Complementi di Programmazione

Debug in Python

CdL Informatica - Università degli studi di Modena e Reggio Emilia
AA 2023/2024

Filippo Muzzini

Debug

come in altri linguaggi, anche in Python vi sono strumenti per effettuare il debug del codice.

- Integrati nell’IDE
- A se stanti (utili in molte situazioni)

Debug

A differenza di altri linguaggi, in Python vi è un modulo per debuggare.

```
import pdb

pdb.run(<stringa di codice>)
```

Debug

run(statement[, globals[, locals]])

Esegue l'istruzione statement (passata come stringa) sotto il controllo del debugger. Il prompt del debugger compare prima che qualunque codice venga eseguito; potete impostare dei breakpoint e digitare "continue", oppure potete avanzare un passo alla volta tra le dichiarazioni utilizzando "step" o "next" (tutti questi comandi vengono spiegati più avanti). Gli argomenti facoltativi globals e locals specificano l'ambiente nel quale il codice viene eseguito; se non specificato diversamente, viene utilizzato il dizionario del modulo __main__. (Vedete la spiegazione dell'istruzione exec o della funzione built-in eval().)

Debug

runeval(expression[, globals[, locals]])

Valuta l'espressione expression (fornita come stringa) sotto il controllo del debugger. Quando runeval()

termina, questa funzione restituisce il valore dell'espressione. Altrimenti questa funzione è simile a run().

runcall(function[, argument, ...])

Esegue la funzione function (una funzione o il metodo di un oggetto, non una stringa) con gli argomenti

forniti. Quando runcall() termina, la funzione restituisce qualunque cosa venga restituito dalla funzione

chiamata. Il prompt del debugger appare al momento dell'entrata nella funzione.

Debug

Oppure lanciato come scirpt

python3 -m pdb <programma>

Il programma che vogliamo debuggare sarà l’argomento

Debug

La maggior parte dei comandi possono venire abbreviati con una o due lettere; per esempio "h(elp)" significa che sia "h" che "help" possono venire utilizzati per avviare l'help (ma non "he","hel", "H", "Help" o "HELP"). Gli argomenti dei comandi devono essere separati da caratteri di spaziatura (spazi o tab). Nella sintassi dei comandi gli argomenti facoltativi vengono racchiusi tra parentesi quadre ("[]"); le parentesi quadre non devono essere digitate. Le varie alternative nella sintassi dei comandi vengono separate da una barra verticale ("|"). Inviando una riga vuota (invio) si otterrà la ripetizione dell'ultimo comando fornito. Eccezione: se l'ultimo comando era "list", vengono elencate le prossime 11 righe.

Debug

w(here)

Stampa la traccia dello stacke, con il frame più recente in fondo. Una freccia indica il frame corrente, che determina il contesto della maggior parte dei comandi. d(own) Sposta il frame corrente in basso di un livello nella traccia dello stack(verso un frame più recente).

u(p)

Sposta il frame corrente in alto di un livello nella traccia dello stacke(verso un frame più vecchio).

b(reak) [[filename:]lineno|function[, condition]]

Con un argomento lineno, imposta in quella riga del file corrente un break. Con un argomento function, imposta un break alla prima istruzione eseguibile in quella funzione. Il numero di riga può essere preceduto da un nome di file seguito da un due punti, in modo da specificare un breakpoint in un'altro file (probabilmente uno che non è ancora stato caricato). Il file viene cercato in sys.path. Notate che ad ogni breakpoint viene assegnato un numero a cui fanno riferimento tutti gli altri comandi dei breakpoint. Se è presente un secondo argomento, esso è un'espressione che deve venire valutata come vera prima che il breakpoint venga rispettato. Senza argomenti, vengono elencati tutti i breakpoint, includendo per ognuno di essi il numero di volte che è stato raggiunto, il contatore corrente dei passi da ignorare e la condizione associata, se presente.

Debug

tbreak [[filename:]lineno|function[, condition]]

Breakpoint temporaneo che viene rimosso automaticamente quando viene raggiunto la prima volta. Gli argomenti sono gli stessi di break.

cl(ear) [bpnumber [bpnumber ...]]

Con una lista di numeri di breakpoint separata da spazi, cancella tutti questi breakpoint. Senza argomenti, cancella tutti i breakpoint (ma prima chiede conferma).

disable [bpnumber [bpnumber ...]]

Disabilita i breakpoint forniti come lista di numeri di breakpoint separati da spazi. Disabilitare un breakpoint significa che esso non può più provocare il blocco dell'esecuzione del programma, ma al contrario della cancellazione del breakpoint, esso resta nella lista dei breakpoint e può venire riabilitato.

enable [bpnumber [bpnumber ...]]

Abilita i breakpoint specificati.

ignore bpnumber [count]

Imposta il contatore dei passi da ignorare per il breakpoint fornito (come numero). Se count viene omesso, il contatore dei passi da ignorare viene impostato a 0. Un breakpoint diventa attivo quando il contatore dei passi da ignorare diventa 0. Quando diverso da zero, il contatore viene decrementato ogni volta che il breakpoint viene raggiunto, a condizione che esso non sia disabilitato e che ogni condizione associata sia stata valutata come vera.

Debug

condition bpnumber [condition]

condition è un'espressione che deve essere valutata come vera prima che il breakpoint venga rispettato. Se condition è assente, tutte le condizioni esistenti vengono rimosse; cioè, il breakpoint viene reso incondizionato.

s(tep)

Esegue la riga corrente, blocca l'esecuzione alla prima occasione possibile (in una funzione chiamata o sulla prossima riga della funzione corrente).

n(ext)

Continua l'esecuzione finché la prossima riga della funzione corrente non viene raggiunta o la funzione termina. (La differenza tra "next" e "step" è che "step" si blocca dentro una funzione chiamata, mentre "next" esegue la funzione chiamata a (circa) piena velocità, fermandosi solo alla prossima riga nella funzione corrente.)

r(eturn)

Continua l'esecuzione fino al termine della funzione corrente.

c(ont(inue))

Continua l'esecuzione, si blocca solo quando viene raggiunto un breakpoint.

j(ump) lineno

Imposta la prossima riga che verrà eseguita. Disponibile solo nel frame più in basso. Questo permette di tornare indietro per rieseguire una porzione di codice o per saltare del codice che non volete eseguire. Attenzione che non tutti i salti sono permessi; tanto per chiarire, non è possibile saltare nel mezzo di un ciclo for o fuori da una clausola finally.

Debug

l(ist) [first[, last]]

Mostra il codice sorgente del file corrente. Senza argomenti, mostra le 11 righe attorno a quella corrente o continua l'elenco precedente. Con un argomento, elenca le 11 righe attorno a quella riga. Con 2 argomenti, mostra le righe nell'intervallo fornito; se il secondo argomento è minore del primo, viene interpretato come un incremento(n.d.T: 11 e 3 indicano le righe dalla 11 alla 14).

a(rgs)

Stampa la lista degli argomenti della funzione corrente.

p expression

Valuta l'espressione expression nel contesto corrente e ne stampa il valore. Note: Si può anche utilizzare "print", ma non è un comando del debugger; esso eseguel'istruzione print di Python.

pp expression

Come il comando "p", ad eccezione del fatto che il valore dell'espressione viene stampato in forma elegante utilizzando il modulo pprint.

Debug

alias [name [command]]

Crea un alias chiamato name che esegue il comando command. Il comando non deve essere racchiuso tra virgolette. I parametri sostituibili possono venire indicati da "%1", "%2" e così via, mentre "%*" viene sostituito da tutti i parametri. Se non viene fornito nessun comando, viene mostrato l'alias corrente di name. Se non viene fornito nessun argomento, vengono elencati tutti gli alias. Gli alias possono venire annidati e possono contenere qualsiasi cosa che sia possibile digitare al prompt di pdb. Notate che i comandi interni di pdb possono venire sovrascritti dagli alias. Un comando sovrascritto rimane perciò nascosto finché l'alias non viene rimosso. Il meccanismo degli alias viene applicato ricorsivamente alla prima parola della riga di comando; tutte le altre parole sulla stessa riga di comando vengono lasciate invariate. Come esempio, ecco due utili alias (specialmente quando inseriti nel file .pdbrc):

#Visualizza le variabili d'istanza (utilizzo: "pi classInst")

alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k]

#Visualizza le variabili d'istanza in self

alias ps pi self

unalias name

Cancella l'alias specificato.

Debug

[!]statement

Esegue l'istruzione (monoriga) statement nel contesto dello stack frame corrente. Il punto esclamativo può venire omesso, a meno che la prima parola dell'istruzione sia anche un comando del debugger. Per impostare una variabile globale potete precedere, sulla stessa riga, il comando d'assegnamento con un comando "global", per esempio:

(Pdb) global list_options; list_options = ['-l']

(Pdb)

q(uit)

Esce dal debugger. Il programma in esecuzione viene interrotto.

Debug

E’ possibile definire porzioni di codice che vengano eseguite in debug.

```
if __debug__:

    …
```

La __debug__ è definita sempre tranne quando si utilizza l’opzione -O che indica all’interprete di ottimizzare il codice per la produzione.

Debug

Vi è un modo più rapido che equivale a **if __debug__**

assert <expr> -> if __debug__:

if not **expr**: raise AssertException

assert <expr1> <expr2> -> if __debug__:

if not **expr1**: raise AssertException(**expr2**)