PYTHON LAB BOOK

Python For Programmers

UCSC Extension Online

Lab 7 Important Trick

Topics

- Module: `sys`
- Important trick:
- `__name__` and `'__main__'`
- Valid identifiers

v.4.0

lab06_2.py

```
#!/usr/bin/env python
""" Lab06.2 (Deitel 5.3)
Use a list to solve the following problem: Read in 5 numbers.  As each
number is read, print it only if it is not a duplicate of a number
already read.

Note the use of the "input" BIF (built-in function):
input([prompt])  <==> eval(raw_input([prompt]))
"""
numbers = []
for i in range(5):
    newnum = input("Number please: ")
    if not newnum in numbers:
        print newnum
        numbers += [newnum]

"""
$ lab06_2.py
Number please: 4
4
Number please: 1
1
Number please: 4
Number please: 3
3
Number please: 2
2
$
"""
```

lab06_3.py

```
#!/usr/bin/env python
"""lab06_3.py Sorts name strings by last name."""

def LastFirst(name):
    parts = name.split()
    return parts[-1] + ', ' + ' '.join(parts[:-1])

names = ["Jack Sparrow", "George Washington", "Tiny Sparrow",
         "Jean Ann Kennedy"]

for name in sorted(names, key=LastFirst):
    print LastFirst(name)

"""
$ lab06_3.py
Kennedy, Jean Ann
Sparrow, Jack
Sparrow, Tiny
Washington, George
$
"""
```

mutability.py

```
#!/usr/bin/env python
"""Mutability of sequence objects."""

x = 3
y = x
x *= 30

print y    # y = 3   not mutable

x = (1, 2)
y = x
x = 5

print y   # y = (1, 2) not mutable

x = [1, 2]
y = x
x[0] = 100
print y   # y == x still, mutable
          # [100, 2]

x = (1, [10, 22], 3)
x[1][0] = 100
print x  # x[1] is a list, mutable
         # So, (1, [100, 22], 3)

"""
$ mutability.py
3
(1, 2)
[100, 2]
(1, [100, 22], 3)
$
"""
```

sys_demo.py

```
#!/usr/bin/env python
"""Demonstrating the sys module."""

import sys

def DemoOpenStreams():
    """Demos stderr, stdout and stdin.  Also sys.exit()"""
    sys.stderr.write('You can write to stderr.\n')
    print >> sys.stdout, "You might like the >> syntax."
    sys.stdout.write('A fancier way to write to stdout.\n')
    print 'Type something: '
    text = sys.stdin.readline()
    print 'You said:', text

def DemoCommandLine():
    """Shows the command line."""
    print 'This program is named:', sys.argv[0]
    print 'The command line arguments are:', sys.argv[1:]

DemoCommandLine()
DemoOpenStreams()

"""
./sys_demo.py -a -b 123
This program is named: ./sys_demo.py
The command line arguments are: ['-a', '-b', '123']
You can write to stderr.
You might like the >> syntax.
A fancier way to write to stdout.
Type something:
jalapenos
You said: jalapenos

"""
```

Important Trick

`dir()` gives a list of all the names in the current scope. Here is the result of `dir()` when the interpreter first comes up:

```
Python 2.5 (r25:51908, Mar 28 2007, 09:49:25)
[GCC 4.1.0 20060304 (Red Hat 4.1.0-3)] on linux2
pType "help", "copyright", "credits" or "license" for more information.
>>> dir()
['__builtins__', '__doc__', '__name__']
```

Now let's add something:

```
>>> x = 3
>>> dir()
['__builtins__', '__doc__', '__name__', 'x']
>>>
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', 'math', 'x']
```

But, what's in that math module? `dir(math)` to find out:

```
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', (much deleted), 'tan', 'tanh']
>>>
```

More exploration:

```
>>> print __name__
__main__
```

```
>>> print math.__name__
math
```

We want to study the behavior of the `__name__` attribute in two different circumstances:

1. when it is `"__main__"`, and
2. when it is not.

We'll use a small module of code for this study (next page):

trick.py
```
#!/usr/bin/env python
"""Simple code to demonstrate __name__"""

print "trick.py's __name__ is", __name__

"""
$ trick.py
trick.py's __name__ is __main__
$ """
```

The output is no surprise, given the experiment we did at the interpreter's prompt.

But, look at the value of `__name__` when trick.py is imported:

```
>>> import trick
trick.py's __name__ is trick
>>>
```

And, we can get to it this way:

```
>>> trick.__name__
'trick'
>>>
```

A module's `__name__` matches the file name and the name on the `import` line, unless the module is the main module being run, that is, unless it is the module being run to start the program, in which case the name is `"__main__"`.

We use this fact for an important testing/developing trick. Python programmers who know the trick write code that only tests when the module's name is `'__main__'`.

Here's some code written that way:

tables.py

```
#!/usr/bin/env python
"""tables.py Unwraps and prints out a 2-D sequence.
Note that the testing only happens when this module
is the __main__ module.
"""

def PrintTable(table):
    """Prints out a 2-D sequence"""
    for row in table:
        for column in row:
            print column,
        print
    print

if __name__ == '__main__':
    tests = (["Hi", "Hola"],
             (('H','i'), ('H','o','l','a')),
             [["Hi"], ["Hola"]]
             )
    for test in tests:
        print test
        PrintTable(test)
"""
$ tables.py
['Hi', 'Hola']
H i
H o l a

(('H', 'i'), ('H', 'o', 'l', 'a'))
H i
H o l a

[['Hi'], ['Hola']]
Hi
Hola
$ """
```

Here's the trick:

```
>>> import tables
>>>
```

Nothing happened! In particular, the tests didn't happen. This is because the name of the imported tables.py module is `"tables"`, not `"__main__"`, so all the testing gets skipped.

```
>>> tables.__name__
'tables'
```

Demonstrating another piece of Pythonic magic:

```
>>> help(tables)
Help on module tables:

NAME
    tables

FILE
    /home/marilyn/python/mm/labs/lab_07_Important_Trick/tables.py

DESCRIPTION
    tables.py Unwraps and prints out a 2-D sequence.
    Note that the testing only happens when this module
    is the __main__ module.

FUNCTIONS
    PrintTable(table)
        Prints out a 2-D sequence
```

All that documentation was lifted from the code. From that, I know I can:

```
>>> tables.PrintTable((('X', 'O', 'O'),
...                    ('X', 'X', ' '),
...                    ('O', 'O', 'X')))

X O O
X X
O O X
```

Or, I can include the `tables.py` module in some other code.

tables2.py
```
#!/usr/bin/env python
"""tables2.py Interactive 2-D string unwrapper.
"""
import tables

def main():
    while True:
        response = raw_input("Say something: ")
        if not response:
            break
        words = response.split()
        tables.PrintTable(words)

if __name__ == '__main__':
    main()
"""
$ tables2.py
Say something: "Pythonic Thinking"
" P y t h o n i c
T h i n k i n g "

Say something:
$ """
```

Lab 07

1. Write a function that, when passed a string of alphanumeric charaters, returns a string of digits. Each character that is in the input string is converted to the digit that corresponds to it on a phone keypad:

   ```
   abc -> 2  ghi -> 4   mno -> 6   tuv -> 8
   def -> 3  jkl -> 5  pqrs -> 7  wxyz -> 9
   ```

   Your module name might be `lab07_1.py`.

   Test your function with the following data:

   DATA = "peanut", "salt", "lemonade", "good time", ":10", "Zilch"

   Be sure that your test does not run when your `lab07_1.py` is `import`ed.

2. Make another program module, perhaps `lab07_2.py` in the same directory as your `lab07_1.py`.

   `lab07_2.py` will ask the user for a word and then print the keypad translation. It will import `lab07_1.py` to do the translation.

3. Continue your PigLatin program from the last lab.