PER ALTRI APPUNTI CONSULTARE IL SITO:

https://luigi-v.github.io/Appunti_Universita/

Per altri appunti consultare il sito: *https://luigi-v.github.io/Appunti_Universita/*

By *Vollono Luigi*

CONCORRENZA:

In Python la principale distinzione è data dall'accesso diretto (ad esempio attraverso memoria condivisa) o indiretto ai dati (ad esempio utilizzando la comunicazione tra processi). La concorrenza può essere implementata in due modi:

1. Basata sui ***thread***, detta ***multithreading***, si ha quando diversi thread di esecuzione operano all'interno dello stesso processo di sistema, accedendo a dati condivisi tramite un accesso alla memoria condivisa.
2. Basata sui ***processi***, detta ***multiprocessing***, si ha quando più processi distinti tra loro vengono eseguiti in modo indipendente. I processi concorrenti condividono i dati tra loro mediante ***IPC*** (***inter-process communication***), anche se possono comunque utilizzare la memoria condivisa nel caso il linguaggio o la libreria lo permettano.

Python supporta entrambi i tipi di concorrenza appena visti, ma i thread vengono utilizzati solo per convenzione, mentre i processi vengono usati perché sono di più alto livello di quello fornito da altri linguaggi, il supporto al multiprocessing utilizza le stesse astrazioni del threading per facilitare il passaggio tra i due approcci, almeno quando non viene usata la *memoria condivisa*.

PROBLEMATICHE LEGATE A GIL:

A causa del ***GIL*** (***Global Interpreter Lock***) l'interprete Python può essere in esecuzione solamente su uno dei core del processore, negando la produzione di incrementi prestazionali nel caso si usino i *thread*: questo perché si utilizzano molteplici thread in un solo processo, quindi il GIL rimane in esecuzione su un solo core.

- Se l'elaborazione è ***CPU-bound*** l'uso dei thread può portare a performance peggiori rispetto a quelle in cui non si fa uso della concorrenza.
  Una soluzione consiste nell'usare Cython che è Python con costrutti C, portando a migliori performance, che tende ad aggirare ***GIL***, ma è meglio evitarlo del tutto scegliendo di utilizzare il modulo ***multiprocessing***. Tale modulo non utilizza i thread, bensì usa ***processi separati***, ognuno dei quali utilizza la propria istanza indipendente dell'interprete Python, senza creare conflitti.
- Se la computazione è ***I/O-bound***, come ad esempio nelle reti, usare la concorrenza può portare a miglioramenti delle performance molto significativi.

LIVELLI DI CONCORRENZA:

Con Python è altamente consigliato scrivere sempre un programma non concorrente, dato che è più semplice da scrivere e testare. Non conta solamente il tipo di concorrenza, bensì anche il livello:

- ***Concorrenza a basso livello***: utilizza operazioni atomiche, azione indivisibile , che viene eseguita indipendentemente da qualsiasi altro processo, indispensabile per operazioni più complesse. Questo tipo di concorrenza è adatto a chi scrive librerie e non a chi scrive applicazioni;
- ***Concorrenza a livello intermedio***: non utilizza operazioni atomiche, bensì ***lock*** espliciti, supportato dalla maggior parte dei linguaggi di programmazione e viene utilizzato per lo sviluppo di applicazioni, offre classi come *threading.Semaphore*, *threading.Lock* e *multiprocessing.Lock* per la gestione della concorrenza;
- ***Concorrenza ad alto livello***: non prevede né operazioni atomiche e né lock espliciti, Python fornisce il modulo ***concurrent.futures*** e le classi ***queue.Queue***, ***multiprocessing.queue*** o ***multiprocessing.JoinableQueue*** per supportare la concorrenza ad alto livello.

Gli approcci di livello intermedio sono semplici da utilizzare ma soggetti ad alti rischi di errori logici, come ***deadlock*** (*processi che si bloccano a vicenda*) e problematiche varie della concorrenza. Problema chiave è la ***condivisione dei dati***.

I ***dati condivisi mutabili*** devono essere protetti mediante ***lock*** per assicurare che tutti gli accessi siano *serializzati*, in modo che quando viene posto un *lock*, un solo thread o processo alla volta possa accedere ai dati condivisi, come se fosse non concorrente. Di conseguenza rimangono due scelte: i lock vengono utilizzati il meno possibili e per tempi brevi, o non si condividono dati mutabili, in modo da evitare totalmente l'utilizzo dei lock.

In quest'ultimo caso, una soluzione alternativa ai lock consiste nell'utilizzare una struttura dati che preveda l'accesso concorrente, come il ***modulo queue*** che offre ***code thread-safe***, mentre per il ***multiprocessing*** ci sono classi apposite come ***multiprocessing.JoinableQueue*** e ***multiprocessing.Queue***. Tali code forniscono sia una *singola sorgente di job* per tutti i thread e tutti i processi, sia un'*unica destinazione dei risultati*.

PACCHETTO MULTIPROCESSING:

Un oggetto ***multiprocessing.Process*** rappresenta un'attività svolta in un processo separato, il costruttore è il seguente:

multiprocessing.Process(group=None, target=None, name=None, args=( ), kwargs={ }, daemon=None)	
group	deve essere sempre ***None*** in quanto è solo per compatibilità con *threading.Thread*
target	oggetto callable invocato da ***run()***: se rimane ***None***, non verrà invocato alcun metodo
name	nome del processo
args	tupla di argomenti da passare a target, cioè l'oggetto callable;
kwargs	dizionario di argomenti keyword da passare a target, cioè l'oggetto callable
deamon	booleano che indica se il processo deve essere creato come regolare oppure come daemon. Se il valore è ***None***, allora viene ereditato dal processo invocante.

È caratterizzato da molteplici metodi e due più usati per dare inizio ad un processo:

run()	Metodo che rappresenta l'attività del processo
Può essere sovrascritto, il metodo standard invoca l'oggetto callable passato al costruttore di Process con gli argomenti presi dagli argomenti args e kwargs, passati anch'essi al costruttore.	
start()	Metodo che dà inizio all'attività del processo
Deve essere invocato al più una volta per un oggetto processo e fa in modo che il metodo *run()* dell'oggetto venga invocato in un processo separato.	
join(*timeout = None****)***	Metodo che se viene invocato da un thread principale, questo attende fino alla chiusura del thread figlio cu cui viene richiamato join. Se join () non viene invocato, il thread principale potrebbe uscire prima del thread figlio.
Se timeout è un numero positivo, join si blocca per alcuni secondi. Può essere invocato più volte per uno stesso oggetto, un processo non può invocare join() su sé stesso in quanto provocherebbe un deadlock.	
spawn()	Processo padre lancia un nuovo processo per eseguire l'interprete Python, il processo figlio erediterà solo le risorse necessarie per eseguire il metodo ***run()***
Questo modo di iniziare i processi è molto lento se confrontato a fork, ma è disponibile sia per Unix che per Windows.	
fork()	Processo padre utilizza la fork per fare il fork dell'interprete Python, il processo figlio sarà praticamente uguale al processo padre, quindi ne erediterà risorse e caratteristiche
Tale metodo è disponibile solo su Unix, dove rappresenta il metodo di default per iniziare i processi.	

È possibile che dati o processi vogliano accedere agli stessi dati: è possibile usufruire di due classi, quali sono ***Queue*** e ***JoinableQueue***, le quali saranno l'unica fonte di task/job per tutti i thread o processi e un'unica destinazione per i risultati. Tali code sono condivise dai processi, quindi qualsiasi oggetto conservato è accessibile da qualsiasi processo.

Alcuni metodi di ***multiprocessing.Queue***, restituiscono un output non affidabile causa semantica multithread/process:

qsize()	Restituisce la dimensione approssimata della coda
empty()	Restituisce True se la coda è vuota, False altrimenti
full()	Restituisce True se la coda è piena, False altrimenti
put(*obj, block, timeout****)***	Inserisce *obj* nella coda, opzionali *block* e *timeout*. *block* = True, attende che la coda liberi uno slot per inserire l'oggetto al massimo per timeout, se specificato, se lo slot non si libera entro timeout secondi, viene lanciata un'eccezione *queue.Full*; *block* = False, tenta l'immediato inserimento, se la coda è piena lancia un'eccezione *queue.Full*.
put_nowait(*obj****)***	equivalente a *put(obj, False)*
get(*block, timeout****)***	Rimuove e restituisce un oggetto dalla coda. *block* = True, attende che ci sia un elemento nella coda per restituirlo al massimo per timeout, se specificato, se non sarà presente un elemento da restituire entro timeout secondi, viene lanciata un'eccezione *queue.Empty*; *block* = False, tenta l'immediata restituzione dell'elemento, se la coda è vuota lancia un'eccezione *queue.Empty*.
get_nowait()	equivalente a *get(False)*

La classe ***multiprocessing.JoinableQueue*** è una sottoclasse di ***Queue*** che ha in aggiunta i metodi:

task_done()	Indica che un task precedentemente inserito in coda è stato completato. Per ciascuna *get()* utilizzata per prelevare un task, deve essere effettuata una chiamata a *task_done()* per informare la coda riguardo il completamento del task. Un *join()* bloccato si sblocca quando tutti i task sono stati completati e dopo che è stata ricevuta una chiamata a *task_done()* per ogni task precedentemente inserito in coda. Si ha un *ValueError* se *task_done()* è invocato un numero di volte maggiore degli elementi in coda;
join()	Causa un blocco sullo scope nel quale viene invocato tale metodo fin quando gli elementi della coda non sono stati tutti prelevati e processati. Il conteggio dei task incompleti incrementa ogni volta in cui viene aggiunto un elemento alla coda e viene decrementato ogni volta che viene invocato task_done(). Quando il conteggio dei task incompleti va a zero, join() si sblocca.

JoinableQueue si comporta in base ai task che contiene, se contiene task non processati allora rende bloccato lo scope, altrimenti lo sblocca.

ESEMPIO CONCORRENZA con JoinableQueue e Queue:

Viene creata una coda ***JoinableQueue*** composta dai jobs da eseguire e una normale ***Queue*** che conserva i risultati.

```
def operation(text, concurrency):
    jobs = multiprocessing.JoinableQueue()
    results = multiprocessing.Queue()
    create_processes(jobs, results, concurrency)
    add_jobs(text, jobs)
    try:
        jobs.join()
    except KeyboardInterrupt:
        print("cancelling...")
    while not results.empty():
        print(results.get_nowait())
```

Successivamente si creano i processi che eseguiranno il lavoro, i quali saranno pronti ma bloccati, perché la coda dei jobs è vuota.

Si prosegue aggiungendo ***jobs*** alla ***JoinableQueue*** per poi eseguire una ***join***, che bloccherà lo scope attuale finchè tutti i processi non avranno concluso i jobs della ***JoinableQueue***.

create_processes() si occupa di creare i processi che eseguiranno il lavoro, a ciascun processo viene dato come parametro la funzione ***worker***, la quale si occupa del reale lavoro da svolgere. Inoltre viene passata anche la coda di ***jobs*** e la coda dei ***results***. Ad ogni processo creato, viene settato a ***True*** l'attributo ***daemon***, il quale fa in modo che il processo principale attendi che esso finisca l'elaborazione.

```
def create_processes(jobs, results, concurrency):
    for i in range(concurrency):
        print("Creo il processo ",i+1)
        process = multiprocessing.Process(target=worker, args=(jobs, results))
        process.daemon = True
        process.start()
```

```
def worker(jobs, results):
    while True:
        try:
            newText = jobs.get()
            result = operationText(newText)
            results.put(result)
        finally:
            jobs.task_done()
```

worker() esegue un ciclo infinito (procedura sicura perché i processi sono ***daemon***) ed in ogni iterazione tenta di ottenere un ***job*** da eseguire dalla coda dei jobs condivisa.

Ricevuto un ***job*** lo si processa, viene inserito il risultato nella coda ***results*** e si comunica alla coda di ***jobs*** che il task è stato concluso.

```
def add_jobs(text, jobs):
    for i in range(4):
        print("Aggiungo il job ",i+1)
        newText = text + str(i+1)
        jobs.put(newText)

def operationText(text):
    return "Hello " + text
#MAIN-------------------------------
if __name__ == '__main__':
    operation("World ", 2)
```

→

```
Creo il processo  1
Creo il processo  2
Aggiungo il job  1
Aggiungo il job  2
Aggiungo il job  3
Aggiungo il job  4
Hello World 1
Hello World 2
Hello World 3
Hello World 4
```

MODULO concurrent.futures:

concurrent.futures costituisce un mezzo di alto livello per realizzare concorrenza utilizzando più thread e processi. I ***future*** sono istanze della classe ***concurrent.futures.Future*** ed eseguono i *callables* in maniera *asincrona*, si creano richiamando il metodo ***concurrent.futures.Executor.submit()***, restituisce un oggetto ***Future***, e possono segnalare il loro stato (interrotto, in esecuzione, completato) e il risultato o eccezione prodotta.

La classe ***Executor*** non può essere utilizzata siccome è una *classe astratta*, ma possiamo utilizzare una delle sottoclassi:

- ***concurrent.futures.ProcessPoolExecutor()*** produce una concorrenza basata sull'utilizzo di più processi. L'uso di un pool significa che ogni ***Future*** utilizzato con esso può eseguire e restituire solamente oggetti ***pickleable***.
- ***concurrent.futures.ThreadPoolExecutor()*** non ha la restrizione di ***ProcessPoolExecutor*** e realizza la concorrenza utilizzando i thread.

L'utilizzo di un pool è molto più semplice che utilizzare le code:

Si crea un ***set*** di ***futures***, per poi creare un oggetto ***ProcessPoolExecutor*** che creerà un numero di processi ***worker***. Adesso si itera sui job restituiti da ***get_jobs()*** e crea per ciascuno di essi un future. Il metodo ***submit()*** accetta una funzione ***worker*** e argomenti, per poi restituisce un oggetto ***Future***, che verrà riconosciuto dal *pool* ed eseguito.

```
def operation(text, concurrency):
    futures = set()
    with concurrent.futures.ProcessPoolExecutor(max_workers=concurrency) as executor:
        for job in get_jobs(text):
            future = executor.submit(operationText, job)
            futures.add(future)
    canceled = wait_for(futures)
    if canceled:
        executor.shutdown()
```

```
def wait_for(futures):
    canceled = False
    try:
        for future in concurrent.futures.as_completed(futures):
            err = future.exception()
            if err is not None:
                raise err
    except KeyboardInterrupt:
        print("canceling...")
        canceled = True
        for future in futures:
            future.cancel()
    return canceled
```

Quando tutti i future sono stati creati, viene chiamata ***wait_for()***, passandole ***futures***, che si bloccherà fino a quando tutti i future sono stati eseguiti o cancellati dall'utente, se si verifica quest'ultimo la funzione dismette il *pool executor*.

Viene invocato ***as_completed()*** che si blocca fino a che non viene completato o cancellato un future e poi lo restituisce.

Se il callable worker eseguito dal future lancia un'eccezione allora il metodo ***future.exception()*** la restituisce, altrimenti restituisce ***None***. Se non si verifica eccezione allora viene recuperato il risultato del future e riportato all'utente.

```
def get_jobs(text):
    for i in range(4):
        print("Aggiungo il job ",i+1)
        newText = text + str(i+1)
        yield newText
```

Con ***get_jobs()***, invece di aggiungere job (come add_jobs()) alla coda, è una funzione generatrice che restituisce ***job*** su richiesta.

```
def operationText(text):
    print("Hello " + text)

#MAIN--------------------
if __name__ == '__main__':
    operation("World ", 2)
```

→

```
Aggiungo il job  1
Aggiungo il job  2
Aggiungo il job  3
Aggiungo il job  4
Hello World 1
Hello World 2
Hello World 3
Hello World 4
```

Process:

Se accodassimo più volte ***operation()***, si avrebbe un tempo di esecuzione maggiore, ma possiamo svolgerli in parallelo, importando il modulo ***multiprocessing*** per utilizzare la classe ***Process*** che ci permette di eseguire funzioni in parallelo.

```
def operation(nameProces):
    print(f"Esecuzione di {nameProces}")
#MAIN-------------------------------------------------------------
if __name__ == '__main__':
    start = time.perf_counter()
    p1 = multiprocessing.Process(target=operation, args=("p1",))
    p2 = multiprocessing.Process(target=operation, args=("p2",))
    p3 = multiprocessing.Process(target=operation, args=("p3",))
    processList = [p1, p2, p3]
    for p in processList:
        p.start()
    for p in processList:
        p.join()
    finish = time.perf_counter()
    print(f"\nFinito in {round(finish - start, 2)} secondi")
```

L'output non è sempre lo stesso, in ogni esecuzione…

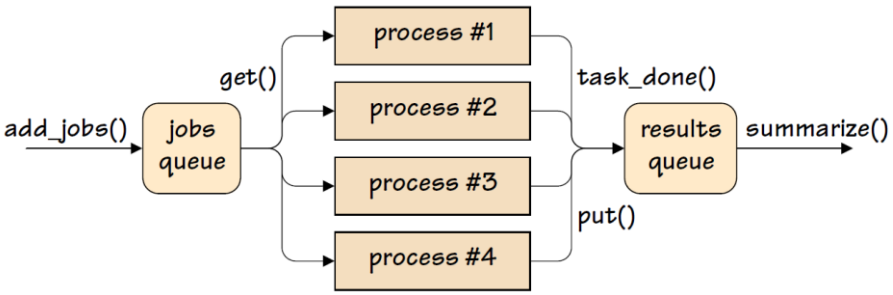
→

Esecuzione di p1
Esecuzione di p2
Esecuzione di p3

Finito in 0.08 secondi

La classe ***Process*** prende come parametro un target, ovvero la funzione che il processo dovrà eseguire, la creazione del processo non implica il suo avvio, difatti è sempre necessario l'utilizzo della ***start()*** per avviarlo.

È necessario che il ***main*** attenda i processi figli, ciò è possibile tramite il metodo ***join()***, il quale blocca l'esecuzione del processo attuale (in questo caso il main) finchè il processo su cui è stata invocata la join non finirà l'esecuzione. Nel caso non venga fatta alcuna join, il processo principale si concluderebbe prima della fine dell'esecuzione del processo figlio.



ProcessPoolExecutor:

Utile strumento per eseguire processi in maniera più semplice ed efficiente.

Ottenuto un oggetto ***ProcessPoolExecutor*** dal *context manager*, dichiarato come ***executor***, è possibile effettuare operazioni sui processi: creare un processo, passargli la funzione da eseguire ed avviarlo, basta il metodo ***submit.***

Il metodo ***executor.submit*** accetta come parametri la *funzione* da eseguire ed i suoi *argomenti*, ritorna un oggetto ***future***, sul quale è possibile effettuare operazioni riguardanti il processo, come l'ottenere il valore di ritorno dell'esecuzione della funzione passata al processo, ciò viene fatto tramite l'esecuzione del metodo ***result***.

```
def operation(nameProces, seconds):
    print(f"{nameProces} dorme per {seconds}")
    time.sleep(seconds)
    return f"Esecuzione di {nameProces}"
#MAIN-------------------------------------------------------------------
if __name__ == '__main__':
    with concurrent.futures.ProcessPoolExecutor() as executor:
        listFutures = []
        for i in range(3):
            listFutures.append(executor.submit(operation,"p"+str(i+1),i+1))
        for i in range(3):
            print(listFutures[i].result())
```

→

p1 dorme per 1
p2 dorme per 2
p3 dorme per 3
Esecuzione di p1
Esecuzione di p2
Esecuzione di p3

Thread:

Come i processi, anche i thread permettono di ottimizzare l'esecuzione dello script. La tecnica migliore dipende dal tipo di operazione da eseguire in concorrenza (ad esempio, l'utilizzo dei processi è consigliato per le operazioni CPU-bound, mentre l'utilizzo dei thread è consigliato per le operazioni I/O-bound).

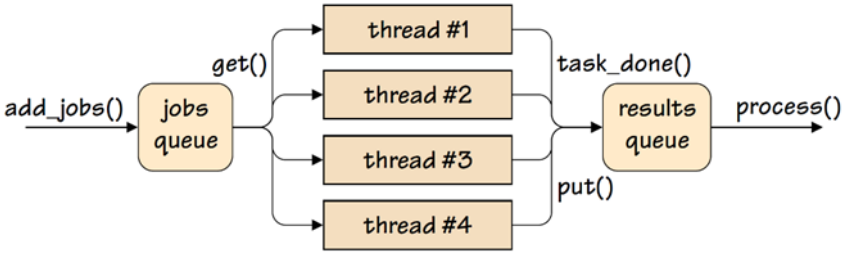
Importiamo il modulo ***threading*** ed utilizzando la classe ***Thread*** per istanziare i threads: proprio come già visto con i processi, basta istanziare un oggetto ***Thread*** passandogli la funzione da eseguire come argomento, per poi avviarlo tramite il metodo ***start()*** ed attenderlo tramite il metodo ***join()***, evitando che il thread principale (il main) si concluda prima dei thread creati. Ovviamente è possibile passare argomenti alla funzione da eseguire proprio come se si stessero gestendo i processi, quindi tramite una lista args.

```
def operation(nameThread):
    print(f"Esecuzione di {nameThread}")
#MAIN-------------------------------------------------------
start = time.perf_counter()
t1 = threading.Thread(target=operation, args=("t1",))
t2 = threading.Thread(target=operation, args=("t2",))
t3 = threading.Thread(target=operation, args=("t3",))
threadList = [t1, t2, t3]
for p in threadList:
    p.start()
for p in threadList:
    p.join()
finish = time.perf_counter()
print(f"\nFinito in {round(finish - start, 2)} secondi")
```

→

Esecuzione di t1
Esecuzione di t2
Esecuzione di t3

Finito in 0.0 secondi



ThreadPoolExecutor:

Anche con i thread esiste un pool che automatizzi l'uso dei threads. Tale pool è detto ***ThreadPoolExecutor*** e restituisce un oggetto ***executor*** nel *context manager*, ottenuto tale oggetto, sarà possibile effettuare operazioni sui threads come creare un thread, passargli la funzione da eseguire ed avviarlo grazie ad ***executor.submit***, il quale accetta come parametri la funzione da eseguire ed i suoi argomenti. Il metodo ***submit()*** ritorna un oggetto ***future***, sul quale è possibile effettuare operazioni riguardanti il thread, per ottenere il valore di ritorno dell'esecuzione della funzione passata al thread viene fatto tramite metodo ***result***. Per utilizzare tale pool è necessario importare il modulo ***concurrent.futures***.

```
def operation(nameThread, seconds):
    print(f"{nameThread} dorme per {seconds}")
    time.sleep(seconds)
    return f"Esecuzione di {nameThread}"
#MAIN-------------------------------------------------------------------
with concurrent.futures.ThreadPoolExecutor() as executor:
    listFutures = []
    for i in range(3):
        listFutures.append(executor.submit(operation, "t"+str(i+1), i+1))
    for i in range(3):
        print(listFutures[i].result())
```

→

p1 dorme per 1
p2 dorme per 2
p3 dorme per 3
Esecuzione di p1
Esecuzione di p2
Esecuzione di p3

CONCORRENZA NELLE COROUTINE:

Per svolgere un insieme di operazioni indipendenti, un approccio può essere quello di effettuare un'operazione alla volta con lo svantaggio che se un'operazione è lenta, il programma deve attendere la fine di questa operazione prima di cominciare la prossima.

Per risolvere questo problema si possono usare le coroutine: ciascuna operazione è una coroutine, un'operazione lenta non influenzerà le altre operazioni almeno fino al momento in cui queste non avranno bisogno di nuovi dati da elaborare. Ciò è dovuto al fatto che le operazioni vengono eseguite indipendentemente.

Una volta che le coroutine non servono più , viene invocato ***close()*** su ciascuna coroutine in modo che non utilizzino più tempo del processore.

Per creare una coroutine bisogna scrivere una funzione che abbia una espressione yield, in un loop infinito, quando la yield viene raggiunta, l'esecuzione della funzione viene sospesa in attesa di dati. Possiamo sfruttare ciò eseguendo funzioni una dopo l'altra linearmente.

Tre coroutines, ***regex_matcher*** vengono salvate in ***matchers***, ognuna riceve come parametri un'altra coroutine ed una stringa ***regex***, formando una catena.

```
receiver = reporter()
matchers = (regex_matcher(receiver, URL_RE), regex_matcher(receiver, H1_RE), regex_matcher(receiver, H2_RE))
```

```
@coroutine
def regex_matcher(receiver, regex):
   while True:
      text = (yield)
      for match in regex.finditer(text):
         receiver.send(match)
```

regex_matcher entra in un loop infinito e subito si mette in attesa che ***yield*** restituisca un testo a cui applicare il ***regex***. Una volta ricevuto il testo, il ***matcher*** itera su ogni *match* ottenuto, inviando ciascun *match* al ***receiver***.

Una volta terminato il matching la coroutine torna a ***yield*** e si sospende nuovamente in attesa di altro testo.

```
try:
  for file in sys.argv[1:]:
    print(file)
    html = open(file, encoding="utf8").read()
    for matcher in matchers:
      matcher.send(html)
finally:
  for matcher in matchers:
    matcher.close()
  receiver.close()
```

Il programma legge i nomi dei file sulla linea di comando e per ciascuno di essi stampa il nome del file e poi salva il testo del file nella variabile *html*.

Il programma itera su tutti i ***matcher*** e invia il testo ad ognuno di essi. Ogni ***matcher*** procede indipendentemente inviando ogni match ottenuto alla coroutine ***reporter***.

Alla fine viene invocato ***close()*** su ciascun ***matcher*** e sul ***reporter*** per impedire che i ***matcher*** rimangano sospesi in attesa di testo e che il ***reporter*** rimanga in attesa di match.

```
@coroutine
def reporter():
  ignore = frozenset({"style.css", "favicon.png", "index.html"})
  while True:
    match = (yield)
    if match is not None:
      groups = match.groupdict()
      if "url" in groups and groups["url"] not in ignore:
        print(" URL:", groups["url"])
      elif "h1" in groups:
        print(" H1: ", groups["h1"])
```

La coroutine ***reporter()*** è usata per dare in output i risultati.

Viene creata dallo statement ***receiver = reporter()*** ed è passata ad ogni ***matcher*** come argomento receiver.

Il ***reporter()*** attende che gli venga spedito un match, quindi stampa i dettagli del *match* e poi continua ad attendere in un loop infinito fino a quando viene invocato ***close()*** su di esso.

DECORATORI DI FUNZIONI:

È un modello di progettazione *strutturale* che consente di associare nuovi comportamenti agli oggetti, posizionandoli all'interno di oggetti ***wrapper*** speciali che contengono comportamenti.

I ***decoratori di funzione*** prendono come argomento una funzione per poi restituirne una funzione ***wrapper***, è un involucro contenente codice aggiuntivo che verrà eseguito prima dell'esecuzione della funzione originale.

```
def double(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        newList = list()
        for item in args:
            item *= 2
            newList.append(item)
        args = newList
        return func(*args, **kwargs)
    return wrapper

@double
def add(a, b):
    return a+b

#MAIN-------------------------------
print(add(2,3))              → 10
print(add.__name__)          → add
print(add.__doc__)           → None
```

wrapper accetta gli argomenti della funzione ***add (*args, **kwargs)*** e li moltiplica per 2.

Dal ***wrapper*** viene ritornata la funzione decorata coi parametri modificati, mentre dal decoratore viene ritornata la funzione wrapper, che verrà eseguita prima di add.

La funzione decorata avrà il valore dell'attributo ***__name__*** settato a "*wrapper*" invece che con il nome originale e non ha una ***docstring***, anche se viene esplicitata. Per ovviare a questo problema, si usa il decoratore ***@functools.wraps*** che può essere usato per decorare il ***wrapper*** dentro il decoratore, e assicurare che ***__name__*** e ***__doc__*** contengano i valori della funzione originale.

DECORATORI DI CLASSE:

I ***decoratori di classe*** prendono come argomento una classe, permettendo modifiche in ogni aspetto e funzione.

```
def dec_counterClass(decoratedClass):
    decoratedClass.numberOfInstances = 0
    decoratedClass.oldInit = decoratedClass.__init__

    def moddedInit(self, *args, **kwargs):
        decoratedClass.numberOfInstances += 1
        decoratedClass.oldInit(self, *args, **kwargs)

    decoratedClass.__init__ = moddedInit
    return decoratedClass

@dec_counterClass
class Counter:
    pass

#Main-------------------------------------------------
a = Counter()
b = Counter()
c = Counter()
print(Counter.numberOfInstances)          → 3
```

Vogliamo fare in modo che la classe, alla creazione di una propria istanza, incrementi un contatore, tenendo conto di tutte le istanze della classe.

Si stabilisce una nuova funzione ***init*** (ovvero ***moddedInit***) che incrementi il contatore e che esegua il vecchio init, salvato in una variabile (***oldInit***), successivamente tale ***moddedInit*** viene salvato nella variabile init originaria. Infine viene ritornata la classe modificata.

PROPERTY:

Si può avere la necessità di variabili di istanza o classe che possano essere modificate solo con alcuni parametri o che possano essere utilizzate solo in alcune condizioni, limitandone l'accesso.

```
class MyClass:
        def __init__(self):
            self.__var = 0

        def getter(self):
            print("GETTER: ")
            return self.__var

        def setter(self, value):
            print("SETTER: ",value)
            if value >0:
                self.__var=value

        def deleter(self):
            print("DELETER: ", self)
            del self.__var

        variabile = property(getter, setter, deleter, "Doc")

#Main-------------------------------------------------------
var = MyClass()
var.variabile = 10                  → SETTER: 10
print(var.variabile)                → GETTER: 10
del var.variabile                   → DELETER: <MyClass>
```

Grazie alla funzione ***property***, una variabile con lo stesso nome della variabile privata (*senza underscore*) potrà esser utilizzata come una normale variabile ma facendo riferimento alla privata, utilizzando le associazioni al posto delle funzioni.

PRPERTY COI DECORATORI:

```
class MyClass:
    __variabile = 0

    @property
    def variabile(self):
        print("GETTER: ")
        return self.__variabile

    @variabile.setter
    def variabile(self, value):
        print("SETTER: ", value)
        if value > 0:
            self.__variabile = value

    @variabile.deleter
    def variabile(self):
        print("DELETER: ")
        del self.__variabile

#Main-------------------------------------
var = MyClass()
var.variabile = 10      → SETTER: 10
print(var.variabile)    → GETTER: 10
del var.variabile       → DELETER: <MyClass>
```

Il getter avrà come descrittore @property, il setter variabile.setter e il deleter @variabile.deleter.

ENSURE:

Questo tipo di decoratore permette di aggiungere ***getter***, ***setter*** e ***deleter*** ad un qualsiasi numero di variabili di una classe, evitando di scrivere codice ridondante.

```
def validation (value):
    if not isinstance(value, int):
        raise ValueError("This value is bad")

def ensure(variableName, validationFunction, documentString):
    def decorator(ClassToEdit):
        privateVar= "__"+ variableName
        setattr(ClassToEdit, privateVar, 0)
        def getter(self):
            return getattr(self, privateVar)
        def setter(self, value):
            validationFunction(value)
            setattr(self, privateVar, value)

        setattr(ClassToEdit, variableName, property(getter, setter, documentString))
        return ClassToEdit
    return decorator

@ensure("myVariable", validation, "Document")
class MyClass:
    pass

#Main-------------------------------------------------------------------------
myVar= MyClass()
myVar.myVariable= 10
print(myVar.myVariable)                                    → 10
```

SINGLETON:

È un modello di progettazione *creazionale* che consente di garantire che una classe abbia solo un'istanza, fornendo al contempo un punto di accesso globale a questa istanza.

```
class Singleton:
    __instance = None

    def __init__(self, Class, *params):
        if Singleton.__instance is None:
            Singleton.__instance = Class(*params)

    def __getattr__(self, attr):
        return getattr(self.__instance, attr)

    def __setattr__(self, attr, newValue):
        return setattr(self.__instance, attr, newValue)

    def getid(self):
        return id(self.__instance)

class MyClass:
    pass

#Main-------------------------------------------------
s1 = Singleton(MyClass)
s1.var = 10
print("ID : ", s1.getid())          → ID :  14561712
print("VAR: ", s1.var)              → VAR:  10
s2 = Singleton(MyClass)
print("ID : ", s2.getid())          → ID :  14561712
print("VAR: ", s2.var)              → VAR:  10
```

Al momento della sua istanziazione, il singleton accetta come parametro una classe e dei parametri, i quali inizializzeranno la classe passata come parametro.

Il problema è che passando la classe dall'esterno, essa è comunque istanziabile più volte tramite il suo costruttore: il singleton deve impedire ciò, perciò la classe viene specificata privata proprio per evitare che essa venga istanziata molteplici volte.

BORG:

Consento di creare più istanze di classe, ma queste condividono tra loro lo stesso stato. Il concetto di Borg è proprio quello di cambiare il ***__dict__*** di default quando si istanzia l'oggetto con la ***__new__*** : si stabilisce un dizionario di classe che conterrà tutte le variabili di istanza, in modo tale che siano comuni a tutte le istanze della classe Borg.

```
class Borg:
    __varDict= {}

    def __new__(cls, *args, **kwargs):
        objToReturn = super().__new__(cls, *args, **kwargs)
        objToReturn.__dict__= cls.__varDict
        return objToReturn

#MAIN-------------------------------------------------------
first = Borg()
second = Borg()
third = Borg()
first.myVar = 10
print(second.myVar)                                  → 10
second.anotherVar = 20
print(third.anotherVar)                              → 20
```

Quando viene creata un'istanza di una classe, viene invocato prima ***__new__*** (che crea l'oggetto), accettando ***cls*** come primo parametro perché quando viene invocato di fatto l'istanza deve essere ancora creata, e poi ***__init__*** (che inizializza le variabili di istanza).

new crea una nuova istanza di ***cls*** invocando il metodo ***__new__*** della superclasse, che modifica l'istanza appena creata.

Se ***new*** restituisce un'istanza di ***cls*** allora viene invocata la ***init*** con gli stessi args.

ADAPTER:

È un modello di progettazione *strutturale* che consente la collaborazione di oggetti con interfacce incompatibili, permette ad una classe di adattarsi ad un'altra tramite una semplice interfaccia, senza cambiare il proprio codice.

```
class Computer:
    def __init__(self, name):
        self.name = name
    def execute(self):
        return f"The {self.name} Computer is executing"

class Synthesizer:
    def __init__(self, name):
        self.name = name
    def play(self):
        return f"The {self.name} Synthesizer is playing"

class Human:
    def __init__(self, name):
        self.name = name
    def speak(self):
        return f"The {self.name} Human is speaking"

#Adattatore-------------------------------------------------------
class Adapter:
    def __init__(self, obj, dictMethods):
        self.obj = obj
        self.__dict__.update(dictMethods)

#MAIN-------------------------------------------------------------
pc = Computer("MyPC")
synth = Synthesizer("MySynth")
human = Human("MyHuman")

lista = [pc]
lista.append(Adapter(synth, dict(execute=synth.play)))
lista.append(Adapter(human, dict(execute=human.speak)))

for item in lista:                  The MyPC Computer is executing
    print(item.execute())       →   The MySynth Synthesizer is playing
                                    The MyHuman Human is speaking
```

L'adapter prende come argomenti un oggetto e un dizionario.

L'oggetto viene conservato nella variabile di istanza, mentre il dizionario è una coppia ***attributo-metodo***, in modo da salvare nel dizionario degli attributi, ovvero un attributo ***execute*** che contenga il metodo da richiamare.

PROXY:

È un modello di progettazione *strutturale* che consente di fornire un sostituto. Un proxy controlla l'accesso all'oggetto originale, consentendo di eseguire qualcosa prima o dopo che la richiesta arriva all'oggetto originale.

```
class MyClass:
    def __init__(self, name):
        self.name = name
    def hello(self):
        print("Hello I am:", self.name)
    def goodbye(self):
        print("Goodbye from: ", self.name)

class GenericProxy:
    def __init__(self, obj):
        self.__internalObj = obj
    def __getattr__(self, attrName):
        print("CONTROLPROXY")
        return getattr(self.__internalObj, attrName, None)

#Main-------------------------------------------------------
proxy = GenericProxy(MyClass("Mario"))
proxy.hello()                → CONTROLPROXY Hello I am: Mario
proxy.goodbye()              → CONTROLPROXY Goodbye from: Mario
```

Il proxy fa da interfaccia completa all'oggetto che effettuerà le operazioni, implementa tutte le funzioni dell'oggetto interno.

Ovviamente il proxy può non implementare tutte le funzioni dell'oggetto interno, in modo da limitare le operazioni eseguibili dall'esterno

CHAIN OF RESPONSBILITY:

È un modello di progettazione *comportamentale* che consente di passare le richieste lungo una catena di gestori. Alla ricezione di una richiesta, ciascun gestore decide di elaborare la richiesta o di passarla al gestore successivo nella catena.

```
def coroutine(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        generator = func(*args, **kwargs)
        next(generator)
        return generator
    return wrapper

@coroutine
def KeyboardHandler(successor=None):
    while True:
        event = (yield)
        print("Invocazione KEYBOARD")
        if event == "Keyboard":
            print("Keyword is pressed")
        elif successor is not None:
            successor.send(event)

@coroutine
def MouseHandler(successor=None):
    while True:
        event = (yield)
        print("Invocazione MOUSE")
        if event == "Mouse":
            print("Mouse is pressed")
        elif successor is not None:
            successor.send(event)

# MAIN-------------------------------------------------------------
chain = MouseHandler(KeyboardHandler(None))  Invocazione MOUSE
chain.send("Mouse")                        → Mouse is pressed
print("\n")
chain.send("Keyboard")                     → Invocazione MOUSE
                                             Invocazione KEYBOARD
                                             Keyword is pressed
```

Un generatore è una funzione che ha una o più espressioni ***yield*** al posto dell'istruzione return.

Una coroutine usa anch'essa l'espressione ***yield***, ma con un differente comportamento: viene eseguito un loop infinito e si sospende ogni volta che si raggiunge uno ***yield***, in attesa di un valore da gestire con quest'ultima espressione.

Se alla coroutine viene passato un valore, quest'ultimo verrà gestito dalla ***yield*** e continuerà a ciclare, fino ad arrivare nuovamente alla ***yield*** che attenderà un nuovo valore.

I valori vengono inseriti nella coroutine tramite i metodi ***send()*** o ***throw()***.

STATE SENSITIVE:

È un modello di progettazione *comportamentale* che consente a un oggetto di modificarne il comportamento quando cambia il suo stato. Nello ***State Sensitive Pattern*** cambia il comportamento dei metodi in base allo stato assunto.

```
class Multiplexer:
    OFF, ON = [0, 1]
    def __init__(self):
        self.state = Multiplexer.ON
    def connect(self):
        if self.state == Multiplexer.ON:
            print("Multiplexer CONNECTED")
        else:
            print("Operation not performed - multiplexer off")
    def disconnect(self):
        if self.state == Multiplexer.ON:
            print("Multiplexer DISCONNECTED")
        else:
            print("Operation not performed - multiplexer off")
#MAIN-----------------------------------------------------------------
multiplexer = Multiplexer()
multiplexer.connect()      → Multiplexer CONNECTED
multiplexer.disconnect()   → Multiplexer DISCONNECTED
multiplexer.state = Multiplexer.OFF
multiplexer.connect()      → Operation not performed-multiplexer is off
```

STATE SPECIFIC:

Nello ***State Specific Pattern*** vengono utilizzati metodi diversi in base allo stato assunto. Lo stato diventa una proprietà, avrà un metodo per ottenere e settare lo stato ed i metodi sono privati, da eseguire in base allo stato assunto.

```
class MultiplexerSpecific:
    OFF, ON = [0, 1]
    def __init__(self):
        self.state = MultiplexerSpecific.ON     #Chiama il setter

    @property
    def state(self):
        if self.connect != self.__active_connect:
            return MultiplexerSpecific.OFF
        else:
            return MultiplexerSpecific.ON
    @state.setter
    def state(self, newState):
        if newState == MultiplexerSpecific.ON:
            self.connect = self.__active_connect
            self.disconnect = self.__active_disconnect
        else:
            self.connect = lambda *args: None
            self.disconnect = lambda *args: None

    def __active_connect(self):
        print(f"Connect, current status: {self.print_status()}")
    def __active_disconnect(self):
        print(f"Disconnect, current status: {self.print_status()}")
    def print_status(self):
        if self.state == MultiplexerSpecific.ON:
            return MultiplexerSpecific.ON
        else:
            return MultiplexerSpecific.OFF

#MAIN-----------------------------------------------------------------
multiplexer = MultiplexerSpecific()
multiplexer.connect()              → Connect,current status:on
multiplexer.disconnect()           → Disconnect,current status:on
multiplexer.state = MultiplexerSpecific.OFF
multiplexer.connect()              →
```

MEDIATOR(convenzionale):

È un modello di progettazione *comportamentale* che consente di ridurre le dipendenze caotiche tra gli oggetti. Il modello limita le comunicazioni dirette tra gli oggetti e li costringe a collaborare solo tramite un oggetto mediatore.

```python
class Component1:
    def __init__(self, mediator = None):
        self.__mediator = mediator
    @property
    def mediator(self):
        return self.__mediator
    @mediator.setter
    def mediator(self, mediator):
        self.__mediator = mediator
    def method1(self):
        print("Component1 esegue method1")
        self.mediator.notify(self)
    def response1(self):
        print("Component1 ha risposto")

class Component2():
    def __init__(self, mediator = None):
        self.__mediator = mediator
    @property
    def mediator(self):
        return self.__mediator
    @mediator.setter
    def mediator(self, mediator):
        self.__mediator = mediator
    def method2(self):
        print("Component2 esegue method2")
        self.mediator.notify(self)
    def response2(self):
        print("Component2 ha risposto")

class Mediator():
    def __init__(self, component1, component2):
        self.__component1 = component1
        self.__component2 = component2
        self.__component1.mediator = self
        self.__component2.mediator = self
    def notify(self, sender):
        if sender == self.__component1:
            print("Mediator reagisce a Component1 e inoltra a Component2")
            self.__component2.response2()
        elif sender == self.__component2:
            print("Mediator reagisce a Component2 e inoltra a Component1")
            self.__component1.response1()

#MAIN--------------------------------------------------------------------------
c1 = Component1()
c2 = Component2()
mediator = Mediator(c1, c2)

c1.method1()
print("\n", end="")
c2.method2()
```

→

```
Component1 esegue method1
Mediator reagisce a Component1 e inoltra a Component2
Component2 ha risposto

Component2 esegue method2
Mediator reagisce a Component2 e inoltra a Component1
Component1 ha risposto
```

(1)

(1) Mediator incapsula le relazioni tra i vari componenti, mantenendo i loro riferimenti. I componenti non devono essere a conoscenza di altri componenti. Se succede qualcosa di importante all'interno o verso un componente, deve solo informare il mediatore. Quando il mediatore riceve la notifica, può facilmente identificare il mittente, che potrebbe essere sufficiente per decidere quale componente deve essere attivato in cambio.

MEDIATOR(coroutine):

```
def coroutine(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        generator = func(*args, **kwargs)
        next(generator)
        return generator
    return wrapper

class Component1:
    def __init__(self):
        self.__mediator = None
    @property
    def mediator(self):
        return self.__mediator
    @mediator.setter
    def mediator(self, mediator):
        self.__mediator = mediator
    def method1(self):
        print("Component1 esegue method1")
        self.mediator.send("method1")
    def response1(self):
        print("Component1 ha risposto")

class Component2:
    #init, getter e setter uguali a Component1
    def method2(self):
        print("Component2 esegue method2")
        self.mediator.send("method2")
    def response2(self):
        print("Component2 ha risposto")

class MediatorSet:
    def __init__(self, component1, component2):
        self.__component1 = component1
        self.__component2 = component2
        self.mediator = self.component1Mediator(self.component2Mediator())
        self.__component1.mediator = self.mediator
        self.__component2.mediator = self.mediator
    @coroutine
    def component1Mediator(self, successor = None):
        while True:
            event = (yield)
            print("component1Mediator interpellato")
            if event == "method1":
                self.__component2.response2()
            elif successor is not None:
                successor.send(event)
    @coroutine
    def component2Mediator(self, successor=None):
        while True:
            event = (yield)
            print("component2Mediator interpellato")
            if event == "method2":
                self.__component1.response1()
            elif successor is not None:
                successor.send(event)

#MAIN----------------------------------------------------------------------
c1 = Component1()
c2 = Component2()
mediator = MediatorSet(c1, c2)
c1.method1()
print("\n", end="")
c2.method2()
```

→

```
Component1 esegue method1
component1Mediator interpellato
Component2 ha risposto

Component2 esegue method2
component1Mediator interpellato
component2Mediator interpellato
Component1 ha risposto
```

Un mediatore può essere considerato come una pipeline che riceve messaggi per poi passarli agli oggetti interessati: ciò può esser realizzabile tramite le coroutine.

TEMPLATE METHOD:

È un modello di progettazione *comportamentale* che definisce lo scheletro di un algoritmo nella superclasse ma consente di definire un algoritmo affidandone l'esecuzione ad opportune sottoclassi.

```
class AbstractWordCouter:
    @staticmethod
    def canCount(filename):
        raise NotImplementedError
    @staticmethod
    def count(filename):
        raise NotImplementedError

class TextWordCounter(AbstractWordCouter):
    @staticmethod
    def canCount(filename):
        return filename.lower().endswith(".txt")
    @staticmethod
    def count(filename):
        if TextWordCounter.canCount(filename):
            counter=0
            with open(filename, "r") as f:
                for line in f:
                    arrayOfWords = line.split()
                    print(arrayOfWords)
                    for word in arrayOfWords:
                        counter +=1
            return counter

#MAIN----------------------------------------------------------------
print("There are", TextWordCounter.count("testFile.txt"), "words")
                    → ['Programming', 'with', 'Python', '!!!']
                      There are 4 words
```

Ogni metodo definito nella superclasse viene dichiarato statico: questo perché non si vede l'utilità di salvare alcuno stato nelle istanze o nella classe.

Se la classe può effettuare il conteggio di parole sul file indicato, allora si esegue il conteggio e si restituisce il valore, se la classe non può effettuare il conteggio di parole, allora viene restituito 0.

OBSERVER:

È un modello di progettazione *comportamentale* che consente di definire un meccanismo di sottoscrizione per notificare a più oggetti eventuali eventi che si verificano sull'oggetto che stanno osservando.

```
class Observed:
    def __init__(self):
        self.__observers = set()
    def observers_add(self, observer, *observers):
        for observer in chain((observer,), observers):
            self.__observers.add(observer)
            observer.update(self)
    def observer_discard(self, observer):
        self.__observers.discard(observer)
    def observers_notify(self):
        for observer in self.__observers:
            observer.update(self)

#Oggetto Osservato:
class SliderModel(Observed):
    def __init__(self, minimum, value, maximum):
        super().__init__()
        self.__minimum = self.__value = self.__maximum = None
        self.minimum = minimum
        self.value = value
        self.maximum = maximum
    @property
    def value(self):
        return self.__value
    @value.setter
    def value(self, value):
        if self.__value != value:
            self.__value = value
            self.observers_notify()

#Oggetto Osservatore:
class HistoryView:
    def __init__(self):
        self.data = []
    def update(self, model):
        self.data.append((model.value, time()))

#MAIN----------------------------------------------------------------
historyView = HistoryView()

model = SliderModel(0, 0, 40)
model.observers_add(historyView)

for value in (7, 23, 37):
    model.value = value

for value, timestamp in historyView.data:
    print("{:3} {}".format(value, datetime.fromtimestamp(timestamp)))
                          → 0 2019-12-21 18:08:43.890876
                            7 2019-12-21 18:08:43.890876
                           23 2019-12-21 18:08:43.890876
                           37 2019-12-21 18:08:43.890876
```

Observer mantiene un insieme di oggetti osservatori ed è caratterizzata da funzioni capaci di gestire gli osservatori.

SliderModel eredita ***Observer*** acquisendo un insieme di osservatori inizialmente vuoto ed i metodi per la gestione degli osservatori, implementati da ***Observer***. Ogni volta che il modello cambia il proprio valore, vengono notificate tutte le viste tramite il metodo ***notifyObserver***.

HistoryView è un osservatore del modello, fornisce un metodo update che verrà richiamato ad ogni cambiamento di un qualsiasi modello collegato.

FACADE:

È un modello di progettazione *strutturale* che fornisce un'interfaccia semplificata a una libreria, un framework o qualsiasi altro insieme complesso di classi.

```
class SubSystem1:
    def __init__(self, val):
        self.value1 = val
    def myFunction1(self):
        print("Valore SubSystem1 : ", self.value1)

class SubSystem2:
    def __init__(self, val):
        self.value2 = val
    def myFunction2(self):
        print("Valore SubSystem2 : ", self.value2)
    def myFunction2double(self):
        print("Valore SubSystem2 : ", self.value2*2)

class Facade:
    def __init__(self, item1, item2):
        self.__subsystem1 = item1
        self.__subsystem2 = item2
    def executeAll(self):
        print("Facade inizializza i sottosistemi: ")
        self.__subsystem1.myFunction1()
        self.__subsystem2.myFunction2()
        self.__subsystem2.myFunction2double()

#MAIN-------------------------------------------------------
sub1 = SubSystem1(100)
sub2 = SubSystem2(200)
myFacade = Facade(sub1, sub2)
myFacade.executeAll()        → Facade inizializza i sottosistemi
                               Valore SubSystem1 :  100
                               Valore SubSystem2 :  200
                               Valore SubSystem2 :  400
```

Le classi dei sottosistemi non sono a conoscenza dell'esistenza della facciata. Operano all'interno del sistema e lavorano direttamente tra loro.

Il client utilizza la facciata invece di chiamare direttamente gli oggetti del sottosistema.

***Context Managers*:**

Permettono di allocare e rilasciare risorse in maniera decisamente semplificata. Esempio la keyword "***with***", la quale apre un file, opera all'interno e lo chiude automaticamente.

```
with MyFile("testFile.txt", "w") as myFile:
    myFile.write("Hello World")
```

=

```
myFile = open("testFile.txt", "w")
myFile.write("Hello World?")
myFile.close()
```

Una classe ***context manager*** deve forzatamente definire i metodi ***__enter__*** ed ***__exit__*.**

__enter__ deve poter ritornare l'oggetto allocato, mentre ***__exit__*** deve poter liberare la memoria riservata all'oggetto allocato. L'allocazione dell'oggetto e la sua inizializzazione avvengono nell'init.

```
class MyFile:
    def __init__(self, filename, method):
        self.obj_file = open(filename, method)
        self.obj_file.write("INIZIALIZZAZIONE \n")
    def __enter__(self):
        return self.obj_file
    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is not None:
            print("Exception not handled!")
            self.obj_file.close()
            return False
        else:
            self.obj_file.close()

#MAIN-------------------------------------------
with MyFile("testFile.txt", "w") as myFile:
    myFile.write("Hello World")
```

1. Eseguita l'istruzione "***with MyFile(…)***", viene aperto il blocco ***with*** e viene eseguita la ***__init__*** della classe ***MyFile***, la quale apre un file e salva il riferimento in una variabile di istanza.
2. Eseguita l'istruzione ***as myFile***, richiamando il metodo ***__enter__,*** che deve restituire la variabile di istanza, ***obj_file***.
3. Eseguite le istruzioni interne al blocco ***with***.
4. Eseguito il metodo ***__exit__***, che libela la memoria allocata da ***__init__*.**

FLYWEIGHT:

È un modello di progettazione *strutturale* che consente di adattare più oggetti alla quantità disponibile di RAM condividendo parti comuni di stato tra più oggetti invece di conservare tutti i dati in ciascun oggetto.

```
class Point:
    __slots__ = ("x", "y", "z", "color")

    def __init__(self, x,y,z,color = None):
        self.x = x
        self.y= y
        self.z= z
        self.color = color

#MAIN-------------------------------------
myPoint= Point(1,2,3,"Blue")
print(myPoint.x)                → 1
print(myPoint.color)            → Blue
```

Conserva le variabili per la raffigurazione tridimensionale senza l'utilizzo del dict, in modo da risparmiare spazio in RAM. In tal modo nessun Point avrà il proprio dict privato. Tuttavia significa che non è possibile aggiungere attributi ai singoli punti, siccome la memoria sarà stata allocata staticamente.

Tale pattern serve a migliorare le prestazioni del programma, o meglio la memoria RAM richiesta per l'allocazione di determinati oggetti.

***La variabile __slots__*:**

Permette di risparmiare RAM: essendo il dict davvero pesante, è possibile sostituirlo con ***__slots__***, il quale occupa circa il 40-50% di spazio in meno. Python usa un dict per conservare gli attributi di un'istanza, allocando memoria dinamica per ogni dict. Python non può allocare un quantitativo statico di memoria alla creazione dell'oggetto per poter conservare gli attributi, è possibile utilizzare ***__slots__*** per allocare spazio solo per quel determinato set di attributi.

È bene precisare che l'utilizzo di ***__slots__*** *esclude* l'utilizzo della variabile ***dict*** contenente le variabili di istanza: ***__slots__*** conterrà le variabili di istanza allocando staticamente memoria, dopo l'inizializzazione, non sarà possibile aggiungere nuove variabili di istanza.

PROTOTYPE:

È un modello di progettazione *creazionale* che consente di copiare oggetti esistenti senza che il codice dipenda dalle loro classi.

```
class Point:
    __slots__= ("x", "y")
    def __init__(self,x,y):
        self.x = x
        self.y = y

#MAIN-------------------------------------------------------------------------
original = Point(1,2)
prototype = copy.deepcopy(original)
prototype.x= 10
prototype.y= 20
print(f"ORIGINAL : x={original.x}, y={original.y}")    → ORIGINAL : x=1, y=2
print(f"PROTOTYPE: x={prototype.x}, y={prototype.y}") → PROTOTYPE: x=10, y=20
```