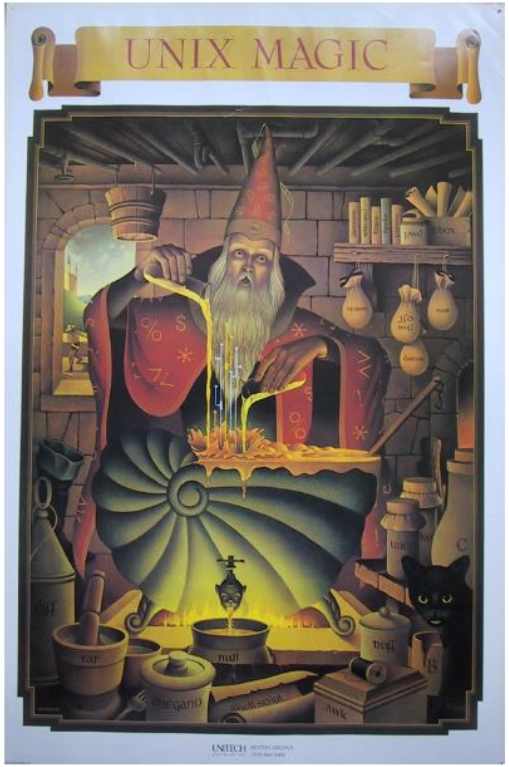
DONDERS INSTITUTE

Introduction to bash

Mariam Zabihi

April 15, 2019

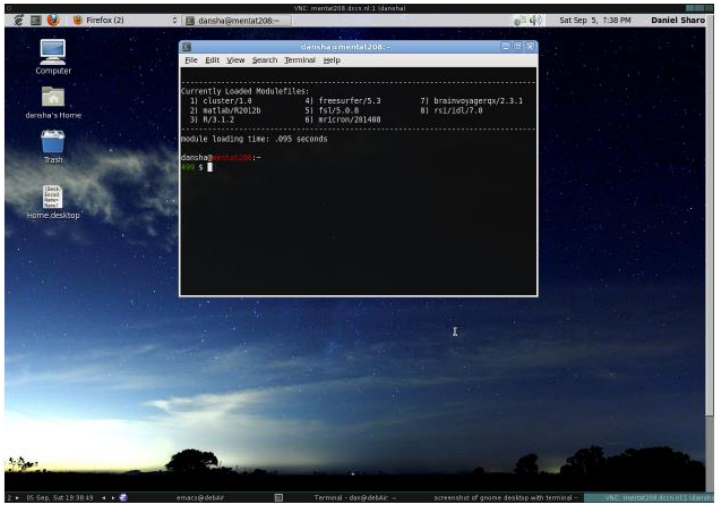


Overview

- Fun, topical, historical background
- Variable Assignment
- Spaces, quotes, escaping characters
- bash *order of operations*
- Parameter Expansion
- Scripting Basics
- The PATH variable
- Command Substitution & Arithmetic Operations
- for-loops and if-statements
- local v. global (environmental) variables

Command Line

- bash is a program that is running in your terminal emulator

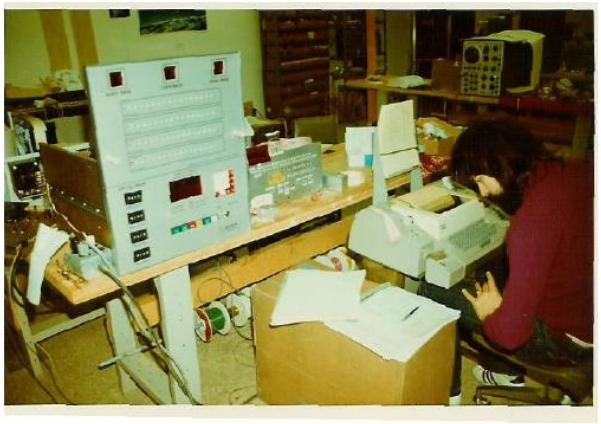


Gnome Terminal running in X

- The black window isn't bash, its a program that emulates the functionality of a teletypewriter plugged into your computer

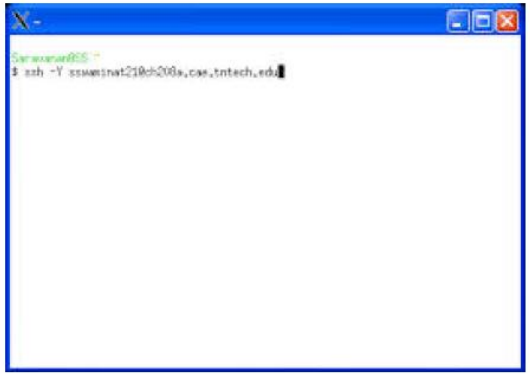
A real teletypewriter! Wow!

Evolution of the Interface



TeleTYpewriters

VT-100 Terminal



Xterm! OG X terminal

These devices are more or less analogous in functionality

What is bash?

bash is a Command Line Interpreter

- bash says, "give me input from the terminal, and I will process the input, tell the kernel what to do, and send output somewhere, usually the term."
  - The term is usually the terminal emulator you're running bash in.

What is bash?

bash is a Command Line Interpreter

- bash says, "give me input from the terminal, and I will process the input, tell the kernel what to do, and send output somewhere, usually the term."
  - The term is usually the terminal emulator you're running bash in.

bash is a Language

- It has variables
- Flow control
- Syntax

bash interprets commands, interacts with the kernel and directs output

Absolutely essential knowledge about bash

White space

- bash treats spaces, tabs and new lines as white space
- White Space separates *words* in bash
- The first word is a command, the following words are arguments

Absolutely essential knowledge about bash

White space

- bash treats spaces, tabs and new lines as white space
- White Space separates *words* in bash
- The first word is a command, the following words are arguments

ls command with arguments

- $ ls -a /path/to/my/favorite/file*txt
  - command SPACE option SPACE operand
- Many commands have similar syntax
- options are usually specified as *-x*
- Read man pages for more documentation
- Bad idea to run a command without knowing its syntax

Essential bash: Variable Assignment

Variables

- Variables are assigned as variable=value
- Not variable = value
- referenced as $variable
- user variables should be lower case

```
#code demo
$ foo=bar
$ echo $foo
bar
```

Essential bash: Variable Assignment

Variables

- Variables are assigned as variable=value
- Not variable = value
- referenced as $variable
- user variables should be lower case

```
#code demo
$ foo=bar
$ echo $foo
bar
```

Note

- In var = val, bash would parse *var* as a command and = *value* as arguments to the command
- Variables can be unset with the *unset* command.
- If a variable is set twice, the original contents are overwritten.

Essential bash: Escaping Special Characters

Quotes

- Anything in quotes will be processed as a single word
- All white space characters will be *escaped*
- White space, or any character, that is not processed by the shell (escaped) is said to be *literal* as opposed to *syntactic*

Essential bash: Escaping Special Characters

Quotes

- Anything in quotes will be processed as a single word
- All white space characters will be *escaped*
- White space, or any character, that is not processed by the shell (escaped) is said to be *literal* as opposed to *syntactic*

Compare

```
$ variable=1 2
bash: 2: command not found
#syntactic space is bad
```

with

```
$ variable="1 2"
$ echo $variable
1 2
#literal space gives desired
#result
```

Essential bash: Escaping Special Characters

Quotes

- Anything in quotes will be processed as a single word
- All white space characters will be *escaped*
- White space, or any character, that is not processed by the shell (escaped) is said to be *literal* as opposed to *syntactic*

Compare

```
$ variable=1 2
bash: 2: command not found
#syntactic space is bad
```

with

```
$ variable="1 2"
$ echo $variable
1 2
#literal space gives desired
#result
```

the \ escape character

- Use the \ character to stop bash from processing only the next char
- In variable=1\ 2, bash will process the space literally.
- In this case variable=1\ 2 has the same effect as variable="1 2"

Escaping Specific Types of Characters

Single Quotes vs. Double Quotes

- Single quotes remove special meaning from characters inside them, i.e everything between single quotes will be processed as a literal character
- Double quotes only escape spaces and single quotes, i.e bash will interpret all characters except spaces and single quotes
- Use the \ character inside double quotes to escape specific characters.

Escaping Specific Types of Characters

Single Quotes vs. Double Quotes

- Single quotes remove special meaning from characters inside them, i.e everything between single quotes will be processed as a literal character
- Double quotes only escape spaces and single quotes, i.e bash will interpret all characters except spaces and single quotes
- Use the \ character inside double quotes to escape specific characters.

Double Quotes

```
$ variable="1 2"
$ echo $variable
1 2
$ echo "$variable"
1 2
$ echo "'$variable'"
'1 2'
$ echo "\$variable"
$variable
```

Escaping Specific Types of Characters

Single Quotes vs. Double Quotes

- Single quotes remove special meaning from characters inside them, i.e everything between single quotes will be processed as a literal character
- Double quotes only escape spaces and single quotes, i.e bash will interpret all characters except spaces and single quotes
- Use the \ character inside double quotes to escape specific characters.

Double Quotes

```
$ variable="1 2"
$ echo $variable
1 2
$ echo "$variable"
1 2
$ echo "'$variable'"
'1 2'
$ echo "\$variable"
$variable
```

Single Quotes

```
$ variable='1 2'
$ echo $variable
1 2
$ echo '$variable'
$variable
$ echo '"$variable"'
"$variable"
```

bash order of operations

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

How Does bash Process Commands?

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

1. Read in everything on one line before continuing parsing

How Does bash Process Commands?

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

1. Read in everything on one line before continuing parsing

Process quotes

- Right after bash reads a line, it triggers a quoted state for everything in quotes
- In case you've forgotten:
  - bash 'reads everything in here literally'
  - bash "reads only spaces and ' literally"

How Does bash Process Commands?

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

1. Read in everything on one line before continuing parsing

Process quotes

- Right after bash reads a line, it triggers a quoted state for everything in quotes
- In case you've forgotten:
  - bash 'reads everything in here literally'
  - bash "reads only spaces and ' literally"
- if we used '$Book' instead of "$Book", would this command work?

How Does bash Process Commands?

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

1. Read in everything on one line before continuing
2. Process quotes

How Does bash Process Commands?

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

1. Read in everything on one line before continuing
2. Process quotes

Process Special Operators

- These include < > » « | {} * ? []
  - In other words, it interprets redirects and globbing characters and parses brackets

How Does bash Process Commands?

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

1. Read in everything on one line before continuing
2. Process quotes
3. Process Special Operators: These include < > » « | {} * ? []

How Does bash Process Commands?

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

1. Read in everything on one line before continuing
2. Process quotes
3. Process Special Operators: These include < > » « | {} * ? []

Parameter Expansions

- A parameter is an entity that stores values and is referenced by a name, a number or a special symbol.
- Variables are parameters, so now bash expands them
- The ~ is a parameter, so now ~ is expanded to `/home/YOU/`

How Does bash Process Commands?

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

1. Read in everything on one line before continuing
2. Process quotes
3. Process Special Operators: These include < > » « | {} * ? []

Parameter Expansions

- A parameter is an entity that stores values and is referenced by a name, a number or a special symbol.
- Variables are parameters, so now bash expands them
- The ~ is a parameter, so now ~ is expanded to `/home/YOU/`
- Other examples include:
  - Command Substitution of the form *$(COMMAND)*
    - Use the output of a command as an argument
  - Arithmetic expansion of the form *$((2 + 2))*
    - Use the arithmetic result as an argument

How Does bash Process Commands?

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

1. Read in everything on one line before continuing
2. Process quotes
3. Process Special Operators: These include < > » « | {} * ? []
4. Perform Expansions

How Does bash Process Commands?

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

1. Read in everything on one line before continuing
2. Process quotes
3. Process Special Operators: These include < > » « | {} * ? []
4. Perform Expansions

Split Commands into Command names and arguments

- Analyze all the white space left
- treat the first word of every line or command-argument block as a command, and the other words as arguments. Then...

How Does bash Process Commands?

Example Code

```
$ book="Harry Potter and the Half-Blood Prince"
$ cat ~/Documents/"$book" > /media/myUSB
```

1. Read in everything on one line before continuing
2. Process quotes
3. Process Special Operators: These include < > » « | {} * ? []
4. Perform Expansions
5. Split Commands into Command names and arguments
6. Send for execution

Scripting

- What is a script?
  - Simply commands in a text file

Scripting

- What is a script?
  - Simply commands in a text file
- Why bother?
  - Reduce monotony
  - Reduce error
  - Create tools, analysis pipeline, etc.

Scripting

- What is a script?
  - Simply commands in a text file
- Why bother?
  - Reduce monotony
  - Reduce error
  - Create tools, analysis pipeline, etc.
- How do I make a script?
  - Open your favorite text editor, create a text file filled with bash commands, then save.

Scripting

- What is a script?
  - Simply commands in a text file
- Why bother?
  - Reduce monotony
  - Reduce error
  - Create tools, analysis pipeline, etc.
- How do I make a script?
  - Open your favorite text editor, create a text file filled with bash commands, then save.
- How do I use it?
  - A few ways, but usually you just set it as executable with a special command

```
$ chmod +x myscript.sh
```

Scripting

- What is a script?
  - Simply commands in a text file
- Why bother?
  - Reduce monotony
  - Reduce error
  - Create tools, analysis pipeline, etc.
- How do I make a script?
  - Open your favorite text editor, create a text file filled with bash commands, then save.
- How do I use it?
  - A few ways, but usually you just set it as executable with a special command

```
$ chmod +x myscript.sh
```

- And then run it by typing it's name at the shell and hitting enter

```
$ /path/to/myscript.sh
```

Example of a script

```
#!/bin/bash
# A sample bash script

cat logfile.txt | grep "Subject[0-9][0-9]"
```

Example of a script

```
#!/bin/bash
# A sample bash script

cat logfile.txt | grep "Subject[0-9][0-9]"
```

- If this script was saved as /home/language/dansha/Documents/myscript.sh, we would now set it as executable with:

```
$ chmod +x ~/Documents/myscript.sh
```

Example of a script

```
#!/bin/bash
# A sample bash script

cat logfile.txt | grep "Subject[0-9][0-9]"
```

- If this script was saved as /home/language/dansha/Documents/myscript.sh, we would now set it as executable with:

```
$ chmod +x ~/Documents/myscript.sh
```

- And run it by typing the following at the shell prompt:

```
$ ~/Documents/myscript.sh
```

PATH

- In bash when you type a command, it's roughly analogous to double clicking on an icon
- bash must know what the string you type (command) refers to
- Bash will search along something called the path to find a string that matches what you entered
- The path is a list of directories
- If a program is not on the path, bash can't find it.
- path is stored as a variable. Test it out with *echo $PATH*

PATH

- In bash when you type a command, it's roughly analogous to double clicking on an icon
- bash must know what the string you type (command) refers to
- Bash will search along something called the path to find a string that matches what you entered
- The path is a list of directories
- If a program is not on the path, bash can't find it.
- path is stored as a variable. Test it out with *echo $PATH*

```
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/abin:/usr/local/bin
```

- notice the uppercase!

PATH

- If a program isn't in your path, you can tell bash where it is by entering the absolute path, the full path from the root directory

```
$ myprogram
bash: myprogram: command not found
$ /usr/local/bin/myprogram
running myprogram...
```

PATH

- You could also enter the relative path, the path relative to your current working directory.

Example

- You're current directory is `/home/language/dansha`
- The program *myprogram* is in /home/language/dansha/Programs
- To run *myprogram* via the *relative path* enter the command

```
$ Programs/myprogram
running myprogram...
```

Changing the PATH

- Notice the strange form with the : character as a delimiter

```
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/abin:/usr/local/bin
```

Changing the PATH

- Notice the strange form with the : character as a delimiter

```
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/abin:/usr/local/bin
```

- Because the path is simply a variable, you can change it as you would any other variable
- Remember though that setting new contents overwrites old contents

```
running the command
$ PATH=/path/to/myprogram
will change the PATH variable so that the only directory
bash searches is /path/to/myprogram
```

Changing the PATH

- Notice the strange form with the : character as a delimiter

```
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/abin:/usr/local/bin
```

- Because the path is simply a variable, you can change it as you would any other variable
- Remember though that setting new contents overwrites old contents

```
running the command
$ PATH=/path/to/myprogram
will change the PATH variable so that the only directory
bash searches is /path/to/myprogram
```

- The proper way to add an item to the path is include the old path in the new path

```
running the command
PATH=$PATH:/path/to/myprogram
changes the PATH variable so that /path/to/myprogram is ADDED.
```

What we have learned so far?

What is BASH:

white space, command syntax, Variable Assignment, single quote and double quote

Process of Commands in BASH:

Read all on one line -> quotes -> Special Operators -> Expansions ->Command and arguments -> execution

Bash Scripting:

- What's and why bash script
- How to make bash script: Chmod +x myscript.sh
- How to run one : /path/to/myscript.sh

PATH:

How to change and add path: PATH=$PATH:/path/to/myprogram

bash Programming Constructs

- We will focus on the for-loop and the if-statement
- We will also teach you about command substitution and arithmetic calculations in bash
  - These are actually treated as parameters in bash, like variables, but they are commonly used in bash scripting.

Command Substitution

- Evaluate a command and capture the output.
- The output can be set to a variable or used as input in a command.
- Syntax is $(COMMAND)

```
$ var=$(echo hi)
$ echo $var
hi
```

for-loop

Implements iteration in bash

- For every item in a list execute a list of commands. When there are no more items left, the for-loop is finished.
- List is a space delimited list of strings

for-loop

Implements iteration in bash

- For every item in a list execute a list of commands. When there are no more items left, the for-loop is finished.
- List is a space delimited list of strings

Syntax is of the form:

```
for item in list; do
    COMMAND
    HOWEVER MANY COMMANDS YOU WANT
done
```

- The commands and item list are arbitrary
  - They can consist of whatever and however many commands and items you want.

for-loop

Example 1

- Simple list of 4 numbers

```
for item in 1 2 3 4; do
    STUFF
done
```

Example 2

- List of strings and numbers

```
for item in asda 12 ds ss za 3sj; do
    STUFF
done
```

for-loop

- At the start of each iteration, bash creates a variable
  - Value of variable is current item in the list
  - Items in list can be incorporated into commands in the loop

Incorporating list items into commands

```
for item in 1 2 3 4; do
    echo $item
done
```

```
1
2
3
4
```

Final Notes on for-loops

Variable names

- You can make the variable name anything you want!

```
for whatever in a b 1 2; do
    echo $whatever
done
a
b
1
2
```

Final Notes on for-loops

Variable names

- You can make the variable name anything you want!

```
for whatever in a b 1 2; do
    echo $whatever
done
a
b
1
2
```

- More on this in the exercise

Quoting the list

- Using quotes around the list will escape the spaces

```
for i in "a b 1 2"; do
    echo $i
done
a b 1 2
```

if-statement

- The purpose of if is to test if a command returns an exit status of 0 (zero) or not 0, and then run some commands if the exit staus is 0. You can also say to run commands if the exit status is not 0. This is what the keyword *else* means.

if-statement

- The purpose of if is to test if a command returns an exit status of 0 (zero) or not 0, and then run some commands if the exit staus is 0. You can also say to run commands if the exit status is not 0. This is what the keyword *else* means.
- The if-statement syntax is:

```
if command-returns-true; then
    run these commands
else
    run-these-commands-instead
fi
```

if-statement

- Generally, *if* is used in conjunction with a command called test to test if certain conditions are true.
  - All test does is exit with 0 if a condition is true or nonzero if a condition is not true.

if-statement

- Generally, *if* is used in conjunction with a command called test to test if certain conditions are true.
  - All test does is exit with 0 if a condition is true or nonzero if a condition is not true.
- *test* is a complex command with the ability to test many comparisons.

if-statement

- Generally, *if* is used in conjunction with a command called test to test if certain conditions are true.
  - All test does is exit with 0 if a condition is true or nonzero if a condition is not true.
- *test* is a complex command with the ability to test many comparisons.
- In bash, there is a special syntax for using *test* that integrates it with the shell syntax
  - Allows for better readibility plus a few bonus features.

```
if [[ 45 -lt 50  ]]; then
     echo "condition was true"
else
    echo "condition was false"
fi
```

if-statement

Variable names can be used in if-statements

```
$ a=45
$ b=50
if [[ $a -lt $b  ]]; then
     echo "condition was true"
else
    echo "condition was false"
fi
```

if-statement

Variable names can be used in if-statements

```
$ a=45
$ b=50
if [[ $a -lt $b  ]]; then
     echo "condition was true"
else
    echo "condition was false"
fi
```

Difference between strings and integers

- In bash, we generally don't need to specify if variables contain strings or integers.
- Some commands, like *test*, distinguish between strings and integers.
  - Make sure you use the correct operators for the comparison you want.

if-statement

Table: Nonexhaustive list of *test* operators

Integer Operators	Definitions
-lt	less than
-le	less than or equal
-gt	greater than
-ge	greater than or equal
-eq	equal
-ne	not equal
String Operators	
==	equal
!=	not equal

More extensive treatment of if-statements and test here and in the exercise

Using BC

- If you need to work with floats, you can pass arguments to the program bc using echo and a pipe:

```
$ echo "2.6884 * 43.223" | bc
116.2007
```

- You could of course save this output to a variable with command substitution

```
$ var=$(echo "2.6884 * 43.223" | bc)
```

Arithmetic Operations

- bash is not so good at doing arithmetic.
  - Only works with whole numbers without calling external programs, i.e the calculator program bc

Arithmetic Operations

- bash is not so good at doing arithmetic.
  - Only works with whole numbers without calling external programs, i.e the calculator program bc
- For simple calculations in bash, the prefered method is to use the `$(( n OPERATOR m ))` syntax

```
$ var=$(( 2 + 6 ))
$ echo $var
8
```

Arithmetic Operations

- bash is not so good at doing arithmetic.
  - Only works with whole numbers without calling external programs, i.e the calculator program bc
- For simple calculations in bash, the prefered method is to use the `$(( n OPERATOR m ))` syntax

```
$ var=$(( 2 + 6 ))
$ echo $var
8
```

- You could think of this as a form of command substitution. As in CS, bash will expand the arithmetic operation in the parens to it's outcome before further evaluation.

Global and Local Variables

- When you run a script you start a new instance of bash, and that is what executes the commands in the script.

Global and Local Variables

- When you run a script you start a new instance of bash, and that is what executes the commands in the script.
- Local variables are variables set in your local environment.
  - They do not get passed to new bash processes.
  - This works the other way too. Variables from the subshell are not automatically brought to the parent shell.

Global and Local Variables

- When you run a script you start a new instance of bash, and that is what executes the commands in the script.
- Local variables are variables set in your local environment.
  - They do not get passed to new bash processes.
  - This works the other way too. Variables from the subshell are not automatically brought to the parent shell.
- To set variables so they will be inhereted by child processes, use the export command
  - Variables available to subprocesses, those created with *export*, are known as global variables
  - environment variable is a synonym for *global variable*.
  - Must be done if a script or an external program needs certain variables set to function properly

Example of Global Variables

Freesurfer

- Freesurfer wants `FUNCTIONALS_DIR` defined
- If the variable is set locally, then it won't be available to Freesurfer
- use the *export* command to set global variables

Example of Global Variables

Freesurfer

- Freesurfer wants `FUNCTIONALS_DIR` defined
- If the variable is set locally, then it won't be available to Freesurfer
- use the *export* command to set global variables

Setting a Global Variable

```
$ FUNCTIONALS_DIR=/path/to/my/epis
will result in a variable being set that is only available to the
current shell
$ export FUNCTIONALS_DIR=/path/to/my/epis
will be available to Freesurfer.
```

Automatically Set Variables When bash Starts

- You could set each local or global variable you need at the start of each bash session.
- Alternatively, you can tell bash to run a series of commands when it starts up

Automatically Set Variables When bash Starts

- You could set each local or global variable you need at the start of each bash session.
- Alternatively, you can tell bash to run a series of commands when it starts up
- this is done with a file called *bashrc*

Automatically Set Variables When bash Starts

- You could set each local or global variable you need at the start of each bash session.
- Alternatively, you can tell bash to run a series of commands when it starts up
- this is done with a file called *bashrc*

Using the bashrc

- bashrc contains commands to be run at the start of every bash session
- bashrc is a text file, so it can be edited with any text editor.
- each user has, or could have their bashrc in their home directory.
  - A system wide .bashrc exists as well, usually as /etc/bashrc or similar. This is read first

Some Points about bashrc

- It is usually a hidden file in the users home directory. In UNIX, hidden files are prefixed with the . character
  - Your personal bashrc should be called `/home/yourgroup/yourname/.bashrc`

Some Points about bashrc

- It is usually a hidden file in the users home directory. In UNIX, hidden files are prefixed with the . character
  - Your personal bashrc should be called `/home/yourgroup/yourname/.bashrc`
- bashrc is not a script, its a series of commands that are *sourced* when a new instance of bash is started
  - *sourcing* is like cutting and pasting every line in the .bashrc into your shell and hitting enter

Some Points about bashrc

- It is usually a hidden file in the users home directory. In UNIX, hidden files are prefixed with the . character
  - Your personal bashrc should be called `/home/yourgroup/yourname/.bashrc`
- bashrc is not a script, its a series of commands that are *sourced* when a new instance of bash is started
  - *sourcing* is like cutting and pasting every line in the .bashrc into your shell and hitting enter
- Changes made to .bashrc will not take effect until the file is read
  - i.e restart bash or run the command

```
$ . ~/.bashrc
```

Some Points about bashrc

- It is usually a hidden file in the users home directory. In UNIX, hidden files are prefixed with the . character
  - Your personal bashrc should be called `/home/yourgroup/yourname/.bashrc`
- bashrc is not a script, its a series of commands that are *sourced* when a new instance of bash is started
  - *sourcing* is like cutting and pasting every line in the .bashrc into your shell and hitting enter
- Changes made to .bashrc will not take effect until the file is read
  - i.e restart bash or run the command

```
$ . ~/.bashrc
```

- The format of the bashrc should be the same as you would format a series of commands on the terminal.
  - Remember it's just a list of commands you want to run to set variables, etc.

Sample bashrc

```
if [ -f /etc/bashrc ]; then
      . /etc/bashrc   # --> Read /etc/bashrc, if present.
fi
alias matlab='/opt/cluster/bin/matlab2014b -nodesktop'
alias matlab-ide='matlab2014b'
alias qsub-out2QsubLogs='qsub -o /home/language/dansha/QsubLogs/'
#setup GIGANTIC TOOLS to do GOD KNOWS WHAT
FS_FREESURFERENV_NO_OUTPUT=0
source ~/SetUpFreeSurfer.sh
PATH=$PATH:/opt/cluster/bin:/opt/mricron/200912/:~/dusr/dbin
```

Thank You For Your Attention