PyShell : Shell Scripting for Python

Paul J. Magnus
Hamilton College

Mark W. Bailey
Hamilton College

Abstract

Shell scripting, specifically in the form of Bash and its related UNIX languages, has grown significantly in both its use and capabilities. Despite this, shell scripting tends to become difficult with the larger and more involved scripts that are being written today. Despite this, shell scripting has still remained relevant due to the ease with which it can be used to serve as a coordination language between a variety of programs through the use of stream redirection and pipes.

PyShell presents an alternative by embedding shell scripts within the syntax of Python. This extends Python to be capable of easily serving as a coordination language using redirection and pipes in much the same way that they are used within shell scripts. Furthermore, PyShell provides a way to create such pipes dynamically at runtime. This paper describes the specifics of the syntax and usage of PyShell. In addition, the fundamental graph structure underlying the creation and execution of shell statements within PyShell is explained.

1 Introduction

Shell scripts such as the Bourne Again Shell (Bash) [2] have become commonplace for many programmers. These languages are powerful tools for interacting with directories, files, and commands in a system. Despite their success, these shell languages become overly complicated when the user is trying to do anything more than write single commands or pipe from one command to another. None other than Dennis M. Ritchie, creator of C and a key developer of Unix, states in his paper "The Evolution of the UNIX Time-Sharing System" that the pipe is "unabashedly linear, though there are situations in which multiple redirected inputs and outputs are called for" [9]. Consider, for example, the statement

```
ls | grep a
```

where the user wants to find all files from their current directory containing the character 'a'. This is quite straightforward due to the pipe, a widely admired contribution of Unix to shell scripting [9]. The `ls` command is capable of producing output through its standard error stream. We might want to filter the result of the standard error from `ls` in addition to its standard output but this is a considerably more complex problem in Bash, shown here.

```
{ { ls 2>&3 | grep a; } 3>&1
  1>&4 | grep bin; } 4>&1 1>&2
```

Redirection using file descriptors, although a powerful tool for those who are well versed in Bash, can be incredibly confusing since it can be challenging to determine what the 'scope' of a specific file descriptor number or what order to place the redirection statements to correctly obtain the desired outcome.

In addition to common confusions regarding file descriptors and redirection, we find that a large variety of operations become forced and unusual within Bash and similar scripting languages. In Bash, all variables are effectively the same type: strings. This made sense when Bash was originally developed because the only things that could be done with the language were execute commands that were passed arguments as strings. Since then, Bash and similar languages have added features such as basic arithmetic but they still use strings as the underlying data type. Consider the Bash statements

```
x=5
y=$(( $x + 3 ))
```

Since arithmetic is not what shell languages are typically used for, they require the use of extra syntax in the form of `$(( ))`. On its own this is not a problem but to add to this, the arithmetic operations only work for integers – floats fail to work in this context. In order to perform float operations or even division that should result in a floating point number, the use of a separate program

called bc which then takes in a string representing the entire arithmetic operation. For example, when calculating division we might write

```
z=$( bc -l <<< "${x}/${y}" )
```

rather than the simple statement

```
z = x / y
```

that we would expect from many other languages like Python.

Small problems like this tend to be uncommon within command line statements and short scripts but when writing longer programs in Bash, it becomes much more likely that arithmetic will be used more often. In fact, these longer scripts are more common than one might expect. On the college's server there are 10,856 shell scripts that we had access to with a total line count of 3,913,366 non-blank lines. That means that on average, there are about 360 lines of code per shell script. This suggests that these scripts are being used for larger problems than they were originally intended to handle. With the growth of computation power, programmers are using scripting languages of all kinds for more applications [7] so these scripting languages should be capable of working effectively for a larger set of applications.

Scripting languages such as Python [1] are much more suited to working with computation on the scale seen in larger scripts. Modern scripting languages like Python have gained popularity due to the ease of rapid development and the terse, readable code made possible by thier syntax [5]. Among a variety of standard scripting languages, Python has been found to be efficient in terms of execution time and memory usage [8]. This is one of the reasons that Python has become one of the leading scripting languages within the industry. Furthermore, Python can easily make use of a variety of types without the programmer having to worry about denoting the specific type of each variable allowing for the ease of scripting that Bash offers while also maintaining the safety that types provide. Using these advantages of Python we can perform complex arithmetic operations that would be nearly unthinkable in Bash. Despite this, Python has trouble with cleanly representing execution of external programs in the way that shell scripting languages tend to do most of their computation. Use of packages such as `os` and `subprocess` can accomplish these operations but they become significantly more complex than Bash when use of an external program is required, especially if redirection and piping is used.

PyShell offers an alternative to Bash by integrating common Bash syntax and practices into Python. This way the programmer can use Python for almost all computation including arithmetic and control flow structures while only needing to use the new Bash-like syntax when the execution of external commands is needed. Within PyShell, all standard Python code remains the same as it would normally and only sections of code that need to deal with shell scripts are affected by the addition of new syntax.

We discuss the syntax of PyShell and how the shell statements are integrated into traditional Python code in Section 2. This includes brief examples and possible uses for the new syntax. Section 2.6 and Section 2.7 discuss two new features of PyShell that do not exist in Bash but instead make use of Python variables to easily construct different kinds of pipes between external commands. In Section 3, we expand on a generic graph structure that is used for the execution model of scripts within the PyShell language. Finally, Section 4 focuses on dynamic construction of PyShell scripts at run-time.

2 Language

PyShell is an extension of the Python language. This means that all Python programs are valid PyShell programs with no changes required. PyShell works by translating a PyShell program into a Python program and then executing the generated Python program as usual. All pieces of the code that are regular Python remain unchanged by the translation process. PyShell statements are identified by dollar signs ($) which appear on both ends of the statement to signal the beginning and end of the PyShell statement. Since the $ character is unused in current Python syntax, all dollar signs that are not a part of a comment or string will be interpreted as delimiters for the PyShell language.

2.1 Simple Commands

Stating a command in PyShell is similar to Bash. The statement

```
$ls$
```

runs the `ls` command within PyShell. Note that the use of dollar signs ($) for delimiters denote a command so that `ls` is not confused with a Python variable.

Commands can have arguments just like any normal Bash program by separating the arguments with whitespace characters. For example,

```
$ls -l /usr/bin$
```

executes `ls` while passing `-l` and `/usr/bin` as arguments.

2.2 Quoting

Command names and arguments may be quoted when the use of whitespace characters or other special

metacharacters is required. Quoting works by using either single quotes (') or double quotes (") as is the case with basic strings in Python. For example, both of the statements

```
$ls './folder name'$
```

and

```
$ls "./folder name"$
```

execute the command `ls` with the single argument `./folder name`. Unlike in Bash, shell expansion does not exist for either type of string.

2.3 Variables

Sometimes, we want to use Python variables from the current Python scope within PyShell script statements. Since command names and arguments require no decoration to make writing common statements simple, variables require the use of an at symbol (@) before the variable name. There cannot be any characters between @ and the variable name. For example,

```
x = '-l'
$ls @x$
```

replaces `@x` with the value of the variable `x` at run time resulting in the above code having the same effect as the statement

```
$ls -l$
```

Variables can be used effectively within loops or where the value of a command or argument is not known when the code is being written. For example,

```
for cmd in ['ls', 'w', 'last']:
    $@cmd$
```

executes each of the three commands (`ls`, `w`, `last`) in sequence.

2.4 File Redirection

Files may be used within PyShell for the standard input, standard output, and standard error file streams resulting from a command.

2.4.1 File Input

The standard input to a command may come from a file by using <. For example,

```
$wc < filename$
```

takes the file named `filename` as the source of standard input for the command `wc`. In this case, `wc` outputs the number of characters, words, and lines found in `filename`. The < and `filename` are not passed as arguments to the command.

2.4.2 File Output

Most commands produce output to the standard output stream. Instead, this output may be sent to a file using >. For example,

```
$ls -l > filename$
```

writes the result of `ls -l` to the file named `filename`. If the file does not exist yet, then the file is created. This writing overwrites any previous data in the file. The > and `filename` are not passed as arguments to the command but instead parsed separately.

Some commands also produce output in the form of error. We can redirect the error output to a file by using `!>` instead of >. For example,

```
$find !> errorfile$
```

results in all of the errors from the find command being sent to the file named `errorfile`. The standard output of `find` is unaffected.

We use `&>` if both standard output and standard error from a command are redirected to the same file. The statement

```
$find &> filename$
```

sends the standard output and standard error of `find` to `filename`.

In order to send the standard output and standard error of a single command to different places, the following syntax is used.

```
$find > (outfile, errfile)$
```

The standard output of `find` is redirected to `outfile` and the standard error is redirected to `errfile`.

2.4.3 Combining Input and Output

When a command handles input and output with respect to files, the input redirection always precedes the output redirection. For example,

```
$wc -l < inputFile > outputFile$
```

results in `wc` taking its input from `inputFile` and sending its output to `outputFile`. On the other hand, the statement

```
$wc -l > outputFile < inputFile$
```

results in a syntax error.

2.4.4 Quoting and Variables

Any non-quoted redirection symbols are interpreted as redirection regardless of the surrounding characters. This means that the redirection operators can be used without

padding whitespace characters and if these character sequences are needed within a string it must be quoted as shown here.

```
$ls 'file>name'$
```

Quoted strings and variables may be used for the file names involved in file redirection.

```
$ls > 'ls>output'$
```

This statement sends the output of the command `ls` to the file named `ls>output`.

Variables may also be used with file redirection. If we save the name of a file into a variable we can then use that instead of explicitly stating the name in the shell statement as shown.

```
f = 'filename'
$ls > @f$
```

We can also use an open file object for redirecting input and output as long as the file is opened for reading or writing respectively. Reading from a file can be written as

```
f = open('filename', 'r')
$wc -l < @f$
```

Similarly, using a file for writing we can use

```
f = open('filename', 'w')
$ls > @f$
```

or

```
f = open('filename', 'a')
$ls > @f$
```

if we want to append to the file instead of overwriting the file.

2.5 Pipelines

"One of the most widely admired contributions of Unix to the culture of operating systems and command languages is the pipe."

– Dennis M. Ritchie [9]

Given the popularity and ease of use of the pipe within Bash, we implement pipes within PyShell in the same form that it appears within Bash.

The pipe links the standard output of one command to the standard input of another command through the use of `|`. For example,

```
$ls | grep test$
```

links the standard output of the command `ls` to the standard input of the command `grep` with `test` being passed as an argument to the command `grep`. The standard output of `grep` is then printed to the terminal as normal.

Similar to file output redirection, pipe comes with a few special kinds of pipes for pipelines involving the standard error of a command. For example,

```
$ls !| grep Permission$
```

pipes the standard error from `ls` into the standard input of `grep` so that only those errors that contain `Permission` are printed. We call this the *error pipe*.

Similarly, the statement

```
$ls &| grep test$
```

pipes both the standard output and standard error from `ls` into the standard input of `grep` so that only the output containing `test` is printed. We call this the *joined pipe*.

For pipes sending the standard output and standard error to separate commands we combine them similarly to file output redirection. For example,

```
$ls | (grep test,
       grep Permission)$
```

pipes the standard output of `ls` into `grep test` while the standard error of `find` is piped into the input of `grep Permission`. We call this the *split pipe*.

Pipelines may contain more than one pipe. The statement

```
$ls | grep test | wc -l$
```

prints the number of file names in the current directory that contain the string `test`. We can also nest pipes when using the split pipe as shown.

```
$ls | (grep test | wc -l,
       grep 'Permission')$
```

This results in printing out the number of files found that contain the string `test` and also printing out all errors that contain the word `Permission`. Since pipes are run in parallel, the order of these split outputs cannot be predicted.

2.6 Incomplete Pipes

In all of the previous examples, we write a statement in PyShell and the program executes the statement immediately. This is done because the entire sequence of commands is known when the process is created. In the rare instance where this is not the case, PyShell includes a syntax for an incomplete pipe, where one end of the pipe is known and the other end will be linked up at a later time. This syntax follows exactly the same pattern as a regular pipe except that the programmer uses a new variable instead of a command. For example,

```
$ls | @p$
```

denotes an incomplete pipe labeled by the variable p. To differentiate this from a regular pipe, the variable p is either undefined or a previously created incomplete pipe. To complete the pipe we can write the statement

```
$@p | grep test$
```

resulting in the same complete structure as the statement

```
$ls | grep test$
```

Since the structure is now complete, the whole statement is executed as normal. The construction of these generic structures will be discussed in more detail in Section 3.

Note that once it is created, p is a variable in Python and it can therefore be used just like any other variable. This means that the programmer can pass incomplete pipes between functions through the use of parameters and return values.

```
def build_start():
    $ls | @p$
    return p

def f():
    p = build_start()
    $@p | grep a$
    p = build_start()
    $@p | grep b$
```

This code is the same as

```
def f():
    $ls | grep a$
    $ls | grep b$
```

except that we can reuse the code within the first function throughout the PyShell file just like any function. Observe that we have to call `build_start` twice within our code since the two incomplete pipes are distinct.

2.7 Join

Consider the pipe statement

```
$ls &| grep test$
```

This statement is a bit different from the other pipe statements because rather than redirecting each of the outputs from the command to separate places it redirects both outputs to the same place, the standard input of `grep` in this case. This means that we are joining the two output streams together which is commonly done in Bash through the use of redirecting file descriptors. Unfortunately, these file descriptors can become confusing to many and the order of the redirection statements can wildly change the result. For example, consider the construction of a split pipe in standard Bash shown here.

```
{ { ls 2>&3 | grep a; } 3>&1
  1>&4 | grep bin; } 4>&1 1>&2
```

This makes use of a variety of file descriptors. If we swap `3>&1` and `1>&4`, the resulting statement will not send any input to `grep bin` which is not what we intended.

Instead, PyShell makes use of incomplete pipes to solve this problem. If we wanted to replace the standard join pipe using incomplete pipes and the join operator we would write

```
$ls | (@r, @s)$
$@r & @s | grep test$
```

The & in the second line serves as the join operator. This means that each of the two incomplete pipes will link their remaining half to the same place, in this case this is the standard input of `grep`.

While replacing the join pipe with separate incomplete pipes is not very useful, we can instead use the join operator to join broken pipes resulting from separate statements.

```
$ls | @s$
$cat filename | @t$
$@s & @t | grep test$
```

This program uses the same `grep` filter on both the standard output of `ls` and the standard output of `cat`. The program will not execute `ls`, `cat`, and `grep` until all pipes are complete. For this reason, `ls`, `cat`, and `grep` all run at the same time, in parallel, so there is no way to predict the order of the results between `ls` and `cat`. A more detailed description of the structure generated by the join operator is discussed in Section 3.1.

2.8 Integration into Python

Regular Python code can interact with PyShell statements in a variety of ways. Programmers may use shell statements as generators of strings where the strings are taken from the standard output of the script, splitting the output on any whitespace characters. One use for this is within `for` statements.

```
for f in $ls$:
    print(f)
```

This results in a loop through the file names in the current directory from the standard output of `ls`. These file names will then be printed using Python. To directly obtain the iterator from a command, use the built in Python function `iter` as shown.

```
files = iter($ls$)
```

PyShell statements can also be used as Booleans based on the exit status of the statement where success results in true and a fialed exit status results in false. This can be

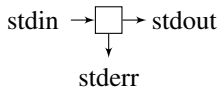


Figure 1: A generic command represented graphically.

used within `if` statements as the Boolean test condition. The program

```
if $test -f filename$:
    print('The file exists')
```

prints `The file exists` only if the `test` command succeeds, in which case the file named `filename` does exist. To directly obtain the Boolean result of the script statement, use the standard Python function, `bool`, as shown.

```
exists = bool($test -f filename$)
```

The return code can also be obtained by using the standard Python functin, `int`.

```
return_code = int($test -f file$)
```

3 Process Graphs

In order to clarify the creation and execution of statements involving pipes and incomplete pipes, discussed in Section 2.5 and Section 2.6 respectively, we provide a general graph structure for representing all processes within the PyShell language. In general, a command has one input, the standard input (stdin), and two outputs, the standard output (stdout) and standard error (stderr). We represent this graphically as shown in Figure 1. The three streams are shown in the directions that they will be used within our further diagrams, stdout being sent out through the right and stderr going out of the bottom of a node. Since stdin is the only input, its position in the figure does not matter. If a stream is not shown in a diagram, then we treat that stream as unhandled resulting in the stream defaulting to the system standard stream of its respective type. The standard streams are typically the input and output of the terminal.

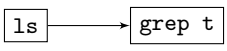


Figure 2: The graph representing

```
$ls | grep t$
```

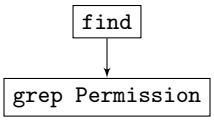


Figure 3: The graph representing the statement

```
$find !| grep Permission$
```

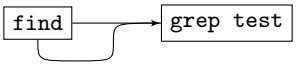


Figure 4: The graph representing the statement

```
$find &| grep test$
```

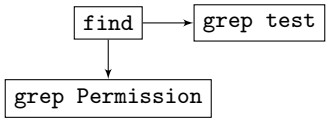


Figure 5: The graph representing the statement

```
$find | (grep test,
         grep Permission)$
```

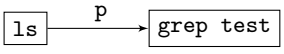
In this formulation, the different forms of pipes are simple to represent. Figures 2 to 5 show each of the basic examples from Section 2.5 in a graphical form. In fact, any directed acyclic graph that uses the generic structure of a process from Figure 1 can be represented using only these pipes. In this way we avoid the limitations of the linear pipe that is present in Bash.

For all such structures, once all pipes are complete, the entire structure executes. Otherwise, PyShell makes use of a form of delayed execution for incomplete pipes. Consider the first example from Section 2.6,

```
$ls | @p$
$@p | grep test$
```

The first statement is represented by a single node with a trailing broken pipe, labeled `p`.

At this point, since there is an incomplete pipe, the process graph cannot be executed yet. Then, we add the second statement and obtain the following graph,



Now, the structure is complete so the program executes the commands within the graph. This means that the `ls`

command from the first statement is not executed until the graph is completed. This is because the PyShell statements are actually creating these generic graph structures rather than just executing the commands within the statements. The program executes the commands in the graph only when all pipes are complete.

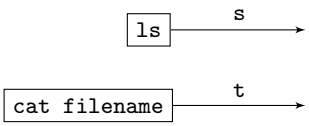
3.1 Graphs with Joins

Consider Figure 4. This represents a joined pipe. As noted in Section 2.7, this pipe is different from the other pipes in that it joins two different pipes together. This is noticeable in the graph by the intersection of two edges in the graph, representing this join.

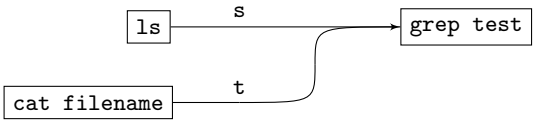
Consider the example from Section 2.7,

```
$ls | @s$
$cat filename | @t$
$@s & @t | grep test$
```

The first two statements result in two separate incomplete pipes.



On the third statement we join these two incomplete pipes and link them to the standard input of `grep`.



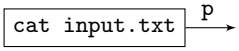
The structure is now complete, so the program executes each part of the graph. The command `grep test` uses the output of both `ls` and `cat` as its input.

4 Dynamic Construction

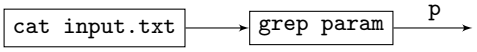
By combining the structures available in PyShell, the programmer can create a variety of structures built dynamically at run-time. We can use this ability to create sequences of pipes. Suppose the programmer wants to use a filter script such as `grep` or `sed` with multiple filter parameters but these parameters are not determined when the program is written. Instead, the list `param_list` is generated dynamically by the program at run-time. This is implemented by the following code.

```
$cat input.txt | @p$
for param in param_list:
    $@p | grep @param | @p$
$@s | cat$
```

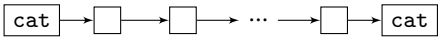
The first statement creates a node and the incomplete pipe `p`.



The first iteration of the loop results in a new node with input from the original pipe `p`. Then, the program labels the output of the same command as `p`.



This program repeats this process. The end result is a structure composed of a sequence of pipes, one for each iteration of the loop.



We can also use other control flow statements to determine the structure of a PyShell graph. For example, the programmer might want to decide whether to apply a specific filter to a previously existing incomplete pipe as shown.

```
if apply_filter:
    $@p | grep t | @p$
```

This program will insert the filter `grep t` into the incomplete pipe `p` if the variable `apply_filter` is true. Use of these and other similar techniques provide the ability to create shell scripts that are dynamic and reusable.

5 Related Work

Several other projects have focused on improvements to the pipe structure. Walker et. al. [10] focuses on achieving many of the same abstractions for pipes within the Bash language. Although they do not gain the advantages of writing code within the context of Python, their project focuses on executing data flow graphs similar to the process graphs described in Section 3. Furthermore, they provide a structure for executing such graphs in parallel on multiple processors efficiently.

The project Dryad [3] similarly makes use of directed acyclic graphs for executing data flow pipelines. The similarities in their graph structure could present possible future prospects for PyShell within the context of multicore systems.

Object-oriented pipes have also been proposed by MacDonald et. al. [6] as a way to improve the speed of standard pipe structures while maintaining the simple construction of pipelines provided by the standard syntax. This has been implemented notably within the Windows PowerShell [4].

6 Future Work and Conclusions

Given the focus on parallel programming across multiple processors and in clusters shown within many of the related projects, the baseline PyShell implementation should be extended to include efficient parallelization techniques. Exploration of the specific techniques employed within Dryad [3] and object pipes [6] may provide improvements compatible with the PyShell language.

PyShell presents a extension to Python allowing for Bash-like shell scripts to be integrated in a variety of contexts within Python. The validity of this proposed syntax has been tested through the implementation of the language within Python making use of a supporting module and translation through the use of the `ply` package which provides access to `lex` and `yacc` within Python.

The ease with which we can combine powerful Python tools with shell scripts results in shorter and simpler scripts. The ability to dynamically create PyShell structures can result in scripts which are more flexible than previous scripts. These scripts can generate the commands and arguments dynamically at run-time which are then connected through pipes to create scripts that are either impossible or too complex to be created within a Bash script.

References

[1] AN, S. U., AND ROSSUM, G. V. Python for unix/c programmers copyright 1993 guido van rossum 1. In *Proc. of the NLUUG najaarsconferentie. Dutch UNIX users group* (1993).

[2] BOURNE, S. R. An introduction to the unix shell. *Bell System Technical Journal 57*, 6 (Oct 1978), 2797–2822.

[3] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev. 41*, 3 (Mar. 2007), 59–72.

[4] JONES, D. Windows powershell: Rethinking the pipeline. *Microsoft TechNet Magazine* (July 2007).

[5] LOUI, R. P. In praise of scripting: Real programming pragmatism. *Computer 41*, 7 (July 2008), 22–26.

[6] MACDONALD, S., SZAFRON, D., AND SCHAEFFER, J. Rethinking the pipeline as object-oriented states with transformations. *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* (2004), 12–21.

[7] OUSTERHOUT, J. K. Scripting: higher level programming for the 21st century. *Computer 31*, 3 (Mar 1998), 23–30.

[8] PRECHELT, L. An empirical comparison of seven programming languages. *Computer 33*, 10 (Oct 2000), 23–29.

[9] RITCHIE, D. M. The evolution of the unix time-sharing system. In *Language Design and Programming Methodology* (Berlin, Heidelberg, 1980), J. M. Tobias, Ed., Springer Berlin Heidelberg, pp. 25–35.

[10] WALKER, E., XU, W., AND CHANDAR, V. Composing and executing parallel data-flow graphs with shell pipes. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science* (New York, NY, USA, 2009), WORKS '09, ACM, pp. 11:1–11:10.