Implementation Support Modules

The dospath module

This module provides **os.path** functionality on DOS platforms. You can also use it if you need to handle DOS paths on other platforms.

Example: Using the dospath module

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

import dospath

file = "/my/little/pony"

print "isabs", "=>", dospath.isabs(file)
print "dirname", "=>", dospath.dirname(file)
print "basename", "=>", dospath.basename(file)
print "normpath", "=>", dospath.normpath(file)
print "split", "=>", dospath.split(file)
print "join", "=>", dospath.join(file, "zorba")
```

```
isabs => 1
dirname => /my/little
basename => pony
normpath => \my\little\pony
split => ('/my/little', 'pony')
join => /my/little/pony\zorba
```

Note that Python's DOS support can use both forward (/) and backwards slashes (\) as directory separators.

The macpath module

This module provides **os.path** functionality on Macintosh platforms. You can also use it if you need to handle Mac paths on other platforms.

Example: Using the macpath module

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

import macpath

file = "my:little:pony"

print "isabs", "=>", macpath.isabs(file)
print "dirname", "=>", macpath.dirname(file)
print "basename", "=>", macpath.basename(file)
print "normpath", "=>", macpath.normpath(file)
print "split", "=>", macpath.split(file)
print "join", "=>", macpath.join(file, "zorba")
```

```
isabs => 1
dirname => my:little
basename => pony
normpath => my:little:pony
split => ('my:little', 'pony')
join => my:little:pony:zorba
```

The ntpath module

This module provides **os.path** functionality on Windows platforms. You can also use it if you need to handle Windows paths on other platforms.

Example: Using the ntpath module

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

import ntpath

file = "/my/little/pony"

print "isabs", "=>", ntpath.isabs(file)
print "dirname", "=>", ntpath.dirname(file)
print "basename", "=>", ntpath.basename(file)
print "normpath", "=>", ntpath.normpath(file)
print "split", "=>", ntpath.split(file)
print "join", "=>", ntpath.join(file, "zorba")
```

```
isabs => 1
dirname => /my/little
basename => pony
normpath => \my\little\pony
split => ('/my/little', 'pony')
join => /my/little/pony\zorba
```

Note that this module treats both forward slashes (/) and backward slashes (\) as directory separators.

The posixpath module

This module provides **os.path** functionality on Unix and other POSIX compatible platforms. You can also use it if you need to handle POSIX paths on other platforms. It can also be used to process uniform resource locators (URLs).

Example: Using the posixpath module

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

import posixpath

file = "/my/little/pony"

print "isabs", "=>", posixpath.isabs(file)
print "dirname", "=>", posixpath.dirname(file)
print "basename", "=>", posixpath.basename(file)
print "normpath", "=>", posixpath.normpath(file)
print "split", "=>", posixpath.split(file)
print "join", "=>", posixpath.join(file, "zorba")
```

```
isabs => 1
dirname => /my/little
basename => pony
normpath => /my/little/pony
split => ('/my/little', 'pony')
join => /my/little/pony/zorba
```

The strop module

(Obsolete) This is a low-level module that provides fast C implementations of most functions in the **string** module. It is automatically included by **string**, so there's seldom any need to access it directly.

However, one reason to use this module is if you need to tweak the path *before* you start loading Python modules.

Example: Using the strop module

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

import strop
import sys

# assuming we have an executable named ".../executable", add a
# directory named ".../executable-extra" to the path

if strop.lower(sys.executable)[-4:] == ".exe":
    extra = sys.executable[:-4] # windows
else:
    extra = sys.executable

sys.path.insert(0, extra + "-extra")

import mymodule
```

In Python 2.0 and later, you should use string methods instead of **strop**. In the above example, replace "**strop.lower(sys.executable)**" with "**sys.executable.lower()**"

The imp module

This module contains functions that can be used to implement your own **import** behavior. The following example overloads the import statement with a version that logs from where it gets the modules.

Example: Using the imp module

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

import imp
import sys

def my_import(name, globals=None, locals=None, fromlist=None):
    try:
        module = sys.modules[name] # already imported?
    except KeyError:
        file, pathname, description = imp.find_module(name)
        print "import", name, "from", pathname, description
        module = imp.load_module(name, file, pathname, description)
    return module

import __builtin__
__builtin__.__import__ = my_import

import xmllib
```

```
import xmllib from /python/lib/xmllib.py ('.py', 'r', 1)
import re from /python/lib/re.py ('.py', 'r', 1)
import sre from /python/lib/sre.py ('.py', 'r', 1)
import sre_compile from /python/lib/sre_compile.py ('.py', 'r', 1)
import _sre from /python/_sre.pyd ('.pyd', 'rb', 3)
```

Note that the alternative version shown here doesn't support packages. For a more extensive example, see the sources for the **knee** module.

The new module

(Optional in 1.5.2). This is a low-level module which allows you to create various kinds of internal objects, such as class objects, function objects, and other stuff that is usually created by the Python runtime system.

Note that if you're using 1.5.2, you may have to rebuild Python to use this module; it isn't enabled by the default on all platforms. In 2.0 and later, it's always available.

Example: Using the new module

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

import new

class Sample:

    a = "default"

    def __init__(self):
        self.a = "initialised"

    def __repr__(self):
        return self.a

#
# create instances

a = Sample()
print "normal", "=>", a

b = new.instance(Sample, {})
print "new.instance", "=>", b

b.__init__()
print "after __init__", "=>", b

c = new.instance(Sample, {"a": "assigned"})
print "new.instance w. dictionary", "=>", c
```

```
normal => initialised
new.instance => default
after __init__ => initialised
new.instance w. dictionary => assigned
```

The pre module

(Implementation). This module is a low-level implementation module for the 1.5.2 **re** module. There's usually no need to use this module directly (and code using it may stop working in future releases).

Example: Using the pre module

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

import pre

p = pre.compile("[Python]+")

print p.findall("Python is not that bad")
```

```
['Python', 'not', 'th', 't']
```

The sre module

(Implementation). This module is a low-level implementation module for the 2.0 **re** module. There's usually no need to use this module directly (and code using it may stop working in future releases).

Example: Using the sre module

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

import sre

text = "The Bookshop Sketch"

# a single character
m = sre.match(".", text)
if m: print repr("."), "=>", repr(m.group(0))

# and so on, for all 're' examples...
```

```
'.' => 'T'
```

The py_compile module

This module allows you to explicitly compile Python modules to bytecode. It behaves like Python's **import** statement, but takes a file name, not a module name.

Example: Using the py_compile module

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

import py_compile

# explicitly compile this module
py_compile.compile("py-compile-example-1.py")
```

The **compileall** module can be used to compile all Python files in an entire directory tree.

The compileall module

This module contains functions to compile all Python scripts in a given directory (or along the Python path) to bytecode. It can also be used as a script (on Unix platforms, it's automatically run when Python is installed).

Example: Using the compileall module to compile all modules in a directory

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

import compileall

print "This may take a while!"

compileall.compile_dir(".", force=1)
```

```
This may take a while!
Listing . ...
Compiling .\SimpleAsyncHTTP.py ...
Compiling .\aifc-example-1.py ...
Compiling .\anydbm-example-1.py ...
...
```

The ihooks module

This module provides a framework for import replacements. The idea is to allow several alternate import mechanisms to co-exist.

Example: Using the ihooks module

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

import ihooks, imp, os

def import_from(filename):
    "Import module from a named file"

    loader = ihooks.BasicModuleLoader()
    path, file = os.path.split(filename)
    name, ext  = os.path.splitext(file)
    m = loader.find_module_in_dir(name, path)
    if not m:
        raise ImportError, name
    m = loader.load_module(name, m)
    return m

colorsys = import_from("/python/lib/colorsys.py")

print colorsys
```

```
<module 'colorsys' from '/python/lib/colorsys.py'>
```

The linecache module

This module is used to read lines from module source code. It caches recently visited modules (the entire source file, actually).

Example: Using the linecache module

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

import linecache

print linecache.getline("linecache-example-1.py", 5)

print linecache.getline("linecache-example-1.py", 5)
```

This module is used by the **traceback** module.

The macurl2path module

(Implementation). This module contains code to map uniform resource locators (URLs) to Macintosh filenames, and back. It should not be used directly; use the mechanisms in **urllib** instead.

Example: Using the macurl2path module

```
# File:macurl2path-example-1.py

import macurl2path

file = ":my:little:pony"

print macurl2path.pathname2url(file)
print macurl2path.url2pathname(macurl2path.pathname2url(file))
```

```
my/little/pony
:my:little:pony
```

The nturl2path module

(Implementation). This module contains code to map uniform resource locators (URLs) to Windows filenames, and back.

Example: Using the nturl2path module

```
# File:nturl2path-example-1.py

import nturl2path

file = r"c:\my\little\pony"

print nturl2path.pathname2url(file)
print nturl2path.url2pathname(nturl2path.pathname2url(file))
```

```
///C|/my/little/pony
C:\my\little\pony
```

This module should not be used directly; for portability, access these functions via the **urllib** module instead:

Example: Using the nturl2path module via the urllib module

```
# File:nturl2path-example-2.py

import urllib

file = r"c:\my\little\pony"

print urllib.pathname2url(file)
print urllib.url2pathname(urllib.pathname2url(file))
```

```
///C|/my/little/pony
C:\my\little\pony
```

The tokenize module

This module splits a Python source file into individual tokens. It can be used for syntax highlighting, or for various kinds of code analysis tools.

In the following example, we simply print the tokens:

Example: Using the tokenize module

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

import tokenize

file = open("tokenize-example-1.py")

def handle_token(type, token, (srow, scol), (erow, ecol), line):
    print "%d,%d-%d,%d:\t%s\t%s" % \
        (srow, scol, erow, ecol, tokenize.tok_name[type], repr(token))

tokenize.tokenize(
    file.readline,
    handle_token
    )
```

```
1,0-1,6:     NAME    'import'
1,7-1,15:    NAME    'tokenize'
1,15-1,16:   NEWLINE '\012'
2,0-2,1:     NL      '\012'
3,0-3,4:     NAME    'file'
3,5-3,6:     OP      '='
3,7-3,11:    NAME    'open'
3,11-3,12:   OP      '('
3,12-3,35:   STRING  '"tokenize-example-1.py"'
3,35-3,36:   OP      ')'
3,36-3,37:   NEWLINE '\012'
...
```

Note that the **tokenize** function takes two callable objects; the first argument is called repeatedly to fetch new code lines, and the second argument is called for each token.

The keyword module

This module contains a list of the keywords used in the current version of Python. It also provides a dictionary with the keywords as keys, and a predicate function that can be used to check if a given word is a Python keyword.

Example: Using the keyword module

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

import keyword

name = raw_input("Enter module name: ")

if keyword.iskeyword(name):
    print name, "is a reserved word."
    print "here's a complete list of reserved words:"
    print keyword.kwlist
```

```
Enter module name: assert
assert is a reserved word.
here's a complete list of reserved words:
['and', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from',
'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or',
'pass', 'print', 'raise', 'return', 'try', 'while']
```

The parser module

(Optional). The parser module provides an interface to Python's built-in parser and compiler.

The following example compiles a simple expression into an *abstract syntax tree* (AST), turns the AST into a nested list, dumps the contents of the tree (where each node contains either a grammar symbol or a token), increments all numbers by one, and finally turns the list back into a code object. At least that's what I think it does.

Example: Using the parser module

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

import parser
import symbol, token

def dump_and_modify(node):
    name = symbol.sym_name.get(node[0])
    if name is None:
        name = token.tok_name.get(node[0])
    print name,
    for i in range(1, len(node)):
        item = node[i]
        if type(item) is type([]):
            dump_and_modify(item)
        else:
            print repr(item)
            if name == "NUMBER":
                # increment all numbers!
                node[i] = repr(int(item)+1)

ast = parser.expr("1 + 3")

list = ast.tolist()

dump_and_modify(list)

ast = parser.sequence2ast(list)

print eval(parser.compileast(ast))
```

```
eval_input testlist test and_test not_test comparison
expr xor_expr and_expr shift_expr arith_expr term factor
power atom NUMBER '1'
PLUS '+'
term factor power atom NUMBER '3'
NEWLINE ''
ENDMARKER ''
6
```

The symbol module

This module contains a listing of non-terminal symbols from the Python grammar. It's probably only useful if you're dabbling with the **parser** module.

Example: Using the symbol module

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

import symbol

print "print", symbol.print_stmt
print "return", symbol.return_stmt
```

```
print 268
return 274
```

The token module

This module contains a list of all tokens used by the standard Python tokenizer.

Example: Using the token module

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

import token

print "NUMBER", token.NUMBER
print "PLUS", token.STAR
print "STRING", token.STRING
```

```
NUMBER 2
PLUS 16
STRING 3
```