L09: Regular Expressions

Eric Franzosa, PhD

franzosa@hsph.harvard.edu

Outline

- Regular expression (RE) syntax
- Python's `re` module
- Practice with RegexOne

Pattern matching

- Pattern matching is an important class of computing problem
  - Given a pattern P, does it occur in a dataset D, and if so, where?
- Often occurs in the context of text analysis
  - Finding a word on a website
  - Using find-and-replace in a document
  - "Googling" a query against the known Internet

- We've seen some limited uses of this in Python already (the `<pattern> in <text>` motif)

In [ ]:
```
"fun" in "fundamental"
```

In [ ]:
```
"i" in "team"
```

String matching

- But what about a more general question, e.g. *is my text an email address*?

In [ ]:
```
def check_if_email( text ):
    return "@" in text
```

In [ ]:
```
check_if_email( "franzosa@hsph.harvard.edu" )
```

In [ ]:
```
check_if_email( "follow me on Twitter @franzosa" )
```

In [ ]:
```
check_if_email( "let's meet @1pm today" )
```

Regular expressions

- A syntax for describing flexible patterns in strings
- Can be used to...
  - Test if a pattern occurs (like `<pattern> in <text>`)
  - Extract instances of patterns (when they occur)
  - Replace patterns with other text
- Available in many places where complex pattern matching is useful
- Also called "regexps", "regexes", or "REs" as shorthand

In [ ]:
```
# regular expression methods/data re bundled in Python's <re> module
import re
```

Support code

- We'll use this function to explore string pattern matching in Python
- It takes a pattern and text as arguments and uses `re.search( )` to compare them
- If the pattern is found, it is highlighted with ^ characters
- (We'll talk more about this code later)

In [ ]:
```
def refind( pattern, text ):
    print( text )
    match = re.search( pattern, text )
    if match is None:
        print( "<NO MATCH>" )
    else:
        start, end = match.start( ), match.end( )
        print( " " * start + "^" * (end - start) )
    return None
```

Raw strings

- Python interprets some character sequences non-literally
  - e.g. `\t` is seen as the tab character and `\n` as a newline
- We can change this behavior by adding `r` in front of the opening quote that defines a string
- This tells Python to treat the string as a "raw string"

In [ ]:
```
print( "Hello\tWorld!" )
```

In [ ]:
```
print( r"Hello\tWorld!" )
```

- I will use raw strings when defining patterns. Most of the time it doesn't matter (and I will probably forget at least once), but it's a good safety feature.

Exact matching

- A regular expression (RE) can be a generic string
- Here, matching a pattern to a text operates just like `<pattern> in <text>`

In [ ]:
```
refind( r"Hello", "Hello, World!" )
```

In [ ]:
```
# matching is case sensivity
refind( r"hello", "Hello, World!" )
```

The wildcard character, `.`

- In a RE, the `.` character will match any single character

In [ ]:
```
refind( r".at", "This is my pet cat, Spot" )
```

In [ ]:
```
# by default, we highlight the FIRST match to the pattern
refind( r".ello", "Hello mother, hello father" )
```

In [ ]:
```
refind( r"..i..", "Life, the Universe, and Everything" )
```

Escaping

- If you want to match a literal `.` (or another special character), write it as `\.`
- This is called "escaping the character"

In [ ]:
```
refind( r".", "I am a short sentence." )
```

In [ ]:
```
refind( r"\.", "I am a short sentence." )
```

Character classes

- `\d` matches any *digit* (`0`-`9`)
- `\D` matches any *non-digit*
- `\w` matches any "*word* character" (`A`-`Z`, `a`-`z`, `0`-`9`, `_`)
- `\W` matches any *non-word* character
- `\s` matches any *whitespace* character (e.g. space `` `, `tab`\t, `newline`\n`)
- `\S` matches any *non-whitespace* character

In [ ]:
```
refind( r"\d\d\d\d", "Yesterday, December 7, 1941--a date which will live in inf
amy" )
```

In [ ]:
```
refind( r"\d\D", "Do you want to see the movie in 2D or 3D?" )
```

In [ ]:
```
refind( r"\s\w\w\w\s", "It's my party and I'll cry if I want to." )
```

Custom character classes

- Sets or spans of characters inside of `[]`s define a custom character class
- `[AB]` matches A or B
- `[A-E]` matches any character A through E
- `[A-Za-z0-9_]` matches any word character (equivalent to `\w`)

In [ ]:
```
refind( r"[ACGT]", "DNA sequences are composed of A, C, G, and T nucleotides." )
```

In [ ]:
```
refind( r"[bcr]at", "This is my pet rat, Spot" )
```

Negation in custom character classes

- Negate a character class with an initial `^` (i.e. match the *opposite* characters)
- `[^AB]` matches any character *except* A and B
- `[^A-E]` matches any character *except* A through B
- `[^A-Za-z0-9_]` matches any *non*-word character (equivalent to `\W`)

In [ ]:
```
refind( r"[^c]at", "My pet cat ate your pet rat." )
```

Boundaries

- ^ matches the start of a string (*before* the first character)
- $ matches the end of a string
- \b matches a "word boundary" (beginning/end of a line, whitespace, or a non-word character)

In [ ]:
```
refind( r"^Jon", "Jane Doe please meet Jon Snow" )
```

In [ ]:
```
refind( r"deer$", "Doe, a deer, a female deer" )
```

In [ ]:
```
refind( r"deer\b", "Doe, a deer, a female deer" )
```

Repetition

- REs support syntax for specify a specific number (or range) of characters to match
- A+ matches 1 or more As
- A* matches 0 or more As
- A? matches 0 or 1 As (an "optional" A)

In [ ]:
```
refind( r"A+", "BBAAAAABBAAABBB" )
```

In [ ]:
```
refind( r"AB*A", "BBAABB" )
```

In [ ]:
```
refind( r"Erick?", "Who spells 'Eric' with a 'ck'?")
```

REs are greedy by default

- They return the match that starts earliest and ends latest
- This a very common source of surprises in RE-based matching

In [ ]:
```
# ".*" matches the longest possible string
refind( r"A.*A", "The band ABBA traveled to England, France, and Australia" )
```

In [ ]:
```
# ".*?" matches the shortest possible string
refind( r"A.*?A", "The band ABBA traveled to England, France, and Australia" )
```

Bounded repetition

- `A{N}` matches exactly *N* As in a row
- `A{N,M}` matches between *N* and *M* As in a row (inclusive)
- `A{N,}` matches *N* or more As in a row
- `A{,M}` matches up to *M* As in a row

In [ ]:
```
refind( r"\bA{3,5}\b", "A AAAAAA AAA AA AAAAAAAAAAAAA AAA A" )
```

- Helpful for defensive coding
- `A{10}` is a lot easier to get right than `AAAAAAAAAA`

Groups

- We can define larger pieces of a pattern within `( )`s
- These groups can be repeated (like a character or character class)
- We'll see later that we can extract these individually with Python code

In [ ]:
```
refind( r"(na)+", "Do you prefer banana or pineapple?" )
```

Groups with options

- Inside a group, `|` can be used to separate multiple options
- Behaves like a logical "or", e.g. `(AA|BB)` matches AA or BB

In [ ]:
```
refind( r"(birthday|wedding|promotion)", "Congratulations on your promotion!!" )
```

In [ ]:
```
refind( r"Eri(c|ck|ch|q)", "I think Eriq La Salle's spelling is the coolest")
```

Referencing groups

- The first defined group can be referenced later with `\1`
- (Likewise for the second `\2`, third `\3`, etc.)
- *This is where things start to get really funky*

In [ ]:
```
# find the first pair of letters repeated once
refind( r"(..)\1", "Do papas like bananas?" )
```

In [ ]:
```
# find any letter followed by <n>, then itself, then <n> again
refind( r"(.n)\1", "Do papas like bananas?" )
```

I think that's enough...

- While there's more RE syntax out there, you can do *a lot* with just the preceding syntax

Back to pattern matching e-mail addresses

- Using what we've seen, can we come up with a better e-mail pattern than just @?

In [ ]:
```
pattern = r"@"
refind( pattern, "franzosa@hsph.harvard.edu" )
refind( pattern, "follow me on Twitter @franzosa" )
refind( pattern, "let's meet @1pm today" )
```

Troubleshooting

- Did you use `\d` when you meant `\D`?
- Did you use a greedy match `.*` when you wanted a minimal match `.*?`?
- Did you forget to escape a symbolic character, e.g. `1 + 2` vs. `1 \+ 2`?
- *Did you make an assumption that wasn't true*?

Regular expressions in Python

- Imported as the `re` module (already did this above)
- Comes with a number of functions, of which we'll cover:
  - `re.search( )`
  - `re.finditer( )`
  - `re.split( )`
  - `re.sub( )`
- Comes with a new data type, the `Match` object

`re.search( PATTERN, TEXT )`

- Takes a pattern and text strings as arguments
- Returns `None` if the pattern isn't found in the text
- Otherwise it returns a `Match` object

In [ ]:
```
re.search( r"i", "team" )
```

In [ ]:
```
re.search( r"eric", "America" )
```

- The above output is Python's way of representing a generic piece of data
- The `Match` object doesn't have a simple representation like a `str`ing or `list`

The `Match` object

- Has methods for describing the result of a pattern match
- Capture the `Match` in a variable, then call methods on that variable

In [ ]:
```
match = re.search( r"(..)\1", "Do papas like bananas?" )
```

In [ ]:
```
# start position
match.start( )
```

In [ ]:
```
# end position
match.end( )
```

In [ ]:
```
# pull out the groups that were matched
match.group( 1 )
```

Aside: the `refind( )` function

- With this information you should be able to unpack what's happening in the `refind( )` utility function we defined and used earlier in the lecture
- Note the use of an `if/else` block to change the functions behavior depending on whether or not a match to the specified pattern was not (testing if the returned value was `None`).

`re.finditer( PATTERN, TEXT )`

- Takes a pattern and text strings as arguments
- Returns matches one-at-a-time inside a `for` loop
  - Similar to the behavior of the `range( )` function
- Finds largest match starting from beginning of text, then restarts after that match

In [ ]:
```
text = "AAAAA AAA A"
for match in re.finditer( r"A+", text ):
    print( text[match.start( ):match.end( )] )
```

`re.split( PATTERN, TEXT )`

- Takes a pattern and text strings as arguments
- Finds all instances of the pattern (similar to `re.finditer( )`)
- Returns a list of strings after splitting the text on those patterns
- A much more powerful version of `str.split( PATTERN )`

In [ ]:
```
# str.split( PATTERN ) can only split on a well-defined pattern, e.g. <tab>
"1\t2\t3".split( "\t" )
```

In [ ]:
```
# re.split( PATTERN, TEXT ) can split on any RE-definable pattern
re.split( r"[aeiou]{2}", "Congratulations on your promotion" )
```

re.sub(PATTERN, SUBSTITUTION, TEXT)

- Takes pattern, substitution, and text strings as arguments
- Finds all instances of the pattern (similar to `re.finditer( )`)
- Replaces all instances of the pattern with the SUBSTITUTION string
- A much more powerful version of `str.replace( PATTERN, SUBSTITUTION )`

In [ ]:
```
# str.replace( PATTERN, SUBSTITUTION ) can only work with well-defined patterns
"banana".replace( "a", "o" )
```

In [ ]:
```
# re.sub( PATTERN, SUBSTITUTION, TEXT ) can replace any RE-definable pattern
re.sub( r"[aeiou]{2}", r"OO", "Congratulations on your promotion" )
```

In [ ]:
```
# SUBSTITUTION can refer to pattern groups
re.sub( r"(\w+), (\w+)", r"\2 \1", "Franzosa, Eric" )
```

Resources

- There's more info in the Python docs, if interested:
  - https://docs.python.org/3/library/re.html#regular-expression-syntax (https://docs.python.org/3/library/re.html#regular-expression-syntax)
- RegExr is an online tool for testing out REs and learning how the work:
  - https://regexr.com/ (https://regexr.com/)
- RegexOne is an online "quiz" for learning RE concepts (today's activity)
  - https://regexone.com/ (https://regexone.com/)