Dr. Marie desJardins on Fairness and Equity in Data Science: Challenges and Possibilities

Friday, Oct 7, 4:00 p.m.
Location:
New England 206

Dr. desJardins will talk about the current state of data science, machine learning, and AI; the future of these technologies; the importance of diversity for creating robust, effective engineering solutions; and how we can thoughtfully ensure that data science and computing will positively affect our lives and the lives of generations to come. Jointly sponsored by the Computer Science Department and Data Science & Society.

Reception to follow in Sanders Physics 105 from 5:00-6:00 p.m.
Where are we?
We’ve been working with tables for the past few weeks.

Last class we saw a new data type: lists.
### grades

<table>
<thead>
<tr>
<th>number-grade</th>
<th>letter-grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>100</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>74</td>
<td>&quot;C&quot;</td>
</tr>
<tr>
<td>84</td>
<td>&quot;B&quot;</td>
</tr>
</tbody>
</table>

```python
list:
    "A",
    "A",
    "C",
    "B"
```
```python
>>> grades

<table>
<thead>
<tr>
<th>number-grade</th>
<th>letter-grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>100</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>74</td>
<td>&quot;C&quot;</td>
</tr>
<tr>
<td>84</td>
<td>&quot;B&quot;</td>
</tr>
</tbody>
</table>

``` python
```python
>>> grades.get_column("letter-grade")

["A", "A", "C", "B"]
```
Columns in a table can contain a mix of different data types, e.g.,

```
table: grades
  row: 98
  row: 56
  row: 74
  row: "F"
  row: "A"
  row: "B"
end
```

And so can a list:

```
list: 98, 56, 74, "F", "A", "B"
```
However, we usually find it easier to work with a column where every value is of the same kind.

We can **annotate** the type of data in the column when we make a table:

```plaintext
table: col :: Number
row: 1
row: 2
row: 3
end
```

```plaintext
table: col :: String
row: "a"
row: "b"
row: "c"
end
```
Likewise, we’ll most often have just one type of data in a list, and we can show that when we write the type annotation for a function:

For example,

\[
\text{[list: 1, 2, 3]} \quad \text{List<Number>}
\]

\[
\text{“a list of numbers”}
\]

\[
\text{[list: "a", "b", "c"]} \quad \text{List<String>}
\]

\[
\text{“a list of strings”}
\]
Much like the rows in a table, the items in a list have numeric indices:

```python
>>> lst = ['a', 'b', 'c']
```

And we can access items using these indices:

```python
>>> L.get(lst, 0)
'a'
>>> L.get(lst, 1)
'b'
```
The length of a list is always one more than the last item index:

```python
>>> lst = ['a', 'b', 'c']
>>> length(lst)
3
```
To check if an item is in a list, we can just ask if the list has it as a member:

```python
>>> lst = ["a", "b", "c"]
>>> lst.member("c")
true
```
We used higher-order functions to work with tables, and we can do the same with lists:

Table: `transform-column`  List: `map`
We used higher-order functions to work with tables, and we can do the same with lists:

*Tables*  
transform-column → map  
filter-with → filter  

*Lists*
>>> lst = ['a', 'b', 'c']
>>> filter(
    lambda i: not(i == 'a')
    , lst)
['b', 'c']
>>> \texttt{lst} = [\texttt{"a"}, \texttt{"b"}, \texttt{"c"}]
>>> \texttt{filter(}
    \texttt{lam(i):} \texttt{not(i == "a") end,}
    \texttt{lst})
[\texttt{"b"}, \texttt{"c"}]

This is an anonymous (i.e., unnamed) function made using a lambda expression.
One difference to be aware of:

\[
\text{filter-with(} \langle \text{table} \rangle, \langle \text{function} \rangle \rangle
\]

\[
\text{filter(} \langle \text{function} \rangle, \langle \text{list} \rangle \rangle
\]

When you’re working with a list, the function argument comes first.
At the end of last class, we considered what we could do if there wasn’t a built-in function, so we needed to write a function that looked at each item in a list.
Designing list functions
How would we write a function that takes a list of numbers and returns its sum?
fun my-sum(lst :: List<Number>) -> Number:
    doc: "Return the sum of the numbers in the list"
    ...
end
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list: ]) is ...
end
We can have a string with no characters in it:
   ""

And, likewise, we can have a list with no items in it:
   [list: ]

For these data types, these values are the equivalent of 0, the number representing no quantity.
fun my-sum(lst :: List<Number>) -> Number:
    doc: "Return the sum of the numbers in the list"
    ...
where:
    my-sum([list: ]) is 0
end
fun my-sum(lst :: List<Number>) -> Number:
    doc: "Return the sum of the numbers in the list"
    ...
    where:
    my-sum([list: ]) is 0
    my-sum([list: 4]) is 4
end
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  ...
where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4
  my-sum([list: 1, 4]) is 1 + 4
end
fun my-sum(lst :: List<Number>) -> Number:
    doc: "Return the sum of the numbers in the list"
    ...
where:
    my-sum([list: ] ) is 0
    my-sum([list: 4] ) is 4
    my-sum([list: 1, 4] ) is 1 + 4
    my-sum([list: 3, 1, 4] ) is 3 + 1 + 4
end
fun my-sum(lst :: List<Number>) -> Number:
    doc: "Return the sum of the numbers in the list"
    ...
where:
    my-sum([list:        ]) is 0
    my-sum([list:        4]) is 4
    my-sum([list: 1, 4]) is 1 + 4
    my-sum([list: 3, 1, 4]) is 3 + 1 + 4
end
fun my-sum(lst :: List<Number>) -> Number:

doc: "Return the sum of the numbers in the list"

... where:

  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + 0
  my-sum([list: 1, 4]) is 1 + 4 + 0
  my-sum([list: 3, 1, 4]) is 3 + 1 + 4 + 0

end
fun my-sum(lst :: List<Number>) -> Number:
   doc: "Return the sum of the numbers in the list"

where:
   my-sum([list: ]) is 0
   my-sum([list: 4]) is 4 + my-sum([list: ])
   my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
   my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
fun my-sum(lst :: List<Number>) -> Number:
    doc: "Return the sum of the numbers in the list"
    ...

where:
    my-sum([list: ]) is 0
    my-sum([list: 4]) is 4 + my-sum([list: ])
    my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
    my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
The secret nature of lists
Writing our input as \([\text{list: } 3, 1, 4]\) is a lie.

It’s just a shorthand for the real structure of a list.
In its secret heart, Pyret knows there are only two ways of making a list.

A list is either:

- **empty**
- `link(item, list)`
A list of one item, e.g.,

```
[list: "A"],
```

is really a link between an item and the empty list:

```
link("A", empty)
```
[list:

"A", link("A",

"A", link("A",

"C", link("C",

"B"] link("B",

empty)))]
Is \texttt{link(3, 4)} a valid list?
Is \texttt{link(3, 4)} a valid list?
Designing functions using the definition of a list
To write our own functions to process a list, item by item, we need to use the true form of a list and think *recursively*.
Recursion is a technique that involves defining a solution or structure using itself as part of the definition.
fun **my-sum**(lst :: List<Number>) -> Number:

doc: "Return the sum of the numbers in the list"

cases (List) lst:

  | empty =>
  |         ...

  | link(f, r) =>
  |         ...

end

where:

my-sum([list: ]) is 0
my-sum([list: 4]) is 4 + my-sum([list: ])
my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
    | empty =>
      ...
    | link(f, r) =>
      ...
  end

  where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
  end

**cases** is like a special if statement that we use to ask "which shape of data do I have?"
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

cases (List) lst:
  | empty => ...  
  | link(f, r) => ...
end

where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
fun my-sum(lst :: List<Number>) -> Number:
    doc: "Return the sum of the numbers in the list"
    cases (List) lst:
    | empty => ...
    | link(f, r) => ...
    end

    where:
    my-sum([list: ]) is 0
    my-sum([list: 4]) is 4 + my-sum([list: ])
    my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
    my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
    end

Denotes the output of a function
Marks the expression to evaluate if the data has the shape on the left.
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
  | empty =>
  ...

  | link(f, r) =>
  ...

  end

  where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
fun my-sum(lst :: List<Number>) -> Number:
    doc: "Return the sum of the numbers in the list"

    cases (List) lst:
        | empty =>
            ...
        | link(f, r) =>
            ...
    end

where:
    my-sum([list: ]) is 0
    my-sum([list: 4]) is 4 + my-sum([list: ])
    my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
    my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
fun my-sum(lst :: List<Number>) -> Number:

doc: "Return the sum of the numbers in the list"

cases (List) lst:
  | empty => 0
  | link(f, r) => ...
end

where:

my-sum([list: ]) is 0
my-sum([list: 4]) is 4 + my-sum([list: ])
my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
fun my-sum(lst :: List<Number>) -> Number:
    doc: "Return the sum of the numbers in the list"

cases (List) lst:
    | empty =>
        0
    | link(f, r) =>
        f + my-sum(r)
end

where:
    my-sum([list: ]) is 0
    my-sum([list: 4]) is 4 + my-sum([list: ])
    my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
    my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"

  cases (List) lst:
  | empty =>
  |   0

  | link(f, r) =>
  |   f + my-sum(r)

end

where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
fun my-sum(lst :: List<Number>) -> Number:

doc: "Return the sum of the numbers in the list"

cases (List) lst:
    | empty => 0
    | link(f, r) => f + my-sum(r)
end

where:

    my-sum([list: ]) is 0
    my-sum([list: 4]) is 4 + my-sum([list: ])
    my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
    my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
fun my-sum(lst :: List<Number>) -> Number:
  doc: "Return the sum of the numbers in the list"
  cases (List) lst:
    | empty   => 0
    | link(f, r) => f + my-sum(r)
  end
where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
When we call this function, it evaluates as:

\[
\text{my-sum(link(3, link(1, link(4, empty))))} \\
\rightarrow 3 + \text{my-sum(link(1, link(4, empty)))} \\
\rightarrow 3 + 1 + \text{my-sum(link(4, empty))} \\
\rightarrow 3 + 1 + 4 + \text{my-sum(\text{empty})} \\
\rightarrow 3 + 1 + 4 + 0
\]
Thinking recursively
Any time a problem is structured such that the solution on larger inputs can be built from the solution on smaller inputs, recursion is appropriate.
All recursive functions have these two parts:

**Base case(s):**

What’s the simplest case to solve?

**Recursive case(s):**

What’s the relationship between the current case and the answer to a slightly smaller case?

You should be calling the function you’re defining here; this is referred to as a *recursive call*. 
fun `recursive-function`(lst :: List) -> ...:

cases (List) lst:

  | empty =>
  |
  | link(f, r) =>
  |
  | ... recursive-function(r) ...

end  
end
Each time you make a recursive call, you must make the input smaller somehow.

If your input is a list, you pass the rest of the list to the recursive call.
link("A",

link("A",

link("C",

link("B",

empty))))
First

\[
\text{link}(\text{"A"}, \text{link}(\text{"C"}, \text{link}(\text{"B"}, \text{empty}))))
\]
```python
>>> lst = [list: "item 1", "and", "so", "on"]
>>> lst.first
"item 1"
>>> lst.rest
[list: "and", "so", "on"]
```
cases (List) lst:
  | empty => ...
  | link(f, r) => ...
end

First
Rest
What happens if we *don’t* make the input smaller?
fun my-sum(lst :: List<Number>) -> Number:
  cases (List) lst:
    | empty => 0
    | link(f, r) => f + my-sum(r)
  end
where:
  my-sum([list: ]) is 0
  my-sum([list: 4]) is 4 + my-sum([list: ])
  my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
  my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end

Recursive call on the rest of the input list
fun my-sum(lst :: List<Number>) -> Number:
    cases (List) lst:
        | empty => 0
        | link(f, r) => f + my-sum(lst)
    end

where:
    my-sum([list: ]) is 0
    my-sum([list: 4]) is 4 + my-sum([list: ])
    my-sum([list: 1, 4]) is 1 + my-sum([list: 4])
    my-sum([list: 3, 1, 4]) is 3 + my-sum([list: 1, 4])
end
When we call this function, it evaluates as:

\[
\begin{align*}
\text{my–sum(link(3, link(1, link(4, empty))))} \\
\rightarrow & \quad 3 + \text{my–sum(link(3, link(1, link(4, empty))))} \\
\rightarrow & \quad 3 + 3 + \text{my–sum(link(3, link(1, link(4, empty))))} \\
\rightarrow & \quad 3 + 3 + 3 + \text{my–sum(link(3, link(1, link(4, empty))))} \\
& \ldots
\end{align*}
\]

This isn’t going to end well.
When a recursive function never stops calling itself, it’s called *infinite recursion*. 
Practice designing recursive functions
The function \texttt{any\text{-}below\text{-}10} should return \texttt{true} if any member of the list is less than 10 and \texttt{false} otherwise.
fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Return true if any number in the list is less than 10"

  ...

where:
  any-below-10([list: 3, 1, 4]) is (3 < 10) or (1 < 10) or (4 < 10)
  any-below-10([list: 1, 4]) is (1 < 10) or (4 < 10)
  any-below-10([list: 4]) is (4 < 10)
  any-below-10([list: ]) is ...

end
fun any-below-10(lst :: List<Number>) -> Boolean:
    doc: "Return true if any number in the list is less than 10"

    ...

where:
    any-below-10([list: 3, 1, 4]) is (3 < 10) or (1 < 10) or (4 < 10)
    any-below-10([list: 1, 4]) is (1 < 10) or (4 < 10)
    any-below-10([list: 4]) is (4 < 10)
    any-below-10([list: ]) is ...
end

What goes here?
fun any-below-10(lst :: List<Number>) -> Boolean:
    doc: "Return true if any number in the list is less than 10"

    ...

where:
    any-below-10([list: 3, 1, 4]) is (3 < 10) or (1 < 10) or (4 < 10)
    any-below-10([list: 1, 4]) is (1 < 10) or (4 < 10)
    any-below-10([list: 4]) is (4 < 10)
    any-below-10([list: ]) is false
end
fun any-below-10(lst :: List<Number>) -> Boolean:
    doc: "Return true if any number in the list is less than 10"

    ...

where:
    any-below-10([list: 3, 1, 4]) is (3 < 10) or (1 < 10) or (4 < 10)
    any-below-10([list: 1, 4]) is (1 < 10) or (4 < 10)
    any-below-10([list: 4]) is (4 < 10)
    any-below-10([list: ]) is false
end
fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Return true if any number in the list is less than 10"

  ...

where:
  any-below-10([list: 3, 1, 4]) is (3 < 10) or any-below-10([list: 1, 4])
  any-below-10([list: 1, 4]) is (1 < 10) or any-below-10([list: 4])
  any-below-10([list: 4]) is (4 < 10) or any-below-10([list: ])
  any-below-10([list: ]) is false
end
fun any-below-10(lst :: List<Number>) -> Boolean:
  doc: "Return true if any number in the list is less than 10"
  cases (List) lst:
    | empty => false
    | link(f, r) => (f < 10) or any-below-10(r)
  end
where:
  any-below-10([list: 3, 1, 4]) is (3 < 10) or any-below-10([list: 1, 4])
  any-below-10([list: 1, 4]) is (1 < 10) or any-below-10([list: 4])
  any-below-10([list: 4]) is (4 < 10) or any-below-10([list: ])
  any-below-10([list: ]) is false
end
Now that we’ve seen how to write `any-below-10`, we can use the same pattern to implement a higher-order function where we can ask if any item in a list satisfies some predicate.
fun my-any(fn :: Function, lst :: List) -> Boolean:
    doc: "Return true if the function fn is true for any item in the given list."
    cases (List) lst:
        | empty => false
        | link(f, r) => fn(f) or my-any(fn, r)
    end
end
fun my-any(fn :: Function, lst :: List) -> Boolean:
  doc: "Return true if the function fn is true for any item in the given list."
  cases (List) lst:
    | empty => false
    | link(f, r) => fn(f) or my-any(fn, r)
  end
end

fun my-all(fn :: Function, lst :: List) -> Boolean:
  doc: "Return true if the function fn is true for every item in the given list."
  cases (List) lst:
    | empty => true
    | link(f, r) => fn(f) and my-all(fn, rst)
  end
end
fun any-below-10(lst :: List<Number>) -> Boolean:
    doc: "Return true if any number in the list is less than 10"
    any(lam(x): x < 10 end, lst)
where:
    any-below-10([list: 3, 1, 4]) is true
    any-below-10([list: 11, 14]) is false
    any-below-10([list: ]) is false
end
This is how you *should* write this function – use built-in higher-order functions like `any` when you can!
Wrap-up practice
fun list-len(lst :: List) -> Number:
  doc: "Compute the length of a list"
  cases (List) lst:
    | empty => 0
    | link(f, r) => 1 + list-len(____)
  end
end
fun list-len(lst :: List) -> Number:
  doc: "Compute the length of a list"
  cases (List) lst:
    | empty => 0
    | link(f, r) => 1 + list-len(r)
  end
end
end
fun list-product(lst :: List<Number>) -> Number:
  doc: "Compute the product of all the numbers in lst"
  cases (List) lst:
    | empty => 1
    | link(f, r) => ____ * list-product(r)
  end
end
fun list-product(lst :: List<Number>) -> Number:
    doc: "Compute the product of all the numbers in lst"
    cases (List) lst:
        | empty => 1
        | link(f, r) => f * list-product(r)
    end
end
fun is-member(item, lst :: List) -> Boolean:
  doc: "Return true if item is a member of lst"
  cases (List) lst:
    | empty => ______
    | link(f, r) =>
      (f == ______) or (is-member(_____, ______))
  end
end
fun is-member(item, lst :: List) -> Boolean:

doc: "Return true if item is a member of lst"

cases (List) lst:
    | empty  => false
    | link(f, r) => (f == item) or (is-member(item, r))
end
end
Final note

Lists, recursion, and cases syntax are not easy concepts to grasp separately, much less all together in a short time.

Don’t feel frustrated if it takes a little while for these to make sense. Give yourself time, be sure to practice working in Pyret, and ask questions.
Class code:

https://tinyurl.com/101-2022-10-05
Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University
Doug Woos, Brown University