Problem Solving

- More useful problem-solving: formatting mail replies
  - In the "old days" e-mail was via a Unix command called **mail**
  - You could pipe stuff into and out of **mail**.
- **Problem 7: Transforming an e-mail into the start of a reply**
  - Extract fields from the original e-mail's header
  - Use these to construct the reply's header
  - Take the body of the e-mail and indent it with a special character sequence.
- Idea is that this text could then be the starting point of a reply.

Example: E-mail replies

```
From elvis Thu Feb 29 9:25 2013
Received: from elvis@localhost by tabloid.org (8.11.3) id KA8CMY
Received: from tabloid.org by gateway.net (8.12.5/2) id N8XBK
To: nigelh@cmpt.uvic.ca (R. Nigel Horspool)
From: elvis@tabloid.org (The King)
Date: Thu, Feb 29 2013 9:25
Message-Id: <2010022939939.KA8CMY@tabloid.org>
Subject: Be seein' ya around
Reply-To: elvis@hh.tabloid.org
X-Mailer: Madam Zelda's Psychic Orb [version 3.7 PL92]

Sorry I haven't been around lately. A few years back I checked
into that ole heartbreak hotel in the sky, ifyaknowwhatImean.
The Duke says "hi".
    Elvis
```

Original e-mail from the spirit world.

Example: E-mail replies

```
To: elvis@tabloid.org (The King)
From: nigelh@cmpt.uvic.ca (R. Nigel Horspool)
Subject: Be seein' ya around

On Thu, Feb 29 2013 9:25 The King wrote:
|> Sorry I haven't been around lately. A few years back I checked
|> into that ole heartbreak hotel in the sky, ifyaknowwhatImean.
|> The Duke says "hi".
|>    Elvis
```

What we want to produce

E-mail replies

- The original e-mail structure was:
  - header lines
  - a single blank line
  - body lines
- The reply's header needs:
  - The original sender (from the "To:" field)
  - The original recipient (from the "From:" field)
  - The original subject (from the "Subject:" field)
- The reply's body needs:
  - The original text
  - The date of the original e-mail (from the "Date:" field)
- We can search the header for the required fields...
  - ... and use the blank line to indicate when we switch to processing the body.
  - This suggests a loop structure

Form letter

```
#!/usr/bin/python

import sys
import re

def main():
    for line in sys.stdin:

        # process the header in this "for" body be extracting required
        # fields

        # if current line is blank, then break out of the loop

    print header stuff

    for line in sys.stdin:

        # at this point we are reading in the body line by line
        # so make sure we indent with the special string sequence

# that's all
```

Example: E-mail replies

```
From elvis Thu Feb 29 9:25 2013
Received: from elvis@localhost by tabloid.org (8.11.3) id KA8CMY
Received: from tabloid.org by gateway.net (8.12.5/2) id N8XBK
To: nigelh@cmpt.uvic.ca (R. Nigel Horspool)
From: elvis@tabloid.org (The King)
Date: Thu, Feb 29 2013 9:25
Message-Id: <201002939939.KA8CMY@tabloid.org>
Subject: Be seein' ya around
Reply-To: elvis@hh.tabloid.org
X-Mailer: Madam Zelda's Psychic Orb [version 3.7 PL92]

Sorry I haven't been around lately. A few years back I checked
into that ole heartbreak hotel in the sky, ifyaknowwhatImean.
The Duke says "hi".
    Elvis
```

E-mail replies

- Some of the required matches are pretty straight forward:
  - Matching the Subject
  - Matching the Date
- The "From" data is a bit trickier
  - There are two "From" fields in the header.
  - We want the data in the field formed like "From:" (i.e., with a colon)
  - The field contains both an e-mail address and a person's name
  - We want both.
  - Regex must match parentheses (although parentheses are used to group matched characters): must escape the right parentheses

```
From elvis Thu Feb 29 9:25 2013
Received: from elvis@localhost by tabloid.org (8.11.3) id KA8CMY
Received: from tabloid.org by gateway.net (8.12.5/2) id N8XBK
To: nigelh@cmpt.uvic.ca (R. Nigel Horspool)
From: elvis@tabloid.org (The King)
Date: Thu, Feb 29 2013 9:25
Message-Id: <2010022939939.KA8CMY@tabloid.org>
Subject: Be seein' ya around
Reply-To: elvis@hh.tabloid.org
X-Mailer: Madam Zelda's Psychic Orb [version 3.7 PL92]
```

```
    for line in sys.stdin:
        if (re.search("^\s*$", line)):
            break

        matchobj = re.search("^Subject: (.*)$", line)
        if (matchobj):
            subject = matchobj.group(1)
            continue

        matchobj = re.search("^Date: (.*)$", line)
        if (matchobj):
            date = matchobj.group(1)
            continue

        matchobj = re.search("^Reply-To: (.*)$", line)
        if (matchobj):
            reply_address = matchobj.group(1)
            continue
```

```
From elvis Thu Feb 29 9:25 2013
Received: from elvis@localhost by tabloid.org (8.11.3) id KA8CMY
Received: from tabloid.org by gateway.net (8.12.5/2) id N8XBK
To: nigelh@cmpt.uvic.ca (R. Nigel Horspool)
From: elvis@tabloid.org (The King)
Date: Thu, Feb 29 2013 9:25
Message-Id: <2010022939939.KA8CMY@tabloid.org>
Subject: Be seein' ya around
Reply-To: elvis@hh.tabloid.org
X-Mailer: Madam Zelda's Psychic Orb [version 3.7 PL92]
```

```
# for continued

        matchobj = re.search(r"^From: (\S+) \(([^()]*)\)", line)
        if (matchobj):
            reply_address, from_name = matchobj.group(1), matchobj.group(2)
            continue
```

E-mail replies

```
    print "To: %s (%s)" % (reply_address, from_name)
    print "From: nigelh@cs.uvic.ca (R. Nigel Horspool)"
    print "Subject: Re: %s" % (subject)
    print

    print "On %s %s wrote:" % (date, from_name)
    for line in sys.stdin:
        line = line.rstrip('\n')
        line = re.sub("^", "|> ", line)
        print line

if __name__ == "__main__":
    main()
```

Problem Solving

- Our last problem is a curious one
- **Problem 8: Add commas to a large number to improve readability**
  - Example: cdn_population = 33894000
  - Yet we want this to appear in output with commas ("33,894,000")
- How do we do this mentally?
  - We group by threes...
  - ... by starting from the right and heading left
  - If a group of three or fewer numbers remains on the leftmost end, that's okay
- But how can a regex help us here?
  - Don't they go from left-to-right?
  - The key is to use one of the features available that are referred together as **lookaround**

Leading up to our answer...

- Let's start instead with a simpler problem
- Given a string:
  - "This is Mikes bicycle"
- Change it so that the possessive it properly punctuated
  - "This is Mike's bicycle"
- There are several ways to to this already
  - We use re.sub()
  - The pattern and replacement can vary given the style of regex.

Giving Mike a bicycle

```
#!/usr/bin/python

import re

s = "This is Mikes bicycle"
print "Before -->", s
s = re.sub("Mikes", "Mike's", s)
print "After -->", s, "\n"

s = "This is Mikes bicycle"
print "Before -->", s
s = re.sub(r"\bMikes\b", "Mike's", s)
print "After -->", s, "\n"

s = "This is Mikes bicycle"
print "Before -->", s
s = re.sub(r"\b(Mike)(s)\b", r"\1'\2", s)
print "After -->", s, "\n"
```

```
Before --> This is Mikes bicycle
After --> This is Mike's bicycle

Before --> This is Mikes bicycle
After --> This is Mike's bicycle

Before --> This is Mikes bicycle
After --> This is Mike's bicycle
```

Lookaround

- Recall that we already have some operators that match **positions**
  - ^
  - $
  - \b
- That is, they do not match individual characters but rather transitions amongst characters
- The idea behind lookahead (**?=**) and lookbehind (**?<=**) is to generalize the notion of position
  - Lookaround operators do not consume text of the string
  - However, the regex machinery still goes through the motions
  - The regex "Chris" matches the string "Christopher Jones" as shown by the underline
  - The regex "?=Chris" matches the position just before the "C" in "Christopher Jones" but after any character preceding the string (i.e., in-between characters)

Lookaround

- Let's apply this to the statement about the bicycle
- We can read the pattern as follows:
  - The regex "matches" the provided string (i.e., "s") if "Mike" is in the string...
  - ... and if the start of "Mike" is at a word boundary
  - and if "s" follows "Mike"
  - but the **actual match** used for substitution starts at the word boundary **and goes up to but does not include the letter "s"**.

```
s = "This is Mikes bicycle"
print "Before -->", s
s = re.sub(r"\bMike(?=s\b)", "Mike'", s)
print "After -->", s
print
```

Lookaround

- We can be more precise (and require less of a replacement string) by using **both** lookahead and lookbehind
- We can read the pattern as follows:
  - Find a spot where we can look behind to "Mike"...
  - ... and look ahead to "s"
  - and at that position (i.e., width 0!) "substitute" with a single quote.

```
s = "This is Mikes bicycle"
print "Before -->", s
s = re.sub(r"(?<=\bMike)(?=s\b)", "'", s)
print "After -->", s
print
```

Surprise, surprise

- Since we're looking at positions, and since we don't consume characters...
- ... we can exchange the order of lookahead and lookbehind yet get the same result!
- To repeat: we're matching a position (i.e., a zero width char).
  - The mind boggles, but this does work.

```
s = "This is Mikes bicycle"
print "Before -->", s
s = re.sub(r"(?<=\bMike)(?=s\b)", "'", s)
print "After -->", s
print
```

```
s = "This is Mikes bicycle"
print "Before -->", s
s = re.sub(r"(?=s\b)(?<=\bMike)", "'", s)
print "After -->", s
print
```

Problem Solving

- Back to the problem...
- **Problem 8: Add commas to a large number to improve readability**
  - Example: cdn_population = 33894000
  - Yet we want this to appear in output with commas ("33,894,000")
- We want to insert commas at specific positions
  - These correspond to locations having digits on the right in exact sets of three.
  - This we can do with a lookahead
  - For the case of "at least some digits on the left", we can use lookbehind
  - We can represent three digits as either "\d\d\d\" or "\d{3}"
  - What we'll use as the replacement string is simply ","

Adding commas

```
#!/usr/bin/python

import re

n = "1234567890"
print "Before -->", n
n = re.sub(r"(?<=\d)(?=(\d{3})+$)", ",", n)
print "After -->", n
```

```
$ ./prob08.py
Before --> 33894000
After --> 33,894,000
```

Don't forget that the "substitute" commands does a global search and replace (i.e., all places where this pattern matches will have the command inserted).

Summary

- Regular expressions enable us to perform many sophisticated searches
  - Can specify repeated sets of characters
  - Can specify positions of matches
- Not only can searches be performed, but results of those searches can be retrieved
  - Using match objects; using compiled patterns
  - Can even use the result of matches within a later part of the match!
- String substitutions are also possible with regexes
  - Many problems normally requiring lots of "splits" and breaking of strings into substrings can be performed with the aid of regular expressions.
- Python's support for regexes in the **re** module is very good...
  - ... although you must remember to check how another language deals with certain corner cases (i.e., using forward slashes in patterns; the way escaped chars are handled with the language's strings; how you access matches; etc.)
  - always remember to quote patterns correctly (use r"<pattern>" when in doubt)

Colophon

- Some examples taken from "Programming Python, 3rd Edition" 2006 © Mark Lutz, O'Reilly
- Others taken from "Mastering Regular Expressions, 3rd edition", 2006 © Jeffrey E.F. Friedl, O'Reilly
- Everything else: © 2010 Michael Zastre, University of Victoria