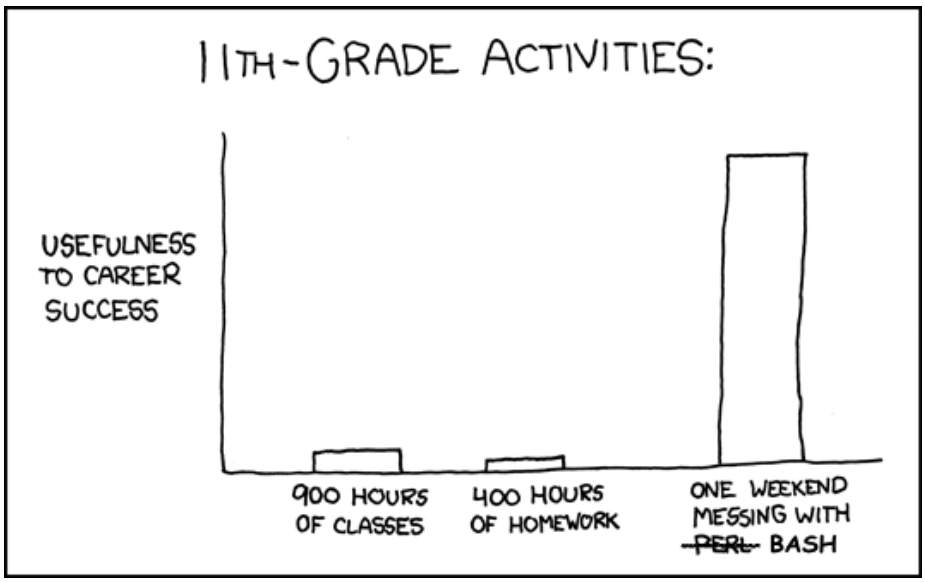
Bash programming

Motivation



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 execution. 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.

   (a) 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 execute 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 difference.

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 typing 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 aggravating 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 international 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 language 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 different 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-splitting 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 usually preferred.

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 disciplin 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

Bash Lab

Useful (key) scripts for rest of workshop

- Configuration script
- Program Wrappers:
  - STAR
  - PICARD

Configuration script

Please **READ/LISTEN CAREFULLY** to the following.

Data paths

First check that the data for the Labs has been loaded correctly. Do the following:

```
$ md5sum /share/data/compgen2016/day45_Intro2Seq_VarCalling/genomes/H.Sapiens/b37_hl/b37_hl
```

and you should see:

```
bc303533c68cf74b6f2c705f3d86398c  /share/data/compgen2016/ /b37_hl.dict
```

where I have dropped part of the long path for clarity.

Now this directory:

- `/share/data/compgen2016/day45_Intro2Seq_VarCalling`

is both long and also going to be used over and over again. There are many ways of dealing with this in UNIX. We are going to create a `config.sh` script that will store and name this and other paths for easy reference and quick loading.

In you `home` directory (`/home/guest`) create a `Day45` sub-directory and then two more sub-sub directories: `code`, `results`

You can do this most simply by:

```
$ cd # This takes you home
$ mkdir Day45
$ mkdir Day45/code
$ mkdir Day45/results
$ cd Day45/code
```

You should now be in the code subdirectory (check with pwd). Now create/edit a file called `config.sh` and in it put:

```
# Compgen2016 Day 4,5 configuration file

# Path to root of lab data directories

ROOT45=/share/data/compgen2016/day45_Intro2Seq_VarCalling
```

Once you have this file you want to `source` it so those variable will be set in your current shell environment. Do the following:

```
$ source ~/Day45/code/config.sh
```

and to make sure everthing is working redo the checksum but this time use the `$ROOT45` variable:

```
$ md5sum $ROOT45/genomes/H.Sapiens/b37_hl/b37_hl.dict
bc303533c68cf74b6f2c705f3d86398c  /share/data/compgen2016/...
```

You will need to redo `source ~/Day45/code/config.sh` (or `. ~/Day45/code/config.sh`) for every new shell you create or when you relogin **OR** whenever you *edit* the

file. This last part is very important. Changing the file does not do anything until you source it.

For now since we will be changing it often just remember to re-`source` it every time you change it or create a new terminal/shell. However you can get it to be sourced every time you create a new shell by adding it to your `.bashrc` or `.profile` file. (If time demonstrate)

Wrappers:

Perhaps one of the most useful thing you can do in `Bash` is *wrap* programs to:

- Give a consistent interface
- Specify default values for certain args
- Create a more convenient command syntax

A great example of a programs begging to be wrapped are STAR and Picard.

> No one perfect/correct way to do this. Many different styles. Pick one that works best for you.

STAR

Compare the default behaviour or `bwa` vs `STAR`. Can we make `STAR` more `bwa` like; *i.e.*, more helpful and easier to use. We can do this by writing a script that wraps the running of STAR.

To start go to your *code* directory: `cd ~/Day45/code` and edit your `config.sh` script to add the following code:

```
# Put our code directory on our PATH
#   [[ ]] operator for regular expressions
#   =~ regEx match
#   ! means not
# [[ ! $X =~ string ]] evaluates to true if
# string is NOT a substring of X
# So if ~/Day45/code not already on PATH
# add it and export it

if [[ ! $PATH =~ Day45/code ]]; then
    PATH=~/Day45/code:$PATH
    export PATH
fi
```

Once you have added this to your `config.sh` source it and check that your code dir is on your `PATH` with `printenv PATH`

Q: What would happen if you left off `export PATH`? Why is it `export PATH` and not `export $PATH`?

Now wrap STAR

Want to address several problems:

- Old versions of STAR (<2.4-ish) had no usage. 2.5+ fixes that, kind of but still hard to read/use.
- STAR writes lots of files and gives them fixed names
- Sane/simple defaults.

Standard STAR command line:

```
STAR \
    --genomeDir $ROOT45/genomes/H.Sapiens/b37_hl/index/star/NoGTF \
    --readFilesIn \
        $ROOT45/Labs/1_Intro2BashScripting/data/testTiny_R1.fastq.gz \
        $ROOT45/Labs/1_Intro2BashScripting/data/testTiny_R2.fastq.gz \
    --readFilesCommand zcat
```

Part of this is not `STAR` but the folders I setup. We can fix this by defining some variables:

```
GENOMESTAR=$ROOT45/genomes/H.Sapiens/b37_hl/index/star/NoGTF
DATADIR=$ROOT45/Labs/1_Intro2BashScripting/data
```

So now the command is a little better

```
STAR \
    --genomeDir $GENOMESTAR \
    --readFilesIn \
        $DATADIR/testTiny_R1.fastq.gz \
        $DATADIR/testTiny_R2.fastq.gz \
    --readFilesCommand zcat
```

But compare this to what we would do with `bwa`

```
bwa mem $GENOMEBWA \
    $DATADIR/testTiny_R1.fastq.gz $DATADIR/testTiny_R2.fastq.gz \
    > out.sam
```

So can we wrap `STAR` to behave more like `bwa` but with some enhancements do not like `bwa` writing to `stdout`, but STAR's output is also kind of crazy.

Again make sure you are in `~/Day45/code` and start a new script called: `wSTAR`

```
#!/bin/bash

# wSTAR; wrapper script for STAR
```

```
if [ "$#" == "0" ]; then
    echo "usage: wSTAR GENOMEDIR FASTQ_R1 FASTQ_R2 OUTDIR"
    exit
fi
```

remember to do `chmod u+x wSTAR` so you will be able to run it and run it:

```
$ wSTAR
usage: wSTAR GENOMEDIR FASTQ_R1 FASTQ_R2 OUTDIR
```

compare to `STAR`'s usage screen. I **strongly** recommend that for any script you will use more than twice or on different days you do a usage "screen".

Now add the rest:

```
#!/bin/bash

# wSTAR; wrapper script for STAR

# Note change here. Want usage if
# we run with zero or if we forget one
# (or add an extra)

if [ "$#" != "4" ]; then
    echo "usage: wSTAR GENOMEDIR FASTQ_R1 FASTQ_R2 OUTDIR"
    exit
fi

mkdir -p $4
cd $4
STAR \
    --genomeDir $1 \
    --readFilesIn $2 $3 \
    --readFilesCommand zcat
```

Now you can run STAR on the same files as before as:

```
./wSTAR $ROOT45/genomes/H.Sapiens/b37_hl/index/star/NoGTF
    $ROOT45/Labs/1_Intro2BashScripting/data/testTiny_R1.fastq.gz \
    $ROOT45/Labs/1_Intro2BashScripting/data/testTiny_R2.fastq.gz \
    OutputTiny
```

Much improved but a number of problems. Try to think how you would improve it further.

PAUSE

Improvement / fixes

```
#!/bin/bash

# wSTAR; wrapper script for STAR

# Note change here. Want usage if
# we run with zero or if we forget one
# (or add an extra)

if [ "$#" != "4" ]; then
    echo "usage: wSTAR GENOMEDIR FASTQ_R1 FASTQ_R2 OUTDIR"
    exit
fi

mkdir -p $4
cd $4
STAR \
    --genomeDir $1 \
    --readFilesIn $2 $3 \
    --readFilesCommand zcat
```

Name arguments

Not necessary but strongly recommended for clarity, self-documentation and greater flexibility.

```
#!/bin/bash

# wSTAR; wrapper script for STAR

# Note change here. Want usage if
# we run with zero or if we forget one
# (or add an extra)

if [ "$#" != "4" ]; then
    echo "usage: wSTAR GENOMEDIR FASTQ_R1 FASTQ_R2 OUTDIR"
    exit
```

```
fi

GENOMEDIR=$1
FASTQ_R1=$2
FASTQ_R2=$3
OUTDIR=$4

mkdir -p $OUTDIR
STAR \
    --genomeDir $GENOMEDIR \
    --readFilesIn $FASTQ_R1 $FASTQ_R2 \
    --readFilesCommand zcat \
    --outFileNamePrefix $OUTDIR/
```

Pre-existing output

What if we re-run our script with the same `OUTDIR`. It will over write the files there which is *maybe* ok. But if the second run has an error then the files there will be the original which could be very confusing. So it is much better in cases like this to do one of the following:

- Warn the files/directory already exists and refuse to run
- Delete the previous results first; probably should warn about this.
- Or have a `force` argument to let the user force case 2

I am going to go with (1) and and exit but again this is a style choice.

```
#!/bin/bash

# wSTAR; wrapper script for STAR

# Note change here. Want usage if
# we run with zero or if we forget one
# (or add an extra)

if [ "$#" != "4" ]; then
    echo "usage: wSTAR GENOMEDIR FASTQ_R1 FASTQ_R2 OUTDIR"
    exit
fi

GENOMEDIR=$1
FASTQ_R1=$2
FASTQ_R2=$3
OUTDIR=$4
```

```
# file test operators
#       -e == exists
# See http://tldp.org/LDP/abs/html/fto.html
# for complete listing

if [ -e $OUTDIR ]; then
    echo
    echo "OUTDIR=[$OUTDIR] already exists; will not over write"
    echo "Choose a new directory or delete OUTDIR to continue"
    echo
    exit
fi

mkdir -p $OUTDIR
cd $OUTDIR
STAR \
    --genomeDir $GENOMEDIR \
    --readFilesIn $FASTQ_R1 $FASTQ_R2 \
    --readFilesCommand zcat
```

Other problems to solve at end:

- ???

Wrap picard (or other jars)

First we need to find them. They should be in

- `/usr/local`

Good first step is to add the JARS to are `config.sh` script

`PICARD=/usr/local/XXX/picard.jar`

This is what my `config.sh` looks like now:

```
# Compgen2016 Day 4,5 configuration file

# Path to root of lab data directories

ROOT45=/share/data/compgen2016/day45_Intro2Seq_VarCalling

# Put our code directory on our PATH

if [[ ! $PATH =~ Day45/code ]]; then
    PATH=~/Day45/code:$PATH
```

```
    export PATH
fi

# JARS

JARDIR=/Users/socci/Desktop/Compgen2016/Work/jars
PICARD=$JARDIR/picard.jar
```

Source your file and then test that it worked by doing.

```
java -jar $PICARD
```

Why wrap?

- Java versions; major headache some programs need different version of java than others. mutect-1.7 needs JAVA7 (1.7) but latest version is JAVA8 (1.8)
- Check you java version `java -version`, find JAVA7
- Java MEMORY arguments; Picard can use a lot of memory. Useful to just specify a large defualt
- Nicer syntax, on path.
- Shortcuts for commonly used modules

Simple/minimal Picard wrapper

```
#!/bin/bash

# Explict choice of java version
# Need to set this for your machine
JAVA=/Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Contents/Home/jre/bin/java

# Path to picard jar
PICARD=/Users/socci/Desktop/Compgen2016/Work/jars/picard.jar

# JAVA VM Size
VMSIZE=4g

$JAVA -Xmx$VMSIZE -jar $PICARD $*
```

So instead of doing:

```
java -jar $PICARD Command Arg1=Val1 Arg2=Val2 ...
```

do

```
picard Command Arg1=Val1 Arg2=Val2 ...
```

Does not seem like a big win but remember we now have a fixed java path so if the sysadmins decided to update it without telling us either it will not break or the script will fail rather than attempt to run with a possible incompatible java version. Also we have set a better memory size default. But can do more. This is my personal `picard` script

```
#!/bin/bash

# Explict choice of java version
# Need to set this for your machine
JAVA=/Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Contents/Home/jre/bin/java

# Path to picard jar
PICARD=/Users/socci/Desktop/Compgen2016/Work/jars/picard.jar

# JAVA VM Size
VMSIZE=4g

# TMPDIR (on many HPC places using /tmp is a bad idea because it is too small)
# so I make one in my home directory
TMPDIR=~/tmp

COMMAND=$1
shift

if [ "$COMMAND" == "" ]; then
    $JAVA -jar $PICARD 2>&1 | less -R
    exit
fi

$JAVA -Xmx$VMSIZE -Djava.io.tmpdir=$TMPDIR \
    -jar $PICARD $COMMAND \
    TMP_DIR=$TMPDIR \
    VALIDATION_STRINGENCY=SILENT \
    $*
```

I have a nicer way of looking at the help screen (it pages), and I also set the TMPDIR explicitly. This is often critical as Picard tmp files can be huge and will over flow many default /tmp installs.

—

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
...
```

Plug for version control

- Learn and use it. Recommendation `git`