Regular Expressions

- Stating a regular expression (more complex)
- Python **`re`** module (more complex)
- Using regexes for control flow

Problem Solving

- A large set of programming problems with strings can be solved with **substitutions**
  - The pattern describes what we want to replace
  - Another string describes how we want it changed.
- There are a variety of substitution routines in the Python re module
  - We are interested in the one named `re.sub()`
  - It takes at least three parameters: **search** pattern, **replacement** pattern, and **target string**
- **Problem 5: Cleaning up stock prices**
  - Numbers arrive as strings from some stock-price service
  - Sometimes they have lots of trailing zeros
  - We want to take the first two digits after the decimal point, and take the third digit only if is not zero; all other digits are removed
  - Example: "3.14150002" --> "3.141"
  - Example: "51.5000" --> "51.50"

First, a warm up

```
#!/usr/bin/env python3

import re

line1 = "Michael Zastre"
line2 = "Michael Marcus Joseph Zastre"
```

```
print ("Before:", line1)
line1 = re.sub("Michael", "Mike", line1)
print ("After:", line1)

print ()

print ("Before:", line2)
line2 = re.sub("Marcus", "M.", line2)
line2 = re.sub("Joseph", "J.", line2)
print ("After:", line2)
```

Substitutions are global (i.e., all instances for a particular string match get substituted).

```
$ ./warmup.py
Before: Michael Zastre
After: Mike Zastre

Before: Michael Marcus Joseph Zastre
After: Michael M. J. Zastre
```

Problem Solving

- **Problem 5: Cleaning up stock prices**
  - Numbers arrive strings from some stock-price service
  - Sometimes they have lots of trailing zeros
  - We want to take the first two digits after the decimal point, and take the third digit only if is not zero; all other digits are removed
  - Example: "3.14150002" --> "3.141"
  - Example: "51.5000" --> "51.50"
- Let's think this through:
  - We are not interested in changing digits to the left of the decimal point.
  - We want at least two digits to the right of the decimal point.
  - If the third digit to the right of the decimal point is not a zero, then we want to keep it…
  - … otherwise we don't want it.
- We'll throw into the mix one other feature
  - Match references (i.e., `\<num>`)

Substitutions

```
#!/usr/bin/env python3

import re

prices =[ "3.141500002", "12.125", "51.500"]

for p in prices:
    print ("Before --> ", p)
    p = re.sub(r"(\.\d\d[1-9]?)\d*", r"\1", p)
    print ("After --> ", p)
    print ()
```

In the second parameter to `re.sub()`, all backslash escapes are processed (i.e. Python string rules), so we need to use `r" "` to denote the string with the backreference.

```
$ ./prob05.py
Before -->  3.141500002
After -->  3.141

Before -->  12.125
After -->  12.125

Before -->  51.500
After -->  51.50
```

Problem Solving

- Python supports **shortstrings** and **longstrings**
  - All of our strings so far have been of the short form
  - Docstrings are longstrings (strings delimited with """)
  - We can use longstrings to format a textual document
- **Problem 6: Spam Form Letters**
  - (Please don't do this at home.)
  - We have a text block that we want to customize
  - There are certain spots in the text block where we have "tags" that must be replaced with specific strings
  - We would like to do this with regular expressions

Example: Form letter

```
=LOCATION=

Attention: =TITLE=

Having consulted with my colleagues and based on the
information gathered from the Nigerian Chambers of Commerce
and industry, I have the privilege to request for your
assistance to transfer the sum of =AMOUNT=
(=AMOUNTSPELLED=) into your accounts.

We are now ready to transfer =AMOUNT= and that is where
you, =SUCKER=, come in.
```

```
place = ‘City, Country'
title = 'The President/CEO'
cash  = '$47,500,000.00'
cashtext = 'forty-seven million, five hundred thousand dollars'
important_person = 'Mr. Elon Musk'
```

Form letter

- To fill out the form letter, we could have the following subtitutions:
  - contents of "`place`" replace all spots with "`=LOCATION=`"
  - contents of "`title`" replace all spots with "`=TITLE=`"
  - contents of "`cash`" replace all spots with "`=AMOUNT=`"
  - contents of "`cashtext`" replace all spots with "`=AMOUNTSPELLED=`"
  - contents of "`important_person`" replace all spots with "`=SUCKER=`"
- This can be implemented via a straight-forward sequence of `re.sub()` operations
  - By default, the operation performs a global replacement on the target string
  - (However, we can `use re.subn()` if we want to limit this.)

Form letter

```
#!/usr/bin/env python3

import re

blank_letter = """
    =LOCATION=

    Attention: =TITLE=

    Having consulted with my colleagues and based on the
    information gathered from the Chambers of Commerce
    and industry, I have the privilege to request for your
    assistance to transfer the sum of =AMOUNT=
    (=AMOUNTSPELLED=) into your accounts.

    We are now ready to transfer =AMOUNT= and that is where
    you, =SUCKER=, come in."""

# continued on next slide
```

Form letter

```
# continued from previous slide

place = ‘City, Country'
title = 'The President/CEO'
cash  = '$47,500,000.00'
cashtext = 'forty-seven million, five hundred thousand dollars'
important_person = 'Mr. Elon Musk'

letter = re.sub(r"=LOCATION=", place, blank_letter)
letter = re.sub(r"=TITLE=", title, letter)
letter = re.sub(r"=AMOUNT=", cash, letter)
letter = re.sub(r"=AMOUNTSPELLED=", cashtext, letter)
letter = re.sub(r"=SUCKER=", important_person, letter)

print (letter)
```

Example: Form letter

```
City, Country

   Attention: The President/CEO

   Having consulted with my colleagues and based on the
   information gathered from the Chambers of Commerce
   and industry, I have the privilege to request for your
   assistance to transfer the sum of $47,500,000.00
   (forty-seven million, five hundred thousand dollars) into your accounts.

   We are now ready to transfer $47,500,000.00 and that is where
   you, Mr. Elon Musk, come in.
```

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 Apr 31 9:25 2022
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, Apr 31 2022 9:25
Message-Id: <2013022939939.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, Apr 31 2022 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:
  1. **header lines**
  2. **a single blank line**
  3. **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

Example: overall code structure

```
#!/usr/bin/env python3

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 Apr 31 9:25 2022
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, Apr 31 2020 9:25
Message-Id: <202004139939.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 Apr 31 9:25 2022
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, Apr 31 2022 9:25
Message-Id: <2020043139939.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 Apr 31 9:25 2022
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, Apr 31 2022 9:25
Message-Id: <2020043139939.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 some regex features 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 is 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/env python3

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`" and **just 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 lookbehind and lookahead
- 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 of zero!) "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 ()
```

"Positive lookbehind assertion"

- In essence we are making statements regarding what must be true before matched text
- Example: Look for a word following a hyphen

```
n = "What a hare-brained idea!"
matchobj = re.search(r"(?<=(-))\w+\b", n)
if matchobj:
    print (matchobj.group(0))
else:
    print ("No match")
```

```
$ ./prob10.py
brained
```

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/env python3

import re

n = "33894000"
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).

Greedy vs. non-greedy

"?" can be used to modify "?", "+" and "*" to be non-greedy (i.e., consume as little as possible of string to perform match)

```
#!/usr/bin/env python3

import re

n = "<p>This is an HTML paragraph</p>"
print (n)

matchobj = re.search(r"<.*>", n)
print ("Match produces --> ", matchobj.group(0))

matchobj = re.search(r"<.*?>", n)   # non-greedy modifier to *
print ("Match produces --> ", matchobj.group(0))
```

```
$ ./prob09.py
<p>This is an HTML paragraph</p>
('Match produces --> ', '<p>This is an HTML paragraph</p>')
('Match produces --> ', '<p>')
```

Regexes in C

- There is a regular-expression library for the C programming language...
- ... and it supports POSIX regular expressions
  - These are substantially similar to what we have seen so far.
  - The big change, however, is in the way metasymbols are specified.
  - Example: "\d" becomes "[[:digit:]]", "\w" becomes "[[:alnum:]]", etc.
- They are substantially harder to use at first, yet do not do anything surprising.

Regexes in C

- Must include: `<regex.h>`
  - As regular expressions involve strings, then should also include `<string.h>`
- Ingredients (i.e., to use in a program):
  - `regex_t` variable: the regular expression itself
  - `regmatch_t` variable: to indicate where match patterns begin and end in the searched string
  - Code that calls `regcomp`: regexes must be compiled in C
  - Code that calls `regfree`: releases memory resources associated with compiled regular expression
  - Code that extracts the matches: working with strings

C regex: example

```
int status;
regex_t re;
regmatch_t match[4];

char *pattern = "([[:digit:]]+)";
char *search_string = "abc def 123 hij";

if (regcomp(&re, pattern, REG_EXTENDED) != 0) {
    return 0;
}

status = regexec(&re, search_string, 2, match, NULL);
if (status != 0) {
    fprintf(stderr, "No match.\n");
    return 0;
}

char match_text[100];
strncpy(match_text, search_string+match[1].rm_so,
    match[1].rm_eo - match[1].rm_so);  /* rm_eo is already plus one */
match_text[match[1].rm_eo - match[1].rm_so + 1] = '\0';

printf("Match was '%s'\n", match_text);
regfree(&re);
```

regcomp regexec

- `regcomp` takes **three parameters**:
  1. Address to a `regex_t` variable
  2. Actual pattern to search for (in POSIX form)
  3. Flags
- `regexec` takes five parameters:
  1. Address to a `regex_t` variable (which has been already initialized by `recomp`)
  2. The string to be searched
  3. The maximum number of groupings in the pattern...
  4. ... and the match array itself which must have a length at least as long as what parameter 3 indicates
  5. flags (i.e., "no flags" == `NULL`)

The `match` variable

- Declared as an array
  - Size is normally one larger than the number of left parentheses
  - Be careful the 0th element is the string that was involved in the match!
- Each element denotes the start and ending position of the match
  - **`match[i].rm_so`**: Index position in the original string at which the ith match starts
  - **`match[i].rm_eo`**: Index position **plus one** in the searched string at which the ith match ends
- Usual practice: Copy the characters in the match from the search string to some temporary string...
  - ... and then use that temporary string