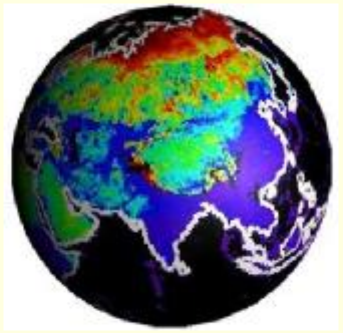


Chapter 14 Files

Jihn-Fa Jan, Ph.D.
Associate Professor
Department of Land Economics
National Chengchi University

Persistence

- Persistent programs: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.
- Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.
- One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapters we will see programs that write them.
- An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, pickle, that makes it easy to store program data.

Reading and writing

- A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. We saw how to open and read a file in Section 9.1.
- To write a file, you have to open it with mode 'w' as a second parameter:

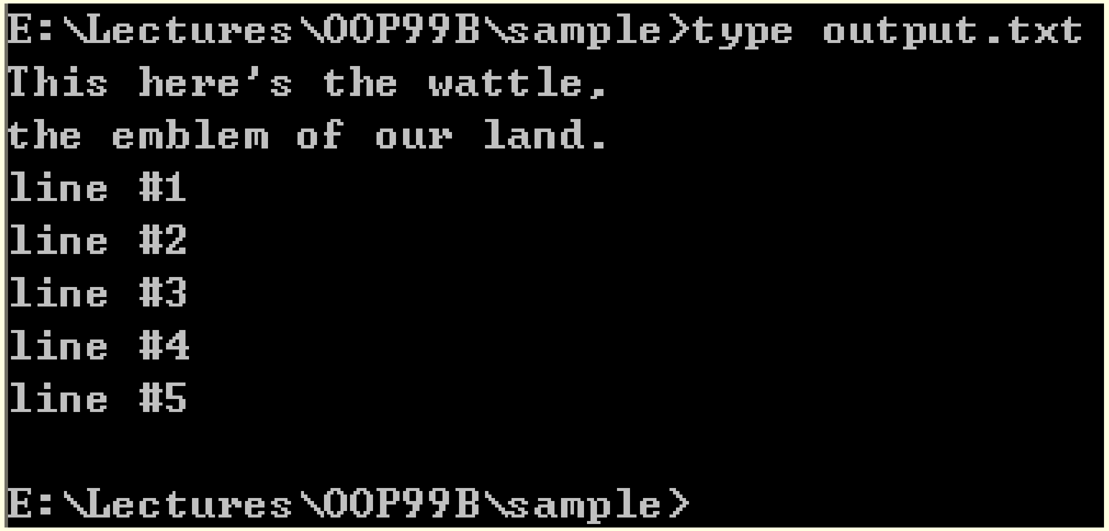
```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0x00D1CF98>
>>>
```

- If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn’t exist, a new one is created.

Reading and writing (cont'd)

- **The write method puts data into the file.**

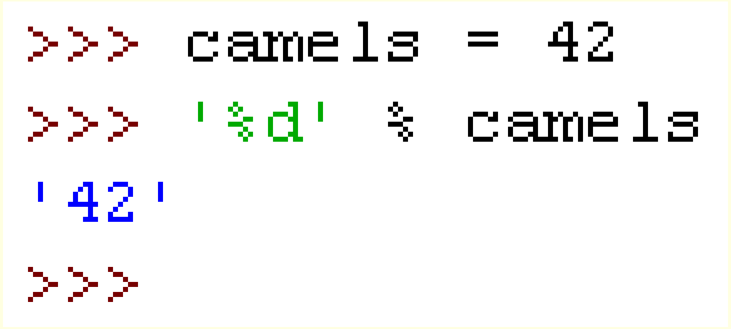
```
fout = open('output.txt', 'w')
print fout
line1 = "This here's the wattle,\n"
fout.write(line1)
line2 = "the emblem of our land.\n"
fout.write(line2)

for i in range(5):
    s = 'line #' + str(i+1) + '\n'
    fout.write(s)
fout.close()
```



```
E:\Lectures\OOP99B\sample>type output.txt
This here's the wattle,
the emblem of our land.
line #1
line #2
line #3
line #4
line #5

E:\Lectures\OOP99B\sample>
```

Format operator

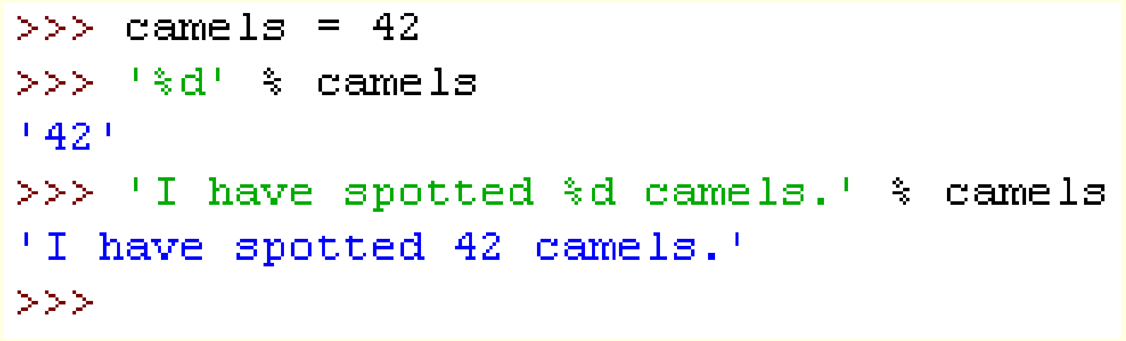
- The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with str.
- An alternative is to use the format operator, %. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.
- The first operand is the format string, which contains one or more format sequences, which specify how the second operand is formatted. The result is a string.
- For example, the format sequence '%d' means that the second operand should be formatted as an integer (d stands for “decimal”). The result is the string '42', which is not to be confused with the integer value 42.



```
>>> camels = 42
>>> '%d' % camels
'42'
>>>
```

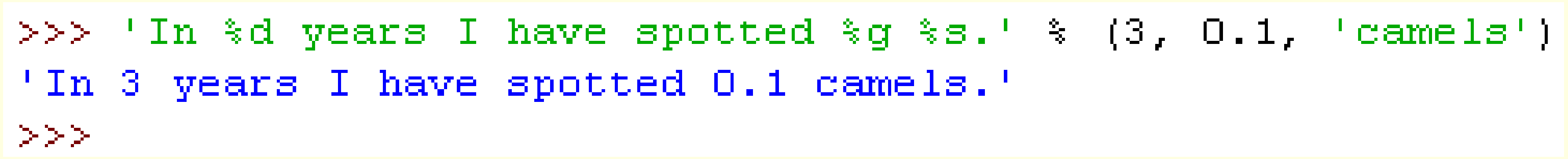
Format operator (cont'd)

- A format sequence can appear anywhere in the string, so you can embed a value in a sentence:



```
>>> camels = 42
>>> '%d' % camels
'42'
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
>>>
```

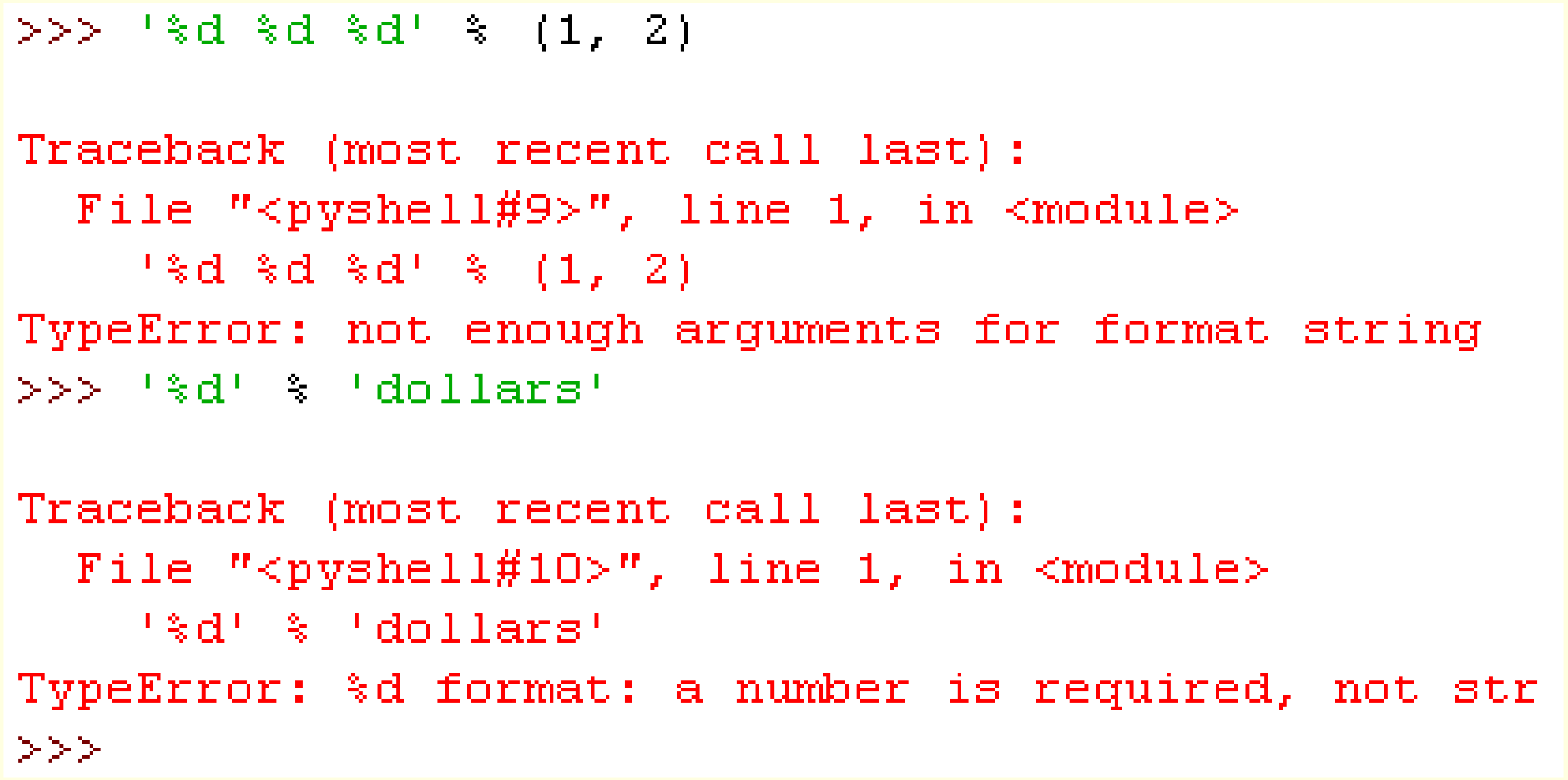
- If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order. The following example uses '%d' to format an integer, '%g' to format a floating-point number (don't ask why), and '%s' to format a string:



```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
>>>
```

Format operator (cont'd)

- The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

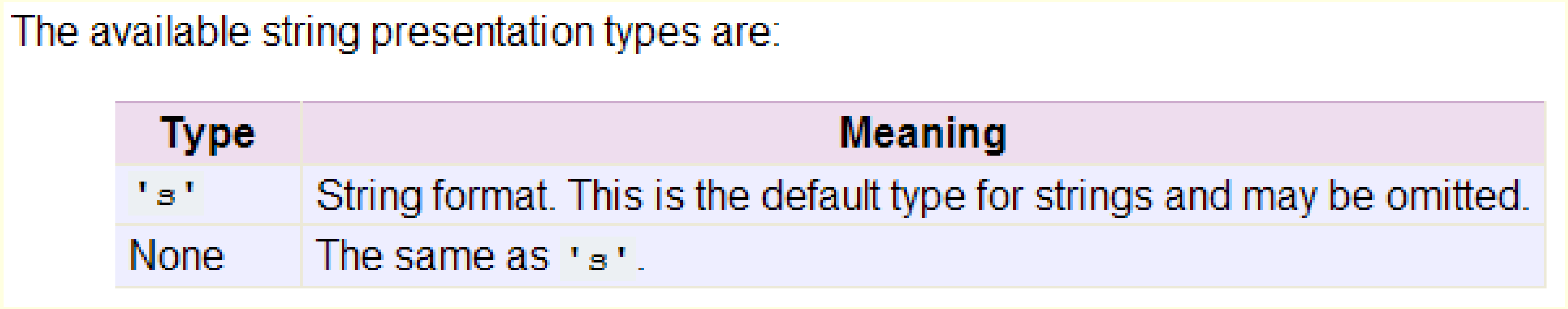


```
>>> '%d %d %d' % (1, 2)

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    '%d' % 'dollars'
TypeError: %d format: a number is required, not str
>>>
```

Format operator (cont'd)

- **The format operator is powerful, but it can be difficult to use. You can read more about it at docs.**

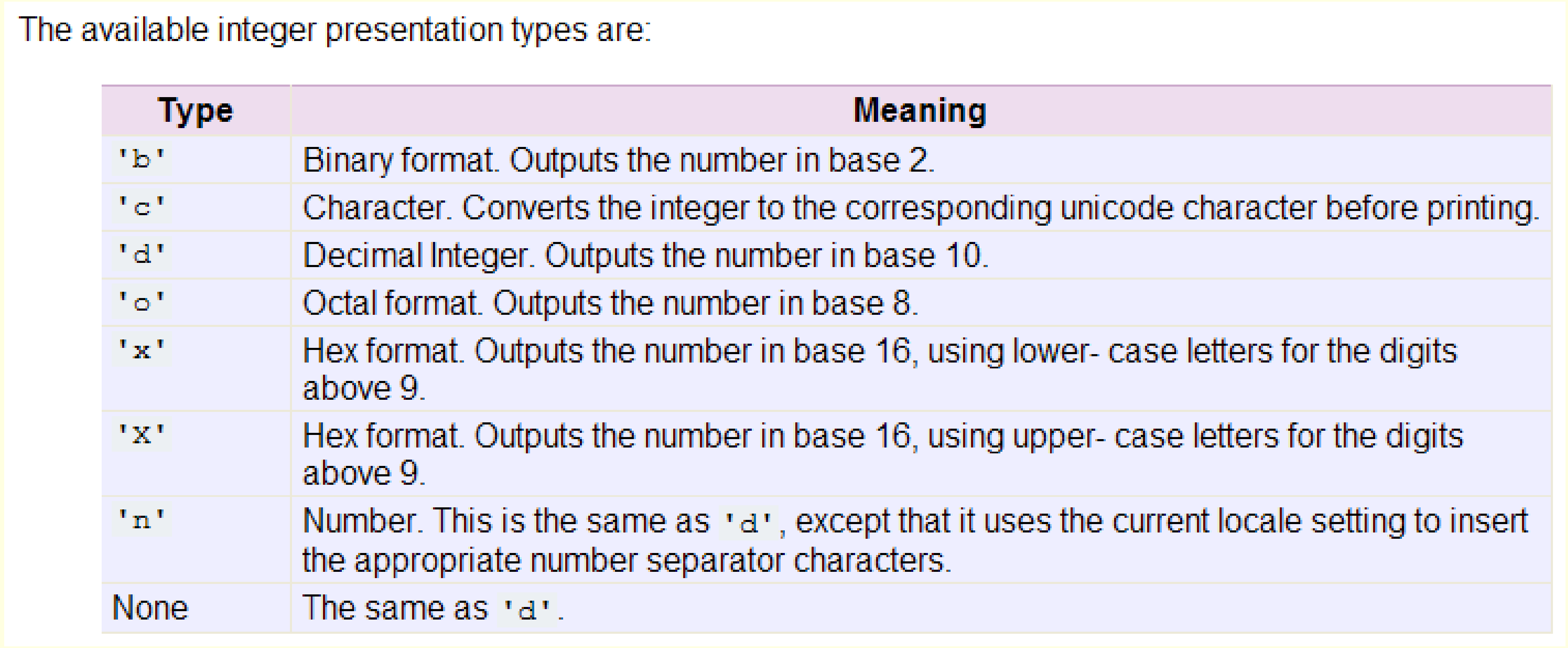
  http://www.python.org/doc//current/library/string.html#format-specification-mini-language



The available string presentation types are:

Type	Meaning
`'s'`	String format. This is the default type for strings and may be omitted.
None	The same as `'s'`.

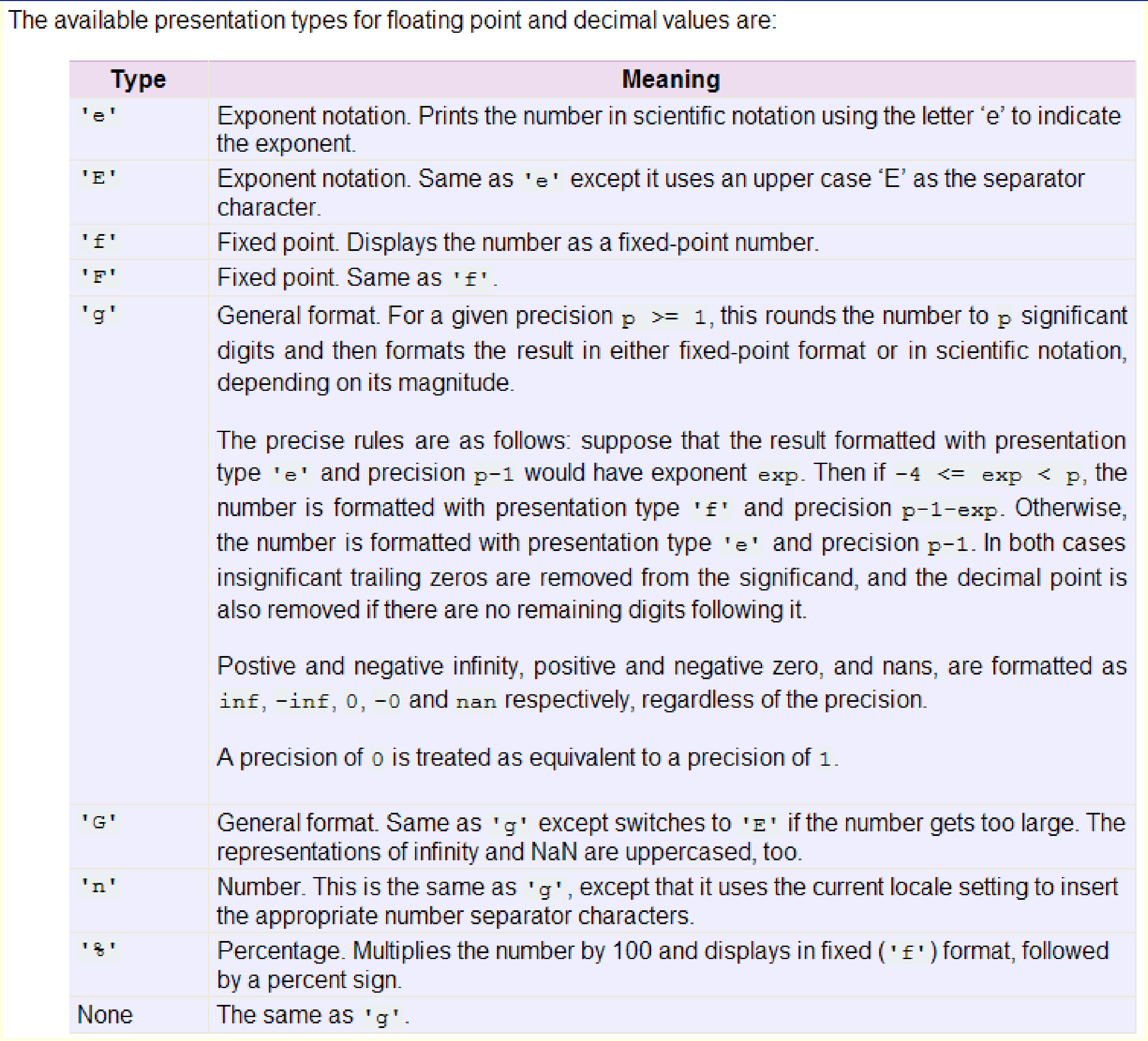
Format operator (cont'd)



The available integer presentation types are:

Type	Meaning
`'b'`	Binary format. Outputs the number in base 2.
`'c'`	Character. Converts the integer to the corresponding unicode character before printing.
`'d'`	Decimal Integer. Outputs the number in base 10.
`'o'`	Octal format. Outputs the number in base 8.
`'x'`	Hex format. Outputs the number in base 16, using lower- case letters for the digits above 9.
`'X'`	Hex format. Outputs the number in base 16, using upper- case letters for the digits above 9.
`'n'`	Number. This is the same as `'d'`, except that it uses the current locale setting to insert the appropriate number separator characters.
None	The same as `'d'`.

Format operator (cont'd)

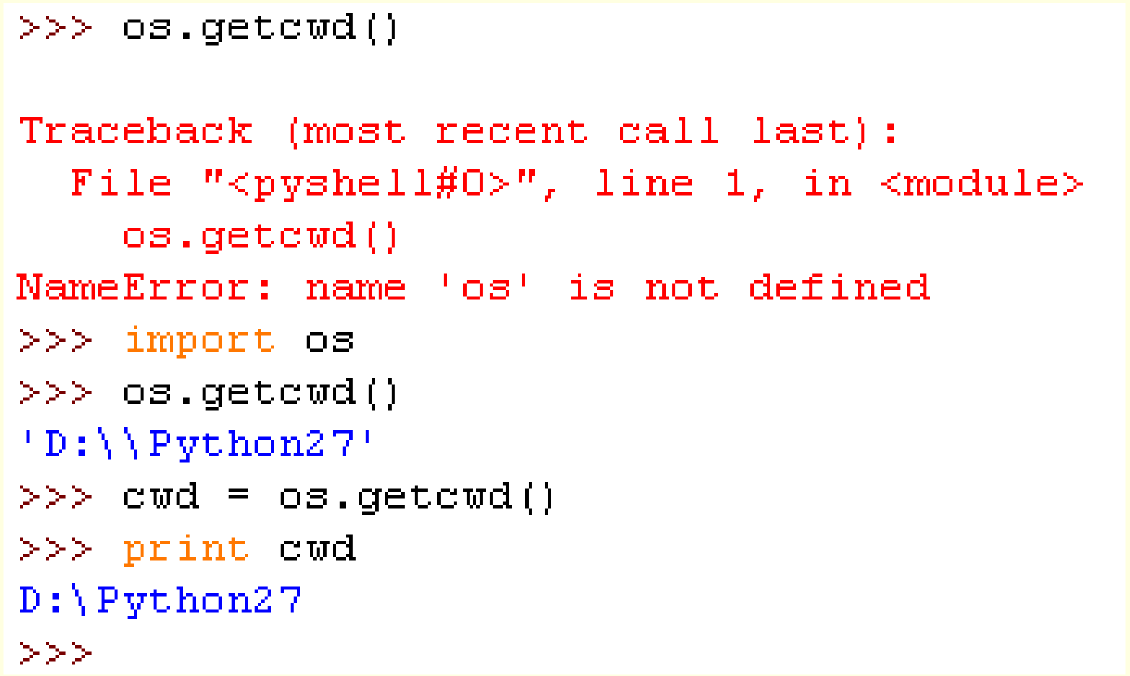


The available presentation types for floating point and decimal values are:

Type	Meaning
`'e'`	Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent.
`'E'`	Exponent notation. Same as `'e'` except it uses an upper case 'E' as the separator character.
`'f'`	Fixed point. Displays the number as a fixed-point number.
`'F'`	Fixed point. Same as `'f'`.
`'g'`	General format. For a given precision `p >= 1`, this rounds the number to `p` significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. The precise rules are as follows: suppose that the result formatted with presentation type `'e'` and precision `p-1` would have exponent `exp`. Then if `-4 <= exp < p`, the number is formatted with presentation type `'f'` and precision `p-1-exp`. Otherwise, the number is formatted with presentation type `'e'` and precision `p-1`. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it. Postive and negative infinity, positive and negative zero, and nans, are formatted as `inf`, `-inf`, `0`, `-0` and `nan` respectively, regardless of the precision. A precision of `0` is treated as equivalent to a precision of `1`.
`'G'`	General format. Same as `'g'` except switches to `'E'` if the number gets too large. The representations of infinity and NaN are uppercased, too.
`'n'`	Number. This is the same as `'g'`, except that it uses the current locale setting to insert the appropriate number separator characters.
`'%'`	Percentage. Multiplies the number by 100 and displays in fixed (`'f'`) format, followed by a percent sign.
None	The same as `'g'`.

Filenames and paths

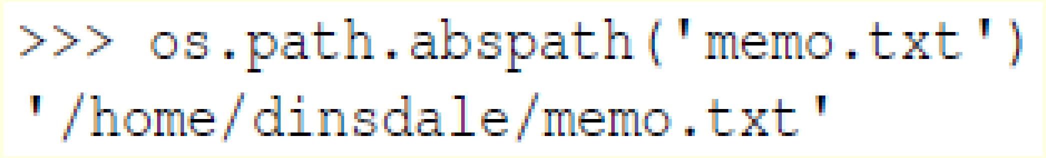
- Files are organized into directories (also called “folders”). Every running program has a “current directory,” which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.
- The os module provides functions for working with files and directories (“os” stands for “operating system”). os.getcwd returns the name of the current directory:



```
>>> os.getcwd()

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    os.getcwd()
NameError: name 'os' is not defined
>>> import os
>>> os.getcwd()
'D:\\Python27'
>>> cwd = os.getcwd()
>>> print cwd
D:\Python27
>>>
```

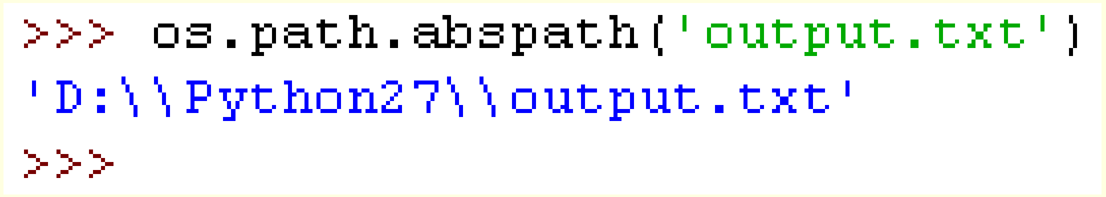
Filenames and paths (cont'd)

- **cwd stands for "current working directory."**
- **A string like cwd that identifies a file is called a path. A relative path starts from the current directory; an absolute path starts from the topmost directory in the file system.**
- **The paths we have seen so far are simple filenames, so they are relative to the current directory. To find the absolute path to a file, you can use os.path.abspath:**
  - On UNIX / Linux



```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

  - On Windows



```
>>> os.path.abspath('output.txt')
'D:\\Python27\\output.txt'
>>>
```

Filenames and paths (cont’d)

- **os.path.exists checks whether a file or directory exists:**

```
>>> os.path.exists('memo.txt')
True
```

- **If it exists, os.path.isdir checks whether it’s a directory:**

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

- **Similarly, os.path.isfile checks whether it’s a file.**
- **os.listdir returns a list of the files (and other directories) in the given directory:**

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

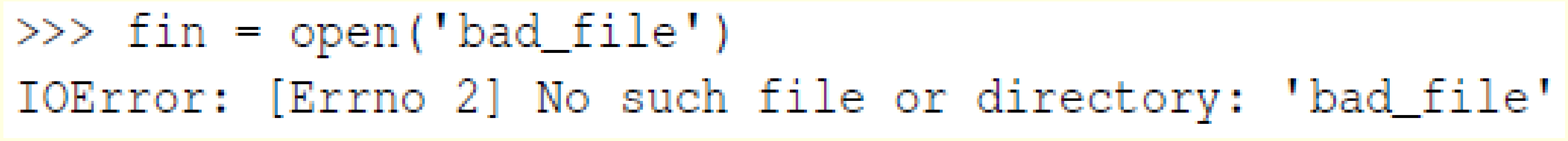
Filenames and paths (cont’d)

- To demonstrate these functions, the following example “walks” through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
import os

def walk(dir):
   for name in os.listdir(dir):
      path = os.path.join(dir, name)
      if os.path.isfile(path):
         print path
      else:
         walk(path)

d = raw_input('enter path name: ')

walk(d)
```

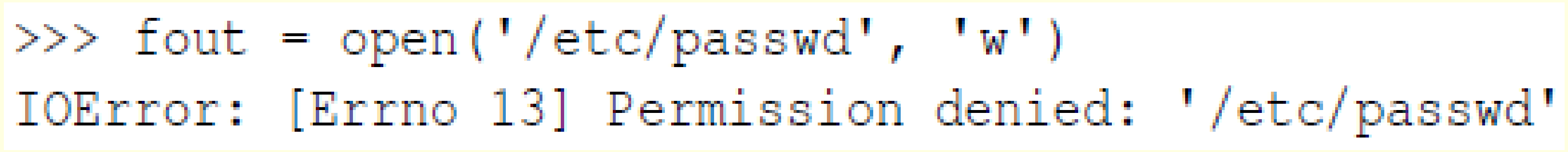
Catching exceptions

- A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn’t exist, you get an IOError:



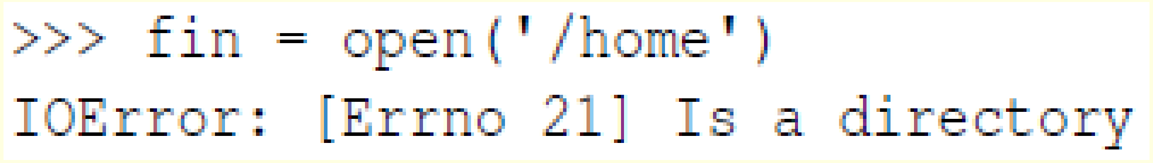
```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

- If you don’t have permission to access a file:



```
>>> fout = open('/etc/passwd', 'w')
IOError: [Errno 13] Permission denied: '/etc/passwd'
```

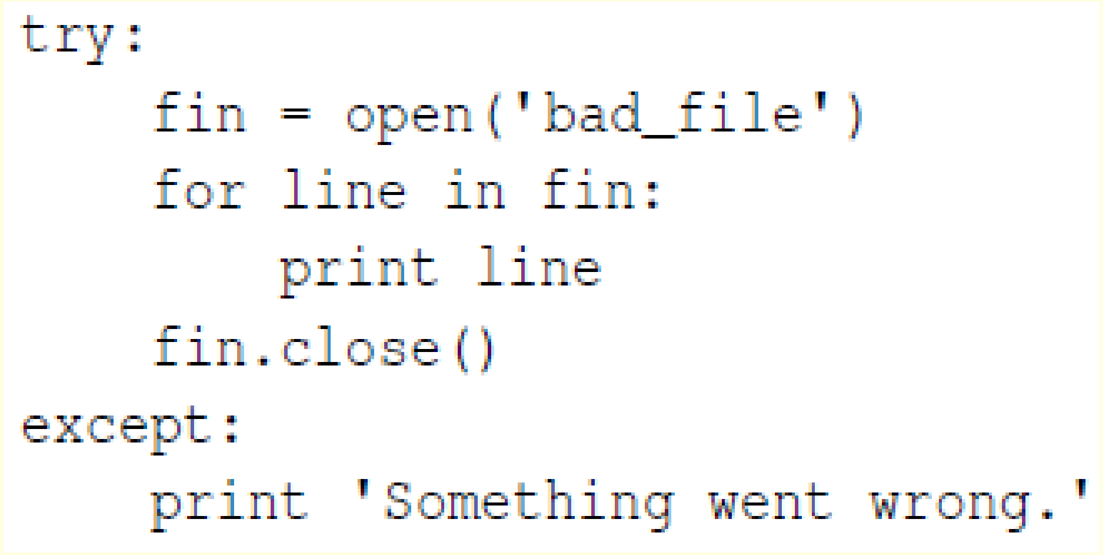
- And if you try to open a directory for reading, you get



```
>>> fin = open('/home')
IOError: [Errno 21] Is a directory
```

Catching exceptions (cont’d)

- To avoid these errors, you could use functions like os.path.exists and os.path.isfile, but it would take a lot of time and code to check all the possibilities (if “Errno 21” is any indication, there are at least 21 things that can go wrong).
- It is better to go ahead and try, and deal with problems if they happen, which is exactly what the try statement does. The syntax is similar to an if statement:



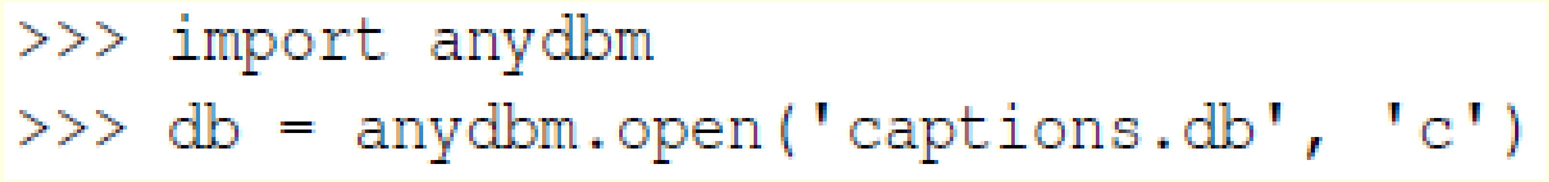
```
try:
    fin = open('bad_file')
    for line in fin:
        print line
    fin.close()
except:
    print 'Something went wrong.'
```

Catching exceptions (cont'd)

- Python starts by executing the try clause. If all goes well, it skips the except clause and proceeds. If an exception occurs, it jumps out of the try clause and executes the except clause.
- Handling an exception with a try statement is called catching an exception. In this example, the except clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

Databases

- **A database is a file that is organized for storing data. Most databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference is that the database is on disk (or other permanent storage), so it persists after the program ends.**
- **The module anydbm provides an interface for creating and updating database files.**
- **Opening a database is similar to opening other files:**



```
>>> import anydbm
>>> db = anydbm.open('captions.db', 'c')
```

  - The mode 'c' means that the database should be created if it doesn’t already exist. The result is a database object that can be used (for most operations) like a dictionary.

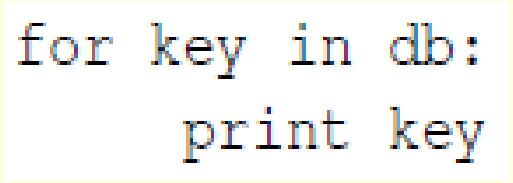
Databases (cont'd)

- If you create a new item, anydbm updates the database file.
- When you access one of the items, anydbm reads the file.
- If you make another assignment to an existing key, anydbm replaces the old value:

```
>>> import anydbm
>>> db = anydbm.open('captions.db', 'c')
>>> db['cleese.png'] = 'Photo of John Cleese.'
>>> print db['cleese.png']
Photo of John Cleese.
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> print db['cleese.png']
Photo of John Cleese doing a silly walk.
>>>
```

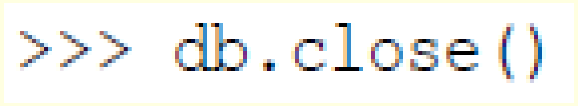
Databases (cont'd)

- Many dictionary methods, like keys and items, also work with database objects. So does iteration with a for statement.



```
for key in db:
    print key
```

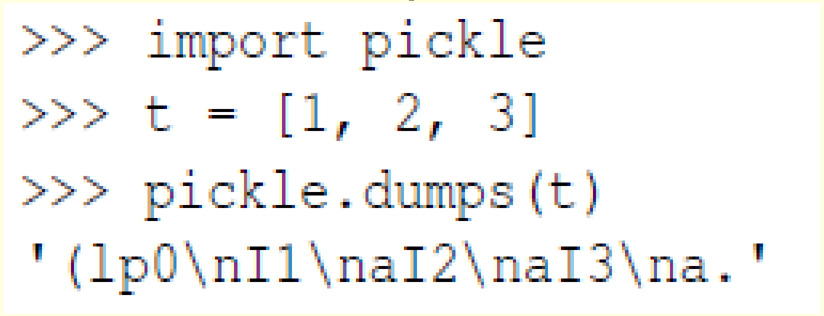
- As with other files, you should close the database when you are done:



```
>>> db.close()
```

Pickling

- A limitation of anydbm is that the keys and values have to be strings. If you try to use any other type, you get an error.
- The pickle module can help. It translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects.
- pickle.dumps takes an object as a parameter and returns a string representation (dumps is short for "dump string"):

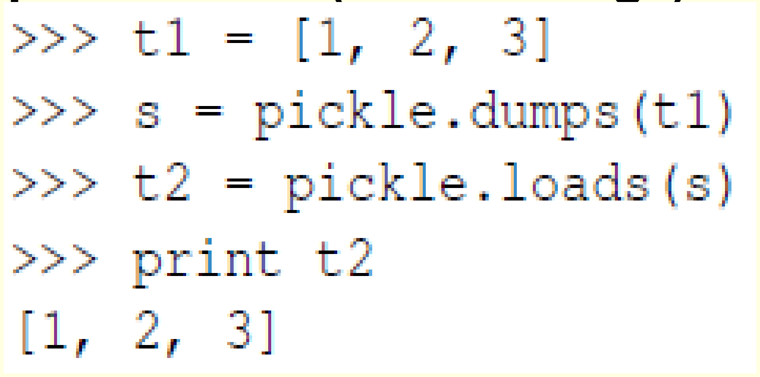


```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
'(lp0\nI1\naI2\naI3\na.'
```

- The format isn't obvious to human readers; it is meant to be easy for pickle to interpret.

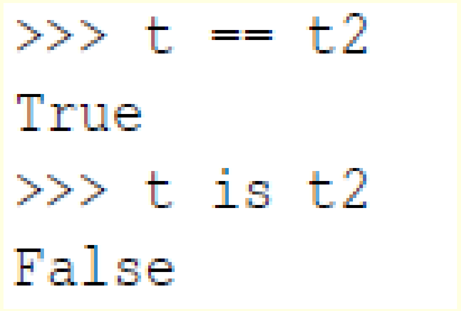
Pickling (cont'd)

- **pickle.loads ("load string") reconstitutes the object:**



```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> print t2
[1, 2, 3]
```

- **Although the new object has the same value as the old, it is not (in general) the same object. In other words, pickling and then unpickling has the same effect as copying the object.**

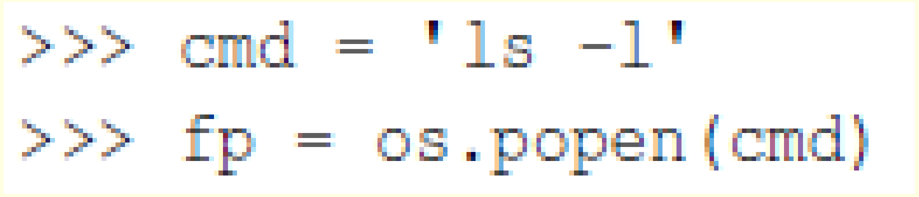


```
>>> t == t2
True
>>> t is t2
False
```

- **You can use pickle to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called shelve.**

Pipes

- Most operating systems provide a command-line interface, also known as a shell. Shells usually provide commands to navigate the file system and launch applications. For example, in Unix, you can change directories with cd, display the contents of a directory with ls, and launch a web browser by typing (for example) firefox.
- Any program that you can launch from the shell can also be launched from Python using a pipe. A pipe is an object that represents a running process.
- For example, the Unix command ls -l normally displays the contents of the current directory (in long format). You can launch ls with os.popen:

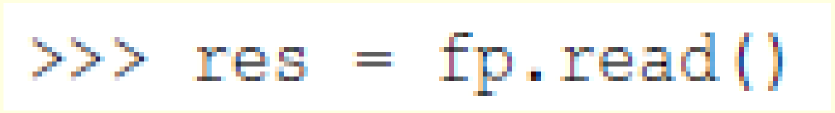


```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

- The argument is a string that contains a shell command. The return value is a file pointer that behaves just like an open file.

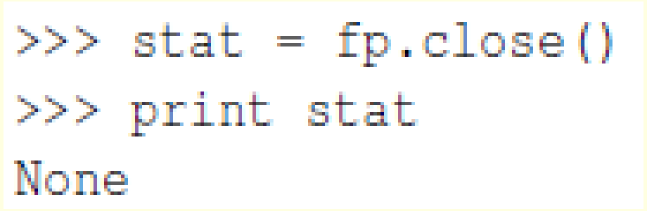
Pipes (cont'd)

- **You can read the output from the ls process one line at a time with readline or get the whole thing at once with read:**



```
>>> res = fp.read()
```

- **When you are done, you close the pipe like a file:**

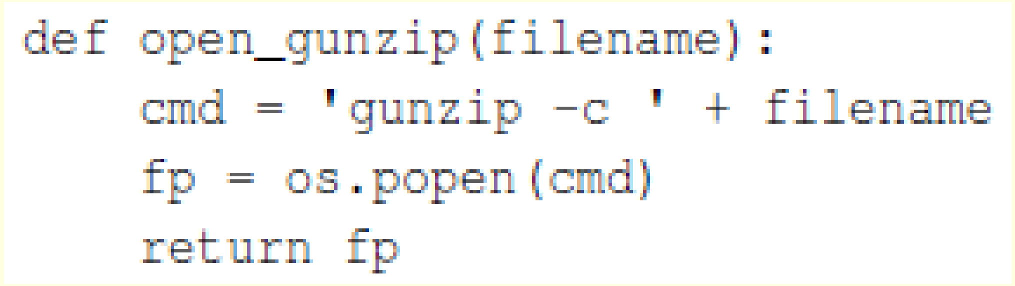


```
>>> stat = fp.close()
>>> print stat
None
```

  - The return value is the final status of the ls process; None means that it ended normally (with no errors).

Pipes (cont'd)

- A common use of pipes is to read a compressed file incrementally; that is, without uncompressing the whole thing at once. The following function takes the name of a compressed file as a parameter and returns a pipe that uses gunzip to decompress the contents:



```
def open_gunzip(filename):
    cmd = 'gunzip -c ' + filename
    fp = os.popen(cmd)
    return fp
```

Writing modules

- Any file that contains Python code can be imported as a module. For example, suppose you have a file named wc.py with the following code:

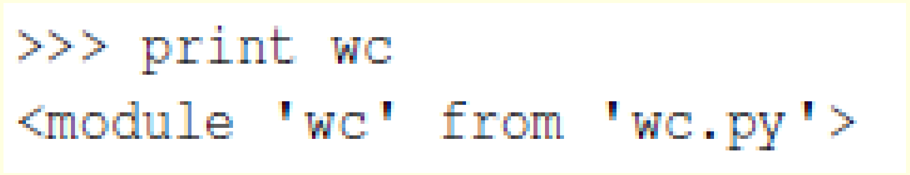
```python
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print linecount('wc.py')
```

- If you run this program, it reads itself and prints the number of lines in the file, which is 7. You can also import it like this:

```
>>> import wc
7
```

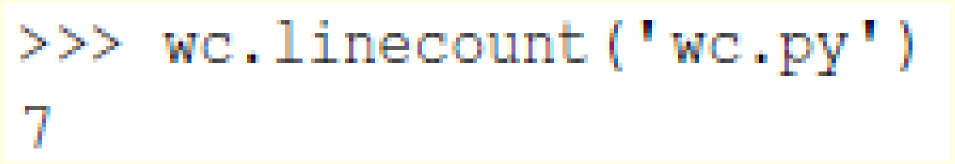
Writing modules (cont'd)

- Now you have a module object wc:



```
>>> print wc
<module 'wc' from 'wc.py'>
```

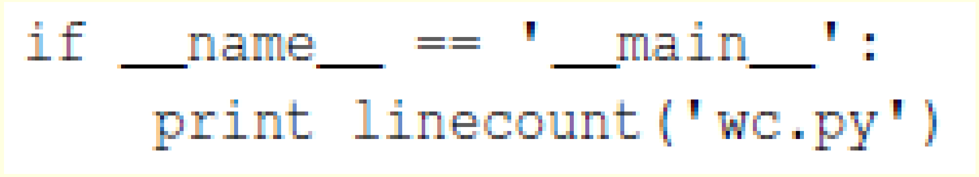
- That provides a function called linecount:



```
>>> wc.linecount('wc.py')
7
```

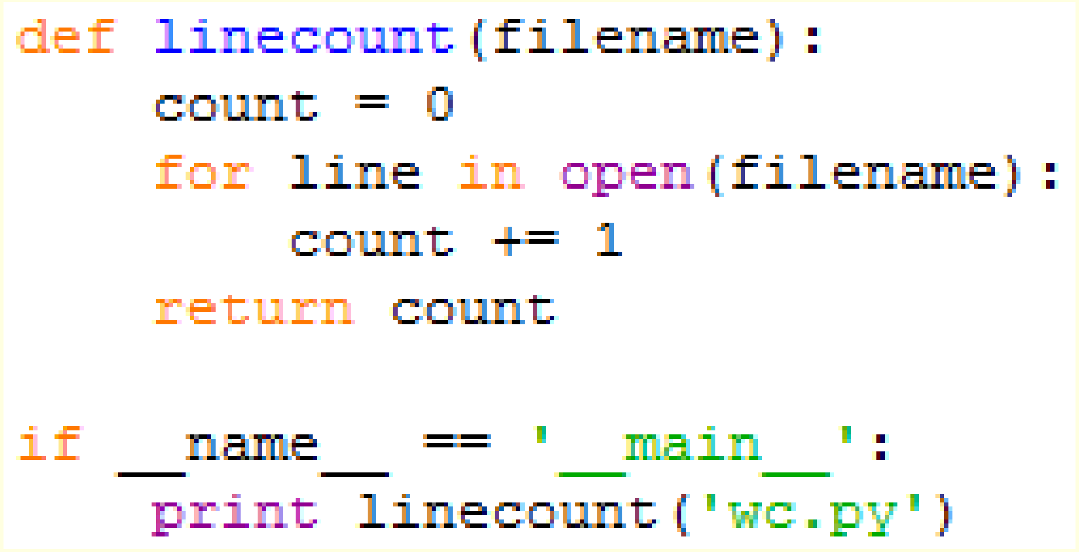
Writing modules (cont'd)

- Programs that will be imported as modules often use the following idiom:



```
if __name__ == '__main__':
    print linecount('wc.py')
```

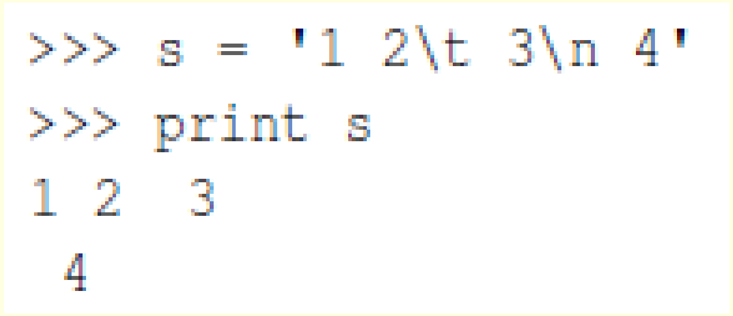
- __name__ is a built-in variable that is set when the program starts. If the program is running as a script, __name__ has the value __main__; in that case, the test code is executed. Otherwise, if the module is being imported, the test code is skipped.



```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

if __name__ == '__main__':
    print linecount('wc.py')
```

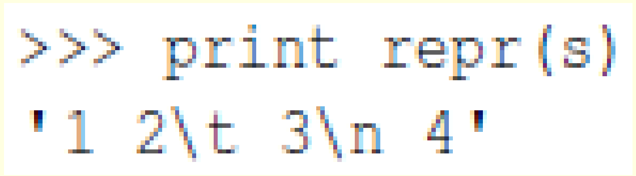
Debugging

- When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs and newlines are normally invisible:



```
>>> s = '1 2\t 3\n 4'
>>> print s
1 2   3
 4
```

- The built-in function repr can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:



```
>>> print repr(s)
'1 2\t 3\n 4'
```

Debugging (cont’d)

- **One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented \n. Others use a return character, represented \r. Some use both. If you move files between different systems, these inconsistencies might cause problems.**