Regular Expressions

Introduction:

Regular Expressions are patterns that we can specify and use to search and replace text in strings (and files, which are just a sequence of strings).
Python (as well as many other languages) includes a module to perform various operations on regular expressions. In these lectures we will cover the form of regular expressions, what functions/methods can take regular expressions as arguments, the how to use the results of matching regular expressions against text: match groups.

In the first lecture we will discuss the components of regular expression patterns. We will discuss each component individually, and ways to combine them into more complicated regular expressions (just as we studied the syntax of a few simple control structures in Python, which can be combined into more complicated control structures). We will cover many but not all of the patterns usable in regular expressions (there are entire books on regular expressions).

In the second lecture we will examine functions and methods that take regular expressions as arguments and produce results. Typically they match a regular expression pattern against some text string, and return information about whether the match succeeded, and what parts of the pattern matched which parts of the text.

Python's module for doing these operations is named re. There is a special tester module that accompanies these lectures, which you can download and run to experiment with regular expressions and learn how they match text strings.

For more complete information about regular expressions, see Section 6.2 of the Python Standard Library.

Lecture 1

General Rule of Matching:
Regular expressions match the most number of characters possible (called a greedy algorithm; there are patterns that match the fewest number of characters possible; we will mention but not discuss nor use those patterns).

Matching:
Characters generally match themselves, except for the following...

Metacharacters

.	Matches any single character exception (newline: \n)
[]	Matches one character specified in []; e.g., [aeiou]
[^]	Matches one character NOT specified in [] after ^; e.g., [^aeiouy]
-	Matches one character in range in []: e.g., [0-9] matches any digit

Anchors (these don't match characters)
^ matches beginning of line (when not used in [])
$ matches end of line

Patterns: R, Ra, Rb are regular expression patterns
RaRb Matches a sequence of Ra followed by Rb
Ra|Rb Matches either alternative Ra or Rb
R? Matches regular expression R 0/1 time: R is optional
R* Matches regular expression R 0 or more times
R+ Matches regular expression R 1 or more times
R{m} Matches regular expression R exactly m times: e.g., R{5} = RRRRR
R{m,n} Matches regular expression R at least m and at most n times:
R{3,5} = RRR|RRRR|RRRRR = RRRR?R?

-----Included but never used
R??,R*?,R+?,R{m,n}? The postfix ? means match as few characters possible (not the most, so not greedy). We will not use these patterns.

Parentheses/Parenthesized Patterns
Parentheses are used for grouping, but can also remember subpatterns (this is also called a "Capturing Group).
By placing subpattern R in parentheses, the text matching R will be remembered (either by its number, starting at 1, or its name, if named) in a group, for use later in the pattern or when extracting information from the matched text.

(R) Matches R and delimits a group (1...) (remembers/capatures matched text)

(?P<name>R) Matches R and remembers/captures matched text in a group using name for the group (it is still numbered as well); see (?P=name) and groupdict method below for use of "name".

(?:R) Matches R but does not remember/capture matched text in a group
So, there () are used only for grouping, not capturing groups;
?: is useful when you want the minimum number (no redundant groups)

-----Included but never used
(?P=name) Matches remembered text with name (for backreferencing which text)

(?=R) Matches R, doesn't remember matched text/consume text matched
For example (?=abc)(.*) matches abcxyz with group 1 'abcxyz';
it doesn't match abxy because this text doesn't start with abc

(?!R) Matches anything but R and does not consume input needed for match
(hint: != means "not equal", ?!R means "not matching R)

Context

- matches itself if not in [], and if not between two characters

Special characters are treated as themselve in []: e.g, [.] matches literal .

Generally, if interpreting a character makes no sense one way, try to find another way to interpret it that fits the context

Escape Characters with Special Meanings

\	Used before .|[]-?*+{}()^$\ (and others) to specify a special character
\#	Backreferencing group # (numbered from 1, 2, ...): see (R) above
\t	tab
\n	newline
\r	carriage return
\f	formfeed
\v	vertical tab

\d	[0-9]	Digit
\D	[^0-9]	non-Digit
\s	[\t\n\r\f\v]	White space
\S	[^ \t\n\r\f\v]	non-White space
\w	[a-zA-Z0-9_]	alphabetic (or underscore): Word character
\W	[^a-zA-Z0-9_]	non non alphabetic: non-Word character

Interesting Equivalences

a+ == aa*

a(b|c)d == a[bc]d only if b and c are single characters

R{0,1} == R?

[\t]* == (|\t)* but is different from (*|\t*)

Problems:

Write the smallest pattern that matches the required characters. Check your patterns with the Regular Expression Tester (see the Sample Programs link) to ensure they match correct exemplars and don't match incorrect ones. Note that for a match, group #0 should include all the required characters.

1. Write a regular expression pattern that matches the strings Jul 4, July 4, Jul 4th, July 4th, July fourth, and July Fourth.
   Hint: my re pattern was 24 characters.

2. Write a regular expression pattern that matches strings representing times on a 12 hour clock. An example time is 5:09am or 11:23pm. Allow only times that are legal (not 1:73pm not 13:02pm)
   Hint: my re pattern was 32 characters.

3. Write a regular expression pattern that matches strings representing phone numbers of the following form.

Normal: a three digit exchange, followed by a dash, followed by a four digit number: e.g., 555-1212

Long Distance: a 1, followed by a dash, followed by a three digit area code enclosed in parentheses, followed by a three digit exchange, followed by a dash, followed by a four digit number: e.g., 1-(800)555-1212

Interoffice: a single digit followed by a dash followed by a four digit number: e.g., 8-2404.

Hint: my re pattern was 30 characters; note that you must use \(and \) to match parentheses.

4. Write a regular expression pattern that matches strings representing simple integers described by the EBNF rules
   digit <= 0|1|2|3|4|5|6|7|8|9
   integer <= [-|+]digit{digit}

   Hint: my re pattern was 7 characters.

5. Write a regular expression pattern that matches strings representing normalized integers (each number is either an unsigned 0 or is unsigned or signed and starts with a non-0 digit) with commas in only the correct positions.
   Hint: my re pattern was 30 characters.

6. Write a regular expression pattern that matches strings representing float values. They are unsigned or signed (but not normalized: see 5) and any number of digits before or after a decimal point (but there must be at least one digit either before or after a decimal point: e.g., just . is not allowed) followed by an optional e or E followed by an unsigned or signed integer (again not normalized).
   Hint: my re pattern was 36 characters.

7. Write a regular expression pattern that matches strings representing trains (specified in Chapter Exercise 7 in the EBNF lecture).
   Hint: my re pattern was 15 characters.

Lecture 2

Generally, the functions discussed in this lecture operate on a regular expression pattern (specified by a string) and text (also specified by a string). These functions produce information (groups: see the parenthesized patterns above) related to attempting to match the pattern and text: which parts of the text matched which parts of the pattern.

We can use the compile function to compile a pattern (producing a regex), and then call methods on that regex as an object to perform the same operations as the functions, but more efficiently if the pattern is to be used repeatedly (since the pattern is compiled into the regex once, not in each function call).

We will omit discussing/using the [,flags] option in this discussion, but see section 6.2 of the Python Library Documentation for a discussion of A/ASCII, DEBUG, I/IGNORECASE, L/LOCALE, M/MULTILINE, S/DOTALL, and X/VERBOSE.

re functions: called like re.match(...) the module name prefaces the function

Returns a regex (compiled pattern) object (see calling methods on regex below)
compile (pattern, [,flags]) Creates compiled pattern object

Returns a match object, consisting of tuple of groups (0,1,...)
match (pattern, text [,flags]) Matches start at the text's beginning
search (pattern, text [,flags]) Matches can start anywhere in the text

re.match ("(a+)b","aaab") matches; re.match ("(a+)b","xaaab")) doesn't match
re.search("(a+)b","aaab") matches; re.search("(a+)b","xaaab") matches

Returns a list of string/of tuples of string (the groups), specifying matches
findall (pattern, text [,flags]) Matches can start anywhere in the text;
the next attempted match starts one character after the previous match terminates.
If the pattern has groups, then the string matching each group is included in the resulting list too: use ?: to avoid these groups
re.findall('a*b','abaabcbdabc') returns ['ab', 'aab', 'b', 'ab']
re.findall('((a*)(b))','abaabcbdabc') returns
[('ab','a','b'), ('aab','aa','b'), ('b','','b'), ('ab','a','b')]

Returns a iterable of the information returned by findall (ignore this one)
finditer (pattern, text [,flags]) Returns iterable equivalent of findall

Returns a list of strings: much like calling text.split(...)
split (pattern, text [,maxsplit, flags]) like the text.split(...) method,
but using a regular expressions pattern to determine how to split the text: re.split('\.|-', 'a.b-c') returns ['a','b','c'],
spliting on either . (which must be written here as \.)or -
which standart string split function, text.split(...) can't do;
note that 'a.b-c'.split(".-") splits on '.-' both . followed by -, so in this case it fails to split anywhere, since '.-' is not in the text at all.
If the pattern has groups, then the text matching each group is included in the resulting list too: use ?: to avoid these groups
So re.split(';+' ,'abc;d;;e') returns ['abc', 'd', 'e'] and
re.split('(;+)','abc;d;;e') returns ['abc', ';', 'd', ';;', 'e']

Returns a string

```
 sub      (pattern, repl, text, [,count, flags])
          in text, replace pattern by repl (which may refer to matched
          groups via \# (e.g. \1) or \g<#>, (e.g., \g<1>), or \g<name>
          (where name comes from ?P<name>) or a function that is passed a
          match object); if there is no match, then it just returns the
          text parameter's value, unchanged
        re.sub('(a+)','"as"','aabcaaadaf') returns    "as"bc"as"d"as"f
        re.sub('(a+)','(\g<1>)','aabcaaadaf') returns (aa)bc(aaa)d(a)f
 subn     same as sub but returns a tuple: (new string, number of subs)
 escape   (string)                    string with nonalphanum back-slashed
```

In findall and sub/subn, only non-overlapping patterns are found/replaced:
in text aaaa there are two non-overlapping occurrence of the pattern aa:
starting in index 0 and 2 (not in index 1, which overlaps with the previous
match in indexes 0-1).

---

regex (compiled pattern) object methods (see the compile method above, which
produces regexes) are called like c = re.compile(p). It is then efficient to
call c.match(...) many times. Calling re.match(p,...) many times with the same
pattern recompiles and matches the pattern each time re.match is called;
whereas c = re.compile(p) compiles the pattern once and c.match(...) matches it
each time it is called.

Using this feature allows us to compile a pattern and reuse it for all the
operations above: re.match(p,s) is just like re.compile(p).match(s); if we
are doing many matches with the same pattern, compile the pattern once and
use it with the match method below many times (as illustrated above).

pos/endpos are options that specify where in text the match starts and ends
(from pos to endpos-1). pos defaults to 0 (the beginning of the text) and
endpos defaults to the length of the text so endpos-1 is its last character).

Each of the re functions above has an equivalent method using a compiled
pattern to call the method, but omitting the pattern from its argument list.

```
 match    (text [,pos][,endpos])          See match above, with pos/endpos
 search   (text [,pos][,endpos])          See search above, with pos/endpos
 findall  (text [,pos][,endpos])  See findall above, with pos/endpos
 finditer (text [,pos][,endpos])  See finditer above, with pos/endpos
 split    (text [,maxsplit])              See split above, with pos/endpos
 sub      (repl, text [,count])    See sub above, with pos/endpos
 subn     (repl, text [,count])    See subn above, with pos/endpos
```

So, for example, instead of writing

```
 for line in open_file:
   ...re.match(pattern_string,line)
```

which implicitly compiles the same pattern_string during each loop iteration
(hen re.match executes) we can write

```
pattern= re.compile(pattern_string)
for line in open_file:
  ...pattern.match(line)
```

which explicitly compiles the pattern_string once, before the loop executes,
and calls "match" on it during each loop iteration. See the grep.py module
in the remethods download that accompanies this lecture for code that calls
re.compile.

Match objects and Groups

Match objects record information about which parts of a pattern match text.
Each group (referred to by either its number or an optional name) can be used
as an argument to a function that specifies information about the start, end,
span, and characters in the matching text.

Calling match/search produces None or a match object
Calling findall produces None, a list of strings (if there are no groups) or a
 list of tuples of strings (if there are groups, with the tuple index
 representing the each group #)
Calling finditer produces None or an iterable of groups (ignore this one)

Each group is indexed by a number or name (only when the group was delimited by
 (?P<name>)); group 0 is all the character in the match, groups 1-n are for
  delimited matches inside. For example, in the pattern (a)(b(c)(d)) the a is
  in group 1, the b is in group 2, c is in group 3, and d in is group 4:
  groups are numbered by in what order we reach their OPENING parenthesis.
  Note that if a parenthesized expression looks like (?:...) it is NOT a group.
  So in (a)(?:b(c)(d)) the a is in group 1, the b is in NO group, c is in
  group 2, and d in is group 3:

If a group matches multiple times (e.g., a(.)*c), only its last match is
available, so axyzc has group 1, the (.) group, is bound to the character z.
If we wrote this as a(.*)c the (.*) group is bound to the characters xyz

Printing the groups of match object prints a tuple of the matching characters
for each group 1-n (not group #0)

We can look at each resulting group by its number (including group #0), using
any of the following methods that operate on match objects

```
  group(g)              text of group with specified name or number
  group(g1,g2, ... )  tuple of text of groups with specified name or number
  groups()              tuple of text of all groups (can iterate over tuple)
  groupdict()           text of all groups as dict (see ?P<name>)
```

start([group])	starting index of group (or entire matched string)
end([group])	ending index of group (or entire matched string)
span([group])	tuple: (start([group]), end([group]))

Unzip remethods.zip and examine the phonecall.py and readingtest.py modules for examples of Python programs that use regular expressions (and groups) to perform useful computations.

A Simple but Illustrative Example:

```
phone = r'^(?:\((\d{3})\))?(\d{3})[-.](\d{4})$'
m = re.match(phone,'(949)824-2704')
assert m != None, 'No match'
area, exchange, number = [int(i) if i != None else None for i in m.group(1,2,3)]
print(area, exchange, number)
```

1) Here, phone is a pattern anchored at both ends.

(a) It starts with ^(?:\((\d{3})\))?...
controlling an optional area code. The ?: means that the parentheses are not used to create a group, but are used with the ? (option) symbol. Inside it is \((\d{3})\): a left parenthesis, group 1 which consists of any 3 digits, and a right parenthesis.

(b) Next is (\d{3}) group 2, which consists of any 3 digits.

(c) Next is [-.] that is one symbol, either a - or . (not in a group).

(d) Next is (\d{4}) group 3, which consists of any 4 digits.

2) Calling the re.match function matches the pattern against some text, it returns a match object that is bound to m.

3) If the match m is None, there is no match (raises AssertionError exception).

4) Converts every non-None string from groups 1, 2, and 3 into an int.

5) Prints the the groups

Try also replacing line 2 by

```
m = re.match(phone,'824-2704')              # area is None
m = re.match(phone,'(949)824.2704')                 # . instead of -
m = re.match(phone,'(94)824.2704')                  # No match
```

Also, we can replace the first two lines by the following equivalent lines

```
phone_pat = re.compile(r'^(?:\((\d{3})\))?(\d{3})[-.](\d{4})$')
m = phone_pat.match('(949)824-2704')
```

Loose Ends:

1) Raw Strings

When writing regular expression pattern strings as arguments in Python it is best to use raw strings: they are written in the form r'...' or r"...". These should be used because of an issue dealing with using the backslash character in patterns, which is sometimes necessary. For example, in regular strings when you write '\n' Python turns that into a 1 character string with the newline character: len('\n') is 1. But with raw strings, writing r'\n' specifies a string with a backslash followed by an n: len(r'\n') is 2. Normally this isn't a big issue because writing '\d' or' *' in regular strings doesn't generate an escape character, since there is no escape character for d or (so len('\d') and len('*') is 2.

2) **d in function/method calls (where d is a dict)

If we call a function we can specify **d as one or more of its arguments. For each **d, Python uses all its keys as parameter names and all its values as default arguments for these parameter names. For example

```
f(**{'b':2, 'a':1, 'c':3} ) is translated by Python into f(b=2,a=1,c=3)
```

Note that this is useful in regular expressions if we use the (?P<name> ...) option and then the groupdict() method for the match it produces.

There is also a version that works the other way. Suppose we have a functions whose header is

```
def f(x,y,**kargs):  # The typical name is **kargs
```

if we call it by f(1,2,a=3,b=4,c=5) then

```
x     is bound to 1
y     is bound to 2
kargs is bound to a dictionary {'b':4, 'a':3, 'c':5}
```

See the argument/parameter matching rules from the review lectuer for a complete description of what happens.

So (in reverse of the order explained above) ** as a parameter creates a dictionary of "extra" named-arguments suppled when the function is called, and ** as an argument supplies lots of named-arguments to the function call. We will cover this information again when we examine inheritance

The parse_phone_named method (in phoncecall.py) uses this language feature.

3) Translation of a Regular Expression Pattern into a NDFA

How do the functions/methods in re compile a regular expression and match it against text? It translates every regular expression into a non-deterministic finite automaton (see Programming Assignment #1), and then matches against the text (ibid) to see if the match succeeds (reaches the special last state).

The general algorithm (known as Thompson's Algorithm) is a bit beyond the scope of this course and uses a concept we haven't discussed (epsilon transitions), but you can look up the details if you are intereseted. Here is an example for the regular expression pattern ((a*|b)cd)+. It produces an NDFA described by

start;a;1;a;2;b;2;c;3
1;a;1;a;2
2;c;3
3;d;start;d;last
last

This pattern matches a text string by starting in state 'start' and exhausting all the characters and having 'last' in its possible states.

Problems:

Write functions using regular expression patterns

8. Write a function named contract that takes a string as a parameter. It substitutes the word 'goal' to replace any occurrences of variants of this word written with any number of o's, e.g., 'gooooal') in its argument. So calling contract('It is a goooooal! A gooal.') returns 'It is a goal! A goal.'.

9. Write a function named grep that takes a regular expression pattern string and a file name as parameters. It returns a list of 3-tuples consisting of the file-name, line number, and line of the file, for each line whose text matches the pattern. Hint: Using enumerate and a comprehension, this is a 3 line function, but you can use explicit looping in a longer function.

10. Write a function named name_convert that takes two file names as parameter. It reads the first file (which should be a Python program) and writes each line into the second file, but with identifiers originally written in camel notation converted to underscore notation: e.g. aCamelName converts to a_camel_name. Camel identifiers start with a lower-case letter followed by

upper/lower-case letters and digits: each upper-case letter is preceded by an underscore and turned into a lower-case letter.