第 1 回 デバッグゼミ

ゼミ担当者 ： 篠原 翔，宮地 正大
開催日 ： 2009 年 5 月 20 日

ゼミ内容: 本ゼミでは，プログラム開発に必要なデバッグの知識と Pydev を用いたデバッグ方法を学ぶ．また，プログラム実行時の様々な状態を得ることにより，その実行効率を調べるプロファイリングについて学習する．

1 はじめに

1.1 デバッグとは

デバッグとはコンピュータプログラムの誤りであるバグを探し，取り除くことである．プログラムは人間が作成するため，どうしても誤りが混入してしまう為，デバッグはプログラム開発過程において非常に重要な意味を持つ．バグを発見したり修正する作業を支援するソフトウェアのことをデバッガという．商用のソフトウェアなどでは，バグを発見する為に発売前の開発途上のバージョンを「β版」として公開してユーザから報告を募る方法も取られている．

1.2 よく現れるバグ

一般的によく現れるバグは，2 種類に分けられる．1 つは，プログラムが途中で止まってしまうバグ，もう 1 つは，実行はできるが結果に不具合があるバグである．前者には ZeroDivisionError, NameError, TypeError などのエラー，後者は変数の型の違いなど入力間違いが挙げられる．

2 デバッガ

デバッガを使うとプログラムの実行について観測，停止，再開，速度を落としての実行，プログラムが使っている変数等を表示させるなどの機能がある．一般的に高級言語では例外処理などの機能が使えるため．異常な振る舞いの原因となっている箇所を特定することが容易である．

近年では，Eclipse のような統合開発環境に含まれており，大体類似した機能を有している．

2.1 ブレークポイント

ソースコードの中に置く，実行の流れを止める機能である．ブレークポイントを置いてから該当のソフトウェアを実行した際，デバッガはそこで処理を止める．これにより，任意の位置での実行状況（変数の値やメモリの内容）を調べることができる．

2.2 ステップ実行

処理を止めた際に，1 ステップずつソースコードを実行する．これにより，ソースコードをステップごとに追いかけながら実行することが可能となり，ロジックの問題点を探すことができる．

2.3 ステップアウト

構造型言語，オブジェクト指向言語などで，関数，またはメソッドを 1 つ飛ばして実行する．これにより，ステップ実行した際に処理を簡略化できる．

2.4 変数確認

指定した変数の中身を出力する．これにより，変数にどんな値が入っており，それが正しいか，誤っているかを確認できる．

3 デバッグの手順

デバッグ作業は対象によって様々であるが，一般的に次のステップでデバッグを行う．

3.1 バグの存在を認識する

バグの存在は事前に予知できることも結果的に判明することもある．

経験を積んだプログラマは，プログラムの部分ごとの複雑性やデータ破壊の可能性などから，どこでエラーが起きやすいかを判断できる．ユーザが入力したデータや，通信によって得たデータなどエラーの兆候らしき箇所にチェックを挿むことで，データがいつ破壊された，または正しく処理されなかったかを検出することができる．

3.2 バグの発生源を分離する

このステップはシステムのどの部分が問題を引き起こしているかを特定するものである．

このステップは反復的なテストを必要とすることが多い．プログラマは最初に入力が正しいことを検証し，次に正しく読み込まれたか，正しく処理されたかなどを検証していく．モジュール化されたシステムではモジュール間のインタフェースを通してやり取りされるデータの妥当性を検査することで，このステップをわずかに楽にすることができる．

3.3 バグの原因を特定する

バグの位置を見つけたら，次のステップはバグの実際の理由を突き止めることである．例えば，関数中に数学的なエラーが出た場合，1 行ごとに逐次実行しながら変数を参照することでエラーの原因を知ることが出来る．

3.4 バグの修正方法を決定する

問題の源を特定したら，次はどのようにその問題を修正するかを決定する作業である．

問題が非常に単純なものである場合を除いて，システムの深い理解が必要不可欠となる．なぜなら，修正によってシステムの現状の振る舞いを変更してしまい，予期しない結果を生むことになるかもしれないからである．その上，既存のバグの修正は新しいバグを生み出したり，修正したバグに隠れていたバグを顕在化させることがよくある．このような問題はコード中でそれまでテストされていなかった部分を実行する際によく発生する．

3.5 修正して，テストを行う

修正が適用された後にシステムをテストしてその修正が以前の問題に正しく対処しているか確認するのは重要である．テストを行うべき理由は 2 つある．

1. その修正が問題に正しく対処しているか．
2. その修正が望ましくない副作用を引き起こしていないか．

3.1〜3.5 節のステップを繰り返すことでシステムを仕様通りに動作することを検証する．

4 Python のデバッグ

Python では一般的に 3 つのデバッグの方法がある．

4.1 print 文でのデバッグ

最も簡易なデバッグ手法である．本来のソースコードでは必要のなさそうな部分に print 文を挿入することで変数の内容を出力する．逐次変数の値をチェックすることでバグの箇所を発見する．

4.2 pdb でのデバッグ

Python には標準で pdb というデバッガが付属している．pdb はソースラインレベルでののでのブレークポイント設定（条件付き）とシングルステップ実行，スタックフレームの点検，ソースコードの列挙等の機能をサポートしている．

pdb を使ってデバッグするにはコマンドラインから次のように実行する．

```
% pdb デバッグ対象の Python プログラム
```

次のようなコマンドを用いることで，デバッグを行う．

- 'b(reak) 行数｜関数名'
  ブレークポイントを設定する．行数も関数名も書かなければ，全ブレークポイントを一覧表示する．'tbreak' で一時的なブレークポイントを設定し，'cl(ear) ブレークポイント番号' でブレークポイントを削除する．ブレークポイント番号を書かなければ全ブレークポイントを削除するので注意して使用する必要がある．

- 's(tep)'
  1 行実行して停止する． 'n(ext)' も同じく 1 行実行して停止するが，実行した文が関数を呼んだ場合，前者は呼ばれた関数内で停止するという違いがある．現在の関数がリターンするまで実行する 'r(eturn)' や，ブレークポイントに出会うまで実行する 'c(ont(inue))' もある．

pdb のようなデバッガを使うと，print 文でのデバッグと比べて，デバッグ時のバグ発見率が格段に上がる．

4.3 Pydev でのデバッグ

Pydev にはソースコードのデバッグ機能である Pydev debugger を持つ．視覚的に操作が可能で，ブレークポイントを動的に追加・削除を行うことができる．変数の値やインスタンスのプロパティや属性も知ることができる．詳細と使い方については次章で説明する．

5 Pydev のデバッグ

Eclipse は，強力なデバッグ機能を持っている．しかし，もともと Eclipse は Java の開発環境であり，Pydev は Eclipse のプラグインにより Python の開発を可能にしたものであるため，Java の開発で利用出来る機能のいくつかは，Pydev では利用する事が出来ない．具体的には，デバッグ中の値の変更をする事は出来ない．

5.1 Pydev のデバッグモードの起動

Pydev のデバッグモードで起動する方法について説明する．

1. 新しい用途の選択を行う (Fig. 1)

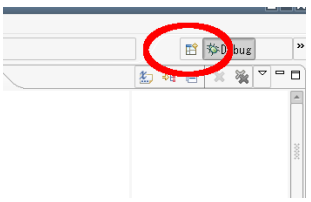

Fig. 1 Pydev デバッグモード (1)

2. 「Other...」を選択 (Fig. 2)
3. 「Debug」を選択し,「OK」を選択する (Fig. 3)

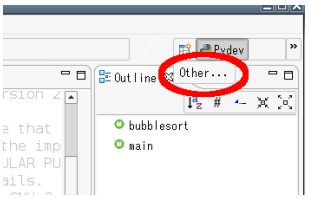

Fig. 2　Pydev デバッグモード (2)

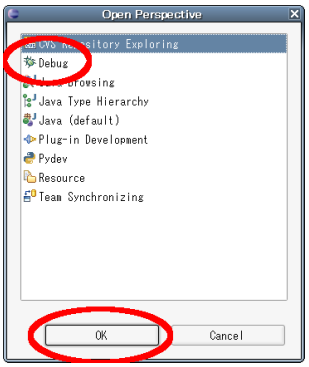

Fig. 3　Pydev デバッグモード (3)

Fig. 4 のようにデバッグモードが表示される．

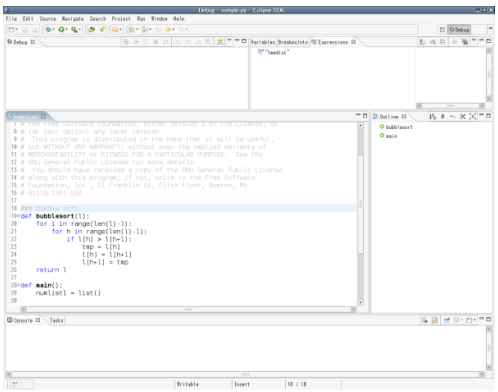

Fig. 4　Pydev デバッグモードの起動画面

5.2　デバッグモード画面

Pydev のデバッグモードは，Fig. 4 のように 5 つの画面から構成されている．

各画面についての説明は，以下の通りです．

画面 1 ステップ実行やプログラムを終了する際のボタンが揃っている．

画面 2 値の参照を行う事が出来る．Pydev では，値の変更をする事は出来ない．

画面 3 ソースコードを表示しています．左側の部分でブレークポイントの設定を行う事が出来る．

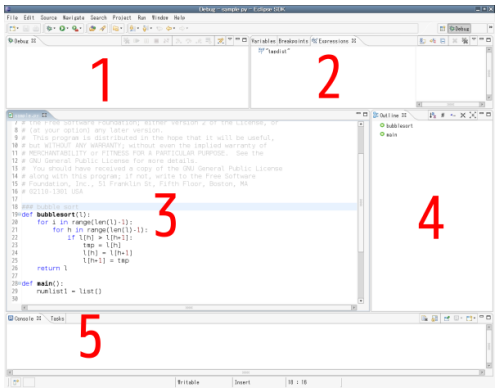

Fig. 5　Pydev デバッグモードの説明

画面 4 プログラムにどのような関数や変数があるか，表示している．

画面 5 プログラムの実行結果を表示する．

ここでは，1 のステップ実行についてのボタンの説明と，3 のブレークポイントについての説明を行う．

5.3　ステップ実行

ステップ実行とは，ソースコードの中で，ステップと呼ばれる，プログラムの塊を設定し，そのステップに従って，実行を行いながら，値の参照をする事が出来る．ステップ実行には，Fig. 5 の画面 1 に Fig. 6 で示したアイコンが並んでいる．それぞれの機能について説明する．

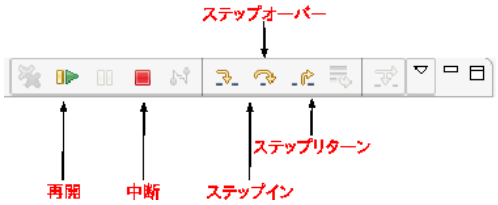

Fig. 6　ステップ実行のボタン

- 再開
  プログラムをブレークポイントで中断した際に，プログラムを再開する際に使う．
- 中断
  プログラムの実行を中断する．
- ステップイン
  次の 1 行を実行します．次の 1 行に関数呼び出しが含まれている場合は，その関数の内部に入った状態でプログラムが中断する．
- ステップオーバー
  次の 1 行を実行します．次の 1 行に関数が含まれて

いる場合でも，その関数が呼び出しが終えた状態でプログラムが中断する．

- ステップリターン
  今，実行中の関数を終えて，呼び出し元に戻す．

5.4 ブレークポイントの方法

ブレークポイントとは，プログラムの実行を中断する場所を示すものである．例えば，次のような場合にブレークポイントを設定する．

「関数に値を入れる前までは，値が正しいのに，関数に入れた際には，値が正しくない値になる」この問題の場合は，この関数の実行前にブレークポイントを設定し，関数を 1 行ずつ，値の変更が不正でないかを確認していく．

ブレークポイントの設定は，Fig. 7 に示したようにソースコードが表示されている左の部分を右クリックし，「Add Breakpoint」をクリックするか，ソースコードが表示されている左の部分でダブルクリックしても，ブレークポイントを設定することが出来る．

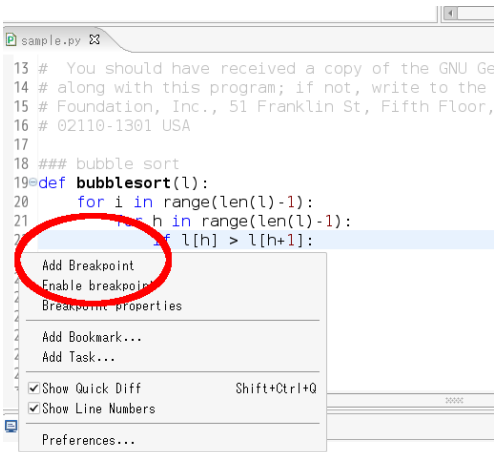

Fig. 7　ブレークポイントの設定

プログラムのデバッグを実行すると，ブレークポイントで設定した部分までプログラムを実行する事が出来る．その際に，変数の値を参照しながら，プログラムを追うことで，バグを発見する事が出来る．

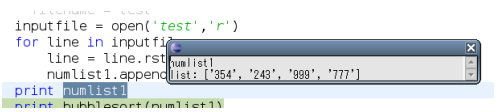

Fig. 8　値の表示

また，ブレークポイントからステップインなどでプログラムを実行している場合，Fig. 5 の画面 2 で値の確認をする事も出来るが，値をドラッグし，右クリックから，「Display」を選択する事で，Fig. 8 のように，値の表示を行う事も出来る．

6　プロファイリング

プロファイラとは，プログラム実行時の様々な状態を得ることにより，その実行効率を調べるためのプログラムである．以下，サンプルプログラムを用いて profile と pstats モジュールが提供するプロファイラ機能について解説する．

サンプルプログラム

```
def fib(n):
    if n <= 0:
        n = 0
    elif n == 1:
        n = 1
    else:
        n = fib(n - 1) + fib(n - 2)
    return n

if __name__=="__main__":
    fib(25)
```

6.1 profile モジュール

main エントリにある関数 fib() をプロファイルしたいとき，モジュールに以下のプログラムを追加することで Fig. 9 の結果を得ることができる．

```
import profile
profile.run('fib(25)')
```

```
         242788 function calls (4 primitive calls) in 3.889 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.012    0.012    0.012    0.012 :0(setprofile)
        1    0.000    0.000    3.877    3.877 <string>:1(<module>)
        1    0.000    0.000    3.889    3.889 profile:0(fib(25))
        0    0.000             0.000          profile:0(profiler)
242785/1    3.877    0.000    3.877    3.877 sample.py:3(fib)
```

Fig. 9　サンプルプログラム結果

Fig. 9 の最初の行は 242788 回の関数呼出しがあったことを示している．またこのうち 4 回はプリミティブなものであることを示している．プリミティブ な呼び出しとは，再帰によるものではない関数呼出しを指す．次の行 Ordered by: standard name は，一番右側の欄の文字列を使ってソートされたことを意味する．各カラムの見出しの意味は次の通りである．

ncalls 呼出し回数

tottime この関数が消費した時間の合計 (サブ関数呼出しの時間は除く)

percall tottime を ncalls で割った値

cumtime サブ関数を含む関数の (実行開始から終了までの) 消費時間の合計

percall cumtime をプリミティブな呼び出し回数で割った値

filenam:lineno(function) その関数のファイル名，行番号，関数名

最初の欄に 2 つの数字が表示されている場合（例：242785/1），最初の値は呼出し回数，2 番目はプリミティブな呼び出しの回数を表している．

```
profile.run('fib(25)', 'fib_prof')
```

また，上記の様に記述することで，'fib_prof' というファイル名で結果をファイルに残すことが可能である．

6.2 pstats モジュール

プロファイル内容を確認するときは，pstats モジュールのメソッドを使用します．統計データの読み込みは以下の様に記述することで可能である．

```
import pstats
p = pstats.Stats('fib_prof')
```

Stats クラス (上記コードはこのクラスのインスタンスを生成するだけの内容) は p に読み込まれたデータを操作したり，表示するための各種メソッドを備えている．以下，Stats クラスについて解説する．

6.3 Stats クラス

profile.run() を実行したとき表示された内容と同じものは，以下のコードにより実現できます．

```
p.strip_dirs().sort_stats(-1).print_stats()
```

最初のメソッドはモジュール名からファイル名の前に付いているパス部分を取り除きます．2 番目のメソッドはエントリをモジュール名/行番号/名前にもとづいてソートします．3 番目のメソッドで全ての統計情報を出力します．

```
p.sort_stats('name')
p.print_stats()
```

上記のようなソート・メソッドも使えます．最初の行ではリストを関数名でソートしています．2 行目で情報を出力しています．

```
p.sort_stats('cumulative').print_stats(10)
```

上記のようにすると，関数が消費した累計時間でソートされ，さらにその上位 10 件だけを表示します．どのアルゴリズムが時間を多く消費しているのか知りたいときは，この方法が役に立ちます．ループで多くの時間を消費している関数はどれか調べたいときは，以下のコードを用います．

```
p.sort_stats('time').print_stats(10)
```

これは関数の実行で消費した時間でソートされ，上位 10 個の関数の情報が表示されます．sort_stats メソッドは Stats オブジェクトを指定した基準に従ってソートします．引数には通常ソートのキーにしたい項目を示す文字列を指定します．以下に，定義されているキー名を示します．

Table 1　ソートキー一覧

キー	内容
'calls'	呼び出し回数
'cumulative'	合計時間
'file'	ファイル名
'module'	モジュール名
'pcalls'	プリミティブな呼び出しの回数
'line'	行番号
'name'	関数名
'nfl'	関数名/ファイル名/行番号
'stdname'	標準名
'time'	内部時間

参考文献

1) PyJUG：
http://www.python.jp/Zope/

2) デバッグの方法論：
http://apollon.cc.u-tokyo.ac.jp/w̃atanabe/tips/debug.html

3) Python プロファイラ：
http://www.python.jp/doc/2.4/lib/profile.html

4) www.hanecci.com：
http://www.hanecci.com/pukiwiki/index.php
?Programming%2FPython%2FProfile