CMPU 101 § 2 · Computer Science I

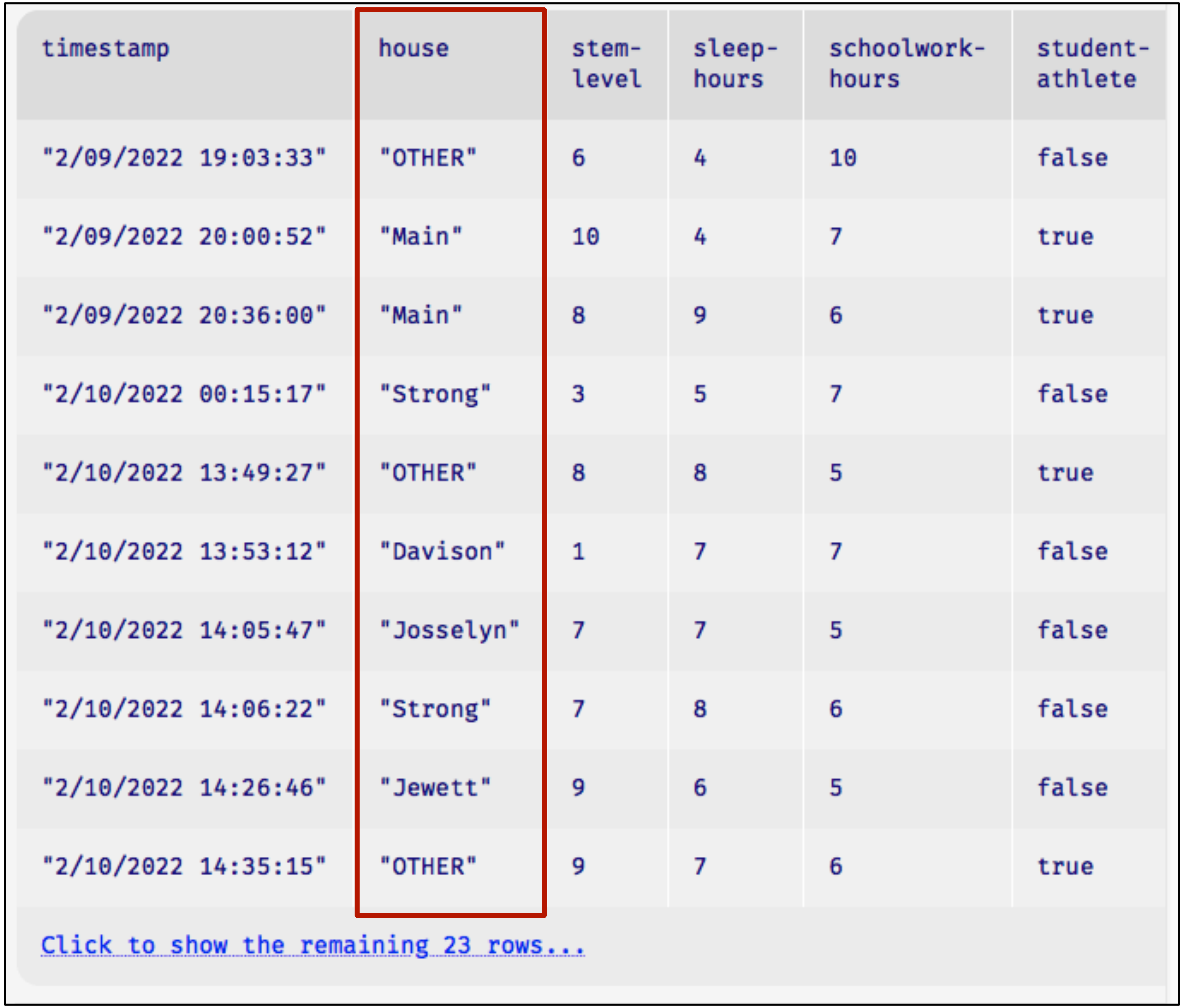
Introduction to Lists

3 October 2022

Where are we?

We’ve seen that when you want a row of a table,
you use **`.row-n`** and get a Row.

What about getting a column?



timestamp	house	stem-level	sleep-hours	schoolwork-hours	student-athlete
"2/09/2022 19:03:33"	"OTHER"	6	4	10	false
"2/09/2022 20:00:52"	"Main"	10	4	7	true
"2/09/2022 20:36:00"	"Main"	8	9	6	true
"2/10/2022 00:15:17"	"Strong"	3	5	7	false
"2/10/2022 13:49:27"	"OTHER"	8	8	5	true
"2/10/2022 13:53:12"	"Davison"	1	7	7	false
"2/10/2022 14:05:47"	"Josselyn"	7	7	5	false
"2/10/2022 14:06:22"	"Strong"	7	8	6	false
"2/10/2022 14:26:46"	"Jewett"	9	6	5	false
"2/10/2022 14:35:15"	"OTHER"	9	7	6	true

Click to show the remaining 23 rows...

```
››› student-data-cleaned.get-column("house")
[list: "OTHER", "Main", "Main", "Strong", ...]
```

When we've been working with tables we've been using the data type *Row*, but we never saw a *Column* data type!

Why not? Well, a column consists of an ordered collection of values, of unbounded length.

A column is really just a *List*!

Lists can be very convenient!

```
fun normalize-house(house :: String) -> String:
  doc: "Return one of the nine Vassar houses or 'Other'"
  if (house == "Main") or
    (house == "Strong") or
    (house == "Raymond") or
    (house == "Davison") or
    (house == "Lathrop") or
    (house == "Jewett") or
    (house == "Josselyn") or
    (house == "Cushing") or
    (house == "Noyes"):
    house
  else:
    "Other"
  end
where:
  normalize-house("Main") is "Main"
  normalize-house("Offcampus") is "Other"
end
```

😫

```
houses = [list: "Main", "Strong", "Raymond",
  "Davison", "Lathrop", "Jewett", "Josselyn",
  "Cushing", "Noyes"]


fun normalize-house(house :: String) -> String:
  doc: "Return one of the nine Vassar houses or
'Other'"
  if member(houses, house):
    house
  else:
    "Other"
  end
where:
  normalize-house("Main") is "Main"
  normalize-house("Offcampus") is "Other"
end
```

Mad Libs!

Plural-Noun

Plural-Noun

Plural-Noun *Number*

Plural-Noun

Noun

Noun *Noun* *Noun*

Body-Part

Alphabet-Letter

Plural-Noun *Plural-Noun*

Plural-Noun

Body-Part *Body-*

Part

Adjective *Noun*

Thousands of ago, there were calendars that enabled the ancient to divide a year into twelve , each month into weeks, and each week into seven . At first, people told time by a sun clock, sometimes known as the dial. Ultimately, they invented the great timekeeping devices of today, such as the grandfather , the pocket , the alarm , and, of course, the watch. Children learn about clocks and time almost before they learn their A-B- s. They are taught that a day consists of 24 , an hour has 60 , and a minute has 60 . By the time they are in Kindergarten, they know if the big is at twelve and the little is at three, that it is Number o'clock. I wish we could continue this lesson, but we've run out of .

How can we represent a text?

```
template = "Thousands of Plural–Noun ago, there were calendars that enabled the
ancient Plural–Noun to divide a year into twelve Plural–Noun , each month into
Number weeks, and each week into seven Plural–Noun . At first, people told time
by a sun clock, sometimes known as the Noun dial. Ultimately, they invented the
great timekeeping devices of today, such as the grandfather Noun , the pocket
Noun , the alarm Noun , and, of course, the Body–Part watch. Children learn
about clocks and time almost before they learn their A–B– Alphabet–Letter s.
They are taught that a day consists of 24 Plural–Noun , an hour has 60 Plural–
Noun , and a minute has 60 Plural–Noun . By the time they are in Kindergarten,
they know if the big Body–Part is at twelve and the little Body–Part is at
three, that it is Number o'clock. I wish we could continue this Adjective
lesson, but we’ve run out of Noun ."
```

```
template = "Thousands of Plural-Noun ago, …"

template-words = string-split-all(template, " ")

››› template-words
[list: "Thousands", "of", "Plural-Noun", "ago", ...]
```

```
template = "Thousands of Plural-Noun ago, …"

template-words = string-split-all(template, " ")

››› template-words
[list: "Thousands", "of", "Plural-Noun", "ago", ...]
```

We need to substitute a random plural noun here!

```
            "Thousands of Plural-Noun ago, ..."
                         |
                         |  string-split-all
                         v
[list: "Thousands", "of", "Plural-Noun", "ago", ...]
```

```
"Thousands of Plural-Noun ago, ..."
```

↓ `string-split-all`

```
[list: "Thousands", "of", "Plural-Noun", "ago", ...]
```

↓ *Something like* `transform-column` *but for lists*

```
[list: "Thousands", "of", "gazebos", "ago", ...]
```

$$\begin{tikzcd}
\texttt{"Thousands of Plural-Noun ago, ..."} \arrow[d, "\texttt{string-split-all}"] \\
\texttt{[list: "Thousands", "of", "Plural-Noun", "ago", ...]} \arrow[d, "\textit{Something like } \texttt{transform-column} \textit{ but for lists}"] \\
\texttt{[list: "Thousands", "of", "gazebos", "ago", ...]}
\end{tikzcd}$$

Needs a helper function!

`"Thousands of Plural-Noun ago, ..."`

↓ `string-split-all`

`[list: "Thousands", "of", "Plural-Noun", "ago", ...]`

↓ *Something like* `transform-column` *but for lists using*

`[list: "Thousands", "of", "gazebos", "ago", ...]`

`substitute-word`

`"Thousands" -> "Thousands"`

`"Plural-Noun" -> "gazebos"`

I’d write the helper function first!

```
fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  ...
where:
  substitute-word("Thousands") is "Thousands"
  substitute-word("Plural-Noun") is ...
end
```

Uh oh! We don't know what particular word it will be!

```
fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  ...
where:
  substitute-word("Thousands") is "Thousands"
  substitute-word("Plural-Noun") is-not "Plural-Noun"
end
```

We know what it isn't!

```
plural-nouns = [list: "gazebos", "avocados", "pandas"]

fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  ...
where:
  substitute-word("Thousands") is "Thousands"
  substitute-word("Plural-Noun") is-not "Plural-Noun"
  member(
    plural-nouns,
    substitute-word("Plural-Noun"))
    is true
end
```

And we know it’s one of the right choices!

```
plural-nouns = [list: "gazebos", "avocados", "pandas"]

fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  ...
where:
  substitute-word("Thousands") is "Thousands"
  substitute-word("Plural-Noun") is-not "Plural-Noun"
  member(
    plural-nouns,
    substitute-word("Plural-Noun"))
    is true
end
```

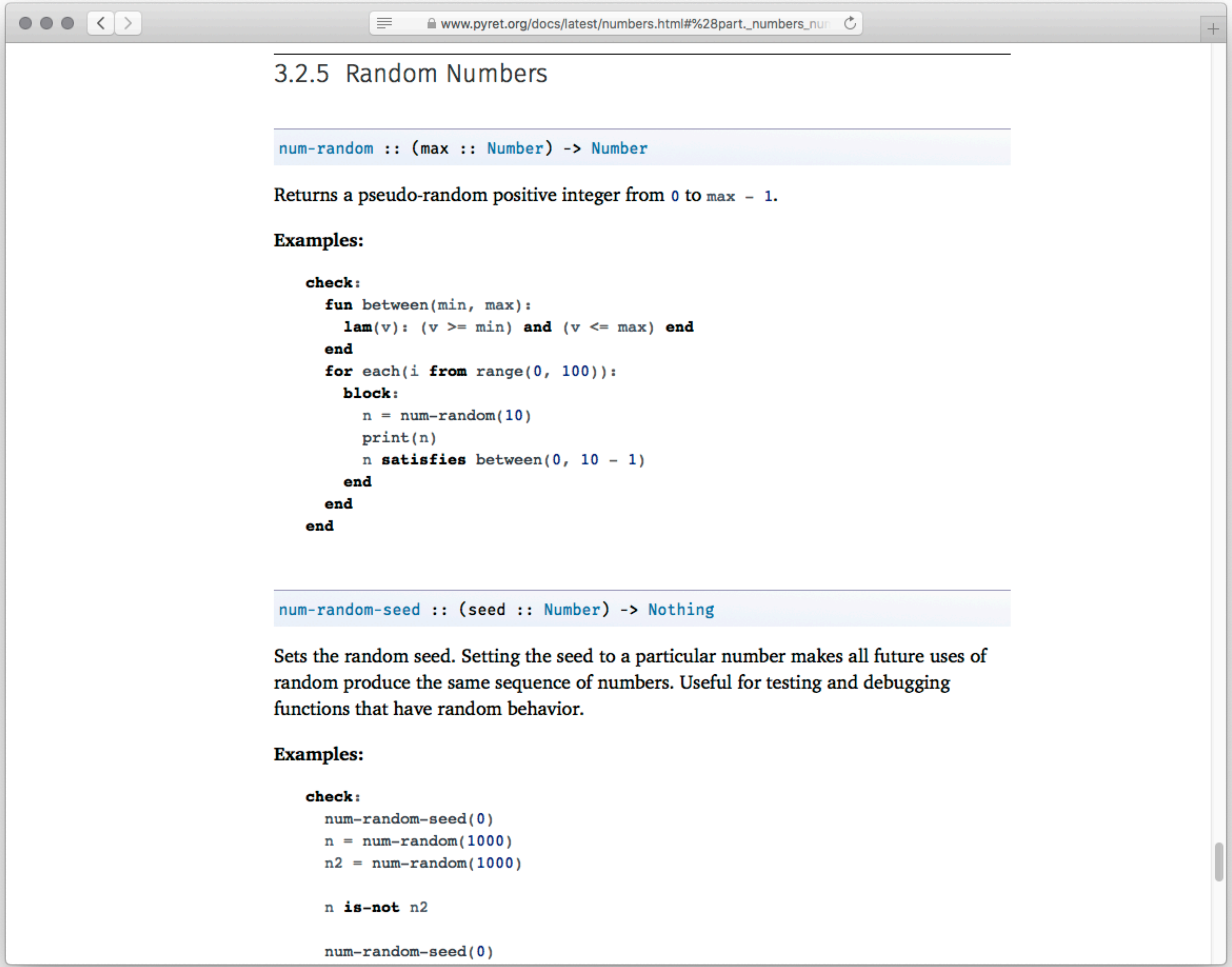
The left part of an example can be any expression!

```
plural-nouns = [list: "gazebos", "avocados", "pandas"]

fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  if w == "Plural-Noun":
    ...
  else:
    w
  end
where:
  ...
end
```

We need a random element of a list.

Time to check the Pyret documentation!



3.2.5 Random Numbers

```
num-random :: (max :: Number) -> Number
```

Returns a pseudo-random positive integer from `0` to `max - 1`.

Examples:

```
check:
  fun between(min, max):
    lam(v): (v >= min) and (v <= max) end
  end
  for each(i from range(0, 100)):
    block:
      n = num-random(10)
      print(n)
      n satisfies between(0, 10 - 1)
    end
  end
end
```

```
num-random-seed :: (seed :: Number) -> Nothing
```

Sets the random seed. Setting the seed to a particular number makes all future uses of random produce the same sequence of numbers. Useful for testing and debugging functions that have random behavior.

Examples:

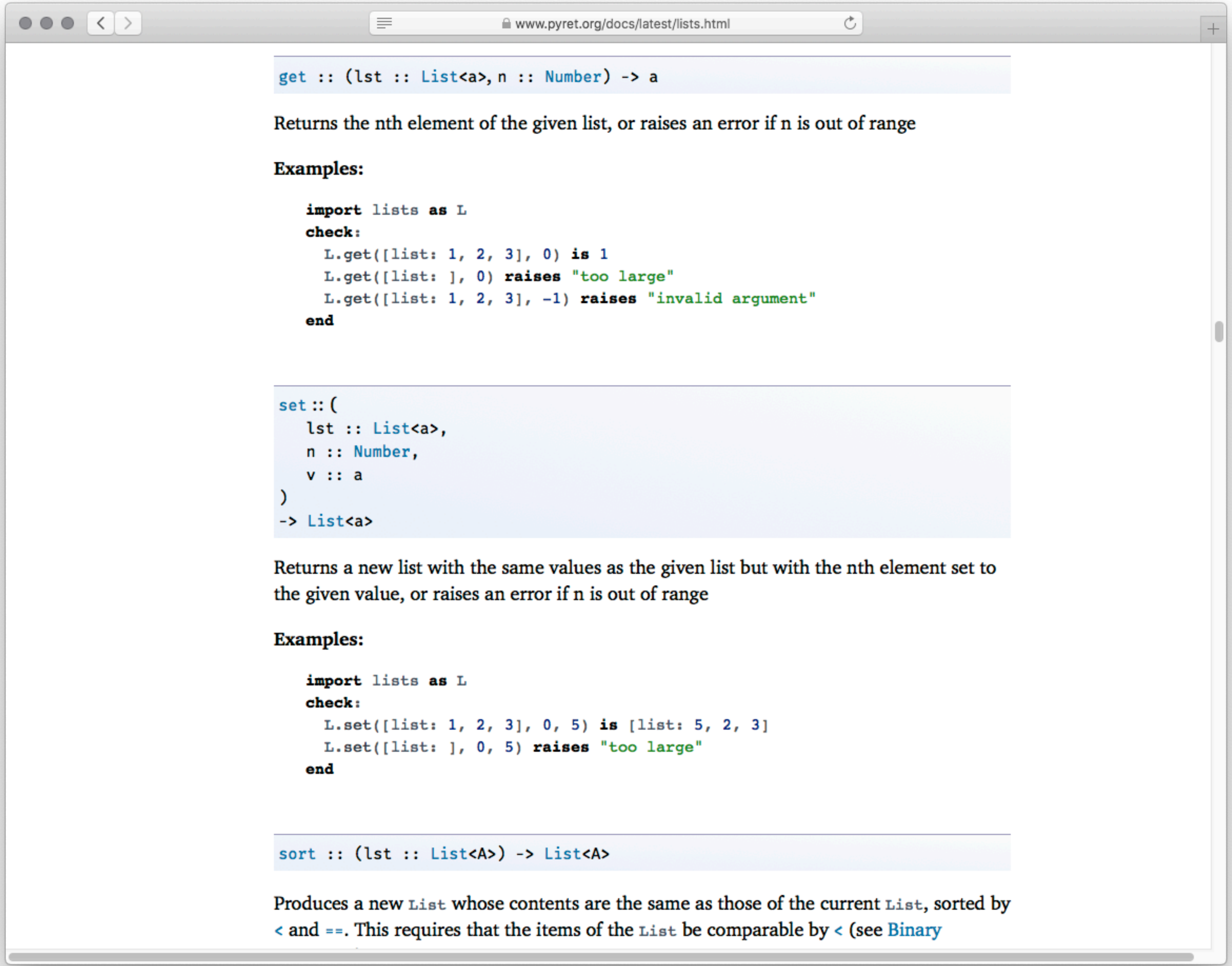
```
check:
  num-random-seed(0)
  n = num-random(1000)
  n2 = num-random(1000)

  n is-not n2

  num-random-seed(0)
```

We didn’t find a built-in way to get a random element of a list, but we found a way to get a random number.

How could we use this?



```
get :: (lst :: List<a>, n :: Number) -> a
```

Returns the nth element of the given list, or raises an error if n is out of range

Examples:

```
import lists as L
check:
  L.get([list: 1, 2, 3], 0) is 1
  L.get([list: ], 0) raises "too large"
  L.get([list: 1, 2, 3], -1) raises "invalid argument"
end
```

```
set :: (
   lst :: List<a>,
   n :: Number,
   v :: a
)
-> List<a>
```

Returns a new list with the same values as the given list but with the nth element set to the given value, or raises an error if n is out of range

Examples:

```
import lists as L
check:
  L.set([list: 1, 2, 3], 0, 5) is [list: 5, 2, 3]
  L.set([list: ], 0, 5) raises "too large"
end
```

```
sort :: (lst :: List<A>) -> List<A>
```

Produces a new `List` whose contents are the same as those of the current `List`, sorted by `<` and `==`. This requires that the items of the `List` be comparable by `<` (see Binary

With a table, we could use **`.row-n`** to get a specific row by its index number.

With a list, we can use **`get(List, Number)`** to get an item.

Get random number

Get list element positioned at that number

```
plural-nouns = [list: "gazebos", "avocados", "pandas"]


fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  if w == "Plural-Noun":
    rand = num-random(3)     # ugh
    get(plural-nouns, rand)
  else:
    w
  end
where:
  ...
end
```

```
plural-nouns = [list: "gazebos", "avocados", "pandas"]


fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  if w == "Plural-Noun":
    rand = num-random(3)
    get(plural-nouns, rand)
  else:
    w
  end
where:
  ...
end
```

```
plural-nouns = [list: "gazebos", "avocados", "pandas",
"quokkas"]

fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  if w == "Plural-Noun":
    rand = num-random(3)
    get(plural-nouns, rand)
  else:
    w
  end
where:
  ...
end
```

```
plural-nouns = [list: "gazebos", "avocados", "pandas",
"quokkas"]

fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  if w == "Plural-Noun":
    rand = num-random(length(plural-nouns))
    get(plural-nouns, rand)
  else:
    w
  end
where:
  ...
end
```

```
template = "Thousands of Plural–Noun ago, there were calendars that enabled the
ancient Plural–Noun to divide a year into twelve Plural–Noun , each month into
Number weeks, and each week into seven Plural–Noun . At first, people told time
by a sun clock, sometimes known as the Noun dial. Ultimately, they invented the
great timekeeping devices of today, such as the grandfather Noun, the pocket
Noun , the alarm Noun , and, of course, the Body–Part watch. Children learn
about clocks and time almost before they learn their A–B– Alphabet–Letter s.
They are taught that a day consists of 24 Plural–Noun , an hour has 60 Plural–
Noun , and a minute has 60 Plural–Noun . By the time they are in Kindergarten,
they know if the big Body–Part is at twelve and the little Body–Part is at
three, that it is Number o'clock. I wish we could continue this Adjective
lesson, but we’ve run out of Noun ."
```

```
plural-nouns =
  [list: "gazebos", "avocados", "pandas", "quokkas"]

numbers =
  [list: "-1", "42", "a billion"]

nouns =
  [list: "apple", "computer", "borscht"]

body-parts =
  [list: "elbow", "head", "spleen"]

alphabet-letters =
  [list: "A", "C", "Z"]

adjectives =
  [list: "funky", "boring"]
```

```
fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  if w == "Plural-Noun":
    rand = num-random(length(plural-nouns))
    get(plural-nouns, rand)
  else if w == "Number":
    rand = ...
  else:
    w
  end
where:
  ...
end
```

```
fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  if w == "Plural-Noun":
    rand = num-random(length(plural-nouns))
    get(plural-nouns, rand)
  else if w == "Number":
    rand = ...
  else:
    w
  end
where:
  ...
end
```

Don't repeat yourself!

```
fun rand-word(l :: List<String>) -> String:
  doc: "Return a random word in the given list"
  rand = num-random(length(l))
  get(l, rand)
where:
  member(plural-nouns, rand-word(plural-nouns))
    is true
end
```

This wasn't on our task plan, but we saw a need for a general utility function, so we wrote it!

```
fun substitute-word(w :: String) -> String:
  doc: "Substitute a random word if w is a category"
  if w == "Plural-Noun":
    rand-word(plural-nouns)
  else if w == "Number":
    rand-word(numbers)
  else if w == "Noun":
    rand-word(nouns)
  else if w == "Body-Part":
    rand-word(body-parts)
  else if w == "Alphabet-Letter":
    rand-word(alphabet-letters)
  else if w == "Adjective":
    rand-word(adjectives)
  else:
    w
  end
end
```

Go back to the task plan.

We’ve completed our helper, and now we need to run it on every word in the list, like **`transform-column`** runs a function on every row of a table.

The way to do that is called **`map`**.

```
fun mad-libs(t :: List<String>) -> String:
  doc: "Randomly fill in the blanks in the mad libs
template"
  map(substitute-word, t)
end
```

This gives us a list of strings. How can we join it back into a single string?

```
fun mad-libs(t :: List<String>) -> String:
  doc: "Randomly fill in the blanks in the mad libs
template"
  with-subs = map(substitute-word, t)
  join-str(with-subs, " ")
end
```

```
fun mad-libs(t :: List<String>) -> String:
  doc: "Randomly fill in the blanks in the mad libs
template"
  with-subs = map(substitute-word, t)
  join-str(with-subs, " ")
where:
  ...
end
```

Preview: Lists and recursion

What if **`join-str`** didn't already exist for our convenience?

To write a function that processes a list element by element, we need to understand the real nature of lists.

A list consists of two parts: a **`first`** element and the **`rest`** of the list.

```
››› l = [list: 1, 2, 3]
››› l.first
1
››› l.rest
[list: 2, 3]
```

The first element is linked to the rest and so on until we reach the empty list:

```
>>> link(1, empty)
[list: 1]
>>> link(1, link(2, link(3, empty)))
[list: 1, 2, 3]
```

When we write a function that recursively processes a list, we deal with these two cases – linking an element of being empty:

```
fun add-nums(l :: List<Number>) -> Number:
  cases (List) l:
    | empty => 0
    | link(f, r) => f + add-nums(r)
  end
end
```

In the case of joining strings, we need to know not just if the current list is empty but is the rest of the rest empty. This is how we know whether to add a space or not.

```
fun join-with-spaces(l :: List<String>) -> String:
  doc: "Join the strings in l with a space between
each one"
  cases (List) l:
    | empty => ""
    | link(f, r) =>
      cases (List) r:
        | empty => f
        | link(fr, rr) =>
          f + " " + join-with-spaces(r)
      end
  end
where:
  join-with-spaces([list: ]) is ""
  join-with-spaces([list: "y"]) is "y" + ""
  join-with-spaces([list: "x", "y"]) is
    "x" + " " + join-with-spaces([list: "y"])
end
```

Class code:

`https://code.pyret.org/editor#share=1gNCCr9cAxOFqewY3Wx221gSqV-JQho5n&v=31c9aaf`