LINUX VOICE TUTORIAL

# CODE NINJA: STDIN AND STDOUT

BEN EVERARD

Make your Python programs work with the Linux shell to build your own command line utilities.

**WHY DO THIS?**

- Understand one of the basic principles of how the command line works.
- Write your own utilities for the Linux shell.
- Get more out of Python.

If you've used the Linux command line for more than the most basic uses, you'll be familiar with the concepts of standard in (**stdin**) and standard out (**sdtout**). These are channels the shell pushes data through when you link commands with the pipe character (**|**). For example :

```
dmesg | grep "USB"
```

Here, the first command (**dmesg**) sends the kernel message buffer to **stdout**. The **|** symbol takes the **stdout** of one command and pushes it into the **stdin** of the next. The **grep** command then goes through each line in its **stdin** and prints out only those containing the string 'USB'. Using this concept of **stdin** and **stdout**, we've managed to build a simple little tool for checking the kernel messages about USB devices.

In the Unix philosophy, as much as possible, programs should be simple commands that work with **stdin** and **stdout**. If they do, then they can be combined with the other Unix commands in a huge number of ways. Working this way means you can focus your program on doing one thing well. Rather than having to program every possible feature, you just focus on the core of the program and the user can use other Unix tools to gain more features by chaining commands together.

**"In Unix, programs should be simple commands that work with stdin and stdout."**

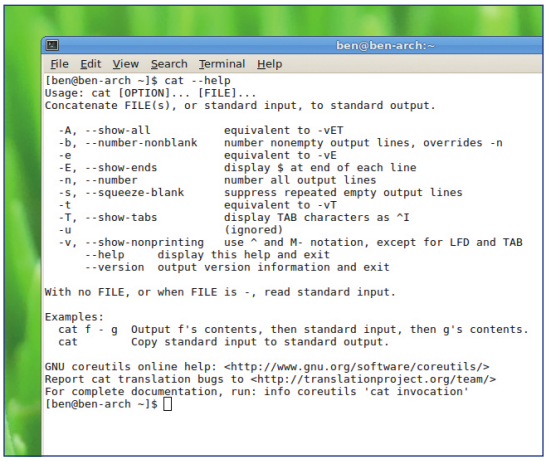
Let's take a look at how we could start building our own implementations of standard Unix tools in Python that interact using **stdin** and **stdout** so they can be chained together.

Let's start with **cat**. This stands for catenate. Basically, it's used to send the output of one or more files to **stdout**. In Python, this is really easy:

```
from __future__ import print_function
import sys

for arg in sys.argv[1:]:
    with open(arg) as file:
        for line in file:
            print(line, end="")
```

All we have to do to send something to **stdout** in Python is print it. Here we've used the **print** function, which is the standard way of doing it in Python 3. This also works in Python 2 if you include the first import line. The reason we've done it this way is because we'll use some features of the **print** function a little later on, and want to keep things consistent.




**GNU's cat utility has many more features than we've implemented here, but it still works on the same basis of reading files and sending their contents to stdout.**

Each line that we read from the file includes the newline character at the end, so we include the parameter **end=""** to stop the **print** function from adding a separate newline character which would leave a blank line between every line form the file.

## In and out

We can get the arguments to the command from **sys.argv**. This is simply a list of everything that's supplied to the Linux Shell (see "Arguments" box). Here, we just use this to loop through every argument to the command and hope it's a file. So, the following would send the output of **file1** and **file2** to **stdout**:

```
python cat.py file1 file2
```

You can chain this with other Unix commands such as **wc** (word count) like this:

```
python cat.py /usr/share/dict/words | wc
```

**wc** outputs the number of lines, words and characters in a file. Let's implement that next:

```
from __future__ import print_function
import sys

num_lines = 0
num_words = 0
num_chars = 0

for line in sys.stdin:
    num_lines += 1
    num_words += len(line.split())
    num_chars += len(line)
```

## Arguments

Interacting nicely with the Linux command line environment isn't all about using **stdin**, **stdout** and **stderr**. Another aspect of the command line environment is properly parsing arguments and flags. For example, in the **cat** Python script, we take a list of files as arguments. We did this using **sys.argv**. This provides a list of everything that's passed to the command (the first item in the list is the command itself; in that example, we only included everything from the second element onwards).

However, if your command takes a range of options, it can be quite complex to correctly parse this. Fortunately, there's a module to help: **argparse**. Let's take a look at a simple example. This will be a command a little like **echo**, but it will send the output to **stdout** unless the **--stderr** flag is present, in which case it will send it to **stderr** (this is just an example, and the normal **echo** command doesn't have these flags).

We'll do this using the **argparser** module (new in Python 2.7, so this won't work if you're using an older version). Using this, you only have to specify what options you have, and the module will take care of parsing them and putting them in variables so you can access them:

```
from __future__ import print_function
import argparse, sys
parser = argparse.ArgumentParser(description='echoing to stdout and stdin')
parser.add_argument('echo',  help='The stuff to print on screen')
parser.add_argument("--stderr", help="print the stuff to stderr (otherwise print to stdout)", action="store_true")
args = parser.parse_args()
if args.stderr:
        print(args.echo, file=sys.stderr)
else:
        print(args.echo)
```

You can now check it's working with the following lines:

```
python echo.py --stderr "hello world" | wc
python echo.py  "hello world" | wc
```

The **argparser** module even takes care of creating your help text (from the description and help parameters used when creating the parser), so you can run:

```
$ python echo.py --help
usage: echo.py [-h] [--stderr] echo
echoing to stdout and stdin
positional arguments:
 echo        The stuff to print on screen
optional arguments:
 -h, --help  show this help message and exit
 --stderr    print the stuff to stderr (otherwise print to stdout)
```

You can fine tune **argparser**'s operation to almost any degree, and it has excellent documentation at: **https://docs.python.org/2/howto/argparse.html**.

```
print("\t", num_lines,"\t", num_words,"\t", num_chars)
```

As you can see, getting input from **stdin** is quite straightforward. It's read in in lines just like files are.

These two tools can be linked together with:

```
python cat.py /usr/share/dict/words | python wc.py
```

So far, we haven't implemented any error checking. So, if you try to **cat** a file that doesn't exist, you get a Python exception:

```
$python cat.py /nosuchfile | python wc.py
```

This might seem a little strange, because the error from the first Python command hasn't been sent to **stdin** of the second command. Instead it's been displayed on the screen. The reason for this is that errors don't go to **stdout**: instead, there's a separate channel for them called standard error (**stderr**). This means that a program can send error messages without corrupting the information going to **stdout**. For example, you might have used **cat.py** to output the contents of several files, but one of them doesn't exist. Using **stderr**, you still send the output of all the files to the next command in the chain, but also alert the user to the problem. Let's take a look at how to modify our **cat** command to deal with this:

```
from __future__ import print_function
import sys, os.path
for arg in sys.argv[1:]:
  if not os.path.isfile(arg):
    print("whoops, there appears to be a mistake. I can't find", arg, file=sys.stderr)
  else:
    with open(arg) as file:
      for line in file:
        print(line, end="")
```

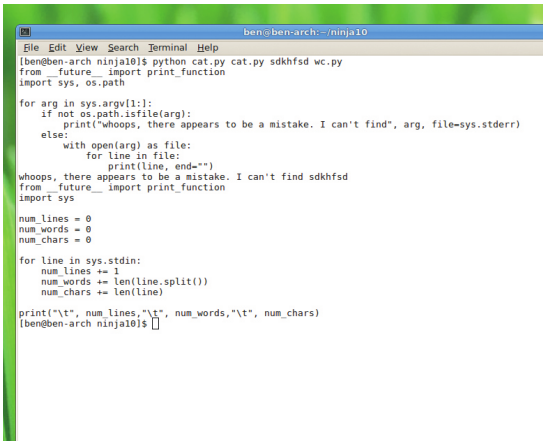
As you can see, all you have to do to print something to **stderr** is include the parameter **file=sys.stderr** in the **print** function, and you can print to it just like you print to **stdout**.

**Stderr** doesn't have to be printed on the screen though; you can send it to a file instead. If you're running commands through some automated means such as **at** or **cron**, it can be useful to collect any errors. This is done with:

```
command 2> file
```

Anything a command sends to **stdout**, will be printed on the screen (you could also pipe it into further commands). Another option is to tell *Bash* to send **stderr** to **stdout**. Doing this will combine **stdout** and **stderr** into a single flow of text and pass it along to any future commands:

```
command1 2>&1 | command 2
```




Even if one of the arguments to our **cat** program isn't a file, it will still handle the rest correctly.