Cheatsheet Programmazione Avanzata

Flet

```
# ListView
# Dichiarazione
self.list_view = ft.ListView(
        expand=1,
        spacing=10,
        padding=20,
        autosccroll=True
)

# Aggiungere elementi
self.list_view.controls.append(
        ft.Text("Nuova riga")
)
# Ma anche
for elemento in lista :
        self.list_view.controls.append(
                ft.Text("Nuova riga")
)

self.page.update()

# Ripulire
self.list_view.clean()
self.page.update()

# ElevatedButton
# Dichiarazione
self.btn_action = ft.ElevatedButton(
        text="Clicca",
        on_click=self.handle_click
)

# Disabilitare/Abilitare
self.btn_action.disabled = True # Blocca il tasto
self.btn_action.disabled = False # Sblocca il tasto
self.page.update()

# Cambiare testo
self.btn_aciton.text = "Riclicca"
self.page.update()

# Dropdown
# Dichiarazione
self.dd_scelta = ft.Dropdown(
```

```
        label="Seleziona",
        width=200,
        options=[
            ft.dropdown.Option("OPZIONE_A"),
            ft.dropdown.Option("OPZIONE_B")
        ]
)

# Lettura dati
valore = self.dd_scelta.value # Restituisce la key della dropdown

# Aggiungere elementi
for elemento in lista :
        self.dd_scelta..options.append(
            ft.dropdown.Option(
                    key="OPZIONE_C", # La key deve essere una stringa
                    text="Opzione C" # Questo è solo il valore che viene visualizzato dall'utente
            )
)

# Ripulire
self.dd_scelta.options.clear() # Oppure self.dd_scelta.clean()
self.page.update()

# TextField
# Dichiarazione
self.txt_input = ft.TextField(
        label="Inserisci valore",
        width=200
)

# Lettura dati
testo = self.txt_input.value

# Ripulire
self.txt_input.value = ""
self.txt_input.focus() # Opzionale: rimette il cursore nel campo
self.page.update()

# Text (Etichetta/Messaggio singolo)
# Dichiarazione
self.lbl_msg = ft.Text(
        value="Benvenuto",
        color="red",
        size=20
)

# Lettura dati
contenuto = self.lbl_msg.value

# Ripulire
```

```
self.lbl_msg.value = ""
self.page.update()
```

NetworkX

```
# Grafo non orientato (bidirezionale)
self.grafo = nx.Graph()

# Grafo orientato (frecce, senso unico)
self.grafo = nx.DiGraph()

# Ripulire il grafo
self.grafo.clear()

# Aggiunta di nodi
self.grafo.add_node("Roma")
self.grafo.add_node(
    "Milano",
    tipo="Città",
    abitanti=1000
)

# Aggiunta di nodi da una lista
lista_nodi = ["Roma", "Napoli", "Torino"]
self.grafo.add_nodes_from(lista_nodi)

# Accesso a dati di un nodo (grafo.nodes[id_nodo]["nome_attributo"])
popolazione = self.grafo.nodes["Milano"]["abitanti"]

# Aggiunta di archi
self.grafo.add_edge("Roma", "Milano", weight=10)

# Aggiunta di archi da lista di tuple
archi = [
    ("Roma", "Napoli", 2),
    ("Napoli", "Bari", 5)
]
self.grafo.add_weighted_edges_from(archi)

# Accesso a dati di un arco (grafo[nodo_partenza][nodo_arrivo]["nome_attributo"])
distanza = self.grafo["Roma"]["Milano"]["weight"]

# Trovare i vicini a partire da un nodo
vicini = list(self.grafo.neighbors("Roma")

# Trovare il grado di un nodo (numero di connessioni)
numero_connessioni = self.grafo.degree("Roma")

# Trovare (per i grafi orientati) il numero di archi uscenti/entranti
archi_uscenti = self.grafo.in_degree("Roma")
```

```
archi_entranti = self.grafo.out_degree("Roma")

# Iterare su tutti i nodi o su tutti gli archi
for nodo in self.grafo.nodes():
        print (nodo)

for u, v, data in self.grado.edges(data=True):
        print (f"Da {u} a {v} con peso {data["weight"]}.")

# Utile per verificare efficacia algoritmo di ricerca di cammino minimo
path = nx.shortest_path(
        self.grafo,
        source="Roma",
        target="Milano",
        weight="weight"
)
```

Scheletri algoritmi ricorsivi

Algoritmo che a partire da uno start fisso, da un end fisso e da una lunghezza del cammino fissa, calcola il percorso che massimizzi la somma dei pesi

```
def cerca_cammino(self, nodo_start, nodo_end, lunghezza_target):
    self.bestPath = []
    self.bestWeight = -1 # Uso -1 perché cerco il massimo

    # Il path contiene i nodi. Un cammino di lunghezza L archi ha L+1 nodi.
    # Start → A → End (2 archi, 3 nodi)

    self._ricorsione(nodo_start, nodo_end, lunghezza_target, [nodo_start], 0)

    return self.bestPath, self.bestWeight

def _ricorsione(self, corrente, target, len_target, parziale, peso_parziale):

    # 1. CONTROLLO LUNGHEZZA (PRUNING)
    # Se ho già troppi nodi rispetto alla lunghezza richiesta, mi fermo.
    # Ricorda: L archi = L+1 nodi
    if len(parziale) == len_target + 1:
        # Se sono arrivato alla lunghezza giusta, controllo se sono sul target
        if corrente == target:
            # Trovata soluzione valida! Controllo se è la migliore (peso massimo)
            if peso_parziale > self.bestWeight:
                self.bestWeight = peso_parziale
                self.bestPath = list(parziale)
        return # Backtrack in ogni caso, non vado oltre L

    # 2. ESPLORAZIONE
    for vicino in self.grafo.neighbors(corrente):
        # Vincolo: nodo non attraversato più volte
```

```
        if vicino not in parziale:

            peso_arco = self.grafo[corrente][vicino]['weight']

            parziale.append(vicino)

            # Ricorsione
            self._ricorsione(vicino, target, len_target, parziale, peso_parziale + peso_arco)

            # Backtracking
            parziale.pop()
```

Trovare il percorso più breve partendo da un nodo specifico

```
def cerca_cammino(self, nodo_start, nodo_end, lunghezza_target):
    self.bestPath = []
    self.bestWeight = -1 # Uso -1 perché cerco il massimo

    # Il path contiene i nodi. Un cammino di lunghezza L archi ha L+1 nodi.
    # Start → A → End (2 archi, 3 nodi)

    self._ricorsione(nodo_start, nodo_end, lunghezza_target, [nodo_start], 0)

    return self.bestPath, self.bestWeight

def _ricorsione(self, corrente, target, len_target, parziale, peso_parziale):

    # 1. CONTROLLO LUNGHEZZA (PRUNING)
    # Se ho già troppi nodi rispetto alla lunghezza richiesta, mi fermo.
    # Ricorda: L archi = L+1 nodi
    if len(parziale) == len_target + 1:
        # Se sono arrivato alla lunghezza giusta, controllo se sono sul target
        if corrente == target:
            # Trovata soluzione valida! Controllo se è la migliore (peso massimo)
            if peso_parziale > self.bestWeight:
                self.bestWeight = peso_parziale
                self.bestPath = list(parziale)
        return # Backtrack in ogni caso, non vado oltre L

    # 2. ESPLORAZIONE
    for vicino in self.grafo.neighbors(corrente):
        # Vincolo: nodo non attraversato più volte
        if vicino not in parziale:

            peso_arco = self.grafo[corrente][vicino]['weight']

            parziale.append(vicino)

            # Ricorsione
```

```
            self._ricorsione(vicino, target, len_target, parziale, peso_parziale + peso_arco)

            # Backtracking
            parziale.pop()
```

Trovare il percorso più breve senza sapere da che nodo partire

```
def get_percorso_start_variabile(self):
    self.bestPath = []
    self.bestScore = 0

    # UNICA DIFFERENZA: Un ciclo for che lancia la ricorsione su tutti i nodi
    for nodo in self.grafo.nodes():
        # Lancio la ricorsione trattando ogni nodo come potenziale start
        self._ricorsione(nodo, [nodo], 0)

    return self.bestPath

# La funzione _ricorsione è identica al caso 1
```

Trovare il percorso più lungo, minimizzando i pesi, a partire da un nodo dato

```
def _ricorsione_complex(self, nodo_corrente, parziale, peso_parziale):
    # --- LOGICA DI AGGIORNAMENTO BEST ---
    # Caso 1: Ho trovato un percorso più lungo di quello salvato → Sostituisco
    if len(parziale) > len(self.bestPath):
        self.bestPath = list(parziale)
        self.bestScore = peso_parziale

    # Caso 2: Lunghezza uguale, ma questo pesa meno → Sostituisco
    elif len(parziale) == len(self.bestPath) and peso_parziale < self.bestScore:
        self.bestPath = list(parziale)
        self.bestScore = peso_parziale

    # --- ESPLORAZIONE ---
    for vicino in self.grafo.neighbors(nodo_corrente):
        if vicino not in parziale:
            # Controllo eventuali vincoli (es. non superare peso massimo)
            peso_arco = self.grafo[nodo_corrente][vicino]['weight']

            parziale.append(vicino)
            self._ricorsione_complex(vicino, parziale, peso_parziale + peso_arco)
            parziale.pop()
```

Trovare il percorso più lungo, minimizzando i pesi, senza sapere da che nodo partire

```
def get_max_nodes_min_weight(self):
    self.bestPath = []
    self.bestScore = float('inf') # Inizializzo a infinito perché devo MINIMIZZARE il peso a parità di lungh
ezza

    for nodo in self.grafo.nodes():
        self._ricorsione_complex(nodo, [nodo], 0)

    return self.bestPath
```

Algoritmo che trova il percorso che massimizza un attributo, con pesi sempre crescenti

```
def get_cammino_pesi_crescenti(self):
    self.bestPath = []
    self.bestScore = 0 # Qui lo score è la DISTANZA totale (km)

    # Provo a partire da OGNI nodo del grafo
    for nodo in self.grafo.nodes():
        # (nodo_curr, path_list, distanza_totale, peso_ultimo_arco)
        # peso_ultimo_arco inizia a -1 per accettare qualsiasi primo arco
        self._ricorsione(nodo, [nodo], 0, -1)

    return self.bestPath, self.bestScore

def _ricorsione(self, nodo_curr, parziale, dist_parziale, peso_last_arco):

    # --- A. AGGIORNAMENTO BEST ---
    # Cerco di massimizzare la distanza percorsa
    if dist_parziale > self.bestScore:
        self.bestScore = dist_parziale
        self.bestPath = list(parziale)

    # --- B. ESPLORAZIONE VICINI ---
    for vicino in self.grafo.neighbors(nodo_curr):
        if vicino not in parziale:

            # 1. Recupero peso dell'arco attuale
            peso_arco_curr = self.grafo[nodo_curr][vicino]['weight']

            # 2. VINCOLO FONDAMENTALE: Pesi crescenti
            # Posso andare avanti solo se il peso attuale è MAGGIORE del precedente
            if peso_arco_curr > peso_last_arco:

                # 3. Calcolo la distanza aggiuntiva (Geodesica)
                # Recupero le coordinate salvate nel grafo nel punto 1
                c1 = (self.grafo.nodes[nodo_curr]['lat'], self.grafo.nodes[nodo_curr]['lng'])
                c2 = (self.grafo.nodes[vicino]['lat'], self.grafo.nodes[vicino]['lng'])
                dist_extra = geodesic(c1, c2).km
```

```
            # 4. Ricorsione e Backtracking standard
            parziale.append(vicino)
            self._ricorsione(vicino, parziale, dist_parziale + dist_extra, peso_arco_curr)
            parziale.pop()
```

Algoritmo che calcola il percorso più breve in cui il peso degli archi è decrescente e gli archi scelti sono i primi k archi (i k archi più pesanti)

```
def get_cammino_decrescente(self, nodo_start):
    self.bestPath = []
    self.bestScore = 0 # Qui lo score è il PESO TOTALE del cammino

    # Lancio la ricorsione
    # peso_last_arco: metto infinito (float('inf')) così il primo arco è sempre valido (qualunque numero è
< infinito)
    self._ricorsione(nodo_start, [nodo_start], 0, float('inf'))

    return self.bestPath, self.bestScore

def _ricorsione(self, nodo_curr, parziale, peso_totale, peso_last_arco):

    # --- A. AGGIORNAMENTO BEST ---
    if peso_totale > self.bestScore:
        self.bestScore = peso_totale
        self.bestPath = list(parziale)

    # --- B. PREPARAZIONE VICINI (PRUNING) ---
    # 1. Recupero tutti i vicini con i loro pesi
    vicini_ordinati = []
    for vicino in self.grafo.neighbors(nodo_curr):
        edge_weight = self.grafo[nodo_curr][vicino]['weight']
        vicini_ordinati.append((vicino, edge_weight))

    # 2. Li ordino per peso DECRESCENTE (così prendo i più pesanti)
    vicini_ordinati.sort(key=lambda x: x[1], reverse=True)

    # 3. PRUNING: Prendo solo i primi K (es. K=3)
    vicini_top_k = vicini_ordinati[:3]

    # --- C. ESPLORAZIONE ---
    for vicino, peso_arco in vicini_top_k:
        # Controllo 1: Non visitato
        if vicino not in parziale:

            # Controllo 2: VINCOLO DECRESCENTE
            # Il nuovo arco deve pesare MENO del precedente
            if peso_arco < peso_last_arco:
```

```
            parziale.append(vicino)

            # Ricorsione
            self._ricorsione(vicino, parziale, peso_totale + peso_arco, peso_arco)

            # Backtracking
            parziale.pop()
```

Algoritmo che cerca il set più pesante sotto un tetto massimo

```
# Algoritmo che cerca il set più pesante sotto un tetto massimo
def get_set_album_ottimo(self, album_a1, d_tot_minuti):
    # 1. Preparazione Dati
    # Ottengo la componente connessa (i candidati)
    componente = nx.node_connected_component(self.grafo, album_a1)

    # Creo una lista di candidati rimuovendo a1 (che è obbligatorio)
    self.candidati = [n for n in componente if n != album_a1]

    # Setup Variabili per Ricorsione
    self.bestSet = []
    self.maxCount = 0 # Cerco il numero massimo di elementi

    # Recupero durata a1
    durata_a1 = self.grafo.nodes[album_a1]['duration']

    # Controllo preliminare: se a1 da solo supera il limite, ritorno solo a1 (o errore)
    if durata_a1 > d_tot_minuti:
        return [album_a1] # O gestisci come vuoi

    # Lancio ricorsione
    # parziale = [album_a1] → Inizio già con a1
    # durata_parziale = durata_a1
    # livello = 0 → Indice per scorrere la lista self.candidati
    self._ricorsione(0, [album_a1], durata_a1, d_tot_minuti)

    return self.bestSet

def _ricorsione(self, indice, parziale, durata_curr, tetto_max):

    # --- A. AGGIORNAMENTO BEST ---
    # Se il set attuale è valido e ha PIÙ elementi del best, aggiorno
    if len(parziale) > self.maxCount:
        self.maxCount = len(parziale)
        self.bestSet = list(parziale)

    # --- B. CONDIZIONE DI FINE ---
    # Se ho finito i candidati scorrendo la lista
    if indice >= len(self.candidati):
        return
```

```
    # --- C. ESPLORAZIONE (Backtracking binario: PRENDO o LASCIO) ---

    # Nodo che stiamo valutando
    nodo_cand = self.candidati[indice]
    durata_cand = self.grafo.nodes[nodo_cand]['duration']

    # OPZIONE 1: PRENDO il nodo (solo se ci sto dentro col peso)
    if durata_curr + durata_cand <= tetto_max:
        parziale.append(nodo_cand)
        # Vado avanti all'indice + 1, con durata aggiornata
        self._ricorsione(indice + 1, parziale, durata_curr + durata_cand, tetto_max)
        parziale.pop() # Backtrack

    # OPZIONE 2: LASCIO il nodo (lo salto)
    # Vado avanti all'indice + 1, mantenendo la stessa durata e stesso parziale
    self._ricorsione(indice + 1, parziale, durata_curr, tetto_max)
```

Query SQL

```
-- Estrarre anno/mese
SELECT YEAR(DateColumn), MONTH(DateColumn)
FROM Table

-- Filtrare per anno specifico
SELECT * FROM Table WHERE YEAR(DateColumn) = 2022

-- Intervallo di date
-- Metodo Standard
SELECT * FROM Table WHERE DateColumn >= '2022-01-01' AND DateColumn <= '2022-12-31'
-- Oppure (se supportato dal DB)
SELECT * FROM Table WHERE DateColumn BETWEEN '2022-01-01' AND '2022-12-31'

-- Calcolare differenza di giorni
DATEDIFF(DataFine, DataInizio) -- Restituisce i giorni di differenza

-- Conteggio semplice (es. quanti avvistamenti per stato)
SELECT state, COUNT(*) as N_Avvistamenti
FROM sighting
GROUP BY state
ORDER BY N_Avvistamenti DESC -- Dal più grande al più piccolo (il contrario è ASC)

-- Somma valori (es. totale vendite per prodotto)
SELECT Product_ID, SUM(Price) as Totale_Venduto
FROM sales
GROUP BY Product_ID

-- Filtro dopo aver raggruppato
SELECT state, COUNT(*) as cnt
```

```
FROM sighting
GROUP BY state
HAVING cnt > 100  -- ← Filtra sui risultati dell'aggregazione

-- Join standard a 3 tabelle
SELECT a.Name, t.Name
FROM Artist a, Album alb, Track t
WHERE t.AlbumId = alb.AlbumId      -- Collega Traccia e Album
  AND alb.ArtistId = a.ArtistId    -- Collega Album e Artista
  AND a.Name = 'Queen'             -- Filtro

-- Distinct (per eliminare duplicate)
SELECT DISTINCT a.Title
FROM Album a ...

-- Pattern standard per coppie non duplicate
SELECT t1.GeneID, t2.GeneID, i.Correlation
FROM genes t1, genes t2, interactions i
WHERE t1.Chromosome = t2.Chromosome    -- Condizione: stesso cromosoma
  AND t1.GeneID < t2.GeneID            -- EVITA DUPLICATI E SELF-LOOPS
  AND i.GeneID1 = t1.GeneID            -- Collega alla tabella interazioni
  AND i.GeneID2 = t2.GeneID

-- Self Join
-- Trova coppie di Album (A, B) che stanno nella stessa Playlist
SELECT DISTINCT t1.AlbumId as A, t2.AlbumId as B
FROM PlaylistTrack t1, PlaylistTrack t2
WHERE t1.PlaylistId = t2.PlaylistId -- Stesso contenitore
  AND t1.AlbumId < t2.AlbumId       -- Evita duplicati (A-B e B-A) e self-loop

-- Many-to-Many Join
-- Usato in questo caso per collegare due cromosomi solo se hanno geni che interagiscono
SELECT g1.Chromosome as C1, g2.Chromosome as C2, i.Expression_Corr
FROM genes g1, genes g2, interactions i
WHERE i.GeneID1 = g1.GeneID      -- Link Gene 1
  AND i.GeneID2 = g2.GeneID      -- Link Gene 2
  AND g1.Chromosome <> g2.Chromosome -- Esclude stesso cromosoma
```