Adam Traub

5/21/2009

Senior Capstone

**Monty: A Python Based Linux Shell  
Reference Manual**

**Introduction**

Monty is a python based Linux shell. Monty was developed to be a very simple tool, bearing this in mind the easiest way to fully understand what Monty does would be to analyze the “Who, What, Why, and how” associated with it.

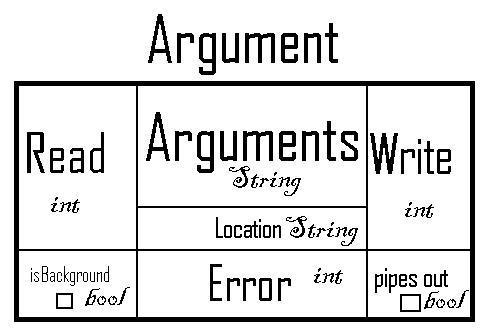
**Who was Monty developed for?** Shells are typically very cryptic and confusing. Reading shell code and trying to fully understand what is going on can be a painful experience. Monty was created to be used as a starting point for programmers interested in shell development. Its’ clear syntax and simplicity make it an ideal launchpad for someone trying to understand shell code. Monty is not simply an educational tool. It is a fully functioning shell that one day may actually be used to the same degree as shells such as XFCE, bash, or even Gnome. It’s conceivable that Monty could be developed further and further until its abilities can rival that of other popular shells.

**What is Monty exactly?** Linux shells act as an interface between the Linux Kernel and the user. It gives the user the ability to use all of the tools Linux has available. Monty is a command line shell built using the Python programming language.

**Why was Monty developed?** Monty was developed based on the desire of brining simplicity to shell programming. The Python programming language is one of the easiest languages for a beginner to computer science. This language is one that is so simple that some schools, including Randolph-Macon College, have opted to use it for the introduction to Computer Science classes. If a functional shell could be created using this language, it could mark a tremendous leap forward in undergraduate students understanding and utilizing Unix/Linux systems. If the shell could be further developed, it may offer educational value for building graphical desktop environments.

**How was Monty developed?** The remainder of this manual will attempt to cover that topic. Monty has been developed over the course of several months using Py3k. Monty posed many challenges and often times displayed the clear difference between C and Python. There are no books that cover this topic, there are no guides, no manuals, no charts, no walkthroughs, no examples. Monty is an original. Several hours of research found zero people who had tried to implement a python based Linux shell and succeeded. All people who tried eventually hit pitfalls or abandoned their projects leaving behind little or no source code. How was Monty developed? Monty was developed through the tireless efforts of one student under the Guidance of his professor.

Monty is a highly object oriented shell. Because of this, an excellent approach to truly grasping how Monty works would be to examine each class individually while also paying attention to how these classes work together. After that will be the key features of Monty, its bugs, and some thoughts about its future. This bottom up approach should create a clear understanding of how Monty works and what the future might hold for it.



**The Argument Class**

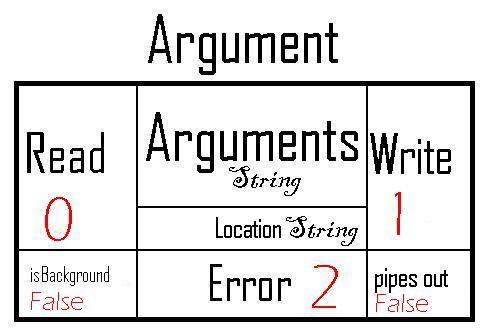
The **Argument** class is arguably the most fundamentally important class to Monty. It helps create a simplicity that simplifies the entire process of shell programming. When a given argument is executed in a shell such as bash, there are a lot of variables that are created associated with it. Monty creates one object that holds all these variable in an intelligent way.

An **Argument** object consists of a Read end, a Write end, and an Error end. These three values are used in **piping** **and redirection**. The **isBackground** property of **Argument** accomodated **background processing**. The Location value is used primarily for user functions in the **User Defined Library** or **UDL**. The Arguments attribute is the code that actually gets executed by the Linux Kernel, except in situations where **Builtin Commands** are used. By encapsulating all this data into one Object, the **Argument** could easily be changed and updated without having to have many variables floating around. All of these values except for arguments and location are given default values. What this means is that certain processes, such as duping, can be automated to occur every time a command is executed rather than based on whether or not it’s actually necessary. The only drawback to this approach is a slight loss in speed. The pipesOut attribute is not currently utilized, however it may prove to be useful in **piping and redirection** as Monty is further developed.

The **Argument** class is used by the **Splitter**, the **Processor,** and the **Executer**. All three of these classes have a very important role for an Argument object. The **Splitter** creates the **Argument**, the **Processor** interprets it, and the **Executer** utilizes it.

**The Splitter Class**

The **Splitter** class is what the user communicates with. The **Splitter** creates a prompt for the user, and then waits for user input. It also recognizes CTRL-D for when the user wants to exit. There are a few interesting things to note about this class. Because it is the only class that recognizes the CTRL-D command, the shell will not exit if the user presses CTRL-D and another object has control of the shell. Another interesting thing to note is that the “makePrompt” method in this class is ran every time the **Splitter** displays it. Therefore, the prompt is dynamic rather than static. Because of this, the prompt will change when the user changes directories, and will display a different username depending on who is using Monty.

After the user enters in a command, the **Splitter** turns this command into an **Argument** object. By default, the isBackground attribute is set to False, but the **Splitter** does check for **background processing** before it returns the newly created **Argument** object. If the user only typed in white space, the **Splitter** returns a value of None to inform the shell that it does not need to try to process the data.

**The Processor Class**

While the **Argument** class may be the most important, no class does more work than the **Processor**. The **Processor** must take the Argument that the **Splitter** creates and utilize it. The **Processor**  must first preprocess the data. This involves looking at the arguments attribute of the **Argument** object and check for special characters. These special characters signify **piping and redirection**. Preprocessing will lead to the creation of new **Argument** objects if there is **piping**. One interesting side note is that the preprocess method is actually recursive in the event of **piping**!

After the **Argument** has been preprocessed. The **Processor** then checks for **Builtin Commands** or functions from the **UDL**. Before the **Executer** can receive any **Argument** objects, the **Processor**  needs to make sure that there are **Arguments** left to be executed since the **Executer** does not touch **Builtin Commands**. This is because the **Processor** handles all the **Builtin Commands**.

**The Executer Class**

The **Executer** class only one method other than debugging methods and its’ constructor. This single method is execute. The execute method forks the and has the child execute the **Argument** while the parent will either wait or continue depending on whether or not the **Argument** is a **background process**. After all the necessary forking has been completed, the Parent closes all the extra file pointers. Before the child process executes, the child must first take care of any necessary duping. The **Argument** class has a method called handleDuping which automatically will take care of all the duping. The child then closes all the extra file pointers and then executes.

**The Background Process Table Class**

The **Background process table** acts as a way of organizing what **background processes**. When Monty is first started, the setup method creates a **Background prcoess table**. This table is used to keep track of which processes are currently running in the background. Because of the nature of the table, it was easy to implement another method that would keep track of how many processes had been run in the background since the start of the shell. The table is checked each time the shell communicates to the user to see if any **background processes** have been completed since the last check.

**The Command List Class**

The **Command List** acts as a history for Monty. When Monty is initially started, it checks for a command history file, and reads that into the list. This list keeps track of the last 100 commands the used by the user. This table makes all of the builtin history methods possible. When Monty closes, the table is written to a file.

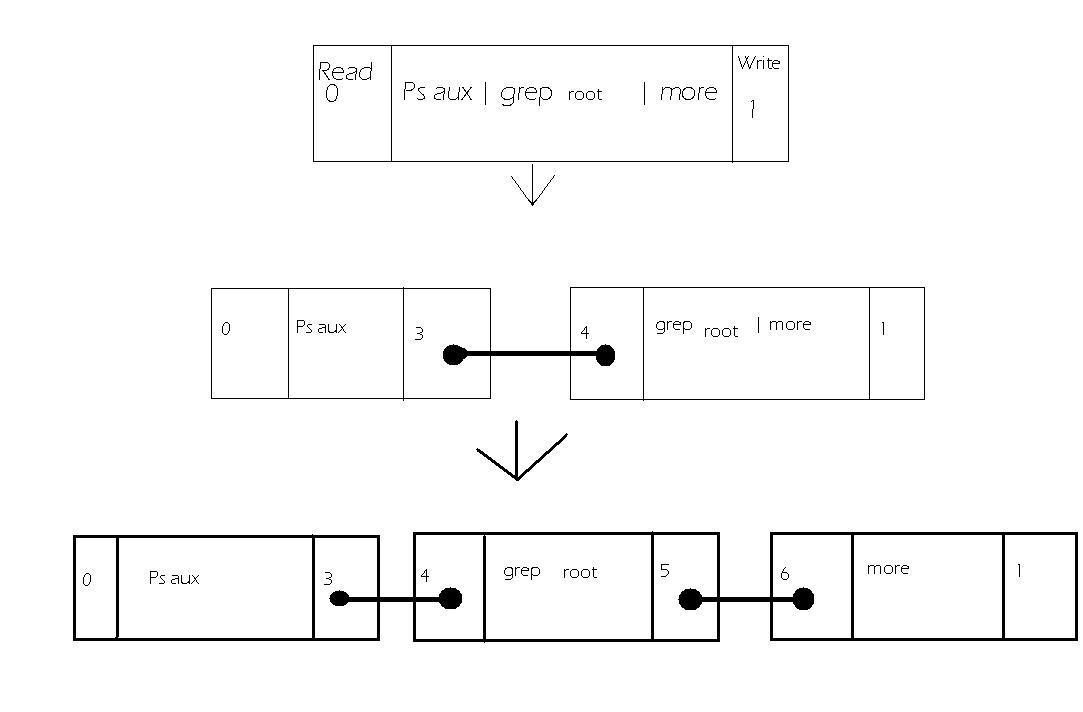
Those are all of the classes that are currently implemented in Monty. Because of the nature of Object oriented code, new classes can be created rather easily. One great feature of this approach is that subclasses can be built that could utilize inheritance. There are more great possibilities in Monty other than just inheritance.

**Piping and Redirection**

Piping and redirection can be a complex subject. Because of this, Monty takes extra care to try to keep it as simple as possible. The design of **Argument** objects is based largely on what would make piping the simplest. **Arguments** have a read end, a write end, and an error end. These ends determine where their data goes to or comes from. The read end represents where on the file table the **Argument** gets its data, the write end represents where on the file table it sends its output, and the error end represents where it will send its error messages. The preprocess class has 10 different symbols for redirection:

* > Redirects stdout, write only.
* >> Redirects stdout, append.
* 2> Redirects stderr, write only.
* 2>> Redirects stderr, append.
* 1>&2 Redirects stdout to stderr.
* 2>&1 Redirects stderr to stdout.
* 1><2, 2><1, 1<>2, 2<>1 Swaps stderr and stdout.
* &> Redirect stderr and stdout, write only.
* &>> Redirect stderr and stdout, append.
* < Redirect stdin.
* | redirect stdout of one **Argument** to stdin of another.
* || Does two different commands regardless of success of the first **Argument**.

Redirection is rather straightforward. Typically, stdout or stderr will be sent to the terminal window or to a file. Piping is a bit more complex. With piping, the output of one command is used as the input for the next. The following example shows how the preprocessor breaks a single **Argument** into multiple ones when its faced with a pipe:



When the preprocessor discovers the pipe, command, it creates a new pipe. This pipe acts as a way for one **Argument** to send its write end data into the read end of another **Argument.** This diagram show very clearly how the piping system works and how the object orientation of **Argument** simplifies the entire process.

**Background Processing**

**Background Processing** is another common feature of shells. It allows a shell to run a command in the background so that the user can do other tasks rather than wait for the command to complete. Monty handles this simply by adjusting the isBackground variable in the **Argument** object. The **Executer** then checks this variable and handles it accordingly.

**User Defined Library and Builtin Commands**

The **UDL** is a personal favorite feature in Monty. If a user were to create his own program and added it to Monty’s library, he could call on that program at any point during Monty’s execution. For example, Monty currently has a program called “About” that will display some basic information about the shell to the user. Commands from the **UDL** handle **piping and redirection** perfectly. There’s also a second category of commands called **Builtin Command**. These are methods that could not be seperated from Monty at this point and had to be integrated directly into the shell. For example, change directory (cd) is not in the UDL or in the Linux Kernel but instead is built directly into Monty. Some, but not all, **Builtin Commands** will handle **piping and redirection** correctly. For example, the jobs command will not currently work with piping, but using one of the history commands will. The following is a listing of all **Builtin Command**:

* cd Change directory.\*
* !x Perform the most recent command that starts with x.
* !!x Perform the command that was performed x times ago.
* exit/quit Exits the shell.
* Jobs displays all currently running background jobs.\*
* jobList Tells the user how many jobs have been executed since the shell started and how many are still being executed.\*

\*Commands that aren’t compatible with redirection

**Known Bugs**

There is only one known bug in Monty. This bug seems to occur when a foreground command is executed after the shell executes a background command with the same location. When the bug activates, Python throws an OSError, and will throw the same OSError for all subsequent commands. Unfortunately, I’ve been unable to determine why this bug is occurring and have been unable to correct it. Once this bug activates, the only way to stop it is to close down the shell and reopen it.

**The Future of Monty**

Monty will be an open source project located on various open source platforms. With any luck, after some further polishing, Monty can be used as a tool to help teach future students or enthusiasts about shell programming. There’s still plenty of improvements that could be made on Monty. For example, the name CommandList is not as clear as a name like History. Changing the class name would be a simple task. It would be nice if Monty could utilize the Curses library or the Xwindow library to give it further functionality such as utilizing the arrow keys or making it graphical. It might be a great idea to great a table superclass and then a BackgroundProcessTable and HistoryTable subclass to further utilize the benefits of object orientation.

**Conclusion**

Monty is a very small and very simple shell at the moment, but it has the potential to one day be so much more. If people are willing to put in the effort, Monty could become a wonderful Linux shell that is not only educational but extremely functional. Perhaps one day, some Linux distributions will be installed with Monty by default. It’s not clear what the future holds, but hopefully this is only the beginning of the Monty Python Shell.