
Augmented Unix Userland Continuation

A Major Qualifying Project, Submitted to the Faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree in Bachelor of Science in Computer Science by

Mark Chaoui _____

12/2/2015

Project Advisor:

Michael Ciaraldi _____

ciaraldi@wpi.edu

This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review. For more information about the projects program at WPI, see <http://www.wpi.edu/Academics/Projects>.

Abstract

This project added some much needed features to PSH, a prototype Python shell based around the passing of objects rather than plain text. The prototype had demonstrated proof of concept but was unpolished. Added features included scripting capability, a solid base of built-in commands, and shell and environment variables. Instructions for use and future development of the shell were also added.

Table of Contents

Abstract.....	i
I. Introduction	1
A. The Python Shell	1
B. Prior Work.....	1
C. Motivation.....	1
1. Shortcomings of today’s shells	1
2. Advantages of the Python Shell	1
D. Project Goals	2
II. Python Shell	3
A. Prior Capabilities	3
B. Added Capabilities	3
III. Instructions for Use.....	4
A. Installing the shell	4
B. Running the shell	4
C. Calling the typical command.....	4
D. Writing a script.....	4
E. Using pipes.....	7
F. Using comments	7
G. Using shell variables.....	8
H. Using environment variables	8
I. Adding commands to the built-ins.....	9
J. Adding formatters.....	10
IV. Architecture	12
A. Directory Structure	12

B.	Data Flow	12
1.	TreeNode	12
2.	Chaining	13
3.	Anatomy of a function	14
C.	Program Files	14
1.	__init__.py	14
2.	console.py	15
3.	run.py	15
4.	core.py	15
D.	Available Commands.....	16
1.	cat.....	16
2.	cd.....	16
3.	echo.....	16
4.	raw	16
5.	resplit	16
6.	reverse	17
7.	setenv.....	17
8.	set.....	17
9.	sort	17
10.	unsetenv.....	17
11.	unset	17
E.	Program Flow.....	18
1.	Flowchart Overview	18
2.	Desugaring	19
3.	Interactive command parsing	19

4.	Script parsing	19
F.	Raw Commands	19
V.	Future Work.....	20
A.	Short term.....	20
B.	Long term	20
VI.	Conclusion.....	21
VII.	References	22

Table of Figures

Figure 1: Sample Script	5
Figure 2: Sample Script Output	6
Figure 3: Sample script with loops	6
Figure 4: Output of looping script	7
Figure 5: cd Command Code	10
Figure 6: Default Formatter Code	11
Figure 7: Chaining with chain() and get_input_generator()	13
Figure 8: Typical Command Algorithm	14
Figure 9: Program Flow	18

I. Introduction

A. The Python Shell

The Python Shell, also known as PSH or the Augmented Unix Userland, is a program that interfaces many programs and utilities of the kernel with each other and the user. It fills the same role as the widely used Bash (Bourne Again Shell), Powershell, or zsh (Z Shell). Currently, it is only specified to work on Unix-like systems, and only heavily tested on Ubuntu.

B. Prior Work

In 2015, Sam Abradi, Ian Naval, and Fredric Silberberg wrote the prototype Python Shell [1] as part of their MQP. Its purpose was to explore the possibility of a shell that passes data between processes with Python objects rather than through plaintext. The prototype could call an extremely limited pool of built-in functions as well as make “raw” calls, which get passed through to the Python interpreter as a system call. I will be referencing the paper that the previous team wrote frequently, addressing them as the first PSH team.

C. Motivation

The last project left a promising skeleton behind with little functionality. The purpose of this MQP is to bring functionality and create further room for growth for the skeleton.

1. Shortcomings of today’s shells

The shells used today are ubiquitous, though as with all things, they can be improved. Their use often requires string manipulation to force the output of one program to look like the input of another. The first PSH team has all this in their Motivation section.

Most current shells only work on select operating systems which require the user to learn different languages for different systems.

2. Advantages of the Python Shell

Most shells have the capability to “pipe” input from one program to another. This means the output of one command is immediately passed to the input of the next. Most of the time, this is done in plaintext. PSH has the unique advantage of piping arguments as objects rather than as

plaintext. Raw text has no semantic meaning until humans read it or it is parsed. When you encapsulate the plaintext in objects with meta information you can bypass the need for the extra step of extracting semantic information from the text. The reader can find more information about this advantage by reading the Motivation section of the first PSH Team.

Another advantage of PSH is that it is written in Python. Python is a versatile and easy to learn language. Anybody with a working knowledge of Python can write scripts, commands, and formatters for the shell. One of the biggest advantages of Python is that it works with most of the operating systems currently in use.

PSH has a “raw” command that allows the user to call the programs specific to the operating system. This means that even if users need a function that does not exist they can use a raw command instead of writing a PSH one. As more PSH commands become standard the need for raw will be phased out. Heavy use of raw may lead to cross platform nightmares but its existence will greatly speed adoption.

PSH uses formatters to simplify printing information. Instead of using complex regexes a user only needs to specify a formatter. A formatter is a program that will render the results in a human readable format.

D. Project Goals

The primary goal of this project was to add scripting capability to PSH. Users can write a script that incorporates any PSH functions along with Python, hence automating simple tasks. No shell is complete without scripting capabilities.

The secondary goal of this project was to add shell and environment variables to PSH to allow called programs to have access to such variables.

The tertiary goal of this project was to develop more built-ins, specialized in the use of passing objects rather than plaintext.

The final goal was to add instructions for the use of PSH for the end users and future developers.

II. Python Shell

A. Prior Capabilities

The Python shell was designed to explore the possibility of programs passing objects to each other. The first design included the capability to write and run functions, tab-complete file names, use history recall, and pipe output.

B. Added Capabilities

PSH was missing scripts when the first group completed their work. No shell is complete without scripting capability, so code was added to PSH to allow end users to create scripts.

PSH was also missing shell and environment variables. So code was also added to support shell and environment variables.

There was no error system in place, and oftentimes any error by the user would result in a Python traceback. A simple error messaging function was added and the user will be less likely to see Python's error messages and more likely to see PSH specific errors. It is still the responsibility of the programmers of the built-ins to ensure the errors are handled properly.

III. Instructions for Use

A. Installing the shell

Accompanying this report is a zip file. Unzip this file, and navigate to the resulting directory *psh*. In this directory, run the *install* script with the command `./install`. This command requires `sudo` and installs for all users by putting files in the current directory and `/usr/local/sbin`. After the installation, do not change the path of the *psh* directory or its subdirectories.

B. Running the shell

To run the shell in interactive mode, use the executable named `psh`. You can also invoke it through Python using the command `python3 -m psh.run`, so long as you are in the `.` To run scripts set them as executable and execute them. See section D on writing a script for more details.

C. Calling the typical command

Calling a command in PSH is rather simple. Simply type the command name followed by the argument, as follows:

```
>>> command_name argument_list
```

This works in both interactive and script mode. See the following example for calling the `echo` command.

```
>>> echo Hello, World!
```

```
Hello, World!
```

D. Writing a script

To run a script, set it as executable and simply execute it in an environment that respects shebangs. Alternatively, you can pipe your script into the following command `python3 -m psh.run -s`. Below is a sample script.

```
#!/usr/local/sbin/pshs
import sys
Echo(['Hello World']).chain(Printer()).call()
echo Python_Path:
print(sys.path)
A = '5'
#Print the value of A. Don't forget to chain a formatter!
Echo([A]).chain(Printer()).call()
raw echo 3 1 4 reversed
reverse 3 1 4
```

Figure 1: Sample Script

Line 1: `#!/usr/local/sbin/pshs`

It signifies to the system that whatever shell was already running that this script should be processed by PSH. Please note that the program called is not psh, but pshs. pshs runs PSH in script mode, which is required for non-interactive execution.

Line 2: `import sys`

It imports the sys Python package.

Line 3: `Echo(['Hello World']).chain(Printer()).call()`

It is mostly there to prove that the user can call desugared versions of each PSH command. Note that this is generally inefficient and should be avoided. Echo is capitalized here because it is the name of the Echo class behind the scenes. For more information on desugaring, see section IV.E.2. An example of how echo would be called by the average user is shown on the next line.

Line 4: `echo Python_Path`

It is a basic PSH echo command. It works right in line with no special delimiter required. Note that echo's registered name is lowercase.

Line 5: `print(sys.path)`

It prints the system Python path proving that the import sys call both worked and persisted.

Line 6: `A = '5'`

It sets the value of A to the string constant 5.

Line 7: `#Print the value of A. Don't forget to chain a formatter!`

This is a comment. It serves the purpose of making the code more readable. It does not print in the output.

Line 8: `Echo([A]).chain(Printer()).call()`

It shows regular Python variables can be used in PSH expressions.

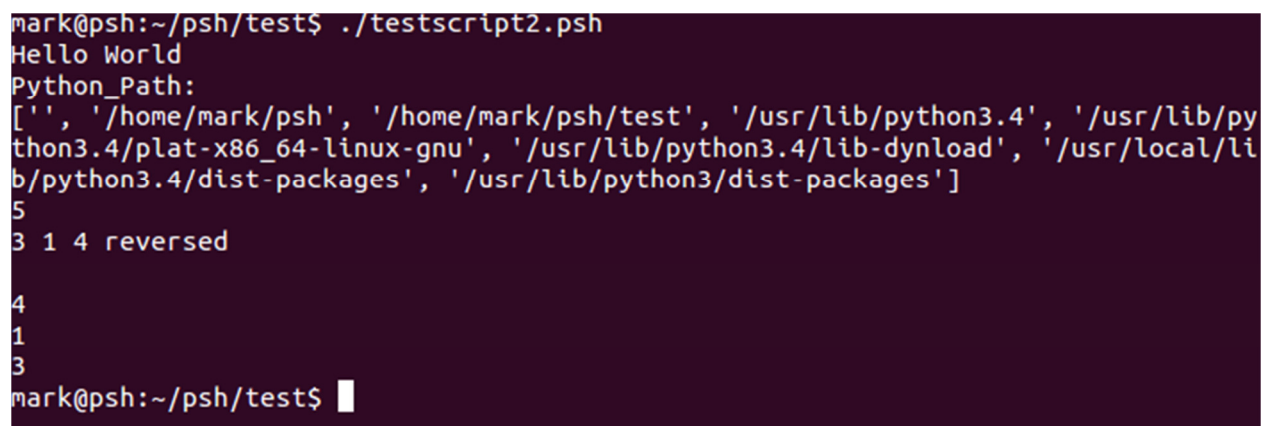
Line 9: `raw echo 3 1 4 reversed`

It shows how the user can run a raw command in script mode.

Line 10: `Reverse 3 1 4`

This is a PSH reverse command.

Below is the screen shot of the above script's output:



```
mark@psh:~/psh/test$ ./testscript2.psh
Hello World
Python_Path:
['', '/home/mark/psh', '/home/mark/psh/test', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-x86_64-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']
5
3 1 4 reversed

4
1
3
mark@psh:~/psh/test$
```

Figure 2: Sample Script Output

In Figure 3, a script which makes use of python control structures such as loops is demonstrated, and in Figure 4 the output is displayed.

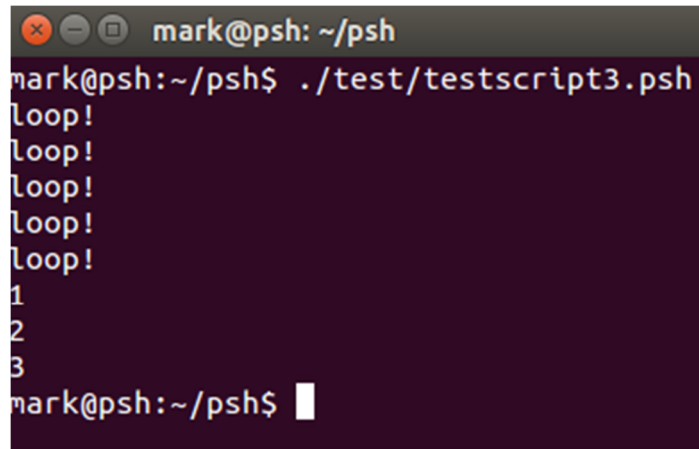
```
#!/usr/local/sbin/pshs

for i in range(0,5):
    echo loop!

sky = "blue"
if(sky == "blue"):
    sort 3 1 2
else:
    echo drip drop
```

Figure 3: Sample Script with Loops

In this example, a PSH command is nested in a Python *for* loop. The echo command is therefore run 5 times. Nested in a Python *if* is a sort PSH command, which is then run because the condition is true. The PSH command in the else statement is skipped as expected.



```
mark@psh: ~/psh
mark@psh:~/psh$ ./test/testscript3.psh
loop!
loop!
loop!
loop!
loop!
1
2
3
mark@psh:~/psh$
```

Figure 4: Output of Looping Script

E. Using pipes

Pipes behave identically to BASH. Put a pipe (|) between two commands and it will automatically feed the output of the first command into the second.

```
command1 | command2 | ... | commandN
```

So the following command will first sort the list then reverse it. Note that the default formatter prints each object on its own line as is reflected below.

```
>>>sort 6 3 5 | reverse
6
5
3
```

F. Using comments

Any line beginning with a # will be ignored. So put comments after a #, as follows:

```
>>>#I'm hiding!
```

G. Using shell variables

Use the set command to set shell variables and unset command to unset them, as follows:

```
>>>set varname value
```

```
>>>unset varname
```

To recall a shell or environment variable use

```
$(varname)
```

If a user were to use the command

```
>>>set USERNAME foo
```

the following command would output foo.

```
>>>echo $(USERNAME)
```

```
foo
```

If a shell variable and an environment variable have the same name, the value of the environment variable will supersede the value of the shell variable. The shell variable will retain its value, but will not be accessible. If the environment variable is later deleted the shell variable with the same name becomes visible again.

H. Using environment variables

Environment variables work in a similar fashion to the shell variables, except spawned processes can access them as well. The commands for setting and unsetting an environment variable are

```
>>>setenv varname value
```

```
>>>unsetenv varname
```

If the following variable is defined

```
>>>setenv USERNAME foo
```

The following command would output foo

```
>>>echo $(USERNAME)
```

```
foo
```

I. Adding commands to the built-ins

Ambitious users versed in Python can write their own functions if they follow a formula. The basic flow can be found in Figure 8.

Each command should be placed in its own file in the *commands* directory. Also, each command must be referenced in `__init__.py` in the *commands* directory and in `console.py`. In `__init__.py` the following command should be added

```
from filename import *
```

In `console.py` the class must be imported

```
from psh.commands import classname
```

Below is pseudo code of the way a command is structured.

```
from psh.commands import BaseCommand, register_cmd
from psh.tree import TreeNode

@register_cmd("cmd_name")
class class_name (BaseCommand):
    def __init__(self, args=[]):
        #run BaseCommand constructor
        super(class_name, self).__init__()
        self.args = args
        any other custom initialization

    def call(self, *args, **kwargs):
        def output_generator:
            function code
            for(condition):
                yield(TreeNode(output))
        return output_generator
```

In the beginning, the programmer has to be sure to list the imports.

The programmer then uses a decorator that attaches the name of the function that will be called by the user to the name of the class.

Then the programmer declares a class inheriting from BaseCommand. This class is required to override BaseCommand's call () method. This method returns a function that is to be called each time the command is invoked. Optionally, the programmer may override the __init__ function which gets called when an object is constructed.

In the call function, the programmer must return a function. This is why in the pseudo code, the output_generator is defined and later returned.

The programmer stores the output of the built-in in TreeNode objects. When the function output_generator is eventually called, all yielded TreeNodes will be output.

Below is the implementation of cd in PSH. This command changes the working directory of the user.

```
from psh.commands.core import BaseCommand, register_cmd
import os;

from psh.tree import TreeNode

@register_cmd("cd")
class cd(BaseCommand):
    """Simple command that just changes directories to the specified path."""

    def __init__(self, args=[]):
        super(cd, self).__init__()
        self.args = args

    def call(self, *args, **kwargs):
        def output_generator():
            if(self.args[0][0] == '/'):
                os.chdir(self.args[0])
            else:
                os.chdir(os.path.join(os.path.abspath(os.path.curdir),self.args[0]))
            yield TreeNode(b'')
        return output_generator
```

Figure 5: cd Command Code

J. Adding formatters

Adding formatters is identical to adding built-ins. The only difference is the outcome. A formatter is designed to output the contents of anything that is piped into it and returns void whereas the built-ins output nothing but return a list. The formatter in Figure 6 takes each object it was passed and outputs it on a separate line.


```
class Printer(BaseCommand):
    """Simple formatter which accepts any object from the input
    generator and simply prints it as if it were a string."""

    def call(self):
        input_generator = self.get_input_generator()
        for node in input_generator:
            line = node.data
            print(str(line.decode('utf-8')))
        return None
```

Figure 6: Default Formatter Code

IV. Architecture

A. Directory Structure

PSH has two nested directories, which can be placed anywhere in the system as long as they don't move after installation. The topmost directory named *psh* has all of the core functionality and the install script whereas the nested *commands* directory holds all built-ins and the BaseCommand superclass. When PSH installs, it puts a few scripts in `/usr/local/sbin`. This places the main executables in a location already in the user's path. There are a few testing modules in the directory under *psh* called *test*. Those modules exercise certain features of PSH to prove they function.

B. Data Flow

1. TreeNode

TreeNodes are the meta objects that store any data to be passed from function to function. Every built-in is capable of returning these and piping them. As the name suggests, each TreeNode is a node in a tree structure and can have many children but one parent. Each node also contains a data field which can be filled with any object or primitive. It is up to the built-ins to handle the data object.

2. Formatters

Formatters are the final step in every PSH chain. They print the results of the commands run. If there is no formatter specified, PSH will automatically add the default formatter which prints each object as a string on a separate line. In general formatters should not change the result. They should simply display it. Currently, only the default formatter is written, though instructions for writing more can be found in section III.J.

3. Chaining

Chaining sequences piped commands. When the command at the end of the sequence needs input from the command before it, the function `get_input_generator()` is called in the call method of a function. It first checks to see whether there is a command before it. If there is not, it returns an empty list. If there is, it calls that function's call method which continues the cycle until there is not a previous command. Figure 7 depicts this process.

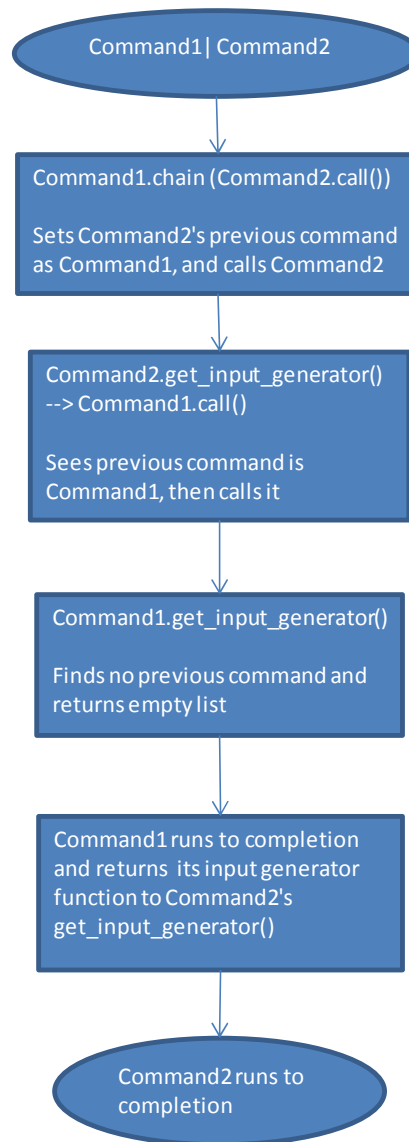


Figure 7: Chaining with `chain()` and `get_input_generator()`

4. Anatomy of a function

Figure 8 describes the way `TreeNode`s propagate through a function. Each function retrieves the output `TreeNode`s from the last command, calling that command in the process, defines a function to return its output using arguments and the input from the last function, and returns that function.

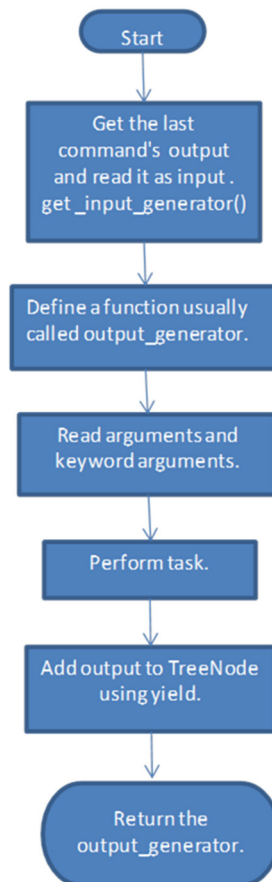


Figure 8: Typical Command Algorithm

C. Program Files

1. `__init__.py`

There are two files with this name, one of which is in the `commands` directory and the other is in `psh` directory. The `__init__` in `commands` simply imports everything from each file that belongs to the built-in commands. The `__init__` at the top level sets up a global variable with all built-in command names.

2. console.py

It is the core of the program. It parses input and calls the proper functions based on that input. It is effectively a read eval print loop or repl. console.py also handles the shell and environment variables.

3. run.py

This is main. It determines whether to run the program interactively or in script mode, then calls either the script interpreter or the repl.

4. core.py

The most important item in core.py is the class BaseCommand. Examples of how to subclass this can be found in section III.I. It is the base class that all commands inherit. The five methods that BaseCommand provides are:

a) __init__

This is the initialization phase of the program. This method is called when the object gets created. This function may be overridden in the subclass.

b) call

This is the method responsible for processing the input and getting an output. This function may take an arbitrary list of arguments and keyword arguments and returns a function that when called populates a list of TreeNodes. This function must be overridden in the subclass.

c) estream

This is a simple error function used to print to the users their errors. It is not to be used for errors fatal to PSH.

d) get_input generator

This function drives the chaining. It is called in every call method once. This function must be called somewhere in the overridden call function. More in-depth information can be found in section IV.B.2.

e) chain

This function sets itself as the previous command and returns the following one (passed in as an argument). This function should never be called by the user.

5. `psh`

This is a bash script which adds the directory PSH is in to the Python path and calls the Python command to start PSH in interactive mode. This is the script which the user will use most often to start the shell. No arguments are required for normal use.

6. `pshs`

This is a bash script placed in the user's path in `/usr/local/sbin` that runs PSH in script mode. This script should not be called directly. It should be invoked using a shebang at the top of a PSH script.

D. Available Commands

1. `cat`

This command works as it does in bash. It concatenates any number of files, including just one.

2. `cd`

This command works as it does in bash. It changes directories, recognizing whether you use an absolute or relative path. Use without an argument to go to your home directory.

3. `echo`

This simple command works as it does in bash. It prints input to screen.

4. `raw`

This command runs the input as a system command. If there is a bash command a user wants a PSH equivalent of, and PSH doesn't have one, a raw command will just call the bash version. It can be invoked explicitly by name, but if the user is in interactive mode, then any time a command is run that doesn't match a built-in, it will be attempted as raw.

5. `resplit`

This command takes delineated objects, concatenates them, then splits them up according to new parameters. This command may not seem that important or useful but its utility in PSH is nearly boundless, especially when dealing with commands that return plain text such as all raw

commands. For example, `resplit` can take a plain text comma separated list and put each entry in its own object.

Besides the input to be `resplit`, this function takes two optional arguments. The separator, `-s`, is the character on which all arguments and inputs will be split. The other argument, `-o`, is the old separator. This character will be placed on the boundary between objects before the splitting occurs. Both the old separator and the separator default to space. Below is an example:

```
>>> resplit 3 53 24 2 1 4 44 -s2 -o
353
4
1444
```

6. `reverse`

This command reverses the list of objects it is given.

7. `setenv`

This command sets environment variables. This function takes the name of the variable and optionally a value.

8. `set`

This command sets shell variables. Like `setenv`, it takes the name of the variable and optionally a value.

9. `sort`

This command sorts arguments and piped string input alphabetically.

10. `unsetenv`

This command unsets environment variables. It takes the name of the variable to unset.

11. `unset`

This command unsets shell variables. Like `unsetenv`, it takes the name of the variable to unset.

E. Program Flow

1. Flowchart Overview

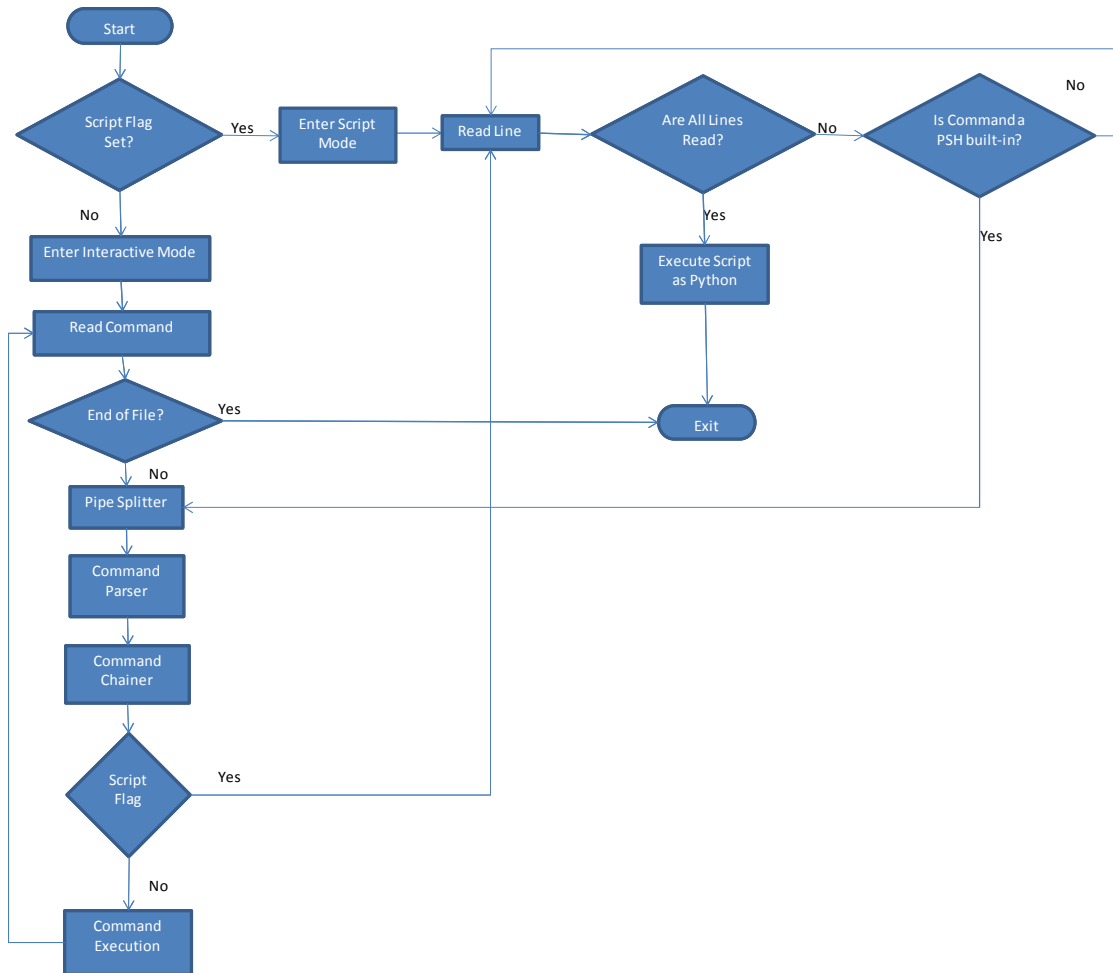


Figure 9: Program Flow

When the program is run, it uses an argument to determine whether it is running a script or running interactively. This argument is hidden behind two macros, one which runs whenever PSH is started with the PSH executable and the other whenever a PSH script is invoked. There exists a third mode for debugging purposes which only executes scripts as pure Python. This mode does not have a macro though it can be accessed by the command `python3 -m psh.run -p`. You must pipe your script to this command.

2. Desugaring

In both modes of operation, desugaring, the process of taking the PSH commands and changing them into executable Python, is a major procedure. In the program flow (Figure 9) above, the desugaring process begins with the pipe splitter and ends with the command chainer. Before the process a default formatter is added to the end of the pipeline if there is none specified. Then, each command name in a pipeline is replaced with the name of the command's python class. Its arguments are next placed between parentheses. The first function calls chain on the next in the pipeline, which calls chain on the command after that until the formatter is reached. When the formatter is reached, its call method (.call()) is called.

It is important to note that if there is no formatter at the end, PSH will chain a default one and attach a call method at the end.

3. Interactive command parsing

When in interactive mode, PSH engages a read eval print loop where it reads the user's input, chains and desugars it, evaluates the resulting Python, prints the output, and reads more input. This loop continues until the end of file character (EOF) is encountered.

4. Script parsing

When PSH is run in script mode, it does a pre-processing run on the entire script where it desugars PSH commands to their equivalent Python, including shell and environment variables. It then runs the Python function eval on the whole script.

F. Raw Commands

The raw commands have two purposes, to give users confidence that they can use commands they know and to cover up for any incompleteness of PSH. This tool is designed to help with adoption but as it greatly limits cross-platform capabilities, it should be used more and more sparingly as PSH gets more and more default commands. Currently, any input piped into raw commands must be in a string embedded in a TreeNode, and the entire output of the raw will be stored in one string in one TreeNode.

V. Future Work

A. Short term

The next improvement that should be made targets the formatters. Currently, there is no way for a program to define its default formatter, requiring the user to specify the “correct” formatter for each program. If BaseCommand had a function that allowed each built-in to specify a formatter, it would allow the correct formatter to be tacked on during desugaring. Also, the user should be able to use formatters between commands, marking them as “passthrough”, meaning they print their arguments and pass them on to the next program rather than exiting.

After that, more built-ins need to be developed and optimized for passing objects. This project added several new built-ins, but they are only the beginning, designed more as examples of what’s possible than fully functional programs.

Currently, each built-in is run in the same process as the main loop and subsequently with each other. The processes are not isolated at all, and they need to be separated to ensure that they don’t step over each other’s boundaries. There is more on this issue in the first PSH team’s paper.

Finally, the built-ins require being acknowledged in two separate source files in different directories, one in `__init__.py` and the other in `console.py` (described further in the section on how to write a built-in, section III.I). It would be nice if the programmer only had to make a built-in and put it in the right directory for it to be detected when the shell starts.

B. Long term

One of the largest advantages of using Python is that it is available on a wide variety of platforms. As it stands, the program is only tested on Unix-like systems such as Linux, though with more development and testing it could be a viable option for other platforms, namely OSX and Windows. In the long term, one of the biggest selling points of PSH is its portability.

The tab completion used now does not complete the names of built-ins or arguments, only file names. Currently, a native Python tab completion function is used, though building one from the ground up specifically for PSH may eventually prove necessary.

Users should be able to store a set of instructions for the shell’s startup in a configuration file. This file should be a PSH script that runs automatically upon PSH’s startup, similar to the `.bashrc` file for bash.

VI. Conclusion

This project provided additional core functionality to PSH to allow it to grow. Adding shell and environment variables, the capability to run scripts, and an installation script created a much more solid foundation for development. The instructions for use and development, built-ins that exercise the shell's power, and the future work section allow later groups to further develop the shell and make it an appealing alternative to existing shells.

VII. References

- [1] Abradi, Sam; Naval, Ian; Silberberg, Fredric (2015). Augmented Unix Userland. Major Qualifying Project, Worcester Polytechnic Institute.