Manipulating Strings

Max Callaghan

2022-09-15

Objectives

Objectives

In this session we'll cover a broad overview of how strings (text sequences) can be manipulated in R/Python, as well as how we can use REGEX expressions to extract information from texts.

But first, we will do a quick recap of the scraping exercise

Recap

Recap

I posted a step-by-step solution

Some useful basic concepts

- Data types:
  - A vector is a 1-dimensional series of values with the same type
  - A list is simply a list of things that can be accessed via index, or via named attributes
- Indexes allow us to access the nth item of a series
- Loops allow us to repeat an action several times (for example do the same operation on each item of a list)

Lists in R

Let's make a list in R. We often encounter these as key, value pairs. This is a list of named attributes

```
l <- list(height=172, weight=75)
l
```

```
## $height
## [1] 172
##
## $weight
## [1] 75
```

We can **set** OR **get** the value of a certain attribute with the $ sign or with square brackets

```
print(l$height)
```

```
## [1] 172
```

```
l[["height"]] <- 188
print(l[["height"]])
```

```
## [1] 188
```

Lists in R II

We can also **get** OR **set** the elements of a list by saying if we want the first, second, etc. item

```
print(l$height)
```

```
## [1] 188
```

```
print(l[[1]])
```

```
## [1] 188
```

Lists and dictionaries in Python

In python, we have two distinct data types - key,value pairs (dictionaries), and lists.

Dictionaries are defined inside curly brackets. We can only get or set the values using the keys

```
d = {"height": 172, "weight": 75}
print(d["height"])
```

```
## 172
```

```
d["height"] = 188
print(d["height"])
```

```
## 188
```

Lists are simply comma-separated series of things, from which we can access the 1st, 2nd, etc. item. Warning! Python uses ZERO-indexing

```
l = [172, 175]
print(l[0])
```

```
## 172
```

JSON Records

The JSON format strongly resembles python lists and dictionaries, which we can flexibly combine as a list of records with attributes

```
people = [
  {"height": 172, "name": "Liz", "top_3_colours": ["Red", "Blue", "Green"]},
  {"height": 180, "weight": "Kwasi", "top_3_colours": ["Yellow", "Orange", "Lilac"]},
]
```

We can also represent this in R with a list of lists

```
people = list(
  list(height=172, name="Liz", top_3_colours=list("Red", "Blue", "Green")),
  list(height=188, name="Kwasi", top_3_colours=list("Yellow", "Orange", "Lilac"))
)
```

How would we get or set the 3rd favourite colour of the 1st person?

Loops in R

Loops allow us to repeat an action multiple times. This is especially helpful when we have lists. In R, the thing we loop through goes in the brackets, and what we do with the thing goes in curly brackets

```
for (person in people) {
  print(person$height)
}
```

```
## [1] 172
## [1] 188
```

An alternative to loops is to use apply, which performs a function on a list and returns a list

```
x <- lapply(people, function(x) return(x$height))
print(x)
```

```
## [[1]]
## [1] 172
##
## [[2]]
## [1] 188
```

Loops in Python

Loops allow us to repeat an action multiple times. This is especially helpful when we have lists.

In python, we write what we call each thing in a list of things we want to loop through before a colon: Then everything else indented below is what we do with the thing

```
for person in people:
    print(person["height"])
```

```
## 172
## 180
```

Basic introduction to strings

Representing strings

To create a string, we need to put quotation marks around its content

```
x <- "Hello world"
print(x)
```

```
## [1] "Hello world"
```

```
x <- Hello world
print(x)
```

```
## Error: <text>:1:12: unexpected symbol
## 1: x <- Hello world
##                ^
```

Quotation marks

You can use either single or double quotation marks, but the same type of quotation mark again will close the string and cause an error. The backslash character "\" is called an "escape" character.

```
x <- "Hello world"
y <- 'Hello world'
cat(c(x,y))
```

```
## Hello world Hello world
```

```
x <- 'They're great'
```

```
## Error: <text>:1:12: unexpected symbol
## 1: x <- 'They're
##             ^
```

```
x <- "They're great"
```

```
x <- 'Tony says "they\'re great"'
cat(x)
```

```
## Tony says "they're great"
```

The difference between strings and numbers

Pay attention to the type of data

```
x <- 1
y <- "2"
x+y
```

```
## Error in x + y: non-numeric argument to binary operator
```

```
x = 1
y = "2"
x+y
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: unsupported operand type(s) for +: 'int' a
```

Inserting variables into strings

We can use sprintf / f strings to insert variables into text and define how they are formatted.

```r
addressee <- "world"
sprintf("Hello %s, Pi is equal to %.2f", addressee, pi)
```

```
## [1] "Hello world, Pi is equal to 3.14"
```

```python
addressee = "world"
print(f"Hello {addressee}, Pi is equal to {math.pi:.2f}")
```

```
## Hello world, Pi is equal to 3.14
```

An exercise with for loops, variables and strings

Using the list of records we made earlier, write and print a message to each person on our list, letting them know that we have T-shirts for sale in their favourite colour.

Special characters in strings

Take note of the special characters \n, \r, \t (new line, carriage return, and tab). You might need these when processing, splitting up text you acquire.

```
x <- 'Tony says\n"they\'re great"'
cat(x)
```

```
## Tony says
## "they're great"
```

```
x <- 'Tony says \r"they\'re great"'
cat(x)
```

```
## Tony says "they're great"
```

```
x <- 'Tony says\t"they\'re great"'
cat(x)
```

```
## Tony says    "they're great"
```

Basic properties of strings

How long is a string?

```
x <- 'Tony says\t"they\'re great"'
nchar(x)
```

```
## [1] 25
```

```
x = 'Tony says\t"they\'re great"'
len(x)
```

```
## 25
```

Basic operations on strings

An example string

Let's start by reading in a poem as a single string.

```
library(readr)
poem <- read_file("the_tiger.txt")
poem
```

```
## [1] "THE TIGER\n\n\nTiger, tiger, burning bright\nIn the forests of the night,\nWhat immortal hand or eye\nCould
```

```
with open("the_tiger.txt", "r") as f:
    poem = f.read()
poem
```

```
## 'THE TIGER\n\n\nTiger, tiger, burning bright\nIn the forests of the night,\nWhat immortal hand or eye\nCould fra
```

Splitting strings

Often we want to split one string (or several strings) into shorter strings according to a certain pattern. In our poem, we can split this into lines using the newline character \n

```r
library(stringr)
lines <- str_split(poem,"\n")[[1]]
lines[1:5]
```

```
## [1] "THE TIGER"                      ""
## [3] ""                               "Tiger, tiger, burning bright"
## [5] "In the forests of the night,"
```

```python
lines = poem.split("\n")
lines[:5]
```

```
## ['THE TIGER', '', '', 'Tiger, tiger, burning bright', 'In the forests of the night,']
```

What if we want to split into stanzas (verses)? What if we want to split each line into words? How could we tell how many words are on each line?

Joining strings

Sometimes we want to combine strings into a single string, for this we use `str_c` or `paste`

```
poem_start <- str_c(lines[1:5], collapse="\n")
poem_start
```

```
## [1] "THE TIGER\n\n\nTiger, tiger, burning bright\nIn the forests of the night,"
```

```
poem_start = "\n".join(lines[:5])
poem_start
```

```
## 'THE TIGER\n\n\nTiger, tiger, burning bright\nIn the forests of the night,'
```

Replacing parts of strings

Sometimes we want to replace occurences of a certain string or sequence of characters with something or nothing.

```r
new_poem <- str_replace_all(poem, "tiger", "lion")
new_poem
```

```
## [1] "THE TIGER\n\n\nTiger, lion, burning bright\nIn the forests of the night,\nWhat immortal hand or eye\nCould
```

```python
new_poem = poem.replace("tiger", "lion")
new_poem
```

```
## 'THE TIGER\n\n\nTiger, lion, burning bright\nIn the forests of the night,\nWhat immortal hand or eye\nCould fram
```

Cleaning whitespace from strings

Often, when we parse strings (especially from html), we include a bunch of whitespace at the beginnings and ends of strings. `str_trim()`, or `strip()` in Python, helps us get rid of these.

```
messy_lines <- c(" Hello world   ", " How are you ")
str_trim(messy_lines)
```

```
## [1] "Hello world" "How are you"
```

```
messy_lines = [" Hello world   ", " How are you "]
[x.strip() for x in messy_lines]
```

```
## ['Hello world', 'How are you']
```

Regex

Why regex

Regex (Regular expressions) offers a way of manipulating strings based on powerful pattern matching. Patterns are defined sequences of characters or character types.

We can use these patterns to **detect, locate, extract, match, replace**, and **split** strings.

Regex allows us to allow for variations in spelling, e.g. colour/color

```
pattern <- "colou?r"
```

```
pattern = "colou?r"
```

Quantifiers

The first fancy REGEX thing we have seen is a quantifier. It defines how often we need to see a character in order to match. In our example below, it says "u" must appear 0 or 1 times

```r
pattern <- "colou?r"
strings <- c("color", "colour")
str_detect(strings, pattern)
```

```
## [1] TRUE TRUE
```

```python
import re
pattern = "colou?r"
strings = ["color", "colour"]
[re.match(pattern,x) for x in strings]
```

```
## [<re.Match object; span=(0, 5), match='color'>, <re.Match object; span=(0, 6), match='colour'>]
```

Other useful quantifiers are * (0 or more times), + (1 or more times), {m} exactly m times, {m,n} from m to n times. See here for a stringr and regex cheatsheet, and here for an online python regex explorer and cheatsheet.

Special characters

. will match any character

```
strings <- c("digitalise", "digitalize")
str_detect(strings, "digitali.e")
```

```
## [1] TRUE TRUE
```

^ matches the beginning of a string, and $ matches the end of a string

```
strings <- c("hello world", "Hi, hello")
str_detect(strings, "^hello")
```

```
## [1]  TRUE FALSE
```

Lists of characters

Square brackets indicate a list of possible character matches

- `[sz]` matches either an s or a z (we can also use the `|` symbol as an or operator, so `s|z` is equivalent)
- `[0-5]` matches any digit from 0 to 5
- `[a-f]` matches any lower case letter from a to f
- `[G-Z]` matches any lower case letter from G to Z

Exercise

Download a list of 1,000 tweets which mention net zero. Use regular expressions to find dates in the future. How many tweets mention a date? If we extract the first date using `str_extract()`, what dates are most common?

Shortcuts to character lists

We also have some shortcuts that represent character lists or types. The upper case letter is often the negated version of the lower case letter

- `\b` word boundary
- `\B` not-word boundary
- `\d` digit
- `\D` non-digit
- `\s` whitespace
- `\S` non-whitespace
- `\w` alphanumeric
- `\W` non-alphanumeric

`[:punct:]` is an additional useful pattern for punctuation (R also uses this representation of these character types)

Groups

By using brackets `()` we can specify parts of a pattern which we want to extract

```
estimates <- c("0.039", "0.042", "-0.003*", "49", "1.3*")
pattern <- "(-*[0-9]*\\.*[0-9]*)\\*"
str_match(estimates, pattern)[,2]
```

```
## [1] NA      NA      "0.003" NA      "1.3"
```

What else can we use regex for

Anything that follows a pattern that we want to detect, or extract a part of. URLs, email addresses, telefone numbers, speakers in a transcript, party names...

Exercise

QJE.txt contains the opening two paragraphs of a recent paper in the Quarterly Journal of Economics. We are interested in finding out what papers are cited. Using what we have covered in this session, can you extract a list of first author, year pairs that represent each citation made in the text?

Outlook

Next week

Next week we'll be brushing up on our ggplot skills, and putting them to use in plotting text data