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…

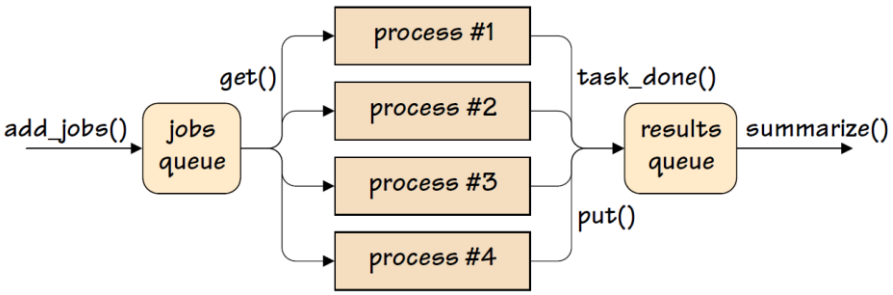
→

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).

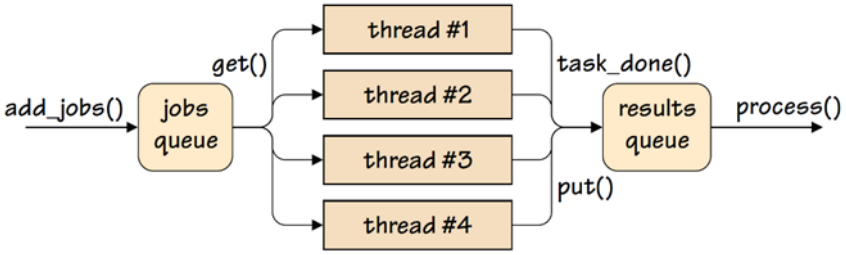
Importiamo il modulo ***threading*** ed utilizzando la classe ***Thread*** per istanziare i threads: proprio come già visto con i processi, basta istanziare un oggetto ***Thread*** passandogli la funzione da eseguire come argomento, per poi avviarlo tramite il metodo ***start()*** ed attenderlo tramite 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.