Python Modules

- A module allows you to logically organize your Python code.
  - Grouping related codes into a module makes the code easier to understand and use.
- A module is a Python object with arbitrarily named attributes that you can bind and reference.
  - Simply, a module is a file consisting of Python code.
- A module can define functions, classes and variables.
- A module can also include runnable code.

The *import* Statement

- You can use any Python source file as a module by executing an import statement in some other Python source file.
  The *import* has the following syntax:

```
import module1[, module2[,... moduleN]
```

- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.
- A search path is a list of directories that the interpreter searches before importing a module.

Example

- To import the module *hello.py*, you need to put the following command at the top of the script:

```
#!/usr/bin/python

# Import module support
import support

# Now you can call defined function that module as follows
support.print_func("Zara")
```

```
Hello : Zara
```

Example

- xmath.py

```python
def max(a, b):
    return a if a > b else b
def min(a, b):
    return a if a < b else b

def sum(*numbers): # numbers 接受可變長度引數
    total = 0
    for number in numbers:
        total += number
    return total

pi = 3.141592653589793
e = 2.718281828459045
```

```
# import xmath
3.14159265359
10
15
# import xmath as math
2.71828182846
# from xmath import min
5
```

```python
import xmath
print '# import xmath'
print xmath.pi
print xmath.max(10, 5)
print xmath.sum(1, 2, 3, 4, 5)

print '# import xmath as math'
import xmath as math # 為 xmath 模組取別名為 math
print math.e

print '# from xmath import min'
from xmath import min  # 將 min 複製至目前模組，不建議 from modu import *，易造
print min(10, 5)
```

The *from...import* Statement

- Python's *from* statement lets you import specific attributes from a module into the current namespace.
- The *from...import* has the following syntax:

```
from modname import name1[, name2[, ... nameN]]
```

- For example, to import the function fibonacci from the module fib, use the following statement:

```
from fib import fibonacci
```

The *from...import* * Statement:

- It is also possible to import all names from a module into the current namespace by using the following import statement:

```
from modname import *
```

Locating Modules:

- When you import a module, the Python interpreter searches for the module in the following sequences:
  - The current directory.
  - If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
  - If all else fails, Python checks the default path.
    - On UNIX, this default path is normally /usr/local/lib/python/.

The *PYTHONPATH* Variable:

- The PYTHONPATH is an environment variable, consisting of a list of directories.
- The syntax of PYTHONPATH is the same as that of the shell variable PATH.
- Here is a typical PYTHONPATH from a Windows system:
  - set PYTHONPATH=c:\python27\lib;
- Here is a typical PYTHONPATH from a UNIX system:
  - set PYTHONPATH=/usr/local/lib/python

Namespaces and Scoping

- Variables are names (identifiers) that map to objects.
- A ***namespace*** is a dictionary of variable names (keys) and their corresponding objects (values).
- A Python statement can access variables in a *local namespace* and in the *global namespace*.
  - If a local and a global variable have the same name, the local variable shadows the global variable.
- Each function has its own local namespace.
  - Class methods follow the same scoping rule as ordinary functions.
- Python assumes that any variables assigns a value in a function is local.

Namespaces and Scoping

- Therefore, in order to assign a value to a global variable within a function, you must first use the *global* statement.
- The statement *global VarName* tells Python that VarName is a global variable.
  - Python stops searching the local namespace for the variable.

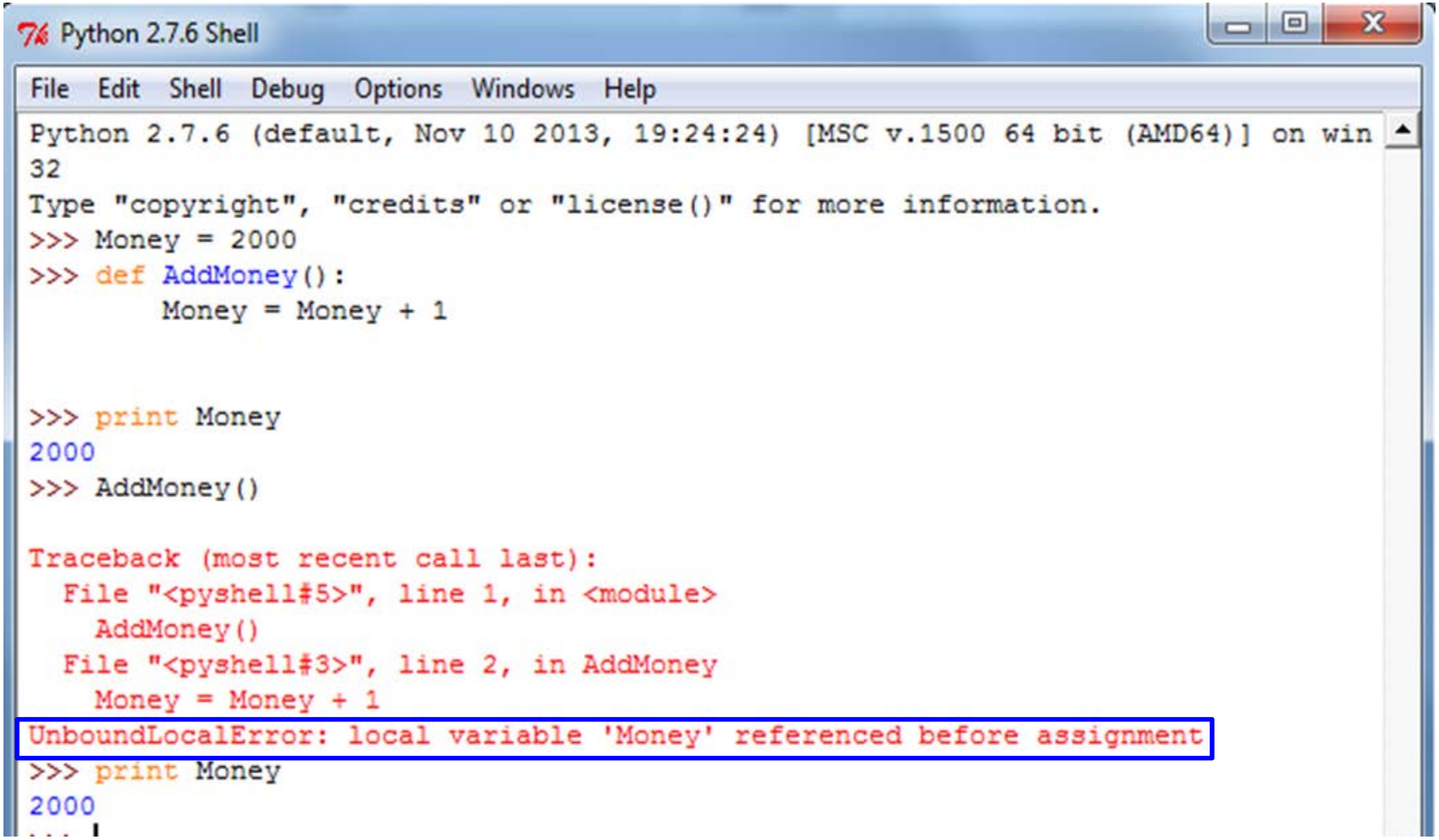
```
#!/usr/bin/python

Money = 2000
def AddMoney():
   # Uncomment the following line to fix the code:
   # global Money
   Money = Money + 1

print Money
AddMoney()
print Money
```

```
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    AddMoney()
  File "<pyshell#3>", line 2, in AddMoney
    Money = Money + 1
UnboundLocalError: local variable 'Money' referenced before assignment
```

Results


Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Nov 10 2013, 19:24:24) [MSC v.1500 64 bit (AMD64)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> Money = 2000
>>> def AddMoney():
Money = Money + 1
>>> print Money
2000
>>> AddMoney()
Traceback (most recent call last):
File "<pyshell#5>", line 1, in <module>
AddMoney()
File "<pyshell#3>", line 2, in AddMoney
Money = Money + 1
UnboundLocalError: local variable 'Money' referenced before assignment
>>> print Money
2000

Example (1)

```
# sample.py
myGlobal = 5

def func1():
    myGlobal = 42

def func2():
    print myGlobal

func1()
func2()
```

5

```
def func1():
    global myGlobal
    myGlobal = 42
```

42

Example (2)

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The nonlocal statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals.

The global statement is a declaration which holds for the entire current code block.

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

globals() locals(), and var()

- The *globals() locals()* and *var()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.
- If locals() is called from within a function, it will return all the names that can be accessed **locally** from that function.
- If globals() is called from within a function, it will return all the names that can be accessed **globally** from that function.
- var() returns *either a* dictionary of the current namespace (if called with no argument) or *the* dictionary of the argument.
- The return type of both these functions (locals and globals)is dictionary.
  - Therefore, names can be extracted using the keys() function.

The dir( ) Function

- The dir() built-in function returns a sorted list of strings containing the names defined by a module.
- The list contains the names of all the modules, variables and functions that are defined in a module.
- Here, the special string variable *__name__* is the **module's name**, and *__file__* is the **filename** from which the module was loaded.

```
#!/usr/bin/python

# Import built-in module math
import math

content = dir(math)

print content;
```

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

Packages in Python

- A package is a hierarchical file directory structure
  - It defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.
- Consider a file *Pots.py* available in *Phone* directory
- We have another two files having different functions with the same directory as above:
  - *Phone/Isdn.py* file having function Isdn()
  - *Phone/G3.py* file having function G3()
- Now, create one more file __init__.py in *Phone* directory:
  - Phone/__init__.py

Packages in Python

- To make all of your functions available when you've imported Phone, you need to put explicit import statements in __init__.py as follows:
  - from Pots import Pots
  - from Isdn import Isdn
  - from G3 import G3

```
#!/usr/bin/python

# Now import your Phone Package.
import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()
```

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

Example

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

```
import sound.effects.echo
```

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

```
from sound.effects import echo
```

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

```
from sound.effects.echo import echofilter
```

```
echofilter(input, output, delay=0.7, atten=4)
```

Python Image Library - Examples

```
import Image
global ext
ext = ".jpg"
imageFile = "test.jpg"
im1 = Image.open(imageFile)
Im1.show()
```

Original image

- **Python Imaging Library (PIL)**
- http://www.pythonware.com/products/pil/
- **PIL 1.1.7**
- http://effbot.org/downloads/PIL-1.1.7.win32-py2.7.exe

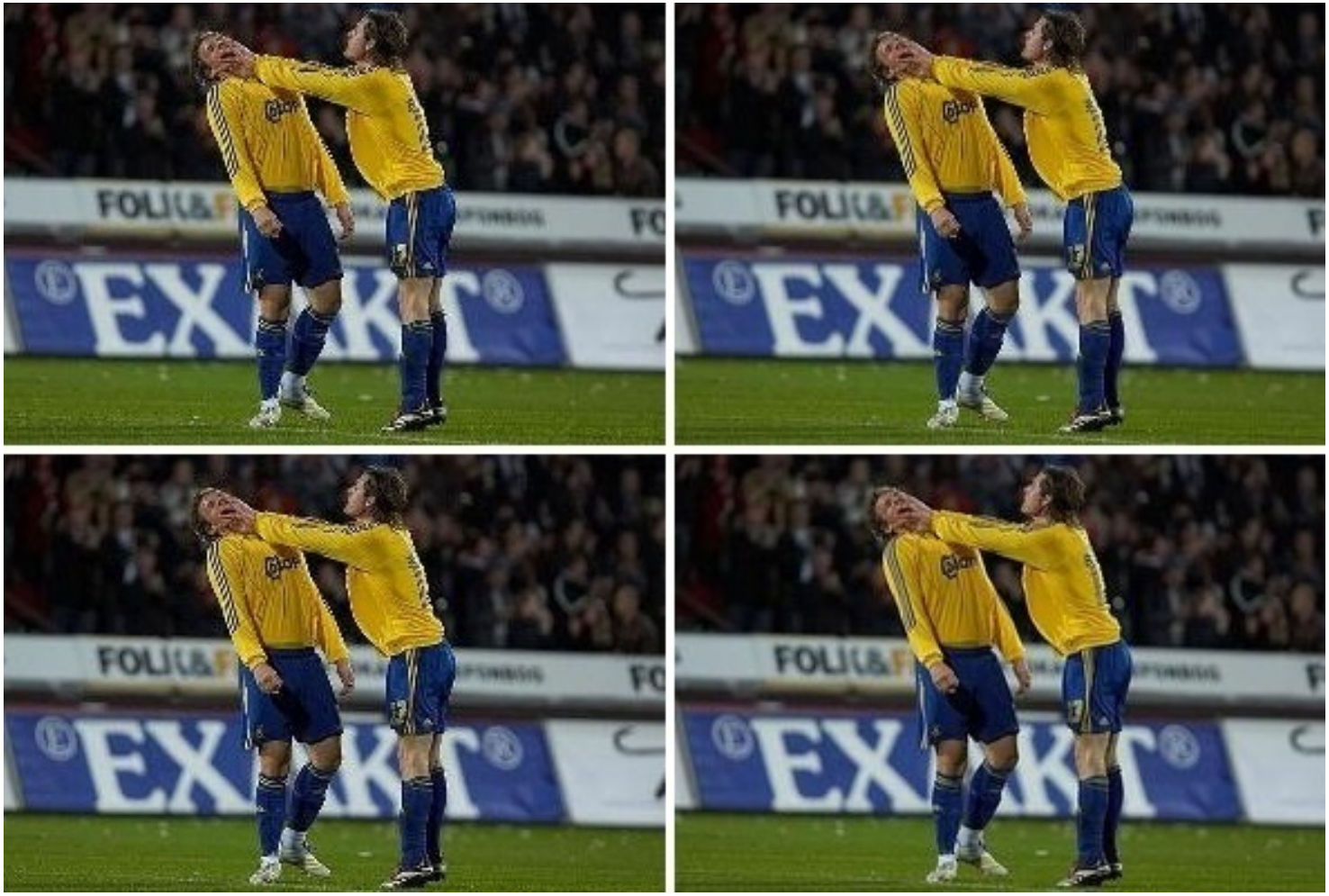
Resize

```
def imgResize(im):
    div = 2
    width = im.size[0] / div
    height = im.size[1] / div

    im2 = im.resize((width, height), Image.NEAREST) # use nearest neighbour
    im3 = im.resize((width, height), Image.BILINEAR) # linear interpolation in a 2x2 environment
    im4 = im.resize((width, height), Image.BICUBIC) # cubic spline interpolation in a 4x4 environment
    im5 = im.resize((width, height), Image.ANTIALIAS) # best down-sizing filter

    im2.save("NEAREST" + ext)
    im3.save("BILINEAR" + ext)
    im4.save("BICUBIC" + ext)
    im5.save("ANTIALIAS" + ext)
imgResize(im1)
```

Resize



Crop

- def imgCrop(im):
- box = (50, 50, 200, 300)
- region = im.crop(box)
- region.save("CROPPED" + ext)
- imgCrop(im1)

Transpose

- def imgTranspose(im):
- box = (50, 50, 200, 300)
- region = im.crop(box)
- region =region.transpose(Image.ROTATE_180)
- im.paste(region, box)
- im.save("TRANSPOSE“+ext)
- imgTranspose(im1)

Band Merge

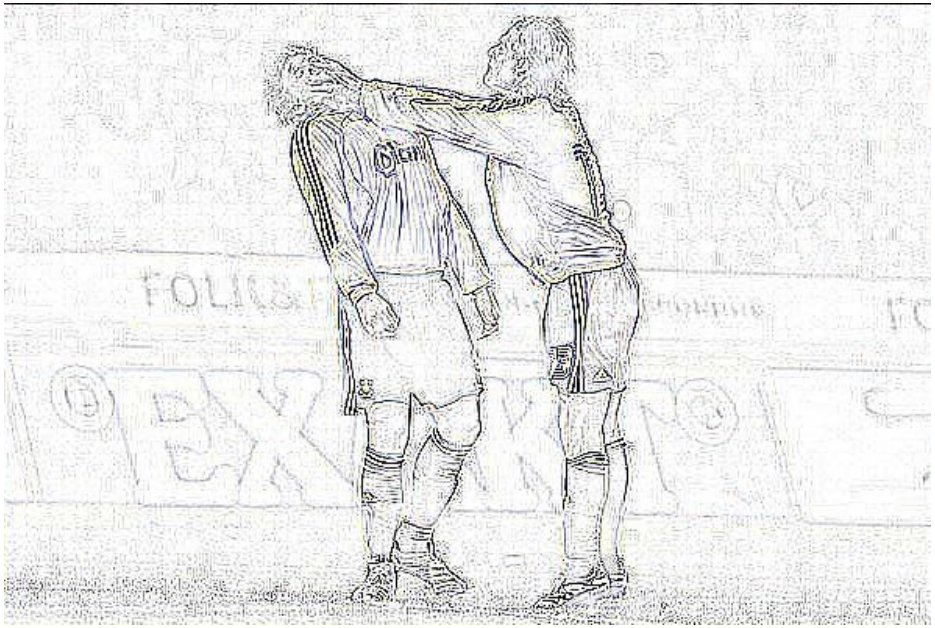
- def bandMerge(im):
- r, g, b = im.split()
- im = Image.merge("RGB", (g,g,g))
- im.save("MERGE" + ext)
- bandMerge(im1)

Blur

- import ImageFilter
- def filterBlur(im):
- im1 = im.filter(ImageFilter.BLUR)
- im1.save("BLUR" + ext)
- filterBlur(im1)

Find contours

- def filterContour(im):
- im1 = im.filter(ImageFilter.CONTOUR)
- im1.save("CONTOUR" + ext)
- filterContour(im1)



Find edges

- def filterFindEdges(im):
- im1 = im.filter(ImageFilter.FIND_EDGES)
- im1.save("EDGES" + ext)
- filterFindEdges(im1)