Quaderno di Python

Tutte le informazioni scritte in corsivo presenti nel testo, portano ad un collegamento sull'argomento con spiegazioni più approfondite

INTRODUZIONE

Come nasce Python?

Python nasce nel 1991, come 'progetto invernale' del programmatore *Van Rossum,* sviluppatore di ABC. Lo scopo di quest'ultimo era quello di creare un linguaggio più semplice e facile per i programmatori futuri. Difatti, come sappiamo, Python è uno dei linguaggi di alto livello più semplici da imparare in circolazione (con pochissima concorrenza). Nell'ultimo periodo ha preso particolare importanza in diversi ambiti, quali possono essere quello del Web Developing, Machine Learing, Game Developing… Facendolo diventare uno dei linguaggi di programmazione più apprezzati al mondo, con una comunità che ,nel suo ultimo periodo, ha raggiunto fino ai 15 milioni di utenti.

Perchè utilizzare Python?

Python, come abbiamo detto precedentemente, è ottimo per iniziare ad introdursi nel mondo della programmazione. Un linguaggio facile e comprensibile, ma sopratutto particolarmente intuitivo nei comandi e nelle parole chiave. Il vantaggio più grande? Essere Open Source. Essendo difatti Python Open Source, può essere modificato dinamicamente dai suoi utenti, introducendo moduli e librerie (che vedremo meglio successivamente) per rendere la programmazione più semplice, facile e divertente. Detto questo, possiamo entrare 'nel vivo' del discorso, iniziando a parlare di:

Variabili, print(), input() e commenti

Queste saranno le prime cose che vedremo in Python. Partiamo descrivendo le prime due:
- La prima cosa che potrebbe venirci in mente, e quella di far visualizzare all'utente un 'qualcosa' ; come un saluto, una domanda, una proposta sul suo schermo. Per 'stampare' su schermo, o nel nostro caso, sull' IDLE, questi messaggi, dovremmo quindi scrivere la nostra prima funzione, pietra miliare di Python… il print(). Ecco la sua sintassi (semplificata)

print( )

Con il print() possiamo quindi far visualizzare all'utente un messaggio. Quest'ultimo potrà essere una variabile, una stringa, oppure una serie di quest'ultimi divisi da una virgola. Ecco un esempio:

```
print("Hello Word !")
print("Hello", "Word", "!")
print(CiaoMondo)
```

Ma cosa stiamo 'stampando'? Cosa sono le stringhe? Cosa sono le variabili? Ecco allora che introduciamo adesso le variabili, il loro funzionamento, la loro dichiarazione e utilizzo all'interno del programma
- Le variabili, nel programma sono come delle 'scatole vuote'. Il loro compito, infatti è quello di contenere le informazioni che dovranno essere utilizzate successivamente nel programma. Adesso, diversamente da altri linguaggi di programmazione, le variabili in Python non necessitano di avere specificato il tipo, e non utilizzano parole chiavi ad esse collegate, rendendo la lodo dichiarazione particolarmente semplice. Ecco come possiamo dichiarare le variabili in Python:

NomeVariabile = Valore ad esso collegato

Capiamo quindi che la variabile, dovrà avere un nome ed un valore contenuto. Per quanto riguarda il nome, potrà essere il più variegato possibile, almeno che quest'ultimo non interferisca con le parole chiave di Python (Per esempio, una variabile non potrà chiamarsi 'print') o che inizi con un numero (Non potrà chiamarsi '1valore'). Per suddividere il nome di una variabile, se necessario, si può usare il simbolo '_' che prende il nome di Snake-Case. Dopo aver capito come possiamo dare un nome ad una variabile, capiamo invece i valori che quest'ultima può contenere:

- **int**: Numeri interi. Il programma potrà contenere valori positivi o negativi come: -42, 134, -1682, 0;

- **float**: Numeri a virgola mobile. Il programma potrà contenere valori positivi e negativi con la virgola;
- **bool**: Valori veri o falsi. Il programma potrà contenere valori “True”(Veri) o “False”(Falsi);
- **complex**: Numeri complessi con parte immaginaria. Il programma potrà contenere, per esempio, polinomi;
- **str**: Stringhe, cioè una serie di caratteri, lettere. Il programma le rappresenta tra virgolette “ ” o apici ‘ ’
- **bytes**: Numeri binari . Il programma potrà contenere una serie di byte all’interno

Ecco un esempio per capire meglio quest’ultimi (ognuno verrà visto meglio successivamente):

NumeroIntero = 43
NumeroReale = 3.14
ValoreBooleano = True
NumeroComplesso = 4 + 6x
Stringa = “Hello Word!”
NumeroBinario = /x00/x01/x02/…

Abbiamo quindi capito perché chiamavamo stringhe le parole prima espresse all’interno delle virgolette nel print(), e cosa significava la variabile ‘CiaoMondo’ . Una cosa importante, però, di quando vogliamo ‘stampare’ delle variabili, dobbiamo far si che queste ultime siano sempre specificate prima, ecco un esempio:

NO	SI
print(CiaoMondo) CiaoMondo = “Hello Word”	CiaoMondo = “Hello Word” print(CiaoMondo)

- Possiamo passare poi all'input(). Diversamente dal print(), quest’ultimo, oltre a mostrare a schermo un messaggio, aspetterà che l’utente gli ‘risponda’ attraverso l’utilizzo della tastiera. In questo modo, il programma potrà ricevere delle informazioni che userà successivamente. Se, per esempio, stessimo creando un Quiz, e dobbiamo registrare un punteggio, dobbiamo sapere quando l’utente sbaglia o no, e di conseguenza togliere o aggiungere punti. Sara quindi necessario utilizzare una variabile. La sintassi del’ input sarà quindi:

VariabileContenitore = input()

Dove la variabile contenitore è una qualsiasi variabile, come abbiamo visto prima, mentre l'input potrà essere vuoto, o contenere una stringa o una variabile (Non una serie di quest’ultimi usando la ‘,’). Il valore di base immagazzinato nell’input sarà una stringa. Se volessimo di conseguenza modificarlo, basterà scrivere int(input()) se volessimo renderlo intero (con la condizione che l’utente inserisca un intero) o float(input()) se volessimo renderlo reale (con la condizione che l’utente inserisca un reale), ecc… Ovviamente la ‘VariabileContenitore’ non è strettamente necessaria, ma se scrivessimo solo input() vorrà dire che il valore ottenuto non sarà poi registrato. Chiudiamo adesso con i commenti.
- Lo scopo dei commenti è di tenere traccia nel programma di ciò che stiamo facendo. Esistono tre metodi per rappresentare i commenti in python:

- Se messi tra tre virgolette aperte e tre virgolette chiuse, scriveremo un paragrafo di commento;
- Se messi tra tre apici aperti e tre apici chiusi, scriveremo ancora un paragrafo di commento;
- Se utilizzassimo un #, scriveremo un commento su una singola riga;

Ecco un esempio per ciascuno dei tre:

“”” PARAGRAFO DI COMMENTO “””
‘’’ PARAGRAFO DI COMMENTO ‘’’
 # COMMENTO A RIGA SINGOLA

Condizioni: if, elif ed else

Scrivendo il nostro codice, potrebbe capitare di dover scegliere tra due strade possibili: mettiamo stessimo progettando una applicazione per Quiz, e volessimo controllare se l'utente ha risposto correttamente oppure no al quesito, per levarli o aggiungerli punti dal punteggio finale. Utilizzando i metodi precedentemente visti, sarebbe impossibile determinare quando l'utente da una risposta corretta oppure una sbagliata. Dobbiamo quindi introdurre un altro metodo, detto blocco condizionale, formato dai seguenti elementi: if, l'elif e l'else. Come funzionano quest'ultimi? Vediamo un po' più nello specifico:

- If: Viene posto come prima condizione rispetto alle altre. Ciò vuol dire che, appena creato un blocco condizionale, non potremmo trascrivere per primo un elif oppure un else. Il suo scopo sarà controllare la propria condizione in maniera costante all'interno del blocco di codice delineato. Se quest'ultimo, primo degli altri if, sarà vero, allora verrà svolto, altrimenti se si presentano altri if prima di quest'ultimo veri, all'ora questo if non verrà svolto. Non è una buona pratica, comunque (eccetto condizioni specifiche) andare a trascrivere più if nello stesso blocco di condizione.
- Elif : a differenza dell'if, non controlla la condizione in maniera costante, bensì controlla la propria condizione se, e solo se, la condizione di livello superiore risulta falsa. Ciò vuol dire che se anche un if risultasse vero sopra di lui, il programma non farebbe neanche il controllo della condizione di quest'ultimo, differentemente dall'if.
- else: Infine, il compito dell'else, sarà quello di venir eseguito solo e soltanto se tutte le condizioni presenti nei vari elif o nei vari if risultano false. A differenza di quest'ultimi, di else se ne può mettere solo uno a blocco di condizione.

Ecco un esempio di programma che richiede l'utilizzo delle condizioni:

```
vocali = "aeiou"
V = input("inserisci una lettera dell'alfabeto: ")
if V in vocali:
   print("la tua lettera è una vocale")
else:
   print("la tua lettera non è una vocale")
```

All'interno di questo esercizio, possiamo già vedere concetti importanti analizzati: per prima cosa, andiamo a dichiarare una variabile, detta vocali, successivamente andiamo a creare un "input di vocali", e poco dopo andiamo a controllare il valore in entrata. Possiamo qua gia vedere l'utilizzo di una parola chiave importante: "IN". Quest'ultima avrà il compito di determinare la presenza o no di una variabile o di un carattere o di un numero, in una lista (generica, intendendo quindi anche stringhe, tuple, ecc…). Possiamo utilizzare il comando IN assieme al NOT, per svolgere l'operazione inversa: al posto di controllare se è presente, controlliamo se non è presente.

Operatori matematici e le indentazioni

Come abbiamo visto con le condizioni, possiamo quindi far si che, all'interno del nostro programma, se si dovessero presentare due strade possibili da prendere, possiamo decidere quale in base ad un espressione ritornante un valore booleano. La domanda è pero la seguente: come possiamo andare a trascrivere correttamente un if, elif ed else, se sappiamo solo che dobbiamo scrivere la corretta parola chiave e la cosiddetta condizione? Per far cio, avremo bisogno di utilizzare i due punti (":") visti precedentemente. Subito dopo, obbligatoriamente, dovremmo anche indentare il nostro codice, al fine che quella determinata riga venga svolta solo se avviene la condizione ad essa collegata. Perché svolgiamo questo tipo di operazione? In realtà, si poteva già notare dall'esempio grafico prima descritto; comunque senza l'indentazione, il nostro codice non potrebbe funzionare, e vedremo meglio anche dopo come quest'ultima sarà vitale per il nostro programma.

Abbiamo pero detto che parleremo anche di operatori matematici. In python quest'ultimi possono essere utilizzati per tutti i tipi di variabili: sia che quest'ultimi siano le variabili da noi viste precedentemente, sia che siano contenitori di variabili (che andremo a vedere meglio tra poco). Come si suddividono però quest'ultimi? Nelle seguenti categorie:

MATEMATICI CONFRONTO LOGICI

Descriviamo adesso ognuno di quest'ultimi:

- MATEMATICI: all'interno degli operatori matematici possiamo trovare la addizione (+), la sottrazione (-), la moltiplicazione(*), l'elevamento a potenza(**), la divisione con resto(/) ,la divisione senza resto(//) e il modulo della

divisione(%). Tutti questi operatori matematici che sono appena stati descritti sono primitivi, e permettono di svolgere tutte quelle operazioni necessarie su una variabile numerica.
- DI CONFRONTO: lo scopo di quest'ultimi, invece, e quello di andare a controllare due variabili tra di loro, determinando se sono uguali, diverse, maggiori… Proprio per questo motivo, noi vediamo gli operatori di confronto sopratutto rappresentati all'interno dei blocchi di condizione. Come sono formati quest'ultimi? Ecco un esempio:

CONTROLLO + =

Questa sintassi, a prima vista, può sembrare strana. Cosa sta a significare tutto ciò? Vuol dire che all'interno del nostro codice, quando dobbiamo successivamente andare a trascrivere questo controllo, per prima cosa andremo a mettere il TIPO di controllo (es, = sarà uguale, ! Sarà diverso, < sarà minore, ecc…) e successivamente esprimiamo l'uguale per andare a collegarlo ad un altra variabile. Mentre per il simbolo = e ! è strettamente obbligato l'utilizzo del secondo uguale, per il < e il > no, visto che quest'ultimi presentano comunque un controllo (di grandezza) anche senza l'uguale. Questo tipo di confronto, se unito agli operatori matematici, può andare a formare un tipo di operazione detta "auto-operation", che adesso approfondiamo.

Con la "auto-operation", quando scriviamo un operazione del genere: variabile += 1; e come se stessimo scrivendo variabile = variabile + 1; quindi una sottospecie di self-assignment, con l'aggiunta di un valore dato da un operazione matematica.
- LOGICI: Per finire abbiamo invece gli operatori logici, ecco i più importanti descritti di seguente:

AND OR NOT

Rispettivamente, AND ritornerà (all'interno di un espressione booleana) True, se tutte le altre condizioni sono vere, OR ritornerà True se almeno una delle altre condizioni espresse è vera e NOT ritornerà il contrario del suo valore booleano. Bisogna difatti fare attenzione all'utilizzo di queste ultime, visto che mettendo in un blocco condizionale prima un OR che un AND, potrebbe risultare un errore.

Una aggiunta che potremmo dire in capitolo sono gli operatori BITWISE. Che cosa sono e come funzionano? Quest'ultimi hanno il compito di effettuare operazioni logiche su dati rappresentati da cifre binarie. Possono per di più essere utilizzati solo con i tipi interi nel programma. Andiamo adesso a vedere ciascuno:

- (-x): provoca una inversione dei bit all'interno dell'intero, se di partenza avessimo avuto 1100, in uscita avremmo 0011;
- x & y: effettua l'operazione AND tra i bit. Ciò vuol dire una "moltiplicazione" tra quest'ultimi;
- x | y: effettua l'operazione OR tra i bit. Ciò vuol dire una "somma (a 2)" tra quest'ultimi;
- x ^ y: effettua l'operazione XOR tra i bit, Cioè vuol dire una "somma a due (XOR version) tra quest'ultimi;
- x >> y: sposta i bit a destra di posizione. Se avessi 1001, diverrebbe 0100;
- x << y: sposta i bit a sinistra di posizione, se avessi 0100, diventerebbe 1000;

ovviamente per poter utilizzare quest'ultimi, bisogna avere una conoscenza approfondita delle operazioni booleane e delle lavorazioni dei bit, qualita non richieste nel linguaggio python.

P.S. In Python esistono altri due operatori, che sono detti operatori di appartenenza e di esistenza, che adesso andremo ad analizzare. Per prima cosa, parliamo degli operatori di appartenenza. Il compito di quest'ultimi sarà determinare se un elemento è, o non è presente all'interno di un "contenitore di variabili". Potrebbe essere "famosa" a sua sintassi, in quanto sono soltanto due:

in not in

Per quanto riguarda gli operatori di esistenza, hanno un compito differente. Difatti servono a determinare quando un elemento è o non è un altro elemento all'interno del programma. Anche qui la loro sintassi potrebbe essere famosa, in quanto ne esistono soltanto due:

is not is

Una peculiarità che possiamo notare, è il fatto che questi operatori sono accompagnati dall'operatore booleano not, che abbiamo precedentemente visto. Detto quindi tutti gli operatori presenti, passiamo adesso a i "contenitori di variabili".

DIVERSI TIPI DI VARIABILE: I "CONTENITORI"

Programmando in python, ci siamo sbizzarriti utilizzando diversi tipi di variabili, le quali abbiamo visto precedentemente. Dobbiamo comunque capire, che all'interno del nostro codice, non utilizzeremo soltanto variabili che permettono di contenere un valore singolo, ma bensì "variabili" che permettono di contenere più valori a loro volta. I "contenitori" a cui sto facendo riferimento sono i seguenti, ai quali faremo pure tra poco un approfondimento:

- Le liste (list): Un insieme di oggetti mutabili eterogenei;
- Le tuple (tuple): Un insieme di oggetti immutabili omogenei;
- I Dizionari (dict): Un insieme di oggetti chiave-valore;
- I Set (set): Un insieme di oggetti unici;
- I frozen set (set): Un insieme di oggetti unici immutabili.

Partiamo descrivendo la prima. Le liste:
le liste, come detto precedentemente, sono dei "contenitori" mutabili eterogenei. Ciò vuol dire che all'interno di quest'ultimi possiamo contenere valori diversi tra di loro, che non sono strettamente collegati dal loro tipo (il type). All'interno di una lista potremmo avere dei numeri interi, dei reali, delle stringhe, dei booleani, ecc… Ma la vera domanda è: come possiamo crearle? Come possiamo modificare, visualizzarle, ecc…? Per prima cosa, per andare a creare una lista basterà utilizzare la seguente sintassi:

```
VARIABILELIST = []
VARIEBILELIST = [1, 2, 3, 4, 5]
```

Adesso, nel primo caso, ciò che siamo andati a fare è stato semplicemente creare una lista vuota, mentre nel secondo caso, sono andando a create una lista che al suo interno presentava già degli elementi, semplicemente dividendoli utilizzando una virgola. Ricordo che all'interno della lista che ho creato, avrei anche potuto inserire stringhe, valori booleani, byte, ecc… Dopo aver capito quindi come possiamo andare a generare una lista, andiamo a vedere tre metodi importanti per accedere ad un elemento (o una serie di elementi) presenti all'interno di una lista:

```
VARIABILELIST[0]
VARIABILELIST.INDEX(5)
VARIABILELIST[1:2:1]
```

I tre metodi che abbiamo visto, rispettivamente, servono per ricavare un valore ad una determinata posizione, ricavare la posizione di un determinato valore e, per finire, ricavare una serie di valori a certi intervalli. Ecco una spiegazione più approfondita:

- Con il primo esempio, stiamo ricavando un elemento presente all'interno della lista in posizione 0. Se mettessimo numeri superiori all'indice della lista otterremmo un errore, almeno che non inseriamo valori negativi; in quel caso, la lista andrebbe a prendere i valori "al contrario"
- Con il secondo esempio, stiamo invece stampando la posizione di un elemento presente all'interno della lista. Se l'elemento non sarà presente, otterremo ovviamente un errore. Dovremmo quindi prima verificare la sua presenza (utilizzando un blocco di condizione, per esempio).
- Per finire, l'ultimo metodo, ha il compito di ricavare una "serie di elementi", che saranno contenuti da l'indice di partenza 1 (incluso) fino all'indice di arrivo 2 (escluso) con un intervallo di raccolta pari ad 1. I valori di inizio e di fine, se non inseriti, si riferiranno sempre all'inizio e alla fine della lista, rispettivamente, inclusi. Il valori di intervallo, poi, non è obbligatorio, ma si possono fare esercizi carini del tipo [::-1] per controllare se una parola sia palindroma. Perché? Perché come detto prima, essendo non specificati inizio e fine, noi partiremo dall'indice 0 della parola, fino alla sua fine (inclusa), e successivamente andremo a prendere gli elementi con raccolta pari a -1 (cioè al contrario). Questo metodo prende il nome specifico di slicing, diversamente dagli altri (indexing).

Dopo aver visto questi tre metodi principali, analizziamo quelli più specifici. Possiamo infatti scrivere:
- List.append(VAL) Per aggiungere un elemento alla fine della nostra lista;
- List.insert(POS, VAL) Per aggiungere un elemento nella nostra lista ad una determinata posizione;
- List.pop(POS) Per rimuovere un elemento della lista ad una certa posizione;
- List.remove(VAL) Per rimuovere un certo elemento da una lista;
- List.extend(List) Per estendere una lista dentro un altra lista già esistente;
- List1 + List2 ([] + []) Per "sommare" due liste, molto simile al metodo extend;
- List.sort() Per riordinare gli elementi di una lista in ordine crescente;
- Len(List) Per controllare il numero di elementi presenti all'interno di una lista;

- List.reverse() Per riordinare gli elementi di una lista in ordine decrescente.

Questi sono solo alcuni dei molti metodi che possono essere utilizzati per le liste, ma permettono comunque di lavorare facilmente e velocemente su queste ultime. Passiamo adesso al prossimo contenitore, le tuple.

Il vantaggio maggiore delle tuple, sarà quello di poter andare a lavorare su valori omogenei non modificabili. Ciò vuol dire che tutti i metodi che abbiamo visto precedentemente con le liste che permettevano di modificare queste ultime, non potranno essere utilizzate nelle tuple. Come possiamo andare invece a creare una Tupla?

```
VARIABILETUPLA = 1, 2, 3, 4
VARIABILETUPLA = (1,)
VARIABILETUPLA = ()
```

Con il primo metodo dai noi utilizzato, stiamo andando a creare una tupla che contiene quattro elementi. Il fatto che non abbia le parentesi e una cosa che può spesso confondere, difatti, come nell'esempio successivo, specifichiamo le parentesi per identificare la tupla (anche se in Python questa cosa non è necessaria). Ricordati che, come si vede anche nell'esempio, se la tupla contiene un solo elemento, dovrà possedere una virgola dopo. Per finire, con l'ultimo esempio andiamo invece a generare una tupla vuota. Molto spesso questi contenitori vengono utilizzati per andare a contenere valori successivamente utilizzati in una grandezza di uno schermo, di una figura o i corretti valori di un colore RGB; come, per esempio, nel modulo PyGame che andremo meglio ad analizzare successivamente. Essendo che non ce altro da dire sule tuple, passiamo anche ai Dizionari:

I dizionari sono un sistema detto chiave-valore, il loro compito sarà quindi quello di collegare una chiave ad un valore ad esso associato, e viceversa. Ma come possiamo andare a creare un dizionario? Vediamolo:

```
VARIABILEDICT = {}
VARIABILEDICT = {1:2, 3: [1, 2, 3, 4], 4: [1, 2: [1, 2]]}
```

Vediamo che il dizionario avrà una definizione più "dura di quello che ci aspettavamo. All'interno di un dizionario, difatti, se è vuoto, basterà definirlo usando una parentesi aperta graffa e una chiusa. Mentre se il dizionario non sarà vuoto, allora la chiave sarà il primo elemento, il valore il secondo elemento e i due sono divisi da i due punti. Per di più anche una lista, come si vede, può essere un valore, e anche un singolo elemento di una lista può essere una chiave. Tutto questo può confondere no? Ma solo perché ci si trova alle prime armi con questo nuovo elemento, che dobbiamo ancora "digerire". Ecco i metodi che possono essere utilizzati su un dizionario.

- DICT.item() Permette di ritornare tutti gli elementi di un dizionario sotto forma di una tupla;
- DICT.keys() Permette di ritornare tutte le chiavi del dizionario;
- DICT.values() Permette di ritornare tutti i valori di un dizionario;
- DICT.get(chiave, default) Permette di ritornare il valore di una chiave di un dizionario, Se non presente restituirà None;
- DICT.pop(chiave, default) Permette di eliminare il valore di una chiave di un dizionario, se non presente restituirà None;
- DICT.popitem() Rimuove e restituisce un elemento dal dizionario, determinata la posizione;
- DICT1.update(DICT2) Aggiunge tutti gli elementi di un dict1 in un dict2 (come un extend per le liste);
- DICT.copy() Permette di copiare e restituire un dizionario uguale a quello voluto;
- DICT.clear() Rimuove tutti gli elementi di un dizionario.

Chiudiamo adesso questo paragrafo, parlando dei SET/FROZENSET:

I SET/FROZENSET, sono dei contenitori i quali presentano soltanto degli elementi unici al loro interno. Sono molto simili alle liste, con l'unica differenza che quando un SET viene trasformato in un FROZENSET, quest'ultimo non potrà più essere modificato. Ecco la sintassi che ci permette di creare un SET in Python:

```
VARIABILESET = SET()
VARIABILESET =  {1, 2, 3, 4}
```

Differentemente dai contenitori da noi visti precedentemente, nel nostro SET possiamo andare soltanto ad inserire degli elementi unici, cioè non duplicati. Adesso, se volessimo rendere uno di quest'ultimi un FROZENSET, al posto di inizializzarlo scrivendo quest'ultimo come set, basterà trascriverlo come FROZENSET (non modificabile), oppure convertirlo successivamente con il metodo FROZENSET(SET). Quali sono i metodi che possiamo utilizzare in un set?

- SET.add(el) Permette di aggiungere un elemento al SET;
- SET.remove(el) Permette di rimuovere un elemento dal SET
- SET.discard(el) Permette di rimuovere un elemento dal SET, ma differentemente da remove, prima controlla se presente senza dar cosi un errore.
- SET.pop(pos) Permette di rimuovere un elemento da un SET ad una certa posizione;
- SET.copy() Permette di creare e restituire un SET uguale a quello voluto;
- SET.clear() Permette di rimuovere tutti gli elementi di un SET.

Il metodo copy, per di più, è solo sopportato nel FROZENSET. Gli ultimi metodi da noi non visti, sono poi il min() e il metodo max(), che permettono di ritornare il valore minore o maggiore di questi "contenitori".

CICLI ITERATIVI:

In Python, non basta conoscere quali variabili possiamo utilizzare, o come modificare queste ultime. Se un giorno ci chiedessero di scorrere lungo tutti gli elementi di una lista, non sapremmo farlo, perché ci mancherebbe una conoscenza essenziale: I cicli iterativi. I cicli iterativi si suddividono in due categorie, che adesso andremo a visualizzare:

WHILE FOR

- Cicli iterativi while: Quando utilizziamo un ciclo iterativo while, sappiamo che stiamo lavorando con iterazioni non costanti(nelle quali, di conseguenza, sappiamo il numero di cicli), bensì dinamiche. Proprio per questo, il ciclo while, dopo essere stati richiamato, richiede una valore booleano per determinare quando quest'ultimo deve interrompersi. Questo valore potrà essere una variabile booleana (es. cicle = True), oppure potrà essere un uguaglianza tra due numeri (es. x == 1); l'importante è capire questo. Appena l'uguaglianza è Falsa, il ciclo finisce. Ecco un esempio:

```
cicle = True
x = 1
while cicle:
    x += 1
    if x = 100:
        cicle = False
```

appena la variabile x arriverà a 100, la variabile che "manteneva in vita" il ciclo while "morira", e di conseguenza quest'ultimo uscirà e passerà alle prossime linee di codice.

- Cicli iterativi for: lo scopo di quest'ultimo, a differenza, e invece lavorare con un numero di iterazioni ben definite. La sintassi del for, a differenza di quella del while, essendo anche più complessa, la rappresenteremo qui:

```
for [iteratore] in range(start, end, step):
for [iteratore] in [list, string, ecc…]
```

I due for qui presenti sono il for x in range (classico for) e il for-each. Lo scopo del primo sarà quello di svolgere un determinato numero di iterazioni, che iniziano da START, finiscono da END (non compresso) con passo STEP. L'iteratore, in questo caso, sarà una variabile che terrà traccia di questi valori. Nel secondo for, differentemente, non andiamo invece a prendere come valore in iterazione un valore numerico, ma bensì gli elementi contenuti all'interno di una lista, di una stringa, ecc… Questi sono i due tipi di for più semplici e basici da imparare. Per quanto riguarda for più complessi che richiedono la funzione reverse, oppure enumarate, gli approfondiremo meglio dopo. Per quanto riguarda il for x in range, possiamo immaginare i tre valori contenuti tra le parentesi come lo slicing visto precedentemente, per capire meglio il suo funzionamento.

LE STRINGHE

Strano pensarlo, ma abbiamo bisogno anche di approfondire le stringhe. Perché? Le stringhe, difatti, sono come le "variabili contenitore" che abbiamo visto prima. La differenza principale, però è che quest'ultima viene immagazzinata all'interno di una variabile. Perché diciamo quindi che è una "variabile contenitore" anche se, difatti, può essere immagazzinata in una singola variabile? La motivazione, purché semplice, non presenta una risposta in Python. Dobbiamo quindi fare un piccolo approfondimento in Java:

"In Java le stringhe non sono un tipo primitivo, bensì wrapper. Ciò è dato perché le stringhe sono formulate da una sequenza di caratteri, detti character, che vanno poi a formare l'intera stringa"

Possiamo quindi intuire, che i metodi visti precedentemente (ma solo in parte) possono essere usati sulle Stringhe. Essendo che i metodi effettivi sono molti, cerchiamo di elencare i più importanti:

- Stringa.split(flag) = Permette di suddividere una stringa in una lista di stringhe, partendo da una stringa iniziale e suddividendola secondo una flag da noi decisa. Se il flag dovesse essere vuoto, allora la stringa dividerà ogni elemento della frase dopo ogni spazio;
- Stringa.lower() = Rende la stringa di partenza tutta in minuscolo;
- Stringa.upper() = Rende la stringa di partenza tutto in maiuscolo;
- Stringa.capitalize() = Rende la stringa con la prima lettera maiuscola (capitalizzata);
- Stringa.index("el") = trova la posizione iniziale di una lettera o una sotto stringa in una stringa;
- Stringa.find("el") = Molto simile all'index, ritorna la posizione dell'elemento;
- Stringa.count("lettera") = conta quante volte una determinata lettera (o sotto stringa) è presente nella stringa iniziale.

Questi, però, sono solo metodi che possiamo andare ad utilizzare per modificare direttamente una stringa. Cosa altro possiamo fare con queste ultime? In primis, possiamo concatenarle tra di loro utilizzando il simbolo "+", e moltiplicarle una serie di volte utilizzando l'operatore "*" (non ne sono accettati altri di operatori, tranne quest'ultimi). Ecco alcuni esempi so ciò che abbiamo appena detto:

```
"Hello" + " World" = "Hello Word"
"Hello" * 3 = "HelloHelloHello"
```

Se dovessimo poi stamparle, possiamo anche andare ad utilizzare il metodo F-STRING. Secondo quest'ultimo, infatti, andremo a trascrivere tutto (dentro un print, per esempio) all'interno delle virgolette, e successivamente, posizionando una F prima dell'apertura delle virgolette, andremo a mettere in parentesi graffe le variabili poi da stampare. Questo metodo permette di rendere il codice più leggibile, ma anche veloce.

```
CiaoMondo = "World"
PRINT(f"Hello {CiaoMondo}")
```

Di metodi sulle stringhe ne esistono moltissimi, ma non gli approfondiremo tutti. Concentriamoci piuttosto sul prossimo capitolo: La list/ser e dictionary comprehension.

LIST/SET E DICTIONARY COMPREHENSION

Quando andiamo a lavorare su questi contenitori di variabili, esistono dei metodi molto facili e veloci che ci permettono di modificare gli elementi presenti al loro interno. Se per esempio dovessimo andare a far si che otteniamo, da una lista di partenza, i quadrati di tutti i numeri contenuti al suo interno, potremmo (con le conoscenze da noi possedute adesso) immaginare il programma in questa maniera:

```
StartList = [1, 2, 3, 4, 5]
NewList = []
for x in StartList:
        NewList.append(x**2)
```

Ma, quale è il problema? Il risultato ottenuto sarà sempre lo stesso. Pensandoci bene, però, sono molte righe di codice, soltanto per andare a modificare una lista. Non esiste quindi un modo più semplice per andare a modificare questi elementi, senza dover scrivere così tanto? Ecco che introduciamo quindi, l'argomento vero è proprio. Prima di tutto, per comprehension, in un programma, si intende lo "scorrimento" degli elementi di un oggetto iterabile, allo scopo di modificare o ricavare un "qualcosa", un "obbiettivo". La sintassi per la List e il Set comprehension è la seguente:

VariabileSet/VariabileList =
[MODIFICA + CICLO ITERATIVO FOR + CONDIZIONE]

Vedere questa sintassi, cosi di primo impatto, potrebbe però dare confusione. Difatti cosa stiamo andando a fare? Presa una lista,o un set, per prima cosa stiamo andando a svolgere una MODIFICA: la modifica in questione, non sarà limitata (almeno che non porti errori nel programma) e potrà essere quindi, per esempio, un operazione matematica, il richiamo di un'altra funzione, di un metodo, una procedura, ecc… Abbiamo poi il CICLO ITERATIVO FOR: ovviamente, il compito di quest'ultimo sarà quello di andare a prendere gli elementi presenti all'interno della lista da cui poi svolgere la cosiddetta modifica. Per finire abbiamo poi la CONDIZIONE: servivrà a determinare quali elementi verranno presi in considerazione. Anche se la condizione non è strettamente necessaria, gli altri due componenti dovranno essere obbligatoriamente presenti. Ecco un esempio di List/Set comprehension che potremmo difatti trovarci davanti.

variabileList = [1, 2, 3, 4, 5]
variabileList = [x**2 for x in variabileList if x != 5]

Quello che faremo, sarà quindi assegnare alla variabileList tutti i quadrati presenti al suo interno (eccetto per il 5), senza dover andare cosi una nuova lista, e avendo pure eliminato i vecchi elementi (visto che lavoriamo direttamente su di loro). È importantissimo ricordare che non potra essere svolta questa pratica se la lista di partenza è vuota. Per quanto riguarda la Dictionary comprehension, dobbiamo invece andare ad analizzare meglio la sintassi:

VariabileDict =
[Chiave: Valore for VAR(chiave, valore) in VarList]

La differenza, di conseguenza, e il fatto che nella Dict comprehension andiamo a assegnare un sistema chiave-valore, prendendo gli elementi da un determinato iterabile. La condizione è stata opportunamente omessa per non creare confusione nella formalizzazione di quest'ultimo. Passiamo adesso ai metodi e alle classi.

METODI E CLASSI

Non faremo tanto un approfondimento tecnico su quest'ultimo argomento, essendo che possiamo già trovarlo nelle informazioni presenti nel quaderno di Java. D'altra parte, infatti, andremo a vedere più un aspetto pratico, per utilizzarlo in maniera concreta nel nostro programma. Prima di tutto, cosa sono i metodi? Ecco una definizione accurata di quest'ultimi:

"con metodi, intendiamo l'insieme di procedure e e funzioni. I primi non ritorneranno nessun valore, mentre i secondi ritorneranno un valore che successivamente dovrà essere immagazzinato"

Il vantaggio, in Python, sarà proprio il fatto che, la differenza tra funzioni e procedure è inesistente, visto che non è necessario andare a specificare il valore di return. Ma come possiamo andare a richiamare un metodo in python? Vediamolo:

def identificatore(parametro1, parametro2, ecc…):

Da questa rappresentazione possiamo già capire una serie di cose. In primis, la keyword utilizzata per identificare la funzione prenderà il nome di "def". Per seconda cosa, utilizzeremo i due punti per andare ad indentare i valori e le procedure all'interno della funzione, che successivamente dovranno essere utilizzati, e per terza cosa, non siamo obbligati a specificare il tipo dei parametri. Ok, detto questo, però, quale è lo scopo nell'utilizzo delle funzioni? Lo scopo di queste ultime è quello di immagazzinare parti di codice che successivamente dovranno essere utilizzate più volte, permettendo così di non ripeterle nel codice. Facendo un esempio; se mi trovassi all'interno di un videogioco, e dovessi richiamare la intro di quest'ultimo più volte, non sarebbe scomodo riscriverla ogni singolo momento? Adesso, anche se è vero che non è necessario specificare i valori di entrata e di return, possiamo comunque farlo, utilizzando il seguente metodo:

def identificatore(parametro1: tipo) → valore_di_return:

Utilizzando questo metodo, posso quindi andare a specificare il tipo dei valori in entrata, facendo si che non ne siano accettati altri, ma posso anche andare a specificare il valore in uscita. Adesso, per quanto riguarda il tipo, possiamo scrivere direttamente i classici da noi conosciuti (per esempio, str, int, bool, ecc…) oppure possiamo andare a specificare anche il contenitore che immagazzinerà questi tipi (per esempio, List[int] vorrà significare una lista di interi. Adesso, quale è il vantaggio nell'utilizzo delle classi? Il compito delle classi sarà quello di rendere il codice più leggibile e fruibile per ricollegarsi alle informazioni presenti. La sintassi per una classe è molto semplice, ed è la seguente:

class identificatore:

Dove dopo andremo ad indentare tutti i metodi presenti al suo interno. Solitamente, però, quando si lavora con le classi, si preferisce utilizzare dei metodi specifici, che adesso andremo ad analizzare, detti metodi speciali; che sono caratterizzati dalla presenza di due underscore iniziali e finali.
Prima di tutto, i metodi speciali, detti anche metodi jolly, underscore method, e cosi via, gli conosciamo già, circa. Anche se non sono parte attiva della nostra programmazione, comunque lavorano dietro le quinte, andando a svolgere tutte le funzioni che gli richiediamo. Facciamo un esempio. Quando in Python vogliamo andare a convertire un numero sotto forma di stringa, andiamo a scrivere str(numero), giusto? Scrivere ciò, però, equivale a scrivere numero.__str__(). Cosa cambia quindi? Effettivamente, tutte e due rappresentano la stessa funzione. La differenza fondamentale, però, è che il metodo str() è quello proposto a noi developer per richiamare il comando, e il metodo __str__ (anche se da noi lo stesso utilizzabile) è la controparte dietro le quinte di ciò che noi abbiamo scritto. Quando stiamo utilizzando questi metodi particolari, quindi, dobbiamo immaginarli come "funzioni già fatte", il quale avranno un compito specifico. Vediamo alcune:

- __init__() = avrà il compito di inizializzare le variabili che successivamente dovranno essere utilizzate all'interno della classe. Il vantaggio sarà quindi ricollegare tutte le variabili ad una keyword detta self, che permetterà poi un richiamo veloce e semplice. È consigliato, però, non variare il nome delle variabili quando si fa il collegamento self-valore;
- __str__() = hai il compito di ritornarci una stringa leggibile dell'oggetto su cui stavamo lavorando all'interno della nostra classe. Il vantaggio è il seguente: se un qualcosa deve essere visualizzato da un utente, non potrà farlo almeno che quest'ultimo non sarà convertito sotto forma di stringa. E non possiamo utilizzare la classica tecnica str(), bensì dovremmo creare un metodo con questo specifico lavoro.
- __repr__() = è molto simile al metodo string, la unica differenza sarà quella che al posto di inviare un messaggio "visualizzabile agli utenti", potrà essere invece utilizzato per gli sviluppatori, sopratutto durante la fase di debug, per visualizzare problemi nel codice
- __add__ () = Permette di addizionare due valori che successivamente dovranno essere ridati da una classe:

Ricordiamo che di questi "metodi magici" ne esistono moltissimi, e noi ne abbiamo visto solo una parte, meglio dire, i più importanti. Bisogna anche ricordarsi una cosa: Quando si lavora con questi tipi di metodi, daranno sempre un tipo di return, obbligatorio, e successivamente, a nostra decisione, potremmo immagazzinarlo, richiamando correttamente il metodo.

Abbiamo parlando in maniera approfondita di come andare a creare metodi e classi, ma come possiamo effettivamente importarli? Parliamone

Import

Nella nostra vita da programmatore Python non potevano mancare loro: gli import, una parte essenziale nella scrittura del nostro codice. Quale è però il loro compio? Il loro compito sarà quello di richiamare librerie esterne a patto di semplificare il codice che stiamo scrivendo, richiamando delle funzioni utili, quali possono essere i numeri random, generatori di password, e cosi via. Ma quale è la sintassi vera e propria? È la seguente

import libreria
import libreria as nome
from libreria import sottomodulo
from libreria import sottomodulo as nome

grazie a questo metodo, potremmo quindi avere accesso ai vari metodi delle librerie, le quali siamo obbligati precedentemente ad installare. Adesso, possiamo anche andare a creare noi delle nostre librerie? Si è possibile, utilizzando proprio le classi che abbiamo precedentemente analizzato. Basterà semplicemente creare i metodi che dovranno essere importati, e salvarli all'interno di una classe. Successivamente, dovremmo andare ad aggiungere una linea di codice con suscritto <if __name__ = "__main__"> per andare a determinare cosa fare nel caso questo modulo venga aperto come eseguibile (.exe). Dopo aver fatto ciò, non ci resta altro che Richiamare il nome del file da noi appena creato (come se

fosse una libreria) e utilizzarlo a nostro piacimento. Adesso, per poter convertire un modulo (o un file py) in eseguibile, è preferibile utilizzare metodi come auto-py-to-exe, dove online possono essere trovate tutti i passi per l'installazione per il converter. Adesso che abbiamo finito di parlare di moduli, parliamo della gestione delle eccezioni.

Try and Except

Quando stiamo lavorando al nostro codice, potrebbe capitare che, all'avvenire di un problema, l'intero programmi si interrompa per colpa di un errore. Adesso, noi come programmatori, non vogliamo che il programma si blocchi totalmente, ma che piuttosto mandi un messaggio, un avviso, che si sia presentato un errore durante lo svolgimento di una determinata azione. Per far ciò, di conseguenza, possiamo introdurre il concetto del blocco di codice try and except. Adesso, basta immaginare che nel blocco try, andranno tutte le procedure che dovranno essere svolte, e alla presenza di un determinato errore, il programma dovrà "prenderlo", uscire dal blocco try, e seguire il blocco except. Vediamo un esempio.

```
Try:
    Array[10] = 100
Except IndexError:
    Print("Numero troppo grande")
```

Capiamo quindi che, se non dovesse avvenire correttamente l'esecuzione del blocco try, quella del blocco except inizierebbe. È importantissimo ricordarsi l'indentatura, ma sopratutto i tipi di errore che potrebbero avvenire durante l'esecuzione del blocco di codice. Ecco quelli più comuni

- IndexError = Si sta cercando di accedere ad una lista, tupla, ecc… selezionando un indice errato;
- SyntaxError = Errore sintattico durante la lettura del codice;
- IndentationError = Errore di indentatura presente nel codice;
- EOFError = Errore che si presenta quando il valore di input "tocca" la fine del file;
- ImportError = Errore che si presenta quando non si importa correttamente un modulo;
- KeyError = Errore che si presenta quando si tenta di accedere ad una chiave non esistente in un dizionario;
- NameError = Errore che si presenta quando non è stato sbagliato o non trovato il nome di una variabile;
- TypeError = Errore che si presenta quando si prova a richiamare una operazione tra valori "incompatibili"

Potremmo immaginare tutto questo come un ciclo con un break (visto che non approfondito il suo funzionamento in questo quaderno, osservare quello di Java). Adesso, non dovendo dir altro in questo argomento, passiamo finalmente alle funzione avanzate in python

TECNICHE AVANZATE NEL LINGUAGGIO

1 PARTE: APPROFONDIMENTO SULLE FUNZIONI

Mettiamo di dover andare a lavorare su una funzione, ma il compito che quest'ultima deve svolgere è facile, corto, ecc.. e non abbiamo bisogno quindi di funzioni complesse, come quelle viste prima, ma di una funzione basilare, semplice e veloce come le procedure viste nella list comprehension. Per farci risparmiare righe di codice, di conseguenza, possiamo utilizzare una funzione particolare, la cui sintassi è la seguente:

```
variabilefunc = lambda x, y: x**y
```

Da questa linea di codice, possiamo già apprendere molto: Per prima cosa, ciò che conterrà il valore di return della nostra funzione sarà direttamente una variabile. Dopo aver richiamato la keyword lambda, basterà inserire tutte le variabili che devono essere utilizzate nel processo, e successivamente descrivere (dopo i due punti) il tipo di operazione che dovrà essere eseguita. Adesso, questo esempio potrebbe essere immaginato in questa maniera:

```
Esponenziale = lambda x, y: x**y
Esponenziale(2, 2)
```

In questa maniera, stiamo richiamando la nostra funzione esponenziale con due valori in entrata, il due e il due. Il risultato finale sarà dato quindi dall'operazione matematica descritta all'interno della funzione lambda, che in questo caso sarà

2**2 (cioè due elevato alla seconda). Della funzione lambda non ce tanto altro da dire, ma molto simile, e comunque importante, è anche il metodo MAP. Differentemente dalle altre funzioni che abbiamo visto, il metodo Map non ha il compito di creare un ulteriore metodo, bensì di svolgere una modifica ad un oggetto iterabile, attraverso l'utilizzo di una funzione. Facciamo un esempio:

```
variabile = list(map(int, input().split()))
```

Cosa stiamo effettivamente facendo con questo metodo map, di conseguenza? Per prima cosa, stiamo richiamando la funzione int(), quale compito sarà quello di convertire i valori in entrata sotto forma di numeri interi, e successivamente, visto che introduciamo il metodo input().split(), è come se l'iterabile lo stessimo creando noi. Di conseguenza, la lista non è effettivamente "creata", bensì si crea nello stesso momento in cui andiamo a premere "invio", e successivamente la funzione map andrà a convertire tutti gli elementi come interi. Grazie al metodo list, per finire, convertiamo il tutto come una lista. La sintassi generale del metodo map, però, è la seguente:

```
map(funzione, iterabile)
```

Quindi, tutti quei discorsi un po confusi che abbiamo detto sul metodo visto precedentemente, erano dati da questa sintassi: Per prima cosa, dobbiamo implementare la funzione che vogliamo utilizzare, e dopo, dobbiamo andare a introdurre una serie di oggetti iterabili i quali subiranno la modifica della funzione. Proprio per questo avevamo detto all'inizio che la funzione Map fosse simile alla funzione Lambda.

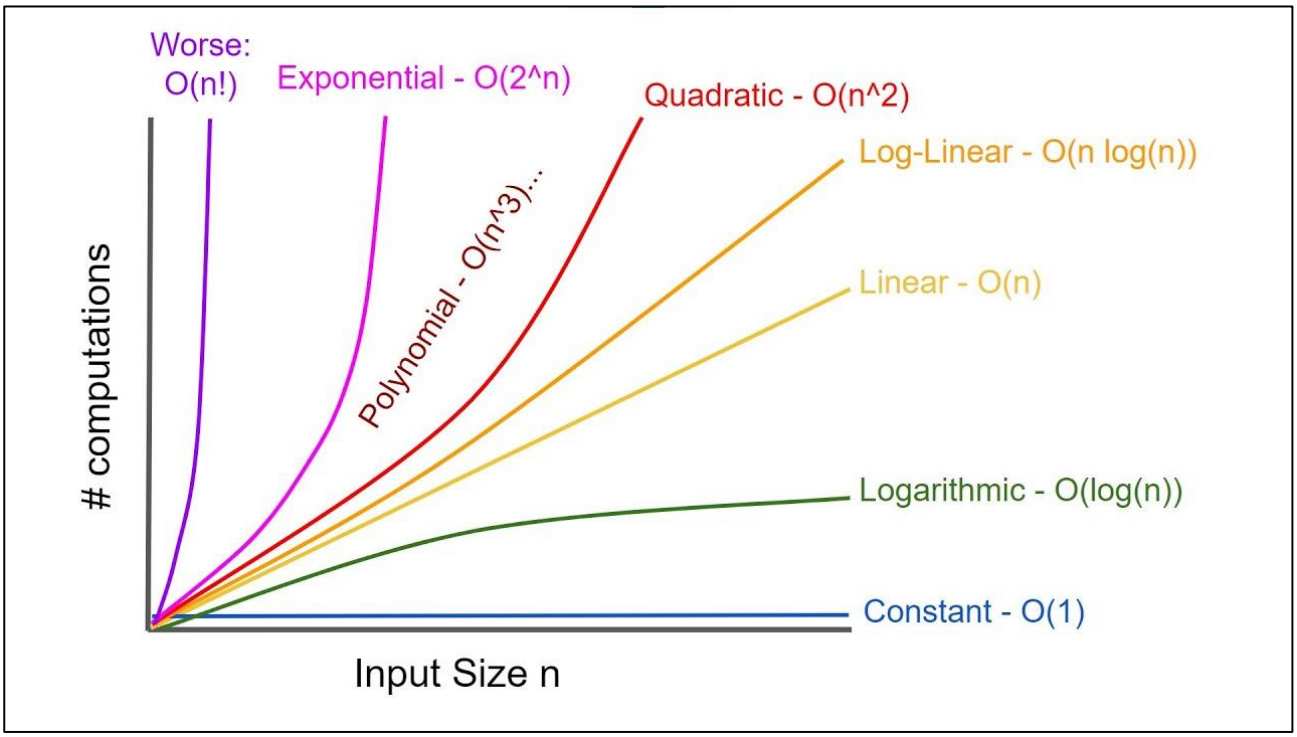
Parlando di funzioni, nel corso della nostra programmazione, potremmo anche incontrare funzioni importanti, come, per esempio, quelle ricorsive. Il loro compito è quello di richiamare al proprio interno o un altra funzione, oppure se stesse, andando cosi a semplificare le linee di codice da noi necessarie. Ecco un esempio:

```
def fact(n: int) → int:
 if n == 1:
        return n
 return n * fact(n-1)
```

Anche se semplice, ci fa capire come una delle pietre miliari delle funzioni (parliamo di funzioni ricorsive, come detto prima) possono essere trascritte all'interno del nostro linguaggio Python. Anche se semplice e veloce, però, dobbiamo fare molta attenzione ad una cosa: Una teoria che "affligge" le funzioni in un programma, che dobbiamo analizzare obbligatoriamente in questo capitolo, essendo strettamente collegato: La RunTime complexity.

2 PARTE: LA RUN TIME COMPLEXITY

Mettiamo di star lavorando alla nostra funzione, quale può essere quella ricorsiva, la lambda, la map, ecc… E, andando ad aumentare i dati da noi inseriti, iniziamo a notare un problema: la velocità con cui il nostro computer svolge le operazioni da noi trascritte inizia a rallentare. Anche se questo può sembrar normale all'inizio , si potrebbero iniziare a vedere i primi problemi subito dopo che la mole di dati aumenta. Tutto questo, perché avviene? Andiamo a capirlo analizzando il grafico della RunTime Complexity:



Cosa vuol dire tutto ciò? Cosa sono queste 0(n), 0(1), ecc…? Per prima cosa, possiamo vedere che questo grafico è formato, sull'asse delle ascisse, dalla grandezza degli input (la mole di dati che viene data in pasto al nostro programma), mentre, sull'asse delle ordinate, possiamo trovare le "computation" (i passaggi) necessari per eseguire l'intero algoritmo. Detto in maniera molto povera, si intende il tempo che ci impiega il programma a essere eseguito. Adesso, cosa centra tutto questo con le funzioni che abbiamo precedentemente visto? Essendo che Python è un linguaggio lento rispetto ad altri, come per esempio Java, dobbiamo fare molta attenzione a ciò che scriviamo e come lo scriviamo, cercando di ridurre al massimo i tempi di esecuzione. Adesso, con 0(n) si intende il numero di passaggi da noi descritti. Per capirci meglio, immaginiamo che il nostro programma sia una funzione con un for dentro, e che come parametro prenda un input n. Adesso, iniziamo a descrivere ognuna di queste RunTime, per capire quando è meglio utilizzarle, perché utilizzarle, e i vantaggi che ci portano.

- O(1) = Con questo tipo di RunTime, indichiamo una operazione singola (o una serie di operazioni singole) che possono essere "eseguite" sul colpo, all'interno del nostro metodo. Mettiamo, per esempio, di dover svolgere una somma tra l'input dato e un numero casuale. Il programma richiederebbe solo un passaggio, sia che l'input dato sia il numero 1 che il numero 10000. Come possiamo vedere, infatti, con questo tipo di notazione, all'aumentare dei dati, non aumenta il tempo. Adesso, ricordiamo (anche per tutte le restanti casistiche), che si rappresenta il caso singolo dell'operazione. Se dovessimo avere una funzione con al suo interno tre somme, non scriveremo O(3), ma sempre O(1). Questo tipo di RunTime è detta costante. Ecco un esempio:

```
def FunCost(n):
    return n + 1
```

- O(log n) = Con questo tipo di run time, invece, indichiamo operazioni solitamente legate al campo della ricerca binaria o della compressione dei dati. Un esempio classicissimo che può essere fatto, difatti, è la compressione di HuffMan. Adesso, come si può notare dal grafico, all'aumentare dei dati, la crescita inizierà piano piano a diminuire, diventando quasi costante. Quest'ultima può, infatti, essere quasi considerata come la RunTime vista precedentemente. Il suo nome è Logaritmica, ecco un esempio:

```
def ricerca_binaria(array, valore):
    minimo = 0
    massimo = len(array) - 1
    while minimo <= massimo:
        centrale = (minimo + massimo) // 2
        if array[centrale] == valore:
            return centrale
        elif array[centrale] < valore:
            minimo = centrale + 1
        else:
            massimo = centrale - 1
```

- O(n) = Questa, a differenza delle altre, è una delle RunTime più comuni che si possono trovare all'interno dei programmi. Il loro funzionamento è semplicissimo: viene svolto un numero di operazioni direttamente determinato dal valore di N. Mettiamo dovessimo andare a lavorare all'interno di un Array di dimensione N, e dovessimo scrollare lungo tutti i suoi elementi, allora avremmo una RunTime complexity di O(n). Adesso, quest'ultima è detta costante, proprio perché il suo tempo di esecuzione è strettamente determinato dalla mole di input e, per di più, forma una proporzionalità diretta (da come si può vedere dal grafico), facendo capire cosi il significato del suo nome. Vediamo adesso un esempio:

```
def ricerca_num(array, valore):
    for x in array:
        if x == valore:
            return True
    return False
```

- O(n^2) = Saltando la funzione logaritmica-costante, presentando pochissime casistiche, passiamo invece alla funzione quadratica. Adesso, quest'ultima può essere solitamente trovata nelle lavorazioni delle matrici, e non presenta quindi delle peculiarità enormi rispetto a i tipi di RunTime visti precedentemente. Iniziamo comunque entrare in tempistiche sempre più lunghe con l'aumentare la mole dei dati. Adesso, andiamo a vedere un esempio, uno tra i più classici:

```
def ricerca_matrix(matrix):
for x in matrix:
    for y in x:
        print(y)
```

- O(2^n) e O(!n) = Queste ultime due tipi di RunTime, che andremo a vedere, sono le più "lunghe" (se possiamo dire, a parità di tempo) rispetto a quelle viste precedentemente. Ci sono veramente pochi casi in cui queste ultime potrebbero e dovrebbero essere utilizzati, e proprio per questo, andiamo solo a darli una definizione generale e superficiale. Per quanto riguarda la esponenziale, possiamo pensare ad un algoritmo di forza bruta, che prova tutte le combinazioni possibili al fine di trovare il risultato da noi voluto, come per esempio, il problema del venditore ambulante. Per quanto riguarda invece la RunTime fattoriale, possiamo pensare alla funzione ricorsiva vista precedentemente (al quale non serve un ulteriore spiegazioni, altrimenti basta guardare il capitolo al di sopra).

Possiamo, per adesso, chiudere il capitolo riguardante le funzioni, e concentrarci adesso, sulle "particolarità" che potremmo trovare nel linguaggio, cioè metodi semplici e veloci per rendere il nostro lavoro più semplice.

3 PARTE: APPROFONDIMENTO SU SINTASSI E SEMANTICA

Una delle prime cose di cui potremmo andare a parlare sono i decoratori. Adesso, che cosa sono i decoratori? E perché li utilizziamo nel nostro codice. Anche se non sembra, i decoratori sono anch'essi strettamente legati al mondo delle funzioni. Ma che compito svolgono? Mettiamo abbiamo una funzione all'interno del nostro programma, della quale non vogliamo e non possiamo andare a modificare la posizione. Adesso, mettiamo che questa funzione debba essere richiamata, successivamente in un altra funzione per aggiungere le operazioni che quest'ultima svolge. Un primo scenario, al quale potremmo già dare la soluzione, sarebbe questo:

```
def funz_aggiuntiva(func):
    def funz_involucro():
        print(f"Sta arrivando il metodo: {func.__name__}")
        func()
        print(f"è finito il metodo: {func.__name}")
    return funz_involucro

funz_base():
    print("ciao!")

funz_aggiuntiva(funz_base)()
```

Scrivendo queste linee di codice, di conseguenza, andiamo a richiamare, in maniera abbastanza macchinosa, la nostra funzione all'interno della nostra funzione. Ma non esiste un altro metodo, più semplice e facile, per far tutto ciò? Andiamo quindi, finalmente, a vedere, come possiamo inserire questi famosi "decoratori", che ci permetteranno di rendere la chiamata più semplice, e l'intero codice leggibile.

```
def funz_aggiuntiva(func):
    def funz_involucro():
        print(f"Sta arrivando il metodo: {func.__name__}")
        func()
        print(f"è finito il metodo: {func.__name}")
    return funz_involucro

@ funz_aggiuntiva
funz_base():
    print("ciao!")

funz_base()
```

Cosa siamo andati a cambiare però? Facendo questa operazione, difatti, siamo andati ad "creare un involucro" sulla nostra funzione di partenza (funz_base). Ciò vuol dire che quando quest'ultima verrà richiamata, non verrà chiamata singolarmente, ma bensì verrà richiamato l'intero blocco di codice wrapper, nella quale è contenuta. Ciò vuol dire che, quando utilizziamo un decoratore, non stiamo facendo altro che ampliare le funzionalità di una funzione di partenza. Cosa succede se però ho bisogno di aggiungere dei parametri? Semplicemente scrivo, al richiamo della funzione (all'interno del codice), *args e **kwargs, per poter inserire un numero "infinito" di parole chiavi e keyword. Ecco un esempio:

```
def funz_aggiuntiva(func):
```

```
def funz_involucro():
        print(f“Sta arrivando il metodo: {func.__name__}”)
        func(*args, **kwars)
        print(f“è finito il metodo: {func.__name}”)
    return funz_involucro

@ funz_aggiuntiva
funz_base():
    return “ciao”

funz_base()
```

faremo nuovamente un accenno di queste ultime quando, nei prossimi capitoli affronteremo librerie e moduli parlando di itertoools e functools. Dopo aver finito quindi con i decoratori, passiamo adesso ad argomenti “più leggeri”. Faremo adesso una lista di “funzioni” integrate in Python che possono semplificare e velocizzare la scrittura del nostro codice. Cominciamo:

PYTHON TRICK

- Se vogliamo andare a rappresentare con più facilità i nostri numeri all’interno del programma, andando a dividere gli zeri coppie di tre (se per esempio avessimo un milione, un bilione, ecc…) possiamo utilizzare il simbolo underscore, in questa maniera: “variabile = 1_000_000_000”. Adesso, se volessimo poi andare a stampare questi valori, tenendo sempre in considerazione la spaziatura, possiamo utilizzare questa linea di codice: f”{val:,}”;
- Se hai bisogno di andare a valutare più possibilità nel tuo codice, utilizzando l’operatore logico Or, ma non vuoi andare a scrivere “troppo”, per esempio: if name == “Sam” or name == “bob” or name == “patrick”, ecc… puoi semplicemente scrivere: if name in [“Sam”, “Bob”, “Patrick”, ec…];
- la funzione eval() in python, permetterà di scrivere una equazione matematica al suo interno (utilizzando ovviamente gli operatori matematici del linguaggio) e successivamente ritornerà il risultato dell’operazione. Questo vuol dire che al suo interno possiamo anche introdurre anche complesse equazione con funzioni importate da moduli, al patto che tutto questo venga rappresentato come una stringa;
- La Keyword max e min, in Python, non prendono come unico parametro un valore, bensì permettono di utilizzare anche una seconda keyword, che permette di determinare “quale max()” effettivo stiamo cercando. Se dovessi avere una lista con più liste, e dovessi determinare la lista più lunga presente all’interno della lista contenitore, potrei andare a scrivere “max(list_contenitore, key = len())”. Grazie a questo metodo, di conseguenza, ho trovato la lista più lunga. Ricorda che all’interno della key devi introdurre una funzione
- Se vogliamo andare a stampare una lista, rimuovendo le parentesi quadre, possiamo scrivere, all’interno del print, la seguente sintassi: “print(*my_list)”. Adesso, l’asterisco non lo utilizzeremo solo in questo caso, ma presenterà moltissimi altri utilizzi. Per esempio, mettiamo stiamo svolgendo un operazione di assegnamento multiplo E scriviamo la seguente sintassi: a, *b, c = “python”. Normalmente, dovremmo ottenere che la a sia uguale alla P, b sia uguale alla Y e c sia uguale alla T. Mettendo, però, l'asterisco prima della b, quest’ultima adesso immagazzinerà i valori che hanno si trovano dopo quello assegnato ad a e prima di quello assegnato a c. Se volessimo far si che l’intera Stinga venga trasformata in una stringa, possiamo anche scrivere *b, = “python”. Tutto questo sarà possibile perché l’asterisco rappresenta un operazione simile a quella dello slicing, quindi la b è una variabile che immagazzinerà il valore, e la “,” sta a rappresentare il divisore da noi utilizzato.
- Possiamo andare a specificare il tipo delle variabili alla loro dichiarazione, come in Java, scrivendo dopo quest’ultima “: tipo”, per esempio: “variabile :int = 100”
- Quando dobbiamo usare e dichiarare una variabile allo stesso tempo, possiamo utilizzare il warlas operator. La sua sintassi è particolarmente semplice, ed è formattata in questa maniera: “variabile := valore assegnato”;
- Se vogliamo assegnare ad una variabile una lista di elementi, o una serie di elementi modificati, possiamo utilizzare la funzione filter. Il compito di quest’ultima sarà quello di andare a prendere come primo parametro una funzione, e successivamente andare ad aggiungere l’oggetto iterabile a cui le modifiche verranno applicate. Ecco un esempio: variabile = filter(func, list).
- Se vogliamo andare a centrare la scrittura di una stringa, possiamo usare un trucco, che abbiamo già visto in parte in precedenza, nel quale abbiamo bisogno dell’utilizzo della f-string. Basterà scrivere: {str: _>Numerocaratteri}. In questa maniera, se la nostra stringa non fosse lunga “NumeroCaratteri”, allora verrebbero inseriti degli spazi a sinistra per riempire la parte mancante. Il simbolo “_” può essere modificato con qualsiasi altro simbolo, e cambiando “>” con “<”,

gli spazi non verranno aggiunti a sinistra ma bensì a destra. Se volessimo centrare il tutto, basterà scrivere: "^" piuttosto che gli altri due simboli visti precedentemente.

L'ultima cosa che andremo a vedere, per passare poi alle librerie, sarà il sistema "switch-case" esistente in Python, detto match-case. Come sappiamo già da Java, il match-case non potrà prendere come parametro un valore booleano (essendo identico allo switch-case), Ma bensì un valore distinto, quale potrà essere un numero, una stringa, un operazione, e cosi via. Il match-case sarà formato in questa maniera:

```
match valore:
    case valore1:

    case valore2:

    case _ : #valore default
```

Se c'è bisogno di andare a fare qualche approfondimento per quanto riguarda questo argomento, si possono trovare all'interno del quaderno di Java, dove il problema in capitolo è stat approfondito ampiamente.

CONTINUO – NUOVE KEYWORD

Prima di introdurci in maniera approfondita nel mondo delle librerie, introduciamo una nuova serie di keyword che possiamo utilizzare, e che ci possono aiutare nella scrittura del nostro codice. La prima è la funzione ord(). Quest'ultima prenderà come argomento una stringa, e successivamente, ritornerà il valore intero della sua posizione nella tabella ASCII. Dopo aver visto quest'ultima, approfondiamo una funzione molto importante. La funzione with(). Quest'ultima ci permette di lavorare con maggiore facilitò sui file, per apertura e chiusura, rispetto alle tecniche che noi tutti conosciamo. Prendiamo un esempio:

```
file = open("file.txt", "r")

for line in file:
    print(line)

file.close()
```

Vediamo quindi, che utilizzando questo metodo, saremo subito obbligati a chiudere il file, per non corrompere quest'ultimo o sprecare risorse di memoria. Utilizzando, però la keyword with(), non saremo obbligati a far tutto ciò, e difatti, il file sarà chiuso automaticamente e sarà meglio gestito dall'editor.

```
With open("file.txt", "r") as "file":
    for line in file:
        print(line)
```

La prossima keyword che vediamo, invece, è la keyword any(). Il suo unico compito è quello di prendere in input un iterabile e determinare se al suo interno sia presente o meno un valore booleano True (ritornandolo), mentre la keyword finally viene utilizzata come blocco di "default" simile al match case in un blocco try ed except.