Introduction to Bioinformatics Programming

Bioinformatics programming basics, scripting, and functionality

Kristine Lacek, MS
Bioinformatics Scientist
US CDC – Influenza Division

Disclaimer

The findings and conclusions in this presentation are those of the authors and do not necessarily represent the official position of the Centers for Disease Control and Prevention.
Use of trade names and commercial sources is for identification only and does not imply endorsement by the U.S. Department of Health and Human Services.
References to non-CDC sites on the Internet do not constitute or imply endorsement of these organizations or their programs by CDC or the U.S. Department of Health and Human Services. CDC is not responsible for the content of pages found at these sites.

Module Key

Lecture Content

Practical Content

Bonus intermediate content

Module Objectives

- Trainees will learn scripting basics applicable to any programming language or infrastructure: variables, loops, logic gates, functions, and shebang lines

Module Objectives

- Trainees will learn scripting basics applicable to any programming language or infrastructure: variables, loops, logic gates, functions, and shebang lines
- Trainees will be able to write simple Bash scripts and customize their shell environment by setting variables, using conditional statements (if/fi), and leveraging loops and aliases for automation and efficiency

Module Objectives

- Trainees will learn scripting basics applicable to any programming language or infrastructure: variables, loops, logic gates, functions, and shebang lines
- Trainees will be able to write simple Bash scripts and customize their shell environment by setting variables, using conditional statements (if/fi), and leveraging loops and aliases for automation and efficiency
- Trainees will understand parallel processing and its utility in optimizing runtime

Module Objectives

- Trainees will learn scripting basics applicable to any programming language or infrastructure: variables, loops, logic gates, functions, and shebang lines
- Trainees will be able to write simple Bash scripts and customize their shell environment by setting variables, using conditional statements (if/fi), and leveraging loops and aliases for automation and efficiency
- Trainees will understand parallel processing and its utility in optimizing runtime
- Trainees will learn pipe, pipelining, and how it applies to productionalized bioinformatics and *ad hoc* analyses

Bash Scripting

- Commands can be run interactively
  - Enter commands one at a time directly in the terminal
  - This is what we learned yesterday
- Commands can be saved in a script file
  - A script is a text file (e.g., .sh) containing multiple commands
- Scripts can be executed as a program
  - Make executable with chmod +x script.sh (permissions!) and run with ./script.sh

- `Echo` : print text to screen

```
#!/bin/bash

echo "Hello world!"
echo "Here are the csv files in your cwd"

pwd

ls *.csv
```

Bash Scripting

- Commands can be run interactively
  - Enter commands one at a time directly in the terminal
  - This is what we learned yesterday
- Commands can be saved in a script file
  - A script is a text file (e.g., .sh) containing multiple commands
- Scripts can be executed as a program
  - Make executable with chmod +x script.sh (permissions!) and run with ./script.sh

```
#!/bin/bash

echo "Hello world!"
echo "Here are the csv files in your cwd"

pwd

ls *.csv
```

```
qgx6@rosalind02:my-data$ chmod +x my-script.sh
qgx6@rosalind02:my-data$ ./my-script.sh
Hello world!
Here are the csv files in your cwd
/scicomp/home-pure/qgx6/my-data
empty_samplesheet.csv  samplesheet.csv  sorting_example.csv
```

- `Echo` : print text to screen

Bash Scripting

```
#!/bin/bash

echo "Hello world!"  1
echo "Here are the csv files in your cwd"  2

pwd  3

ls *.csv  4
```

Shebang line: tells computer how to interpret script

```
qgx6@rosalind02:my-data$ chmod +x my-script.sh
qgx6@rosalind02:my-data$ ./my-script.sh
Hello world!  1
Here are the csv files in your cwd  2
/scicomp/home-pure/qgx6/my-data  3
empty_samplesheet.csv  samplesheet.csv  sorting_example.csv  4
```

vim

- Vim is a terminal-based text editor
- Common on servers, HPC systems, and remote machines
- Designed for speed and efficiency
- **Keyboard-driven** (mouse won't work!)
- Uses different modes
  - Commands behave differently depending on the mode
- Highlights syntax usefully

- Vim demo

vim

- Start and exit
  - **vim file.txt** — open a file
  - :w save
  - :q quit
  - **:wq save & quit**
  - q! quit without saving
- Modes
  - **i — insert** (edit text)
  - Esc — return to normal mode
- Navigation
  - Arrow keys or h j k l
  - gg top of file
  - G bottom of file
  - :<number> go to line number
- Editing
  - dd delete line
  - yy copy line
  - p paste
  - u undo
- Search
  - /text search forward
  - n next match

Vim practical

1. Students download the pre-built file from the github

   Use VIM to navigate through it, chmod it, and modify it so that it can run and get the correct output

   a. Open the “cores_incorrect.sh” file in VIM
   b. Fix the shebang line to run a bash script
   c. Modify the “echo” command to fix the typo
   d. Delete the line that says “Delete this line”
   e. Navigate to the next line, follow the instructions in the file
   f. Navigate to the end of the file
   g. Save the modified file as
   h. Make the file executable
   i. Run the script

2. Extra practice: use VIM to create a bash file that lists all the files in a directory that end in .sh files and outputs a message that lists all bash files to stdout

Vim practical

1. Students download the pre-built file from the github
   Use VIM to navigate through it, chmod it, and modify it so that it can run and get the correct output
   a. Open the “cores_incorrect.sh” file in VIM > vim cores_incorrect.sh
   b. Fix the shebang line to run a bash script #!/bin/MISTAKE → #!/bin/bash (i, esc)
   c. Modify the “echo” command to fix the typo eecho →echo (navigate using arrows or letter keys, i, esc)
   d. Delete the line that says “Delete this line” dd
   e. Navigate to the next line, follow the instructions in the file delete #, copy line with vv
   f. Navigate to the end of the file G, paste line with p
   g. Save the modified file as :wq enter
   h. Make the file executable chmod +x mem
   i. Run the script ./mem_incorrect.sh

1. Extra practice: use VIM to create a bash file that lists all the files in a directory that end in .sh files and outputs a message that lists all bash files to stdout

Syntax

- **Syntax is the set of rules for writing code**
  - Like grammar in a spoken language
  - Strong coffee is good
  - El café fuerte es bueno.

- **Different programming languages have different syntax**
  - The same task looks different in Bash, Python, R, etc.
  - These two scripts do the same exact task

- **Small differences matter**
  - Symbols, spacing, and keywords must be used correctly
  - Using a text editor or Integrative Development Environment (IDE) can help detect errors

- **Errors often come from syntax issues**
  - Missing characters or incorrect formatting can prevent code from running

```bash
#!/bin/bash

my_sequence="ACTGACTG"
echo "the reverse compliment of $my_sequence is"
echo $my_sequence | tr "ACTG" "TGAC" | rev
```

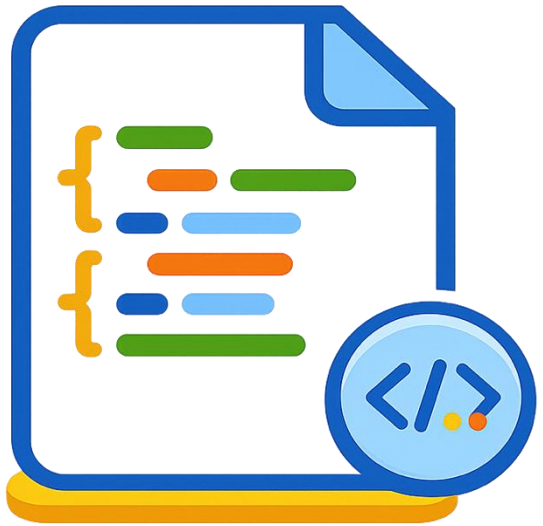
```python
#!/usr/bin/env python

seq = "ACTGACTG"
complement = {
    "A": "T",
    "T": "A",
    "C": "G",
    "G": "C"
}
print("the reverse compliment of", seq, "is")
rev_comp = "".join(complement[base] for base in reversed(seq))
print(rev_comp)
```

```
qgx6@rosalind-100:my-data$ ./syntax_example.sh
the reverse compliment of ACTGACTG is
CAGTCAGT
qgx6@rosalind-100:my-data$ ./syntax_example.py
the reverse compliment of ACTGACTG is
CAGTCAGT
```

Pseudo Code

- Focuses on logic, not syntax
  - Describe what the program does without worrying about language rules
  - One pseudocode outline can be implemented in Bash, Python, R, etc.
- Helps plan before coding
  - Breaks complex problems into clear, manageable steps
- Improves communication
  - Easy to read and discuss with collaborators, even non-programmers



Pseudo Code

Pseudocode for reverse compliment:

Input sequence = “my sequence”

A change to T
T change to A
G change to C
C change to G

Translate nucleotides in “my sequence”

Reverse the translated “my sequence”

Output “my sequence”

Variables

- Variables store values like text or numbers
  - Commonly used for file names, paths, or sequences
  - In syntax example, I set the variable my_sequence to hold “ACTGACTG”
- Assigned without spaces
  - name=value (spaces will cause errors)
- Accessed with a $
  - Use $name to reference the stored value
  - Good habit to use “$name” to reference
- Useful for making scripts reusable
  - Change a variable once instead of editing multiple commands

```
#!/bin/bash

my_sequence="ACTGACTG"
echo "the reverse compliment of $my_sequence is"
echo $my_sequence | tr "ACTG" "TGAC" | rev
```

Variables

- Variables can hold many kinds of things:
  - Bash variables are untyped by default: everything is treated like a string (not true for every coding language)
  - Numbers are handled through context
  - Arrays hold multiple values (also called lists in other languages)
- Variables can even hold the output of another command!
  - Subshell

```
[qgx6@cdp-client-01 ~]$ whoami
qgx6
[qgx6@cdp-client-01 ~]$ username=$(whoami) ; echo "hello $username!"
hello qgx6!
```

```
[qgx6@cdp-client-01 ~]$ date
Wed Jan 28 12:49:09 EST 2026
[qgx6@cdp-client-01 ~]$ current_time=$(date); echo "the current time now is $current_time"
the current time now is Wed Jan 28 12:49:17 EST 2026
[qgx6@cdp-client-01 ~]$ current_time=$(date); echo "the current time now is $current_time"
the current time now is Wed Jan 28 12:50:04 EST 2026
```

Current_time as a variable let me run the same code with different results: time changed!

Variables

Choose unique variable names: if you reuse a common variable name, you'll overwrite it
Certain variables in certain programming languages are already assigned (file, date, etc)

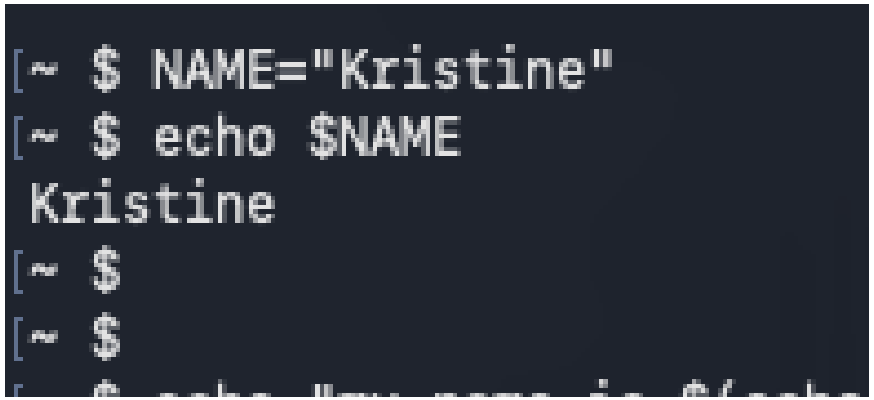
- VAR=value : Assign a variable
- $VAR : Use a variable

```
qgx6@nettie:my-data$ example_seq="ACTG"
qgx6@nettie:my-data$ echo $example_seq
ACTG
qgx6@nettie:my-data$ export example_seq="ACTG"
qgx6@nettie:my-data$ echo $(echo $example_seq)
ACTG
```

Variables

Choose unique variable names: if you reuse a common variable name, you'll overwrite it
Certain variables in certain programming languages are already assigned (file, date, etc)

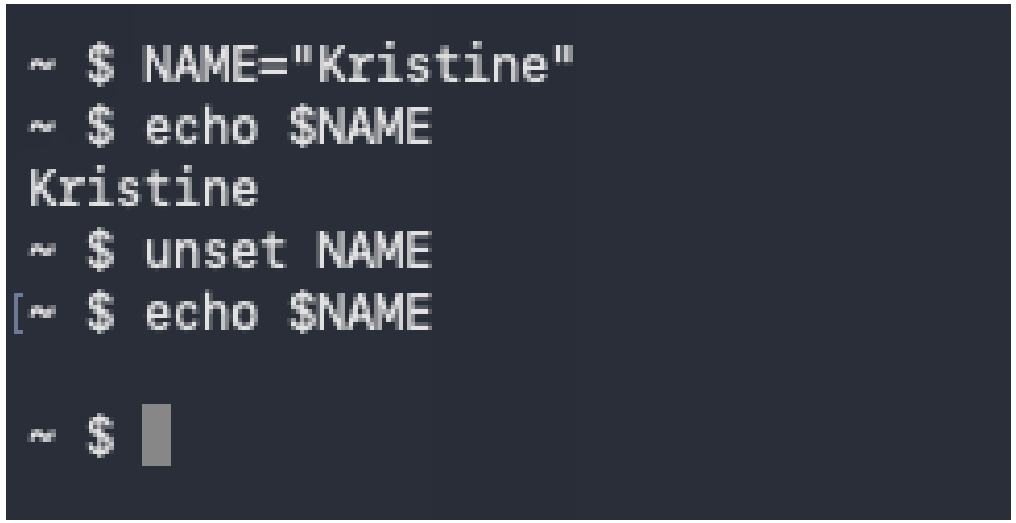
- VAR=value : Assign a variable
- $VAR : Use a variable
- export : Make variable available to subshells
- read : Read input from user
- set : Set or unset shell options
- unset : Remove variable or function



Variables

Choose unique variable names: if you reuse a common variable name, you'll overwrite it
Certain variables in certain programming languages are already assigned (file, date, etc)

- VAR=value : Assign a variable
- $VAR : Use a variable
- export : Make variable available to subshells
- read : Read input from user
- set : Set or unset shell options
- unset : Remove variable or function



Variables

- VAR=value : Assign a variable
- $VAR : Use a variable
- export : Make variable available to subshells
- read : Read input from user
- **set : Set or unset shell options**
- unset : Remove variable or function

```
~ $ set
BASH=/bin/bash
BASH_ARGC=()
BASH_ARGV=()
BASH_LINENO=()
BASH_REMATCH=([0]="6")
BASH_SOURCE=()
BASH_VERSINFO=([0]="3" [1]="2" [2]="57" [3]="1" [4]="release" [5]="arm64-apple-darwin25")
BASH_VERSION='3.2.57(1)-release'
COLORTERM=truecolor
COLUMNS=120
DIRSTACK=()
EUID=502
GROUPS=()
HISTFILE=/Users/qgx6/.bash_sessions/E4797256-FBC2-4526-84D6-4A70E0C2B405.historynew
HISTFILESIZE=500
HISTSIZE=500
HOME=/Users/qgx6
HOSTNAME=ML255447.local
HOSTTYPE=arm64
IFS=$' \t\n'
LANG=en_US.UTF-8
```

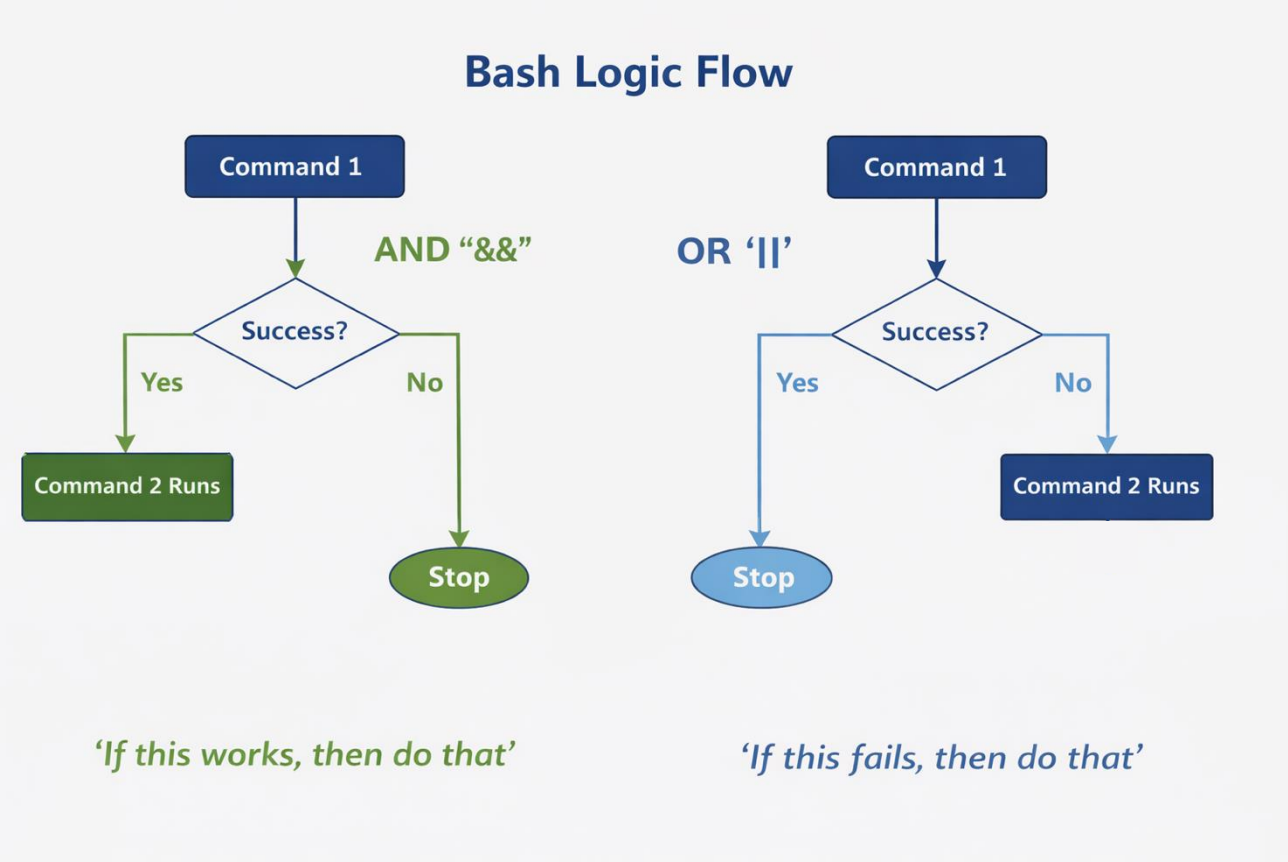
```
#!/bin/bash

set -euo pipefail
```

-e exit on error
-u error on undefined variable
-o pipefail : fail if any command in pipeline fails

Logic: and, or

- You can string together multiple commands using and/or logic
- Command1 && Command2
  - Execute command2 if command1 succeeds
- Command1 || Command2
  - Execute command 2 if command1 fails, otherwise, do not execute command 2



Logic: and, or

cd data && echo "Entered data directory" || echo "Could not enter data"

```
[qgx6@cdp-client-01 ~]$ cd data && echo "Entered data directory" || echo "Could not enter data"
-bash: cd: data: No such file or directory
Could not enter data
[qgx6@cdp-client-01 ~]$
```

```
[qgx6@cdp-client-01 ~]$ mkdir data
[qgx6@cdp-client-01 ~]$ cd data && echo "Entered data directory" || echo "Could not enter data"
Entered data directory
[qgx6@cdp-client-01 data]$
```

Logic: if, then

Bash if / then Statements

- Used to make decisions in a script
  - Run commands only when a condition is true
- Based on command success or a test condition
  - Exit status 0 = true, non-zero = false

```
#!/bin/bash

if [ -f data.txt ]; then
    echo "File exists"
fi
~
```

Basic structure

if [**condition**]; then
  commands
fi

```
[qgx6@cdp-client-01 data]$ bash check_file.sh
[qgx6@cdp-client-01 data]$ touch data.txt
[qgx6@cdp-client-01 data]$ bash check_file.sh
File exists
[qgx6@cdp-client-01 data]$
```

Logic: if, then

Common if conditional flags

File tests

-e file — path exists
-f file — regular file exists
-d file — directory exists
-s file — file exists and is not empty

String tests

-z string — string is empty
-n string — string is not empty
string1 = string2 — strings are equal
string1 != string2 — strings are not equal

Numeric comparisons

-eq — equal
-ne — not equal
-lt — less than
-le — less than or equal
-gt — greater than
-ge — greater than or equal

```
#!/bin/bash

if [ -f data.txt ]; then
    echo "File exists"
fi
~
```

```
[qgx6@cdp-client-01 data]$ bash check_file.sh
[qgx6@cdp-client-01 data]$ touch data.txt
[qgx6@cdp-client-01 data]$ bash check_file.sh
File exists
[qgx6@cdp-client-01 data]$
```

Logic: else, case

- if / else / fi handles yes-or-no decisions
  - Run one set of commands or another based on a condition

```
if [ -f data.txt ]; then
   echo "File exists"
else
    echo "File not found"
fi
```

- case / esac handles multiple choices
  - Cleaner than many `if / else` statements

```
case "$option" in
    start) echo "Starting" ;;
    stop)  echo "Stopping" ;;
    *)     echo "Unknown option" ;;
esac
```

Logic: else, case

- case / esac handles multiple choices
  - Cleaner than many `if / else` statements

```
day=$1

case "$day" in
  "Monday")
    echo "Start of the work week!"
    ;;
  "Tuesday")
    echo "Second day of the work week."
    ;;
  "Wednesday")
    echo "Midweek already!"
    ;;
  "Thursday")
    echo "Almost there!"
    ;;
  "Friday")
    echo "Last day of the work week!"
    ;;
  "Saturday" | "Sunday")
    echo "It's the weekend, time to relax!"
    ;;
  *)
    echo "Invalid day: $day. Please enter a valid day of the week."
    ;;
esac
```

```
qgx6@rosalind02:2026-BIFX-TRAINING$ ./day.sh Tuesday
Second day of the work week.
qgx6@rosalind02:2026-BIFX-TRAINING$ ./day.sh January
Invalid day: January. Please enter a valid day of the week.
qgx6@rosalind02:2026-BIFX-TRAINING$
```

Pseudo Code Practical

1. From a list of numbers, determine the mean
2. Calculate the percentage of numbers in the file that are below 6
3. How many different flu subtypes appear in a list?
4. Translate this DNA into its possible protein sequences (keeping in mind frames, coding and non-coding)

Pseudo Code Practical

1. From a list of numbers, determine the mean
   Possible solution:
   a. Count up the number of entries in the file, store that number as a variable (e.g. total_entries)
   b. Use a function to go through each of the numbers and add them up, line by line until you reach the end of the file, then store that number as a variable (sum_entries)
   c. Divide the sum_entries by the total_entries, but make sure to capture the final value as a "float" so that it is not rounded to the nearest whole integer

2. Calculate the percentage of numbers in a list that are below 6
3. How many different flu subtypes appear in a list?
4. Translate this DNA into its possible protein sequences (keeping in mind frames, coding and non-coding)

Pseudo Code Practical

1. From a list of numbers, determine the mean
2. Calculate the percentage of numbers in a list that are below 6

   Possible solution:

   a. Instantiate two variables, x and y set them equal to zero to begin (x is number of values less than six, y is number of values greater than six)

   b. Write a function to determine whether a value is less than 6

   c. Iterate through the list, using each number as an input to the function

      i. If the value is true, increment x by "1"

      ii. If the value is false, increment y by "1"

   d. Create a new variable, z, which is the sum of x and y (total number of values)

   e. Divide x by z, capture the result as a float

3. How many different flu subtypes appear in a list?
4. Translate this DNA into its possible protein sequences (keeping in mind frames, coding and non-coding)

Pseudo Code Practical

1. From a list of numbers, determine the mean
2. Calculate the percentage of numbers in the file that are below 6
3. How many different flu subtypes appear in a list?

   Possible solution:

   a. Create a new empty list to hold unique variables (e.g. unique_list)

   b. Iterate through the flu subtype list, comparing the value to the unique list. If the subtype is not in the empty list, add it

   Possible Solution:

   a. Take the list, sort it, and condense it to unique variables (see the pipeline practical!)
4. Translate this DNA into its possible protein sequences (keeping in mind frames, coding and non-coding)

Pseudo Code Practical

1. From a list of numbers, determine the mean
2. Calculate the percentage of numbers in the file that are below 6
3. How many different flu subtypes appear in a list?
4. Translate this DNA into its possible protein sequences (keeping in mind frames, coding and non-coding)

   Possible solution:

   a. Obtain a codon table for converting DNA codons to amino acids

   b. Starting at the first nucleotide, iterate through the sequence by three, and use the codon table to create a protein sequence

   c. Starting at the second nucleotide, repeat step b

   d. Starting at the third nucleotide, repeat step b

   e. Convert the DNA into its reverse complement (can expand here based on the steps in the lecture)

   f. Repeat steps b through d

Logic and Variables Practical

1. Using the ordinal number file on github (ordinal_check.sh)
    a. Exercise 1: change the ordinal statement to execute as true if the number is greater than 50 and less than 100
    b. Exercise 2: change the ordinal statement to execute as true if the number is less than 25 or greater than 75
    c. Exercise 3: change the ordinal statement to execute as true if the number is greater than 1 and less than 10, or greater than or equal to 90 and less than 100
2. Set a variable to be the current working directory. Change directory to the top level directory. Navigate back to the directory you were in using the variable

Logic and Variables Practical

1. Using the Random number generator file on github (random_number_generator.sh)
   a. Exercise 1: change the ordinal statement to execute as true if the number is greater than 50 and less than 100
      if [“$n” –gt 50 && “$n” –lt 100]; then (amend echo statements to reflect exercise instructions)
   b. Exercise 2: change the ordinal statement to execute as true if the number is less than 25 or greater than 75
      if [“$n” –lt 25] || [“$n” –gt 75]; then (amend echo statements to reflect exercise instructions)
   c. Exercise 3: change the ordinal statement to execute as true if the number is greater than 1 and less than 10, or greater than or equal to 90 and less than 100
      if [“$n” –gt 1] && [“$n” –lt 10] || [“$n” –ge 90] && [“$n” –lt 100]; then (amend echo statements to reflect exercise instructions)
2. Set a variable to be the current working directory. Change directory to the top level directory. Navigate back to the directory you were in using the variable

Logic and Variables Practical

1. Using the Random number generator file on github (random_number_generator.sh)
   a. Exercise 1: change the ordinal statement to execute as true if the number is greater than 50 and less than 100
   b. Exercise 2: change the ordinal statement to execute as true if the number is less than 25 or greater than 75
   c. Exercise 3: change the ordinal statement to execute as true if the number is greater than 1 and less than 10, or greater than or equal to 90 and less than 100
2. Set a variable to be the current working directory. Change directory to the top level directory. Navigate back to the directory you were in using the variable in one command

   Possible solution:

   > thispath=”$(pwd)”

   > cd

   > cd $thispath

Math

- Bash does not do math by default
- Arithmetic must be explicitly requested
  - Uses integers only by default
  - Can use bc –l for floating-point math
- Common operators
  - + add
  - - subtract
  - * multiply
  - / divide
  - % remainder
- Math is often used for counters
  - Loops, file counts, and simple logic

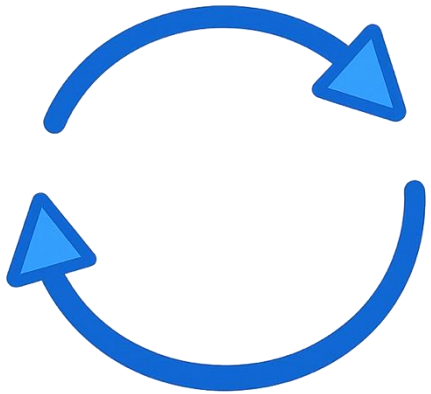
```
qgx6@nettie:fastq_pass$ cd barcode03/
qgx6@nettie:barcode03$ ls
FBA26656_pass_barcode03_2962c16b_05ab27bf_0.fastq.gz   FBA26656_pass_barcode03_2962c16b_05ab27bf_14.fastq.gz
FBA26656_pass_barcode03_2962c16b_05ab27bf_10.fastq.gz  FBA26656_pass_barcode03_2962c16b_05ab27bf_15.fastq.gz
FBA26656_pass_barcode03_2962c16b_05ab27bf_11.fastq.gz  FBA26656_pass_barcode03_2962c16b_05ab27bf_16.fastq.gz
FBA26656_pass_barcode03_2962c16b_05ab27bf_12.fastq.gz  FBA26656_pass_barcode03_2962c16b_05ab27bf_17.fastq.gz
FBA26656_pass_barcode03_2962c16b_05ab27bf_13.fastq.gz  FBA26656_pass_barcode03_2962c16b_05ab27bf_1.fastq.gz
qgx6@nettie:barcode03$ expr $(zcat * | wc -l ) / 4
7673
```

Loops

- Loops repeat commands automatically
  - Avoid copying and pasting the same command
- Useful for files, samples, and pipelines
  - Common in batch processing and automation
  - If you have a nanopore run of 24 samples, you want the same assembly and analysis to happen on each sample
- Run until a condition is met
  - Based on lists, counters, or logical tests

When to Use Loops

- **Process multiple files**
  - Run the same command on many inputs
- **Iterate over values**
  - Sample IDs, numbers, or directories
- **Control workflow logic**
  - Continue until a condition changes



Loops

`for / do / done`
Loop over a list of values

```
for file in *.txt; do
    echo "$file"
done
```

`while`
Loop while a condition is true

```
while read line; do
    echo "$line"
done < sorting_example.txt
```

```
qgx6@nettie:my-data$ bash loop.sh
sorting example.txt
```

```
qgx6@nettie:my-data$ bash loop.sh
1       One
2       Two
3       Three
4       Four
5       Five
6       Six
7       Seven
8       Eight
9       Nine
10      Ten
11      Eleven
12      Twelve
13      thirteen
14      Fourteen
15      Fifteen
```

Loops

`for / do / done`
Loop over a list of values

```
for file in *.txt; do
    echo "$file"
done
```

`while`
Loop while a condition is true

```
while read line; do
    echo "$line"
done < file.txt
```

```
i=1
while (( i <= 5 )); do
    echo "Count: $i"
    i=$((i + 1))
done
```

```
qgx6@nettie:my-data$ bash loop.sh
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

Loops

- Loops are powerful but can cause problems
  - Infinite loops occur when conditions never change
    - Example: while true; do … done
  - Forgetting to update loop variables
    - Counters or conditions must change inside the loop
- Easy ways to stop a runaway loop
  - Ctrl + C to interrupt
  - Test with echo before running real commands

Nested Loops

- A nested loop is a loop inside another loop
- The inner loop runs completely for each iteration of the outer loop

```
for i in 1 2 3
do
     for j in A B
     do
          echo “$i $j”
     done
done
```

Nested Loops

```
for i in 1 2 3
do
      for j in A B
      do
            echo “$i $j”
      done
done
```

OUTPUT:

```
1 A
1 B
2 A
2 B
3 A
3 B
```

- Outer loop runs 3 times
- Inner loop runs 2 time for each outer iteration
- Total executions: 3x2=6

Loops Practical

1. For every file in the loops_practical directory, if the file is not empty, print the name of the file to stout. (wc - -byte < filename can be used to give the size of a file)
2. For each file in a directory, find out if the file contains a shebang line as the first line, if so, print the filename to stout
3. Find the sick cat! (Hint: execute the files with shebangs!)

Loops Practical

1. For every file in the loops_practical directory, if the file is not empty, print the name of the file to stout. (“wc - -byte < filename” can be used to give the size of a file)

```
for i in *; do
   bytesize=$(wc --byte < $i);
   if [ "$bytesize" -gt 0 ]; then
      echo $i;
   fi;
done
```

2. Find the sick cat! (Hint: execute the files with shebangs!)

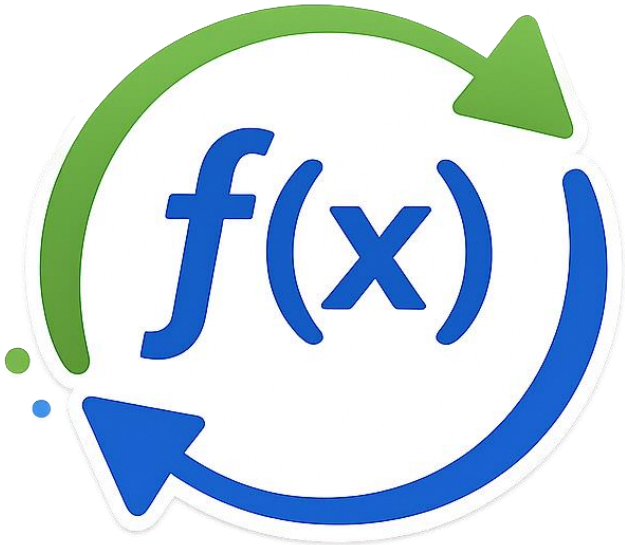
Loops Practical

1. For every file in the loops_practical directory, if the file is not empty, print the name of the file to stout. (wc - -byte < filename can be used to give the size of a file)
2. Find the sick cat! (Hint: execute the files with shebangs!)

```
for i in *; do
    bytesize=$(wc --byte < $i);
    if [ "$bytesize" -gt 0 ]; then
        if bash $i 2>/dev/null; then
            echo $i;
        fi;
    fi;
done
```

Functions

- Functions group related commands
- Package repeated logic into a single unit
- Improve script organization and make long scripts easier to read and maintain
- Defined once, reused many times
- Should have meaningful names to describe what the function does



Functions

- Function called reverse_complement
- $1 is the first argument passed to the function
- tr performs base complementation
- rev reverses the sequence
- Function output goes to script output, like a command

```
#!/bin/bash

reverse_complement() {
    seq="$1"
    echo "$seq" \
      | tr 'ATCG=' 'TAGC' \
      | rev
}

reverse_complement "ATGCCGTA"

reverse_complement "TCGATCGA"
```

\ is the opposite of ;

```
qgx6@rosalind02:my-data$ vim revc.sh
qgx6@rosalind02:my-data$ ./revc.sh
TACGGCAT
TCGATCGA
```

Functions

- If you notice that you are writing the same code over and over, it can be useful to package that part as a function
  - then replace the repeats with said function
  - DRY = Don’t Repeat Yourself
- You define the arguments for a function, so it is useful to remember the variables you have already used and make sure not to overwrite them or cause logic errors in the script
- Can define the output of a function as a variable, output to screen, or even a logic evaluation (true/false)

Errors

Common Bash Errors

- Syntax errors
  - Missing `do`, `done`, `then`, or `fi`
  - Extra or missing brackets `[ ]`
  - Missing or extra “ “ ( )
- Infinite loops
  - Loop condition never changes
  - Missing counter update or exit condition
- Permission errors
  - Script not executable (`chmod +x script.sh`)
  - No access to files or directories
- Variable mistakes
  - Using `$var` before it is set
  - Missing `$` when referencing a variable

A good IDE can help with many of these syntax errors, because it will color code things, make suggestions about what might be common fixes

Debugging Bash scripts

- Start with Pseudocode!
- Run in debug mode
  - bash -x script.sh shows each command as it runs
- Print values to check logic
  - Use echo "$variable" inside loops or conditions
- Test commands step by step
  - Run lines manually in the terminal
- Start simple and build up
  - Confirm each part works before combining commands

```
TCGATCGA
qgx6@rosalind02:my-data$ bash -x revc.sh
+ '[' -z '' ']'
+ case "$-" in
+ __lmod_vx=x
+ '[' -n x ']'
+ set +x
Shell debugging temporarily silenced: export LMOD_SH_DBG_ON=1 for this output (/usr/share/lmod/lmod/init/bash)
Shell debugging restarted
+ unset __lmod_vx
+ reverse_complement ATGCCGTA
+ seq=ATGCCGTA
+ echo ATGCCGTA
+ tr ATCG= TAGC
+ rev
TACGGCAT
+ reverse_complement TCGATCGA
+ seq=TCGATCGA
+ echo TCGATCGA
+ tr ATCG= TAGC
+ rev
TCGATCGA
```

Standard Output and Standard Error

- Standard Output (stdout)
  - Normal program output (results, messages)
  - File descriptor 1
- Standard Error (stderr)
  - Error, logging, and warning messages
  - File descriptor 2
- Allows errors to be handled differently from results
- Useful for scripting and debugging: redirect output and errors independently

Standard Output and Standard Error

- Both go to the terminal by default, but can be redirected separately or together
- Common redirection operators
  - > redirect stdout
  - 2> redirect stderr
  - &> redirect both
  - | tee : writes standard input to standard output
- Useful patterns
  - Save results while still seeing errors
  - Suppress errors during batch processing

Example

command > output.txt 2> error.log

Pipelines

- Pipelines connect commands together
- Output of one command becomes input to the next
- Use the pipe symbol |
- Pass data through a sequence of tools
- Each tool does one job well
- Common in data processing
    - Text files, FASTQ files, and command output

```
qgx6@nettie:fastqs$ ls
24-003692-001-original_R1.fastq.gz  24-009110-009-original_R1.fastq.gz  24-010354-015-original-300_subset_R1.fastq.gz
24-003692-001-original_R2.fastq.gz  24-009110-009-original_R2.fastq.gz  24-010354-015-original-300_subset_R2.fastq.gz
qgx6@nettie:fastqs$ for file in $(ls *.fastq.gz); do echo $file && zcat $file | grep "^@" | wc -l ; done
24-003692-001-original_R1.fastq.gz
275125
24-003692-001-original_R2.fastq.gz
274509
24-009110-009-original_R1.fastq.gz
183609
24-009110-009-original_R2.fastq.gz
183609
24-010354-015-original-300_subset_R1.fastq.gz
800000
24-010354-015-original-300_subset_R2.fastq.gz
800000
```

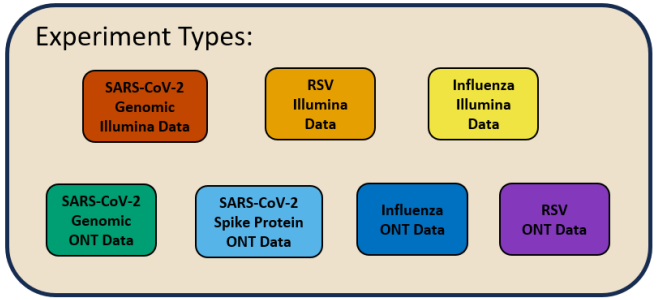
Pipelines

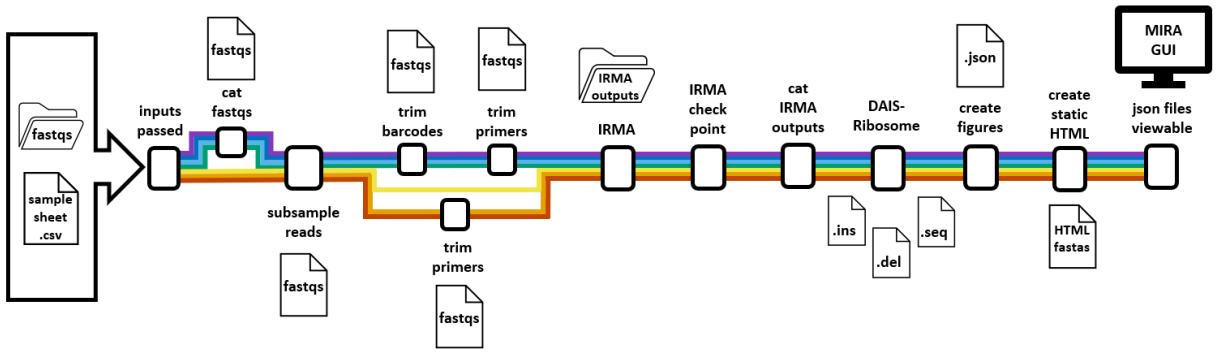
Bioinformatic pipelines grow from this concept

Example pseudo code:

#!/bin/bash

demultiplex > fastq
genome assembly on fastq > fasta
update database with genome assembly output
curate assembly fastas





Pipelines within pipelines:
Genome Assembly → MIRA
check inputs | combine fastqs | subsample reads | trim barcodes | trim primers | run IRMA | check IRMA output | combine output | annotate | create figures

Parallel processing

- Two common approaches
  - *Task parallelism*: same command on many inputs
  - *Data parallelism*: split one dataset into parts

  - Within tools (multi-threaded software)
  - Across tools (running jobs simultaneously)
- Common Bash patterns
  - Background jobs (`&`)
  - Job control with `wait`
- Common bioinformatics tools support multithreading
  - Use flags like `-t`, `-p`, or `--threads`

```
bwa mem -t 8 ref.fa reads.fq > aln.sam
```

Pipeline practical

1. List the contents of a directory, pipe that output to word count to find how many files there are (may be helpful to use man wc to find out what wc can do)
2. List the contents of a file, sort the contents and find a list of the unique values
3. Decode the secret message: 1%76q#948^4q5@23q2q492q07&/@i5#q#76
    a. Convert all the numbers into letters using the provided variable
    b. Reverse the order of the sequence
    c. Cut using “/” as delimiter, take the second field
    d. convert all the letters into uppercase

Pipeline practical

1. List the contents of a directory, pipe that output to word count to find how many files there are (may be helpful to use man wc to find out what wc can do)
   Possible solution:
   ls | wc
2. List the contents of a file, sort the contents and find a list of the unique values
3. Decode the secret message: 1%76q#948^4q5@23q2q492q07&/@i5#q#76
   a. Convert all the numbers into letters using the provided variable
   b. Reverse the order of the sequence
   c. Cut using “/” as delimiter, take the second field
   d. convert all the letters into uppercase

Pipeline practical

1. List the contents of the /usr/bin directory, pipe that output to word count to find how many files there are (may be helpful to use “man wc” to find out what wc can do)

2. List the contents of a file, sort the contents and find a list of the unique values (bonus: count the number of unique values and output to screen)
   Possible solution:
   cat flu_types.txt | sort | uniq
   cat flu_types.txt | sort | uniq | wc

3. Decode the secret message: 1%76q#948^4q5@23q2q492q07&/@i5#q#76
   a. Convert all the numbers into letters using the provided variable
   b. Reverse the order of the sequence
   c. Cut using “/” as delimiter, take the second field
   d. convert all the letters into uppercase

Pipeline practical

1. List the contents of a directory, pipe that output to word count to find how many files there are (may be helpful to use man wc to find out what wc can do)
2. List the contents of a file, sort the contents and find a list of the unique values
3. Decode the secret message: 1%76q#948^4q5@23q2q492q07&/@i5#q#76
   a. Convert all the numbers into letters using the provided conversion
   b. Reverse the order of the sequence
   c. Cut using "/" as delimiter, take the second field
   d. convert all the letters into uppercase

      Possible solution:

      > secret_message="$(cat decode_the_secret_message.txt)"

      > secret_key="$(cat secret_message_key.txt)"

      > echo $secret_message | tr $secret_key | rev | cut -d"/" -f 2 | tr '[:lower:]' '[:upper:]'

      Possible solution:

      > echo "1%76q#948^4q5@23q2q492q07&/@i5#q#76" | tr '123456789@#0%^&q=' '!abehnoprstuwxy_' | rev | cut -d"/" -f 2 | tr '[:lower:]' '[:upper:]'

More useful tools and shortcuts

- **history : Show command history**
- **!! : Repeat last command**
- **!n : Run nth command from history**
- alias : Create command shortcut
- unalias : Remove alias
- clear : Clear terminal screen
- man : Display command manual
- help : Show built-in help
- date : Show or set system date
- time : Show the runtime of the following command
- dos2unix : convert Windows file to readable Unix file, remove carriage returns

```
qgx6@nettie:my-data$ ls
empty_samplesheet.csv  fastqs  fastqs_copy  fastqs.tar  loop.sh  my-script.sh  samplesheet.csv
qgx6@nettie:my-data$ !!
ls
empty_samplesheet.csv  fastqs  fastqs_copy  fastqs.tar  loop.sh  my-script.sh  samplesheet.csv
```

```
qgx6@nettie:my-data$ ls
empty_samplesheet.csv  fastqs  fastqs_copy  fastqs.tar  loop.sh  my-script.sh  samplesheet.csv
qgx6@nettie:my-data$ pwd
/scicomp/home-pure/qgx6/my-data
qgx6@nettie:my-data$ echo "hello world"
hello world
qgx6@nettie:my-data$ history | tail -3
10069  pwd
10070  echo "hello world"
10071  history | tail -3
qgx6@nettie:my-data$ !10069
pwd
/scicomp/home-pure/qgx6/my-data
```

Bash tip! Ctrl+r to SEARCH your history in command line!

More useful tools and shortcuts

- history : Show command history
- !! : Repeat last command
- !n : Run nth command from history
- **alias : Create command shortcut**
- **unalias : Remove alias**
- **clear : Clear terminal screen**
- man : Display command manual
- help : Show built-in help
- date : Show or set system date
- time : Show the runtime of the following command
- dos2unix : convert Windows file to readable Unix file, remove carriage returns

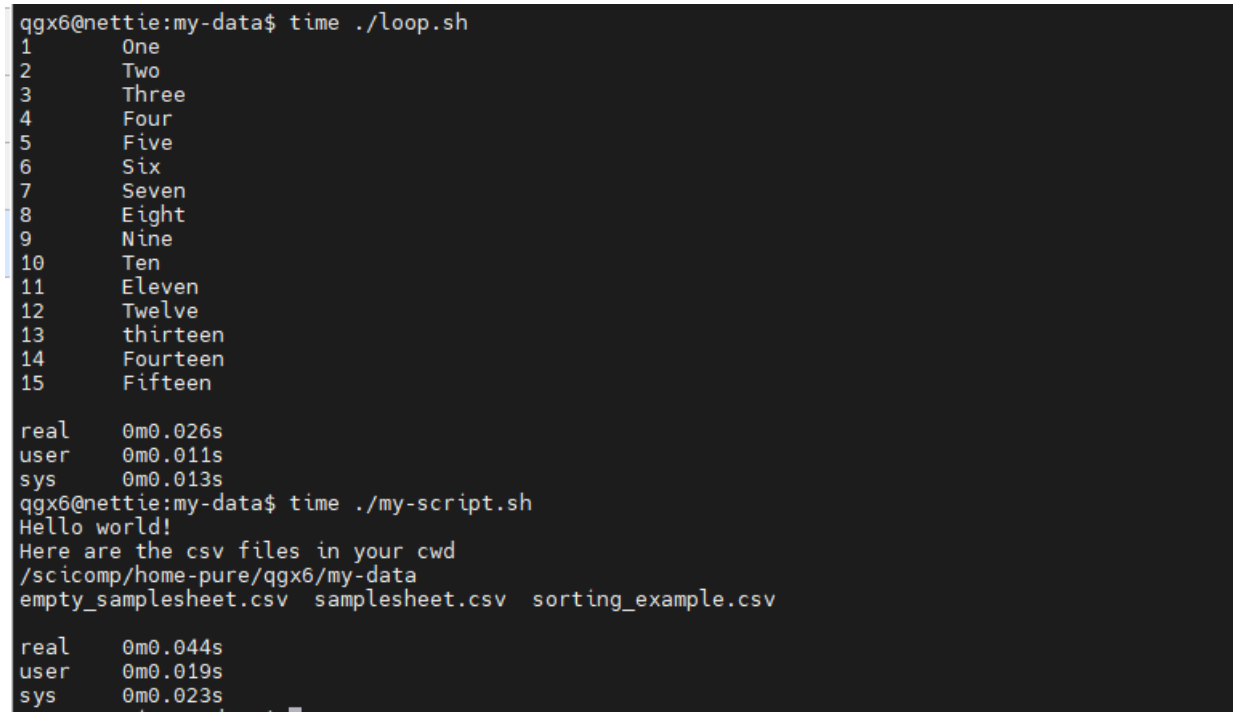
```
qgx6@nettie:my-data$ cat ~/.bashrc
# .bashrc


# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions
alias GWA="cd /scicomp/groups/OID/NCIRD/ID-OD/ISA"
alias la="ls -lahtr"
alias st="/scicomp/groups-pure/sars2seq/bin/samtools-1.11/samtools"
alias h5="cd /scicomp/groups/OID/NCIRD/ID-OD/VSDB/BIA/FLU_SC2_SEQUENCING"
```

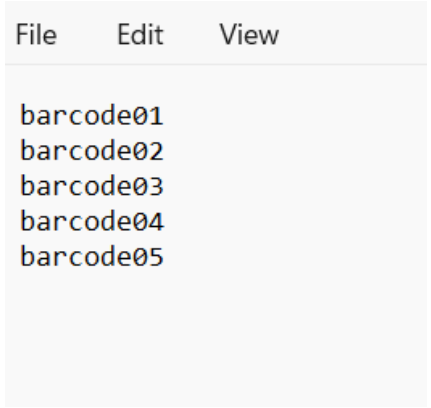
More useful tools and shortcuts

- history : Show command history
- !! : Repeat last command
- !n : Run nth command from history
- alias : Create command shortcut
- unalias : Remove alias
- clear : Clear terminal screen
- **man : Display command manual**
- **help : Show built-in help**
- **date : Show or set system date**
- **time : Show the runtime of the following command**
- dos2unix : convert Windows file to readable Unix file, remove carriage returns



More useful tools and shortcuts

- history : Show command history
- !! : Repeat last command
- !n : Run nth command from history
- alias : Create command shortcut
- unalias : Remove alias
- clear : Clear terminal screen
- man : Display command manual
- help : Show built-in help
- date : Show or set system date
- time : Show the runtime of the following command
- **dos2unix : convert Windows file to readable Unix file, remove carriage returns**




```
qgx6@nettie:my-data$ cat -v windows_barcodes.txt
barcode01^M
barcode02^M
barcode03^M
barcode04^M
barcode05^M
qgx6@nettie:my-data$ dos2unix windows_barcodes.txt
dos2unix: converting file windows_barcodes.txt to Unix format...
qgx6@nettie:my-data$ cat -v windows_barcodes.txt
barcode01
barcode02
barcode03
barcode04
barcode05
```

Higher level coding languages

- Bash *scripting* differs from other coding languages
- Abstract away low-level details
  - Manage memory, data types, and errors automatically
  - Can be compiled into binary
- More expressive and readable
- Fewer lines of code to perform complex tasks
- Rich ecosystems and libraries
  - Built-in tools for data analysis, visualization, and networking

Object oriented programming

- Group data and actions together
- Similar to treating a file + its operations as one unit
- Classes act like templates
  - Define a “sample,” “read set,” or “experiment” once
- Objects represent real things
  - Each object holds its own data and methods
- Bash: “Run commands on files”
- Python/R: “Create objects, operate on data”

When to stop using BASH

- Logic becomes complex
  - Many nested loops, if/else, or long one-liners that are hard to read
- Data structures are needed
  - Lists of samples, tables, dictionaries, or metadata don’t fit cleanly in Bash
- Error handling gets messy
  - Too many &&, ||, and manual checks for failures
- Scripts grow large
  - Files longer than ~100–200 lines become difficult to maintain
- You need reproducibility and testing
  - Unit tests, versioned packages, and structured logging are easier in Python/R
- Parallelization becomes necessary
  - Wrap the bash in a workflow manager like snakemake, wdl, or nextflow
- Visualization is needed
  - Better packages and libraries in Python and R
- Other coding languages or packages exist to better handle specific use-cases (biopython, etc)

Packages and tools!

- As workflows grow, reuse becomes important
  - Copy-pasting scripts leads to errors and drift
- Packages bundle code and functionality
  - Install once, use everywhere
- Tools provide standard, tested solutions
  - Avoid rewriting common tasks
  - Utilize code written by others that is optimized for a certain task
- Easier collaboration and reproducibility
  - Others can install the same package and get the same results
  - Use Bash to *orchestrate* tools (in a pipeline)