Regular Expressions

- Background
- Sets of strings
- Stating a regular expression (simple)
- Python **re** module (simple)
- A bit of theory
- Stating a regular expression (more complex)
- Python **re** module (more complex)
- Using regexes for control flow

String patterns

- We all use searches where we provide strings or substrings to some module or mechanism
  - Google search terms
  - Filename completion
  - Command-line wildcards
  - Browser URL completion
  - Python string routines find(), index(), etc.
- Quite often these searches are simply expressed as a particular pattern
  - An individual word
  - Several words where some are strictly required while some are not
  - The start or end of particular words -- or perhaps just the string appearing within a larger string
- This works well if strings follow the format we expect…

String patterns

- Sometimes, however, we want to express a more complex pattern
  - The set of all files ending with either ".c" or ".h"
  - The set of all files starting with "ical".
  - The set of all strings in which "FREQ" appears as a string (but not "FREQUENCY" or "INFREQUENT", but "fReQ" is fine)
  - The set of all strings containing dates in MM/DD/YYYY format.
- Such a variety of patterns used to require language-specific operations
  - SNOBOL
  - Pascal
- More troubling was that most non-trivial patterns required several lines of code to express (i.e., a series of "if-then-else" statements)
  - This is a problem as the resulting code can obscure the patterns for which we are searching
  - Even worse, changing the pattern is tedious and error-prone as it means changing the structure of already written code.

Regular expressions

- Needed: a language-independent approach to expressing such patterns
- Solution: a **regular expression**
  - Sometimes called a **regex** or **regexp**
- They are written in a formal language and have the property that we can build very fast recognizers for them
- Part of a hierarchy of languages
  - Type 0: unrestricted grammars
  - Type 1: context-sensitive grammars
  - **Type 2: context-free grammars**
  - **Type 3: regular grammars**
- Type 2 and 3 grammars are used in Computer Science
  - Type 2 is used in parsers for computer languages (i.e., compilers)
  - Type 3 is used in regular expressions and lexical analyzers for compilers

grep

- We already can use regular expressions in Unix at the command line
- The grep utility accepts two sets of arguments
  - grep: **global regular expression print**
  - argument 1: A regular expression
  - argument 2: A set of files through which grep will try to find strings matching the regex
- The syntax for a regex is grep is somewhat similar to what we will use in Python
  - grep is a very old tool (i.e., from 1973)
  - superseded somewhat by fgrep (fixed-string grep)
  - a variety of extensions, optimizations, etc. exist
- Example: search for variants on "apple"

grep

contents of fruitstuff.txt

```
apple
apples
Apple Pie
APPLE SUX!
apple-
apple-fruit
"Apple is the greatest!"
My best friend is an apple.
pineapple
Crabapple
fruit-apple
```

```
unix$ grep -i ^apple fruitstuff.txt
apple
apples
Apple Pie
APPLE SUX!
apple-
apple-fruit
```

```
unix$ grep apple fruitstuff.txt
apple
apples
apple-
apple-fruit
My best friend is an apple.
pineapple
Crabapple
fruit-apple
```

```
unix$ grep ^a.ple fruitstuff.txt
apple
apples
apple-
apple-fruit
```

```
unix$ grep -w apple fruitstuff.txt
apple
apple-
apple-fruit
My best friend is an apple.
fruit-apple
```

```
unix$ grep apple$ fruitstuff.txt
apple
pineapple
Crabapple
fruit-apple
```

More general regular expressions

- Our grep examples were relatively simple
- Sometimes we want to denote more complex sets of strings
  - strings where the beginning and end match a pattern, while everything in between can vary
  - all possible spellings of a particular name
  - match non-printable characters
  - catch possible misspellings of a particular word
  - match Unicode code points
- And we may want even more:
  - when matching patterns to strings, extract the actual match itself
  - look for strings where the matched pattern repeats exactly later in the same string
  - extract multiple matches from one string

Metasymbols

- Fully-fledged regexes initially look intimidating because of the metasymbols
- However, all that is required to understand them is patience
- Regexes never loop…
- … nor are they ever recursive
- Understanding them means reading from left-to-right!
- However, first some metasymbols

symbol/example	meaning
`.`	match any char except \n
`a*`	zero or more reps of 'a'
`a+`	one or more reps of 'a'
`a?`	zero or one rep of 'a'
`a{5}`	exactly 5 reps of 'a'
`a{3,7}`	3 to 7 reps of 'a'
`[abc]`	any one character in the set {a, b, c}
`[^abc]`	any one character not in the set of {a, b, c}
`a|b`	match 'a' or 'b'
`(...)`	group a component of symbols in the regex
`\`	escape any metasymbol (caution!)

Special pattern elements

symbol	meaning
`\d`	Any decimal digit character
`\w`	Any alphanumeric character
`\s`	Any whitespace character (\t\n\r\f\v)
`\b`	Empty string at a word boundary
`^`	match 0 characters at the start of the string
`$`	match 0 characters at the end of the string
`\D`	match any non-digit character (opposite of \d)
`\W`	match any non-alphanumeric character (opposite of \w)
`\S`	match any non-whitespace character
`\B`	empty string (i.e., 0 characters) not at a word boundary
`\number`	matches text of group **`number`**

Python regular expressions

- The **re** module
  - Introduced into Python in version 1.5
  - (Don't use the **regex** module which is an older release of a regular-expression library)
  - Use to be slower than regex, but is now as fast if not faster
  - Supports **named groups**
  - Supports **non-greedy matches** (we'll cover this later)
- Note:
  - Regular expression syntax is generally the same from language to language and library to library (e.g., Python, Perl, Ruby)
  - However, sometimes there are differences in the way some features are expressed (e.g., groups, escaped characters)
  - Whenever you move to different implementations, always have the library reference nearby.

Simple example

```
>>> text1 = 'Hello spam...World'
>>> text2 = 'Hello spam...other'


>>> import re
>>> matchobj = re.match('Hello.*World', text2)
>>> print matchobj
None


>>> if re.match('Hello.*World', text2):
...     print "It's the end of the World"
... else:
...     print "The end of the world is nowhere in sight"
...
The end of the world is nowhere in sight
>>
```

Previous example

- The regular-expression match was applied to string **text2**
  - Regex specified a string with "Hello" followed by 0 or many characters followed by "World"
  - The match did not succeed, therefore the value None was returned
  - In Python, None may be used as part of a conditional expression (i.e., has similar meaning to "False".
- Even though the name of the RE method was match(), we did not use any syntax to extract out some result of the match
  - Which is just as well as there was no match.
  - However, if we wanted to extract out the some result, we must use parentheses.
- Let's look at the example again, but this time include the other string in our use of match
  - Note that in the following example the "import re" is left out (i.e., we assume it was executed earlier in the session)

Simple example

```
>>> text1 = 'Hello spam...World'
>>> text2 = 'Hello spam...other'

>>> matchobj = re.match('Hello(.*)World', text1)
>>> print matchobj
<_sre.SRE_Match object at 0x10043b8a0>

>>> hello_list = [text1, text2]
>>> for t in hello_list:
...     matchobj = re.match('Hello(.*)World', t)
...     if matchobj:
...             print t, " --> match --> ", matchobj.group(1)
...     else:
...             print t, " --> no matches"
...
Hello spam...World  --> match -->   spam...
Hello spam...other  --> no matches
```

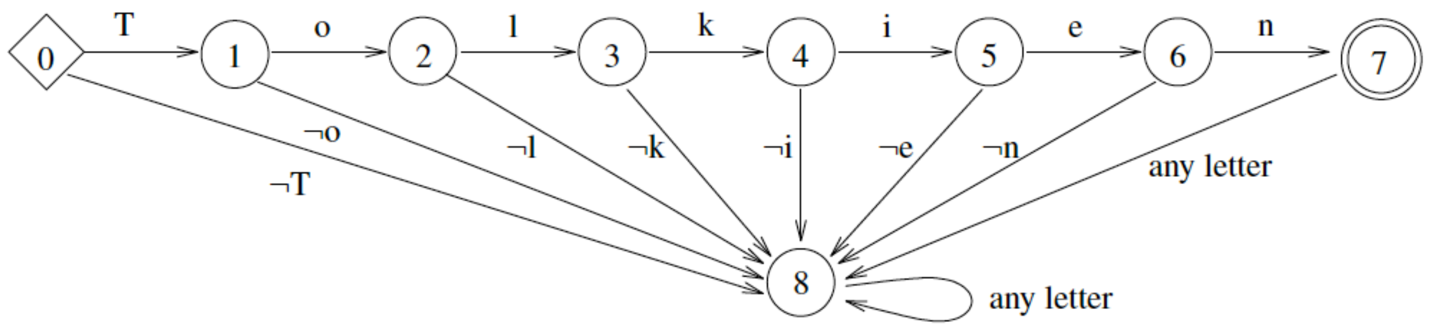
Previous example

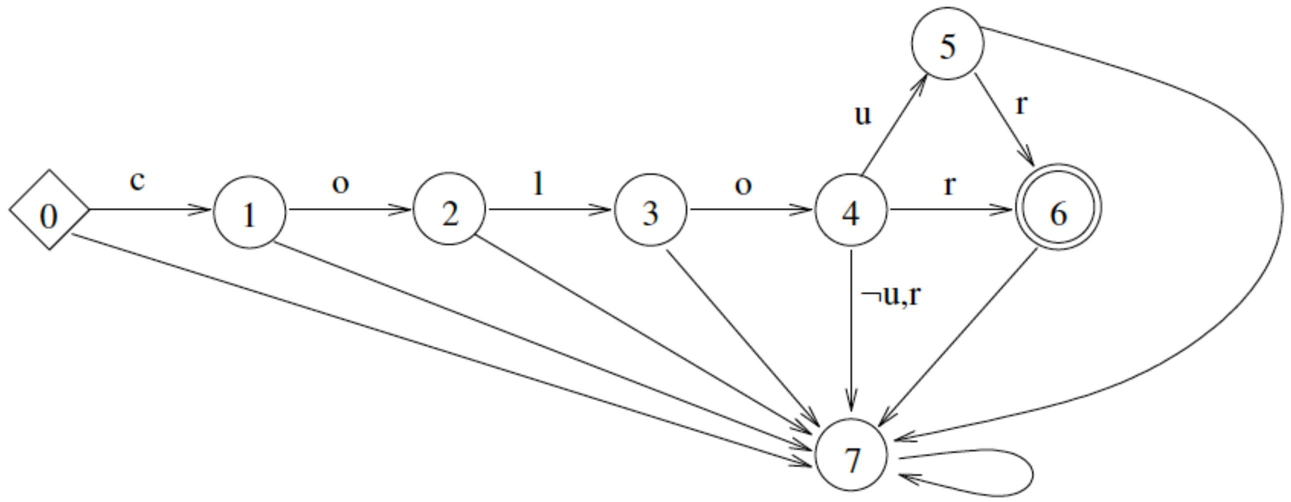
- The match did succeed when applied to **text1**
  - The result is a **match object**
  - This has an interface which is used to extract matched substrings
  - In this case, we extracted the substring matching the pattern in the parentheses
- The parameter passed to **group** corresponds to the order of left parenthesis
  - A regular expression can have several such groups given the use of parentheses
  - Groups can even be nested (i.e., nested parentheses)…
  - … but **they can never overlap**.
  - Programmers make extensive use of groups in regular expressions
  - It helps make code more robust and less dependent on an exact format.

Speed concerns

- So far we have seen the use of re.match
- For occasional regex matching this is fine
- However, each time the match is performed the Python interpreter must re-interpret the regex
  - This means the regex must be re-parsed and the state machine re-constructed.
  - If we want to search many strings using the same regex, it makes sense to eliminate the overhead of repeating this work.
  - To eliminate the repeated work, we must **compile the pattern**
- When using this style of regex matching, we work with a pattern object
  - Resulting code is much, much faster
  - Note, however, the compilation itself takes up some cycles.
  - For now, just be aware there exist the two styles of matching (i.e., one directly using re.match(), the other using a pattern object returned from re.compile()).

Regex as a state machine


0
T
1
o
2
l
3
k
4
i
5
e
6
n
7
¬o
¬T
¬l
¬k
¬i
¬e
¬n
any letter
8
any letter


5
u
r
0
c
1
o
2
l
3
o
4
r
6
¬u,r
7