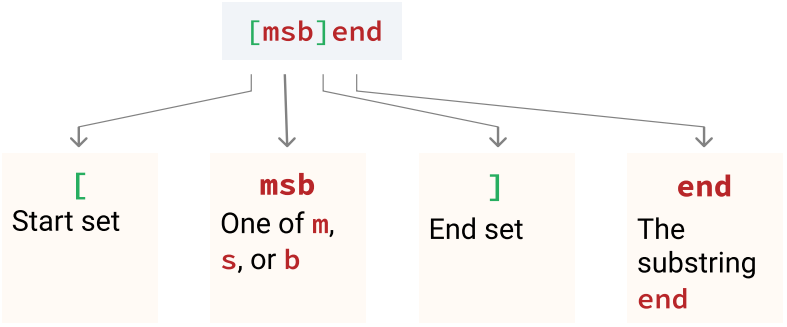
Benefits of Regex:

- complex operations with string data can be written a lot quicker, which will save you time.
- Regular expressions are often faster to execute than their manual equivalents.
- Regular expressions are supported in almost every modern programming language, as well as other places like command line utilities and databases. Understanding regular expressions gives you a powerful tool that you can use wherever you work with data.

Don't memorize syntax !!

- One thing to keep in mind before we start: don't expect to remember all of the regular expression syntax. The most important thing is to understand the core principles, what is possible, and where to look up the details. This will mean you can quickly jog your memory whenever you need regular expressions.
- As long as you are able to write and understand regular expressions with the help of documentation and/or other reference guides, you have all the skills you need to excel.

- When working with regular expressions, we use the term pattern to describe a regular expression that we've written. If the pattern is found within the string we're searching, we say that it has matched.
- We previously used regular expressions with pandas, but Python also has a built-in module for regular expressions: The re module. This module contains a number of different functions and classes for working with regular expressions. One of the most useful functions from the re module is the re.search() function, which takes two required arguments:
  - The regex pattern
  - The string we want to search that pattern for
- The re.search() function will return a Match object if the pattern is found anywhere within the string. If the pattern is not found, re.search() returns None
- The first of these we'll learn is called a set. A set allows us to specify two or more characters that can match in a single character's position.



- We've learned that we should avoid using loops in pandas, and that vectorized methods are often faster and require less code.we learned that the Series.str.contains() method can be used to test whether a Series of strings match a particular regex pattern.

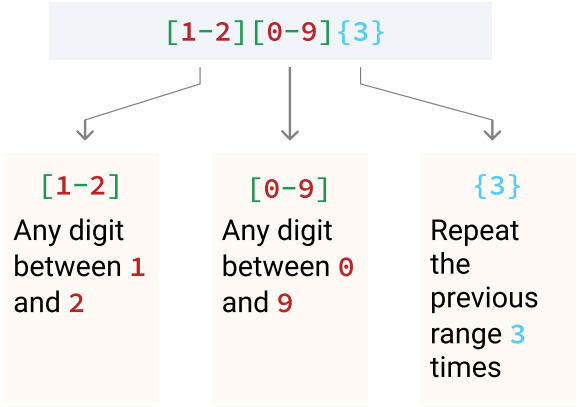
```
eg_list = ["Julie's favorite color is green.",
           "Keli's favorite color is Blue.",
           "Craig's favorite colors are blue and red."]

eg_series = pd.Series(eg_list)

pattern = "[Bb]lue"

pattern_contained = eg_series.str.contains(pattern)
print(pattern_contained)
0    False
1     True
2     True
dtype: bool
```

- One of the neat things about boolean masks is that you can use the Series.sum() method to sum all the values in the boolean mask, with each True value counting as 1, and each False as 0. This means that we can easily count the number of values in the original series that matched our pattern:

```
pattern_count = pattern_contained.sum()
```

Set [ ] and Quantifiers { }

- we learned that we could use braces ({}) to specify that a character repeats in our regular expression. For instance, if we wanted to write a pattern that matches the numbers in text from 1000 to 2999 we could write the regular expression below:



- The name for this type of regular expression syntax is called a quantifier. Quantifiers specify how many of the previous character our pattern requires, which can help us when we want to match substrings of specific lengths. As an example, we might want to match both e-mail and email. To do this, we would want to specify to match - either zero or one times.
- The specific type of quantifier we saw above is called a numeric quantifier. Here are the different types of numeric quantifiers we can use:

Quantifier	Pattern	Equivalent	Explanation
Zero or more	a*	a{0,}	The character a zero or more times
One or more	a+	a{1,}	The character a one or more times
Optional	a?	a{0,1}	The character a zero or one times

- quantifiers with other stuff:

Character Class	Pattern	Explanation
Set	[fud]	Either f , u, or d
Range	[a-e]	Any of the characters a, b, c, d, or e
	[0-3]	Any of the characters 0, 1, 2, or 3
	[A-Z]	Any uppercase letter
Set + Range	[A-Za-z]	Any uppercase or lowercase letter

- Just like with quantifiers, there are some other common character classes which we'll use a lot.

Character Class	Pattern	Explanation
Digit	`\d`	Any digit character (equivalent to `[0-9]`)
Word	`\w`	Any digit, uppercase, lowercase, or underscore character (equivalent to `[A-Za-z0-9_]`)
Whitespace	`\s`	Any space, tab or linebreak character
Dot	`.`	Any character except newline

-Let's quickly recap the concepts we learned in this screen:

```
-We can use a backslash to escape characters that have special meaning in regular express
ions (e.g. \ will match an open bracket character).
-Character classes let us match certain groups of characters (e.g. \w will match any word
character).
-Character classes can be combined with quantifiers when we want to match different numbe
rs of characters.
```

- since set [] has a special meaning but say you want to find a pattern where the titles had [] in them you can use escape characters to match those [].

```
print(r'hello\b')
hello
```

- \b has a special meaning(backsplash) But by using raw string 'r' infront these special character interpreatations will be turned off.It's strongly recommend using raw strings for every regex you write, rather than remember which sequences are escape sequences and using raw strings selectively. That way, you'll never encounter a situation where you forget or overlook something which causes your regex to break.

Capture Groups "()"

- What if we wanted to find out what the text of matched part of the string? In order to do this, we'll need to use capture groups. Capture groups allow us to specify one or more groups within our match that we can access separately. In this mission, we'll learn how to use one capture group per regular expression, but in the next mission we'll learn some more complex capture group patterns.

```
tag_5 = tag_titles.head()

67      Analysis of 114 propaganda sources from ISIS, Jabhat al-Nusra, al-Qaeda [pdf]
101                                Munich Gunman Got Weapon from the Darknet [German]
160                                      File indexing and searching for Plan 9 [pdf]
163    Attack on Kunduz Trauma Centre, Afghanistan  Initial MSF Internal Review [pdf]
196                                                [Beta] Speedtest.net  HTML5 Speed Test
Name: title, dtype: object

pattern = r"(\[\w+\])"
tag_5_matches = tag_5.str.extract(pattern)
print(tag_5_matches)
67        [pdf]
101    [German]
160       [pdf]
163       [pdf]
196      [Beta]
Name: title, dtype: object

pattern = r"\[(\w+)\]"
tag_5_matches = tag_5.str.extract(pattern)
print(tag_5_matches)

67        pdf
101    German
160       pdf
163       pdf
196      Beta
Name: title, dtype: object
```

- **Negative character classes**

Character Class	Pattern	Explanation
Negative Set	`[^fud]`	Any character except `f` , `u`, or `d`
	`[^1-3Z\s]`	Any character except `1` , `2`, `3`, `Z`, or whitespace characters
Negative Digit	`\D`	Any character except digit characters
Negative Word	`\W`	Any character except word characters
Negative Whitespace	`\S`	Any character except whitespace characters

```
PS : when using negative classes it fails for the cases where the pattern occurs at the e
nd of the string.
for example: pattern = r'([Jj]ava[^Ss])'it helps in avoiding to find strings that contain
s Javascript and helps in finding strings containing java but it fails if java occurs at
 the end of the string since there is no string left to do the matching it fails to matc
h.
```

- **word boundary anchor**: In the above case if we use r'\b[Jj]ava\b' we can find only strings containing word Java not java script and it will match even if the word java occurs at the end of the sentence.
- **Start and End anchor* *:

Anchor	Pattern	Explanation
Beginning	^abc	Matches abc only at the start of the string
End	abc$	Matches abc only at the end of the string

```
Note that the ^ character is used both as a beginning anchor and to indicate a negative s
et, depending on whether the character preceding it is a [ or not.
e.g [^sS] not containing S or s.
    ^s starts with s
```

Ignore Capitilization cases

- Up until now, we've been using sets like [Pp] to match different capitalizations in our regular expressions. This strategy works well when there is only one character that has capitalization, but becomes cumbersome when we need to cater for multiple instances.

  Within the titles, there are many different formatting styles used to represent the word "email." Here is a list of the variations:

  ```
  email
  Email
  e Mail
  e mail
  E-mail
  e-mail
  eMail
  E-Mail
  EMAIL
  ```

  ```
  To write a regular expression for this, we would need to use a set for all five letter
  s in email, which would make our regular expression very hard to read.
  ```

  Instead, we can use flags to specify that our regular expression should ignore case.

  Both re.search() and the pandas regular expression methods accept an optional flags argument. This argument accepts one or more flags, which are special variables in the re module that modify the behavior of the regex interpreter.

  Usage:
  ```
  import re
  email_tests.str.contains(r"email",flags=re.I)
  ```

Advanced Regular Expressions

- As we learned in the previous mission, to extract those mentions, we need to do two things:

  1. Use the Series.str.extract() method.
  2. Use a regex capture group.

Look arounds

- Lookarounds let us define a character or sequence of characters that either must or must not come before or after our regex match. There are four types of lookarounds:

Lookaround	Pattern	Explanation
Positive Lookahead	`zzz(?=abc)`	Matches `zzz` only when it is followed by `abc`
Negative Lookahead	`zzz(?!abc)`	Matches `zzz` only when it is **not** followed by `abc`
Positive Lookbehind	`(?<=abc)zzz`	Matches `zzz` only when it is preceded by `abc`
Negative Lookbehind	`(?<!abc)zzz`	Matches `zzz` only when it is **not** preceded by `abc`

- These tips can help you remember the syntax for lookarounds:

  1. Inside the parentheses, the first character of a lookaround is always ?.
  2. If the lookaround is a lookbehind, the next character will be <, which you can think of as an arrow head pointing behind the match.
  3. The next character indicates whether the is lookaround is positive (=) or negative (!).

Back references (\w)\1--oo-bb-aa

- Let's say we wanted to identify strings that had words with double letters, like the "ee" in "feed." Because we don't know ahead of time what letters might be repeated, we need a way to specify a capture group and then to repeat it. We can do this with backreferences.

  ``` test_cases = [

```
"I'm going to read a book.",
"Green is my favorite color.",
"My name is Aaron.",
"No doubles here.",
"I have a pet eel."
]
```

  for tc in test_cases: print(re.search(r"(\w)\1", tc))

```
_sre.SRE_Match object; span=(21, 23), match='oo'
_sre.SRE_Match object; span=(2, 4), match='ee'
None
None
_sre.SRE_Match object; span=(13, 15), match='ee'
```

```
### string replace or regex sub (substitute)


- When we learned to work with basic string methods, we used the str.replace() method to
 replace simple substrings. We can achieve the same with regular expressions using the r
e.sub() function. The basic syntax for re.sub() is:
```

string = "aBcDEfGHIj"

print(re.sub(r"[A-Z]", "-", string))

a-c--f---j
```

```
### Example of using capture groups to capture mutiple parts of an URL

 - we'll extract each of the three component parts of the URLs:

    1. Protocol
    2. Domain
    3. Page path

![url_examples_2.svg](attachment:url_examples_2.svg)

- In order to do this, we'll create a regular expression with multiple capture groups. Mu
ltiple capture groups in regular expressions are defined the same way as single capture g
roups — using pairs of parentheses.

- if you have some data that looks like this below:

0     8/4/2016 11:52
1    1/26/2016 19:30
2    6/23/2016 22:20
3     6/17/2016 0:01
4     9/30/2015 4:12
Name: created_at, dtype: object


- We'll use capture groups to extract these dates and times into two columns:

|Date|Time|
|---|---|
|8/4/2016|11:52|
|1/26/2016|19:30|
|6/23/2016|22:20|
|6/17/2016|0:01|
|9/30/2015|4:12|



- In order to do this we can write the following regular expression:

![multiple_capture_groups.svg](attachment:multiple_capture_groups.svg)

- Similarly I have these test URL's as shown below:
```

test_urls = pd.Series(['https://www.amazon.com/Technology-Ventures-Enterprise-Thomas-Byers/dp/0073523429' (https://www.amazon.com/Technology-Ventures-Enterprise-Thomas-Byers/dp/0073523429'), 'http://www.interactivedynamicvideo.com/' (http://www.interactivedynamicvideo.com/'), 'http://www.nytimes.com/2007/11/07/movies/07stein.html?_r=0' (http://www.nytimes.com/2007/11/07/movies/07stein.html?_r=0'), 'http://evonomics.com/advertising-cannot-maintain-internet-heres-solution/' (http://evonomics.com/advertising-cannot-maintain-internet-heres-solution/'), 'HTTPS://github.com/keppel/pinn', 'Http://phys.org/news/2015-09-scale-solar-youve.html', 'https://iot.seeed.cc' (https://iot.seeed.cc'), 'http://www.bfilipek.com/2016/04/custom-deleters-for-c-smart-pointers.html' (http://www.bfilipek.com/2016/04/custom-deleters-for-c-smart-pointers.html'), 'http://beta.crowdfireapp.com/?beta=agnipath' (http://beta.crowdfireapp.com/?beta=agnipath'), 'https://www.valid.ly?param' (https://www.valid.ly?param')])

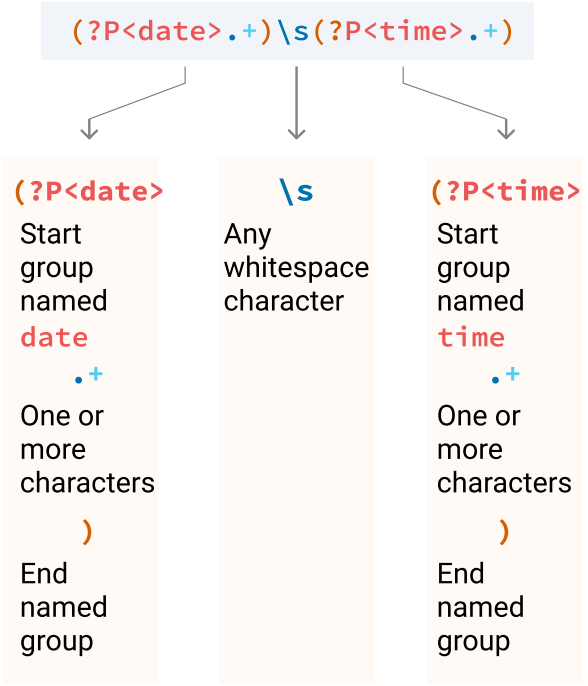
pattern = r '(.+)://([\w+.])*[/]?(.)'

url_parts = test_urls.str.extarct(pattern, flags = re.I)

```

- The resulting dataframe looks like this:

| 0 | 1 | 2 |
|---|---|---|
```

0	1	2
https	www.amazon.com (http://www.amazon.com)	Technology-Ventures-Enterprise-Thomas-Byers/dp...
http	www.interactivedynamicvideo.com (http://www.interactivedynamicvideo.com)	
http	www.nytimes.com (http://www.nytimes.com)	2007/11/07/movies/07stein.html?_r=0
http	evonomics.com	advertising-cannot-maintain-internet-heres-sol...
HTTPS	github.com	keppel/pinn

- if we want names for our capture groups we can use the following syntax



Thank

In [ ]: