Standard Input, Output, and Error

Command-line gurus are already familiar with the concept of standard input, standard output, and standard error. This section is for the rest of you.

Standard output and standard error (commonly abbreviated `stdout` and `stderr`) are pipes that are built into every UNIX-like system, including Mac OS X and Linux. When you call the `print()` function, the thing you're printing is sent to the `stdout` pipe. When your program crashes and prints out a traceback, it goes to the `stderr` pipe. By default, both of these pipes are just connected to the terminal window where you are working; when your program prints something, you see the output in your terminal window, and when a program crashes, you see the traceback in your terminal window too. In the graphical Python Shell, the `stdout` and `stderr` pipes default to your "Interactive Window".

sys.stdin, sys.stdout, sys.stderr.

```
for i in range(3):
     print('PapayaWhip')              #①
#PapayaWhip
#PapayaWhip
#PapayaWhip

import sys
for i in range(3):
     l = sys.stdout.write('is the')      #②
#is theis theis the

for i in range(3):
     l = sys.stderr.write('new black')  #③
#new blacknew blacknew black
```

① The `print()` function, in a loop. Nothing surprising here.

② `stdout` is defined in the `sys` module, and it is a stream object. Calling its `write()` function will print out whatever string you give it, then return the length of the output. In fact, this is what the `print` function really does; it adds a carriage return to the end of the string you're printing, and calls `sys.stdout.write`.

③ In the simplest case, `sys.stdout` and `sys.stderr` send their output to the same place: the Python ide (if you're in one), or the terminal (if you're running Python from the command line). Like standard output, standard error does not add carriage returns for you. If you want carriage returns, you'll need to write carriage return characters.

`sys.stdout` and `sys.stderr` are stream objects, but they are write-only. Attempting to call their `read()` method will always raise an `IOError`.

```
import sys
sys.stdout.read()
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 2, in <module>
# sys.stdout.read()
#io.UnsupportedOperation: not readable
```

Redirecting Standard Output

`sys.stdout` and `sys.stderr` are stream objects, albeit ones that only support writing. But they're not constants; they're variables. That means you can assign them a new value — any other stream object — to redirect their output.

```
import sys

class RedirectStdoutTo:
    def __init__(self, out_new):
        self.out_new = out_new

    def __enter__(self):
        self.out_old = sys.stdout
        sys.stdout = self.out_new

    def __exit__(self, *args):
        sys.stdout = self.out_old

print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
    print('B')
print('C')
```

Check this out:

```
you@localhost:~/diveintopython3/examples$ python3 stdout.py
A
C
you@localhost:~/diveintopython3/examples$ cat out.log
B
```

> Did you get this error?

```
you@localhost:~/diveintopython3/examples$ python3 stdout.py
  File "stdout.py", line 15
    with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
                                                              ^
SyntaxError: invalid syntax
```

If so, you're probably using Python 3.0. You should really upgrade to Python 3.1.

> Python 3.0 supported the `with` statement, but each statement can only use one context manager. Python 3.1 allows you to chain multiple context managers in a single `with` statement.

> Let's take the last part first.

```
print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
    print('B')
print('C')
```

That's a complicated `with` statement. Let me rewrite it as something more recognizable.

```
with open('out.log', mode='w', encoding='utf-8') as a_file:
    with RedirectStdoutTo(a_file):
        print('B')
```

As the rewrite shows, you actually have *two* with statements, one nested within the scope of the other. The "outer" `with` statement should be familiar by now: it opens a UTF-8-encoded text file named `out.log` for writing and assigns the stream object to a variable named `a_file`. But that's not the only thing odd here.

```
with RedirectStdoutTo(a_file):
```

Where's the `as` clause? The `with` statement doesn't actually require one. Just like you can call a function and ignore its return value, you can have a `with` statement that doesn't assign the `with` context to a variable. In this case, you're only interested in the side effects of the `RedirectStdoutTo` context.

What are those side effects? Take a look inside the `RedirectStdoutTo` class. This class is a custom context manager. Any class can be a context manager by defining two special methods: `__enter__()` and `__exit__()`.

```
class RedirectStdoutTo:
    def __init__(self, out_new):    #①
        self.out_new = out_new

    def __enter__(self):            #②
        self.out_old = sys.stdout
        sys.stdout = self.out_new

    def __exit__(self, *args):      #③
        sys.stdout = self.out_old
```

① The `__init__()` method is called immediately after an instance is created. It takes one parameter, the stream object that you want to use as standard output for the life of the context. This method just saves the stream object in an instance variable so other methods can use it later.

② The `__enter__()` method is a special class method; Python calls it when entering a context (i.e. at the beginning of the `with` statement). This method saves the current value of `sys.stdout` in `self.out_old`, then redirects standard output by assigning `self.out_new` to `sys.stdout`.

③ The `__exit__()` method is another special class method; Python calls it when exiting the context (i.e. at the end of the `with` statement). This method restores standard output to its original value by assigning the saved `self.out_old` value to `sys.stdout`. Putting it all together:

```
print('A')                                                                     #①
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):  #②
    print('B')                                                                 #③
print('C')                                                                     #④
```

① This will print to the ide “Interactive Window” (or the terminal, if running the script from the command line).

② This with statement takes a *comma-separated list of contexts*. The comma-separated list acts like a series of nested `with` blocks. The first context listed is the “outer” block; the last one listed is the “inner” block. The first context opens a file; the second context redirects `sys.stdout` to the stream object that was created in the first context.

③ Because this `print(`) function is executed with the context created by the `with` statement, it will not print to the screen; it will write to the file `out.log`.

④ The `with` code block is over. Python has told each context manager to do whatever it is they do upon exiting a context. The context managers form a last-in-first-out stack. Upon exiting, the second context changed `sys.stdout` back to its original value, then the first context closed the file named `out.log`. Since standard output has been restored to its original value, calling the `print()` function will once again print to the screen.

Redirecting standard error works exactly the same way, using `sys.stderr` instead of `sys.stdout`.