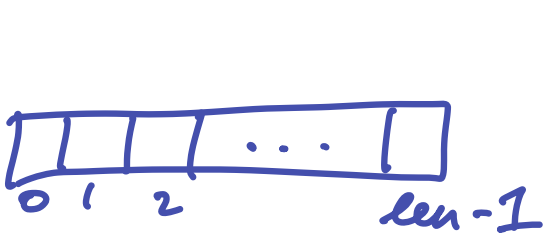
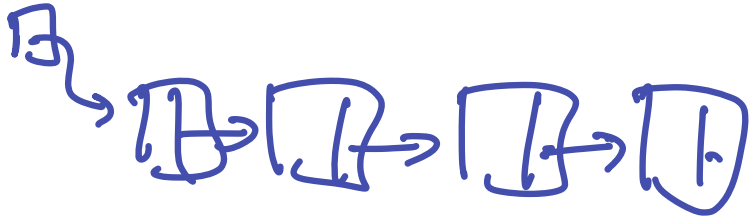
Recursion

CSC148, INTRODUCTION TO COMPUTER SCIENCE

DIANE HORTON, JONATHAN CALVER & SADIA SHARMIN





Data structure informs code structure

```
i = 0
while i < len(lst):
    ... lst[i] ...
    i += 1
```

```
curr = lst._first
while curr is not None:
    ... curr.item ...
    curr = curr.next
```

Data structure informs code structure

```
List[int]
```

```
for x in lst:
    ... x ...
```

```
List[
  List[int]
]
```

```
for lst in lst_of_lsts:
    for x in lst:
        ... x ...
```

Data structure informs code structure

```
List[
  List[
    List[int]
  ]
]
```

```
for lst_of_lsts in lst_of_lsts_of_lsts:
    for lst in lst_of_lsts:
        for x in lst:
            ... x ...
```

Data structure informs code structure

A nested list is...

- An integer
- A list of nested lists

```
def nested_f(obj):

  if isinstance(obj, int):
    ...
  else:
    for sublist in obj:
      … nested_f(sublist) …
```

Partial tracing practice

Attempting to fully trace recursive code is time-consuming and error prone. And it's unnecessary!

*When tracing recursive code, **don't** trace into recursive calls!*

Instead, assume each call is correct, and make sure the rest of the code uses those calls correctly.

`first_at_depth` – base case

A single integer is always at depth 0.

```
>>> first_at_depth(100, 0)
100
>>> first_at_depth(100, 3) is None
True
```

`first_at_depth` – recursive case

```
>>> first_at_depth([10, [[20]], [30, 40]], 2)
30
```

sublist	depth	first_at_depth(sublist, depth)
10		
[[20]]		
[30, 40]		

`first_at_depth` – multiple base cases!

```
first_at_depth(obj, d)
    -> first_at_depth(sublist, d - 1)
```

We are actually recursing on both `obj` and `d`.

Can't recurse when:

- `isinstance(obj, int)`
- `d == 0`

Tips for writing recursive functions

Think lazy.

- What smaller instance(s) of the same problem can you ask someone to solve for you?
- When the problem is so small that even lazy you can do it, write the code directly.

Mind your own business.

- Don’t concern yourself with how a recursive call works!
- Or with what the caller is going to do with your result.

Nested list mutation

Last worksheet on nested lists!

Also really good review for a classic memory-related error.

A common error: missing `return`

```
def nested_list_contains(obj, item) -> bool:
    if isinstance(obj, int):
        return obj == item
    else:
        for sublist in obj:
            if nested_list_contains(sublist, item):
                nested_list_contains(sublist, item)
        return False
```

A common error: missing `return`

```
def nested_list_contains(obj, item) -> bool:
    if isinstance(obj, int):
        return obj == item
    else:
        for sublist in obj:
            if nested_list_contains(sublist, item):
                nested_list_contains(sublist, item)
        return False
```

A common error: missing `return`

```
def nested_list_contains(obj, item) -> bool:
    if isinstance(obj, int):
        return obj == item
    else:
        for sublist in obj:
            if nested_list_contains(sublist, item):
                return True
        return False
```

A common error: missing `return`

A `return` statement exits from *one function call*.

When writing a recursive function that should return something, both the base case and recursive step must have a `return`!

More generally, if a function returns something, then every execution path through the function must have a `return`.