CS 61A
Summer 2023

Regular Expressions
Discussion 10: August 1, 2023

Regular Expressions

Regular expressions are a way to describe sets of strings that meet certain criteria, and are incredibly useful for pattern matching.

The simplest regular expression is one that matches a sequence of characters, like `aardvark` to match any “aardvark” substrings in a string.

However, you typically want to look for more interesting patterns. We recommend using an online tool like regexr.com or regex101.com for trying out patterns, since you’ll get instant feedback on the match results.

Character Classes

A character class makes it possible to search for any one of a set of characters. You can specify the set or use pre-defined sets.

Class	Description
`[abc]`	Matches a, b, or c
`[a-z]`	Matches any character between a and z
`[^A-Z]`	Matches any character that is not between A and Z.
`\w`	Matches any “word” character. Equivalent to `[A-Za-z0-9_]`.
`\d`	Matches any digit. Equivalent to `[0-9]`.
`[0-9]`	Matches a single digit in the range 0 - 9. Equivalent to `\d`.
`\s`	Matches any whitespace character (spaces, tabs, line breaks).
`.`	Matches any character besides new line.

Character classes can be combined, like in `[a-zA-Z0-9]`.

Combining Patterns

There are multiple ways to combine patterns together in regular expressions.

Combo	Description
`AB`	A match for A followed immediately by one for B. Example: `x[.,]y` matches “x.y” or “x,y”.
A|B	Matches either A or B. Example: \d+|Inf matches either a sequence containing 1 or more digits **or** “Inf”.

A pattern can be followed by one of these quantifiers to specify how many instances of the pattern can occur.

Symbol	Description
`*`	0 or more occurrences of the preceding pattern. Example: `[a-z]*` matches any sequence of lower-case letters or the empty string.
`+`	1 or more occurrences of the preceding pattern. Example: `\d+` matches any non-empty sequence of digits.
`?`	0 or 1 occurrences of the preceding pattern. Example: `[-+]?` matches an optional sign.
`{1,3}`	Matches the specified quantity of the preceding pattern. `{1,3}` will match from 1 to 3 instances. `{3}` will match exactly 3 instances. `{3,}` will match 3 or more instances. Example: `\d{5,6}` matches either 5 or 6 digit numbers.

Groups

Parentheses are used similarly as in arithmetic expressions, to create groups. For example, `(Mahna)+` matches strings with 1 or more “Mahna”, like “MahnaMahna”. Without the parentheses, `Mahna+` would match strings with “Mahn” followed by 1 or more “a” characters, like “Mahnaaaa”.

Anchors

- `^`: Matches the beginning of a string. Example: `^(I|You)` matches I or You at the start of a string.
- `$`: Normally matches the empty string at the end of a string or just before a newline at the end of a string. Example: `(\.edu|\.org|\.com)$` matches .edu, .org, or .com at the end of a string.
- `\b`: Matches a “word boundary”, the beginning or end of a word. Example: `s\b` matches s characters at the end of words.

Special Characters

The following special characters are used above to denote types of patterns:

```
\ / ( ) [ ] { } + * ? | $ ^ .
```

That means if you actually want to match one of those characters, you have to *escape* it using a backslash. For example, `\(1\+3\)` matches “(1 + 3)”.

Using Regular Expressions in Python

Many programming languages have built-in functions for matching strings to regular expressions. We'll use the Python re module in 61A, but you can also use similar functionality in SQL, JavaScript, Excel, shell scripting, etc.

The search method searches for a pattern anywhere in a string:

```
re.search(r"(Mahna)+", "Mahna Mahna Ba Dee Bedebe")
```

That method returns back a match object, which is considered truth-y in Python and can be inspected to find the matching strings. If no match is found, returns `None`.

For more details, please consult the re module documentation or the re tutorial.

Q1: CS Classes

On reddit.com, there is an r/berkeley subreddit for discussions about everything UC Berkeley. However, there is such a large amount of EE and CS-related posts that those posts are auto-tagged so that readers can choose to ignore them or read only them.

Write a regular expression that finds strings that resemble a CS or EE class- starting with “CS” or “EE”, followed by a number, and then optionally followed by “A”, “B”, or “C”. Your search should be case insensitive, so both “CS61A” and “cs61a” would match.

```
import re

def cs_classes(post):
    """
    Returns strings that look like a Berkeley CS or EE class,
    starting with "CS" or "EE", followed by a number, optionally ending with A, B, or C
    and potentially with a space between "CS" or "EE" and the number.
    Case insensitive.

    >>> cs_classes("Is it unreasonable to take CS61A, CS61B, CS70, and EE16A in the
    summer?")
    True
    >>> cs_classes("how do I become a TA for cs61a? that job sounds so fun")
    True
    >>> cs_classes("Can I take ECON101 as a CS major?")
    False
    >>> cs_classes("Should I do the lab lites or regular labs in EE16A?")
    True
    >>> cs_classes("thoughts on ee127?")
    True
    >>> cs_classes("Is 70 considered an EECS class?")
    False
    >>> cs_classes("What are some good CS upper division courses? I was thinking about CS
     161 or CS 169a")
    True
    """
    return bool(re.search(__________, post))
```

[Ee][Ee]

2 copies

r'([Cc][Ss]|[Ee]{2}) \s? \d+ [A-Ca-c]?'

CS / EE

optional space

course number

optional A/B/C

case insensitive

Q2: Greetings

Let's say hello to our fellow bears! We've received messages from our new friends at Berkeley, and we want to determine whether or not these messages are *greetings*. In this problem, there are two types of greetings - salutations and valedictions. The first are messages that start with "hi", "hello", or "hey", where the first letter of these words can be either capitalized or lowercase. The second are messages that end with the word "bye" (capitalized or lowercase), followed by either an exclamation point, a period, or no punctuation. Write a regular expression that determines whether a given message is a greeting.

[ ] \ ^ -

r' ^[Hh](i|ello|ey)\b | \b[Bb]ye[!.]? $ '

salutations valedictions

[!\.]

```
import re

def greetings(message):
    """
    Returns whether a string is a greeting. Greetings begin with either Hi, Hello, or
    Hey (first letter either capitalized or lowercase), and/or end with Bye (first letter
    either capitalized or lowercase) optionally followed by an exclamation point or
    period.

    >>> greetings("Hi! Let's talk about our favorite submissions to the Scheme Art
    Contest")
    True
    >>> greetings("Hey I love Taco Bell")
    True
    >>> greetings("I'm going to watch the sun set from the top of the Campanile! Bye!")
    True
    >>> greetings("Bye Bye Birdie is one of my favorite musicals.")
    False
    >>> greetings("High in the hills of Berkeley lived a legendary creature. His name was
     Oski")
    False
    >>> greetings('Hi!')
    True
    >>> greetings("bye")
    True
    """
    return bool(re.search(__________, message))
```

Q3: Phone Number Validator

Create a regular expression that matches phone numbers that are 11, 10, or 7 numbers long.

Phone numbers 7 numbers long have a group of 3 numbers followed by a group of 4 numbers, either separated by a space, a dash, or nothing.

Examples: `123-4567`, `1234567`, `123 4567`

Phone numbers 10 numbers long have a group of 3 numbers followed by a group of 3 numbers followed by a group of 4 numbers, either separated by a space, a dash, or nothing.

Examples: `123-456-7890`, `1234567890`, `123 456 7890`

Phone numbers 11 numbers long have a group of 1 number followed by a group 3 numbers followed by a group of 3 numbers followed by a group of 4 numbers, either separated by a space, a dash, or nothing.

Examples: `1-123-456-7890`, `11234567890`, `1 123 456 7890`

It is fine if spacing/dashes/no space mix! So `123 456-7890` is fine. \\ → \

Note: The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

[\s\-]

```
import re        r'((\d[\-\s]?)? \d{3}[\-\s]?  )?  \d{3}[\-\s]?\d{4}          '
def phone_number(string):   11-digit      10-digit              7-digit
    """
    >>> phone_number("Song by Logic: 1-800-273-8255")
    True
    >>> phone_number("123 456 7890")
    True
    >>> phone_number("1" * 11) and phone_number("1" * 10) and phone_number("1" * 7)
    True
    >>> phone_number("The secret numbers are 4, 8, 15, 16, 23 and 42 (from the TV show
    Lost)")
    False
    >>> phone_number("Belphegor's Prime is 1000000000000066600000000000001")
    False
    >>> phone_number(" 1122334455 ")
    True
    >>> phone_number(" 11 22 33 44 55 ")
    False
    >>> phone_number("Tommy Tutone's '80s hit 867-5309 /Jenny")
    True
    >>> phone_number("11111111") # 8 digits isn't valid, has to be 11, 10, or 7
    False
    """
    return bool(re.search(______________________________________________________))


# You can use more space on the back if you want
```

Q4: Address First Line

Write a regular expression that parses strings and returns whether it contains the first line of a US mailing address.

US mailing addresses typically contain a block number, which is a sequence of 3-5 digits, following by a street name. The street name can consist of multiple words but will always end with a street type abbreviation, which itself is a sequence of 2-5 English letters. The street name can also optionally start with a cardinal direction (“N”, “E”, “W”, “S”). Everything should be properly capitalized.

Proper capitalization means that the first letter of each name is capitalized. It is fine to have things like “WeirdCApitalization” match.

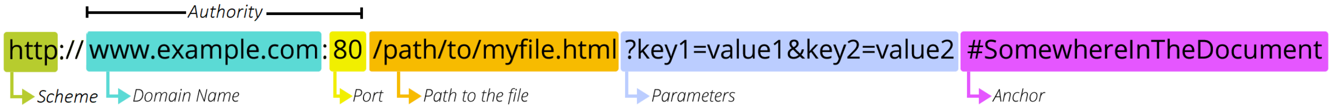
See the doctests for some examples.

```python
import re

def address_oneline(text):
    """
    Finds and returns if there are expressions in text that represent the first line
    of a US mailing address.

    >>> address_oneline("110 Sproul Hall, Berkeley, CA 94720")
    True
    >>> address_oneline("What's at 39177 Farwell Dr? Is there a 39177 Nearwell Dr?")
    True
    >>> address_oneline("I just landed at 780 N McDonnell Rd, and I need to get to 1880-
    ish University Avenue. Help!")
    True
    >>> address_oneline("123 Le Roy Ave")
    True
    >>> address_oneline("110 Unabbreviated Boulevard")
    False
    >>> address_oneline("790 lowercase St")
    False
    """
    block_number = r'___'
    cardinal_dir = r'___' # whitespace is important!
    street = r'___'
    type_abbr = r'___'
    street_name = f"{cardinal_dir}{street}{type_abbr}"
    return bool(re.search(f"{block_number} {street_name}", text))
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Q5: Basic URL Validation

In this problem, we will write a regular expression which matches a URL. URLs look like the following:



URL

For example, in the link `https://cs61a.org/resources/#regular-expressions`, we would have:

- Scheme: `https://`
- Domain Name: `cs61a.org`
- Path to the file: `/resources`
- Anchor: `/#regular-expressions`

The port and parameters are not present in this example and you will not be required to match them for this problem.

You can reference this documentation from MDN if you're curious about the various parts of a URL.

For this problem, a valid **domain name** consists of two "words" separated by a single period. Recall that a "word" can consist of letters, numbers, and underscores. The second "word" should be exactly 3 characters long and represents the domain's extension. In the case of the above example, "cs61a" and "org" are the two "words" that are joined by a period.

For a URL to be "valid," it must contain a valid domain name and will optionally have a scheme, path, and anchor. (Note: In this problem, "scheme" does not refer to the programming language.)

A valid **scheme** will either be `http://` or `https://`.

A valid **path** starts with a slash and then must be a valid path to a file or directory. A path to a directory should look something like `path/to/directory`, while a path to a file might look something like `/composingprograms.html` (note the period followed by the extension). Paths should not end with a slash or have more than one period – `/composing.programs.html/` is not a valid path. Any non-slash and non-period character in a path should be a letter, number, or underscore.

A valid **anchor** starts with `/#`. While they are more complicated, for this problem assume that valid anchors will then be followed by letters, numbers, hyphens, or underscores.

> **Hint**: You can use \ to escape special characters in regex.

```
import re
def match_url(text):
    """
    >>> match_url("https://cs61a.org/resources/#regular-expressions")
    True
    >>> match_url("https://pythontutor.com/composingprograms.html")
    True
    >>> match_url("https://pythontutor.com/should/not.match.this")
    False
    >>> match_url("https://link.com/nor.this/")
    False
    >>> match_url("http://insecure.net")
    True
    >>> match_url("htp://domain.org")
    False
    """
    beginning = r'___'
    scheme = r'___'
    domain = r'___'
    path = r'___'
    anchor = r'___'
    end = r'___'
    full_string = beginning + scheme + domain + path + anchor + end
    return bool(re.match(full_string, text))
```

Q6: Email Domain Validator

Create a regular expression that makes sure a given string `email` is a valid email address and that its domain name is in the provided list of `domains`.

An email address is valid if it contains letters, \w number, or underscores, followed by an @ symbol, then a domain.

All domains will have a 3 letter extension following the period.

> **Hint**: For this problem, you will have to make a regex pattern based on the elements in the `domains` parameter. A for loop can help with that.
>
> ~~**Extra**:~~ There is a particularly elegant solution that utilizes join and replace instead of a for loop.
>
> **Note:** The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

```
import re
def email_validator(email, domains):
    """
    >>> email_validator("oski@berkeley.edu", ["berkeley.edu", "gmail.com"])
    True
    >>> email_validator("oski@gmail.com", ["berkeley.edu", "gmail.com"])
    True
    >>> email_validator("oski@berkeley.com", ["berkeley.edu", "gmail.com"])
    False
    >>> email_validator("oski@berkeley.edu", ["yahoo.com"])
    False
    >>> email_validator("xX123_iii_OSKI_iii_123Xx@berkeley.edu", ["berkeley.edu", "gmail.
    com"])
    True
                                                          + '\.' + edu
    >>> email_validator("oski@oski@berkeley.edu", ["berkeley.edu", "gmail.com"])
    False
    >>> email_validator("oski@berkeleysedu", ["berkeley.edu", "gmail.com"])
    False
                d = domains [0]
    """
    pattern = r' \w+ @ (' + d[:-4] + r'\.' + d[-3: ]
    for ______________________:
        'Use as many lines as necessary'
    return bool(re.search(pattern, email))

                                   \w+
                              ___________  @ ( ____ | ____ | ____ )
                                address

                                for d in domains[1:] :
                                  pattern += d[:-4] + r'\.' + d[-3: ] + '|'

# You can use more space on the back if you want

                              pattern += ')'
```

(1 2 3 ~~4~~)

(filter (lambda (x)) ')