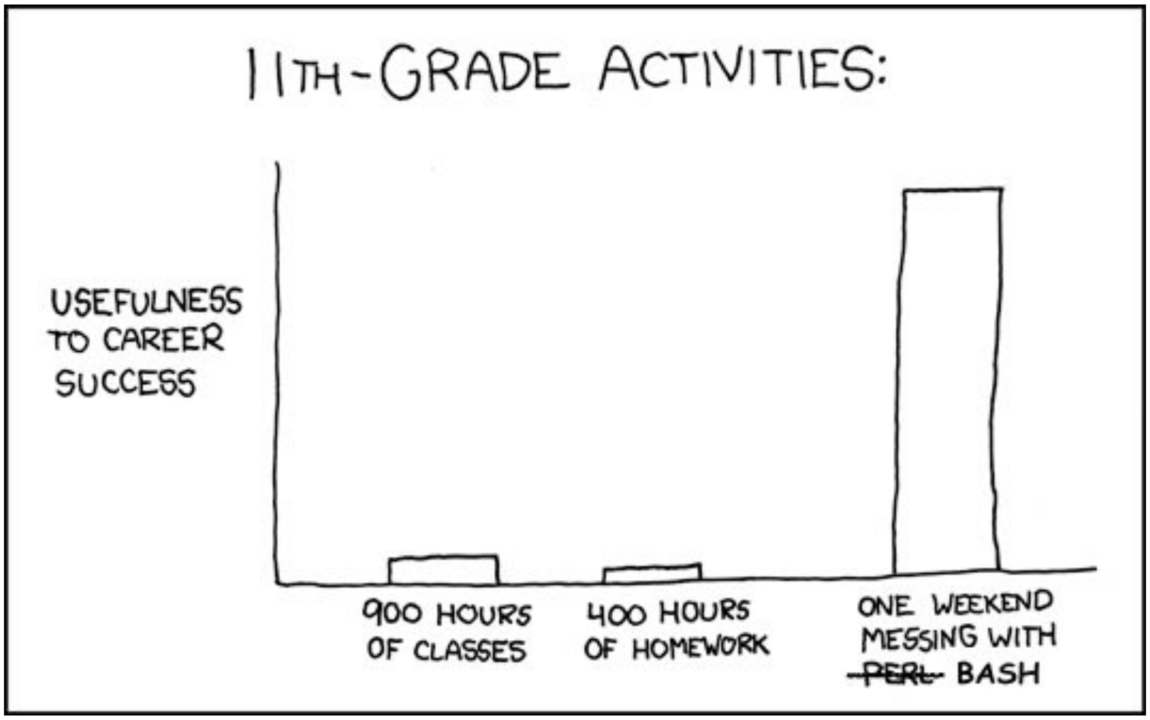
Bash programming

Motivation


11TH-GRADE ACTIVITIES:
USEFULNESS TO CAREER SUCCESS
900 HOURS OF CLASSES
400 HOURS OF HOMEWORK
ONE WEEKEND MESSING WITH ~~PERL~~ BASH

Programming language poll:

- [ ] How many people know some programming language?
- [ ] Which ones?

Hello world

```
$ cat > hello.sh

echo "Hallo Welt."

^D
```

You can check the contents of your file with `cat` or `more`. There are two ways you can “run” or execute this program.

1. **Source** the file. You can source the file with `source hello.sh` Try this you should see:

```
$ source hello.sh
Hallo Welt.
$
```

N.B. you might not see `$` but instead some other prompt. `source` is used so often that there is a shortcut for it the *dot* operator (.):

```
$ . hello.sh
Hallo Welt.
$
```

1. Direct executation. You can change/edit this file to make it an executable script. There are two things you need to do in order to do so.

   i. Edit the file and add the following line as the **first** line of the file.

   ```
   #!/bin/bash

   echo "Hallo Welt."
   ```

   Save the file.

   Now try to exectuate it (run it) by typing:

   ```
   $ ./hello.sh
   ```

   you should see:

   ```
   $ ./hello.sh
   -bash: ./hello.sh: Permission denied
   ```

As a safety/security measure files when created or normally defaulted to be not-*executable*. You make a file executable using the `chmod` command. To do so type:

```
$ chmod u+x hello.sh
```

Now you can run the command directly by typing using `./hello.sh`

```
$ ./hello.sh
Hallo Welt.
$
```

What is the meaning of the `./` why do we need it here but not for `chmod` or any other UNIX command we type?

Variables

Setting vs Getting

Bash different from most other programming langues in that setting a variables has a different syntax from getting the value form a variable. In pretty much ever other language to set you do something like:

```
x = 1
x := 1
x <- 1
```

which will set the value of the variable `x` to 1. To *get* the value of a variable you simple use its name:

```
y=x     # Set y to x
```

```
print x # Print the value of x
f(x)    # Call the function f with value x
```

However Bash is different (most shell languages follow this convention)

```
x = 1   # Set variable x to 1

y = $x  # Set variable y to value of x
echo $x # Print the value of x
f $x    # call function f with value of x
```

Using variables:

Here is hello.sh with a variable for output string:

```
#!/bin/bash
GREETING="Hallo Welt."
echo $GREETING
```

Environment variables

There are a number of preset variables that are create whenever you start a bash shell (command line / terminal). These variable are (mostly) inherited in any bash script you write. They function as *global* variables but the term used in shells is **Environment** variables.

To see the environment variables set for your current shell use the `printenv` or `env` command. You should see something like:

```
$ printenv
TERM_PROGRAM=Apple_Terminal
SHELL=/bin/bash
TERM=xterm-256color
HISTSIZE=
```

> - [ ] Exercise: Figure out what environment variables are different between the command shell and a bash script. Can you explain why there is this differnce.

PATH variable

The `PATH` variable is a colon (:) separate list of directories which the shell will look for when looking for commands. You can see what your *path* is by typeing any of the following

```
$ printenv PATH

$ echo $PATH
```

Here is mine:

```
$ printenv PATH
/Library/Frameworks/Python.framework/Versions/2.7/bin:
/Users/socci/Transporter/Work/Compgen2016/bin:
/Users/socci/bin:/usr/local/bin:/usr/bin:/bin:
/usr/sbin:/sbin:/opt/X11/bin:/Library/TeX/texbin
```

I added line breaks to make it readable you will see one long line. The path is the reason you have to type:

```
./hello.sh
```

To run your script. If type just `hello.sh` (try it) you will see:

```
$ hello.sh
-bash: hello.sh: command not found
```

because your current directory, where the hello.sh script is, is *not* on your path.

> - [ ] Exercise: How would you add the current directory to your path? Are you sure this way is a good idea? What might be a better way? HINT google. By the way make sure your

> path is still intact. Does ls work?

Scope: Inheriting and exporting variables.

Try the following. Remove the `GREETING="Hallo Welt."` from the script:

```
#!/bin/bash
echo $GREETING
```

Run it. You should see nothing because GREETING was never set. Now try the following in the command shell.

```
$ GREETING="Hallo Welt."
$ ./hello.sh
```

and you still see nothing. Why? Environment variable are not inherited between shells by default. To get the `GREETING` variable to be visible to the script you need to `export` it. Try:

```
$ export GREETING="Hallo Welt."
$ ./hello.sh
```

`export` marks a variable as *exportable* and is made available to programs that are sub-shells of the current shell (ie were spawned from it). Without export variables are *private* to that shell.

Control flow: if-then-else

`bash` like most all computer languages has a condition expression to enable branches in control flow. The basic syntax is:

```
if [ $X == "YES" ]; then
    echo "variable X equals YES"
fi
```

The syntax is fairly brittle; many of those spaces are required (the ones in the brackets for sure). Note in bash in this context `=` and `==` (as opposed to many other languages). Also do not bash has this annoying (some say charming) way of doing being and end delimiters: `if` — `fi`. But it is not consistent (for loops `for` – `done`). The most aggrevating is case: `case` — `esac`

```
if [ $X = "YES" ]; then
    echo "variable X equals YES"
fi
```

And for not equal

```
if [ "$X" != "YES" ]; then
    echo "variable X does not equal YES"
fi
```

For more comparison operators see: (http://tldp.org/LDP/abs/html/comparison-ops.html)

There is also an `if-else`:

```
if [ "$X" == "YES" ]; then
    echo "variable X equals YES"
else
    echo "variable X does not equal YES"
fi
```

and finally if, else if is

```
if [ "$X" == "YES" ]; then
    echo "variable X equals YES"
elif [ "$X" == "NO" ]; then
    echo "variable X equal NO"
else
    echo "not sure what X is"
fi
```

Multi-lingual hello script

Now lets make a more internationall script. Create a new script called `polyglot.sh` and do the following:

```
#!/bin/bash

if [ "$LOCALE" == "" ]; then
    GREETING="WARNING: Did you forget to set LOCALE"
elif [ "$LOCALE" == "de" ]; then
    GREETING="Hallo Welt."
elif [ "$LOCALE" == "it" ]; then
    GREETING="Ciao Mondo."
elif [ "$LOCALE" == "en" ]; then
    GREETING="Hello World"
else
    echo "ERROR: Unknown LOCALE==$LOCALE"
    exit 1
fi

echo $GREETING
```

Feel free to add more; you can find some of the 2 character langague codes at: (https://en.wikipedia.org/wiki/ISO_639-1).

Runs the command. How would you set the `$LOCALE` variable? Remember `export`.

In the next section we will look at a better way to specify variables but there is a way that is more convient then doing:

```
$ LOCALE=it
$ ./polyglot.sh
```

You can use the following syntax:

```
LOCALE=de ./polyglot.sh
```

will allow you to set environment variables that are exported to the script. **N.B.**, doing it this way *does not* change the current shells variable:

```
$ export LOCALE=en
$ echo $LOCALE
$ ./polyglot.sh
$ LOCALE=de ./polyglot.sh
$ echo $LOCALE
```

Command line args

There is a more common/convenient way to pass variables to a shell via the command line. Command line arguments get assigned to special variables, called *position* variables: `$1`, `$2`, `$3`, … More special variables:

- `$#` is equal to the number of arguments passed to the script
- `$*` is set to all the arguments passed to a script (also `$@` which is slightly differnt in a way I still do not fully understand)

So we can not modify the `polyglot.sh` script to take its language from the command line.

```
#!/bin/bash

LOCALE=$1
if [ "$LOCALE" == "" ]; then
    GREETING="WARNING: Did you forget to set LOCALE"
elif [ "$LOCALE" == "de" ]; then
    GREETING="Hallo Welt."
elif [ "$LOCALE" == "it" ]; then
    GREETING="Ciao Mondo."
elif [ "$LOCALE" == "en" ]; then
    GREETING="Hello World"
else
    echo "ERROR: Unknown LOCALE==$LOCALE"
    exit 1
fi

echo $GREETING
```

What happens if you call `polyglot.sh` with more than one argument.

Loops: for

Another useful control structure is `for`-loops. In `bash` the variant used is `for-in`:

```
#!/bin/bash

STRINGS="A B C D E"
for s in $STRINGS; do
    echo $s
done
```

will print each string in the variable. Not this auto-spliting of a variable does not work for literals or when it is quoted. Try:

```
#!/bin/bash

STRINGS="A B C D E"
for s in "$STRINGS"; do
    echo $s
done
```

```
#!/bin/bash

for s in "A B C D E"; do
    echo $s
done
```

> ☐ Exercise: Modify the polyglot.sh script to print a greeting for every argument on the command line. Make sure it does something helpfull if no arguments are given.

Sub Shells

Type the following into the terminal:

```
$ date
$ today=date
$ echo $today
```

you do not get what you would like/expect. Spacing is key in bash and can be really difficult to get. Try:

```
$ today= date
$ echo $today
```

This looks like it might have work but it does not. Try:

```
$ today = date
```

Here the extra space makes bash think `today` is a command which for most people is not.

So how do we get what we want. Get the variable `today` assigned with the output of date. There are two special operators for this:

```
$ today=$(date)
$ echo $today
$ today=`date`
$ echo $today
```

The two have slightly different semantics and `$()` is usally prefered.

Resources

Have only scratched the surface. If you are not completely horrified or turned off then read on

- Google: `bash programming tutorials`

  - http://tldp.org/LDP/Bash-Beginners-Guide/html/
  - http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
  - http://tldp.org/LDP/abs/html/
    - http://www.tldp.org/guides.html
  - http://ryanstutorials.net/bash-scripting-tutorial/

Warning

`bash` often require much more disclipine to write *good*: readable, maintainable, re-useable code then other languages. If you have a script with more than a few lines; you should work hard to

1. Comment it well
2. Structure it cleanly

Plug for version control

- Learn and use it. Recomendation `git`

Exercise:

Scan for the location of all programs listed in the file:

- `computationalGenomicsPrograms`

The file `computationalGenomicsPrograms` contains a list of files which are programs you should have in your PATH. Write a script called `scanPaths.sh` that takes one argument which is a file like `computationalGenomicsPrograms` and prints out the path to each of them. It should print an error message if the program is not found on the path.

You know everything you need to to do this with one exception. To get the path to a program in your path you need to use the builtin command `which`:

```
$ which ls
```

```
/bin/ls
$
```

When you are done you should see something like:

```
$ ./scanPaths.sh ../computationalGenomicsPrograms  | head
STAR is in /Users/socci/Transporter/Work/Compgen2016/bin/STAR
askMeAnything is NOT ON PATH!
bedtools is in /Users/socci/Transporter/Work/Compgen2016/bin/bedtools
blast is NOT ON PATH!
bowtie is NOT ON PATH!
bwa is in /Users/socci/Transporter/Work/Compgen2016/bin/bwa
...
```