Genitore):
  pass
```

```
Figlio.colore_capelli
```

Classi: ereditarietà

La classe figlio può anche sovrascrivere gli attributi ereditati:

```
class Genitore:
  colore_capelli = 'marrone'

class Figlio(Genitore):
  colore_capelli = 'celeste'
```

```
Figlio.colore_capelli
```

La classe figlio restituirà il valore impostato nella propria definizione

Classi: ereditarietà

ITINERIS

La classe figlio può anche estendere gli attributi ereditati:

```
class Genitore:
  def __init__(self):
    self.speaks = ['Inglese']

class Figlio(Genitore):
  def __init__(self, extend=False):
    super().__init__()
    if extend:
      self.speaks.append('Tedesco')
```

```
Figlio.speaks(True)
```

Classi: Polimorfismo

ITINERIS

Si possono specificare metodi con lo stesso nome ma con azioni diverse a seconda della classe:

```
class Animale:
  def verso(self):
      return 'verso generico'

class Cane(Animale):
  def verso(self):
      return 'bau'

class Gatto(Animale):
  def verso(self):
      return 'miao'
```

```
animali =[Cane(), Gatto(), Animale()]
for animale in animali:
   print(animale.verso())
```

Classi: Esempio

ITINERIS

Modelliamo un parco per cani in cui ci sono più cani di diverse razze:

```
class Cane:
  species = 'Canis familiaris'

  def __init__(self, nome, eta, razza):
    self.nome = nome
    self.eta = eta
    self.razza = razza

  def __str__(self):
    return f"{self.nome} ha {self.eta} anni"

  def verso(self, verso):
    return f"{self.nome} dice {verso}"
```

Classi: Esempio

ITINERIS

Creiamo delle istanze:

```
miles = Cane("Miles", 4, "Jack Russell Terrier")

buddy = Cane("Buddy", 9, "Dachsund")

jack = Cane("Jack", 3, "Bulldog")

jim = Cane("Jim", 5, "Bulldog")
```

Ipotizziamo che ogni cane faccia un verso leggermente diverso:

```
buddy.verso("Bau")

jim.verso("Woof")

jack.verso("Woof")
```

Classi Genitori e Classi Figlie

ITINERIS

E’ ripetitivo dover specificare la razza e anche dare come variabile di output il suono ogni volta. Possiamo risolvere creando più classi figlio:

```
class Cane:
  species = 'Canis familiaris'

  def __init__(self, nome, eta):
    self.nome = nome
    self.eta = eta

  def __str__(self):
    return f"{self.nome} ha {self.eta} anni"

  def verso(self, verso):
    return f"{self.nome} dice {verso}"
```

```
class JackRussellTerrier(Cane):
    pass

class Dachshund(Cane):
    pass

class Bulldog(Cane):
    pass
```

```
miles = JackRussellTerrier("Miles", 4)

buddy = Dachsund("Buddy", 9)

jack = Bulldog("Jack", 3)

jim = Bulldog("Jim", 5)
```

Classi Genitori e Classi Figlie

Tutte le istanze delle classi figlio ereditano gli attributi e i metodi della classe genitore :

```
miles.species
```

```
print(jack)
```

```
buddy.nome
```

```
jim.verso("Bau")
```

Classi: Esempio

Possiamo verificare la classe di appartenenza con la funzione `type()`:

```
type(miles)
```

Possiamo anche verificare l'appartenenza ad una classe (genitore o figlio) con `isinstance()`:

```
isinstance(miles, Cane)
```

`isinstance()` accetta due valori di input, un oggetto e una classe e restituisce un valore booleano

Tutti gli oggetti creati sono istanze della classe genitore Cane(), ma ognuna apparterrà alla propria sottoclasse

Classi: Override

ITINERIS

Proviamo a implementare differentemente il metodo `verso()`. Vogliamo dare un valore di default di verso per ogni razza. Ciò può essere effettuato con un **override** (sovrascrittura) nelle classi figlio

```
...
class JackRussellTerrier(Cane):
   def verso(self, suono="Arf"):
      return f"{self.nome} dice {suono}"
```

Se si chiama il metodo `verso()` della classe `JackRussellTerrier()`, verrà eseguito il codice della classe figlio:

```
miles = JackRussellTerrier("Miles", 4)
miles.verso()
```

Attenzione: se vengono effettuati cambi sul metodo `verso()` della classe genitori **NON** verranno rispecchiati nelle classi figlio.

Classi: super()

ITINERIS

Vogliamo che il metodo nelle classi figlie, anche se modificato, erediti le modifiche fatte sul metodo nella classe genitore. Si può utilizzare `super()`:

```
...
class JackRussellTerrier(Cane):
   def verso(self, suono="Arf"):
      return super().verso(suono)
```

Quando si chiama `super()`, Python cerca il metodo `verso()` nella classe genitore e lo esegue con la variabile suono

```
miles = JackRussellTerrier("Miles", 4)
miles.verso()
```

In questo caso, tutti i cambi sulla classe genitore verranno riportati nelle classi figlio.

Esercizio 1

ITINERIS

Creare una classe GoldenRetriever() che eredita dalla classe Cane. GoldenRetriever deve avere come verso “Bark”.

Esercizio

ITINERIS

Creare una classe GoldenRetriever() che eredita dalla classe Cane. GoldenRetriever deve avere come verso “Bark”.

Soluzione

```
class GoldenRetriever(Cane):
   def verso(self, suono="Bark"):
      return f"{self.nome} dice {suono}"
```

```
class GoldenRetriever(Cane):
   def verso(self, suono="Bark"):
      return super().verso(suono)
```

Funzioni Lambda

ITINERIS

Le funzioni lambda sono uno strumento di Python per definire delle funzioni aventi le seguenti caratteristiche:

- Semplici, poche stringhe di codice;
- Anonime, ovvero non hanno un nome definito;
- Possono essere usate in combinazione con altre funzioni built-in di Python

Funzioni Lambda

ITINERIS

Consideriamo una funzione semplice:

```
def somma(x, y):
    return x + y
```

Usando le funzioni lambda si può riscrivere come :

```
lambda x, y: x + y
```

Funzioni Lambda: Esempi

ITINERIS

Salviamo in una variabile una funzione lambda che valuta se un numero è pari:

```
pari = lambda x: x%2 == 0
```

Si ottiene una funzione che prende un numero come input e restituisce un valore booleano:

```
pari(5)
```

```
pari(2)
```

Esercizio

ITINERIS

Dato un numero, trovare il suo quadrato

Esercizio

ITINERIS

Dato un numero, trovare il suo quadrato

Soluzione

```
quadrato = lambda x: x**2
y = quadrato(2)
print(y)
```

Funzioni Lambda: filter

ITINERIS

Le funzioni lambda sono spesso usate nella programmazione funzionale in combinazione con altre funzioni. Vediamo alcuni esempi:

```
numeri = list(range(10))
print(numeri)
```

```
pari = filter(lambda x: x%2 == 0, numeri)
print(list(pari))
```

Esercizio

Scrivere una funzione lambda che trova i numeri divisibili per 19 o per 13 in una lista e li restituisce in un’altra lista. Usare questa lista: [19, 65, 57, 39, 152, 639, 121, 44, 90, 190]

Esercizio

Scrivere una funzione lambda che trova i numeri divisibili per 19 o per 13 in una lista e li restituisce in una lista. Usare questa lista: [19, 65, 57, 39, 152, 639, 121, 44, 90, 190]

Soluzione

```
numeri =  [19, 65, 57, 39, 152, 639, 121, 44, 90, 190]
print("lista originale")
print(numeri)
div = lambda x: (x%19 == 0 or x%13 == 0)
nuova_lista = list(filter(div, numeri))
print("Numeri divisibili per 19 o per 13")
print(nuova_lista)
```

Funzioni Lambda: map e sorted

ITINERIS

La funzione `map()` applica la funzione lambda a tutti gli elementi della lista

```
quadrati = map(lambda x: x**2, numeri)
print(list(quadrati))
```

La funzione `sorted()` ordina una lista dopo aver applicato la funzione lambda:

```
numeri = [1, 10, -1, 3, -10, 5]
numeri_ordinati = sorted(numeri, key=lambda x:
abs(x))
print(numeri_ordinati)
```

Funzioni Lambda: zip e reduce

ITINERIS

La funzione `zip()` unisce più liste in una sola e insieme a `map()` si possono applicare funzioni lambda:

```
lista_1 = [1, 2, 3]
lista_2 = [4, 5, 6]
lista = list(map(lambda x: x[0] * x[1], zip(lista_1,
lista_2)))
print(lista)
```

La funzione `reduce()` applica una funzione lambda in modo cumulativo:

```
from functools import reduce
numeri = [1, 10, -1, 3, -10, 5]
#numeri = []
somma = reduce(lambda x, y: x+y, numero, 0)
print(somma)
```

Esercizio

Scrivere una funzione lambda che calcola il prodotto degli elementi di una lista. Usare la lista [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] come test

Esercizio

Scrivere una funzione lambda che calcola il prodotto degli elementi di una lista. Usare la lista [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] come test

Soluzione

```
numeri = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
prodotto = reduce(lambda x,y: x*y, numeri, 1)
print(prodotto)
```

Funzioni Lambda: esempio 1

ITINERIS

Supponiamo di avere un negozio di frutta e di voler sapere il ricavo totale.

```
dati_vendita = [
    {'frutto': 'pesca', 'prezzo': 1.41, 'quantita': 3},
    {'frutto': 'pera', 'prezzo': 1.21, 'quantita': 2},
    {'frutto': 'mango', 'prezzo': 0.56, 'quantita': 3},
]
```

Trasformiamo i dati in modo da ottenere un dizionario che contiene il ricavo per ogni frutto

```
dati_trasformati = list(map(
    lambda entrata: {**entrata, 'vendite_totali':
round(entrata['prezzo']*entrata['quantita'],2)},
    dati_vendita
))

for entrata in dati_trasformati:
  print(entrata)
```

Funzioni Lambda: esempio 2

ITINERIS

Scrivere una funzione lambda che restituisce gli elementi della serie di Fibonacci fino al n-esimo. Suggerimento: usare 2 funzioni lambda e reduce()

```
from functools import reduce
somma = lambda x, _: x + [x[-1] + x[-2]]
red_2 = reduce(somma, [0,1,2], [0,1])
print(red_2)
## Noi vogliamo che però restituisca la lista al variare di n, e per n=0,
## n=1, n=2 restituisca [0,1]
serie_fib = lambda n: reduce(somma, range(n-2), [0,1])
print(serie_fib(2))
print(serie_fib(5))
```

Funzioni Lambda: errori comuni

ITINERIS

- Usare le funzioni lambda per funzioni troppo complesse. Il codice diventa difficile da leggere e da debuggare;
- Bisogna fare attenzione alla sintassi, può succedere di dimenticare parti dell'istruzione (come l'input);
- Non considerare dei casi limiti, p.e. lo 0 nel caso delle divisioni. Si può risolvere aggiungendo un if alla funzione lambda;

```
divisione = lambda x, y: x / y #errore per y = 0!!

divisione_corretta = lambda x, y: x / y if y != 0 else "non definita"
```

- Bisogna ricordare di trasformare l'output in una lista quando si usano le funzioni map, sorted, etc. (di base è un iterabile);

Manipolazione del testo

ITINERIS

- Spesso quando si lavora con variabili contenenti stringhe, che siano testi interi o liste di oggetti (p.e. email), si vuole filtrare per trovare le informazioni a cui siamo interessati;
- Si cercano le occorrenze di specifiche espressioni, ovvero combinazioni di caratteri;
- In molti linguaggi di programmazione, tra cui anche Python, si possono usare le regular expression (**regex**);
- Un’espressione regolare è una sequenza di caratteri che definisce uno schema di ricerca (pattern);

Regex: vantaggi

ITINERIS

- Flessibilità: Pattern complessi in una forma compatto e leggibile;
- Corrispondenza Pattern: si possono cercare ed estrarre facilmente parti di testo;
- Manipolazione del testo: si possono usare per trovare e sostituire e altre operazioni;
- Efficienza: usato correttamente, velocizza le operazioni;

Regex in Python

- In python le funzioni per lavorare con le espressioni regolare sono contenute nel modulo `re`

```
import re
```

```
?re
```

```
Type:        module
String form: <module 're' from '/usr/lib/python3.11/re/__init__.py'>
File:        /usr/lib/python3.11/re/__init__.py
Docstring:
Support for regular expressions (RE).

This module provides regular expression matching operations similar to
those found in Perl.  It supports both 8-bit and Unicode strings; both
the pattern and the strings being processed can contain null bytes and
characters outside the US ASCII range.

Regular expressions can contain both special and ordinary characters.
Most ordinary characters, like "A", "a", or "0", are the simplest
regular expressions; they simply match themselves.  You can
concatenate ordinary characters, so last matches the string 'last'.
```

Modulo re: search

ITINERIS

La funzione `search()` cerca un pattern all'interno del testo e restituisce `None` se non è presente oppure un oggetto contenente delle informazioni

```
testo = "Io amo la programmazione in Python"
pattern = 'Python'
match = re.search(pattern, testo)
print(match)
```

```
if match:
  print("Trovato: ", match.group())
  print("Posizione: ", match.span())
else:
  print("Non Trovato")
```

Modulo re: findall

La funzione `findall()` restituisce una lista contenente tutte le occorrenze

```
testo = "La rana in Spagna gracida in campagna"
x = re.findall("agn", testo)
print(x)
```

Restituisce una lista vuota se il pattern non è trovato

```
y = re.findall("Italia", testo)
print(y)
```

Modulo re: split

ITINERIS

La funzione `split()` restituisce delle parti della stringa che sono separate dal pattern

```
testo = "La rana in Spagna gracida in campagna"
x = re.split("in", testo)
print(x)
```

con il parametro maxsplit si può decidere in quanti parti dividere le frasi

```
testo = "La rana in Spagna gracida in campagna"
y = re.split("in", testo, 1)
print(y)
```

Modulo re: sub

ITINERIS

La funzione `sub()` sostituisce un pattern ad un altro

```
testo = "La rana in Spagna gracida in campagna"
x = re.sub("in", "nella",  testo)
print(x)
```

si può anche specificare quale delle occorrenze sostituire

```
testo = "La rana in Spagna gracida in campagna"
y = re.sub("in", "nella",  testo, 1)
print(y)
```

Esercizio

Sostituire punti, spazi e virgole con punti e virgola (;) nel seguente testo “Oggi, 17 giugno, studiamo regex.”

Esercizio

Sostituire punti, spazi e virgole con punti e virgola (;) nel seguente testo “Oggi, 17 giugno, studiamo regex.”

Soluzione

```
testo = "Oggi, 17 giugno, studiamo regex."
testo_2 = re.sub("[. ,]", ";", testo)
print(testo_2)
```

Regex: classi di caratteri

ITINERIS

Le classi di caratteri sono gruppi di caratteri che permettono di trovarli in un testo:

- \d: corrisponde alle cifre [0-9];
- \D: corrisponde a tutti caratteri che non sono cifre;
- \w: corrisponde a ogni carattere alfanumerico (sta per word);
- \W: corrisponde a ogni carattere non-alfanumerico;
- \s: corrisponde a caratteri di tipo “spazio bianco” (spazio, nuova linea, tabulazioni);
- \S: corrisponde a caratteri non di tipo “spazio bianco”;

Regex: classi di caratteri

ITINERIS

Se vogliamo dividere una frase nelle singole parole divise da uno spazio possiamo quindi scrivere:

```
testo = "La rana in Spagna gracida in campagna"
x = re.split("\s", testo)
print(x)
```

Oppure invece di dividerli per spazi, dividerli in righe diverse:

```
testo = "La rana in Spagna gracida in campagna"
x = re.sub("\s", "\n", testo)
print(x)
```

Regex: classi di caratteri

ITINERIS

Si possono anche usare combinazioni di classe per cercare pattern più complessi:

```
testo = "La rana in Spagna gracida in campagna"
x = re.search(r"\bS\w+", testo)
print("Trovato: ", x.group())
```

La r prima del pattern indica che la stringa deve essere interpretata cosi com’è (non considera il \).

Il + invece è un carattere speciale in regex, facente parte dei **metacaratteri**

Regex: metacaratteri

ITINERIS

Sono caratteri con uno speciale significato:

- \ : carattere di “escape”, serve a definire e/o escludere caratteri speciali;
- ^: cerca il pattern all’inizio della stringa;
- $: cerca il pattern alla fine della stringa;
- *: cerca 0 o più occorrenze del carattere precedente;
- +: cerca 1 o più occorrenze del carattere precedente;
- ?: cerca 0 o 1 occorrenza del carattere precedente
- |: or logico, alternative nel pattern;

Regex: metacaratteri

ITINERIS

in questo esempio si cerca l'indirizzo email:

```
text = "email@example : Ciao sono John"
pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+"
matches = re.findall(pattern, text)
print(matches)
```

in questo esempio si cerca il numero di telefono nella stringa

```
text = "Contact us at +123-456-7890 or
email@example.com."
pattern = r"\+\d{3}-\d{3}-\d{4}"
matches = re.findall(pattern, text)
print(matches)
```

Regex: quantificatori

ITINERIS

I quantificatori servono a definire quante occorrenze devono essere presenti:

- {n}: ci devono essere n occorrenze del carattere precedente;
- {n,}: ci devono essere n o più occorrenze del carattere precedente;
- {n,m}: ci devono essere tra n e m occorrenze del carattere precedente;

in questo esempio troviamo le parole con 3 o più vocali

```
testo = "Ci sono dei fiori nell'aiuola, piantati
con l'aiuto del giardiniere"
pattern = r"\b\w*[aeiou]{3,}\w*\b"
matches = re.findall(pattern, testo,
re.IGNORECASE)
print(matches)
```

Regex: gruppi

I gruppi sono sottoespressioni racchiusi da parentesi tonde. Utili quando si vogliono applicare quantificatori a caratteri multipli o creare pattern più complessi:

```
text = "Paolo Rossi: 20 anni, Lucia Bianchi: 25 anni"
pattern = r"(\w+\s\w+):\s(\d+)\sanni"
matches = re.findall(pattern, text)
print(matches)
```

Possono essere anche essere usati per recuperare multiple parti di un testo a cui puoi accedere separatamente utilizzando il metodo groups()

```
testo = "Data: 2025-18-06"
pattern =  r"(\d{4})-(\d{2})-(\d{2})"
match = re.search(pattern, testo)
if match:
    anno, mese, giorno = match.groups()
    print("anno:", anno)
    print("mese:", mese)
    print("giorno:", giorno)
```

Esercizio

Rimuovere gli spazi multipli, sostituendoli con uno spazio singolo, dalla stringa "   Programmazione   Python "

Esercizio

Rimuovere gli spazi multipli, sostituendoli con uno spazio singolo, dalla stringa "   Programmazione   Python "

Soluzione

```
testo = "   Programmazione    Python "
print("testo originale:", testo)
testo_2 = re.sub(' +', ' ', testo)
print("testo pulito:", testo_2)
```

Esercizio

Convertire la data 2025-06-18 dal formato YYYY-MM-DD al formato DD-MM-YYYY (18-06-2025) utilizzando tra le varie funzioni la funzione sub

Soluzione 1

```
testo = "2025-06-18"
pattern =  r"(\d{4})-(\d{2})-(\d{2})"
match = re.search(pattern, testo)
if match:
    anno, mese, giorno = match.groups()
    print("anno:", anno)
    print("mese:", mese)
    print("giorno:", giorno)

testo_2 = re.sub(r"(\d{4})", giorno, testo)
print(testo_2)
testo_2 = re.sub(r"-(\d{2})", '-' + mese, testo_2, 0)
print(testo_2)
testo_2 = re.sub(r"-(\d{2})$", '-' + anno, testo_2, 2)
print(testo_2)
```

Esercizio

Convertire la data 2025-06-18 dal formato YYYY-MM-DD al formato DD-MM-YYYY (18-06-2025) utilizzando tra le varie funzioni la funzione sub

Soluzione 2

```
testo = "2025-06-18"
pattern =  r"(\d{4})-(\d{2})-(\d{2})"
match = re.search(pattern, testo)
if match:
    anno, mese, giorno = match.groups()
    print("anno:", anno)
    print("mese:", mese)
    print("giorno:", giorno)

testo_2 = re.sub(r"(\d{4})", giorno, testo)
print(testo_2)
testo_2 = re.sub(r"-(\d{2})", '-' + mese, testo_2, 0)
print(testo_2)
testo_2 = re.sub(r"-(\d{2})$", '-' + anno, testo_2, 2)
print(testo_2)
```

THANKS!

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani