Regular Expressions in programming

CSE 307 – Principles of Programming Languages

Stony Brook University

http://www.cs.stonybrook.edu/~cse307

1

What are *Regular Expressions*?

- Formal language representing a text pattern interpreted by a regular expression processor
  - Used for **matching**, **searching** and **replacing** text
  - There are no variables and you cannot do mathematical operations (such as: you cannot add **1+1**) – it is not a programming language
  - Frequently you will hear them called *regex* or *RE* for short (or pluralized "*regexes*")

What are Regular Expressions?

- **<u>Usage</u>** examples:
  - Test if a phone number has the correct number of digits
  - Test if an email address has the correct format
  - Test if a Social Security Number is in the correct format
  - Search a text for words that contain digits
  - Find duplicate words in a text
  - Replace all occurrences of "*Bob*" and "*Bobby*" with "*Robert*"
  - Count the number of times "*science*" is preceded by "*computer*" or "*information*"
  - Convert a tab indentations file with spaces indentations

(c) Paul Fodor (CS Stony Brook)

What are Regular Expressions?

- But what is "*Matches*"?

  - a text *matches* a regular expression if it is correctly described by the regex

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m
<re.Match object; span=(0, 12), match='Isaac Newton'>

>>> m.group(0) # The entire match
'Isaac Newton'

>>> m.group(1) # The first parenthesized subgroup.
'Isaac'

>>> m.group(2) # The second parenthesized subgroup.
'Newton'
```

(c) Paul Fodor (CS Stony Brook)

History of Regular Expressions

- 1943: Warren McCulloch and Walter Pitts developed models of how the nervous system works
- 1956: Steven Kleene described these models with an algebra called "*regular sets*" and created a notation to express them called "*regular expressions*"
- 1968: Ken Thompson implements regular expressions in **ed**, a Unix text editor
  - Example: **g/Regular Expression/p**
    - meaning Global Regular Expression Print (**grep**)
      - g = global / whole file; p= print

History of Regular Expressions

- grep evolved into egrep
  - but broke backward compatibility
- Therefore, in 1986, everyone came together and defined POSIX (Portable Operating Systems Interface)
  - Basic Regular Expressions (BREs)
  - Extended Regular Expressions (EREs)
- 1986: Henry Spencer releases the **regex** library in C
  - Many incorporated it in other languages and tools
- 1987: Larry Wall released Perl
  - Used Spencer's regex library
  - Added powerful features
    - Everybody wanted to have it in their languages: ***Perl Compatible RE (PCRE)*** library, Java, Javascript, C#/VB/.NET, MySQL, PHP, Python, Ruby

(c) Paul Fodor (CS Stony Brook)

Regular Expressions Engines

- Main versions / standards:
  - PCRE
  - POSIX BRE
  - POSIX ERE
- Very subtle differences
  - Mainly older UNIX tools that use POSIX BRE for compatibility reasons
- In use:
  - Unix (POSIX BRE, POSIX ERE)
  - PHP (PCRE)
  - Apache (v1: POSIX ERE, v2: PCRE)
  - MySQL (POSIX ERE)
- Each of these languages is improving, so check their manuals

Python Regular Expressions

-
  - It is more powerful than String splits:

```
>>> "ab bc cd".split()
['ab', 'bc', 'cd']
```

- Import the re module:

```
import re
>>> re.split(" ", "ab bc cd")
['ab', 'bc', 'cd']
```

```
>>> re.split("\d", "ab1bc4cd")
['ab', 'bc', 'cd']
```

```
>>> re.split("\d*", "ab13bc44cd443gg")
['', 'a', 'b', '', 'b', 'c', '', 'c', 'd',
'', 'g', 'g', '']
```

(c) Paul Fodor (CS Stony Brook)

Python Regular Expressions

```
>>> re.split("\d+", "ab13bc44cd443gg")
['ab', 'bc', 'cd', 'gg']

>>> m = re.search('(?<=abc)def', 'abcdef')

>>> m
<re.Match object; span=(3, 6), match='def'>
```

Online Regular Expressions

- https://regexpal.com

(c) Paul Fodor (CS Stony Brook)
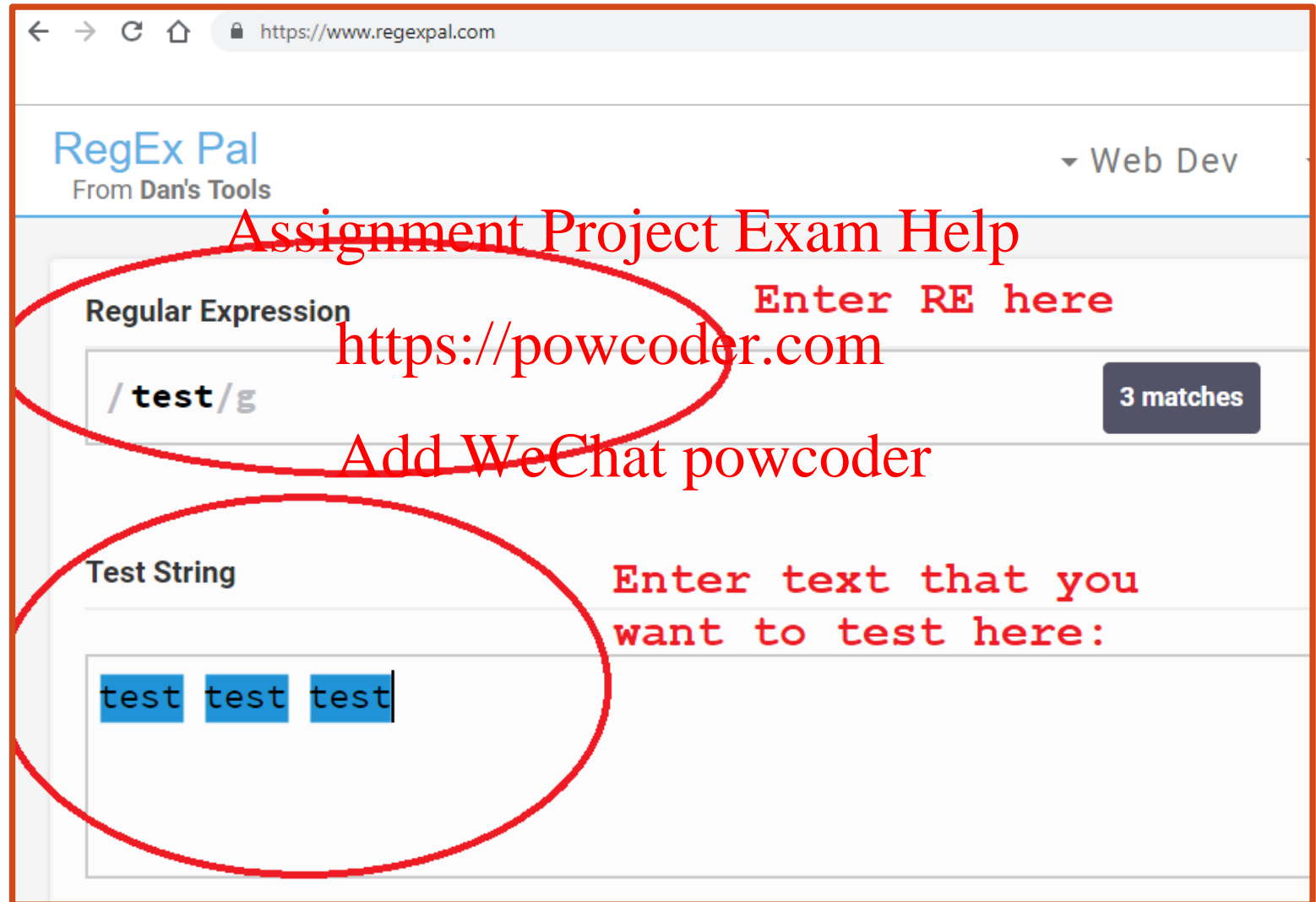
Regular Expressions

- **_Strings_**:
  - "car" matches "car"
  - "car" also matches the first three letters in "cartoon"
  - "car" does not match "c_a_r"
- Similar to search in a word processor
- Case-sensitive (by default): "car" does not match "Car"
- **_Metacharacters_**:
  - Have a special meaning
  - Like mathematical operators
    - Transform char sequences into powerful patterns
  - Only a few characters to learn: \ . * + - { } [ ] ( ) ^ $ | ? : ! =
  - May have multiple meanings
    - Depend on the context in which they are used
  - Variation between regex engines

11

The *wildcard character*

- Like in card games: one card can replace any other card on the pattern

Metacharacter	Meaning
.	Any character except newline

- Examples:
  - "h.t" matches "hat", "hot", "heat"
  - ".a.a.a" matches "banana", "papaya"
  - "h.t" does not match "heat" or "Hot"
- Common mistake:
  - "9.00" matches "9.00", but it also match "9500", "9-00"
- **We should write regular expressions to match what we want and ONLY what we want (We don't want to be overly permissive, we don't want false positives, we want the regular expression to match what we are not looking for)**

(c) Paul Fodor (CS Stony Brook)

Escaping Metacharacter

- Allow the use of metacharacters **as characters**:
  - "\." matches "."

Metacharacter	Meaning
\	Escape the next character

  - "9\.00" matches only "9.00", but not "9500" or "9-00"
  - Match a backslash by escaping it with a backslash:
    - "\\" matches only "\"
    - "C:\\Users\\Paul" matches "C:\Users\Paul"
  - Only for metacharacters
    - literal characters should never be escaped because it gives them meaning, e.g., r"\n"
    - Sometimes we want both meanings
      - Example: we want to match files of the name: "1_report.txt", "2_report.txt",…
      - "._report\.txt" uses the first . as a wildcard and the second \. as the period itself

(c) Paul Fodor (CS Stony Brook)

Other special characters

- Tabs: \t

- Line returns: \r *(line return),* \n *(newline),* \r\n

- Unicode codes: \u00A9

- ASCII codes: \x00A9

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Character sets

Metacharacter	Meaning
[	Begin character set
]	End character set

- Matches any of the characters inside the set
  - But only one character
  - Order of characters does not matter
  - Examples:
    - "[aeiou]" matches a single vowel, such as: "a" or "e"
    - "gr[ae]y" matches "gray" or "grey"
    - "gr[ae]t" does not match "great"

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Character ranges

- [a-z] = [abcdefghijklmnoprqstuxyz]
  - Range metacharacter - is only a character range when it is inside a character set, a dash line otherwise
    - represent all characters between two characters
  - Examples:
    - [0-9]
    - [A-Za-z]
    - [0-9A-Za-z]
    - [0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9] matches phone "631-632-9820"
    - [0-9][0-9][0-9][0-9][0-9] matches zip code "90210"
    - [A-Z0-9][A-Z0-9][A-Z0-9] [A-Z0-9][A-Z0-9][A-Z0-9] matches Canadian zip codes, such as, "VC5 B6T"
  - Caution:
    - What is [50-99]?
      - It is not {50,51,…,99}
      - It is same with [0-9]: the set contains already 5 and 9

Negative Character sets

Metacharacter	Meaning
^	Negate a character set

- Caret (^) = not one of several characters
  - Add ^ as the first character inside a character set
  - Still represents one character
  - Examples:
    - [^aeiou] matches any one character that is not a lower case vowel
    - [^aeiouAEIOU] matches any one character that is not a vowel (non-vowel)
    - [^a-zA-Z] matches any one character that is not a letter
    - see[^mn] matches "seek" and "sees", but not "seem" or "seen"
    - see[^mn] matches "see " because space matches [^mn]
    - see[^mn] does not match "see" because there is no more character after see

Metacharacters

- Metacharacters inside Character sets are already escaped:
  - Do not need to escape them again
  - Examples:
    - h[o.]t matches "hot" and "h.t"
  - Exceptions: metacharacters that have to do with character sets:]-^\
  - Examples:
    - [[\]] matches "[" or "]"
    - var[[(][0-9][)\]] matches "var()" or "var[]"
  - Exception to exception: "10[-/]10" matches "10-10" or "10/10"
    - - does not need to be escaped because it is not a range

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Shorthand character sets

Shorthand	Meaning	Equivalent
\d	Digit	[0-9]
\w	Word character	[a-zA-z0-9_]
\s	Whitespace	[\t\n\r]
\D	Not digit	[^0-9]
\W	Not word character	[^a-zA-z0-9_]
\S	Not white space	[^ \t\n\r]

- Underscore (_) is a word character
- Hyphen (-) is not a word character
  - "\d\d\d" matches "123"
  - "\w\w\w" matches "123" and "ABC" and "1_A"
  - "\w\s\w\w" matches "I am", but not "Am I"
  - "[^\d]" matches "a"
  - "[^\d\w]" is not the same with "[\D\W]" (accepts "a")

Introduced in Perl
Not in many Unix tools

(c) Paul Fodor (CS Stony Brook)

POSIX Bracket Expressions

POSIX	Description	ASCII	Unicode	Shorthand	Java
[:alnum:]	Alphanumeric characters	[a-zA-Z0-9]	[\p{L}\p{Nl} \p{Nd}]		\p{Alnum}
[:alpha:]	Alphabetic characters	[a-zA-Z]	\p{L}\p{Nl}		\p{Alpha}
[:ascii:]	ASCII characters	[\x00-\x7F]	\p{InBasicLatin}		\p{ASCII}
[:blank:]	Space and tab	[\t]	[\p{Zs}\t]	\h	\p{Blank}
[:cntrl:]	Control characters	[\x00-\x1F\x7F]	\p{Cc}		\p{Cntrl}
[:digit:]	Digits	[0-9]	\p{Nd}	\d	\p{Digit}
[:graph:]	Visible characters (anything except spaces and control characters)	[\x21-\x7E]	[^\p{Z}\p{C}]		\p{Graph}
[:lower:]	Lowercase letters	[a-z]	\p{Ll}	\l	\p{Lower}
[:print:]	Visible characters and spaces (anything except control characters)	[\x20-\x7E]	\P{C}		\p{Print}
[:punct:]	Punctuation (and symbols).	[!"\#$%&'()*+, \-./:;<=>?@\[\\\]^_`{|}~]	\p{P}		\p{Punct}
[:space:]	All whitespace characters, including line breaks	[\t\r\n\v\f]	[\p{Z}\t\r\n\v\f]	\s	\p{Space}
[:upper:]	Uppercase letters	[A-Z]	\p{Lu}	\u	\p{Upper}
[:word:]	Word characters (letters, numbers and underscores)	[A-Za-z0-9_]	[\p{L}\p{Nl} \p{Nd}\p{Pc}]	\w	\p{IsWord}
[:xdigit:]	Hexadecimal digits	[A-Fa-f0-9]	[A-Fa-f0-9]		\p{XDigit}

20

Repetition

Metacharacter	Meaning
*	Preceding item zero or more times
+	Preceding item one or more times
?	Preceding item zero or one time

- Examples:
  - apples* matches "apple" and "apples" and "applessssssss"
  - apples+ matches "apples" and "applessssssss"
  - apples? matches "apple" and "apples"
  - \d* matches "123"
  - colou?r matches "color" and "colour"

(c) Paul Fodor (CS Stony Brook)

Quantified Repetition

Metacharacter	Meaning
{	Start quantified repetition of preceding item
}	End quantified repetition of preceding item

- {min, max}
  - min and max must be positive numbers
  - min must always be included
  - min can be 0
  - max is optional
- Syntaxes:
  - \d{4,8} matches numbers with 4 to 8 digits
  - \d{4} matches numbers with exactly 4 digits
  - \d{4,} matches numbers with minimum 4 digits
  - \d{0,} is equivalent to \d*
  - \d{1,} is equivalent to \d+

(c) Paul Fodor (CS Stony Brook)

Greedy Expressions

- Standard repetition quantifiers are **<u>greedy</u>**:
  - expressions try to match the longest possible string
  - \d* matches the entire string "1234" and not just "123", "1", or "23"
- Lazy expressions:
  - matches as little as possible before giving control to the next expression part
  - ? makes the preceding quantifier into a lazy quantifier
    - *?
    - +?
    - {min,max}?
    - ??
  - Example:
    - "apples??" matches "apple" in "apples"

(c) Paul Fodor (CS Stony Brook)

Grouping metacharacters

Metacharacter	Meaning
(	Start grouped expression
)	End grouped expression

- Group a large part to apply repetition to it
  - "(abc)*" matches "abc" and "abcabcabc"
  - "(in)?dependent" matches "dependent" and "independent"
- Makes expressions easier to read
- Cannot be used inside character sets

Metacharacters

- $ Matches the ending position of the string or the position just before a string-ending newline.
  - In line-based tools, it matches the ending position of any line.
  - [hc]at$ matches "hat" and "cat", but only at the end of the string or line.
- ^ Matches the beginning of a line or string.
- | The choice (also known as alternation or set union) operator matches either the expression before or the expression after the operator.
  - For example, abc|def matches "abc" or "def".
- \A Matches the beginning of a string (but not an internal line).
- \z Matches the end of a string (but not an internal line).

Summary: Frequently Used Regular Expressions

Regular Expression	Meaning	Example
x	A character literal	"good" matches "good"
.	Any single character	"good" matches "goo."
(ab|cd)	ab or cd	"good" matches "a|g"
[abc]	a, b, or c	"good" matches "[ag]"
[^abc]	any character except a, b, or c	"good" matches "[^ac]"
[a-z]	a through z	"good" matches [a-i]oo[a-d]
[^a-z]	any character except a through z	"good" matches goo[^i-x]
\d	a digit, same as [0-9]	"good3" matches "good\d"
\D	a non-digit	"good" matches "\D\Dod"
\w	a word character	"good3" matches "goo\w\w"
\W	a non-word character	$good matches "\Wgood"
\s	a whitespace character	"good 2" matches "good\s2"
\S	a non-whitespace char	"good" matches "\Sood"
p*	zero or more occurrences of pattern p	"good" matches "a*" bbb matches "a*"
p+	one or more occurrences of pattern p	"good" matches "o+" bbb matches "b+"
p?	zero or one occurrence of pattern p	"good" matches "good?" bbb matches "b?"
p{n}	exactly n occurrences of pattern p	aaa matches "a{3}" good does not match "go{2}d"
p{n,}	at least n occurrences of pattern p	good matches "go{2,}d" good does not match "g{1,}"
p{n,m}	between n and m occurrences (inclusive)	aa matches "a{1,9}" bb does not match "b{2,9}"

Python `match` and `search` Functions

- **`re.match(r, s)`** returns a match object if the regex **`r`** matches at the start of string **`s`**

```python
import re
regex = "\d{3}-\d{2}-\d{4}"
ssn = input("Enter SSN: ")
match1 = re.match(regex, ssn)
if match1 != None:
    print(ssn, " is a valid SSN")
    print("start position of the matched text is "
        + str(match1.start()))
    print("start and end position of the matched text is "
        + str(match1.span()))
else:
    print(ssn, " is not a valid SSN")

Enter SSN: 123-12-1234  more text
123-12-1234 more text  is a valid SSN
start position of the matched text is 0
start and end position of the matched text is (0, 11)
```

(c) Paul Fodor (CS Stony Brook)

Python `match` and `search` Functions

- Invoking `re.match` returns a match object if the string matches the regex pattern at the start of the string.
  - Otherwise, it returns `None`.
- The program checks whether if there is a match.
  - If so, it invokes the match object's `start()` method to return the start position of the matched text in the string (line 10) and the `span()` method to return the start and end position of the matched text in a tuple (line 11)

Python `match` and `search` Functions

- **`re.search(r, s)`** returns a match object if the regex **`r`** matches anywhere in string **`s`**

```python
import re
regex = "\d{3}-\d{2}-\d{4}"
text = input("Enter a text: ")
match1 = re.search(regex, text)
if match1 != None:
    print(text, " contains a valid SSN")
    print("start position of the matched text is "
        + str(match1.start()))
    print("start and end position of the matched text is "
        + str(match1.span()))
else:
    print(ssn, " does not contain a valid SSN")


Enter a text: The ssn for Smith is 343-34-3490
The ssn for Smith is 343-34-3490 contains a SSN
start position of the matched text is 21
start and end position of the matched text is (21, 32)
```

(c) Paul Fodor (CS Stony Brook)

Flags

- For the functions in the **re** module, an optional flag parameter can be used to specify additional constraints
- For example, in the following statement

`re.search("a{3}", "AaaBe", re.IGNORECASE)`

The string "Aaa" matches the pattern a{3} case-insensitive

30

Findall

- **findall(pattern, string [, flags])** return a list of strings giving all nonoverlapping matches of pattern in string. If there are any groups in patterns, returns a list of groups, and a list of tuples if the pattern has more than one group

```
>>> re.findall('<(.*?)>','<spam> /<ham><eggs>')
['spam', 'ham', 'eggs']
>>> re.findall('<(.*?)>/?<(.*?)>',
    '<spam>/<ham> ... <eggs><cheese>')
[('spam', 'ham'), ('eggs', 'cheese')]
```

Findall

- **`sub(pattern, repl, string [, count, flags])`**
  returns the string obtained by replacing the (first count) leftmost nonoverlapping occurrences of pattern (a string or a pattern object) in string by **`repl`** (which may be a string with backslash escapes that may back-reference a matched group, or a function that is passed a single match object and returns the replacement string).

- **`compile(pattern, [, flags])`** compiles a regular expression pattern string into a regular expression pattern object, for later matching.

(c) Paul Fodor (CS Stony Brook)

Groups

- Groups: extract substrings matched by REs in '()' parts
  - (R) Matches any regular expression inside (), and delimits a group (retains matched substring)
  - \N Matches the contents of the group of the same number N: '(.+) \1' matches "42 42"

```python
import re
patt = re.compile("A(.)B(.)C(.)") # saves 3 substrings
mobj = patt.match("A0B1C2") # each '()' is a group, 1..n
print(mobj.group(1), mobj.group(2), mobj.group(3))
patt = re.compile("A(.*)B(.*)C(.*)") # saves 3 substrings
mobj = patt.match("A00B11C22") # groups() gives all groups
print(mobj.groups())
print(re.search("(A|X)(B|Y)(C|Z)D", "..AYCD..").groups())
print(re.search("(?P<a>A|X)(?P<b>B|Y)(?P<c>C|Z)D",
    "..AYCD..").groupdict())
patt = re.compile(r"[\t ]*#\s*define\s*([a-z0-9_]*)\s*(.*)")
mobj = patt.search(" # define spam 1 + 2 + 3") # parts of C #define
print(mobj.groups()) # \s is whitespace
```

(c) Paul Fodor (CS Stony Brook)

Groups

```
python re-groups.py
0 1 2
('000', '111', '222')
('A', 'Y', 'C')
{'a': 'A', 'c': 'C', 'b': 'Y'}
('spam', '1 + 2 + 3')
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Groups

- When a match or search function or method is successful, you get back a match object
  - **group(g) group(g1, g2, ...)** Return the substring that matched a parenthesized group (or groups) in the pattern. Accept group numbers or names. Group numbers start at 1; group 0 is the entire string matched by the pattern. Returns a tuple when passed multiple group numbers, and a group number defaults to 0 if omitted
  - **groups()** Returns a tuple of all groups' substrings of the match (for group numbers 1 and higher).
  - **start([group]) end([group])** Indices of the start and end of the substring matched by group (or the entire matched string, if no group is passed).
  - **span([group])** Returns the two-item tuple: **(start(group), end(group))**

(c) Paul Fodor (CS Stony Brook)