Tools and Utilities

The standard library comes with a number of modules which can be used both as modules and as command line utilities.

The dis module

This is the Python disassembler. It converts bytecodes to a format that is slightly more appropriate for human consumption.

You can run the disassembler from the command line. It compiles the given script, and prints the disassembled byte codes to the terminal.

```
$ dis.py hello.py

  0 SET_LINENO               0

  3 SET_LINENO               1
  6 LOAD_CONST               0 ('hello again, and welcome to the show')
  9 PRINT_ITEM
 10 PRINT_NEWLINE
 11 LOAD_CONST               1 (None)
 14 RETURN_VALUE
```

You can also use **dis** as a module. The **dis** function takes a class, method, function, or code object as its single argument.

Example: Using the dis module

```
# File:dis-example-1.py

import dis

def procedure():
    print 'hello'

dis.dis(procedure)
```

```
  0 SET_LINENO               3

  3 SET_LINENO               4
  6 LOAD_CONST               1 ('hello')
  9 PRINT_ITEM
 10 PRINT_NEWLINE
 11 LOAD_CONST               0 (None)
 14 RETURN_VALUE
```

The pdb module

This is the standard Python debugger. It is based on the **bdb** debugger framework.

You can run the debugger from the command line. Type **n** (or **next**) to go to the next line, **help** to get a list of available commands.

```
$ pdb.py hello.py
> hello.py(0)?()
(Pdb) n
> hello.py()
(Pdb) n
hello again, and welcome to the show
--Return--
> hello.py(1)?()->None
(Pdb)
```

You can also start the debugger from inside a program.

Example: Using the pdb module

```
# File:pdb-example-1.py

import pdb

def test(n):
    j = 0
    for i in range(n):
        j = j + i
    return n

db = pdb.Pdb()
db.runcall(test, 1)
```

```
> pdb-example-1.py(3)test()
-> def test(n):
(Pdb) s
> pdb-example-1.py(4)test()
-> j = 0
(Pdb) s
> pdb-example-1.py(5)test()
-> for i in range(n):
...
```

The bdb module

This module provides a framework for debuggers. You can use this to create your own custom debuggers.

To implement custom behavior, subclass the **Bdb** class, and override the **user** methods (which are called whenever the debugger stops). To control the debugger, use the various **set** methods.

Example: Using the bdb module

```
# File:bdb-example-1.py

import bdb
import time

def spam(n):
    j = 0
    for i in range(n):
        j = j + i
    return n

def egg(n):
    spam(n)
    spam(n)
    spam(n)
    spam(n)

def test(n):
    egg(n)

class myDebugger(bdb.Bdb):

    run = 0

    def user_call(self, frame, args):
        name = frame.f_code.co_name or "<unknown>"
        print "call", name, args
        self.set_continue() # continue

    def user_line(self, frame):
        if self.run:
            self.run = 0
            self.set_trace() # start tracing
        else:
            # arrived at breakpoint
            name = frame.f_code.co_name or "<unknown>"
            filename = self.canonic(frame.f_code.co_filename)
            print "break at", filename, frame.f_lineno, "in", name
        print "continue..."
        self.set_continue() # continue to next breakpoint
```

```
    def user_return(self, frame, value):
        name = frame.f_code.co_name or "<unknown>"
        print "return from", name, value
        print "continue..."
        self.set_continue() # continue

    def user_exception(self, frame, exception):
        name = frame.f_code.co_name or "<unknown>"
        print "exception in", name, exception
        print "continue..."
        self.set_continue() # continue

db = myDebugger()
db.run = 1
db.set_break("bdb-example-1.py", 7)
db.runcall(test, 1)
```

```
continue...
call egg None
call spam None
break at C:\ematter\librarybook\bdb-example-1.py 7 in spam
continue...
call spam None
break at C:\ematter\librarybook\bdb-example-1.py 7 in spam
continue...
call spam None
break at C:\ematter\librarybook\bdb-example-1.py 7 in spam
continue...
call spam None
break at C:\ematter\librarybook\bdb-example-1.py 7 in spam
continue...
```

The profile module

This is the standard Python profiler.

Like the disassembler and the debugger, you can run the profiler from the command line.

```
$ profile.py hello.py

hello again, and welcome to the show

     3 function calls in 0.785 CPU seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.001    0.001    0.002    0.002 <string>:1(?)
     1    0.001    0.001    0.001    0.001 hello.py:1(?)
     1    0.783    0.783    0.785    0.785 profile:0(execfile('hello.py'))
     0    0.000             0.000          profile:0(profiler)
```

It can also be used to profile part of a program.

Example: Using the profile module

```
# File:profile-example-1.py

import profile

def func1():
   for i in range(1000):
      pass

def func2():
   for i in range(1000):
      func1()

profile.run("func2()")
```

```
     1003 function calls in 2.380 CPU seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)
     1   0.000   0.000   2.040   2.040 <string>:1(?)
  1000   1.950   0.002   1.950   0.002 profile-example-1.py:3(func1)
     1   0.090   0.090   2.040   2.040 profile-example-1.py:7(func2)
     1   0.340   0.340   2.380   2.380 profile:0(func2())
     0   0.000           0.000         profile:0(profiler)
```

You can modify the report to suit your needs, via the **pstats** module (see next page).

The pstats module

This tool analyses data collected by the Python profiler.

The following example sorts the output in a non-standard order (internal time, name):

Example: Using the pstats module

```
# File:pstats-example-1.py

import pstats
import profile

def func1():
    for i in range(1000):
        pass

def func2():
    for i in range(1000):
        func1()

p = profile.Profile()
p.run("func2()")

s = pstats.Stats(p)
s.sort_stats("time", "name").print_stats()
```

```
         1003 function calls in 1.574 CPU seconds

   Ordered by: internal time, function name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    1.522    0.002    1.522    0.002 pstats-example-1.py:4(func1)
        1    0.051    0.051    1.573    1.573 pstats-example-1.py:8(func2)
        1    0.001    0.001    1.574    1.574 profile:0(func2())
        1    0.000    0.000    1.573    1.573 <string>:1(?)
        0    0.000             0.000          profile:0(profiler)
```

The tabnanny module

(New in 2.0) This module checks Python source files for ambigous indentation. If a file mixes tabs and spaces in a way that the indentation isn't clear, no matter what tab size you're using, the nanny complains.

In the **badtabs.py** file used in the following examples, the first line after the **if** statement uses four spaces followed by a tab. The second uses spaces only.

```
$ tabnanny.py -v samples/badtabs.py
'samples/badtabs.py': *** Line 3: trouble in tab city! ***
offending line: '\tprint "world"\012'
indent not equal e.g. at tab sizes 1, 2, 3, 5, 6, 7, 9
```

Since the Python interpreter interprets a tab as eight spaces, the script will run correctly. It will also display correctly, in any editor that assumes that a tab is either eight or four spaces. That's not enough to fool the tab nanny, of course...

You can also use **tabnanny** from within a program.

Example: Using the tabnanny module

```
# File:tabnanny-example-1.py

import tabnanny

FILE = "samples/badtabs.py"

file = open(FILE)
for line in file.readlines():
    print repr(line)

# let tabnanny look at it
tabnanny.check(FILE)
```

```
'if 1:\012'
'    \011print "hello"\012'
'\tprint "world"\012'
samples/badtabs.py 3 '\tprint "world"\012'
```

(To capture the output, you can redirect **sys.stdout** to a **StringIO** object.)