Intro to Computing Lab 7 February 25, 2009

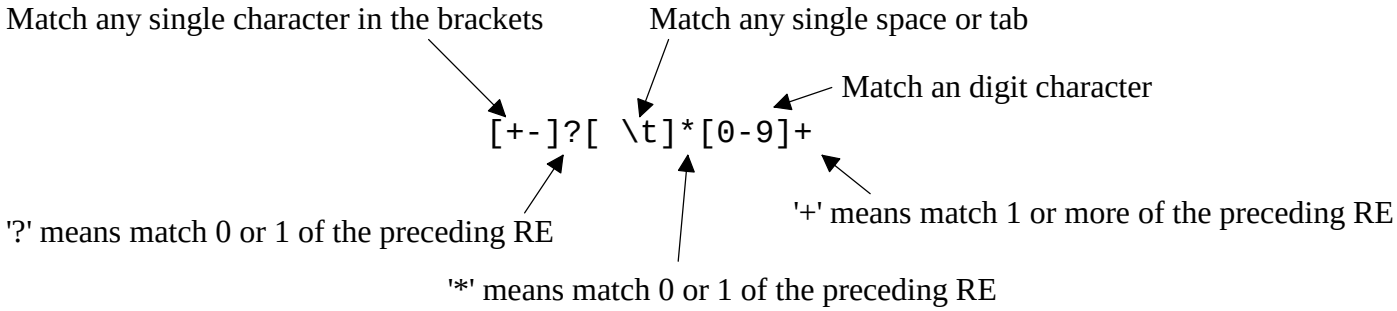
For today's lab you'll need several files, so start the lab by copying the directory P:\Math-CS\810-051-fienup\common\lab7 to your P: space (or desktop or flash drive).

Background: A common task in many programs is processing textual data (i.e., strings in Python). Regular expressions (REs) are a common tool in many programming languages to aid in pattern matching, extraction and searching-and-replacing. In Python the `re` module contains the functions for regular expressions.

A regular expression is a notational tool that allows us to specify patterns of interest. For example, we might be looking for an integer value defined as

- an optional sign character ('+' or '-') followed by
- zero or more blank spaces or tabs ('\t') followed by
- one or more consecutive digits ('0' to '9') characters

In Python the regular express for this would be:



The following table contains common *metacharacter* (i.e., special characters and symbols) used in RE notation.

Notation	Description	Example regular expression
literal	Match only the specified string literal	`Mrs`
re1 | *re2*	Match regular expression *re1* or *re2*	`Mrs|Mr|Ms|Miss`
.	Match any character, except \n	`x.y`
[ ]	Match any single character between the brackets (defines a *character class*)	`[aeiouy]`
[*x*-*y*]	Match any single character in the range from *x* to *y*	`[0-9]`
[^]	**Do not match** any character from the character class or range specified in the brackets.	`[^a-zA-Z0-9_]`
*	Match 0 or more occurrences of the preceding RE	`[a-zA-Z]*`
+	Match 1 or more occurrences of the preceding RE	`[0-9]+`
?	Match 0 or 1 occurrence of the preceding RE	`[+-]?`
{ *N* }	Match *N* occurrences of the preceding RE	`[0-9]{4}`
{ *M*, *N* }	Match *M* to *N* occurrences of the preceding RE	`[0-9]{3,6}`
^	Match at the start of string	`^Dear`
$	Match at the end of string	`bye&`
( )	Match RE enclosed by the paranthese and save as a subgroup	`([0-9]{4})`
`(*|+|?|{})?`	Apply non-greedy versions of above repetition symbols `(*|+|?|{})`	`.*?[0-9]`

If you what to match any of the special symbols `(*,+,?,.,[,],{,},|,\)` they must be preceeded by a '\'. For example, `[0-9]+\.[0-9]+` specifies a period character with digits on both sides.

The following special characters allow for even more cryptic RE notation.

Special Character	Description	Example regular expression
`\d`	Match any digit character (same as [0-9])	`\d{3}-\d{4}`
`\w`	Match any alphanumeric character (same as [A-Za-z0-9])	`\w{6,10}`
`\s`	Match any white-space character (same as `[ \t\n\r\v\f]`)	`foo\sbar`
`\b`	Match any word boundary (same as [0-9])	`\bfoobar\b`
`\nn`	Match saved subgroup *nn* (see ( ) in above table)	`name: \5`
`\A`	Match at the start of string (see ^ in above table)	`\ADear`
`\Z`	Match at the end of string (see $ in above table)	`bye\Z`

The following table describes some common `re` module functions. Python differentiates between trying to *match* a whole string to a RE pattern, and *searching* where it looks for the RE as a substring within the string. In either case, a "match object" is used to store the result of successfully finding an RE pattern. The `None` object is returned on an unsuccessful search/match.

re Function	Description
`match(`*`pattern`*`,` *`string`* `[,` *`flags = 0`*`])`	Attempts to match RE *`pattern`* to the whole *`string`* with optional *`flags`* for case-insensitive matching, etc. Returns a match object, or `None` on failure.
`search(`*`pattern`*`,` *`string`* `[,` *`flags = 0`*`])`	Returns the first occurrence of the RE *`pattern`* in the *`string`* with optional *`flags`* for case-insensitive matching, etc. Returns a match object, or `None` on failure.
`split(`*`pattern`*`,` *`string`* `[,` *`max`* `= 0])`	Splits the *`string`* into a list according to the RE *`pattern`* delimiter. An optional *`max`* parameter to limit the number of splits.
`findall(`*`pattern`*`,` *`string`* `[, flags])`	Returns a list of all non-overlapping substrings in *`string`* that match the RE *`pattern`*. Optional *`flags`* can be specified.
`finditer(`*`pattern`*`,` *`string`* `[, flags])`	Like `findall`, but returns an iterator instead of a list for all non-overlapping substrings in *`string`* that match the RE *`pattern`*. Optional *`flags`* can be specified.
`sub(`*`pattern`*`,` *`replace`*`,` *`string`* `[, max=0])`	Returns a string will all occurrences of the RE *`pattern`* in *`string`* replaced by *`replace`*. Optional *`flags`* can be specified.

Part A: You copied the following program (example1.py) from the P: already. It illustrates the usage of some of these re functions looking for the integer definition defined above.

```
import re

stringToSearch = 'This number is -   123.  The next is +8 and +++ last of 987!'

print "stringToSearch: '" + stringToSearch
result = re.search('[+-]?[ \t]*[0-9]+', stringToSearch)
if result is not None:
    print 'search:', result.group()
else:
    print 'Could not find an integer'

result = re.findall('[+-]?[ \t]*[0-9]+', stringToSearch)
print 'findall:', result

result = re.split('[+-]?[ \t]*[0-9]+', stringToSearch)
print 'split:', result

result = re.sub('[+-]?[ \t]*[0-9]+', '###', stringToSearch)
print 'sub:', result
```

Predict what the completed output of the above program would be:

```
stringToSearch: 'This number is -   123.  The next is +8 and +++ last of 987!

search:

findall:

split:

sub:
```

Write a program similar to the above that looks for valid floating point constants in a string. Some examples of valid floating point constants are: ’123’, ‘123.45’, ‘-123’, ‘+123.45e45’, ‘123e-45’

After you have the program for part A working correctly, raise your hand and we'll check your work.

Part B: Use the remaining time to develop three REs that will be useful in HW #3.

- an RE that can be used to split a string into sentences, where a s*entence* is considered to be any substring ending in a `'.'`, `':'`, `';'`, `'?'`, or `'!'`.
- an RE that can be used to split a sentence into words, where a *word* is considered to be any elements of a sentence that is separated by spaces, tabs or new lines.
- an RE that can be used to determine the number of *syllables* in a word, where the number of syllables is the number of groups of adjacent vowels with the exception of an 'e' appearing at the end of a word which does not count as a syllable.

After you have the program for part B working correctly, raise your hand and we'll check your work.

When you are done with all parts, hand in the following: both program for part A and B

If you don’t get all parts done in lab, don’t worry about it. Try to have the lab completed by next lab.