

Novità di Base

Genericità:

- Istruzioni e Comandi
- Data Objects
- Stringhe: sequenza case sensitive di caratteri
- Indicizzazione:
 - + Classica: $s[0]$ = primo elem
 - + Inversa: $s[-1]$ = ultimo elem
- Slice: notazione $s[start : stop(escluso) : passo]$

Input/Output:

- Print/ln
- Input: stampa a schermo la stringa indicata ed attende un inserimento da tastiera, necessita di typecasting qualora il mostro input non sia di tipo String

Flussi:

- Python's if
- Indentazione: fondamentale in python
- Ciclo for: for variable in range (start, stop, step):
 - ↳ Break
 - Uso del forEach pythomico: (esempio per stringhe) for char in s: if char ==

Funzioni:

- Keyword def \rightarrow def function (parametrs):
- docstring (optional)
- body
- return clause

Scope:

- Variabili locali
- Globali: richiamabili anteponendo "global" all'interno della funzione
- Parametren:
 - + Formal
 - + Actual
- Assenza di return o return privo di valore: Funzione ritorna None

Tuple, Liste e Dizionari

Tupla:

- Sequenza ordinata ed immutabile di oggetti diversi
- $t = (2, "mit", 3)$
- $(2, "mit", 3) + (5, 4) : (2, "mit", 3, 5, 4) \rightarrow$ Stampa la tupla aggiornata ma non la aggiorna sul serio
- 1° dei tipi dati python usati per l'archiviazione dati
- Possibile iterare su esse

Lista:

- Sequenza di oggetti solitamente omogenei, mutabile
- $L = []$
- Sono oggetti iterabili tramite ciclo for.
- Operazioni:
 - + Add $\rightarrow L.append(value)$
 - + Extend $\rightarrow L_3 = L_1 + L_2$ oppure $L_1.extend([0, 6])$
 - + Remove:
 - $del(L[positione])$
 - $L.pop() \leftarrow$ rimuove ultimo della lista
 - $L.remove(elemento)$
 - + Conversioni:
 - $list(s)$: Trasforma s in una Lista
 - $s.split$: Split della stringa al parametro indicato
 - $!'.join(L)$: da lista a stringa

Alias:

- alias = Variabile : Sono associate alla stessa posizione in memoria
- modificare l'alias comporta anche una modifica dell'originale.
- Per evitare ciò, al posto dell'alias, si utilizza la clonazione: $s1 = s2[:]$

Ordinamento:

- $Sort()$: modifica la lista su cui è chiamata
- $Sorted()$: crea una nuova lista ordinata

Dizionari:

- Una lista dove ad ogni elemento viene associata una chiave (di qualsiasi tipo)
- Definiti come: $dictionary = \{ \}$
- Operazioni:
 - + Add
 - + Delete
 - + $lookup$: ricerca di valore tramite chiave ($dictionary['Key']$)
 - + Lista Key e Values: $dict.keys()$ e $dict.values()$
- Valori: di qualunque tipo ed anche duplicati
- Keys: uniche ed immutabili

Gestione File ed eccezioni

With :

- Permette di associare una risorsa ad una variabile e di rilasciarla automaticamente al termine del blocco di istruzioni

Open e Write:

- Open ("file.txt"; "modalità")
- Modalità:
 - + r: Read
 - + w: Write (crea il file se non esiste mentre, se esiste, sovrascrivere il suo contenuto)
 - + a: Append (una write senza sovrascrittura)
 - + r+: Modifica (Read + Write)
 - + w+: Modifica (ma con sovrascrittura)

Lettura da file:

- For line in File: legge una riga per volta, ogni riga diventa una singola String.
- ReadLines(): legge il contenuto del testo nella sua interezza e restituisce una lista di Stringhe.
- Seek(): permette il posizionamento di un puntatore ad una stringa della lista ↴

Modulo OS:

- funzioni per modifica di file, simili a quelle del nostro OS
- Ex: os.remove("nome_file").

Eccezioni:

- Tipi fondamentali:
 - + SyntaxError
 - + NameError
 - + TypeError
 - + AttributeError
 - + ValueError
 - + IOError
- Gestione:
 - + Tramite blocchi try-except (esistono anche i blocchi 'else', eseguiti dopo un try privo di errori, ed i blocchi 'finally')
 - + raise: istruzione per il lancio di una Exception [raise <typeError> ("text") <param>]

Assertioni:

- assert <condition>, <Error>: Se la condizione è "true", assert lancia una exception, altrimenti si passa all'istruzione successiva
- Programmazione difensiva
- Scopo: Evitare che valori erronei si propaghino nel programma

Python e OOP

Oggetti:

- In python ogni cosa è un oggetto
- Python ha un proprio ed "automatico" garbage collector.

Classi:

- keyword erazionale: `class ClassNome (parent class)` (es: `class Coordinate (object)`)
- Metod `__init__()`: Permette di inizializzare un'istanza della classe.
- Attributi:
 - + Di classe
 - + Di istanza
- Metod `__str__()`: Permette di stampare gli attributi dell'istanza; dato che la sola `print(o.value)` stamperei del contenuto non informativo. (definito tale metodo, potremo usare la `print`, e come una sorta di `Overwrite`).
- Altri Overload:

<code>__add__(self, other)</code>	→	<code>self + other</code>
<code>__sub__(self, other)</code>	→	<code>self - other</code>
<code>__eq__(self, other)</code>	→	<code>self == other</code>
<code>__lt__(self, other)</code>	→	<code>self < other</code>
<code>__len__(self)</code>	→	<code>len(self)</code>
<code>__str__(self)</code>	→	<code>print self</code>
... e altri		
- Metodi Getter e Setter: utili per information hiding.

Ereditarietà:

- Super Class: padre
- Subclass: figlio
- Se un metod della classe figlio ha stesso nome di uno del padre, viene eseguito quello del figlio.
- Se per una classe viene eseguito un metod non definito in essa, si risale la gerarchia delle classi fino a trovarlo.
- Polimorfismo: un'interfaccia gestisce più classi differenti.

Class Exception:

- Possibile definirle come figlie di: `BaseException`
- Sottoclassi fondamentali:
 - + `SystemExit`
 - + `KeyboardInterrupt`
 - + `GeneratorExit`
 - + `Exception`

Main:

- `__name__` variable: contiene il nome del modulo.py; esso sarà `__main__` se stiamo eseguendo direttamente il modulo in questione e non uno importato.

Programmazione Concorrente in Python

Moduli utili:

- `threading`
- `multiprocessing`

Threading:

- interfaccia di alto livello per la gestione dei programmi multithreading
- a partire dal modulo `thread`
- Metodi di creazione di un thread:

```
import threading
def func():
    print("Thread running")

if __name__ == "__main__":
    # creating thread
    t1 = threading.Thread(target=func)

    # starting thread
    t1.start()

    # wait until the thread finishes
    t1.join()
```

```
import threading
class myThread(threading.Thread):
    def run(self):
        print("Thread running")

if __name__ == "__main__":
    # creating thread
    t1 = myThread()

    # starting thread
    t1.start()

    # wait until the thread finishes
    t1.join()
```

- Costruzione thread, parametri:
 - + Groups: per specificare il thread group
 - + target: Callable object da passare al metodo run() del thread
 - + Name: nome del thread
 - + Args: tupla con argomenti da passare all' iniezione
 - + KwArgs: Dizionario delle Keyword.
 - + Daemon: booleano che indica se il thread è daemon o meno
- Classe local: permette di gestire dati locali ad un thread

Lock Class:

- lock
- Metodi:
 - + acquire
 - + release
 - + locked: verifica lo stato del lock

Reentrant Lock:

- RLock
- Metodi:
 - + acquire: decrementa il recursion level
 - + release: incrementa il recursion level, se arriva a 0, il lock diventa unlocked

Condition Class:

- Implementa le condition variables ed i vari metodi che le riguardano.
- Condition(lock=" "): costruzione, se il lock a cui associa la CV non viene indicato, viene creato sul momento un RLock
- with CV-name: permette di indicare una sezione critica senza l'uso di acquire e release.

Semaphore Class:

- Come in java, in tutto e per tutto

Event Class:

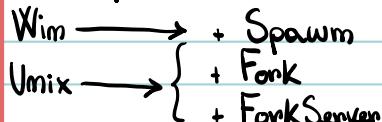
- Si basa su un semplice meccanismo di comunicazione tra threads.
- ha un flag interno per il controllo dell'evento, modificabile tramite metodo set().

Python Global Interpreter Lock:

- Gil: mutex che permette ad un unico thread per volta di trovarsi in esecuzione, ossia di avere accesso all'interprete python.

Multiprocessing:

- Si hanno più interpreti, ciascuno con il proprio Gil, ergo, è possibile eseguire più processi contemporaneamente
- Metodi di avvio di un processo:



- Modulo multiprocessing: + estensione del modulo multiprocessing
+ permette una maggiore serializzazione
+ passaggio di oggetti e risorse tra processi

Process Class:

- Analogia dei thread, ma per i processi
- Anche qui è possibile usare gli oggetti atti alla sincronizzazione

Comunicazione tra Processi:

- Canali di comunicazione:

- + Pipe:

- Pipe ([duplex]): se duplex è true abbiamo una pipe bidirezionale
 - I metodi sono send(obj), recv()
 - Attenzione: Non usare uno stesso endpoint per prelievo e deposito dei dati di entrambi gli endpoint della Pipe.

- + Queue:

- Formisce una shared queue
 - Quando un oggetto viene inserito nella coda, viene avviato un thread che lo porta da un buffer alla pipe che implementa la coda.
 - Queue ([maxsize])
 - Metodi fondamentali: put(obj, block(true, timeout=?)) e get()

- Ulteriori Queue:

- + SimpleQueue
 - + JoinableQueue

Shared Memory:

- `InitializeFromIt`:

- + `Value (type, *args, lock (true, timeout))`: ritorna un oggetto ctype allocato nella shared memory.

- + `Array (type, size_on_init, *, lock)`: ritorna un array ctype allocato nella shared memory.

Programmazione in Rete con Python

Socket:

- si crea tramite il metodo `socket.socket(socket_family=[1], socket_type=[2])`
 - [1.] famiglia socket, ad esempio 'socket.AF_INET' si per una socket IPv4
 - [2.] tipo della socket:
 - 'socket.SOCK_STREAM' per TCP
 - 'socket.SOCK_DGRAM' per UDP
- Metodi fondamentali:
 - + `bind(address)`
 - + `listen(backlog)`: indica alla socket di mettersi in ascolto su diversi porti, backlog me indica il numero massimo.
 - + `accept()`
 - + `connect(address)`

Scambio su socket:

- TCP:
 - + `send(String)`
 - + `recv(bufferSize)`: ritorna un oggetto stringa
- UDP:
 - + `sendto(String, address)`
 - + `recvfrom(bufferSize)`: ritorna la coppia String + Address

TCP passaggi:

- Server:
 1. Inizializza socket ed effettua il `bind(Port, Address)`
 2. Pone la socket in ascolto con `listen()`
 3. Salva nelle variabili `conn` ed `addr` i risultati di `socket.accept`
 4. Crea il canale di ricezione con `conn.recv(bufferSize)`
 5. Sfrutta le `send` ed effettua la `close`
- Client:
 1. Inizializza socket ed effettua la `connect(Port, Address)`
 2. Effettua le `send()`
 3. Crea il canale di ricezione (`data = s.recv(bufferSize)`)
 4. `Close`
- Nb:
 - + Invio: `obj.encode("utf-8")`
 - + Ricezione: `obj.decode("utf-8")`

UDP:

- è connectionless
- Si comporta come TCP ma, essendo priva di connect, indirizzo e port vengono indicati nella `sendto` e `recvfrom`.

Server Multithread:

- Ogni thread gestisce la connessione con uno specifico client
- Casi:
 - + TCP: il thread deve ricevere una connessione rilasciata da connect()
 - + UDP: il thread deve ricevere l'output della readfrom
- Metodo run contiene la gestione della richiesta

Proxy - Skeleton:

- Proxy: fornisce un'interfaccia o un surrogato di un oggetto remoto, con il quale funge da intermediario per interagire con esso.
- Skeleton: analogo del proxy ma lato server, viene implementato per delega (quindi si occupa solo della parte comunicativa) o per ereditarietà (effettiva implementazione dell'interfaccia oggetto)

Interfaccie in Python:

- Interfacce informali: la classe interfaccia pone la keyword "pass" come contenuto dei metodi di classe, mentre la classe implementativa si pone come figlia dell'interfaccia e li implementa.
- Esempio appunti: Scrivere: Fondamentale

Python STOMP

Simple Text Oriented Messaging Protocol:

- o Protoocollo frame based che richiede una 2 way streaming network protocol (TCP)
- o Permette l'interoperabilità tra linguaggi e piattaforme differenti
- o stomp.py: libreria del servizio di messaging

Connessione al provider:

- o `import stomp`
- o `conn = Stomp.Connection([(IP, port), (IP, port), ...])`
- o `conn.connect()`
- o disconnessione: `conn.disconnect()`

STOMP messages:

- o Invio:
 - + `conn.send('destination', body)`, la destinazione può essere sia una queue sia un Topic
- o Ricezione:
 - + `conn.setListener('monna'; istomp.Listener)`
 - `MyListener()`
 - `PrintingListener()`
 - + `conn.subscribe('destination', id, ack = "auto" / "client" / "client individual")`

STOMP Listener:

- o Va creata una classe figlia di "stomp.ConnectionListener" per far sì che implementi il metodo `onMessage(self, frame)`

Frame:

COMPONENTI frame:
 {
 · body
 · headers
 · command

STOMP JMS:

Esempio con ActiveMQ e JMS (Text Message)

```
# sender.py
import stomp

conn = stomp.Connection([('127.0.0.1', 61613), auto_content_length=False])
conn.connect(wait=True)

conn.send('/queue/mytesttopic', 'test message')

conn.disconnect()
```

In più, è necessario utilizzare il `physical-name` del topic/queue specificato lato JMS

```
// jndi topic name physical topic-name
jndiProperties.put( key:"topic.test", value:"mytesttopic" );
```

Porto del servizio ActiveMQ

Il codice è simile al `sender.py` precedente, ma ha un'opzione aggiuntiva nella creazione dell'oggetto di connessione STOMP:

`conn = stomp.Connection([('127.0.0.1', 61613), auto_content_length=False])`

L'inclusione/esclusione dell'header `auto_content_length` specifica il tipo di messaggio da creare quando si invia un messaggio STOMP a JMS.

Inclusion of content-length header	Resulting Message
yes	BytesMessage
no	TextMessage

DataScience

Misure:

- Misura di tendenza: (media, mediana, moda) sono valori descrittivi di un set di misure
- In python nel modulo statistics, abbiamo:
 - + statistic.median(iterable, obj)
 - + statistic.mode(it, obj)
 - + statistic.variance(it, obj)
- Misure di dispersione: per comprendere la distribuzione dei valori:
 - + varianza
 - + deviazione standard

Numpy:

- libreria per DataScience
- offre i ndarray: array multidimensionali
- funzione array: ritorna un array da un array/List ...
- Attributi:
 - + dtype: tipo array
 - + ndim: numer dimensioni
 - + shape: ritorna (n. righe, n. colonne)
- Riempimento array:
 - + np.zeros (m)
 - + np.ones ((righe, colonne), dtype=?)
 - + np.full ((//, //), value)
- arange (start, stop_excluded, passo): crea array
- linspace (start, stop, numbers_to_generate)
- reshape (n. righe, n. colonne): trasforma array in matrice
- Operatori:
 - ↳ + aritmetiche: non modificano l'array in memoria
 - ↳ + Broadcasting: il secondo elemento è uno scalare che interagisce con tutti gli elementi dell'array
 - + Tra array
 - + Comparazione: stampa un array di booleani
- Calcolo matriciale:
 - + per riga: axis=1
 - + per colonna: axis=0
- Universal Functions:
 - + add (array1, array2)
 - + sqrt (array1)
 - + multiply (array1, number)
- View(): restituisce un puntatore all'array.
- Copy(): restituisce una copia indipendente dell'array
- Reshaping: modifica dimensioni array
 - + reshape (view) /.ravel
 - + resize (modifica originario) / flatten
- Trasposizione: inversione indici righe e colonne
 - + transpose()

Pandas

Genericità:

- Libreria per la gestione di Big Data
- Supporti per collections:
 - + Unidimensionali: Series
 - + Bidimensionali: DataFrames
 - + Multidimensionali: MultiIndex
- Le implementazioni (estremamente simili a NumPy) sono tutte negli appunti

Modulo CVS

CVS:

- Comma Separated Value
- Scrittura:
 - + Apertura file.csv
 - + Usare funzione writer del modulo csv
 - + Uso writerow sull'oggetto 'writer' che richiede in ingresso un iterabile
- Lettura:
 - + Uso dell'oggetto reader, il quale, è anche iterabile.

CVS + Pandas:

- .read_csv('file.csv', names=[...], skiprows=...): permette di passare il contenuto del csv in un DataFrame
- .to_csv('file.csv', index=False): permette di salvare il contenuto di un DataFrame in un CSV
- Metodi utili:
 - + head: stampa le prime 5 righe del dataset
 - + tail: stampa le ultime 5 righe del dataset
 - + .columns = ['newname', 'newname', ...]

Data Visualization:

• librerie:

- + Seaborn
- + Matplotlib

Competenze finali:

- Serie temporali: include dati le cui osservazioni sono ordinate nel tempo.
 - + Univariate
 - + Multivariate

- Regressione Lineare semplice:
 - + descrive la relazione tra una variabile indipendente ed una dipendente tramite l'utilizzo della retta di regressione
 - + Componenti $y = mx + b$:
 - m : pendente lineare
 - b : interetta con l'asse y
 - y/x : var dipendente / var indipendente

Funzione linregress della Libreria SciPy!