CSC148 winter 2017

generative recursion, Exceptions, functional programming week 5

Danny Heap
heap@cs.toronto.edu / BA4270 (behind elevators)
http://www.cdf.toronto.edu/~csc148h/winter/
416-978-5899

February 5, 2017





Outline

a1 topics

idiomatic python

getting that recursive insight for Tower of Hanoi

In order to implement a function that moves n cheeses from, say, stool 1 to stool 3, we'd first think of a name and parameters. We can start with move_cheeses(n, source, dest), meaning show the moves from source stool to destination stool for n cheeses.

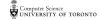
stating that recursive insight:

The doodling on the previous slide breaks into a pattern, at least for the 2- and 3-cheese case:

- ▶ move all but the bottom cheese from source to intermediate stool (sounds recursive...)
- ▶ move the bottom cheese from the source to the destination stool (sounds like the 1-cheese move)
- ▶ move all but the bottom cheese from the intermediate to the destination stool (sounds recursive...)

The original problem repeats, except with different source, destination, and intermediate stools!

New name: move cheeses(n, source, intermediate, destination)





write some code!

fill in the three steps from the previous slide

```
"""Print moves to get n cheeses from source to destination.
@param int n:
Oparam int source
Oparam int intermediate:
Oparam int destination:
Ortype: None
.. .. ..
if n > 1: # fill this in!
   move_cheeses( ?, ?, ?,
   move_cheeses( ?, ?, ?, ?)
   move_cheeses( ?, ?. ?.
else: # just 1 cheese --- leave this out for now!
```

def move_cheeses(n, source, intermediate, destination) -> None:



complete that code!

Now, fill in what you do to move just one cheese — don't use any recursion! You will be returning a string that specifies you are moving from source to destination.

```
def move_cheeses(n, source, intermediate, destination) -> None:
    """Print moves to get n cheeses from source to destination.
```

```
@param int n:
Oparam int source
@param int intermediate:
Oparam int destination:
Ortype: None
.. .. ..
if n > 1: # fill this in!
   move_cheeses(
                    ?, ?, ?,
   move_cheeses( ?, ?,
   move_cheeses(?,?,
                                         ?)
else: # just 1 cheese --- leave this out for now!
   print(???)
```



python gratification

Once you have your code entered into some Python environment, you should run it with a few small values of n. As usual, you can get more intuition about it by tracing examples, working from small to larger n

richer communication

return types are not appropriate in all cases

what's wrong with Stack returning a "special" integer for remove when empty?

▶ add usually has return type None, but what if stuff happens?

▶ what if the calling code doesn't know what to do?



cause existing Exceptions:

▶ int("seven")

= 1/0

▶ [1, 2][2]

raise existing Exceptions:

▶ raise ValueError or...

raise ValueError("you can't do that!")

roll your own Exceptions:

Lass ExtremeException(Exception):
pass

raise ExtremeException

raise ExtremeException('I really take exception
to that!')



going with the (pep) tide

Python is more flexible than the community you are coding in. Try to figure out what the python way is

- don't re-invent the wheel (except for academic exercises), e.g. sum, set
- ▶ use comprehensions when you mean to produce a new list (tuple, dictionary, set, ...)
- ▶ use ternary if when you want an expression that evalutes in different ways, depending on a condition



example: add (squares of) first 10 natural numbers

➤ You'll be generating a new list from range(1, 11), so use a comprehension

➤ You want to add all the numbers in the resulting list, so use sum

list differences, lists without duplicates

▶ python lists allow duplicates, python sets don't

python sets have a set-difference operator

python built-in functions list() and set() convert types



valid sudoku

what makes a sudoku square valid?

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	ო	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	80	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

- valid rows
- valid columns
- valid subsquares





code it!

```
def valid_sudoku(grid, digit_set):
.. .. ..
Return whether grid represents a valid, complete sudoku.
@type grid: list[list]
Otype digit_set: set
@rtype: bool
.. .. ..
assert all([len(r) == len(grid) for r in grid])
assert len(grid) == len(digit_set)
return (_all_rows_valid(grid, digit_set) and
        _all_columns_valid(grid, digit_set) and
        _all_subsquares_valid(grid, digit_set))
```



code those non-existent helpers!

```
def _all_rows_valid(grid, digit_set):
    11 11 11
    Return whether all rows in grid are valid and complete.
    @type grid: list[list]
    Otype digit_set: set
    @rtype: bool
    Assume grid has same number of rows as elements of digit_set
    and grid has same number of columns as rows.
    11 11 11
    assert all([len(r) == len(grid) for r in grid])
    assert len(grid) == len(digit_set)
    return all([_list_valid(r, digit_set) for r in grid])
```

code the helpers' helpers...

```
def _list_valid(r, digit_set):
    11 11 11
    Return whether r contains each element of digit_set
    exactly once.
    Otype r: list
    Otype digit_set: set
    Ortype: bool
    Assume r has same number of elements as digit_set.
    11 11 11
    assert len(r) == len(digit_set)
    return set(r) == digit_set
```

