CSC148-Section:L0301
Week#6-Monday

Instructed by
AbdulAziz Al-Helali
a.alhelali@mail.utoronto.ca
Office hours: Wednesday 11-1, BA2230.

Slides adapted from Professor Danny Heap course material
winter17
Outline

• Recursion
  • Quick review
  • Template
  • Implement recursive functions:
    • max_depth/rec_max/concat_strings/flatten/nested_contains of nested lists
Recursion

• A method calls itself

```python
def factorial(num: int) -> int:
    if num == 1:
        return 1
    else:
        return num * factorial(num-1)
```

Base case

General case
Recursion

• A Recursive Function has a **conditional structure** that consists of:
  • Base case
    • species when/how to stop – has no recursion (does not call the function)
  • General case
    • species **how to combine recursive subcalls**
Template for structural recursion

recursion when input is a recursive structure:

• Base case
  • if input cannot be decomposed into recursive sub-structures, you have a base case and you directly return a result without recursion

• General case
  • if input can be decomposed into recursive sub-structures, solve them recursively and combine the result(s)
Template for structural recursion

This reduces your job to figuring out:

(a) how to detect whether the input can be decomposed or not.

(b) how what result to return for the base case, and

(c) which substructures to solve recursively and how to combine their solutions
Depth of list: \texttt{max\_depth}

What is the depth of this list?

\[ [14, 7, [5, [3, [1,2,[2],3], 1], [2,[6,8] ,4], 6], 9 , 10 , 11 , [12],[[]] ] \]
What is the depth of this list?

\[
[14, 7, [5, [3, [1,2,2,3], 1], [2,[6,8,4], 6], 9 , 10 , 11 , \ [12],[]]]
\]
The depth of list is 1 plus the maximum depth of list's elements if list is a list, otherwise 0.
What is the depth of this list?

• The depth of list is 1 plus the maximum depth of list 's elements if list is a list, otherwise 0

• The definition is almost exactly the Python code you write!
What is the depth of this list?

• The depth of list is 1 plus the maximum depth of list 's elements if list is a list, otherwise 0

• The definition is almost exactly the Python code you write!

```python
if not isinstance(obj, list):  # not list - part of Base case
    return 0
elif obj == []:  # empty list - part of Base case
    return 1
else:  # General case
    return 1 + max([max_depth(x) for x in obj])
```
Trace: max_depth

• Trace in increasing complexity;
• Trace max_depth(17)
• Trace max_depth([])
• Trace max_depth([3, 17, 1])
• Trace max_depth([5, [3, 17, 1], [2, 4], 6])
• Trace max_depth([14, 7, [5, [3, 17, 1], [2, 4], 6], 9])

\[
\text{if not isinstance(obj, list):} \\
\hspace{1cm} \text{return 0} \\
\text{elif obj == []:} \\
\hspace{1cm} \text{return 1} \\
\text{else:} \\
\hspace{1cm} \text{return 1 + max([max_depth(x) for x in obj])}
\]

at each step fill in values for recursive calls that have (basically) already been traced
Trace: max_depth

• Trace max_depth(17)
  -> 0

• Trace max_depth("")
  -> 1

• Trace max_depth([3, 17, 1])
  -> 1 + max([max_depth(3), max_depth(17), max_depth(1)])
  -> 1 + max([0, 0, 0])
  -> 1

if not isinstance(obj, list):
    return 0
elif obj == []:
    return 1
else:
    return 1 + max([max_depth(x) for x in obj])
Trace: `max_depth`

- Trace `max_depth([5, [3, 17, 1], [2, 4], 6])`
  -> 1 + max([max_depth(5), max_depth([3, 17, 1]), max_depth([2, 4]), max_depth(6)])
  -> 1 + max([0, 1, 1, 0])
  -> 1 + 1
  -> 2

When tracing on paper, at each step fill in values for recursive calls that have (basically) **already been traced**

```python
if not isinstance(obj, list):
    return 0
elif obj == []:
    return 1
else:
    return 1 + max([max_depth(x) for x in obj])
```
• Trace $\text{max\_depth}([14, 7, 5, [3, 17, 1], [2, 4], 6], 9)$

$\rightarrow 1 + \max([\text{max\_depth}(14), \text{max\_depth}(7), \text{max\_depth}([5, [3, 17, 1], [2, 4], 6]), \text{max\_depth}(9)])$

$\rightarrow 1 + \max([0, 0, 2, 0])$

$\rightarrow 1 + 2$

$\rightarrow 3$
def max_depth(obj: object) -> int:
    """
    Return 1 + the maximum depth of obj’s elements if obj is a list. Otherwise return 0.
    >>> max_depth(17)
    0
    >>> max_depth([])
    1
    >>> max_depth([1, "two", 3])
    1
    >>> max_depth([1, ["two", 3], 4])
    2
    >>> max_depth([1, [2, ["three", 4], 5], 6])
    3
    """
    if obj == []:
        return 1
    elif isinstance(obj, list):
        return 1 + max([max_depth(x) for x in obj])
    else:
        return 0
Maximum number in nested list: \texttt{rec\_max}

What is the max number in this list?

\[ [14, 7, [5, [3, [1,2,[2],3], 1], [2,[6,8] ,4], 6], 9 , 10 , 11 , [12] ] \]
Maximum number in nested list
What is the max number in this list?

\[14, 7, [5, [3, [1,2,[2],3], 1], [2,[6,8],4], 6], 9, 10, 11, [12] \]
Maximum number in nested list: \texttt{rec\_max}

- how would you find the \texttt{max} of non-nested list?
  \texttt{max(...)}

- how would you \texttt{build that list} using a comprehension?
  \texttt{max([x for x in list])}

- what should you do with list items that \texttt{were themselves lists}?
  \texttt{max([rec\_max(x) for x in list_])}

- get some intuition by tracing through:
  - flat lists,
  - lists nested one deep,
  - then two deep...
Maximum number in nested list: \texttt{rec\_max}

```python
if isinstance(list_, list):
    return max([rec_max(x) for x in list_])
else:
    return list_
```
Trace: \texttt{rec\_max}

• Trace in increasing complexity;

• Trace \texttt{rec\_max}([3, 5, 1, 3, 4, 7])

• Trace \texttt{rec\_max}([4, 2, [3, 5, 1, 3, 4, 7], 8])

• Trace \texttt{rec\_max}([6, [4, 2, [3, 5, 1, 3, 4, 7], 8], 5])

\begin{code}
\begin{verbatim}
if isinstance(list_, list):
    return max([rec_max(x) for x in list_])
else:
    return list_
\end{verbatim}
\end{code}

at each step fill in values for recursive calls that have (basically) already been traced
Trace: rec_max

• Trace in increasing complexity;

• Trace rec_max([3, 5, 1, 3, 4, 7])
  -> max([rec_max(3), rec_max(5), rec_max(1), rec_max(3), rec_max(4), rec_max(7)])
  -> max([3, 5, 1, 3, 4, 7])
  -> 7

• Trace rec_max([4, 2, [3, 5, 1, 3, 4, 7], 8])
  -> max([rec_max(4), rec_max(2), rec_max([3, 5, 1, 3, 4, 7]), rec_max(8)])
  -> max([4, 2, 7, 8])
  -> 8

if isinstance(list_, list):
    return max([rec_max(x) for x in list_])
else:
    return list_

at each step fill in values for recursive calls that have (basically) already been traced

When tracing on paper, at each step fill in values for recursive calls that have (basically) already been traced
Trace: \texttt{rec\_max}

- Trace in increasing complexity;

- Trace \texttt{rec\_max([6, [4, 2, [3, 5, 1, 3, 4, 7], 8], 5])}
  \[
  \rightarrow \max([\texttt{rec\_max}(6), \texttt{rec\_max}([4, 2, [3, 5, 1, 3, 4, 7], 8]), \texttt{rec\_max}(5)])
  \rightarrow \max([6, 8, 5])
  \rightarrow 8
  \]

\begin{Verbatim}
if \textbf{isinstance}(\texttt{list\_}, \texttt{list}):
    \textbf{return} \max([\texttt{rec\_max}(x) \textbf{for} x \textbf{in} \texttt{list}\_])
\textbf{else}:
    \textbf{return} \texttt{list\_}
\end{Verbatim}
```python
def rec_max(list_: Union[List, int]) -> int:
    """
    Return the maximum int in nested lists
    lists and sublists must Not be empty or None
    >>> rec_max(17)
    17
    >>> rec_max([-1, 0])
    0
    >>> rec_max([1, 2, 3])
    3
    >>> rec_max([1, [5, 3], 4])
    5
    >>> rec_max([1, [2, [9, 4], 5], 6])
    9
    """
    if isinstance(list_, list):
        return max([rec_max(x) for x in list_])
    else:
        return list_
```

Concatenating strings in nested list:

```
concat_strings

["one", ["two", ["three", ["four", "five"], "six"]]]
```
Concatenating strings in nested list:

```python
concat_strings = ['one', ['two', ['three', ['four', 'five'], 'six']]]
```

```
["one", ]

["two", ]

["three", , "six"]

["four", "five"]
```
Concatenating strings in nested list:

```python
if isinstance(string_list, list):  # General Case
    return ''.join([concat_strings(x) for x in string_list])
else:  # Base Case
    return string_list
```
Trace: `concat_strings`

- Trace in increasing complexity;
- Trace `concat_strings(["now", "brown"])`
  
  -> `"".join([concat_strings("now"), concat_strings("brown")])`

- Trace `concat_strings(["how"])`

- Trace `concat_strings([])`

```python
if isinstance(string_list, list):  # General Case
    return "".join([concat_strings(x) for x in string_list])
else:  # Base Case
    return string_list
```

at each step fill in values for recursive calls that have (basically) already been traced
• Trace in increasing complexity;

• Trace `concat_strings(["how", ["now", "brown"], "cow"])

  -> ".join([concat_strings("now"), concat_strings("brown")])

• Trace `concat_strings(["how", ["now", ["brown", "cow"]]])

• Trace `concat_strings(["one" ["two", ["three", ["four", "five"], "six"]]])
def concat_strings(string_list):
    ""
    Concatenate all the strings in possibly-nested string_list.
    @param list[|str|] str string_list:
    @rtype: str
    >>> list_ = ["how", ["now", "brown1"], "cow1"]
    >>> concat_strings(list_)
    'hownowbrown1cow1'
    ""
    return ''.join([concat_strings(x) if isinstance(x, list) else x
                     for x in string_list])

    # if isinstance(string_list,list):
    #     return ''.join([concat_strings(x) for x in string_list])
    # else:
    #     return string_list
def flatten(list_: List) -> List:
    """
    Return a flattened list of nested lists
    
    >>> flatten([1, [5, 3], 4])
    [1, 5, 3, 4]
    """
def flatten(list_: List) -> List:
    ""
    Return a flattened list of nested lists
    
    >>> flatten([1, [5, 3], 4])
    [1, 5, 3, 4]
    ""
    if isinstance(list_, list):
        return sum([flatten(x) for x in list_], [])
    else:
        return [list_]
def flatten(list_: List) -> List:
    
    """
    Return a flattened list of nested lists
    """

    return sum([flatten(x) if isinstance(x, list) else [x] for x in list_], [])

using functional if
functional if

<expression 1> if <condition> else <expression 2>

- expression 1 is evaluated if condition is True
- expression 2 is evaluated if condition is False
```python
def flatten(list_: List) -> List:
    """
    Return a flattened list of nested lists
    """
    >>> flatten([1, [5, 3], 4])
    [1, 5, 3, 4]
    """
    # return sum([flatten(x) if isinstance(x,list)
    #             else [x]
    #             for x in list_],[])
    if isinstance(list_, list):
        return sum([[flatten(x) for x in list_],[]])
    else:
        return [list_]
```
def nested_contains(list_: list, value: object) -> bool:
    """Return whether list_, or any nested sub-list of list_ contains value.
    >>> list_ = ["how", ["now", "brown"], 1]
    >>> nested_contains(list_, "now")
    True
    >>> nested_contains([], 5)
    False
    >>> nested_contains([5], 5)
    True
    """
    # check out Python built-in any
def nested_contains(list_: list, value: object) -> bool:

    if isinstance(list_, list):
        return any([nested_contains(x, value) for x in list_])
    else:
        return value == list_
```python
def nested_contains(list_: list, value: object) -> bool:
    """
    Return whether list_, or any nested sub-list of list_ contains value.
    >>> list_ = ["how", "now", "brown", 1]
    >>> nested_contains(list_, 1)
    True
    >>> nested_contains(list_, 3)
    False
    >>> list_ = ["how", ["now", "brown"], 1]
    >>> nested_contains(list_, "now")
    True
    >>> nested_contains([], 5)
    False
    >>> nested_contains([5], 5)
    True
    """
    # check out Python built-in any
    if isinstance(list_, list):
        return any([nested_contains(x, value) for x in list_])
    else:
        return value == list_
```