CSC148: Week 6
http://www.cdf.utoronto.ca/~csc148h/summer/

Sophia Huynh
Summer 2018
Announcements

● Lab 5 and Exercise 5 are out
● Assignment 1 due Sunday @ 11PM
  ○ Office Hours: Friday, 2 - 4PM in BA3201
● Midterm 1 marks released
  ○ Working on getting tests scanned
● No classes next week (or the week after) -- resumes July 5th
Outline

● Idiomatic Python
● Recursion
Idiomatic Python
Idiomatic Python

- Code should be **readable, clean, easy to understand**
- Re-use functions instead of re-implementing everything
  - Unless it's for practice
Summing Numbers

● How do we get the sum of [1, 4, 2, 7, 3]?
Summing Numbers

• How do we get the sum of [1, 4, 2, 7, 3]?

count = 0
for i in lst:
    count += i
Summing Numbers

- How do we get the sum of [1, 4, 2, 7, 3]?

  \[ \text{sum}(\text{lst}) \]

- Python has a built-in function for this! Use it!
Adding Strings

● How do we add the strings ['pi', "e", "ce", "s"]?

sum(lst, "")

● sum can be given a starting value -- for a string, we want to start with "" instead of 0.

Removed.
I was using an outdated version of Python :( 
Adding Lists

● How do we add the lists `[[1, 2], [3], [4, 5]]`?

```
sum(lst, [])
```

● Starting value for adding lists together: an empty list
Don't reinvent the wheel!

- If Python has a built-in we can use, just use it!
How to tell if there's at least one True in a list?

[False, True, True, False]

Use `any(lst)`
all()

- How to tell if all items in a list are True?
  
  ```python
  [True, True, False, True]
  ```

- Use `all(lst)`
Use Python's built-in functions!

- `sum()`
- `any()`
- `all()`
- And many, many more.
Ternary ifs

● Condense assignment-based if-statements

```python
if a:
    x = "Something"
else:
    x = "Something else"
```
Ternary ifs

- Simplify it to

\[
x = "Something" \text{ if } a \text{ else } "Something else"
\]
Ternary ifs

```python
if a:
    x = "Something"
else:
    x = "Something else"
```

... is equivalent to...

```python
x = "Something" if a else "Something else"
```
Ternary ifs

● Can also condense return statements

```python
if x == 10:
    return 100
return 0
```
Ternary ifs

- Simplify it to

```
return 100 if x == 10 else 0
```
Ternary ifs

```python
return 100 if x == 10 else 0
```

- Evaluate the if-statement
- Then `return` it
- else is necessary here -- no elifs.
  - Can chain the ternary ifs, but it'll look ugly.
List Comprehensions

● Build a list using a for-loop

```python
lst = []
for i in range(20):
    lst.append(i ** 2)
```
List Comprehensions

- Empty a Stack with a for-loop:

```
lst = []
for i in range(len(self._content)):
    lst.append(self.remove())
```
List Comprehensions

```python
lst = []
for i in range(20):
    lst.append(i ** 2)
... is equivalent to...

lst = [i ** 2 for i in range(20)]
```
List Comprehensions

```python
lst = []
for i in range(len(self._content)):
    lst.append(self.remove())

... is equivalent to...

lst = [self.remove() for i in range(len(self._content))]
```
List Comprehensions

\[ i ** 2 \text{ for } i \text{ in range}(20) \]

- Concise, readable.
- You don't \textit{have to} use it.
  - Just understand how to read it.
- Mostly personal preference.
List Comprehensions with if/else

- Combing list comprehensions with if:

```python
lst = [i ** 2 for i in range(20) if i % 2 == 0]
```

EDITTED: The if comes after 'for i in range' if there's no else. If there's an else, then we format it like ternary if.
List Comprehensions with if/else

```python
lst = []
for i in range(20):
    if i % 2 == 0:
        lst.append(i ** 2)
... is equivalent to...

dlst = [i ** 2 for i in range(20) if i % 2 == 0]
```

EDITTED: The if comes after 'for i in range' if there's no else. If there's an else, then we format it like ternary if.
List Comprehensions with if/else

- Can also add in an 'else' statement

```python
lst = [i ** 2 if i % 2 == 0
      else i for i in range(20)]
```
List Comprehensions with if/else

```python
lst = []
for i in range(20):
    if i % 2 == 0:
        lst.append(i ** 2)
    else:
        lst.append(i)

... is equivalent to...

lst = [i ** 2 if i % 2 == 0 else i for i in range(20)]
```
What does this list comprehension give?

\[
[i \text{ if } i < 3 \text{ else } -1 \text{ for } i \text{ in } \text{range}(5)]
\]

a) \([0, 1, 2, 3, 4]\)

b) \([-1, -1, -1, 3, 4]\)

c) \([0, 1, 2, -1, -1]\)
Answer:

c) [0, 1, 2, -1, -1]
What does this ternary if give us?

\[
a = 0 \\
x = "Something" \text{ if } a > 10 \text{ else } "Nothing"
\]

a) "Something"   b) "Nothing"
Answer:

b) "Nothing"
Idiomatic Python

- Ternary if:
  \[ x = "This" \text{ if } a \text{ else } "That" \]

- List comprehensions:
  \[ ["This" \text{ if } a \text{ else } "That" \text{ for } i \text{ in something}] \]
Recursion
Example: return_sum

- Takes an int or a list
- Returns the sum of all ints

```
return_sum(1) == 1
return_sum([1, 2, 3]) == 6
return_sum([1, [2, 3], [4, [5]]]) == 15
```
Example: return_sum

- Takes an int or a list
- Returns the sum of all ints

\[
\text{return\_sum}(1) \equiv 1
\]

\[
\text{return\_sum}([1, 2, 3]) \equiv 6
\]

\[
\text{return\_sum}([1, [2, 3], [4, [5]]]) \equiv 15
\]

We can have some oddly nested lists!
Example: return_sum

- Helper function:
  
  `sum_items()`: Takes an int or a list, returns the sum
  
  - Exception: You cannot pass in the item from return_sum.

  `return_sum(x)`: cannot call `sum_items(x)`
Example: return_sum

If we had:

return_sum([[1, [2, 3], [4, [5]]]])

We could have:

- `sum_items(1) == 1`
- `sum_items([2, 3]) == 5`
- `sum_items([4, [5]]) == 9`
Example: return_sum

Implement return_sum.

def return_sum(x: Union[list, int]) -> int

Use if-statements, for-loops, etc. and sum_items().

def sum_items(y: Union[list, int]) -> int
Example: return_sum

● What if x is an int?
  ○ What would we want to return?

● What if x is a list?
  ○ What do we call sum_items on?
  ○ How do we put the results of the calls to sum_item together?
Example: return_sum
Example: return_sum

- Simpler case: if x is an int.
  - Just need to return x!

```python
if isinstance(x, int):
    return x
```
Example: return_sum

● What if x is a list?
  ○ Can't call `sum_items(x)`, but can call `sum_items()` on each item in x!

```python
sum = 0
for item in x:
    sum += sum_items(item)
return sum
```
Example: return_sum

Alternatively (list comprehension):

```
return sum([sum_items(item) for item in x])
```

```
sum = 0
for item in x:
    sum += sum_items(item)
return sum
```
Example: return_sum

● We don't actually have anything called `sum_items()`

● Would need to define it... But how?
  ○ We have `return_sum()`
  ○ It does the exact same thing as `sum_items()` should have done!
Example: return_sum

```python
if isinstance(x, int):
    return x

sum = 0
for item in x:
    sum += sum_items(item)

return sum
```
**Example: return_sum**

```python
if isinstance(x, int):
    return x

sum = 0
for item in x:
    sum += return_sum(item)

return sum
```
Example: return_sum

```python
if isinstance(x, int):
    return x
sum = 0
for item in x:
    sum += return_sum(item)
return sum
```

**Recursive call**: using a function within itself!
Example: return_sum

- Previously, we didn't have to understand *how* `sum_items()` worked.
  - Just assumed that it *did* work properly.
  - Didn't have to trace through it or anything.
- Can use `return_sum()` in the same way
Writing Recursive Functions

2 main "Steps":

1. **Base case**: No recursive calls needed; simplest case to return.
2. **Recursive step**: Assume recursive calls work. Make recursive calls and use their result.
Recursive Step

● What recursive calls are you making?
  ○ Usually a 'smaller' problem.

● How do you use the values returned and put them together?
  ○ For return_sum(): knew we wanted to add the results together
Recursive Step

- If we keep making recursive calls on simpler/smaller problems we'll eventually reach the **base case**!
- If we put things together properly and our base case is right, then correct recursion should follow!
Example: return_sum

```python
if isinstance(x, int):
    return x
sum = 0
for item in x:
    sum += return_sum(item)
return sum
```

Does this work for `return_sum(10)`?
Example: return_sum

if isinstance(x, int):
    return x

sum = 0
for item in x:
    sum += return_sum(item)

return sum

Does this work for return_sum([1, 2, 3])?
Example: return_sum

```python
if isinstance(x, int):
    return x

sum = 0
for item in x:
    sum += return_sum(item)

return sum
```

Does this work for `return_sum([1, 2, 3])`?

We call:

```python
return_sum(1)
return_sum(2)
return_sum(3)
```
Example: return_sum

```python
if isinstance(x, int):
    return x

sum = 0
for item in x:
    sum += return_sum(item)

return sum
```

Does this work for `return_sum([1, [2], [3, 4]])`?
Example:

```python
if isinstance(x, int):
    return x

sum = 0
for item in x:
    sum += return_sum(item)

return sum
```

Does this work for:

```python
return_sum([1, [2], [3, 4]])
```

We call:

- `return_sum(1)`
- `return_sum([2])`
- `return_sum([3, 4])`
Recursion

- No matter how complicated our example, we can always simplify it and make smaller cases.
- If our base case and recursive step is right, our function will be right.
- Requires some blind faith.
Example: get_factorial

def get_factorial(x: int) -> int:

- Takes in positive integer
- Returns factorial of that number
  - E.g. if x == 5, the factorial (5!) is:
    - \( 5 \times 4 \times 3 \times 2 \times 1 \)
  - 4! = 4 \times 3 \times 2 \times 1
Example: get_factorial

def get_factorial(x: int) -> int:
    product = 1
    for i in range(1, x + 1):
        product = product * i
    return product
def get_factorial(x: int) -> int:
    product = 1
    for i in range(1, x + 1):
        product = product * i
    return product

Example: get_factorial

How do we write this using recursion?
Example: get_factorial

1. Find the base case
   ● What is the simplest value of $x$ (where we can just return a single value with no other work)?
Example: get_factorial

1. Find the base case
   ● x == 1: Just return 1
   ● x == 2 also works, but we'd still have to consider the case where x == 1. 2! is also 2 * 1; so we could rely on get_factorial(1) anyways.
Example: get_factorial

def get_factorial(x: int) -> int:
    if x == 1:
        return 1

What is our recursive step?

Hint: 5! = 5 * 4 * 3 * 2 * 1  2! = 2 * 1
     4! = 4 * 3 * 2 * 1  1! = 1
     3! = 3 * 2 * 1
Example: get_factorial

● Recursive call:

get_factorial(x - 1)
Example: get_factorial

- Recursive call:
  \[ \text{get\_factorial}(x - 1) \]
- Multiply by \( x \) to get the factorial we want:
  \[ x \times \text{get\_factorial}(x - 1) \]
- After all: \( 20! = 20 \times 19! \)
def get_factorial(x: int) -> int:
    if x == 1:
        return 1
    return x * get_factorial(x - 1)
Example: get_max

- Takes an int or list
- Return the largest number

```
get_max(10) == 10
get_max([1, [2, [3, [4]]]]) == 4
get_max([1, [2, [3, 4], [5, [6], [7, [[8]]]]]]) == 8
```
get_max(x: Union[list, int])

What is the base case? Pick the SIMPLEST.

a) When x is an int
b) When x is a list containing only ints
c) When x is a list
Answer:

a) When x is an int
get_max(x: Union[list, int])

What is the base case? Pick the SIMPLEST.

a) When x is an int
b) When x is a list containing only ints
c) When x is a list

(b): Works, but have to check the type of all items.

Instead: just defer to making recursive calls with (a) as the base case.
x is a list.

What recursive calls do we want to make?.

a) get_max(x)

b) results = []
   for item in x:
      results.append(get_max(item))

c) results = []
   for item in x:
      if isinstance(item, int):
         results.append(item)
      else:
         results.append(get_max(item))
Answer:

b) \[
\begin{align*}
\text{results} & = [] \\
\text{for item in x:} \\
\text{results.append(get_max(item))}
\end{align*}
\]
Example: get_max

def get_max(x):
    if isinstance(x, int):
        return x

results = [get_max(item) for item in x]

# What do we return?
Example: `get_max`

```
[5, [10, 2], [1, [[11], [8]]]]
```

Recursive calls:

- `get_max(5) == 5`
- `get_max([10, 2]) == 10`
- `get_max([1, [[11], [8]]]) == 11`
Example: get_max

\[
[5, [10, 2], [1, [[11], [8]]]]
\]

results = [5, 10, 11]

● If we have those results, how do we get the max that we want to return?
Example: get_max

def get_max(x):
    if isinstance(x, int):
        return x

results = [get_max(item) for item in x]

return max(results)
Steps to Writing Recursive Functions

1. Find your **base case**
2. Make a **larger example** and the **recursive calls** you'd make
3. Assume your recursion works: find the **results** of the recursive calls
4. Use the results from (3) to find out how to use the results
Example: count_occurrences

count_occurrences(x: Union[int, list],
                   value: int)

- Count the number of times value appears in x

count_occurrences(2, 5) == 0
count_occurrences(5, 5) == 1
count_occurrences([1, 2, 5, 5], 5) == 2
count_occurrences([1, [2, [5], [3, [[5]]], 5], 5], 5) == 4
Example: count_occurrences

- **Base case:** x is an int
  - Then we just return 1 if x == value; 0 otherwise

```python
if isinstance(x, int):
    return 1 if x == value else 0
```
Example: count_occurrences

- **Base case:** $x$ is an int
  - Then we just return 1 if $x == value$; 0 otherwise

```python
if isinstance(x, int):
    return 1 if x == value else 0
```

Example: `count_occurrences(x, 5)`

\[x = [1, [2, 3], [[5], 5], [5, [6, [5, [[5]], 2], 5]]]\]

Figure out the recursive step!

1. What would we expect to be returned?
2. What recursive calls do we make?
3. What do we expect from the recursive calls?
4. How do we use those results?
5. Write the code!
Example: \texttt{count\_occurrences(x, 5)}

\begin{align*}
x &= [1, [2, 3], [[5], 5], [5, [6, [5, [[5]], 2], 5]]] 
\end{align*}

1. **What would we expect to be returned?** 6
2. What recursive calls do we make?
3. What do we expect from the recursive calls?
4. How do we use those results?
5. Write the code!
Example: `count_occurrences(x, 5)`

\[ x = [1, [2, 3], [[5], 5], [5, [6, [5, [[5]], 2], 5]], 5] ] \]

2. What recursive calls do we make?

- `count_occurrences(1, 5)`
- `count_occurrences([2, 3], 5)`
- `count_occurrences([[5], 5], 5)`
- `count_occurrences([[5], 5], 5)`
- `count_occurrences([5, [6, [5, [[5]], 2], 5]], 5)`
Example: count_occurrences(x, 5)

x = [1, [2, 3], [[5], 5], [5, [6, [5, [[5]], 2], 5]]]

3. What do we expect from the recursive calls?

count_occurrences(1, 5) == 0
count_occurrences([2, 3], 5) == 0
count_occurrences([[5], 5], 5) == 2
count_occurrences([5, [6, [5, [[5]], 2], 5]], 5) == 4
Example: count_occurrences(x, 5)

\[ x = [1, [2, 3], [[5], 5], [5, [6, [5, [[5]], 2], 5]]] \]

4. How do we use those results?

\[
\begin{align*}
count\_occurrences(1, 5) &= 0 \\
count\_occurrences([2, 3], 5) &= 0 \\
count\_occurrences([[5], 5], 5) &= 2 \\
count\_occurrences([5, [6, [5, [[5]], 2], 5]], 5) &= 4
\end{align*}
\]

Add the results together!
Example: count_occurrences

return sum([count_occurrences(item, value)
             for item in x])
Example: count_occurrences

```python
def count_occurrences(item, value):
    # Function to count occurrences of a value in an item

lst = []
for item in x:
    lst.append(count_occurrences(item, value))

return sum(lst)
```

```python
return sum([count_occurrences(item, value)
            for item in x])
```
Example: count_occurrences

```python
if isinstance(x, int):
    return 1 if x == value else 0

return sum([count_occurrences(item, value)
             for item in x])
```
Iterative vs. Recursive

- Anything we write iteratively, we can write recursively.
  - If we can write it with a loop, we can write using recursion
  - The opposite is also true: anything we write with recursion, we can write with a loop
Writing Recursion via Loops

- Requires a lot of book-keeping
- Recall from Stack lecture: "Call Stack"
  - Same idea: have to keep track of everything returned.
- An example of this will be in your 2nd assignment
Writing Loops via Recursion

- Usually straightforward
- Gives us some good practice problems
Example: linked_list_count

linked_list_count(
    x: Union[LinkedListNode, None],
    value: Any) -> int

- Return how many times value appears in the LinkedList starting at node x
Example: `linked_list_count`

```python
linked_list_count(
    x: Union[LinkedListNode, None],
    value: Any) -> int
```

- Return how many times value appears in the LinkedList starting at node `x`
Example: linked_list_count

def linked_list_count(x, value):
    cur_node = x
    count = 0
    while cur_node != None:
        if cur_node.value == value:
            count += 1
        cur_node = cur_node.next_
    return count
Example: `linked_list_count`

```python
def linked_list_count(x, value):
    cur_node = x
    count = 0
    while cur_node is not None:
        if cur_node.value == value:
            count += 1
        cur_node = cur_node.next_
    return count
```

How do we write this recursively?
Example: linked_list_count

def linked_list_count(x, value):

1. Find the base case!
Example: linked_list_count

def linked_list_count(x, value):

1. Find the base case!

If x is None, then we can return 0!
(There's no nodes to check.)

if x is None:
    return 0
Example: linked_list_count

```python
def linked_list_count(x, value):

2. Figure out our recursive calls.
We have access to:
• x.next_
• x.value
```
Example: linked_list_count

```python
def linked_list_count(x, value):
```

2. Figure out our recursive calls.

```python
linked_list_count(x.next_, value)
```
Example: linked_list_count

Recurse on these nodes
Example: linked_list_count

Expected result: 2
From calling
linked_list_count(x.next_, 1)
Example: linked_list_count

Expected result: 3

From calling
linked_list_count(x, 1)
Example: linked_list_count

Recursive call: 2

We want to return: 3

We return 3 because x.value == 1
Example: linked_list_count

def linked_list_count(x, value):

3. How do we use the results of the recursive call?

Return the count from the recursive call, 
+ 1 if x.value == value
Example: `linked_list_count`

```python
return ((1 if x.value == value
else 0) +
        linked_list_count(x.next_,
                          value))
```
Example: `linked_list_count`

```python
return ((1 if x.value == value else 0) +
        linked_list_count(x.next_, value))
```

Alternatively:

```python
count = linked_list_count(x.next_, value)
if x.value == value:
    count += 1
return count
```
Example: linked_list_count

```python
if x is None:
    return 0

return ((1 if x.value == value
          else 0) +
        linked_list_count(x.next_,
                          value))
```
Example: find_greater

find_greater(
    x: Union[LinkedListNode, None],
    value: Any) -> int

- Return True if a value greater than value appears in the LinkedList starting at node x
Example: find_greater

find_greater(
    x: Union[LinkedListNode, None],
    value: Any)

● Return True if a value greater than value appears in the LinkedList starting at node x

Example: find_greater(x, 5) == True

value = 5

Linked List:

1 → 5 → 3 → 1 → 2 → 6 → 8
Example: find_greater

find_greater(
    x: Union[LinkedListNode, None],
    value: Any) -> int

- Return True if a value greater than value appears in the LinkedList starting at node x

1. What is the base case?
Example: find_greater

1. What is the base case?

If \( x \) is None, then there's no value to check: return False.

```python
if x == None:
    return False
```
Example: `find_greater`

2. What is our recursive step?

We have access to `x.next_` and `x.value`

`find_greater(x.next_, value)`
Example: find_greater

2. What is our recursive step?

```python
find_greater(x.next_, value)
```

should return True!
Example: find_greater

We also know we'd want to return True if x.value > value!

(This would be considered an additional base case!)
Example: find_greater

return ((x.value > value) or find_greater(x.next_, value))
Example: find_greater

```python
return ((x.value > value) or find_greater(x.next_, value))
```

Alternatively:

```python
if x.value > value:
    return True
return find_greater(x.next_, value)
```
Example: find_greater

```python
if x > None:
    return False

return ((x.value > value) or
        find_greater(x.next_,
                     value))
```
Assignment 1

- Documentation (docstrings, etc.) will be graded!
  - We've given you documentation to fill in for `a1_battle_queue`
  - You don't need docstring examples for `a1_playstyle`
  - You need to document everything else that you write!
Assignment 1

- You're responsible for testing your BattleQueue!
  - I don't give you the unit tests (aside from using it in the special_attack() tests)
  - If I did give them to you, the hidden tests would be much, much harder (and would be worth much, much more!)
Assignment 1

● Submit all files that you used
  ○ Exceptions: the sprites, a1_ui.py, the unittest files

● I should be able to download your submission and use it with the sprites + a1_ui.py and be able to run it.
Homework

● Exercise 5 (due next Thursday @ 11PM) (+ Lab 5 is out)

● Assignment 1 due Sunday @ 11PM
  ○ TA Office Hours Tomorrow:
    2 - 4 PM in BA3201

● No lecture next week/the week after!
  ○ I'll still hold office hours.