# Coding: Pocket

**PART 2** Can't remember all those fiddly command-line flags? **Mike Saunders** shows you how to create GUI-like alternatives for common admin utilities.

**Our expert**

**Mike Saunders** started programming at age eight on the ZX Spectrum, and loves to explore every language in existence. Except COBOL
**http://mikeos.sourceforge.net**

Picture the scene: you're logged into a remote server via SSH, or you've installed a new graphics card and you're left staring at the command line. You need to enter a command, but you can't remember the zillion options that go along with it. You're stuck – all you can do is consult the manual pages and pore through pages of waffling technical gobbledygook. We've all been there, and no matter how experienced you are with Linux, sometimes you need to accomplish a job quickly without sifting through masses of reading material.

In this month's coding project, we're going to solve this problem – and have fun along the way! We'll show you how to write a dialog-based program that gives you options one-by-one, so that you don't need to consult the man pages. In this guide, we'll show you how to write a nifty front-end for the *useradd* utility, a command which (unsurprisingly) lets you add user accounts to your Linux installation. Like many administration tools, **useradd** requires a long string of options and parameters; we're going to make it much simpler by creating an interactive dialog-driven version called *UserMaster*.

Now, **useradd** is a fairly trivial tool and there's an alternative command, **adduser**, which prompts you step-by-step. But here we'll give you the code and techniques to make your own config tools – so at the end, you'll be able to write user-friendly utilities for starting services, clearing temporary files, or anything else you'd normally do at the shell prompt. Not only will this save you vital time when you're at the command line, but it means you can write config tools for other people who may log into your machines. If you run a server and don't want users trying to guesstimate super-complex commands, you can knock together a quick dialog-based version.

As with last month's code project (an IRC bot in Perl), this guide assumes a smattering of programming knowledge. Don't worry: you don't need to be a coding whizz, but if you've played around with another language before, you'll be off to a flying start. But even if you've never written a line of code in your life, hopefully this tutorial will show you how programs fit together and what you can do. Enough chit-chat – let's get started with *UserMaster*...

## Pint-sized Python

We'll base our project on Python, a hugely popular programming language that's used in everything from tiny scripting tools to fully-fledged applications. If you've ever installed Fedora or Red Hat Enterprise Linux, you'll have seen a Python application in all its glory: the *Anaconda* installer is written almost entirely in the language. Python is blissfully easy to read (perhaps the most human- readable computing language around), so we don't need to detail minutiae of the syntax. Indeed, if you're a regular Pythoner you can skip over this bit. But for a quick overview of the language, enter this into a text file and save it in your home directory as test.py:

```
print "How easy is this?"

x = 1
y = 2
z = x + y

print "Result of x + y is", z
```

Now open up a terminal and enter:

```
python test.py
```

The Python interpreter runs and executes our code. It prints a string to the screen, then adds together two variables, and displays the results. You can call variables almost anything you want, providing they don't conflict with Python keywords. So here, **x**, **y** and **z** are fine generic variable names. You don't need to faff around with curly braces either, as Python uses indentation for code blocks, as follows:

```
def saysmithers():
    print "Excellent, Smithers"

print "Calling saysmithers function..."
saysmithers()
```

» **Last month** The first in the series: using Perl to build your very own IRC bot.

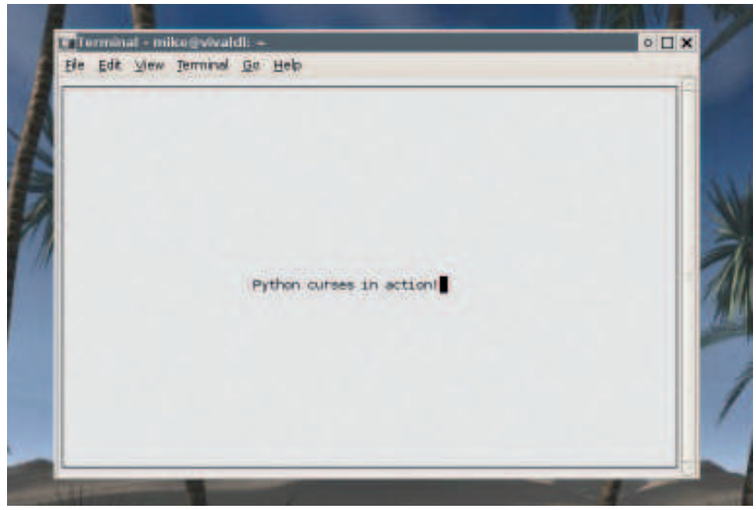
# config tools

Here, we declare a subroutine (*aka* function) at the start – it's not executed immediately, but only when we call it. So Python skips over the first chunk of code, as it's a separate function, and begins with the **"Calling"** print line. Then our code calls the **saysmithers()** routine which prints the **"Excellent"** message. Code that belongs to a function or code block is marked with tab indents, as you can see from our examples. You can pass parameters to functions thusly:

```
def saystuff(mystring):
    print "You said:", mystring

saystuff("Bach rules")
saystuff("So does Telemann")
```

So here, we have a function that just spits out whatever text it receives. Again, it doesn't execute straight away – as it's a standalone function – and code starts at the first non-function line. We call the **saystuff** function twice, with different text as parameters, and our function simply prints the supplied text to the screen. Super simple!




**› Our first Python *curses* program! It's not much to look at, but at least we have some control over the terminal window.**

## Commence cursing

For our *UserMaster* tool, we don't want a boring command line interface – after all, that's what the standard user management programs use. Instead, we'll create a friendlier menu and dialog-driven interface using the *curses* library. If you've never heard of it before, *curses* is a library of routines that let you create boxes, windows and text-entry panels on text-based displays. It's a play on the word 'cursor', and on modern Linux systems, its implementation is usually called *ncurses*.

Essentially, *curses* bridges the gap between the command line and graphical desktops. In text mode, you can't have zillions of colours and pixel-perfect mouse control, but you can still have more than a string of letters. *curses* lets you create semi-GUI applications with windows and boxes – if you've ever used the text-based Debian, Ubuntu or Slackware installers, you'll know what we mean.

Here's an example. As before, tap this into a text file and save it into your home directory as test.py. Then execute it with **python text.py** in a terminal:

```
import curses

myscreen = curses.initscr()

myscreen.border(0)
myscreen.addstr(12, 25, "Python curses in action!")
myscreen.refresh()
myscreen.getch()

curses.endwin()
```

Python includes a set of bindings to the *curses* library – in other words, you can utilise the features of *curses* in your Python programs, providing you **import** the *curses* module as specified in the first line of this code. The second line of code creates a new *curses* screen object. Python is an object-oriented language, so in this case, we create a new screen workspace (called **myscreen**) and tell *curses* to initialise itself (**initscr**). Now we have control of the terminal window.

Next, we have four lines which tell curses what to do with our new **myscreen** object. First, we tell **curses** to draw a border around our screen, which immediately makes it look prettier than a vanilla CLI program. In the following line, we tell **curses** to add a text string to our **myscreen** object, specifying the position where it should be printed. Normally, text terminals are 80x25 characters if you're at the raw Linux terminal, or 80x24 if you're in GUI mode and have launched *xterm*, *Konsole* or *Gnome-Terminal*. So we print a line of text 12 characters down and 25 across – the middle of the screen, unless you've resized your terminal of course!

The third line is very important: we need to call **refresh** on our **myscreen** object to tell **curses** that we're ready to go. Hypothetically, **curses** could refresh the screen every for every command, but that would be very slow with complex programs. So instead, it lets us build up our display bit-by-bit, and then draw the whole lot in one fell swoop. Calling **refresh** here renders our border and text screen to the screen.

Finally, we have two remaining lines of code: the first, which calls **getch** on our **myscreen** object, tells *curses* to wait for a keypress (get character). This pauses execution until we hit a key. Finally, we call **endwin** which shuts down *curses* and returns us to our normal command-line interface. When you run this program, you'll see the results in the first screenshot – very basic, but now we're ready to write *UserMaster*!

## UserMaster 1.0

Here's the code for our program. You can find it on our DVD in the **Magazine/Python** section, but for now, just read over it – you should be able to understand what it's doing. If you've never used Python before, there are a few new concepts in here, but we'll go over them in a moment. »

```
#!/usr/bin/env python

from os import system
import curses

def get_param(prompt_string):
    screen.clear()
    screen.border(0)
    screen.addstr(2, 2, prompt_string)
    screen.refresh()
    input = screen.getstr(10, 10, 60)
    return input

def execute_cmd(cmd_string):
    system("clear")
    a = system(cmd_string)
    print ""
    if a == 0:
        print "Command executed correctly"
    else:
        print "Command terminated with error"
    raw_input("Press enter")
    print ""

x = 0

while x != ord('4'):
    screen = curses.initscr()

    screen.clear()
    screen.border(0)
    screen.addstr(2, 2, "Please enter a number...")
    screen.addstr(4, 4, "1 - Add a user")
    screen.addstr(5, 4, "2 - Restart Apache")
    screen.addstr(6, 4, "3 - Show disk space")
    screen.addstr(7, 4, "4 - Exit")
    screen.refresh()

     x = screen.getch()

    if x == ord('1'):
        username = get_param("Enter the username")
        homedir = get_param("Enter the home directory, eg /home/nate")
        groups = get_param("Enter comma-separated groups, eg adm,dialout,cdrom")
        shell = get_param("Enter the shell, eg /bin/bash:")
        curses.endwin()
        execute_cmd("useradd -d " + homedir + " -g 1000 -G " + groups + " -m -s " + shell + " " + username)
    if x == ord('2'):
        curses.endwin()
        execute_cmd("apachectl restart")
    if x == ord('3'):
        curses.endwin()
        execute_cmd("df -h")

curses.endwin()
```

That's quite a bit of code, but the majority of it is spent dealing with the screen, so it's actually not very complicated. This program starts up with a menu offering four choices:

1. Add a user account
2. Restart the *Apache* web server
3. Display available disk space
4. Exit.

It doesn't matter if you don't have *Apache* installed – this is just to demonstrate how you can expand the program into a more versatile config tool.

The first line lets us run the program without having to specify the Python interpreter by hand (**python usermaster.py**). More on that in a moment. Then we have two **import** lines: the first gives us access to Python's **system** command, which lets us execute external programs (*ie* binaries you'd find in **/usr/bin** *etc*), while the second rolls in **curses** as described before.

Next we have two functions. The first takes a prompt string parameter, and then asks the user for a string before returning it back. Basically, in human-speak, it says, "Tell me what question to put on the screen, then I'll get some text from the user and send it back." We use this function later in the code to get information for the **useradd** command.

The second function, meanwhile, executes a command on the system. It takes a string parameter, clears the screen, and then executes the command in the

```
a = system(cmd_string)
```

line. Now, what's that **a** doing there? Well, we want to know whether the command was executed successfully. When we use Python's **system** routine to run a program, it returns **0** if the program ran successfully, or another number if it encountered an error. Thanks to this, we can print out a message to show whether the command worked – for instance, whether the user-adding process succeeded or failed. Later on, you may expand *UserMaster* to do other tasks, some of which may not generate any visible output (*eg* running a **cron** job), but you'll still know if it worked or messed up.

So, those two functions lay the foundations for our OS. Python will start executing the program at the **x = 0** line, which just sets up a variable for the next line, which checks to see if **x** contains the character **4**. Well, at this stage, we haven't got any input from the user yet – but we will in a few steps. We fire up **ncurses**, clear the screen, set a border and then print some options strings to the screen. Then we have:

```
x = screen.getch()
```

The user has a list of commands to choose from; whatever he/she enters is stored in the **x** variable. So, the following **if** lines check to see what's in **x**: if it's one, we're in user-adding mode; if it's **2**, we restart *Apache*; **3**, and we show disk space; **4**, and we exit. But hang on a minute, where are we checking for **4** there? Well, we're doing that at the start of the loop, in the

```
while x != ord('4'):
```

line. If the user enters anything other than **1**, **2** or **3**, control jumps back to the **while** line, which then stops executing the code block. As mentioned earlier, code blocks in Python are marked with tab indents, so after our program has done checking for the **3** key to be pressed, it goes back to the start of the loop – the **while** line. It does this because the following (and final) line, the **endwin()**, is not indented and therefore not part of the **while** loop code block. So the end of the indents shows where the code block finishes.

**Quick tip**

When programming with Python and *curses*, you may find that your terminal window goes funny or locks up if your program crashes. This is because *curses* didn't shut down properly and return your terminal to its normal state. If this happens, just enter **reset** into your terminal (even if you can't see the characters!) and you'll be back to CLI normality

## Cultivating curses

In this tutorial we're using some of **curses**'s most common routines – clearing the screen, drawing a border, getting input *etc*. But it's a lot more capable than that, providing a complete set of window management routines. Yes, it seems futile to have the concept of windows at a text terminal, but it lets you render and remove panels (*eg* question dialogs) while keeping the text beneath intact. You can even interface with the mouse in X terminals!

Annoyingly, documentation for the Python *curses* bindings is rather sketchy at present; the official Python documentation only has a short chapter on the library and there's not much else doing the rounds. If you want to do more with *curses*, your best bet is the Linux Doc Project HOWTO at **http://tinyurl.com/rz7fs** – it was originally written with C coders in mind, but you'll find many of the same function calls in the Python bindings.

## Python: going further

If this has been your first foray into programming, hopefully you've been pleasantly surprised by Python's simplicity. Unlike many programming languages which have their roots in the 1970s (or even earlier), Python is relatively modern, having been developed in the early 1990s. It's not tangled up in obscure throwbacks and awkward fudges to maintain compatibility.

So it's an excellent starting language, and you can find out more at **www.python.org**. Also see the **Magazine/Python** section of our DVD for the complete Python documentation – extract the Zip archive in your file manager and open **index.html** in the resulting **Python-Docs-2.5** directory to get started. You'll find an outstandingly thorough tutorial which explains all aspects of the language. Yes, the name was inspired by *Monty Python's Flying Circus*. He's probably pining for the fjords! *etc*.

## No buts, only ifs

Let's look at those **if** statements. Depending on what the user entered, we run a specified program using our **execute_cmd()** function. For options **2** and **3**, we simply disable **curses** (so that it doesn't interfere with our program output), and then run the program. **execute_cmd()** tells us if the program executed correctly, before handing control back to the main code – so we see the menu again.

But it's the first option that's the most interesting. Here, we use our **funky get_param()** function to display a series of prompts, getting input from the user each time. We ask the user for an account name, default shell and so forth; nothing too taxing. Then we join all these bits of info together and feed them to the system's standard **useradd** command, with appropriate command-line parameters (for example **-d** for home directory). Hence this is why the **execute_cmd()** line here is very long – it joins together all the strings and flags that the command needs in order to work.

Grab **usermaster.py** from the **Magazine/Python** section of our DVD, copy it to your home directory, then fire up a terminal and switch to root (**sudo bash** in Ubuntu or **su** in others). Enter:

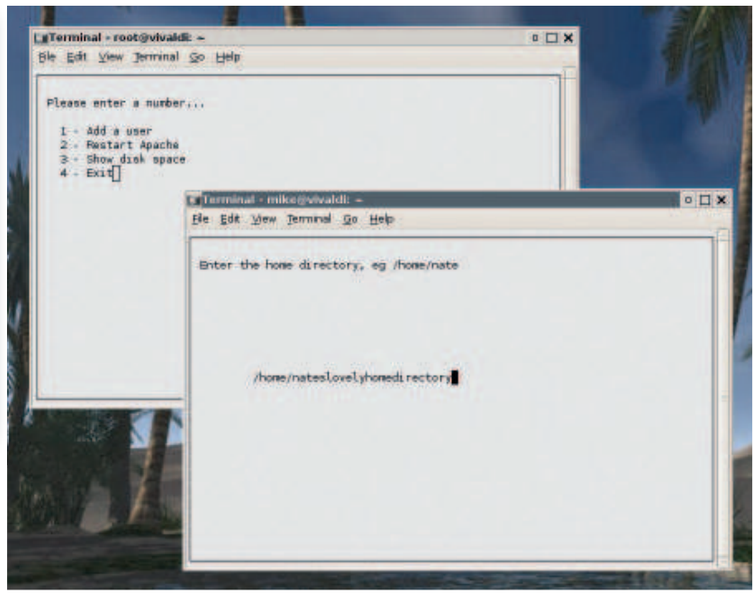
```
chmod +x usermaster.py
mv usermaster.py /usr/sbin/usermaster
```

Now **usermaster** is installed into the system path (for administrator commands), so you can run it simply by entering **usermaster**. Note that you need to run it as root though, as normal users can't create other user accounts! You can now try creating a user via option 1 of the program – don't worry about setting up the exact right groups now. When you've created a user via our tool, you'll also need to set up a password as follows:

```
passwd nate
```

Replace **nate** with the username that you created. To keep the code listing short, we haven't included this facility in *UserMaster*, but it's an ideal first mini-project for you: adding a password setting/changing facility! You'd just need to add another option to the menu, get the account name via a **get_param()** call, then execute **passwd** on that name.

The exact details of users and groups and how they are related varies quite distinctly from linux distribution to distribution, so you'll want to tailor *UserMaster* for your own needs. As your normal user account, enter **groups** in a terminal to see which groups a standard user belongs to, and also see the *useradd* manual page (**man useradd**) to learn more about the options we pipe to the command.




› ***UserMaster 1.0*** **in action: the top-left window showing the main menu, and the other window getting input from the user.**

There's so much more you can do with *UserMaster* – you can add options for nigh-on any system administration job, helping you (and other users) avoid constant command line hassles. Good luck, and if you have any questions, try our programming forum at **www.linuxformat.co.uk/forums/** – if you're encountering a problem, there's bound to be someone who's already experienced a similar one and thought of a solution! LXF

## IRC bot tutorial

**Doesn't a bot enslave other people's PCs or harvest email addresses for nefarious purposes?**

Not the one we're using! Many remote monitoring tools tend to flood your inbox with extraneous data when you're away from your machine. Instead, you can use your own IRC bot to find out basic things like disk space, RAM used etc. purely by private message! See **LXF100**'s tutorial for more info!


» **Next Month** Revolutionise your programming efficiency the easy way – cheat!