The Turing Test

- One of Turing's last papers, written in 1950, was “Computing Machinery and Intelligence”
  - the topic: was a computer really an “electronic brain”?
  - what would it mean for a computer to be called “intelligent”??
- What is now called the Turing Test:
  - connect a terminal to a person in one room and a computer in a different room
  - a user is allowed to type questions on the terminal and send them to either room

Alan Turing (1912 - 1954)

- English mathematician
  - inventor of the Turing machine, an abstract computer used to study theoretical limits of computation
  - we’ll look at Turing machines during the last week of the term
- The Turing Award is presented each year by ACM, a professional society for computer scientists
  - the “Nobel Prize” of computer science
- For more info: Alan Turing: The Enigma, by Andrew Hodges
  - the Enigma was a computer designed by Turing and used by the British in WWII

The Turing Test

- Think of the test as a game with three players
  - Player A is a person typing questions on the terminal -- they win if they correctly guess which room has the computer
  - Player B is the human in one of the rooms -- he wins if he can fool A into thinking he is a computer
  - Player C is the computer -- it wins if it can fool A into thinking it is a human
- C passes the Turing Test if it wins the game
  - it would have to have a fairly high degree of “artificial intelligence” to win....
Chatbots

- For over 50 years the Turing Test was simply an intellectual exercise.
- In the last few years “chatbots” have appeared on the Internet.
  - generate spam and other advertising in chat rooms and instant messaging sites.
- Some chatbots identify messages as being automatically generated.
- Malicious ‘bots try to fool people
  - entice them away to join other chat rooms
  - send credit card or other bank information
- 419 scams
  - “Dearest, my husband died and left millions of dollars....”

ELIZA

- One of the first programs to attempt to carry on a conversation was named ELIZA
  - written by Joseph Weizenbaum at MIT in 1966.
- Applied simple “transformation rules” to create a response to each sentence typed by a user.

  H: My father wouldn’t buy me a puppy.
  C: Tell me more about your family.

  In these slides H and C indicate strings typed by the (human) user and the computer’s response.

- The program does not need to know about parents, kids, pets, buying things, or anything else implied by the input sentence.
- just respond to any sentence with the word “father” by “tell me more....”

The ELIZA Effect

- Weizenbaum was amazed at how involved people got in their “conversations” with ELIZA.
- Computer Power and Human Reason, published in 1976, includes several anecdotes:
  - even when people knew they were talking to a computer they would open up and type very personal details.
  - people often refused to give him transcripts, saying they were “private communications.”
- Weizenbaum compared using ELIZA to going to the theater
  - “willing suspension of disbelief”
  - we know people on the stage are actors, but we are drawn in and think of the characters as real people.

ELIZA Scripts

- The name “ELIZA” comes from the character Eliza Doolittle in G B Shaw’s Pygmalion.
- ELIZA the program has no built-in knowledge:
  - the rules for transforming sentences are defined in a script.
  - the more rules there are in a script, the better the program (presumably) becomes at carrying on a conversation.
  - there can be different scripts for different situations.
- A script designed to act like a psychotherapist would have rules that react to sentences with words like “sad”, “depressed”, “dream”, etc.

My Fair Lady (play and movie) was based on Pygmalion.
The Eliza Module

- The implementation of ELIZA that we will use for the lab is a Ruby module.
  - The module is named Eliza.
  - It's in a file named eliza.rb.
- Modules are like classes.
  - A collection of related methods.
- To call a method in a module, put the name of the module in front of the method name.
  - Call `Eliza.info` to see information about the script currently being used.
  - Call `Eliza.run` to start a conversation.

Doctor

- The script we will use for our project is named Doctor.
  - It is based on the script Weizenbaum first wrote for ELIZA in 1966.
- Use the `load` method in the Eliza module to load a script file.
  - Call `Eliza.load "doctor.ez"`.
  - `Eliza.run`.

The Eliza Module

- When `eliza.rb` is first loaded, there is no script.
- It's possible to start a conversation, but the results are pretty boring.
  - Call `Eliza.run`.
    - `Hi there`.
    - `Hi there`.
    - `Stop repeating what I say`.
    - `Stop repeating what I say`.
    - `Mom!!! Eliza is making fun of me`.
    - `Mom!!! Eliza is making fun of me`.
    - `quit`.
  - `nil`.

Doctor

- The Doctor script is a set of rules that has the computer play the role of a non-judgmental psychotherapist.
  - The doctor simply urges the patient to talk about what is bothering them.
- The simplest responses are statements like “tell me more” or “go on”.
- Slightly more complicated responses use part of the patient’s sentence.
    - `I need a hug`.
    - `Why do you need a hug?`.
A Session with the Doctor

>> Eliza.run
How do you do. Please tell me your problem.
H: I had a dream about cows.
C: What does that dream suggest to you?
H: I think it means I am afraid of them.
C: Is it because you are afraid of them that you came to me?
H: Yes.
C: You seem quite positive.
H: Cows terrify me.
C: I am not sure I understand you fully.
H: I can't go past a farm without breaking out in sweat.
C: How do you know you can't go past a farm without breaking out in sweat?

A Session with the Doctor

Most sessions quickly degenerate...
H: Do you have a recipe for eggplant parmigiana?
C: Oh, I have a recipe for eggplant parmigiana?
H: You're a cooking expert, aren't you?
C: You're not really talking about me, are you?
H: Yes.
C: You are sure?
H: I thought I was talking to a chef.
C: That's quite interesting.
H: Are you a computer?
C: Why do you mention computers?

Getting Information About a Script

For a conversation to be realistic the user needs to enter sentences that have words the script reacts to
To get information about the current script:
>> Eliza.load "doctor.ez"
=> true
>> Eliza.info
Script: doctor.ez
28 rules
Keys:
   alike, always, are, because, believe, brother, certainly, child, children, computer, computers, dad, deutsch, dream, dreamed, dreams, dreamt, ...

Regular Expressions

Since responses are based on patterns we need a way to describe sentence patterns
The obvious choice: regular expressions
Pattern: “I don’t _____”
Response: “Why don’t you _____?”
Pattern: “I need _____”
Response: “Why do you need _____?”

Pattern: /I need/ 
Response: “Why do you mention computers?”

>> s = "I need a hug"
=> "I need a hug"

Pattern: /I need/ 
Response: “Why do you mention computers?”

>> s = 0
=> 0
Word Breaks

- The goal: patterns should match key words in an input sentence
- Example: a pattern that finds the word "cow"
  ```ruby
  >> "How now, brown cow" =~ /cow/
  => 15
  
  But that pattern is too flexible
  ```
  ```ruby
  >> "My cat is scowling at me" =~ /cow/
  => 11
  
  We need to put anchors around the key word:
  ```
  ```ruby
  >> "How now, brown cow" =~ /\bcow\b/  
  => 15
  >> "My cat is scowling at me" =~ /\bcow\b/  
  => nil
  ```

Groups

- It's time to introduce a useful new construct used in regular expressions
- A **group** in a pattern is a sequence of characters enclosed in parentheses
- Example:
  ```ruby
  >> "How now, brown cow" =~ /\w+ cow\b/  
  => 9
  >> "How now, brown cow" =~ /\(\w+) cow\b/  
  => 9
  ```

Adding parentheses doesn't affect what the expression matches
- like adding parentheses to an arithmetic expression
- $(4 * 5) * 6$ has the same value as $4 * 5 * 6$

- The useful thing about groups: Ruby saves the part of the input sentence that matches the group
  ```ruby
  >> "How now, brown cow" =~ /(\w+) cow\b/  
  => 15
  >> $1  
  => "brown"
  ```

- If a regular expression has $n$ groups Ruby stores the characters that match the groups in variables named $\$1$, $\$2$, ..., $\$n$
  ```ruby
  r = /I'm (.*) I (.*)/  
  => /I'm (.*) I (.*)/  
  r.class  
  => Regexp
  >> r  
  => /I'm (.*) I (.*)/  
  r.class  
  => Regexp
  >> "I'm happy I stayed" =~ r  
  => 0
  >> $1  
  => "happy"
  >> $2  
  => "stayed"
  >> "I'm glad I didn't buy that computer" =~ r  
  => 0
  >> $1  
  => "glad"
  >> $2  
  => "didn't buy that computer"
  ```

Make a regular expression with 2 groups, save it in $r$ (so we can use it more than once)

See if this sentence matches $r$

$\$1$ and $\$2$ have the parts of the sentence that matched the groups
Fill in the Blanks

- Recall from previous slides that we can use “interpolation” to insert values of variables into strings
  
  ```
  >> n = 101
  => 101
  >> s = "She had #{n} dalmatians"
  => "She had 101 dalmatians"
  ```

- Interpolation and groups give us just what we need to make “fill in the blank” types of sentence patterns
  
  ```
  >> r = /I don't (.*)/
  => /I don't (.*)/
  >> "I don't like cows" =~ r
  => 0
  >> $2
  => "cows"
  ```

Summary

- The “building blocks” we’ll use for making patterns to match against input sentences are
  
  - individual words, surrounded by word break anchors
  ```
  /\bdream\b/  
  ```
  
  - wild cards and character classes
  ```
  /I dreamed I .*/  
  ```

- groups used to capture pieces of the input sentence
  ```
  /I dreamed I (.*)/  
  ```

- alternatives (so we can make one common pattern instead of lots of separate but very similar patterns)
  ```
  /I (dreamed|dreamt) I (.*)/  
  ```

Alternatives

- Another useful construct for regular expressions: **alternatives**
  
  - We can put a list of words, separated by a vertical bar, inside a group
  
  ```
  >> r = /I don't (.*) (cows|snakes|spiders)/
  => /I don't (.*) (cows|snakes|spiders)/
  >> "I don't like snakes" =~ r
  => 0
  >> $2
  => "snakes"
  >>> "I don't eat cows" =~ r
  => 0
  >> $2
  => "cows"
  ```

Pattern Objects

- To make it easier to create scripts the Eliza module defines a new type of object called a Pattern
  
  - think of a Pattern object as a “fill-in-the-blank” type of transformation
  
  - The two parts that define such a pattern are
  ```
  p = Pattern.new(/I (want|need) (.*)/, "Why do you $1 $2?")
  ```

Note: the constructor automatically adds the `{}` around variables...
Pattern Objects

- After you make a Pattern object you can apply it to sentences
  ```ruby
  p = Pattern.new( /I (want|need) (.*)/, "Why do you $1 $2?"
  => /I (want|need) (.*)/: ["Why do you $1 $2?"]
  p.class
  => Pattern
  p.apply("I want a new car")
  => "Why do you want a new car?"
  p.apply("I need a hug")
  => "Why do you need a hug?"
  ```
- The apply method simply combines all the things we've seen on previous slides into one convenient method
  - see if the sentence matches the pattern's regular expression
  - if so collect the pieces in $1, ... and plug them into the reassembly string

A Tiny Bit of Realism

- Pattern objects help add a little more realism to a conversation
  - a real person would respond with a variety of different sentences
- You can pass a list of reassembly strings to Pattern.new
  ```ruby
  p = Pattern.new( /I (want|need) (.*)/, ["Why do you $1 $2?", "Would you be happy if you got $2?"]
  p.apply("I need a hug")
  => "Why do you need a hug?"
  p.apply("I need a hug")
  => "Would you be happy if you got a hug?"
  p.apply("I need a hug")
  => "Why do you need a hug?"
  ```

Postprocessing

- Our sample pattern doesn't do a very good job with this sentence:
  ```ruby
  p = Pattern.new( /I (want|need) (.*)/, "Why do you $1 $2?"
  p.apply("I need to take my pills")
  => "Why do you need to take my pills?"
  ```
- This sort of thing is easily fixed by postprocessing
  ```ruby
  p = Pattern.new( /I (want|need) (.*)/, ["Why do you $1 $2?", "Would you be happy if you got $2?", "Are you really $1?", "Why do you think you are $1?""]
  p.apply("I am going crazy")
  => "Are you really going crazy?"
  ```
- optional parameter passed to apply
  ```ruby
  p.apply("I need to take my pills", {"my" => "your"})
  => "Why do you need to take your pills?"
  ```

Summary

- The Pattern class defined in the Eliza module makes it easy to define “fill in the blanks” types of sentence patterns
  ```ruby
  p = Pattern.new(/I am (.*)/, ["Are you really $1?","Why do you think you are $1?"])
  ```
- pass a sentence to the apply method
  ```ruby
  p = Pattern.new(/I am (.*)/, ["Are you really $1?","Why do you think you are $1?"])
  p.apply("I am going crazy")
  => "Are you really going crazy?"
  ```
- apply takes care of capitalization, word breaks, and interpolation
Summary

- An optional “dictionary” parameter allows `apply` to do postprocessing
  ```ruby
  >> p.apply("I am sure I saw a Martian")
  => "Are you really sure I saw a Martian?"
  >> p.apply("I am sure I saw a Martian", \{"I" => "you"\})
  => "Are you really sure you saw a Martian?"
  ```

Details

- The Pattern class takes care of a lot of messy little details automatically
- It adds the `\b` anchors around single words:
  ```ruby
  >> p = Pattern.new(/cow/, "Tell me more about your farm.")
  >> p.apply("The cow jumped over the moon")
  => "Tell me more about your farm."
  >> p.apply("We met a cowardly lion.")
  => nil
  ```

- It deals with capitalization:
  ```ruby
  >> "Cow tipping" =~ /cow/  # Regular expressions are case-sensitive
  => nil
  >> p.apply("Cow tipping is not a sport.")
  => "Tell me more about your farm."
  ```

Aside: Hash Objects

- The optional second parameter passed to `apply` is an **associative array**
  - a set of **key-value pairs** enclosed in braces
  - any number of pairs can be given, separated by commas
  ```ruby
  >> dict = \{ "me" => "you", "my" => "your", "myself" => "yourself" \}
  >> p.apply("I want to pace myself")
  => "Why do you want to pace myself?"
  >> p.apply("I want to pace myself", dict)
  => "Why do you want to pace yourself?"
  ```
- Ruby (using terminology from Perl) calls these things “hashes”
  ```ruby
  >> dict.class
  => Hash
  ```

Hash Objects

- The name “hash” gives a hint of how these objects are represented internally
  - the keys are stored in a hash table
  - the table has a pointer to a value associated with each key
Hash Objects

- Associative arrays ("hashes") are useful in a wide range of situations known as "maps" in Java, C++, and other languages.
- An example in Ruby:

  ```ruby
  a = Hash.new
  a["apple"] = "fruit"
  a["banana"] = "fruit"
  a["onion"] = "vegetable"
  ```

  Make a new object by calling the constructor
  ```ruby
  a = Hash.new
  => {} 
  ```

  Add a key-value pair
  ```ruby
  a["apple"] = "fruit"
  => "fruit"
  ```

  Returns the value associated with key k
  ```ruby
  a["banana"]
  => "fruit"
  ```

- The items in a hash are not stored in order.
  - The class figures out the most efficient way to organize the keys.
  - May be a hash table, or may be a tree.

  ```ruby
  a.keys
  => ["onion", "apple", "zucchini", "banana", "eggplant", "tomato"]
  ```

  ```ruby
  a.values
  => ["vegetable", "fruit", "vegetable", "fruit", "fruit", "fruit"]
  ```

- Examples of other methods in the Hash class:

  ```ruby
  a.each_key { |k| puts k if a[k] == "vegetable" }
  onion
  zucchini
  ```

Overview of ELIZA

- One way to implement ELIZA would be to make an array of Pattern objects.

  ```ruby
  patterns = []
  patterns << Pattern.new( /.../, ["..."] )
  ```

  ```ruby
  sentence = readline
  patterns.each do |p|
    if response = p.apply(sentence)
      puts response
      break
    end
  end
  ```

  This loop is basically a linear search.
  Will be very inefficient when a script has a lot of patterns.
Overview of ELIZA

- The actual implementation uses an associative array
  - store key words in a Hash
  - associate keys with a set of Pattern objects that transform sentences containing those keys

The data structure used to hold the rules is called a **priority queue**

- think of it as a list that is automatically re-sorted each time something is added
- Rules can be assigned a priority
  - higher priority rules end up at the front of the list
- After scanning the input sentence try to apply the rules in the queue
  - if a rule does not apply move on to the next one

The ELIZA Algorithm

- ELIZA breaks a sentence into words
- If there is a rule for a word the word is added to a list

```ruby
> s = "I had another dream, this time about cows"
> s.scan(/\w+/) { |w| puts w }
```

```
# simplified version; see eliza.rb
# for the actual code
line.scan(/\w+/) do |w|
  if r = rules[w]
    queue.insert(r)
  end
end
```

- The ELIZA Algorithm

```ruby
# simplified version; see eliza.rb
# for the actual code
while queue.length > 0
  rule = queue.shift
  if result = rule.apply(line)
    return result
  end
end
return nil
```

- If you want to see this process in action, tell ELIZA to work in “verbose mode”

```ruby
> Eliza.verbose
=> true
```

```
H: I think I am afraid of cows.
```

```
> preprocessing: line = 'i think i am afraid of cows'
```

```
preprocessing expands contractions ("I’m" => "I am", etc)
```

```
"I" is associated with lots of patterns; they are tried in order
```

```
postprocess am afraid of cows
```

```
C: But you are not sure you are afraid of cows?
```
It is possible to build up a script interactively, but it's easier to save the rules in a script file:

- Use any text editor to make a file containing rule definitions.
- Load the file by calling `Eliza.load`.

**Analogy:**
Using a text editor to edit definitions of Ruby methods, loading them into IRB.

There are lots of things that can go into a script file:

- The "user manual" promised in the chapter is not finished.
- Here are some of the basics -- it should be enough to get you started if you want to try the extra credit assignment.
- The general format of a rule is shown by this example:

```
i (want|need) (.*)/!
  "What would it mean to you if you got $2?"
  "Why do you $1 $2?"
i am .*(sad|unhappy|depressed|sick)(.*)/!
  "I am sorry to hear you are $1$2."
  "Do you think coming here will help you not to be $1$2?"
```

A simple rule just has a keyword and a list of responses:

```
perhaps
  /* .*/
  "You don't seem quite certain."
  "Why the uncertain tone?"
```

You can define a variable and use it as a keyword:

```
:alias $computer computer computers machine machines

$computer 50

/i (computer|machine)s?/
  "Do $1s worry you?"
  "Why do you mention $1s?"
  "What do you think $1s have to do with your problem?"
```

This rule is added to the queue if any one of the four words is found in the input sentence.

Can you see how this rule handles sentences like "I am afraid of computers"?

What about "That computer scares me"?
Class Project: The Argument Clinic

Patient: Is this the right room for an argument?
Clinician: I told you once.
Patient: No you haven’t.
Clinician: Yes I have.
Patient: When?
Clinician: Just now.
Patient: No you didn’t.
Clinician: Yes I did.
Patient: You did not!!
Clinician: Oh, I’m sorry, just one moment. Is this a five minute argument or the full half hour?
Patient: Oh, just the five minutes.
Clinician: Ah, thank you. Anyway, I did.

Copyright (?)
Find full text by Google search for “python ‘all the words’”
or watch the skit on YouTube

The Argument Clinic

Patient: Oh look, this isn’t an argument.
Clinician: Yes it is.
Patient: No it isn’t. It’s just contradiction.
Clinician: No it isn’t.
Patient: It is!
Clinician: It is not.
Patient: Look, you just contradicted me.
Clinician: I did not.
Patient: Oh you did!!
Clinician: No, no, no.
Patient: You did just then.
Clinician: Nonsense!
Patient: Oh, this is futile!
Clinician: No it isn’t.

Can we make an ELIZA script that will play the role of the “clinician”?
If sentence has “this is” or “it is” reply with “no, it isn’t”
If sentence has “isn’t” reply with “yes, it is”
If you want to give it a try download python.ez

Another extra credit idea:
sports teams

Review

- ELIZA is a simple system that generates responses by applying simple transformation rules
  - a rule associates key words with sentence patterns that control how ELIZA will respond to sentences that contain those words
  - a pattern is defined by a regular expression and associated reassembly strings
  - “fill in the blanks” patterns reuse parts of the input sentence to construct the response

- How ELIZA responds is determined by a script
  - different scripts can contain different key words
  - psychotherapy clinic, argument clinic, sports bar, ...

Questions

- How “smart” is ELIZA?
- With the “doctor” script it clearly won’t pass the Turing Test. Could it ever?
  - i.e. will adding more rules to this script ever make it “intelligent enough”?
- Why or why not?
Language is More than Syntax

- Suppose we hope to write an ELIZA script to answer questions like this: How much Chinese silk was exported to Western Europe by southern merchants?
- You might think it’s just a matter of adding a “knowledge base” with facts and figures about the silk trade.
- But now consider this question: How much Chinese silk was exported to Western Europe by the 18th century?
- The word “by” takes on a completely different meaning:
  - in the first sentence it refers to actors
  - in the second sentence it refers to time
- Words have very different semantics (meanings) in different contexts.

Example from Speech and Language Processing (see link on class web site)

Mental Models

- ELIZA does not have any memory:
  - it does not have a way of remembering anything that has gone on earlier in the conversation.
- More generally, ELIZA does not have any way to form a mental model of the person it is conversing with.
- When you reply to a question like “Is it raining outside?” you unconsciously ask yourself “why did they ask that?”
  - if you are in the same room and about to go outside: “yes, take a jacket”
  - if you are talking on the phone: “we’re having our usual winter”
  - if you are planning a picnic: “let’s go to the park with the covered gazebo”
- Context plays a big role...

Inference

- The responses on the previous slide are examples of a more general process known as inference generation.
- We are constantly “filling in the blanks” and making inferences as we listen to sentences.
- Consider this story:
  - “The cows walked across the pasture.  George was waiting for them at the barn.”
  - How would you answer the question: “Where were the cows going?”
- By “connecting the dots” between the two sentences it seems clear the cows were headed to the barn.
  - this inference depends on knowing what “waiting” means
  - you might also make lots of other inferences (it was late in the day, George was getting ready to do the milking, ...)

Real-World Knowledge

- Simple answers to the question about George and the cows are based on common sense knowledge:
  - if George was waiting for the cows the cows must have been going to where George was standing.
- Other answers might be based on expert knowledge of farms and cows:
  - cows walk to the barn when they know it’s time to be milked, cows are milked twice a day, ...
- Paradoxically, more progress has been made in creating programs that use expert knowledge:
  - medical diagnosis
  - engineering (e.g. tracking down faults in systems)
  - business and finance
Aside: TALE-SPIN

- James Meehan, an AI researcher at Yale, wrote a program named TALE-SPIN.
  - The program's goal was to write children's stories.
- In order to write stories, the program had to know a large number of simple concepts.
  - Things we learn before we are 5 years old and take for granted.
- Examples of real-world knowledge that has to be in such a program:
  - If a character throws a ball, the ball and the character are in different places.
  - If a character pushes an object, the character and object are in the same place.
  - Something falls because it is pushed by gravity.
  - If a character falls in water, it must get out.
  - Someone can get out of water if they can swim or if they have friends to save them.

Language is More Than Syntax (II)

- What is the meaning of this sentence when different words are stressed?
  - \( I \) never said she stole my money.
  - \( I \) never said she stole my money.
  - \( I \) never said \( she \) stole my money.
  - \( I \) never said she stole \( my \) money.
  - \( I \) never said she stole \( my \) money.
- The meaning depends on auditory clues, not just the meaning of the individual words.

A Hard Problem

- Several hard problems were described in Ch 1 of the textbook.
  - Playing the perfect game of chess:
    - A problem of scalability.
    - We'll see another one next week -- the traveling salesman.
- The “halting problem”:
  - Will a program ever terminate?
  - A mathematically impossible problem.
- Natural language processing:
  - A problem that is very hard to define.
  - Will computers ever be as good as humans at understanding and using language?