Python

Advanced features

Python: ancora sulle funzioni

In python, funzioni e metodi sono oggetti.

Una categoria di oggetti particolari, si chiamano “callable” che come suggerisce il termine possono essere chiamati.

Essendo oggetti, seguono le regole delle variabili. Possono essere creati, distrutti, passati come parametro, copiati, ri-definiti etc…

Esempio

```
def funzione(a):
    print(a)


funzione(5)
b=funzione
b(10)
del funzione
funzione(10)
```

```
5
10
Traceback (most recent call last):
  File ".\funz.py", line 9, in <module>
    funzione(10)
NameError: name 'funzione' is not defined
```

Side effect del passaggio di metodi\funzioni: **duck typing**

Ingredienti:

Due classi diverse, non legate tra loro tra relazioni di ereditarietà…

Due metodi con nomi uguali in ciascuna di queste classi…

Una dose abbondante di tipizzazione dinamica…

… il gioco è fatto….

Implementazione di esempio

```
class Duck:
    def verso(self):
        print("QUACK")
class Donkey:
    def verso(self):
        print("IOOOOHH")


def fai_verso(a):
    a.verso()


fai_verso(Duck())
fai_verso(Donkey())
```

Questa particolarità è nota come “duck typing”:

"If it walks like a duck and it quacks like a duck, then it must be a duck"

Il programma stamperà “QUACK” ed “IOOOOHH”: assistiamo quindi ad una sorta di polimorfismo pur non essendo le classi Duck e Donkey imparentate tra di loro.

Ciò è reso possibile dalla natura interpretativa di python, unitamente alla sua gestione completamente dinamica del concetto di “tipo di dato”.

Altri trabocchetti: function\method default args\params

Abbiamo visto come una funzione\metodo python sia in grado di accettare dei parametri di default:

```
def stampa_nome(nome=”inserisci nome”):
    print(“Ciao ” + nome)
```

e quindi possibile chiamare:

```
stampa_nome() #senza parametri
```

Quindi?

Quindi occorre chiedersi **quando** questi parametri vengono istanziati: ricordatevi che tutto in python è un oggetto. Quindi dinamico, quindi allocato dinamicamente!

- I parametri di default vengono istanziati ogni volta che viene invocato il metodo\funzione
- I parametri di default vengono istanziati una volta sola
- Se esiste, sfrutterà un oggetto passato esplicitamente da una precedente invocazione

Quale delle tre? Scopriamolo con degli esempi.

```
class Oggetto:
    def __init__(self):
        print("Mi creo")
        self.lista= "l i s t a".split()


def funzione(o=Oggetto()):
    o.lista.insert(0,"X")
    for e in o.lista:
        print(e,end="")
```

```
print("Prima invocazione def param")
funzione()
print("\n==================")
print("Seconda invocazione con parametro")
o = Oggetto()
o.lista.append("X")
funzione(o)
print("\n==================")
print("Terza invocazione senza parametro")
funzione()
print("\n==================")
```

Output

```
Mi creo

Prima invocazione def param

Xlista

==================

Seconda invocazione con parametro

Mi creo

XlistaX

==================

Terza invocazione senza parametro

XXlista

==================
```

Prima invocazione con parametro non passato\di default. L'oggetto viene creato\inizializzato. Lo si vede dalla stampa. Il programma funziona "as expected".

Seconda invocazione con parametro passato esplicitamente. L'oggetto viene **da noi** creato\inizializzato. Il programma funziona "as expected".

Terza invocazione, fatta come la prima. **Nessun** terzo oggetto viene creato\inizializzato (no stampe "mi creo"). A giudicare dall'output, **viene modificato l'attributo dell'oggetto instanziato nel primo caso**.

Riassumendo

- I parametri di default vengono istanziati ogni volta che viene invocato il metodo\funzione **NO**
- I parametri di default vengono istanziati una volta sola **SI**
- Se esiste, sfrutterà un oggetto passato **esplicitamente** da una precedente invocazione **NO**

Implicazioni:

E’ una pessima idea avere argomenti di default di **tipo mutabile**.

E se volessi comunque esprimere l’idea “se non ti passo un oggetto, createlo da solo **tutte le volte e non una tantum al caricamento del modulo?**

Funzione “migliore”

```
def funzione_migliore(o : Oggetto = None):
    if o==None:
        o = Oggetto()
    [...resto della funzione…]
```

Var args in python

Python, così come tanti altri linguaggi ammette il concetto di var args nei propri metodi\funzioni.

Con var args s’intende la possibilità di specificare un numero variabile di argomenti in ingresso ad un “callable”.

In Python esistono strutture grammaticali specifiche per esprimere al meglio questo concetto. **Si parla quindi di lista di Non-keyword args e Keyword-args**

List di non keyword args.

Ho una generica lista di parametri da passare ad un metodo\funzione.

Ciascun parametro non si differenzia “semanticamente” da quello successivo.

Ecco come si usa:

```
def fvarargs_v1(*args):
    for a in args:
        print(a)
```

```
def fvarargs_v2(a,b,c,d,e,f):
    for i in [a,b,c,d,e,f]:
        print(i)
```

```
l = ["a",4,4.3,True,[1,2],False]
fvarargs_v1(*l)
print("========")
fvarargs_v2(*l)
```

Spiegazione

L’operatore unario ***** “esplode” strutture dato come per esempio liste per riempire i singoli parametri in ingresso alle nostre funzioni.

La prima versione della funzione accetta un numero imprecisato di argomenti.

La seconda versione ne accetta un numero fisso.

Grammaticalmente, non ci sono differenze nel modo in cui vengono chiamate.

Se “l” fosse delimitato da “(“ e “)” anzichè “[” e ”]” l’esplosione di parametri funziona ugualmente.

In entrambi i casi abbiamo una dinamicità nel numero di parametri, siano essi racchiusi in una tupla o lista…

Dict con keyword args.

Ho un dizionario {K:V} di parametri da passare ad un metodo\funzione.

Ogni parametro si differenzia “semanticamente” da quello successivo.

```
d = { "eta":77, "l_mansioni": ["Giardino", "Reception", "Magazzino"],
"nome":"Mario", "cognome":"Rossi"}

dict_varargs_v1(**d)
dict_varargs_v2(**d)
```

```
def dict_varargs_v1(**kwargs):
 for e in kwargs:
    print(str(e)+":"+str(kwargs[e]))
```

```
def dict_varargs_v2(nome, cognome, eta, l_mansioni):
    print(nome+" "+cognome+" di anni " +str(eta)+ " si occupa di ")
    for s in l_mansioni:
        print(s)
```

Spiegazione

L’operatore unario ** “esplode” strutture dato a dizionario per riempire i singoli parametri in ingresso alle nostre funzioni. Le chiavi ed i nomi dei parametri della funzione devono essere congruenti in valore e numero.

Si noti come non ci siano requisiti di ordinamento per quello che riguarda il set di chiavi all’interno del dizionario passato come parametro…

La prima versione della funzione accetta un numero imprecisato di argomenti.

La seconda versione ne accetta un numero fisso.

Grammaticalmente, non ci sono differenze nel modo in cui vengono chiamate.

** e * oltre le funzioni

```
>>> l = [*"ciao"]
>>> l
['c', 'i', 'a', 'o']

>>> l1 = [1,2]
>>> l2 = [3,4]
>>> l12 = [*l1,*l2]
>>> l12
[1, 2, 3, 4]

>>> d1 = {"a":1,"b":2}
>>> d2 = {"c":2, "d":3}
>>> d12 = {**d1,**d2}
>>> d12
{'a': 1, 'b': 2, 'c': 2, 'd': 3}
```

Function decorators

Un decoratore in python è una funzione che prende in ingresso una funzione e ne restituisce una versione “pre\post” processata.

Chiariamo questa definizione con un semplice esempio.

Esempio

```
def funz_decoratrice(f):

    def inner(a):

        print("Pre process della funzione")

        f(a)

        print("Post process della funzione")

    return inner



@funz_decoratrice

def funzione(a):

    print("Eseguo la funzione con parametro " +
str(a))




funzione("args")
```

Pre process della funzione
Eseguo la funzione con parametro args
Post process della funzione

Esempio due

```
def div_decor(f):
    def inner(a,b):
        if b==0:
            return "Non posso dividere per zero!"
        return f(a,b)
    return inner


@div_decor
def divisione(a,b):
    return a/b


print(divisione(5,0))
print(divisione(5,2))
```

Non posso dividere per zero!
2.5

Perché abbiamo visto i decorators?

In Django verranno utilizzati per subordinare le funzionalità delle views (quindi i moduli python che dettagliano la logica della nostra applicazione) a logiche di preprocessing.

Esempio: L’accesso da parte di un utente\client ad una pagina scatena un aggiornamento ad un DB gestito lato server. Tale aggiornamento però deve essere subordinato alla condizione da parte dell’utente di essersi autenticato nel sistema.

Python threads

Come in Java, python mette a disposizione diversi costrutti per il multi-threading.

Ne vedremo solo uno, il più semplice che deriva dal package threading.

Esempio: n threads in parallelo

Un thread viene “creato” specificando la sua funzione “target” e gli argomenti di tale funzione.
Una volta creato, verrà eseguito con il metodo “start”.
Una volta che è in esecuzione un altro thread può “aspettare” che finisca…

Vi ricorda qualcosa?

```python
import threading
import time

def func(a):
    print("Ciao! sono il thread " + threading.current_thread().name + "dormiro per " + str(a) + " secs")
    time.sleep(a)
    print(threading.current_thread().name+": ho finito di dormire")


if __name__ == "__main__":
    NUM_THREADS = 2

    print("Sono il " + threading.current_thread().name  + " e gestiro " + str(NUM_THREADS) + " threads")

    l = []
    for i in range(NUM_THREADS):
        t = threading.Thread(target=func, args=(i+1,))
        t.start()
        l.append(t)

    for t in l:
        t.join()

    print(threading.current_thread().name  + " ho finito di aspettare i threads")
```

Sono il MainThread e gestiro 2 threads
Ciao! sono il thread Thread-1dormiro per 1 secs
Ciao! sono il thread Thread-2dormiro per 2 secs
Thread-1: ho finito di dormire
Thread-2: ho finito di dormire
MainThread ho finito di aspettare i threads

asyncio e co-routines: oltre i thread…

Oltre ai thread, versioni recenti di python (>=3.4) permettono il lancio di funzioni e metodi asincroni.

Ciò aggiunge una dimensione di parallelismo aggiuntivo, senza necessariamente coinvolgere thread paralleli al main thread della nostra applicazione.

Queste funzionalità sono in costante evoluzione…

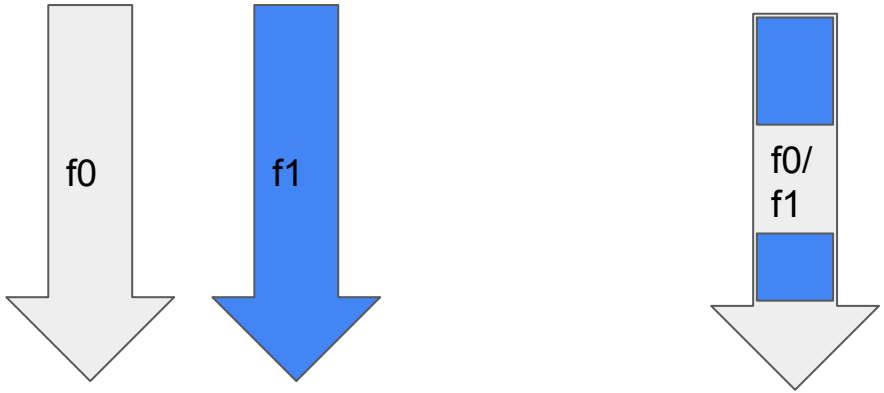
Rationale

Parallelism vs. Concurrency.

Parallelism: l'idea di avere diversi flussi di programma in esecuzione in **parallelo**.

Concurrency: l'idea di avere diversi flussi di programma in esecuzione in finestre temporali che si sovrappongono.

Spieghiamoci meglio…



Essere paralleli: i flussi di istruzioni f0 ed f1 eseguono parallelamente. Possono essere 2 thread ciascuno assegnato ad un CPU core diverso.

Essere in concurrency: f0 ed f1 possono alternarsi in “time-sharing” sullo stesso processore.

Python asyncio

Il package di asyncio e relative nuove keyword di Python (await, run, async …)

permettono al programmatore di specificare flussi di esecuzione di tipo **concurrent and cooperative**.

Inoltre, sono congruenti con quello che in Java si chiama “Future”:

Un flusso parallelo\concorrente di istruzioni che “promette di avere un risultato in futuro” https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html

Miguel Grinberg's 2017 PyCon talk

Chess master Judit Polgár hosts a chess exhibition in which she plays multiple amateur players. She has two ways of conducting the exhibition: synchronously and asynchronously.

Assumptions:

- 24 opponents
- Judit makes each chess move in 5 seconds
- Opponents each take 55 seconds to make a move
- Games average 30 pair-moves (60 moves total)

Synchronous version: Judit plays one game at a time, never two at the same time, until the game is complete. Each game takes *(55 + 5) * 30 == 1800* seconds, or 30 minutes. The entire exhibition takes *24 * 30 == 720* minutes, or **12 hours**.

Asynchronous version: Judit moves from table to table, making one move at each table. She leaves the table and lets the opponent make their next move during the wait time. One move on all 24 games takes Judit *24 * 5 == 120* seconds, or 2 minutes. The entire exhibition is now cut down to *120 * 30 == 3600* seconds, or just **1 hour**.

https://youtu.be/iG6fr81xHKA?t=4m29s

Implementazione seriale

```python
import time

def mia_mossa(tempo):
    time.sleep(tempo)

def mossa_avversaria(tempo):
    time.sleep(tempo)

if __name__ == "__main__":
    OPPONENTS = 5
    OPP_TIME_TO_THINK = 1 #tempo necessario per far pensare l'avversario
    MY_MOVE_TIME = 1 #tempo necessario per far pensare me stesso
    MAX_MOVES = 2 #in quante mosse vogliamo vincere

    # totale = 2*5*(1+1) = circa 20 secs.

    s = time.perf_counter()

    for _ in range(MAX_MOVES):          #per ogni mossa
        for _ in range(OPPONENTS):      #per ogni avversario
            mia_mossa(MY_MOVE_TIME)
            mossa_avversaria(OPP_TIME_TO_THINK)


    e = time.perf_counter()

    print("ELAPSED " + str(e-s))
```

ELAPSED 20.2079932

Implementazione con asyncio

```python
import time
import asyncio

async def mossa(mio_tempo,suo_tempo):
    mia_mossa(mio_tempo)
    await mossa_avversaria(suo_tempo)

def mia_mossa(tempo):
    time.sleep(tempo) #dormi e non fare nient'altro.

async def mossa_avversaria(tempo):
    await asyncio.sleep(tempo) #dormi, ma se hai meglio da fare controlla la lista di altre coroutine da eseguire.

async def main(): #async è necessario per poter essere aspettato!
    OPPONENTS = 5
    OPP_TIME_TO_THINK = 1
    MY_MOVE_TIME = 1
    MAX_MOVES = 2

    # totale = 2 mosse * 5 avversari * 1 (mio tempo di mossa) + spicci... = circa >10

    l=[]
    #riempiamo un array di funzioni di tipo async
    for _ in range(OPPONENTS*MAX_MOVES):
        l.append(mossa(MY_MOVE_TIME,OPP_TIME_TO_THINK))

    await asyncio.gather(*l) #gather schedula ed esegue "awaitable objects" uno dopo l'altro.


if __name__ == "__main__":
    s = time.perf_counter()
    asyncio.run(main()) #aspettiamo che finiscano tutti.
    e = time.perf_counter()
    print("ELAPSED " + str(e-s))
```

ELAPSED 11.1309703

Osservazioni

Chi esegue cosa?

Inserendo “print(threading.current_thread().name)” in ogni funzione

ottengo che ogni funzione è eseguita dal MainThread.

Eppure rispetto alla versione “seriale\sincrona” ci si mette la metà…

Ecco cosa significa “concurrency”: la funzione “await asyncio.sleep(x)” ci dice “dormi per x secondi, e se nel mentre hai altro da fare… fallo pure”.

Spiegazioni delle keyword

async def: indica che la funzione è una co-routine..

await: indica una funzione da “aspettare”. Tale funzione deve essere una co-routine. Ovvero: deve essere in grado di sospendersi per poter lasciare CPU time ad altre funzioni

gather: schedula ed esegue "awaitable objects" uno dopo l'altro

run: esegue ed aspetta il risultato di una o più co-routine.

Pensate alle funzioni async come “generatori”. Esistono persistentemente fino ad esaurimento… il generatore “esaurisce” valori da restituire, la co-routine istruzioni.

Perchè abbiamo visto asyncio?

In django è come vengono implementate i metodi di risposta asincroni e bidirezionali tra client\server tramite canali e websockets, sfruttando il protocollo ASGI.

ASGI (Asynchronous Server Gateway Interface) è il successore spirituale di WSGI, progettato per fornire un’interfaccia standard per i web servers scritti in Python, per i suoi frameworks e applicazioni derivate. Rispetto al suo successore, appunto è in grado di utilizzare co-routines.