Computer Science Principles

CHAPTER 10 – NATURAL LANGUAGE PROCESSING AND REGULAR EXPRESSIONS

Announcements

Read Chapter 10 in the Conery textbook (Explorations in Computing)

Acknowledgement: These slides are revised versions of slides prepared by Prof. Arthur Lee, Tony Mione, and Pravin Pawar for earlier CSE 101 classes. Some slides are based on Prof. Kevin McDonald at SBU CSE 101 lecture notes and the textbook by John Conery.

The Turing Test

Alan Turing (1912-1954) was an English mathematician who laid some of the important theoretical groundwork of computer science

In addition to other topics, Turing was interested in the idea of computers being able to think as human beings do

He devised what he called the **imitation game**, now known as the **Turing test**

- A human judge (the interrogator) engages in an online chat with another person and a computer, but isn't told which is a human and which is a computer

The Turing Test

The interrogator asks the human being and the computer a series of questions to carry on a conversation

If the interrogator cannot tell which is the person and which is the computer, then the computer has passed the Turing Test because it is simulating human intelligence

So the Turing Test touches on two important areas of computer science:

- **Artificial intelligence** (AI)
- **Natural language processing** (NLP)

ELIZA: One of the First Chatbots

ELIZA is a program first developed in the 1960s at MIT that attempts to converse in English with a human participant

- More specifically, the program “pretends” to be a psychotherapist, so its responses often consist of questions that use words the user entered

Based on an internal set of rules that analyze what the user types, the software attempts to simulate a natural conversation

For example, one kind of rule will look for key words in the human's entry (H) and respond with generic sentences (C). Here's an example:

H: My father wouldn't buy me a puppy.

C: Tell me more about your family.

ELIZA: One of the First Chatbots

Since ELIZA is simulating a psychotherapist, its rules pay special attention to words like "dream" and various words that connote emotions:

H: I'm sad I never got a puppy.

C: Can you explain what made you sad you never got a puppy?

These rules have limits. Consider the following exchange, which the software could generate:

H: I'm sad we never had a chance to play fetch.

C: I am sorry to hear you are sad we never had a chance to play fetch.

ELIZA: One of the First Chatbots

The software's intelligence is limited by its rules, or **script**

- The program should become better at conversing as more rules are added to its script

The simplest rule would be that the program repeats or *echoes* the words entered by the user:

```
from PythonLabs.ElizaLab import *
Eliza.run()
```

Sample run:

```
H: hello
C: hello
H: are you imitating me?
C: are you imitating me
bye
```

ELIZA: One of the First Chatbots

To make things more interesting, let's load the DOCTOR script, which simulates a psychotherapist.

```
Eliza.load(path_to_data("doctor.txt"))
Eliza.run()
```

We could create our own scripts to simulate other kinds of conversations.

How do you do. Please tell me your problem.

H: I am afraid of spiders.

C: Is it because you are afraid of spiders that you came to me?

H: Yes, it is.

C: You seem quite positive.

H: I am very happy.

C: How have I helped you to be happy?

H: Just by chatting with me.

C: I am not sure I understand you fully.

H: You are not very smart.

C: What makes you think I am not very smart?

H: bye

Sentence Patterns

To find key words in a sentence we need to search through strings

The **find** method for strings will get the job done

- Given a string to search through and a target substring, the **find** method will return the index of the first character of the substring
- If the substring is not found, the method returns **–1**
- If the substring appears in several places, the **find** method returns the index of the first instance

Example:

```
s1 = 'I was afraid of the cow.'
s1.find('cow')      # returns 20
```

Sentence Patterns

Now we will see that the method will return **-1** if we search for a substring that is not in the string:

```
s1 = 'I am afraid of the dark.'
s1.find('cow')      # returns -1
```

Now let's see what happens if we search for a substring that appears multiple times in a string:

```
s1 = 'I saw birds on the house, birds at school, birds everywhere!'
s1.find('birds')    # returns 6
```

See eliza_functions.py for these examples and additional examples from the following slides

Sentence Patterns

The **find** method takes an optional argument that indicates at what index the search should begin

For example, the first instance of “birds” in the string below is at index 6.

- If we start the search at index 7, the **find** method will find the next instance of “birds” in the string, which starts at index 26:

```
s1 = 'I saw birds on the house, birds at school, birds everywhere!'
s1.find('birds', 7)     # returns 26
```

Sentence Patterns

One issue with **find** is that does not have any notion of *words*

For example, the letters of “cow” appear in the word “scowl”, among others

- If we were trying to build a chatbot to have conversations about farm animals, it might incorrectly see the word “scowl” and think that the person had typed “cow”

So to extract key words from a user's entry, we actually need to look for *words*, not simply substrings

To help address this problem we will use **regular expressions**, also called **regexes**, for short

Regular Expressions

Regular expressions give us a formal ways of expressing **patterns** of characters we want to find in an input string

- The simplest kind of regular expression looks for a particular substring in an input string

Initially we will not look at the details of how to write regular expressions, but rather use some capabilities from the ElizaLab

- The **Pattern** class in ElizaLab lets us create regular expressions using a user-friendly notation
- The **Pattern** object below can be used to detect when a sentence contains the word “cow”:

```
from PythonLabs.ElizaLab import Pattern
p = Pattern('cow')
```

Regular Expressions

Given a **Pattern** object, we can then provide possible responses the chatbot might produce when it sees a sentence that matches the pattern:

```
p.add_response('Tell me more about your farm')
p.add_response('Go on')
```

We can also give a list of responses when we create the **Pattern** object, instead of doing it later:

```
p = Pattern('lamb', ['I love lambs', 'How cute'])
```

Regular Expressions

To see if a sentence matches the pattern, we call the **apply** method of the **Pattern** object

- If the match is successful, a response string is returned
- If not, the method returns **None**

An example:

p.apply('I milked the cow and fed the chickens.')

Return value:

'Tell me more about your farm'

Regular Expressions

The **apply** method is “smart” enough to look for words, not simply substrings

For the example below, “scowled” will not trigger a match for the word “cow”, so **None** is returned:

```
p.apply('I scowled at the horse yesterday.')
```

Regular Expressions

Regular expressions let us define patterns that will match several different words

- We separate the words by vertical bars to form a **group**
- This means we don't have to make a separate pattern for each word

Below is a pattern that could be applied to any sentence containing “cow”, “pig” or “horse”:

```
p = Pattern('cow|pig|horse', ['Really?', 'Go on'])
p.apply('The horse jumped the fence.')
```

For this input, **apply** returns **'Really?'**

Decomposing Sentences into Parts

Finding individual words in a sentence is useful, but it's not enough to understand the context of the word

We need some way of decomposing a sentence into parts

- First, we will need to save the part of the sentence that matches a group in a pattern
- Each group in a pattern has an associated variable name, with **$1** identifying the first group, **$2** for the second group, and so on
- We can then use these variable names in responses

This is easiest to understand by example (next slide)

Decomposing Sentences into Parts

```
p = Pattern('cow|pig|horse')
p.add_response('You had a $1?')
p.add_response('How many $1s were on the farm?')
```

Now, call the **apply** method to try the new pattern:

```
p.apply('The horse jumped the fence.')
```

Return value: **'You had a horse?'**

Decomposing Sentences into Parts

We can start to build up pretty complex patterns that contain multiple groups, such as:

```
p = Pattern('I (like|love|adore) my (cat|dog|ducks)')
p.add_response('Why do you $1 your $2?')
p.add_response('What about your $2 do you $1?')
```

Call the **apply** method to try the new pattern:

```
p.apply('I adore my cat.')
```

Return value: **'Why do you adore your cat?'**

Decomposing Sentences into Parts

This approach looks pretty good, but we would have to list every word we are interested in

What if we wanted to write a pattern to anticipate everything a person might be afraid of?

- We really couldn't list everything

We could write a pattern containing one or more wildcards that match any piece of text

- A wildcard is written using a period followed by an asterisk

Decomposing Sentences into Parts

Example of a wildcard expression:

```
p = Pattern('I am afraid of .*')
p.add_response('Why are you afraid of $1?')
p.apply('I am afraid of little green men')
```

Return value:

```
'Why are you afraid of little green men?'
```

This looks pretty good, but it can lead to some nonsensical results, as we'll see in the next example

Response Pre-/Post-processing

The patterns we just wrote use a basic copy-and-paste strategy to transfer a string fragment from the input to the response, but can lead to silly responses:

```
p = Pattern('I am (.*)', ['Are you really $1?'])
p.apply('I am happy to see you')
```

Return value:

```
'Are you really happy to see you?'
```

What we need to do after matching the input is to do some **postprocessing** to replace some words or phrases

- For example, the response above really should have been “Are you really happy to see *me*?”

Response Pre-/Post-processing

Let's create a dictionary that will provide substitutions for pronouns that should be made for a pattern:

pronouns = { 'I': 'you', 'your': 'my' }

Now let's try a few examples:

p.apply('I am sorry I dropped your computer', post=pronouns)

Return value:

'Are you really sorry you dropped my computer?'

Response Pre-/Post-processing

Another example:

p.apply("I'm happy I lost", post=pronouns)

Return value:

None (Why?)

The **apply** function can match **"I"**, but not **"I'm"**

- Let's define a dictionary that can recognize contractions

We will use it to perform **preprocessing** on the input string before we attempt to make a match:

contractions = { "I'm": "I am" }

p.apply("I'm happy I lost", pre=contractions, post=pronouns)

Return value:

'Are you really happy you lost?'

Response Pre-/Post-processing

When ElizaLab is used to load a script, the preprocessing and postprocessing dictionaries are available via **Eliza.pre** and **Eliza.post**, respectively:

Eliza.pre has the value:

{"can't": 'can not', "don't": 'do not', "i'm": 'I am', "won't": 'will not', "you're": 'you are'}

Eliza.post has the value:

{'am': 'are', 'are': 'am', 'i': 'you', 'me': 'you',
'my': 'your', 'myself': 'yourself', 'you': 'I',
'your': 'my', 'yourself': 'myself'}

Algorithm for Having a Conversation

The summary so far:

- A **Pattern** object is defined by a string that has a key word, groups of words, or wildcards
- We can specify any number of response strings, and the object's **apply** method will return one of these strings if a sentence matches its pattern
- Pattern-matching variables (**$1**, **$2**, etc.) along with preprocessing and postprocessing help the **apply** method extract parts of the input sentence and reuse them as part of the response

Now, we need to take this functionality and develop an algorithm for carrying on a conversation

Algorithm for Having a Conversation

One simple approach is to try patterns until we find one that matches an input sentence, but this means that patterns will always be applied in the same order

To solve this problem we can assign a priority to each word

- The creator of ELIZA anticipated that people would ask questions like “Are you a computer?” so he gave higher priority to words like “computer” and “machine” than others

Another problem is that the same word (like **"I"**) can appear in multiple patterns - which one should be applied?

- In ElizaLab, this problem is addressed by creating a **Rule** object, which is a collection of patterns that pertain to a particular word

Algorithm for Having a Conversation

The **rule_for** method tells us which rules (if any) pertain to a particular word:

```
Eliza.rule_for('remember')
```

Return value:

```
PythonLabs.ElizaLab.Rule [5]
  'I remember (.*)'
  'do you remember (.*)'
```

The **5** indicates that this rule has high priority (higher number = higher priority)

We can use a priority queue (from our study of Huffman coding) to keep track of the highest-priority rules

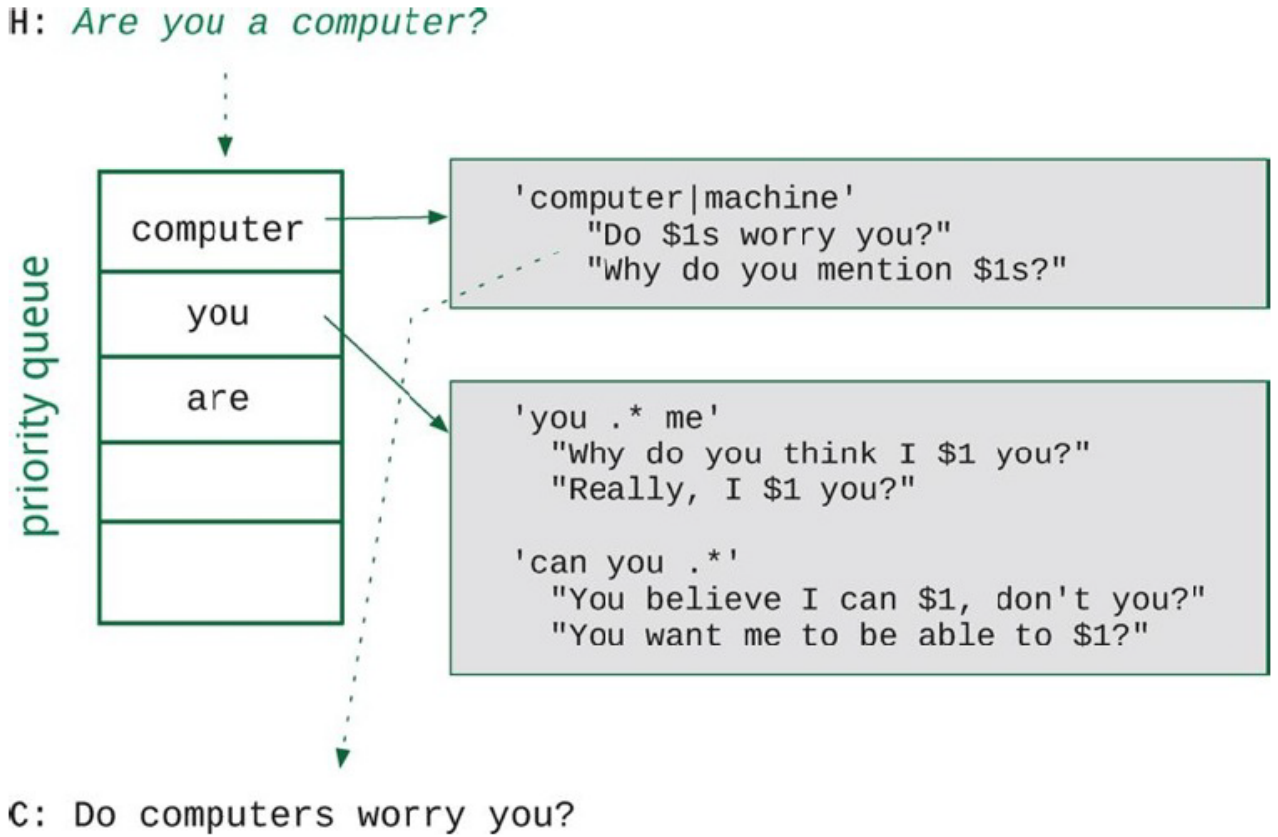
Algorithm for Having a Conversation

A complete algorithm for transforming an input sentence into a response has three steps:

1. Initialization
2. Tokenization (borrowed from the SpamLab) and insertion of rules into priority queue
3. Rule processing

Algorithm for Having a Conversation

The priority queue and rules relevant to this sentence are depicted in the figure below



Algorithm for Having a Conversation

```
from PythonLabs.Tools import tokenize
from PythonLabs.ElizaLab import *

def transform(sentence):
    queue = PriorityQueue()
    sentence = Eliza.preprocess(sentence, Eliza.pre)

    for word in tokenize(sentence):
        rule = Eliza.rule_for(word)
        if rule is not None:
            queue.insert(rule)

    while len(queue) > 0:
        rule = queue.pop()
        response = rule.apply(sentence, post=Eliza.post)
        if response is not None:
            return response
```

See eliza_transform.py

Algorithm for Having a Conversation

Let's call the **transform** function a few times:

transform('Are you a computer?')

Sample return values:

Do computers worry you?

Why do you mention computers?

What do you think computers have to do with your problem?

Don't you think computers can help people?

What about computers worries you?

ELIZA and the Turing Test

Can a chatbot like ELIZA pass the Turing Test?

One challenge in developing natural language processing systems is algorithmically assigning meanings to words in a sentence. This is the problem of **semantics**.

Consider the following exchange:

H: That cow was huge.

C: How big was it?

This kind of response is not possible with a rule-based algorithm like Eliza's because the software doesn't understand that “huge” and “big” have similar definitions.

- So we could give the program a dictionary and/or thesaurus

ELIZA and the Turing Test

Giving the program a dictionary of word definitions is not enough, however. Consider the following exchange:

H: The cow fell into the canal.

C: Was it able to get out?

Having the definitions of all the words in the human being's sentence will not empower a computer to draw conclusions, make inferences, and apply other kinds of "real-world knowledge" or "common sense"

So, although natural language processing systems have been deployed successfully in systems like airline reservation systems and other interactive telephone-based systems, there is still a long way to go before a computer could consistently pass the Turing Test

“Real” Regular Expressions

Part of string processing in real applications (e.g., online forms in websites) often involves ensuring that a string matches a particular format, called a **pattern**

We have been expressing patterns using the ElizaLab module, but these aren't “real” regular expressions (although they are pretty close)

Now, we will look at how regular expressions are actually defined and see briefly how we can write Python programs that use them

Simplest Regular Expressions

The simplest regular expressions are those that contain exactly the characters you are looking for

So if you want to see if the letters “CSE” appear in a string, the regular expression is: **CSE**

It would match the occurrences of “CSE” in these strings:

- **CSE**
- **CSE 101**
- **I am taking CSE 101**

There are useful websites that let you try out regular expressions to see which strings they will match:

- http://regex101.com
- https://regexr.com

We will use these sites throughout this lesson

Digits

The notation **\d** represents a single digit

A US Social Security number is in the format ###-##-####, where # is any digit

- For example: 347-54-2146
- So a generic US Social Security Number could be expressed with regexes as:

\d\d\d-\d\d-\d\d\d\d

But, this is a little cumbersome, so we can use curly braces instead to denote repetition. For example, **{3}** would mean to repeat something 3 times. Let's rewrite our regex:

\d{3}-\d{2}-\d{4}

Any US Social Security Number would ***match*** the above pattern

The Dot Metacharacter

A period by itself matches any character

So **.{5}** would match a 5-character sequence of any characters:

- **12345**
- **abcde**
- **a&c(9**
- **1 3 4** Yes, spaces are matched too!

If you want to match an actual period, you ***escape it*: \.**

\d{3}\.\d{3}\.\d{4} will match phone numbers written in a form like **010.777.1223**

Matching Specific Characters

Sometimes the period metacharacter is too “powerful” because it can match any character

We can match specific characters using regular expressions, by defining them inside [] (square brackets)

- For example, the pattern **[abc]** will only match a single **a**, **b**, or **c** letter and nothing else

One usage is locating words that are spelled differently in British and American English

- For example to match: analyze or analyse
- You could match this with: **analy[zs]e**

Excluding Specific Characters

In some cases, we might know that there are specific characters that we don't want to match

To exclude specific characters we use the [] (square brackets) with the **^** (hat) character

- For example, the pattern **[^abc]** will match any single character except for the letters **a**, **b**, or **c**

Character Ranges

We can match a character from a list of sequential characters by using the – (dash) to indicate a character range

For example, the pattern **[0-6]** will only match any single digit character from zero to six, and nothing else

[^n-p] will only match any single character except for letters **n** through **p**

Multiple character ranges can also be used in the same set of brackets

- **[A-Za-z0-9_]** is often used to match characters in English text

Alphanumeric Characters

The notation **\w** denotes any word character: **a-z**, **A-Z**, **0-9** and the underscore _

So the regex **\w{3}\d{4}** would match any string containing three alphanumeric characters followed by any four digits

- **abc1234**
- **ABC1234**
- **1AB7892**
- **1234567**

Related is the **\W** notation, which matches any non-alphanumeric character

Repetitions

We saw that the notation **{m}** means to match the character *m* times

We can also use the repetition notation with the square bracket notation

- For example, **[wxy]{5}** would match five characters, each of which can be a **w**, **x**, or **y**

The notation **{m,n}** means we want to match a pattern from **m** to **n** times, inclusive of **m** and **n**

For example:

.{2,6} would match from two to six consecutive copies of any character, inclusive

[a-z]{3,7} would match from three to seven lowercase letters, inclusive

Zero or More Repetitions

The ***** symbol (the **Kleene star**) means that we want to match 0 or more repetitions of the character or group of characters that it follows

The **+** symbol (the **Kleene plus**) matches 1 or more repetitions of the character or group of characters that it follows

\d* would match any number of digits

\d+ would match one or more digits

[abc]+ would match one or more of any **a**, **b**, or **c** character in any combination

Optional Characters

The **?** (question mark) metacharacter denotes optionality

For example, the pattern **ab?c** will match either the strings **abc** or **ac** because the **b** is considered optional

A simple usage of this is similarly matching on spelling differences

- For example: color and colour
- The pattern **colou?r** would match either spelling

If you want to match an actual question mark, you ***escape*** *it*: **\?**

Any Whitespace

Real-world input contains a lot of **whitespace**: spaces, tabs, and newlines

The whitespace special character **\s** will match any of the specific whitespaces

Similarly, **\S** will match any non-whitespace character

Starting and Ending Patterns

We can define a pattern that describes both the start and the end of the line using the special **^** (hat) and **$** (dollar sign) metacharacters

- Note that **^** is used in a different context here; when used in square brackets it means to exclude characters

^success would match only a line that begins with the word **success**

target$ would match only a line that ends with the word **target**

Match Groups

We can use groups of characters and capture them using the parentheses metacharacters

For example, imagine that you had a command line tool to list all the PNG image files you have stored on a computer that start with the letters “IMG” followed by some numbers

You could then use a pattern such as **^(IMG\d+\.png)$** to capture and extract the full filename

Conditional Matches

When using groups, you can use the | (the **pipe**, which represents logical OR) to denote different possible sets of characters

We could write the pattern

Buy more (milk|bread|juice)

That will match only the strings “Buy more milk”, “Buy more bread”, or “Buy more juice”

You can use any sequence of characters or metacharacters in a condition. For example:

([cb]ats*|[dh]ogs?)

This would match “cats”, “cat”, “batsssss”, “dog”, “hogs” and many other strings

Regexes in Python

Python offers some advanced capabilities for working with regexes, but this simple example gives the basics. The **r** character indicates we are using a **raw string**:

```
import re
phone = '123-456-7890'
pattern = r'^\d{3}-\d{3}-\d{4}$'
if re.search(pattern, phone):
   print('The string matches the pattern.')
else:
   print('The string does not match.')
```

Regexes in Python

The **sub** function lets us perform replacements in a string

sub takes three arguments:

- A regex that describes the input string
- A regex that describes what substitutions to perform
- The input string itself

Groups in a regex are numbered starting with 1 and can be identified by **\1**, **\2**, etc.

A simple example that replaces **dog** with **fox**:

```
line = 'the dog and cat'
result = re.sub(r'(.*)(dog)(.*)',
                r'\1fox\3', line)
```

result will be **'the fox and cat'**

Regexes in Python

Another example, which reformats a date given in

MM/DD/YY as **YYYY-MM-DD**:

```
date = '12/25/05'
result = re.sub(r'(\d\d)/(\d\d)/(\d\d)',
                r'20\3-\1-\2', date)
```

result will be '2005-12-25'

Example: Match Filenames

If you use the Linux or Mac OS terminal frequently, you are often dealing with lists of files

Most files have a filename component as well as an extension, but in Linux, it is also common to have hidden files that have no filename

- A hidden file starts with a period

Let's write a regex to match filenames of only image files (jpg, png, gif) that are not hidden

- Note that the filenames must *end* with one of these three file name extensions

Example: Match Filenames

A regular expression that will match the image file names:

\w+\.(png|gif|jpg)

Test cases (green strings match, red ones don't match):

photo1.jpg

fun_times.png

baby22.gif

1.png

schooljpg

.jpg

Example: Match Credit Cards

A credit card number is a sequence of 13 to 16 digits, with a few specific digits at the start that identify the card issuer

- Visa card numbers start with a 4 and have 16 digits
- MasterCard (MC) numbers start with the numbers 51 through 55 and have 16 digits
- American Express (AmEx) card numbers start with 34 or 37 and have 15 digits

A single regular expression to match all three kinds of credit cards (Visa, MC, AmEx):

```
(4\d{15})|(5[1-5]\d{14})|(3[47]\d{13})
```

Example: Match Credit Cards

The regular expression:

(4\d{15})|(5[1-5]\d{14})|(3[47]\d{13})

Test cases (green strings match, red ones don't match):

4829193238102932
4817293823123 23
5180129380293232
5484128379172212
5698274398734598
348282019492392
379187239817298
359820938023232

Example: Match ISBN-10

Suppose a 10-digit ISBN can be written in any of the following sample formats:

- ISBN 0 93028 923 4
- ISBN 1-56389-668-0
- ISBN 1-56389-016-X

The first digit must be a 0 or 1 and the last symbol can be a digit (0-9) or the capital letter X

Let's construct a regular expression for matching ISBNs:

ISBN [01][\s-]\d{5}[\s-]\d{3}[\s-][0-9X]

Example: Match ISBN-10

The regular expression:

ISBN [01][\s-]\d{5}[\s-]\d{3}[\s-][0-9X]

This is not perfect because it matches strings that mix dashes and spaces, like: **ISBN 0-83923 823-0**

- The solution here would be to combine regexes with string processing methods and functions
- More on this issue later

Test cases (green strings match, red ones don't match):

ISBN 0-82918-392-0

ISBN 1 99238 123 X

ISBN 0-83923 823-6

isbn 0-82918-392-0

ISBN-1-33223-233-X

Example: Match Decimal Numbers

Our goal is to match numbers like these:

- **3.14529**
- **-255.34**
- **128**
- **1.9e10**
- **123,340.00**

As with programming, it's best NOT to try to solve the entire problem in one shot

- How might we solve this problem in an incremental fashion (similar to successive prototyping)?
- What kinds of numbers would be easiest to match first?

Example: Match Decimal Numbers

Here is how we might try to piece together the regular expression:

\d+

-?\d+

-?\d+(,\d{3})*

-?\d+(,\d{3})*(\.\d+)?

-?\d+(,\d{3})*(\.\d+)?(e-?\d+)?

Numbers to match:

128

-255.34

3.14529

1.9e10

123,340.00

Example: Match Decimal Numbers

The regular expression:

-?\d+(,\d{3})*(\.\d+)?(e-?\d+)?

Test cases (green strings match, red ones don't match):

3.14529
-255.34
128
1.9e10
1.9e-10
-3.1e8
123,340.00
2,55.3
,213.6
.241

Example: Match Phone Numbers

Suppose we wanted to match US phone numbers that could be written in a variety of forms:

- 1416-555-3456
- 14165553456
- 1416 555 3456
- 1(416) 555-3456
- 1(416)555-3456
- 1 416 555 9292
- 1-416-555-9292
- 1 (416) 555-3456

How could we build a single regular expression that will eventually match all these forms (and possibly more)?

Example: Match Phone Numbers

Here is how we might try to piece together the regular expression:

1\d{3}-\d{3}-\d{4}

1\d{3}-?\d{3}-?\d{4}

1\d{3}[\s-]?\d{3}[\s-]?\d{4}

1[\s-]?\d{3}[\s-]?\d{3}[\s-]?\d{4}

1[\s-]?\(?\d{3}\)?[\s-]?\d{3}[\s-]?\d{4}

Potential Phone Numbers:

1416-555-3456

14165553456

1416 555 3456

1(416) 555-3456

1(416)555-3456

1 416 555 9292

1-416-555-9292

Example: Match Phone Numbers

The regular expression:

1[\s-]?\(?\d{3}\)?[\s-]?\d{3}[\s-]?\d{4}

Test cases (green strings match, red ones don't match):

1416-555-3456
14165553456
1416 555 3456
1(416) 555-3456
1(416)555-3456
1 416 555 9292
1-416-555-9292
1 (416) 555-3456
(416) 555-3456

Example: Match Phone Numbers

This is not a perfect solution. For example, this regex would accept **1(800)-5552468.**

- Sometimes a regular expression that matches only the correct input is very, very complicated.
- In such cases it is usually advisable to use a simpler regex along with string processing functions and methods

For example, with the phone number example, we could do string processing to ensure that dashes are present or absent in the correct places

- Then we could avoid problems like accepting the phone number at the top of this slide

Questions?