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.

C code to check for DD/MM/YYYY format

```
int is_date_format(char *check) {

    if (!isdigit(check[0]) || !isdigit(check[1])) {
        return 0;
    }

    if (!isdigit(check[3]) || !isdigit(check[4])) {
        return 0;
    }

    for (i = 6; i < 10; i++) {
        if (!isdigit(check[i])) {
            return 0;
        }
    }

    if (check[2] != '/' || check[5] != '/') {
        return 0;
    }

    /* Still haven't even figured out of the DD makes sense, let alone
     * the MM!!!!
     */

    return 1;
}
```

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.

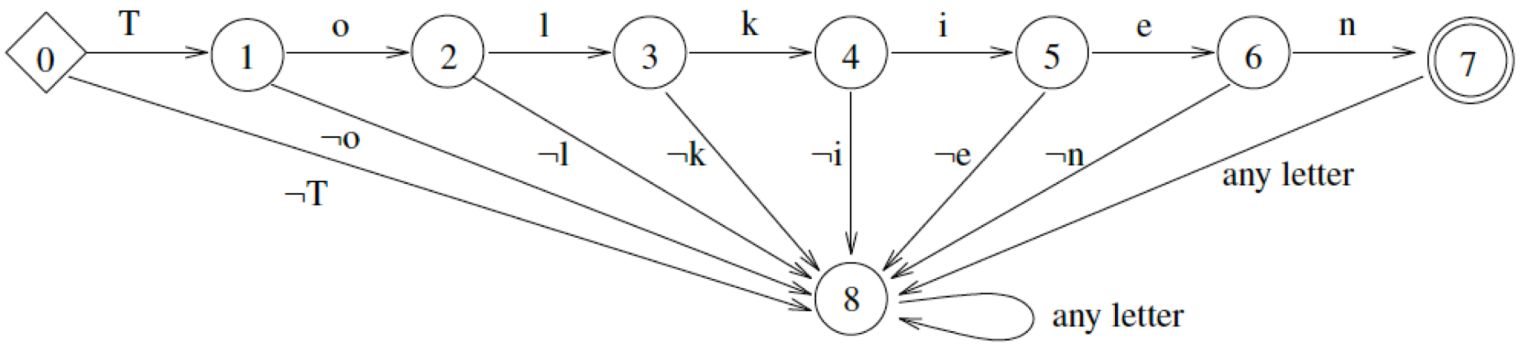
Speed concerns

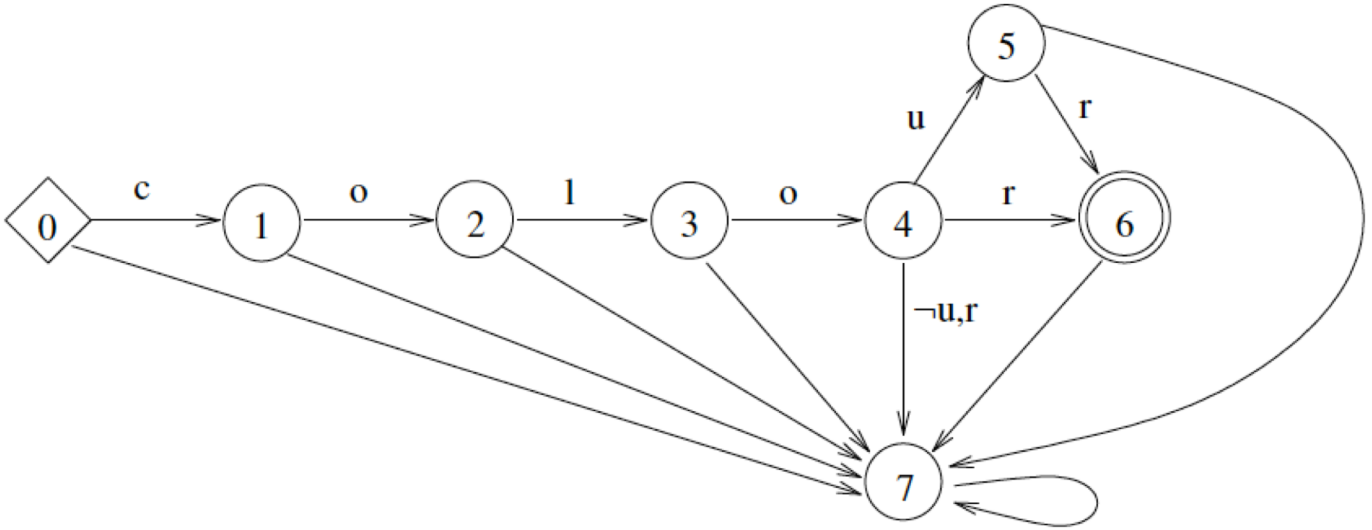
- So far we have specified the regular expression for every use of an **re** operation
- For occasional regex matching this is fine
- However, each time the match is performed the Python interpreter must re-interpret the regex
  - This means the regex must be re-parsed and the state machine re-constructed.
  - 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**

Speed concerns

- 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 invoking re operations
  - Onedirectly specifying the regex in call to match(), search(), etc.
  - The other using a pattern object returned from re.compile() for which we call match(), search(), etc.).

Regex as a state machine





Compiled pattern

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

>>> hello_list = [text1, text2]
>>> for t in hello_list:
...     matchobj = pattobj.match(t)
...     if matchobj:
...             print (t, " --> match --> ", matchobj.group(1))
...     else:
...             print (t, " --> no matches")
...
Hello spam...World  --> match -->   spam...
Hello spam...other  --> no matches
```

Lots in the **re** module

- Python's **re** module has methods for:
  - matching (i.e., finding a match that must start at the beginning of the string)
  - searching (i.e., finding a match that may occur anywhere in the string)
  - substituting
  - precompiling
  - splitting
  - iterating through matches
- Match objects also have several methods
  - We've already seen group()
  - There are also groups(), groupdict()
- Let us look at a few examples, this time with a few more metasymbols included

More complex pattern

```
>>> datetime1 = "20180211T110000"
>>> datetime2 = "20171225T000000Z"
>>> datetime3 = "11/06/2016"
>>> datetime4 = "21/4/14"

>>> matchobj = re.match("(\d{4})(\d{2})(\d{2})T.*", datetime1)
>>> if matchobj:
...     (year, month, day) = matchobj.groups()
...     print (year, month, day)
... else:
...     print ("Error")
...
2018 02 11
```

More complex pattern

```
>>> dates = ["20180230T110000", "20170615T000000Z", "11/11/2017",
             "21/4/18""]

>>> pattobj = re.compile( "(\d\d?)/(\d\d?)/(\d\d(\d\d)?)" )

>>> for d in dates:
...     matchobj = pattobj.match(d)
...     if matchobj:
...             (day, month, year, _) = matchobj.groups()
...             print ("%4d%02d%02d" % (int(year), int(month), int(day)))
...     else:
...             print (d, "doesn't match")
...
20180230T110000 doesn't match
20170615T000000Z doesn't match
20171111
  180421
```

Another pattern

```
>>> line1 = ".LM +5"
>>> line2 = ".LM filled"
>>> line3 = ".LM 10x"
>>> line4 = ".LM 22"
>>> lines = [line1, line2, line3, line4]

>>> for line in lines:
...     matchobj = re.match("\.LM (\d+)\s*$", line)
...     if matchobj:
...             values = matchobj.groups()
...             print (line, ": matches with value ", values[0])
...     else:
...             print (line, ": DOESN'T match")

.LM +5 : DOESN'T match
.LM filled : DOESN'T match
.LM 10x : DOESN'T match
.LM 22 : matches with value  22
```

Notes from previous example

- Although grouping may be used to control the regular-expression match, not all results need to be extracted
  - Notice that sometimes part of the extracted matches is ignored
  - Always be aware the extracted matches are indexed by opening left parenthesis (i.e., not by your intent as a programmer to extract out particular parts of the match)
- There is often more than one way to phrase the same regular expression
  - Note that "\d\d" is the same as "\d{2}"
  - Which one is better? Depends perhaps on style of programmer, amount of change expected with code, etc. etc.

Variety

- Sometimes our needs vary when working with regexes
  - Sets of strings may be best expressed by alternative strings
  - Regexes may need to be carefully crafted sets of characters
  - Matches may sometimes be required on word boundaries
  - Sometimes all we want is the starting location of the match.
- Python string rules can sometimes interfere with regular expressions
  - The problem is with backslashes
  - Sometimes you must double-up on them (e.g., "\\")

Using search()

```
>>> pattern, string = "A.C.", "xxABCDxx"
>>> matchobj = re.search(pattern, string)
>>> if matchobj:
...     print (matchobj.start())
...
2
>>> pattobj = re.compile("A.*C.*")
>>> matchobj = pattobj.search("xxABCDxx")
>>> if matchobj:
...     print (matchobj.start())
...
2

>>> print (re.search(" *A.C[DE][D-F][^G-ZE]G\t+ ?", "..ABCDEFG\t..").start())
2
>>> print (re.search("A|XB|YC|ZD", "..AYCD..").start())
2
>>> print (re.search("\bABCD", "..ABCD").start())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'start'
>>> print (re.search(r"\bABCD", "..ABCD").start())
2
>>> print (re.search(r"ABCD\b", "..ABCD").start())
2
```

Problem Solving

- We have seen a variety of **metasymbols** (sometimes referred to as **metacharacters**)
  - Most of them match one or more characters
  - Some, however, are meant to catch a particular position (i.e., they catch zero characters!)
- The simplest positional symbols are ^ and $
  - ^: match beginning of string
  - $: match end of string
  - Note that re.**match**("<pattern>", string) is **exactly** the same as re.**search**("^<pattern>", **string**) if **string** is not multiline
- Another positional symbol is **\b**
  - Matches a word boundary (i.e., zero characters)
  - That is, it matches the position in between characters (one of which is a word character, the other a non-word character)
  - Word characters: [a-zA-Z0-9_]
- **Problem 1: Match the word "Chris" in a string, but not "Christmas", "Christine", etc.**

Problem Solving

```
#!/usr/bin/python3

import re

lines = ['''I said to Chris, "Hey, watch out!"''',
    '''I'll be home for Christmas!''',
    'Christine Faulkner',
    'Chris Flynn',
    'Evert, Chris']

for li in lines:
    if (re.search(r'\bChris\b', li)):
        print (li)
```

```
$ ./prob01.py
I said to Chris, "Hey, watch out!"
Chris Flynn
Evert, Chris
```

Problem Solving

- Words need not be textual
  - They can also be numerical
  - Key point is that non-word characters are neither numbers nor letters (nor the underscore)
- Sometimes we are interested in the shape of number sequences
  - Course numbers
  - Room numbers
  - Serial numbers, product codes, etc.
- **Problem 2: Extract the last four digits from a North American phone number**
  - May be of the form "250-472-5000"...
  - or "250 472 5000"...
  - or "472-5000"
  - or perhaps "250.472.5000" or "+1 250 472 5000"

Problem Solving

```
#!/usr/bin/python3

import re

lines = ["250-472-5000", "472-5000", "250.472.5000", \
    "+1 250 472 5000", "011 49 9602 4241", "2504725000", \
    "mom's number", "12345 678 90"]

for l in lines:
    matchobj = re.search(r"(\b\d{3}\b[- \.])?\b\d{3}\b[- \.](\b\d{4}\b)", l)
    if matchobj:
        print (l, "-->", matchobj.group(2))

    matchobj = re.search(r"\b(?:\d{3})?\d{3}(\d{4})\b", l)
    if matchobj:
        print (l, "-->", matchobj.group(1))
```

```
$ ./prob02.py
250-472-5000 --> 5000
472-5000 --> 5000
250.472.5000 --> 5000
+1 250 472 5000 --> 5000
2504725000 --> 5000
```

Notice the "?:" used in the second search. It makes a set of parentheses "non-matching" but still useful to structure the regex. That is why the group number is still 1 even though the match we want is denoted by the second left-parenthesis

Problem Solving

- We can also use regexes to verify that the format provided as input matches what we expect
  - Example: Input string is in "DD/MM/YYYY" or "MM/DD/YYYY" format
  - Example: String provided is a URI (i.e., proper sets of characters)
- **Problem 3: Obtain a temperature (assumed to be Celsius) and return the number in Fahrenheit**
  - Number is to be an integer
  - There must be only one number in the string
  - No other characters (such as "C") should be at the end
  - Fahrenheit = (Celsius * 9 / 5) + 32

Input validation

```
#!/usr/bin/python3

import re

celsius = input("Enter a temperature in Celsius: ")
celsius = celsius.rstrip("\n")

matchobj = re.search(r"^[0-9]+$", celsius)  # same as re.match("\d+",...)
if matchobj:
    celsius = int(celsius)
    fahrenheit = (celsius * 9 / 5) + 32
    print ("%d C is %d F" % (celsius, fahrenheit))
else:
    print ("Expecting a number, so I don't understand", celsius)
```

```
$ ./prob03.py
Enter a temperature in Celsius: 30
30 C is 86 F
```

Problem Solving

- However, we should do a bit more
  - The problem statement is perhaps a bit too restrictive.
  - Negative temperatures cannot be given as values.
  - Decimal temperatures also cannot be provided
- The regular expression should accept these
  - And the other code changed to suit (i.e., use "float()" instead of "int()")

Input validation

```
#!/usr/bin/python3

import re

celsius = input("Enter a temperature: ")
celsius.rstrip("\n")


matchobj = re.search(r"^([-+]?[0-9]+(\.[0-9]*)?)$", celsius)
if matchobj:
    celsius, _ = matchobj.groups()
    celsius    = float(celsius)

    fahrenheit = (celsius * 9 / 5) + 32
    print ("%.2f C is %.2f F" % (celsius, fahrenheit))
else:
    print ("Expecting a number, so I don't understand", celsius)
```

```
 ./prob03.py
Enter a temperature in Celsius: 12.2
12.20 C is 53.96 F
```

Problem Solving

- Our little script could be even more general
  - Rather than just convert from celsius to fahrenheit, it could convert the other direction
  - The starting value can be indicated by a "C" or "F" (or "c" or "f")
- **Problem 4: Obtain a temperature. If it is in celsius, return the number in fahrenheit; if in fahrenheit, return the number in celsius.**
  - Number can be an integer or a float, positive or negative
  - There must be only one number in the string
  - Character "C" or "F" implies what we are converting from and to.

Input validation plus more

```
#!/usr/bin/python3

import re

input = input("Enter a temperature: ")
input.rstrip("\n")

matchobj = re.search(r"^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$", input, re.IGNORECASE)
if matchobj:
    input_num, _, type = matchobj.groups()
    input_num          = float(input_num)

    if type == "C" or type == "c":
        celsius    = input_num
        fahrenheit = (celsius * 9 / 5) + 32
    else:
        fahrenheit = input_num
        celsius = (fahrenheit - 32) * 5 / 9

    print ("%.2f C is %.2f F\n" % (celsius, fahrenheit))
else:
    print ('Expecting a number followed by "C" or "F",')
    print ('so I cannot interpret the meaning of', input)
```

Notice how we indicate that case is to be ignored. The "re" module contains are large number of these kinds of options.

Problem Solving

- Our solution to Problem 4 still has some flaws
  - Cannot enter a number less than one without a leading zero.
  - No leading spaces are permitted (i.e., we have general whitespace issues)
  - We are using [0-9] instead of \d
  - etc. etc.
- There are many ways to "skin" a regular expression
  - The lesson so far, however, is that coming up with a full regular expression for these kinds of matches can be an iterative process.
  - Must also be aware of how a language deals with metasymbols within strings (e.g., Perl and Ruby are a bit different than Python)

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/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/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: Nigerian ~~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 = 'Lagos, Nigeria'
title = 'The President/CEO'
cash  = '$47,500,000.00'
cashtext = 'forty-seven million, five hundred thousand dollars'
important_person = 'Mr. Justin Trudeau'
```

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/python3

import re

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."""

# continued on next slide
```

Form letter

```
# continued from previous slide

place = 'Lagos, Nigeria'
title = 'The President/CEO'
cash  = '$47,500,000.00'
cashtext = 'forty-seven million, five hundred thousand dollars'
important_person = 'Mr. Justin Trudeau'

letter = re.sub(r"=LOCATION=", place, 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

```
Lagos, Nigeria

   Attention: The President/CEO

   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 $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. Justin Trudeau, 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 2017
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 2017 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 2017 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/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 2017
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 2017 9:25
Message-Id: <201506139939.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 2017
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 2017 9:25
Message-Id: <2015063139939.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 2017
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 2016 9:25
Message-Id: <2015063139939.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/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** 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 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/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/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, 0);
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

- **recomp** 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

Complete aside: SQL injection attacks

- Problem:
  - Untrusted text from an application (i.e., web page) may be inserted without modification into a query
  - If carefully crafted, untrusted text could wreak havoc with security of our site.
- Although we may want to take extra steps...
  - ... using the "execute()" parameter passing mechanism will prevent arbitrary SQL from appearing in SQL statements
- Typical problem example: e-mail address

Problematic code...

- Note query is innocent on the face of it
  - Simple **select**
  - Fetches data from a table
  - **where** clause specifies row(s) when the contact attribute equals the value stored in email_address
  - email_address is a reference to a Python string
  - For our examples, we don't necessarily know how the string's value was set.

```
email_address = "i.love.to.be.me@donaldtrump.gov"

# Assuming we're using something like psychopg2 driver in order to connect
# a Python script to some PostgreSQL server instance. Variable cursor was
# assigned a value earlier in the script...

cursor.execute("""
   select somefield1, somefield2, somefield3
   from really_important_table
   where contact = '%s'""" % email_address)
```

Attack (version 1)

- Changing where clause to a trivially true
- Will cause all rows in the table to be returned

```
email_address = "anything' or 'x'='x"

cursor.execute("""
   select somefield1, somefield2, somefield3
   from really_important_table
   where contact = '%s'""" % email_address)
```

```
   select somefield1, somefield2, somefield3
   from really_important_table
   where contact = 'anything' or 'x'='x'
```

Attack (version 2)

- Trying to discover the names of attributes in tables
- In essence, guessing in a way that the query returns information that tells if the guess was right
- (Incorrect guess == syntax error)

```
email_address = "x' and email is NULL --"

cursor.execute("""
   select somefield1, somefield2, somefield3
   from really_important_table
   where contact = '%s'""" % email_address)
```

```
   select somefield1, somefield2, somefield3
   from really_important_table
   where contact = 'x' and email is NULL --'
```

Attack (version 3)

- Damage the database
  - Assumes the attacker knows the names of tables in the database
  - Drop table, view, index

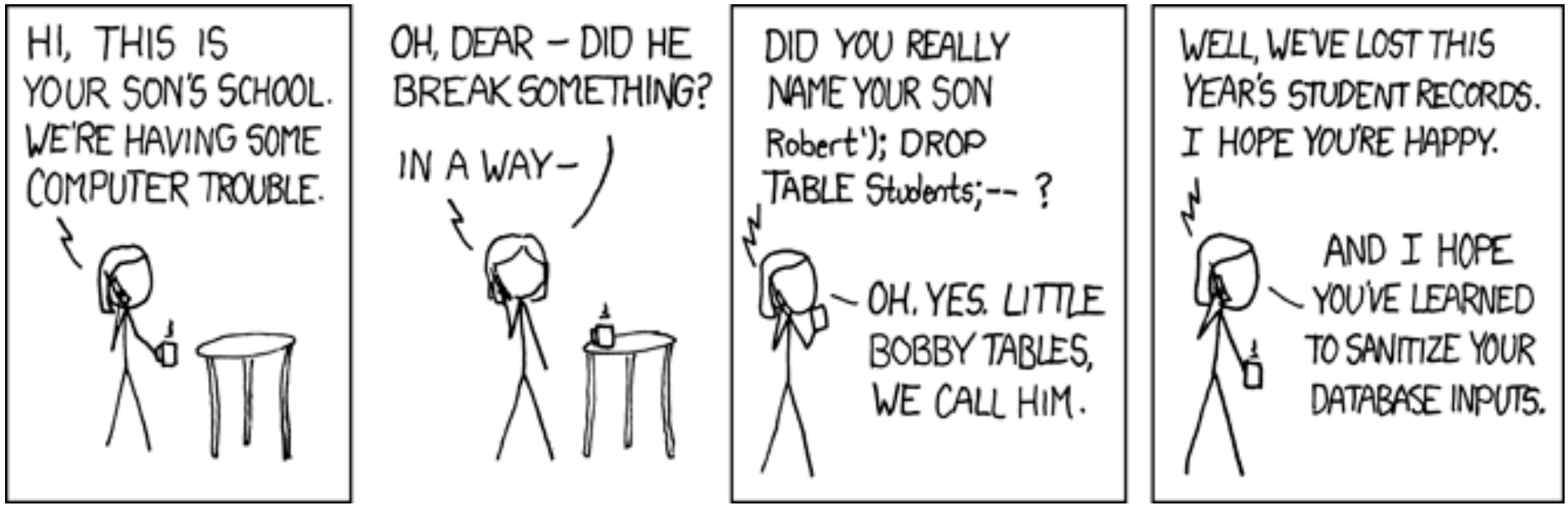
```
email_address = "x'; drop table members --"

cursor.execute("""
   select somefield1, somefield2, somefield3
   from really_important_table
   where contact = '%s'""" % email_address)
```

```
   select somefield1, somefield2, somefield3
   from really_important_table
   where contact = 'x'; drop table members --'
```

SQL injection attacks

- Solution:
  - Use regular expressions to check for input
  - If input doesn't match suitable pattern, then reject it (i.e., error)
- For example:
  - queries should not contains quotation marks
  - nor should they contain references to NULL
  - etc. etc.

http://xkcd.com/327/



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: © 2019 Michael Zastre, University of Victoria