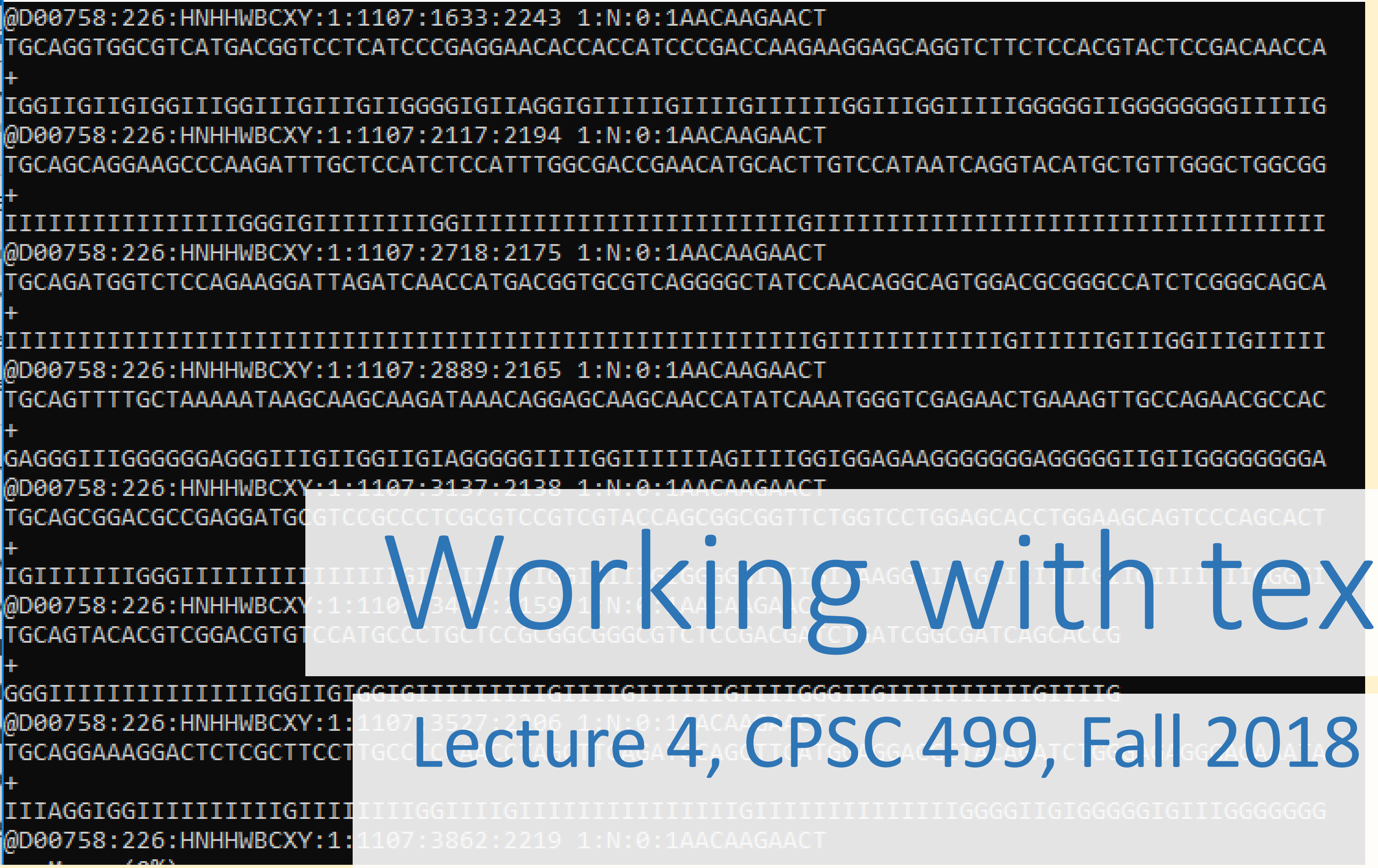


```
@D00758:226:HNHHWBCXY:1:1107:1633:2243 1:N:0:1AACAAGAACT
TGCAGGTGGCGTCATGACGGTCCTCATCCCGAGGAACACCACCATCCCGACCAAGAAGGAGCAGGTCTTCTCCACGTACTCCGACAACCA
+
IGGIIGIIGIGGIIIGGIIIGIIIGIIGGGGIGIIAGGIGIIIIIGIIIIGIIIIIIGGIIIGGIIIIIGGGGGIIGGGGGGGGIIIIIG
@D00758:226:HNHHWBCXY:1:1107:2117:2194 1:N:0:1AACAAGAACT
TGCAGCAGGAAGCCCAAGATTTGCTCCATCTCCATTTGGCGACCGAACATGCACTTGTCCATAATCAGGTACATGCTGTTGGGCTGGCGG
+
IIIIIIIIIIIIIIIIGGGIGIIIIIIIIIGGIIIIIIIIIIIIIIIIIIIIIIIIGIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
@D00758:226:HNHHWBCXY:1:1107:2718:2175 1:N:0:1AACAAGAACT
TGCAGATGGTCTCCAGAAGGATTAGATCAACCATGACGGTGCGTCAGGGGCTATCCAACAGGCAGTGGACGCGGGCCATCTCGGGCAGCA
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIGIIIIIIIIIIIIGIIIIIIGIIIGGIIIGIIIII
@D00758:226:HNHHWBCXY:1:1107:2889:2165 1:N:0:1AACAAGAACT
TGCAGTTTTGCTAAAAATAAGCAAGCAAGATAAACAGGAGCAAGCAACCATATCAAATGGGTCGAGAACTGAAAGTTGCCAGAACGCCAC
+
GAGGGIIIGGGGGGAGGGIIIGIIGGIIGIAGGGGGIIIIGGIIIIIIAGIIIIGGIGGAGAAGGGGGGGAGGGGGIIGIIGGGGGGGGA
@D00758:226:HNHHWBCXY:1:1107:3137:2138 1:N:0:1AACAAGAACT
TGCAGCGGACGCCGAGGATGCGTCCGCCCTCGCGTCCGTCGTACCAGCGGCGGTTCTGGTCCTGGAGCACCTGGAAGCAGTCCCAGCACT
+
IGIIIIIIIGGGIIIIIIII[illegible]
@D00758:226:HNHHWBCXY[illegible]
TGCAGTACACGTCGGACGTG[illegible]ATCGGCGATCAGCACCG
+
GGGIIIIIIIIIIIIIIIIGGIIGIGGIGIIIIIIIIIIGIIIIGIIIIIIGIIIIGGGIIGIIIIIIIIIIIGIIIIG
@D00758:226:HNHHWBCXY:1:1107:3527:2[illegible]
TGCAGGAAAGGACTCTCGCTTCCT[illegible]
+
IIIAGGIGGIIIIIIIIIIGIIIIIIIIGGIIIGIIIIIIIIIIIIIIIIIGIIIIIIIIIIIIIIIGGGGIIGIGGGGGIGIIIGGGGGGG
@D00758:226:HNHHWBCXY:1:1107:3862:2219 1:N:0:1AACAAGAACT
```

# Working with text

## Lecture 4, CPSC 499, Fall 2018

## Intro to working with strings

- We have already seen a lot of strings in this course (anytime something is in quotes)
- You can have a vector of strings, which is class `character`
- Typing within either single or double quotes will create a string

# Importing data as text

- With `read.csv`, `read.table`, etc., use `stringsAsFactors = FALSE` if you want to do text processing on a column
- (Leave `stringsAsFactors = TRUE` if you just want to treat it as a categorical variable)
- `readLines` will import a file as a character vector rather than data frame

# String operators

- ==, !=, >, <, >= , <= all work
  - (> and < refer to alphabetical order)
- + does not work (sorry Python fans)

```
mysequence == "AGG"
```

## `paste`

- Very handy function for concatenating strings together
- Element-wise concatenation of strings from multiple vectors
    - `sep` argument indicates the separator string
- Concatenation of all strings within one vector
    - `collapse` argument indicates the separator string

`"Height:"` `1.5` `"meters"` $\rightarrow$ `"Height: 1.5 meters"`

## `nchar`

- Returns the number of characters in a string
- Vectorized; you can give it a vector of strings and get the length of each element

$$\texttt{"}\underbrace{\texttt{Height: 1.5 meters}}_{\text{18 characters}}\texttt{"}$$

# `substring`

- Returns part of a string, as specified by positions
- Give it the string, the start position, the end position.
- All three of those arguments can be vectors

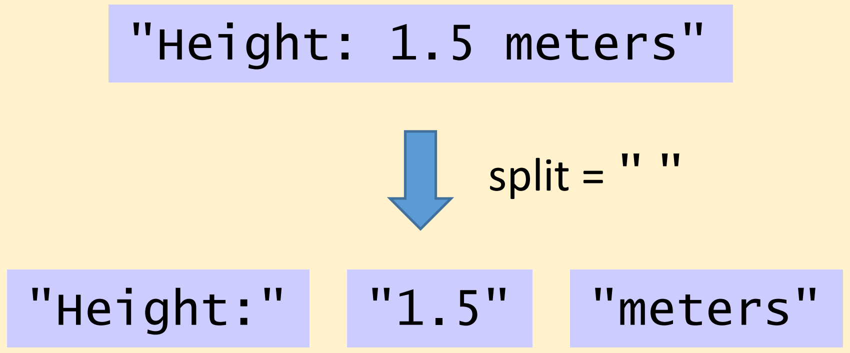
```
"Height: 1.5 meters"
```

$\downarrow$ first = 2, last = 5

```
"eigh"
```

# `strsplit`

- Splits a string into multiple strings
- The `split` argument is a character or longer string that is used for delimiting the new strings
- Returns a list, one vector for each original string
- (Note that even if you pass it just one string, it still returns a list)



## `formatC` and `prettyNum`

- Convert number to text
- Can set number of significant digits or number of digits after decimal place
- Trailing or leading zeros `"1.80"` `"015"`
- Scientific notation `"5.36e-04"`
- Commas `"1,246,782"`
- Handy for output to a table for publication, getting the right format for a file, numbering a list of genes, etc.

## Mini exercise

- Make a new column for our table of plant germplasm information
- It should contain a new sample id, in the format “Sample001” to “Sample558”.

# match

- Works for numbers or strings
- Find first match of a value inside another vector
- Handy for reordering one data set based on another

$$\text{match}\left\{\begin{array}{l}\texttt{c("a", "b")}\\ \texttt{c("b", "d", "a", "e")}\end{array}\right. \Rightarrow \quad 3, 1$$

## Indexing with strings – named vectors

- Any vector can have a “names” attribute giving a name to each element
- Ideally all names are unique
- Many functions will carry names over to results
- Functions to retrieve or assign
  - `names` for vectors and lists
  - `rownames`, `colnames` for data frames and matrices
- Can set up at vector creation without quotes

```
c(a = 5, b = 6)
```

## Mini exercise

- Take the “X.sample” column of the plant germplasm data table, and assign these same values to the row names of the table using `rownames`.
- Index the data frame by row names to get just the rows for JY001, JY012, and JY028.

# Pattern matching

- `grep`: find matches across a vector of strings
  - Returns numeric index or string itself (`value` arg)
  - `grepl` returns TRUE/FALSE vector
  - Named after Unix command

grep $\left\{ \begin{array}{l} \texttt{"ember"} \\ \texttt{c("September", "October", "November")} \end{array} \right. \Rightarrow$ 1, 3

# Pattern matching

- "find and replace" functions for strings:
  - `sub`: replaces first instance of pattern
  - `gsub`: replaces all instances of pattern

gsub $\left\{ \begin{array}{l} \texttt{"ember", "month"} \\ \texttt{c("September", "October", "November")} \end{array} \right.$

```
"ember", "month"
c("September", "October", "November")
```

$\Downarrow$

```
c("Septmonth", "October", "Novmonth")
```

# Pattern matching

- Functions to give positions and lengths of pattern matches within a string:
  - `regexpr` returns vector with first match position
  - `gregexpr` returns list with all match positions

gregexpr

```
"e"
c("September", "October", "November")
```

$\downarrow$

2, 5, 8
6
4, 7

# The `fixed` argument

- `fixed = TRUE` means pattern must be matched exactly
- `fixed = FALSE` (the default) matches strings using **regular expressions**

- If you already know regular expressions in Perl, you can use `perl = TRUE` to use that syntax

## Regular expressions

- Used by `grep`, `grepl`, `gsub`, `sub`, `regexpr`, `gregexpr`, `strsplit`
- For when you need to match not an exact string, but an overall pattern
- For example, “Chr” followed by two numbers, and it must be at the beginning of a string

```
"^Chr[0-9]{2}"
```

# Regular expression basics

- Most characters (letters and numbers) match themselves
- . (period) indicates any character
- ^ indicates the beginning of the string
- $ indicates the end of the string

```
"^Chr..$"
```

→

Start with ^ and end with $ to indicate that this must be the whole string.  This would match “Chr26” but not “Chr30A”.

## Repeating characters

- Put one of these after a character to indicate how many times that character should repeat:
  - `?` : one or zero matches
  - `*` : zero or more matches
  - `+` : one or more matches
  - `{2}` : two matches
  - `{2,}` : two or more matches
  - `{2,4}` : two to four matches
- Enclose a group of characters in parentheses if you want to repeat a group

## Matching one of several characters or strings

- Group characters with square brackets `[]`
  - Start with `^` to indicate any character *except* any of these
  - Use hyphen to indicate range, e.g. `[3-6]`
- Group strings with parentheses, with pipe to separate them `(|)`

```
"(Chr|Scaffold)"
```

To match “Chr” or “Scaffold”

## Mini exercise

- Find all sample names from the plant germplasm table that start with "`JM2014-S`" or "`JM2014-K`"
- Use `grep` and regular expressions

## Escaping characters

- Say you actually want to search for the characters ^, [, $, etc. in your string.
- A backslash before the character **escapes** that character, meaning it should be interpreted literally
- Since a backslash is also an escape character for creating strings in R, you generally need to type two of them out.

```
"locus2\\.5"
```

Searches for “locus2.5”, rather than using the period as a wildcard character.

## Character classes

- `[:digit:]` : any number
- `[:alpha:]` : any letter
- `[:blank:]` : space or tab
- `[:space:]` : space, tab, or newline
- `[:punct:]` : any punctuation
- `[:upper:]` : upper case letters
- `[:lower:]` : lower case letters
- `[:alnum:]` : letters and numbers

## Using character classes

- Generally have to enclose them in brackets again
  - `[[:digit:]]+` : one or more digits
  - `[[:digit:][:punct:]]` : a digit or punctuation mark

## Mini exercise

- Write a regular expression to find samples that start with "JY" and then three numbers
- Make it so that it will not match "JY118-1"