Intro to Regular Expressions

Richard Mills

Outline

Introduction

Working with file-names

Using patterns within code

Wrap up

Introduction

What are regular expressions?

- Also known as *regex*.

What are regular expressions?

- Also known as *regex*.
- The concept dates back to the 1950s.

What are regular expressions?

- Also known as *regex*.
- The concept dates back to the 1950s.
- A sequence of characters that define a search *pattern* for text.

What are regular expressions?

- Also known as *regex*.
- The concept dates back to the 1950s.
- A sequence of characters that define a search *pattern* for text.
- Frequently used for:
  - Checking to see if part/all of a string meets some criteria.

What are regular expressions?

- Also known as *regex*.
- The concept dates back to the 1950s.
- A sequence of characters that define a search *pattern* for text.
- Frequently used for:
  - Checking to see if part/all of a string meets some criteria.
  - Perform (advanced) find-and-replace operations within a string.

What are regular expressions?

- Also known as *regex*.
- The concept dates back to the 1950s.
- A sequence of characters that define a search *pattern* for text.
- Frequently used for:
  - Checking to see if part/all of a string meets some criteria.
  - Perform (advanced) find-and-replace operations within a string.
  - Commonly used in compilers/interpreters for parsing user code.

What are regular expressions?

- Also known as *regex*.
- The concept dates back to the 1950s.
- A sequence of characters that define a search *pattern* for text.
- Frequently used for:
  - Checking to see if part/all of a string meets some criteria.
  - Perform (advanced) find-and-replace operations within a string.
  - Commonly used in compilers/interpreters for parsing user code.
  - Input validation for websites.

What are regular expressions?

- Also known as *regex*.
- The concept dates back to the 1950s.
- A sequence of characters that define a search *pattern* for text.
- Frequently used for:
  - Checking to see if part/all of a string meets some criteria.
  - Perform (advanced) find-and-replace operations within a string.
  - Commonly used in compilers/interpreters for parsing user code.
  - Input validation for websites.
  - Frankly... Useful any time when working with text data!

Wildcards

- Certain Excel formulae allow the use of *wildcards* (ie. regex-lite) when matching against text.

Wildcards

- Certain Excel formulae allow the use of *wildcards* (ie. regex-lite) when matching against text.
  - Count number of strings with only 3 characters:

```
=COUNTIF(A1:A100, "???")
```

Wildcards

- Certain Excel formulae allow the use of *wildcards* (ie. regex-lite) when matching against text.
  - Count number of strings with only 3 characters:

    `=COUNTIF(A1:A100, "???")`
  - Count all strings beginning with `ABC`:

    `=COUNTIF(A1:A100, "ABC*")`

Wildcards

- Certain Excel formulae allow the use of *wildcards* (ie. regex-lite) when matching against text.
  - Count number of strings with only 3 characters:

    `=COUNTIF(A1:A100, "???")`
  - Count all strings beginning with `ABC`:

    `=COUNTIF(A1:A100, "ABC*")`
  - `?` represents any single character.

Wildcards

- Certain Excel formulae allow the use of *wildcards* (ie. regex-lite) when matching against text.
  - Count number of strings with only 3 characters:

    `=COUNTIF(A1:A100, "???")`
  - Count all strings beginning with `ABC`:

    `=COUNTIF(A1:A100, "ABC*")`
  - `?` represents any single character.
  - `*` represents zero or more of any character.

Wildcards

- Certain Excel formulae allow the use of *wildcards* (ie. regex-lite) when matching against text.
  - Count number of strings with only 3 characters:

    `=COUNTIF(A1:A100, "???")`
  - Count all strings beginning with `ABC`:

    `=COUNTIF(A1:A100, "ABC*")`
  - `?` represents any single character.
  - `*` represents zero or more of any character.
- Other supporting functions include `MATCH`, `VLOOKUP`, `SUMIF` and `SUMIFS`.

Wildcards

- Certain Excel formulae allow the use of *wildcards* (ie. regex-lite) when matching against text.
  - Count number of strings with only 3 characters:

    `=COUNTIF(A1:A100, "???")`

  - Count all strings beginning with `ABC`:

    `=COUNTIF(A1:A100, "ABC*")`

  - `?` represents any single character.
  - `*` represents zero or more of any character.
- Other supporting functions include `MATCH`, `VLOOKUP`, `SUMIF` and `SUMIFS`.
- There is similar functionality in Access via the `LIKE` operator.

Wildcards

- Certain Excel formulae allow the use of *wildcards* (ie. regex-lite) when matching against text.
  - Count number of strings with only 3 characters:

    `=COUNTIF(A1:A100, "???")`
  - Count all strings beginning with `ABC`:

    `=COUNTIF(A1:A100, "ABC*")`
  - `?` represents any single character.
  - `*` represents zero or more of any character.
- Other supporting functions include `MATCH`, `VLOOKUP`, `SUMIF` and `SUMIFS`.
- There is similar functionality in Access via the `LIKE` operator.

  For example... `TABLE_NUMBER IS LIKE "73?"`

Working with file-names

What's the issue?

- Suppose we have the following files within a folder:

What's the issue?

- Suppose we have the following files within a folder:

```
TOP_SECRET_DATA_01-03-2020.CSV
TOP_SECRET_DATA_20-03-2020.CSV
TOP_SECRET_DATA_25-04-2020.CSV
```

What's the issue?

- Suppose we have the following files within a folder:

  `TOP_SECRET_DATA_01-03-2020.CSV`
  `TOP_SECRET_DATA_20-03-2020.CSV`
  `TOP_SECRET_DATA_25-04-2020.CSV`

- The date within the file-name gives us the effective date of the data within.

What's the issue?

- Suppose we have the following files within a folder:

  `TOP_SECRET_DATA_01-03-2020.CSV`
  `TOP_SECRET_DATA_20-03-2020.CSV`
  `TOP_SECRET_DATA_25-04-2020.CSV`

- The date within the file-name gives us the effective date of the data within.
- **How could we extract those dates?**

What's the issue?

- **What about now?**

```
TOP_SECRET_DATA_01-03-2020.CSV
TOP_SECRET_DATA_V2_20-03-2020.CSV
TOP_SECRET_DATA_Adj_25-04-2020.CSV
```

Possible solution

If we were *particularly* determined, we could use the following pseudo-code:

Possible solution

If we were *particularly* determined, we could use the following pseudo-code:

```
for i from 1 to LEN(string)-10:
```

Possible solution

If we were *particularly* determined, we could use the following pseudo-code:

```
for i from 1 to LEN(string)-10:
  if is_digit(string[i]) and is_digit(string[i+1]):
```

Possible solution

If we were *particularly* determined, we could use the following pseudo-code:

```
for i from 1 to LEN(string)-10:
  if is_digit(string[i]) and is_digit(string[i+1]):
    if string[i+2] = "-":
```

Possible solution

If we were *particularly* determined, we could use the following pseudo-code:

```
for i from 1 to LEN(string)-10:
  if is_digit(string[i]) and is_digit(string[i+1]):
    if string[i+2] = "-":
      if is_digit(string[i+3]) and is_digit(string[i+4]):
```

Possible solution

If we were *particularly* determined, we could use the following pseudo-code:

```
for i from 1 to LEN(string)-10:
  if is_digit(string[i]) and is_digit(string[i+1]):
    if string[i+2] = "-":
      if is_digit(string[i+3]) and is_digit(string[i+4]):
        if string[i+5] = "-":
```

Possible solution

If we were *particularly* determined, we could use the following pseudo-code:

```
for i from 1 to LEN(string)-10:
  if is_digit(string[i]) and is_digit(string[i+1]):
    if string[i+2] = "-":
      if is_digit(string[i+3]) and is_digit(string[i+4]):
        if string[i+5] = "-":
          if is_digit(string[i+6]) ....  :
```

Possible solution

If we were *particularly* determined, we could use the following pseudo-code:

```
for i from 1 to LEN(string)-10:
  if is_digit(string[i]) and is_digit(string[i+1]):
    if string[i+2] = "-":
      if is_digit(string[i+3]) and is_digit(string[i+4]):
        if string[i+5] = "-":
          if is_digit(string[i+6]) ....  :
            date_part = string[i:i+10]
```

Possible solution

If we were *particularly* determined, we could use the following pseudo-code:

```
for i from 1 to LEN(string)-10:
  if is_digit(string[i]) and is_digit(string[i+1]):
    if string[i+2] = "-":
      if is_digit(string[i+3]) and is_digit(string[i+4]):
        if string[i+5] = "-":
          if is_digit(string[i+6]) ....  :
            date_part = string[i:i+10]
            return date_part
```

Possible solution

If we were *particularly* determined, we could use the following pseudo-code:

```
for i from 1 to LEN(string)-10:
  if is_digit(string[i]) and is_digit(string[i+1]):
    if string[i+2] = "-":
      if is_digit(string[i+3]) and is_digit(string[i+4]):
        if string[i+5] = "-":
          if is_digit(string[i+6]) ....  :
            date_part = string[i:i+10]
            return date_part

error ‘No date!’
```

Possible solution

If we were *particularly* determined, we could use the following pseudo-code:

```
for i from 1 to LEN(string)-10:
  if is_digit(string[i]) and is_digit(string[i+1]):
    if string[i+2] = "-":
      if is_digit(string[i+3]) and is_digit(string[i+4]):
        if string[i+5] = "-":
          if is_digit(string[i+6]) ....  :
            date_part = string[i:i+10]
            return date_part

error 'No date!'
```

Is there an alternative way?

Possible solution

- In *regex speak* we are looking for the following pattern:

```
\d\d-\d\d-\d\d\d\d
```

Possible solution

- In *regex speak* we are looking for the following pattern:

```
\d\d-\d\d-\d\d\d\d
```

- Where each `\d` corresponds to a single digit (0-9).

Possible solution

- In *regex speak* we are looking for the following pattern:

```
\d\d-\d\d-\d\d\d\d
```

- Where each `\d` corresponds to a single digit (0-9).
- This would match any of the dates contained within the above file names.

Possible solution

- In *regex speak* we are looking for the following pattern:

```
\d\d-\d\d-\d\d\d\d
```

- Where each `\d` corresponds to a single digit (0-9).
- This would match any of the dates contained within the above file names.
- However, it *could* leave to false positives, eg:
  - `RANDOM_00-01-0002.CSV`

Possible solution

- In *regex speak* we are looking for the following pattern:

```
\d\d-\d\d-\d\d\d\d
```

- Where each `\d` corresponds to a single digit (0-9).
- This would match any of the dates contained within the above file names.
- However, it *could* leave to false positives, eg:
  - `RANDOM_00-01-0002.CSV`
  - `1234-56-78901234.CSV`

Possible solution

- In *regex speak* we are looking for the following pattern:

```
\d\d-\d\d-\d\d\d\d
```

- Where each `\d` corresponds to a single digit (0-9).
- This would match any of the dates contained within the above file names.
- However, it *could* leave to false positives, eg:
  - `RANDOM_00-01-0002.CSV`
  - `1234-56-78901234.CSV`
- **Can we refine it?**

Possible solution

- Previously we used `\d` to match a single digit.

Possible solution

- Previously we used `\d` to match a single digit.
- We could instead use a **character class** to restrict this behaviour.

Possible solution

- Previously we used `\d` to match a single digit.
- We could instead use a **character class** to restrict this behaviour.
- This allows us to specify a list (or range) of permitted characters.

Possible solution

- Previously we used `\d` to match a single digit.
- We could instead use a **character class** to restrict this behaviour.
- This allows us to specify a list (or range) of permitted characters.
- As an example, we'll assume that:

Possible solution

- Previously we used `\d` to match a single digit.
- We could instead use a **character class** to restrict this behaviour.
- This allows us to specify a list (or range) of permitted characters.
- As an example, we'll assume that:
  - The *day* part can start with any of `0`, `1`, `2` or `3`.

Possible solution

- Previously we used `\d` to match a single digit.
- We could instead use a **character class** to restrict this behaviour.
- This allows us to specify a list (or range) of permitted characters.
- As an example, we'll assume that:
  - The *day* part can start with any of `0`, `1`, `2` or `3`.
  - The *month* part can start with either `0` or `1`.

Possible solution

- Previously we used `\d` to match a single digit.
- We could instead use a **character class** to restrict this behaviour.
- This allows us to specify a list (or range) of permitted characters.
- As an example, we'll assume that:
  - The *day* part can start with any of `0`, `1`, `2` or `3`.
  - The *month* part can start with either `0` or `1`.
  - The *year* part will start with either `19` or `20`.

Possible solution

- Previously we used `\d` to match a single digit.
- We could instead use a **character class** to restrict this behaviour.
- This allows us to specify a list (or range) of permitted characters.
- As an example, we'll assume that:
  - The *day* part can start with any of `0`, `1`, `2` or `3`.
  - The *month* part can start with either `0` or `1`.
  - The *year* part will start with either `19` or `20`.

- We can update our pattern to be:

`[0-3]\d-[01]\d-(19|20)\d\d`

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d\d
```

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d\d
```

- Note the use of the following elements:

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d\d
```

- Note the use of the following elements:
  - `[0-3]` which matches against any of `0`, `1`, `2` or `3`.

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d\d
```

- Note the use of the following elements:
  - `[0-3]` which matches against any of `0`, `1`, `2` or `3`.
  - `[01]` which matches either `0` or `1`.

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d\d
```

- Note the use of the following elements:
  - `[0-3]` which matches against any of `0`, `1`, `2` or `3`.
  - `[01]` which matches either `0` or `1`.
  - `(19|20)` which matches either `19` or `20`.

    Note the use of `(` and `)` above.

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d\d
```

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d\d
```

- Note the 2x `\d` at the end.

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d\d
```

- Note the 2x `\d` at the end.
- We could instead use a **quantifier** to remove duplication:

```
[0-3]\d-[01]\d-(19|20)\d{2}
```

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d\d
```

- Note the 2x `\d` at the end.
- We could instead use a **quantifier** to remove duplication:

```
[0-3]\d-[01]\d-(19|20)\d{2}
```

- Both approaches are equivalent!

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d{2}
```

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d{2}
```

- **What if the day and month components didn't always have a leading 0?**

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d{2}
```

- **What if the day and month components didn't always have a leading 0?**
  - `TOP_SECRET_DATA_1-03-2020.CSV`
  - `TOP_SECRET_DATA_V2_20-3-2020.CSV`
  - `TOP_SECRET_DATA_Adj_5-4-2020.CSV`

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d{2}
```

- **What if the day and month components didn't always have a leading 0?**
  - `TOP_SECRET_DATA_1-03-2020.CSV`
  - `TOP_SECRET_DATA_V2_20-3-2020.CSV`
  - `TOP_SECRET_DATA_Adj_5-4-2020.CSV`

- We could use `?` which is another type of quantifier.

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d{2}
```

- **What if the day and month components didn't always have a leading 0?**
  - `TOP_SECRET_DATA_1-03-2020.CSV`
  - `TOP_SECRET_DATA_V2_20-3-2020.CSV`
  - `TOP_SECRET_DATA_Adj_5-4-2020.CSV`

- We could use `?` which is another type of quantifier.
  - This matches the prior element 0 **or** 1 times.

Possible solution

```
[0-3]\d-[01]\d-(19|20)\d{2}
```

- **What if the day and month components didn't always have a leading 0?**
  - `TOP_SECRET_DATA_1-03-2020.CSV`
  - `TOP_SECRET_DATA_V2_20-3-2020.CSV`
  - `TOP_SECRET_DATA_Adj_5-4-2020.CSV`

- We could use `?` which is another type of quantifier.
  - This matches the prior element 0 **or** 1 times.

```
[0-3]?\d-[01]?\d-(19|20)\d{2}
```

Possible solution

- The last change we *might* want to make is to ensure that either end of a potential date does **not** touch a digit.

Possible solution

- The last change we *might* want to make is to ensure that either end of a potential date does **not** touch a digit.
- For this, we can use:

Possible solution

- The last change we *might* want to make is to ensure that either end of a potential date does **not** touch a digit.
- For this, we can use:
  - `\D` to match a non-digit character.

Possible solution

- The last change we *might* want to make is to ensure that either end of a potential date does **not** touch a digit.
- For this, we can use:
  - `\D` to match a non-digit character.
  - `(?<=...)` to match an element immediately preceding our date.

Possible solution

- The last change we *might* want to make is to ensure that either end of a potential date does **not** touch a digit.
- For this, we can use:
  - `\D` to match a non-digit character.
  - `(?<=...)` to match an element immediately preceding our date.
  - `(?=...)` to match an element immediately after our date.

Possible solution

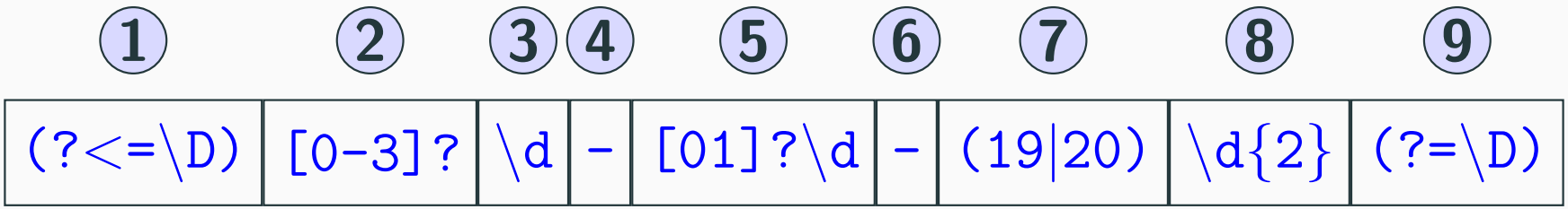
- The last change we *might* want to make is to ensure that either end of a potential date does **not** touch a digit.
- For this, we can use:
  - `\D` to match a non-digit character.
  - `(?<=...)` to match an element immediately preceding our date.
  - `(?=...)` to match an element immediately after our date.
- Our final proposed pattern is:

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

Possible solution

①	②	③	④	⑤	⑥	⑦	⑧	⑨
`(?<=\D)`	`[0-3]?`	`\d`	`-`	`[01]?\d`	`-`	`(19|20)`	`\d{2}`	`(?=\D)`

Possible solution



1 The character immediately before our date must be a non-digit

Possible solution

①	②	③	④	⑤	⑥	⑦	⑧	⑨
`(?<=\D)`	`[0-3]?`	`\d`	`-`	`[01]?\d`	`-`	`(19|20)`	`\d{2}`	`(?=\D)`

1 The character immediately before our date must be a non-digit

2 The **day** part must start with an optional 0, 1, 2 or 3

Possible solution

①	②	③	④	⑤	⑥	⑦	⑧	⑨
`(?<=\D)`	`[0-3]?`	`\d`	`-`	`[01]?\d`	`-`	`(19|20)`	`\d{2}`	`(?=\D)`

1 The character immediately before our date must be a non-digit

2 The **day** part must start with an optional 0, 1, 2 or 3

3 . . . followed by a 0–9

Possible solution

①	②	③	④	⑤	⑥	⑦	⑧	⑨
`(?<=\D)`	`[0-3]?`	`\d`	`-`	`[01]?\d`	`-`	`(19|20)`	`\d{2}`	`(?=\D)`

1 The character immediately before our date must be a non-digit

2 The **day** part must start with an optional 0, 1, 2 or 3

3 . . . followed by a 0–9

4 . . . followed by a –

Possible solution

①	②	③	④	⑤	⑥	⑦	⑧	⑨
`(?<=\D)`	`[0-3]?`	`\d`	`-`	`[01]?\d`	`-`	`(19|20)`	`\d{2}`	`(?=\D)`

1 The character immediately before our date must be a non-digit

2 The **day** part must start with an optional 0, 1, 2 or 3

3 . . . followed by a 0–9

4 . . . followed by a –

5 . . . followed by the **month** part which can start with an optional 0 or 1

Possible solution

①	②	③	④	⑤	⑥	⑦	⑧	⑨
`(?<=\D)`	`[0-3]?`	`\d`	`-`	`[01]?\d`	`-`	`(19|20)`	`\d{2}`	`(?=\D)`

1 The character immediately before our date must be a non-digit

2 The **day** part must start with an optional 0, 1, 2 or 3

3 . . . followed by a 0–9

4 . . . followed by a –

5 . . . followed by the **month** part which can start with an optional 0 or 1

6 . . . followed by a –

Possible solution

①	②	③	④	⑤	⑥	⑦	⑧	⑨
`(?<=\D)`	`[0-3]?`	`\d`	`-`	`[01]?\d`	`-`	`(19|20)`	`\d{2}`	`(?=\D)`

1 The character immediately before our date must be a non-digit

2 The **day** part must start with an optional 0, 1, 2 or 3

3 . . . followed by a 0–9

4 . . . followed by a –

5 . . . followed by the **month** part which can start with an optional 0 or 1

6 . . . followed by a –

7 . . . followed by the **year** part which must start with 19 or 20

Possible solution

①	②	③	④	⑤	⑥	⑦	⑧	⑨
`(?<=\D)`	`[0-3]?`	`\d`	`-`	`[01]?\d`	`-`	`(19|20)`	`\d{2}`	`(?=\D)`

1 The character immediately before our date must be a non-digit

2 The **day** part must start with an optional 0, 1, 2 or 3

3 . . . followed by a 0–9

4 . . . followed by a –

5 . . . followed by the **month** part which can start with an optional 0 or 1

6 . . . followed by a –

7 . . . followed by the **year** part which must start with 19 or 20

8 . . . followed by 2x digits

Possible solution

①	②	③	④	⑤	⑥	⑦	⑧	⑨
`(?<=\D)`	`[0-3]?`	`\d`	`-`	`[01]?\d`	`-`	`(19|20)`	`\d{2}`	`(?=\D)`

1 The character immediately before our date must be a non-digit

2 The **day** part must start with an optional 0, 1, 2 or 3

3 . . . followed by a 0–9

4 . . . followed by a –

5 . . . followed by the **month** part which can start with an optional 0 or 1

6 . . . followed by a –

7 . . . followed by the **year** part which must start with 19 or 20

8 . . . followed by 2x digits

9 The character immediately after our date must be a non-digit

Possible solution

Below shows the ‘evolution’ of our pattern:

```
\d\d-\d\d-\d\d\d\d
```

Possible solution

Below shows the 'evolution' of our pattern:

```
\d\d-\d\d-\d\d\d\d
```

```
[0-3]\d-[01]\d-(19|20)\d\d
```

Possible solution

Below shows the 'evolution' of our pattern:

```
\d\d-\d\d-\d\d\d\d
```

```
[0-3]\d-[01]\d-(19|20)\d\d
```

```
[0-3]\d-[01]\d-(19|20)\d{2}
```

Possible solution

Below shows the 'evolution' of our pattern:

```
\d\d-\d\d-\d\d\d\d
```

```
[0-3]\d-[01]\d-(19|20)\d\d
```

```
[0-3]\d-[01]\d-(19|20)\d{2}
```

```
[0-3]?\d-[01]?\d-(19|20)\d{2}
```

Possible solution

Below shows the 'evolution' of our pattern:

```
\d\d-\d\d-\d\d\d\d
```

```
[0-3]\d-[01]\d-(19|20)\d\d
```

```
[0-3]\d-[01]\d-(19|20)\d{2}
```

```
[0-3]?\d-[01]?\d-(19|20)\d{2}
```

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- This pattern will successfully match against (for example):

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- This pattern will successfully match against (for example):
  - `TOP_SECRET_DATA_01-03-2020.CSV`

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- This pattern will successfully match against (for example):
  - `TOP_SECRET_DATA_01-03-2020.CSV`
  - `TOP_SECRET_DATA_1-03-2020.CSV`

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- This pattern will successfully match against (for example):
  - `TOP_SECRET_DATA_01-03-2020.CSV`
  - `TOP_SECRET_DATA_1-03-2020.CSV`
  - `TOP_SECRET_DATA_V2_20-3-2020.CSV`

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- This pattern will successfully match against (for example):
  - `TOP_SECRET_DATA_01-03-2020.CSV`
  - `TOP_SECRET_DATA_1-03-2020.CSV`
  - `TOP_SECRET_DATA_V2_20-3-2020.CSV`
  - `TOP_SECRET_DATA_5-4-1999_Adj.CSV`

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- Given the above pattern... **Which of the following would be successfully matched?**

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- Given the above pattern... **Which of the following would be successfully matched?**
  - `OUR_DATA_FILE_01-03-2120.CSV`

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- Given the above pattern... **Which of the following would be successfully matched?**
  - `OUR_DATA_FILE_01-03-2120.CSV`

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- Given the above pattern... **Which of the following would be successfully matched?**
  - **OUR_DATA_FILE_01-03-2120.CSV**
  - SKETCHY_INPUTS_V31-10-1999.CSV

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- Given the above pattern... **Which of the following would be successfully matched?**
  - `OUR_DATA_FILE_01-03-2120.CSV`
  - `SKETCHY_INPUTS_V31-10-1999.CSV`

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- Given the above pattern... **Which of the following would be successfully matched?**
  - `OUR_DATA_FILE_01-03-2120.CSV`
  - `SKETCHY_INPUTS_V31-10-1999.CSV`
  - `20-10-1999_NO_PEEKING.TXT`

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- Given the above pattern... **Which of the following would be successfully matched?**
  - `OUR_DATA_FILE_01-03-2120.CSV`
  - `SKETCHY_INPUTS_V31-10-1999.CSV`
  - `20-10-1999_NO_PEEKING.TXT`

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- Given the above pattern... **Which of the following would be successfully matched?**
  - OUR_DATA_FILE_01-03-2120.CSV
  - SKETCHY_INPUTS_V31-10-1999.CSV
  - 20-10-1999_NO_PEEKING.TXT
  - NO10_PARTY_INVITES_31_10_1999.CSV

Possible solution

```
(?<=\D)[0-3]?\d-[01]?\d-(19|20)\d{2}(?=\D)
```

- Given the above pattern... **Which of the following would be successfully matched?**
  - `OUR_DATA_FILE_01-03-2120.CSV`
  - `SKETCHY_INPUTS_V31-10-1999.CSV`
  - `20-10-1999_NO_PEEKING.TXT`
  - `NO10_PARTY_INVITES_31_10_1999.CSV`

Using patterns within code

How do we actually use this pattern?

- Many programming languages provide *Regex* functionality.

How do we actually use this pattern?

- Many programming languages provide *Regex* functionality.
  - In VBA via the `Microsoft VBScript Regular Expressions 5.5` reference.

How do we actually use this pattern?

- Many programming languages provide *Regex* functionality.
  - In VBA via the `Microsoft VBScript Regular Expressions 5.5` reference.
  - In Python via the `re` library.

How do we actually use this pattern?

- Many programming languages provide *Regex* functionality.
  - In VBA via the `Microsoft VBScript Regular Expressions 5.5` reference.
  - In Python via the `re` library.
  - In R via the `stringr` library.

How do we actually use this pattern?

- Many programming languages provide *Regex* functionality.
  - In VBA via the `Microsoft VBScript Regular Expressions 5.5` reference.
  - In Python via the `re` library.
  - In R via the `stringr` library.
- However, there *may* be some differences in the respective implementations.

How could we do this within R?

- Suppose we have some `.CSV` files within a folder.

How could we do this within R?

- Suppose we have some `.CSV` files within a folder.
- Let's assume that the variable `FOLDER_PATH` contains the path.

How could we do this within R?

- Suppose we have some `.CSV` files within a folder.
- Let's assume that the variable `FOLDER_PATH` contains the path.
- Within our code, we can list all of the files in the folder...

How could we do this within R?

- Suppose we have some `.CSV` files within a folder.
- Let's assume that the variable `FOLDER_PATH` contains the path.
- Within our code, we can list all of the files in the folder...
- ...and then extract the dates from those files with a valid file name.

A worked example in R

```
tibble::tibble(
  FILE_NAME =
    fs::dir_ls(
      path = FOLDER_PATH,
      regexp = '(?i)CSV$'  # ...another pattern!
    ),

  DATE_PART =
    stringr::str_extract(
      FILE_NAME,
      pattern =
'(?<=\\D)[0-3]?\\d-[01]?\\d-(19|20)\\d{2}(?=\\D)'
    )
)
```

A worked example in R

Suppose our folder contained the following files:

```
20_10_1999_NO_PEEKING.CSV
NO10_PARTY_INVITES_31-10_1999.CSV
OUR_DATA_FILE_01-03-2120.csv
SKETCHY_INPUTS_V31-10-1999.CSV
TOP_SECRET_DATA_01-03-2020.CSV
TOP_SECRET_DATA_1-03-2020.CSV
TOP_SECRET_DATA_5-4-1999_Adj.CSV
TOP_SECRET_DATA_V2_20-3-2020.CSV
NOT_A_CSV.TXT
```

A worked example in R

Below shows the output from our code on the above folder:

FILE_NAME	DATE_PART
20_10_1999_NO_PEEKING.CSV	NA
NO10_PARTY_INVITES_31-10_1999.CSV	NA
OUR_DATA_FILE_01-03-2120.csv	NA
SKETCHY_INPUTS_V31-10-1999.CSV	31-10-1999
TOP_SECRET_DATA_01-03-2020.CSV	01-03-2020
TOP_SECRET_DATA_1-03-2020.CSV	1-03-2020
TOP_SECRET_DATA_5-4-1999_Adj.CSV	5-4-1999
TOP_SECRET_DATA_V2_20-3-2020.CSV	20-3-2020

A worked example in R

- If we *pipe* our `DATE_PART` into `lubridate::dmy`, we can convert our extracted date into an actual date object that we can more easily work with.

A worked example in R

- If we *pipe* our `DATE_PART` into `lubridate::dmy`, we can convert our extracted date into an actual date object that we can more easily work with.
- We can also use `dplyr::filter` to only retain those file names with valid dates.

A worked example in R

- If we *pipe* our `DATE_PART` into `lubridate::dmy`, we can convert our extracted date into an actual date object that we can more easily work with.
- We can also use `dplyr::filter` to only retain those file names with valid dates.

```
tibble::tibble(
  ...
  DATE_PART =
    ... %>%
    lubridate::dmy()
) %>%
dplyr::filter(
  !is.na(DATE_PART)
)
```

A worked example in R

Our revised output is shown below:

FILE_NAME	DATE_PART
SKETCHY_INPUTS_V31-10-1999.CSV	1999-10-31
TOP_SECRET_DATA_01-03-2020.CSV	2020-03-01
TOP_SECRET_DATA_1-03-2020.CSV	2020-03-01
TOP_SECRET_DATA_5-4-1999_Adj.CSV	1999-04-05
TOP_SECRET_DATA_V2_20-3-2020.CSV	2020-03-20

Wrap up

What can I take away from this?

- That regex patterns can be used to match characters within some wider text.

What can I take away from this?

- That regex patterns can be used to match characters within some wider text.
- Using various elements within our pattern, we can refine what we are looking for.

What can I take away from this?

- That regex patterns can be used to match characters within some wider text.
- Using various elements within our pattern, we can refine what we are looking for.
- We are not just restricted to numbers!

What didn't we see?

- In our example above, we could have used *capture groups* to extract the individual day, month and year parts.

What didn't we see?

- In our example above, we could have used *capture groups* to extract the individual day, month and year parts.
- ...these can also be used to make a pattern dependent on what has already been matched.

What didn't we see?

- In our example above, we could have used *capture groups* to extract the individual day, month and year parts.
- ...these can also be used to make a pattern dependent on what has already been matched.
- Additional *quantifiers* such as `+` and `*`.

What didn't we see?

- In our example above, we could have used *capture groups* to extract the individual day, month and year parts.
- ...these can also be used to make a pattern dependent on what has already been matched.
- Additional *quantifiers* such as `+` and `*`.
- Using *anchors* such as `^` and `$` to ensure that matches begin and/or finish at the start/end of a string respectively.

What didn't we see?

- In our example above, we could have used *capture groups* to extract the individual day, month and year parts.
- ...these can also be used to make a pattern dependent on what has already been matched.
- Additional *quantifiers* such as `+` and `*`.
- Using *anchors* such as `^` and `$` to ensure that matches begin and/or finish at the start/end of a string respectively.
- Many other character classes such as `\w`, `\s` and `\n`.

What didn't we see?

- In our example above, we could have used *capture groups* to extract the individual day, month and year parts.
- ...these can also be used to make a pattern dependent on what has already been matched.
- Additional *quantifiers* such as `+` and `*`.
- Using *anchors* such as `^` and `$` to ensure that matches begin and/or finish at the start/end of a string respectively.
- Many other character classes such as `\w`, `\s` and `\n`.
- *Flags*; for example `(?i)` makes a pattern case-insensitive.

Useful resources

For those wanting to know more:

Useful resources

For those wanting to know more:

- https://regexone.com/

Useful resources

For those wanting to know more:

- https://regexone.com/
- https://unicode-org.github.io/icu/userguide/strings/regexp.html

Useful resources

For those wanting to know more:

- https://regexone.com/
- https://unicode-org.github.io/icu/userguide/strings/regexp.html
- https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285

Useful resources

For those wanting to know more:

- https://regexone.com/
- https://unicode-org.github.io/icu/userguide/strings/regexp.html
- https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285
- Speak to me.

That's it!

Any questions?

. . . comments?