

FORMULARIO

- class NomeOggetto → class crea un contenitore di oggetti personale come un dizionario

per es:

```
class Libro:  
    def __init__(self):  
        self.titolo = t  
        self.numpag = n } attributi  
        self.autore = a }
```

creazione contenitore
ad oggetto di libro

creo un oggetto di tipo LIBRO!

Libro1 = Libro("La Pimpa", 50, "Giovio")

print(Libro1.titolo) → stampereà La Pimpa

se continuarsi con
Libro2 = Libro(.....)
posso!!

In alternativa posso scrivere:

Libro1 = Libro()

Libro1.titolo = "La Pimpa"

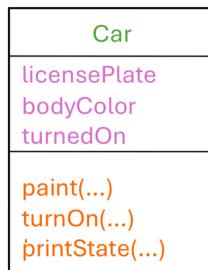
print(Libro1.titolo) → uscirà La Pimpa

se però scrivo:

print(Libro1.autore) → uscirà la a PREDEFINITA OR SU se non
aggiungo niente

Altro esempio:

```
class Car:  
    def __init__(self):  
        self.licensePlate = 0  
        self.bodyColor = ''  
        selfturnedOn = False  
  
    def paint(self, color):  
        self.bodyColor = color  
  
    def turn_on(self):  
        self.turned_on = True  
  
    def printCar(self):  
        print(f"Plate: {licensePlate}, color: {bodyColor}")
```



UTILIZZO DEI METODI!!

Cambiamo e modificano
lo stato dei nostri oggetti!

def NONE-METODO (self, VARIABILI):
 SVOLGIMENTO METODO

```

class Car:
    wheels = 4 # class attribute

    def __init__(self, licensePlate, bodyColor):
        self.licensePlate = licensePlate # instance attr.
        self.bodyColor = bodyColor # instance attr.

print(Car.wheels) # prints 4
c1 = Car('AB123CD', 'red') # print(c1.wheels) also prints 4
c2 = Car(...)
c2.wheels = 6 # separate instance variable created in c2
print(c1.wheels) # prints 4
Car.wheels = 6 # class attribute can be changed w/ class name

```

in questo momento
wheels sarà 4 per ogni oggetto

qui CREO UNA NUOVA VARIABILE
DI ISTANZA! → non la cambia
ma me crea una nuova

qui invece cambio quello sopra
della classe

le variabili scritte NAME o NAME_ o NAME-- sono uguali ma più
underscore metto e più la variabile è protetta

GETTER e SETTER

GETTER:

```

class Car:
    def __init__(self, licensePlate, bodyColor):
        self.licensePlate = licensePlate
        self.bodyColor = bodyColor
        selfturnedOn = False
        self._voltage = 12

    @property
    def voltage(self):
        return self._voltage

```

lo posso usare senza sapere quanto
è protetta la variabile e la
prendo comunque

SETTER:

```

class Car:
    def __init__(self, licensePlate, bodyColor):
        self.licensePlate = licensePlate
        self.bodyColor = bodyColor
        selfturnedOn = False
        self._voltage = 12

    @property
    def voltage(self):
        return self._voltage

    @voltage.setter
    def voltage(self, volts):
        print('Warning: this can cause problems')
        self._voltage = volts

```

nel setter posso scrivere delle
condizioni da applicare a variabili
protette con gli underscore

NB!! come scrivere
str(c1)

Per produrre una rappresentazione stampabile
in stringa di un oggetto posso utilizzare:

-- str__(self)

```

class Car:
    ...
    def __str__(self):
        return f'{self.licensePlate}\n{self.bodyColor}\n{self.turnedOn}'

c1 = Car('AB123CD', 'red')
print(c1) # equivalent to print(str(c1))
>>> AB123CD red False

```

CREAZIONE DELLE SUBCLASSI!

PROGRAMMA che manda in processo le due classi Employee e Manager

```

from employee import Employee      → così importo le classi NEL FILE
from manager import Manager

e1 = Employee('M. Rossi', 20000)   → Nome file → NOME CLASSE
m = Manager('Big Boss', 50000, 'Marketing')

print(m) # Visto che nome e paga sono attributi EREDITATI, quindi anche se non ho specificato le funzioni di stampa
        # __str__ e __repr__, tali funzioni sono state ereditate dalla class Employee.
        # ATTENZIONE!! Manca nella stampa ManagedUnit, che dovrà essere aggiunta. → nel file MANAGER

e2 = Employee('A. Arancioni', 3000)

# Creo una struttura dati per memorizzare i miei oggetti
employees = [e1, e2, m] # POLIMOSFIRMO, python si adatta in base all'oggetto che viene passato
for impiegato in employees:      → UTILIZZO UNA LISTA e un for per stampare tutto
    impiegato.increment_wage() # Prima incremento la paga, poi stampo.
    print(impiegato)          → applico la funzione creata nel file EMPLOYEE
                                e modificata ad hoc per il MANAGER
    print([impiegato.__class__.__name__]) → così STAMPO IL NOME DELLE CLASSI CHE UTILIZZO
    # Posso usare isinstance() per verificare

eA = Employee('G. Verdi', 20000)
eB = Employee('G. Verdi', 20000)

if eA == eB:
    print('Sono uguali')
else:
    print('Sono diversi')
    } semplice confronto
}

```

CLASSE EMPLOYEE → CLASSE PADRE

```

class Employee:
    def __init__(self, name, wage):
        self._name = name
        self._wage = wage

    @property
    def name(self):
        return self._name
    @name.setter
    def name(self, value):
        self._name = value

    @property
    def wage(self):
        return self._wage
    @wage.setter
    def wage(self, value):
        self._wage = value
    } GETTER e SETTER
    } posso anche scrivere self._NAME = NAME
    } NOME NON IMPORTANTE
    } mi fa stampare bene!
    def __str__(self):
        return f'{self._name}, {self._wage}' → ME USO SOLO UNO DEI DUE!
    def __repr__(self):
        return f'name={self._name}, wage={self._wage}'

    # Manager erediterà tale metodo.
    # Se tale metodo non ci va bene, possiamo ridefinirlo nel manager
    def increment_wage(self):
        self._wage += 5000 → il MANAGER PUÒ CANSINNE IL VALORE
    } Algoritmo di confronto
    def __eq__(self, other): → CREO UN ALGORITMO DI CONFRONTO!
        return self._name == other.name and self._wage == other.wage
}

```

CLASSE MANAGER → CLASSE FIGLIA

```

from employee import Employee → IMPORTO CLASSE PADRE!
# Prima di costruire la classe Manager, bisogna costruire la 'pancia' Employee del Manager.
# Per fare ciò uso la funzione super()

class Manager(Employee):
    def __init__(self, name, wage, managedUnit):
        super().__init__(name, wage) → prende ATTRIBUTI PADRE!
        self.managedUnit = managedUnit → AGGIUNGO UNA VARIABILE
    } aggiungo VARIABILE solo per la figlia
    def __str__(self):
        # return f'{self.name}, {self.wage}, {self.managedUnit}' → così ricevuto tutto
        return f'{super().__str__()}, {self.managedUnit}' # potrei delegare ad Employee il compito di stampare ciò che
    } così DELEGUO → lo riguarda e aggiungere solo gli attributi di Manager.
    def increment_wage(self): → NB!!
        self.wage += 20000 → se delego DEVO AGGIUNGERE LA VARIABILE!
}

```

```

from nome import CLASSE_PADRE
class CLASSE_FIGLIA(CLASSE_PADRE):
    super().__init__(variabili) → prendo attributi dal PADRE
    AGGIUNGO ATTRIBUTI FIGLIE STAMPA DAC PADRE HA DEVO AGGIUNGERE FIGLIE!
    def __str__(self):
        return f'{super().__str__()}, {self.variabili}' → POI CAMBIO LE VARIABILI RICHIAMANDOLE!
}

```

- La classe figlia eredita tutti gli attributi della classe padre associata.
- Possiamo poi definire altri attributi che appartengono solo alla classe figlia
- Gli attributi vengono ereditati automaticamente, è SBAGLIATO reinizializzarli come attributi della classe figlia.
- Una classe figlia può modificare anche ciò che nel padre non va bene, quindi può cambiare internamente il valore per esempio di una funzione come qua con increment_wage

Sovrascrivendo `eq()` rende possibile che due diverse istanze sono considerate uguali in base ai loro contenuti (attributi), piuttosto che le loro identità (cioè, Indirizzo di memoria, comportamento predefinito), sono uguali

In questo modo, viene definita una versione sovraccaricata di ==

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        if isinstance(other, Person):
            return self.name == other.name and self.age == other.age
        return False
```

N.B.! dove utilizzo `isinstance` per il controllo

CON QUESTO METODO LI CONFRONTO!!
↓ così per es. posso togliere i doppiioni

```
p1 = Person("Alice", 30)
p2 = Person("Alice", 30)
p3 = Person("Bob", 25)

print(p1 == p2) # True (contents match, i.e., same class type and attributes)
print(p1 == p3) # False
```

Simile a come possiamo cambiare == possiamo anche implementare < operatore utilizzando la funzione `__lt__(self, other)` e `__gt__(self, other)` per il >

Sennò per le comparazioni posso anche utilizzare `le_()` e `ge_()` per le comparazioni con <=, >=

Posso anche importare dei pacchetti come al solito `from math import sqrt`

Ma posso installare anche dei **pacchetti esterni** utilizzando il modulo pip

- Install with pip `install project_name`
 - `pip install flet`
 - `pip install mariadb`

in questo modo installiamo il programma flet

Aggiungendo il file **requirements.txt** lui in automatico mi aggiungerà a questo file se glielo dico i comandi principali del pacchetto esterno che ho installato

Di default `lt`, `gt`, `le`, `ge`, non sono definite come metodi ma se aggiungo `@dataclass(order=True)`

IMPLEMENTA TUTTO!
↑ DA APPROFONDIRE

IN QUESTO MODO HO LA POSSIBILITÀ DI ORDINARE GLI OGGETTI DELLA MIA CLASSE!!

```
def cabine_ordinate_per_prezzo(self): 1 usage & RobertaMacaluso *
    return sorted(self.cabine, key = lambda c: c.prezzo_base)
```

aggiungo la lista
dove sono contenuti
i miei dati

NB!!

al posto di attributi
posso utilizzare funzioni!!

QUESTI UGUALI
SCELGO COSÌ PER COSA ORDINARE!!

e se lo voglio decrescente
metto ,reverse=True)

per es: sorted_books = sorted(books, key = lambda book: book.pages)

se vogliamo ordinare per + elementi → utilizzo __lt__() nella parentesi di book

```
class Book: 7 usages new *
```

```
def __lt__(self, other): new *
```

ALGORITMO DI ORDINAMENTO, ES. PER ANNO

```
if self._year == other._year:
```

→ con other._year CONFRONTO PER TUTTI
GLI ALTRI VALORI

```
    return self._pages < other._pages
```

→ così ordino per anni
ma se poi uno è UGUALE
ordino per PAGINE

```
else:
```

```
    return self._year < other._year
```

utilizzerà queste condizioni!

e nell'altra parte del programma scrivo: sorted_books = sorted(books)

Per scaricare flet vado in basso a sx e vado su "Packages", cerco flet
e lo installo, poi import flet, poi metto un nuovo "requirements.txt" e di automatico
mi mette le regole.

import flet as ft → lo abbrevio perché lo userò spesso
modo di dire come sono i parametri

```
def main(page : ft.Page): # or simply main(page)
    # Here, controls are added and updated
    pass
```

```
ft.app(target = main)
```

→ crea un app flet e con target=main prenderà dentro il main

Se voglio aggiungere un testo in visione devo creare un controllo e mettere dentro il main cosa
vogliamo scrivere e mettere in app view= ft.AppView.FLET_APP

```
import flet as ft
def main(page):
    # Here, controls are added and updated
    myText = ft.Text(value = "Hello!", color = "blue")
    ft.app(target = main, view = ft.AppView.FLET_APP)
```

NB!!

sempre devo scrivere
sotto

```
page.controls.append(myText)
page.update()
```

→ posso aggiungerci un po' tutto!!
 tipo size=50 lo ingrandisce a 50

```

import flet as ft

def main(page):
    # Here, controls are added and updated
    myText = ft.Text(value = "Hello!", color = "blue")
    page.controls.append(myText)
    page.update()

ft.app(target = main, view = ft.AppView.FLET_APP)

```

Posso aggiungere dopo altre modifiche tipo cambiare colore!! poi devo update() se non cambia

Se utilizzo la funzione **sleep(n)** lui aspetterà n secondi prima di andare avanti!!!

PER METTERE UN PULSANTE: btnPress = ft.ElevatedButton(text = “Premi qui”)

Poi devo aggiungere i controlli con l'append e con update!!

```

def main(page: ft.Page):
    page.window.width = 400
    page.window.height = 300
    page.vertical_alignment = ft.MainAxisAlignment.CENTER
    page.window_resizable = True
    page.title = "Counter app"

    def handleAdd(e):
        currentVal = txtOut.value
        txtOut.value = currentVal + 1
        txtOut.update()

    def handleRemove(e):
        currentVal = txtOut.value
        txtOut.value = currentVal - 1
        txtOut.update()

#https://gallery.flet.dev/icons-browser/

btnMinus = ft.IconButton(icon = ft(Icons.REMOVE,
                           icon_color="green",
                           icon_size= 24, on_click= handleRemove)
btnAdd = ft.IconButton(icon = ft(Icons.ADD,
                           icon_color="green",
                           icon_size= 24, on_click= handleAdd)

txtOut = ft.TextField(width=100,disabled=True,
                      value=0, border_color="green",
                      text_align=ft.TextAlign.CENTER)

row1 = ft.Row([btnMinus, txtOut, btnAdd],
             alignment=ft.MainAxisAlignment.CENTER)
page.add(row1)

```

esempio con bottoni che togliamo
e mettiamo valori

così allineo CENTRATO
from flet import Checkbox
la devo IMPORTARE

Posso anche aggiungere Checkbox come temp= Checkbox(label=“Beer”, value = True)

Secondo i principi del modello **Model-View-Controller (MVC)**, quando si creano interfacce grafiche utente (GUI) è necessario considerare due aspetti principali:

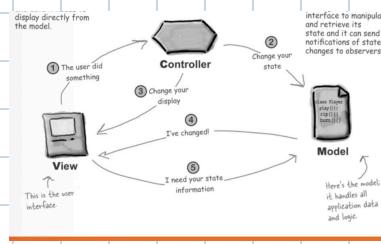
§ Layout (Vista): come posizionare gli elementi grafici per ottenere un aspetto visivo

§ Eventi (Controller): quale comportamento associare agli eventi degli elementi

§ La logica e i dati dell'applicazione (Modello) devono rimanere, per quanto possibile, separati dall'interfaccia utente

Creo un cartella MVC in cui dentro aggiungo 4 file: app.py view.py, model.py, control.py

In questi 4 file devo metterne determinate cose dividendole nel modo corretto e grazie a questo giusto smistamento riesco a scrivere la mia interfaccia



Esempio MVC

Questo è il file "app.py"

```
import flet as ft
from model import Model
from view import View
from controller import Controller

def main(page: ft.Page):
    m = Model()
    print(str(m.currentVal))
    v = View(page)
    c = Controller(v, m)
    v.setController(c)
    v.initInterface()

ft.app(target=main)
```

"model.py"

```
class Model():
    def __init__(self):
        self._currentVal = 0

    @property
    def currentVal(self):
        return self._currentVal

    @currentVal.setter
    def currentVal(self, value):
        self._currentVal = value
```

"controller.py"

```
import flet as ft
from model import Model

class Controller():
    def __init__(self, view: ft.View, model: Model):
        self._view = view
        self._model = model

    def handleAdd(self, e):
        self._model.currentVal += 1
        self._view.setTxtOutValue(self._model.currentVal)

    def handleRemove(self, e):
        self._model.currentVal -= 1
        self._view.setTxtOutValue(self._model.currentVal)
```

→ importa flet, qua è dove inizializzo l'app e
dove poi runno per vedere tutto

} → importo le tre classi di MVC

→ creo la classe Model
qua è dove aggiungo i valori da utilizzare!

} → getter e setter
NB! se li utilizzo dovrò mettere private la
variabile

→ nel controller IMPORTO FLET e MODEL!

→ per prendere dagli altri: **self._NAMEFILE.VARIABILE**
o **FUNZIONI!**

} aggiungo le funzioni che gestiscono quando
il pulsante viene premuto

→ così il valore che uscirà sarà il current value!

"view.py"

```
import flet as ft
from mvc.controller import Controller

class View():
    def __init__(self, page: ft.Page):
        self._page = page
        self._controller = None

    def setController(self, controller):
        self._controller = controller

    def initInterface(self):
        self._page.window.width = 400
        self._page.window.height = 300
        self._page.vertical_alignment = ft.MainAxisAlignment.CENTER
        self._page.window.resizable = True
        self._page.title = "Counter app"

    btnMinus = ft.IconButton(icon=ft(Icons.REMOVE,
                                    icon_color="green",
                                    icon_size=24, on_click=
                                    self._controller.handleRemove)
    btnAdd = ft.IconButton(icon=ft(Icons.ADD,
                                    icon_color="green",
                                    icon_size=24, on_click=
                                    self._controller.handleAdd))

    # Instance var., accessible from other methods
    self._txtOut = ft.TextField(width=100, disabled=True,
                                value=0,
                                border_color="green",
                                text_align=ft.TextAlign.CENTER)

    row1 = ft.Row([btnMinus, self._txtOut, btnAdd],
                  alignment=ft.MainAxisAlignment.CENTER)
    self._page.add(row1)

    def setTxtOutValue(self, value):
        self._txtOut.value = value
        self._txtOut.update()
```

→ nel view IMPORTO FLET e CONTROLLER!

nella view creo tutto quello che si vede!!
⇒ interfaccia, bottoni, testi ecc...

→ ripetendo la funzione nel controller

È fondamentale capire come suddividere i vari file con i propri compiti

Su Python posso lavorare con database come MySQL ecc

Bisogna quindi creare una connessione per l'accesso ai database e poterli lavorare direttamente da Python

I sistemi di gestione di database (DBMS) sono sistemi software utilizzati per archiviare, recuperare ed eseguire query sui dati, nonché per amministrarli.

Per connettere MySQL dobbiamo andare nei package e scaricare **mysql-connector-python**

Il pacchetto mysql-connector-python è un driver Python autonomo per la comunicazione con i server MySQL, utilizzando un'API conforme alla specifica Python Database API v2.0 (PEP 249).

Per la connessione devo scrivere:

```
import mysql.connector
cnx = mysql.connector.connect(user='admin',
                               password='',
                               host='localhost',
                               database='test2')

cnx.close()
```

→ importo il connector

FUNZIONE CONNECT connette
python e database

→ devo dentro aggiungere

- NOME UTENTE
- PASSWORD
- HOST → dove sta
- NOME DATABASE

La funzione connect() può generare eccezioni (ad esempio, se la connessione fallisce a causa di un'autenticazione errata).

È possibile/consigliato gestire queste eccezioni con una clausola **try-except-else-finally**:

1. Provare a connettersi
2. Gestire le eccezioni
3. Se non si sono verificate eccezioni, chiudere la connessione con close()

```
try:  
    cnx = mysql.connector.connect(user='admin',  
                                    password='',  
                                    host='localhost',  
                                    database='test2')  
  
except mysql.connector.Error as err:  
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:  
        print("Something is wrong w/ uname or password")  
    elif err.errno == errorcode.ER_BAD_DB_ERROR:  
        print("Database does not exist")  
    else:  
        print(err)  
else:  
    cnx.close()
```

Scrivere la configurazione del database e le informazioni di autenticazione nel codice non è l'ideale, soprattutto se il file viene elaborato in modo collaborativo (git).

È possibile/consigliabile utilizzare un file di configurazione separato e caricarlo nella funzione connect().

```
cnx = mysql.connector.connect(option_files=  
                               "/etc/mysql/connector.cnf")
```

Un cursore viene creato da una MySQLConnection utilizzando la funzione **cursor()**

- Esistono diverse classi di cursori che ereditano da MySQLCursor e possono essere create passando un argomento appropriato alla funzione cursor()
 1. MySQLCursorDict restituisce le righe come dizionari
 2. MySQLCursorNamedTuple restituisce le righe come tuple denominate
 3. MySQLCursorPrepared viene utilizzato per eseguire l'istruzione preparata

```
import mysql.connector  
  
cnx = mysql.connector.connect(...)  
cursor = cnx.cursor()  
cursor_dict = cnx.cursor(dictionary=True)  
cursor_tuple = cnx.cursor(named_tuple=True)  
cursor_prepared = cnx.cursor(prepared=True)  
  
cnx.close()
```

Un oggetto cursore ha un metodo **execute()** che consente di eseguire un'istruzione SQL, espressa come stringa

```
query = """SELECT id, name FROM user"""  
cursor.execute(query)
```

Utilizzando il cursore, è possibile eseguire operazioni SQL INSERT, UPDATE e DELETE come istruzioni parametriche

1. Definire l'istruzione (eventualmente utilizzando il blocco multi-riga Python """ """); i valori possono essere scritti nell'istruzione o lasciati non specificati come %s (poiché potrebbero dipendere dai dati dell'utente)
2. Eseguire l'istruzione (impostando tutti i valori non specificati)
3. Eseguire il commit delle modifiche nel database con **commit()**
4. Chiudere il cursore

→ se non metto il commit appena eseguo una istruzione non la posso poi più modificare

Insert data

```
add_user = """ INSERT INTO user  
    (id, name)  
    VALUES (%s, %s) """  
cursor.execute(add_user, (3, "John Doe"))  
cnx.commit()  
cursor.close()
```

generale così
posso riutilizzarla

codice utente aggiungo user

Update data

```
update_user = """ UPDATE user  
    SET name = %s  
    WHERE id = %s """  
cursor.execute(update_user, ("John Doe Jr.", 3))  
cnx.commit()  
cursor.close()
```

Delete data

```
delete_user = """ DELETE FROM user  
    WHERE id = %s """  
cursor.execute(delete_user, (3, ))  
cnx.commit()  
cursor.close()
```

↑ così cancellerà il numero 3

Per la stampa:

```
# LEGGO E STAMPO TUTTE LE RIGHE  
rows = cursor.fetchall() # Prende tutte le righe della tabella  
print(rows)
```

legge tutte le righe

```
# LEGGO E STAMPO LE RIGHE AD UNO AD UNO  
row = cursor.fetchone()  
while row is not None:  
    print(row)  
    row = cursor.fetchone()
```

legge una riga per volta

L'oggetto cursore ha anche altri metodi per recuperare i risultati ottenuti eseguendo un'istruzione di query.

- **fetchone()** recupera la riga successiva di un set di risultati di query e restituisce una singola sequenza, oppure None se non sono disponibili altre righe.
- **fetchmany(N)** recupera il set successivo di N righe di un risultato di query e restituisce un elenco di tuple (o dizionari o tuple denominate, se si utilizzano altri cursori specializzati).
- **fetchall()** recupera tutte (o tutte le rimanenti) righe di un set di risultati di query e restituisce un elenco di tuple (o dizionari o tuple denominate, se si utilizzano altri cursori specializzati); se non sono disponibili altre righe, restituisce un elenco vuoto.

Pattern DAO

- DAO (Data Access Object) è un pattern che funge da astrazione tra il database e l'applicazione principale

Si occupa di aggiungere, modificare, recuperare ed eliminare i dati. Non è necessario sapere come lo fa, questo è un'astrazione.

DAO è implementato in un file separato, ad esempio in una classe con metodi appropriati; questi metodi vengono poi chiamati nell'applicazione principale.

Esempio di database

creo il file che connette l'SQL con Python
database-connect.py:

```
import mysql.connector  
  
class DatabaseConnect:  
    def __init__(self):  
        pass  
  
    def get_connection(self):  
        try:  
            cnx = mysql.connector.  
                connect(option_files="connector.cnf")  
            return cnx  
        except mysql.connector.Error as err:  
            print(err)  
            return None
```

SCARICO IL PACCHETTO "mysql.connector"
e lo IMPORTO per connettere SQL e Python

mysql.connector.connect()
serve a connettere SQL con Py
e con questo try, except
posso stampare l'errore

User - DTO. py:

```
class UserDTO:  
    def __init__(self, id, name):  
        self.id = id  
        self.name = name  
  
    def __str__(self):  
        return f'{self.id} {self.name}'  
  
    def __eq__(self, other):  
        return self.id == other.id and  
               self.name == other.name  
  
    def __hash__(self):  
        return hash(self.id)
```

Classe DTO, per "MAPPIARE" una riga della tabella User del database in un oggetto Python
fondamentali: SEMPRE
init, str, eq

User-dao.py: → importo DTO e DatabaseConnect per connettere tutto

```

from user_dto import UserDTO
from database_connect import DatabaseConnect

class UserDAO:
    def __init__(self):
        self.database_connect = DatabaseConnect()

    def get_users(self):
        cnx = self.database_connect.get_connection()
        cursor = cnx.cursor(dictionary=True)
        query = """ SELECT *
                    FROM user """
        cursor.execute(query)
        result = []
        for row in cursor:
            result.append(UserDTO(row["id"], row["name"]))
        cursor.close()
        cnx.close()
        return result

    def add_user(self, user : UserDTO):
        cnx = self.database_connect.get_connection()
        if cnx is None:
            print("Database connection failed")
            return

        cursor = cnx.cursor(dictionary=True)
        query = """ INSERT INTO user (id, name)
                    VALUES (%s, %s) """
        cursor.execute(query, (user.id, user.name))
        cnx.commit()
        cursor.close()
        cnx.close()

if __name__ == "__main__":
    user_dao = UserDAO()
    users = user_dao.get_users()
    for user in users:
        print(user)

```

→ CONNETTO la classe DatabaseConnect()

→ aprovo la funzione per prendere gli users e prendo get_connection() da database connect

→ metto il cnx.cursor e aggiungo la query!

→ faccio execute(query)

→ chiudo cursor e cnx

→ creo un for nel cursor e appendo alla lista da DTO i dati riga per riga

→ aprovo la funzione che aggiunge gli utenti

→ commetto il database

→ aprovo il cursor e aggiungo la query per AGGIUNGERE elementi

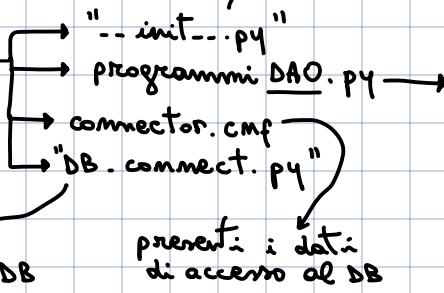
→ metto tutti gli users in una lista!

→ così mando in stampa gli utenti!

Come organizzo il mio programma?

- CARTELLA "Database"

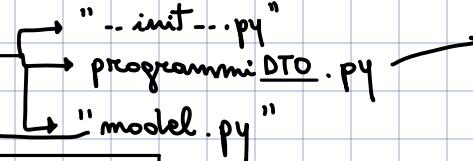
import mysql.connector
creo una classe ConnessioneDB per gestire un pool di connessioni al DB e creo la funzione get_connection



- ci connetto database.DB.connect e model.in DTO
- classe con pass, funzioni per leggere in cui aggiunge la QUERY con il cursor
- gestisce le operazioni di accesso al database

- CARTELLA "model"

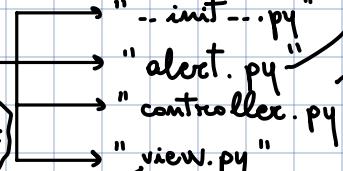
importo i database.DAO
dove poi creo la class Model e li metto dentro!
e aprovo le funzioni di ricerca nel database partendo dalle funzioni di lettura nel DAO e le analizzo a mio piacimento



from dataclasses import dataclass
utilizzo le dataclass per creare le classi! poi faccio in più __eq__, __str__ e __repr__

- CARTELLA "UI"

importo flet, model, view, controller e con def main faccio partire tutto



importo flet e creo class e funzioni per far usare gli alert di errore

importo flet, model e view
intermediario tra MODELLO e VIEW e gestisce il flusso dell'applicazione in cui popolo i dropdown con le funzioni del model, creo la funzione di selezione e servo il button handler

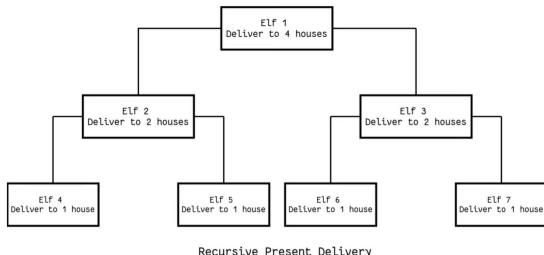
importo il flet, rappresenta l'interfaccia utente in cui quando uso l'interfaccia e ci aggiungo le funzioni create nel control

Recursion

La recursion si riferisce ad una tecnica di coding in cui la **funzione si richiama all'interno di essa**

Example: Santa Claus deliveries

- But it would probably be more effective to divide the work in chunks, among different workers



Example: Santa Claus deliveries

```
def deliver_presents_recursively(houses):  
    if len(houses) == 1:  
        house = houses[0]  
        deliver_to(house)  
    else:  
        mid = len(houses) // 2 #floored quotient x/y  
        first_half = houses[:mid]  
        second_half = houses[mid:]  
        deliver_presents_recursively(first_half)  
        deliver_presents_recursively(second_half)
```

↳ fa la funzione nella funzione
FINCHÉ la len è DIVERSA DA 1

Esempio: fattoriale

Dal contesto globale, che per primo invoca questo metodo, lo stack di chiamate crescerà fino a raggiungere il caso banale (1!)

Quindi la pila di chiamate si slitterà, passando i risultati indietro fino a raggiungere il contenuto globale

```
def factorial(n):  
    if n==0 or n==1: → se gli dico 0 o 1 dà 1  
        return 1  
    else: → se do un numero ≠ da 0 e 1 lui  
        return n*factorial(n-1) # ! moltiplica sempre per il numero -1  
                                e riporta la funzione!
```

```
f = factorial(387)  
print(f)
```

Come possiamo migliorare l'efficienza di runtime del metodo ricorsivo?

§ Utilizzare strutture di dati appropriate (miglioramenti tipicamente trascurabili su piccoli problemi)

§ Salta i thread di ricorsione che non producono risultati (può portare enormi miglioramenti)

§ Risultati intermedi della cache, se il sottoproblema corrispondente viene riscontrato più volte (i miglioramenti dipendono dal problema, c'è un costo di memoria).

Memoizzazione: tecnica di ottimizzazione utilizzata principalmente per accelerare i programmi per computer memorizzando i risultati di costose chiamate di funzione a funzioni pure e restituendo il risultato memorizzato nella cache quando gli stessi input si verificano di nuovo

Esercizio: anagrammi

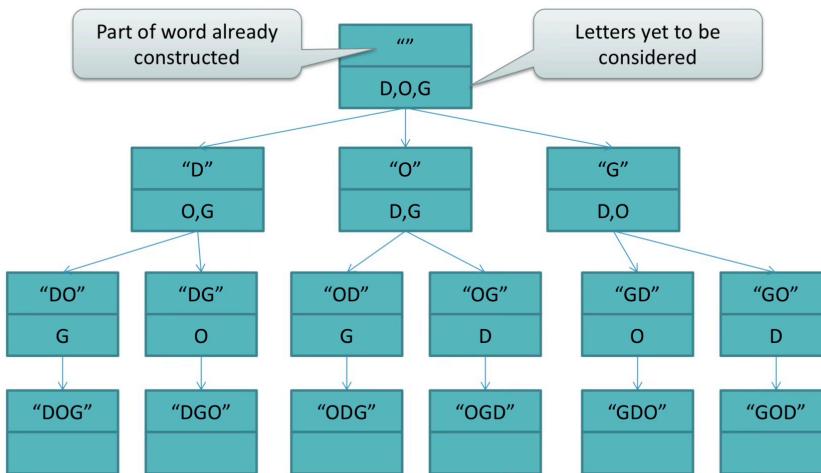
§ Data una parola, scrivi un programma che trovi tutti i possibili anagrammi di quella parola

§ Trova tutte le permutazioni degli elementi in un insieme

§ Le permutazioni sono $N!$

§ Ad esempio: da "cane", si ottiene cane, dgo, dio, gdo, odg, ogd

REALIZZAZIONE PROGRAMMA



ci sono anche i DB connect ecc...

"controller.py"

```

1 import flet as ft
2
3
4 class Controller:
5     def __init__(self, view, model):
6         # the view, with the graphical elements of the UI
7         self._view = view
8         # the model, which implements the logic of the program and holds the data
9         self._model = model
10
11     def calcola_anagrammi(self, e):
12         parola = self._view.txt_word.value
13         if parola == "":
14             self._view.create_alert("Inserisci una parola")
15         else:
16             anagrammi = self._model.calcola_anagrammi(parola)
17             for anagramma in anagrammi:
18                 self._view.lst_correct.controls.append(ft.Text(anagramma))
19             self._view.update_page()
20
21     def reset(self, e):
22         pass

```

implemento la classe e collego view e model

si inseriscono gli anagrammi e decido dove indirizzarli dopo averli calcolati

"model.py"

```

class Model:
    def __init__(self):
        pass

    def calcola_anagrammi(self, parola):
        self._anagrammi = []
        self.ricorsione("", parola)
        return self._anagrammi

    def ricorsione(self, anagramma_parziale, lettere_rimanenti):
        if len(lettere_rimanenti) == 0:
            self._anagrammi.append(anagramma_parziale)
            return
        else:
            for i in range(len(lettere_rimanenti)):
                anagramma_parziale = anagramma_parziale + lettere_rimanenti[i]
                nuove_lettere_rimanenti = lettere_rimanenti[:i]+lettere_rimanenti[i+1:]
                self.ricorsione(anagramma_parziale, nuove_lettere_rimanenti)
                anagramma_parziale = anagramma_parziale[:-1]

    if __name__ == "__main__":
        m = Model()
        print(m.calcola_anagrammi("dog"))

```

inserisco la funzione ricorsione che svolge veramente il calcolo

diviso in due blocchi

[] []

ANA. PARZIALE LETTERE RIMANENTI

e ci passo le lettere, finché in lettere rimanenti rimango a 0 e aggiungo l'anagramma

Richiamo RICORSIONE che risolve la funzione!

non vado solo AVANTI HA ANCHE INDIETRO! se mi perdo anagrammi

parola da ANAGRAMMA

BACKTRACKING →

```

tabs_results = ft.Tabs(
    selected_index=0,
    animation_duration=300,
    tabs=[

        ft.Tab(
            text="Anagrammi corretti",
            content=ft.Container(
                content=self.lst_correct,
                alignment=ft.alignment.center
            ),
        ),
        ft.Tab(
            text="Anagrammi errati",
            content=ft.Container(
                content=self.lst_wrong,
                alignment=ft.alignment.center
            ),
        ),
    ],
    expand=1,
)
self._page.controls.append(tabs_results)
self._page.update()

@property
def controller(self):
    return self._controller

@controller.setter
def controller(self, controller):
    self._controller = controller

def set_controller(self, controller):
    self._controller = controller

def create_alert(self, message):
    dig = ft.AlertDialog(title=ft.Text(message))
    self._page.open(dig)
    self._page.update()

def update_page(self):
    self._page.update()

```

così creo due pannelli uno per gli anagrammi giunti e uno per quelli sbagliati
apro Tabs e i Tab che mi servono

Container → contenuto del Tab!
content = ~~

NB!!
devo aggiungere il DAO per collegare il DB e vedere quali sono gli anagrammi giunti e quelli sbagliati, ma NON l'ho FATTO!

richiamo il controller

creo gli alert per gli errori

ricordo di dover aggiornare la pagina!

ESEMPIO REGINE: Data una griglia di lato N dire quante regine possono essere e dove!

```

import copy

class NRegine:
    def __init__(self):
        self._num_solutions = 0
        self._num_iterazioni = 0
        self._soluzioni = []

    def _risolvi_n_regine(self, N):
        self._num_solutions = 0
        self._num_iterazioni = 0
        self._soluzioni = []
        self._ricorsione([ ], N)

    def _ricorsione(self, parziale, N):
        self._num_iterazioni+=1
        # caso terminale
        if len(parziale)==N:
            #print(parziale)
            self._num_solutions+=1
            if self._soluzione_ammissibile(parziale):
                self._soluzioni.append(copy.deepcopy(parziale))

        # caso ricorsivo
        else:
            for row in range(N):
                for col in range(N):
                    parziale.append( (row, col) ) # regina
                    if self._nuova_regina_ammissibile(parziale):
                        self._ricorsione(parziale, N)
                    parziale.pop() # rimuovo l'ultima regina

    def _nuova_regina_ammissibile(self, parziale):
        # True se ammissibile, False altrimenti

        ultima_regina = parziale[-1]
        for regina in parziale[:len(parziale)-1]:
            # controllare righe
            if ultima_regina[0] == regina[0]: # stessa riga
                return False
            # controllare colonne
            if ultima_regina[1] == regina[1]: # stessa colonna
                return False
            # controllare diagonale
            if ((ultima_regina[0]-ultima_regina[1]) == (regina[0]-regina[1])):
                return False
            if ((ultima_regina[0]+ultima_regina[1]) == (regina[0]+regina[1])):
                return False
        return True

    def _soluzione_nuova(self, soluzione_nuova):
        for soluzione in self._soluzioni:
            for regina in soluzione_nuova:
                if regina in soluzione:
                    self._num_solutions-=1
                    return False
        return True

if __name__ == '__main__':
    nr = NRegine()
    nr._risolvi_n_regine(4)
    print(f"Trovate {nr._num_solutions} soluzioni")
    print(f"Chiamata {nr._num_iterazioni} la funzione ricorsiva")
    print(nr._soluzioni)

```

creo la classe dove metto all'interno i dati da dover utilizzare

soluzione dove implemento la ricorsione

RICORSIONE!

per mettere la soluzione devo usare sleepcopy

se è AMMISSIBILE dalla funzione di dopo la appendo

BACKTRACKING della ricorsione tolgo l'ultima regina così posso continuare la ricorsione

nella funzione controllo le regole per cui non ci può essere un'altra regina in orizzontale, verticale ed diagonale ora

utilizzo una formula matematica!

se ho già quella regina non la rimetto!

senon la aggiungo come soluzione fattibile

creo il main da riunire

✓	X	X
X	X	✓
X	✓	X

Quadrato magico

- Un insieme quadrato di numeri, solitamente numeri interi positivi, è chiamato quadrato magico se le somme dei numeri in ogni riga, in ogni colonna ed in entrambe le diagonali principali sono uguali.
- L'"ordine" del quadrato magico è il numero di numeri interi lungo un lato (n)
- Un insieme quadrato di numeri, solitamente positivi
- I numeri in un quadrato magico di ordine n sono $1, 2, \dots, n^2$ e interi, è chiamato quadrato magico se le somme dei numeri in ogni riga, in ogni colonna ed in entrambe le diagonali principali sono uguali.
- La somma costante è chiamata "costante magica".

2	7	6	→ 15
9	5	1	→ 15
4	3	8	→ 15

15 15 15 15

```

import copy

class QuadratoMagico:
    def __init__(self):
        self._soluzioni = [] # Lista di soluzioni
        self._num_iteorazioni = 0
        self._num_soluizioni = 0

    ...
    def _genera_rimanenti(self, N):
        rimanenti = []
        for i in range(1, N+N+1):
            rimanenti.append(i)
        ...
    def stampa_soluzione(self, soluzione, N):
        print("-----")
        for row in range(N):
            print([v for v in soluzione[row:N:(row+1)*N]])
        print("-----")

    def risolvi_quadrato_magico(self, N):
        self._soluzioni = [] # Lista di soluzioni
        self._num_iteorazioni = 0
        self._num_soluizioni = 0
        self._ricorsione([], set(range(1, N+N+1)), N) #self._genera_rimanenti()
        # Funzione utilizzata per verificare se la soluzione
        # CHE SI E' TROVATA sia valida o no
        def _is_soluizione_valida(self, parziale, N):
            # Numero magico
            M = (N*(N+N+1))/2
            # Verifica del vincolo sulle righe
            for row in range(N): # Per ognuna delle righe
                somma = 0
                # Verifica del vincolo sulle righe
                for row in range(N): # Per ognuna delle righe
                    somma = 0
                    sottolista = parziale[0:N+row : N] # Elementi di quella riga
                    for elemento in sottolista:
                        somma += elemento
                    if somma != M:
                        return False
            # Verifica del vincolo sulle colonne
            for col in range(N):
                somma = 0
                sottolista = parziale[0:N+col : (N-1)+N+col+1: N]
                for elemento in sottolista:
                    somma += elemento
                if somma != M:
                    return False
            # Verifica del vincolo sulla prima diagonale
            somma = 0
            for row_col in range(N):
                somma += parziale[row_col+N+row_col]
            if somma!=M:
                return False
            # Verifica del vincolo sulla seconda diagonale
            somma = 0
            for row_col in range(N):
                somma+= parziale[row_col+N-(N-1)-row_col]
            if somma!=M:
                return False
            # Funzione utilizzata per verificare se la soluzione
            # CHE SI STA TROVANDO (MENTRE LA SI TROVA) sia valida o no
            def _is_soluizione_valida_in_itinere(self, parziale, N):
                # Numero magico
                M = (N*(N+N+1))/2
                # Verifica del vincolo sulle righe
                n_righe = len(parziale)/N # Per le sole righe nella soluzione fino a quel momento
                for row in range(n_righe):
                    somma = 0
                    sottolista = parziale[0:N+row : N:(row+1)*N]
                    for elemento in sottolista:
                        somma += elemento
                    if somma != M:
                        return False
                # Verifica del vincolo sulle colonne
                n_col = max(len(parziale) - N*(N-1), 0) # Per le sole colonne nella soluzione fino a quel momento
                for col in range(n_col):
                    somma = 0
                    sottolista = parziale[0:N + col : (N-1)+N+col + 1: N]
                    for elemento in sottolista:
                        somma += elemento
                    if somma != M:
                        return False
                # Verifica del vincolo sulla prima diagonale
                if len(parziale) == N*N: # Effettuata solo se parziale ha lunghezza N*N
                    somma = 0
                    for riga_col in range(N):
                        somma += parziale[riga_col*N + riga_col]
                    if somma != M:
                        return False
            return True
    ...

```

copy per il deepcopy

nella ricorzione vengono sempre richiamate nella classe

fanno la stessa cosa solo che quello sotto è più compatto
↳ stampano la soluzione nel metodo corretto

la funzione di risoluzione richiama SEMPRE la ricorzione!
self.-ricorzione (LISTA VUOTA, utilizzo set per eliminare i duplicati!)

funzione che controlla, quando il parziale è COMPLETO, se è valida o no

qua calcolo direttamente per ogni riga o colonna
creo come sotto le sottoliste e faccio la somma
stessa cosa le colonne!

verifica anche della prima e seconda diagonale
come sotto!

le due funzioni sono molti simili ma la prima controlla già quando ho trovato la soluzione totale, mentre la seconda verifica mentre sto cercando una soluzione se il parziale è già valido!

funzione che controlla già se la soluzione parziale ha RIGHE, COLONNE, DIAGONALI = NUM. MAGICO

calcolo il num. righe, prendo solamente la riga come sottolista e faccio la somma di elementi e se è diversa dal num. magico => False

raccolgo il numero di colonne presenti, calcola la sottolista per prendere solo la colonna e somma elemento × elemento

calcolo per la prima diagonale la somma

```

# Verifica del vincolo sulla seconda diagonale
if len(parciale) == N*(N-1)/2: # E' effettuata solo se si è arrivati ad inserire in parziale il primo elemento
    somma = 0
    for riga_col in range(N):
        somma += parziale[riga_col] * N + (N-1-riga_col)
    if somma != M:
        return False

# tutti vincoli soddisfatti
return True

```

set di rimanenti
ogni volta che la utilizzo
incremento!
sono presenti tutti i numeri!

```

def _ricorsione(self, parziale, rimanenti, N):
    # Caso/condiz. terminale
    if len(parciale) == N:
        # print(parziale)
        # Così si verifica la soluzione dopo averla trovata:
        if self._is_soluzione_valida(parziale, N): # Io si potrebbe anche fare mentre la si cerca, per risparmiare ancora
            self._num_soluzioni += 1
            self.soluzioni.append(copy.deepcopy(parziale))

```

```

# Caso/condiz. ricorsiva
else:
    # Per evitare di provare sempre tutti i numeri da 1 a N+1 si usa set di rimanenti
    for i in rimanenti:#range(1, N+1):
        ...

```

```

        parziale.append(i)
        # Prepara della nuova lista di numeri rimanenti da provare
        nuovi_rimanenti = copy.deepcopy(rimanenti)
        nuovi_rimanenti.remove(i)
        self._ricorsione(parziale, nuovi_rimanenti, N)
        parziale.pop()
        ...

```

```

        parziale.append(i)
        # In alternativa, o in aggiunta, si può verificare la soluzione mentre la si cerca
        if self._is_soluzione_valida_in_itinere(parziale, N):
            # Prepara della nuova lista di numeri rimanenti da provare
            nuovi_rimanenti = copy.deepcopy(rimanenti)
            nuovi_rimanenti.remove(i)
            self._ricorsione(parziale, nuovi_rimanenti, N)
            parziale.pop()

```

```

if __name__ == '__main__':
    N = 3
    qm = QuadratoMagico()
    qm.risolvi_quadrato_magico(N)
    #print(qm._soluzioni)
    print(f"Risoluzione quadrato magico di lato N = {N} (numero magico M = {int((N*(N+N-1))/2)})")
    print(f"Numero soluzioni: {qm._num_soluzioni}")
    for soluzione in qm.soluzioni:
        qm.stampa_soluzione(soluzione, N)

```

per stampare!!
richiamo sempre la classe!

dell'ultima riga → verifico anche la 2° diagonale
e se tutti i vincoli sono soddisfatti lancio TRUE!
sempre → quasi sempre da utilizzare nella ricorsione

ricorsione (self, PARZIALE, RIMANENTI, N)

SE IL PARZIALE HA TUTTI I NUMERI → smetto la ricorsione
e cerco di capire se la soluzione è valida richiamando
la funzione che controllerà i requisiti e se la soluzione
è valida la appendo a soluzione UTILIZZANDO
append (copy.deepcopy (parziale))

per i miei numeri rimanenti li appendo a parziale
funzione che già controlla se la riga/colonna/diag formano
tutte e tre il numero magico.

Se si rimuovo la i oldi rimanenti e lancio i nuovi
rimanenti tramite la ricorsione
se la soluzione parziale già non è valida allora la
Tolgo per il backtracking

Creazione dei grafi

Si creano dal pacchetto NetworkX, che li crea, manipola e analizza le strutture dei grafi

Le strutture dei dati spesso devono essere rappresentate da grafi (per es. la metro di Milano)

I nodi sono gli oggetti Python da cui si creano gli archi che possono contenere dati arbitrati.

(Per esempio Famagoste è un nodo come Milano Centrale, invece la loro connessione è un arco tra i due nodi)

I nodi devono essere oggetti hashable, invece gli archi sono tuple con opzionale attributo
(per es. il peso)

Ci sono diverse tipologie di grafo:

- nx.Graph(): non direzionale, grafico semplice
- nx.DiGraph(): grafico semplice ma direzionale (A → B)
- nx.MultiGraph(): Supporta più archi tra due stessi nodi in ambe le direzioni
- nx.MultiDiGraph(): Supporta più archi tra due stessi nodi ma in modo direzionale

Ecco come si scrivono i grafi su Python:

```

import networkx as nx
import flet as ft

# Creo il grafo
g = nx.Graph()

# Aggiungo un nodo
# Tutti gli oggetti impostati come nodo DEVONO essere Hashable, cioè univoci
g.add_node("abc")
# g.add_node(ft.Text())
# g.add_node(2)

# Aggiungo altri archi
g.add_edge(1, 2)
g.add_edge(1, 2, attributo = 'pipi')
g.add_edge(1, 2, attributo = 'oppure no!')
g.add_edge(2, 3)
g.add_edge(2, 3, attributo = 'non ha effetto su un grafo semplice')
altri_nodi = [4, 5, 6, 7]
g.add_nodes_from(altri_nodi)
# aggiungere più nodi presi da una struttura dati
altri_archi = [(2, 4), (4, 5), (6, 7), (6, 8), (1, 2)]
# aggiunge altri archi da una struttura dati
g.add_edges_from(altri_archi)

```

N.B!

Creo un arco → all'arco posso aggiungere un ATTRIBUTO!
g.add_edge(1, 2, attributo = 'pipi')
g.add_edge(1, 2, attributo = 'oppure no!')

NB!
aggiungere più nodi presi da una struttura dati
g.add_nodes_from(altri_nodi)
dalle liste
altri_archi = [(2, 4), (4, 5), (6, 7), (6, 8), (1, 2)]
anche per gli archi!
aggiunge altri archi da una struttura dati
g.add_edges_from(altri_archi)

Mi dice quanti nodi e archi ha il grafo
print(g)

Mi stampa la lista degli id associati all'oggetto Nodo
print(g.nodes)
mi stampa i nodi

Stampa la lista di tuple degli archi mi stampa come tuple gli archi
print(g.edges)

Stampo tutti i nodi collegati ad arco con 1 e eventuali attributi.
primo_nodo = g[1] # Dizionario con chiave i nodi e valore gli attributi(opzionali)
print(primo_nodo) # in questo modo stamperebbe tutti i nodi collegati a 1 che saranno la chiave e in caso l'attributo

```

# Creo grafo orientato
#directed graph, grafo diretto quello di prima semplice
dg = nx.DiGraph() → GRAFICO DIREZIONALE!!
→ vale solo per una direzione!!

dg.add_nodes_from(altri_nodi)
dg.add_edges_from(altri_archi)      solo (u,m)
print(dg)                          NO (m,u)

print(dg[4]) # esiste un arco verso 5.
# Non mi escono però quelli scritti (n,4) perché QUESTO è DIREZIONALE

print(dg[5]) # non esistono archi che partono con 5

```

MULTIGRAFO = + archi x stessi nodi!

```

# Creo un multigrafo
mg = nx.MultiGraph()

mg.add_edge(1,2, weight = 5)

mg.add_edge(1,3) # possono essere aggiunti più grafi che connettono due nodi nel multigrafo

mg.add_edge(1,2) → NB!
mg.add_edge(1,2, attributo = 'pippo')
#nel multigrafo posso mettere più grafi che connettono due nodi e nella stampa di connessione
#saranno dentro una graffa grossa e numerati dal primo scritto all'ultimo con eventuali attributi o peso

print(mg[1]) → printerà i 3 con il 2 e quello con il 3
print(f'Arco tra 1 e 2 0-esimo: {mg[1][2][0]}' → print il primo arco tra [1][2]
#questo stampa il valore dell'arco numero 0 tra [num_iniziale][num_finale][num_attributo]
#ovviamente si parte da zero, se mettessi per esempio [1][2][2] uscirebbe pippo

```