Regex and Extra Scheme Practice Solutions

CSM 61A

April 18 - April 22, 2022

1 Regular Expressions

Regular expressions help you match "patterns" to strings. We use the `re` module. For example, the `re.match()` function returns a Match object, which can tell you if a certain "pattern" matches a certain string.

```
>>> import re
>>> bool(re.match(r"Hello", "Hello World"))
True
>>> bool(re.match(r"Bye", "Hello World"))
False
```

We use the raw string syntax, r"regex",to specify a pattern because it stops any backslash characters from being interpreted as special characters such as newlines.

```
>>> print("my\nstring") # \n is interpreted as newline
my
string
>>> print(r"my\nstring")
my\nstring
```

There are many components to a pattern.

Character Classes: Character classes are one of the ways we can specify what characters your pattern matches. For each example, we match characters from the string, "Hello World! Today is 10/15/2021". Boxed characters are matched.

Character class	Description	Example
`[oa]`	A singular o or singular a	"Hell[o] W[o]rld! T[o]d[a]y is 10/15/2021"
`[0-9]`, `\d`	Any digit	"Hello World! Today is [1][0]/[1][5]/[2][0][2][1]"
`[^a-z]`	Anything except a lowercase letter	"[H]ello[ ][W]orld[!][ ][T]oday[ ]is[ ][1][0][/][1][5][/][2][0][2][1]"
`\s`	Any whitespace	"Hello[ ]World![ ]Today[ ]is[ ]10/15/2021"
`[a-zA-z0-9_]`, `\w`	Any letter, digit, or underscore	[Hello] [World]! [Today] [is] [10]/[15]/[2021]
`.`	Anything except newline	[Hello World! Today is 10/15/2021]
`\b`	Word boundary (this is an anchor class!)	[ ]Hello[ ] [ ]World![ ] [ ]Today[ ] [ ]is[ ] [ ]10[ ]/[ ]15[ ]/[ ]2021[ ]

We can combine character classes in the following two ways.

- `[a-z]|[A-Z]` matches a lowercase letter OR an uppercase letter
- `[a-z][A-Z]` mathces a lowercase letter AND then an uppercase letter

There are also two special characters used outside of a character class: ^and $. These are called anchors.

Character	Description	Example
^	Matches the beginning of a string	`^abc` matches [abc]defabcdef
$	Matches the end of a string	`abc$` matches abcdef[abc]

Quantifiers: Quantifiers tell you how many of something you will match.

Quantifier	Description
`[a-z]+`	Matches one or more lowercase letters
`[0-9]*`	Matches zero or more digits
`[A-Z]?`	Matches zero or one uppercase letters
`a{1,3}`	Matches 1, 2, or 3 a's
`a{2,}`	Matches 2 or more a's
`a{,2}`	Matches 0, 1, or 2 a's
`a{2}`	Matches exactly 2 a's

Python Functions: The Python `re` module has a lot of functions for matching patterns.

```
import re
ptn, stn = r"...", "..."    # ptn = pattern, stn = string
# note: regex patterns use raw strings!
re.search(ptn, stn)         # does `string` contain `pattern`?
  .group(idx)               # `idx`th match of `pattern` in
                            # `string`
re.fullmatch(ptn, stn)      # does all of `string` follow
                            # `pattern`?
re.match(ptn, stn)          # does `string` start with `pattern`?
re.finditer(ptn, stn)       # iter over `pattern` matches in
                            # `string`
re.sub(ptn, "<nope>", stn)  # replace `pattern` with `"<nope>"`
re.findall(ptn, stn)        # list of all matches of `pattern
```

Groups: Groups allow you to group quantifiers.

```
>>> re.findall(r"ab{1,3}", "abab abb abbb aaab aabb")
['ab', 'ab', 'abb', 'abbb', 'ab', 'abb']
>>> re.findall(r"(ab){1,3}", "abab abb abbb aaab aabb")
['ab', 'ab', 'ab', 'ab', 'ab']
```

They also allow you to ”capture” part of a string and return it instead of the entire match.

```
>>> re.findall(r"CSM(\w*)", "CSM61A, CSM61B, CSM70")
['61A', '61B', '70']
```

1. We are given a linear equation of the form $mx + b$, and we want to extract the m and b values. Remember that '.' and '+' are special meta-characters in Regex. The variable name can be either lower case or upper case.

```
import re
def linear_functions(eq_str):
    """
    Given the equation in the form of 'mx+b', returns a list
       containing a tuple of m and b values.
    >>> linear_functions("1x+0")
    [('1', '0')]
    >>> linear_functions("100y+44")
    [('100', '44')]
    >>> linear_functions("99.9z+.23")
    [('99.9', '.23')]
    >>> linear_functions("55t+0.4")
    [('55', '0.4')]
    >>> linear_functions("123T+456")
    [('123', '456')]
    """
    return re.findall(r"__________", eq_str)
```

```
r"(\d*\.?\d+)[a-zA-Z]\+(\d*\.?\d+)"
```

2. We are given a sentence, and we want to find all the words that end with ing! The word "ing" does not count a word ending with "ing".

```
import re
def extract_ing(sentence):
    """
    Given a string sentence, finds all words that end with
        "ing". For the purpose of this function, words can
        only have word characters.
    >>> extract_ing("Extracting single word") # single
       does not end with "ing"
    ['Extracting']
    >>> extract_ing("cool wording!")
    ['wording']
    >>> extract_ing("thising, ising,exciting...")
    ['thising', 'ising', 'exciting']
    >>> extract_ing("sad-dening")
    ['dening']
    """
    return re.findall(r"_________", sentence)
```

```
    r'\b(\w+ing)\b'
```

3. We are given a space-separated string of filenames with their extensions. Return all the file extensions. Assume that filenames and extensions can only have word characters, including the . character, and no spaces in them; if a filename has no extension or an extension no filename, ignore it. In addition, filenames are required to start with a word character, not a period.

```
import re
def extensions(s):
    """
    Given a space-separated string of filenames with their
       extensions, finds all extensions.
    Returns a list of the filenames as tuples of the name and
       file extension type.
    Assume that filenames can only have word characters and no
        spaces in them.
    >>> extensions("vol1.txt vol2.txt vol3.jpg") # Last . not
       kept in output
    [('vol1', 'txt'), ('vol2', 'txt'), ('vol3', 'jpg')]
    >>> extensions("my_diary.abc.txt")
    [('my_diary.abc', 'txt')]
    >>> extensions("my_diary my_diary2.def .file") # If a
       filename has no extension or vice-versa, ignore it.
    [('my_diary2', 'def')]
    """
    return re.findall(r"____", s)
```

```
r"\b(\w[\w\.]*)\.(\w+)\b"
```

2 Extra Scheme Practice

1. Fill in `skip-list`, which takes in a potentially nested list `lst` and a single-argument filter function `filter-fn` that returns a boolean when called, and goes through each element in order. It returns a new list that contains all elements that return true when passed into `filter-fn`. The returned list is *not nested*.

```
;Doctests
scm> (skip-list '(1 (3)) even?)
()
scm> (skip-list '(1 (2 (3 4) 5) 6 (7) 8 9) odd?)
(1 3 5 7 9)

(define (skip-list lst filter-fn)
    (define (helper lst lst-so-far next)
        (cond
            ((null? lst)
                (if (null? next)
                    lst-so-far
                    (helper car next lst-so-far cdr)))
            ((list? (car lst))
                (helper (car lst)))
            ((filter-fn (car lst))
                (_____))
            (else
                (_____))
        )
    )
    (helper _____)
)
```

```
(define (skip-list lst filter-fn)
    (define (helper lst lst-so-far next)
```

```
            (cond
                ((null? lst) (if (null? next)
                    lst-so-far
                    (helper (car next) lst-so-far (cdr next)))
                )
                ((list? (car lst))
                    (helper (car lst)
                        lst-so-far
                        (cons (cdr lst) next)))
                ((filter-fn (car lst))
                    (helper (cdr lst) (append lst-so-far (list (
                      car lst))) next))
                (else (helper (cdr lst) lst-so-far next))
            )
        )
        (helper lst nil nil)
    )
```

2. Implement `slice`, which takes in a a list `lst`, a starting index `i`, and an ending index `j`, and returns a new list containing the elements of `lst` from index `i` to `j - 1`.

```
;Doctests
scm> (slice '(0 1 2 3 4) 1 3)
(1 2)
scm> (slice '(0 1 2 3 4) 3 5)
(3 4)
scm> (slice '(0 1 2 3 4) 3 1)
()

(define (slice lst i j)









)
```

```
(define (slice lst i j)
      (cond ((or (null? lst) (>= i j)) nil)
                ((= i 0) (cons (car lst) (slice (cdr lst) i (-
                   j 1))))
                (else (slice (cdr lst) (- i 1) (- j 1)))))
```

3. Now implement slice with the same specifications, but make you implementation tail recurisve.
   You may wish to use the built-in append function, which takes in two lists and returns a new list containing the elements of the two lists concatenated together.

```
(define (slice lst i j)









)
```

```
(define (slice lst i j)
  (define (slice-tail lst i j lst-so-far)
      (cond ((or (null? lst) (>= i j)) lst-so-far)
                ((= i 0) (slice-tail (cdr lst) i (- j 1) (
                  append lst-so-far (list (car lst)))))
                (else (slice-tail (cdr lst) (- i 1) (- j 1) lst
                  -so-far))))
  (slice-tail lst i j nil))
```

Alternate Solution:

```
(define (slice lst i j)
  (define (slice-tail lst index lst-so-far)
      (cond ((or (null? lst) (= index j)) lst-so-far)
                ((<= i index) (slice-tail (cdr lst) (+ index 1)
                    (append lst-so-far (list (car lst)))))
                (else (slice-tail (cdr lst) (+ index 1) lst-so-
                  far))))
  (if (< i j) (slice-tail lst 0 nil) nil))
```