Python
Debuggeur

Nicolas Delestre

INSA
ROUEN NORMANDIE

Debugguer

Pourquoi utiliser un debuggeur ?

- Les logs permettent de contrôler l'exécution général d'un programme
- Ils ne permettent pas de débugger un algorithme
- Les `print` sont à proscrire

pdb3 (ou ipdb3)

- C'est à la fois un outil et un module
- Il est interactif (ipdb l'est encore plus)
- Il est fourni de base avec python

INSA ROUEN NORMANDIE

Code pour la démonstration 1 / 3

lib_polyligne.py

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

from polyligne import Polyligne
from point import Point2D
import lib_point as lp

def est_a_l_interieur(pt: Point2D, pl: Polyligne) -> bool:
    def angle(pt, pt1, pt2):
        vect1 = lp.vecteur(pt, pt1)
        vect2 = lp.vecteur(pt, pt2)
        if vect1 != vect2:
            return lp.angle(vect1, vect2)
        else:
            return 0

    somme_angles = 0
    pt_courant = pl[0]
    for pt_suivant in pl[1:]:
        somme_angles += angle(pt, pt_courant, pt_suivant)
        pt_courant = pt_suivant
    somme_angles += angle(pt, pt_courant, pl[0])
    return abs(abs(somme_angles) - 3.14159) < 1e-3 # C'est ici le bug 2 * 3.14159
```

Code pour la démonstration 2 / 3

exemple_utilisation_polyligne.py

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

from polyligne import Polyligne
from point import Point2D
from lib_polyligne import est_a_l_interieur

pl = Polyligne(True, Point2D(0,0), Point2D(1,1), Point2D(2,0))
for pt in pl:
    print(pt)
print(est_a_l_interieur(Point2D(1, 0.5), pl))
```

Exécution du script

```
$ python3 exemple_utilisation_polyligne.py
(0, 0)
(1, 1)
(2, 0)
False
```

Code pour la démonstration 3 / 3

Lancement du debuggeur

```
$ ipdb3 exemple_utilisation_polyligne.py
> ... /exemple_utilisation_polyligne.py(4)<module>()
      3
----> 4 from polyligne import Polyligne
      5 from point import Point2D

ipdb>
```

INSA ROUEN NORMANDIE

Les commandes 1 / 4

Affichage

h[com]	*help*, affiche toute les commandes (si suivi d'une commande affiche l'aide de la commande)
l	*list*, affiche quelques lignes de code avant et après l'instruction courante
ll	*long list*, affiche le code du module (ou de la fonction) courant
w	*where*, affiche la pile des appels
p	*print*, suivi de `locals()` (ou `globals()`) permet d'afficher toutes les variables locales (ou globales).
pp	*pretty print*, *idem*
a	*args*, affiche les arguments de la fonction courante
display [expr]	affiche une expression dès qu'elle change (si pas d'argument, affiche toutes les expressions)
undisplay [expr]	arrêt l'affichage d'une ou de toutes les expressions

Les commandes 2 / 4

Exécution

n	*next*, exécute l'instruction courante
s	*step in*, « rentre » dans l'instruction courante
j num	*jump*, va directement à la ligne `num`
unt num	*until*, exécute les instructions jusqu'à ce que leurs numéros de ligne soit $\geq$ `num`
r	*return*, exécute toutes les instructions de la fonction courante
c	*continue*, continue l'exécution du programme normalement jusqu'au prochain point d'arrêt
run|restart [args ...]	relance le debuggeur

INSA ROUEN NORMANDIE

Points d'arrêts

b[[fic :](num|func) [, cond]] *breakpoint*, met un point d'arrêt au fichier *fic* à la ligne numéro *num* ou fonction *fun*. Ce point d'arrêt peut être conditionné par *cond*. Si aucun numéro de ligne ou nom de fonction, alors affiche la liste des points d'arrêt

tbreak *temporary break*, *idem break* mais disparaît au premier arrêt

condition bp [cond] fixe ou annule une condition d'un point d'arrêt

disable [bp [bp ..]] désactive un ou plusieurs points d'arrêt

enable [bp [bp ..]] active un ou plusieurs points d'arrêt

cl [fic :num|bp [bp ...]] *clear*, supprime tous ou quelques points d'arrêt

commands bp [cmd1 ; cmd2... end] associe une liste de commandes (souvent de l'affichage) à un point d'arret

INSA ROUEN NORMANDIE

Alias

alias [nom [instr]] permet d'ajouter (ou de remplacer une commande du debuggeur) en y associant une instruction python. Cette commande peut être paramétrée (utilisation de %1, %2, etc. %*).

unalias nom supprime un alias

Modification de l'état

- Il est possible de modifier l'état du programme par des affectations
- Attention les commandes sont prioritaires (si utilisation d'une variable p dans le code, on ne peut pas faire `p = 0`). Utilisation dans ce cas du préfixe `!` pour indiquer au debuggeur d'utiliser explicitement la variable `!p = 0`.

INSA ROUEN NORMANDIE

PDB et les tests unitaires 1 / 3

test_lib_polyligne.py

```python
#!/usr/bin/python3
# -*- coding: utf-8 -*-

import pytest
from polyligne import Polyligne
from point import Point2D
from lib_polyligne import est_a_l_interieur

@pytest.mark.parametrize("polyligne, pt, resultat_attendu",
                         [(Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                     Point2D(2, 0)), Point2D(1, 0.5), True),
                          (Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                     Point2D(2, 0)), Point2D(0.5, 1), False),
                          (Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                     Point2D(2, 0)), Point2D(0.5, 0.5), False),
                        ])
def test_est_a_l_interrieur(polyligne, pt, resultat_attendu):
    assert est_a_l_interieur(pt, polyligne) == resultat_attendu
```

INSA ROUEN NORMANDIE

PDB et les tests unitaires 2 / 3

Exécution du test unitaire

```
$ py.test-3
===== test session starts =======
platform linux -- Python 3.5.2, pytest-2.8.7, py-1.4.31, pluggy-0.3.1
rootdir: .../Cours/10-PDB/Version1.0/exemples, inifile:
collected 3 items

test_lib_polyligne.py F..

==== FAILURES ==================
___ test_est_a_l_interrieur[polyligne0-pt0-True] ________

polyligne = Polyligne(True, Point2D(0.000000,0.000000), Point2D(1.000000,1.000000), Point2D(2.000000,0.000000)
pt = Point2D(1.000000,0.500000), resultat_attendu = True

    @pytest.mark.parametrize("polyligne, pt, resultat_attendu",
                             [(Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                         Point2D(2, 0)), Point2D(1, 0.5), True),
                              (Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                         Point2D(2, 0)), Point2D(0.5, 1), False),
                              (Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                         Point2D(2, 0)), Point2D(0.5, 0.5), False),
                             ])
    def test_est_a_l_interrieur(polyligne, pt, resultat_attendu):
>       assert est_a_l_interieur(pt, polyligne) == resultat_attendu
E       assert est_a_l_interieur(Point2D(1.000000,0.500000), Polyligne(True, Point2D(0.000000,0.000000), Point

test_lib_polyligne.py:18: AssertionError
```

Lancement du debuggeur au premier test unitaire échoué

- option `--pdb` à `py.test`
- utilisation de la commande *debug* qui permet de débugger une expression python, celle qui a fait échouer le test (utilisation du debugger dans le debugger).

INSA ROUEN NORMANDIE

Le module pdb

En fait. . .

- pdb est un module
- il propose entre autres les fonctions (paramètres formels non précisés ici, voir la documentation [a]) :

`run`	pour débugger une expression python représentée par une chaîne
`runcall`	pour debugger une fonction
`set_trace`	pour insérer dans le code un point d'arrêt

a. `https://docs.python.org/3.6/library/pdb.html`

INSA ROUEN NORMANDIE