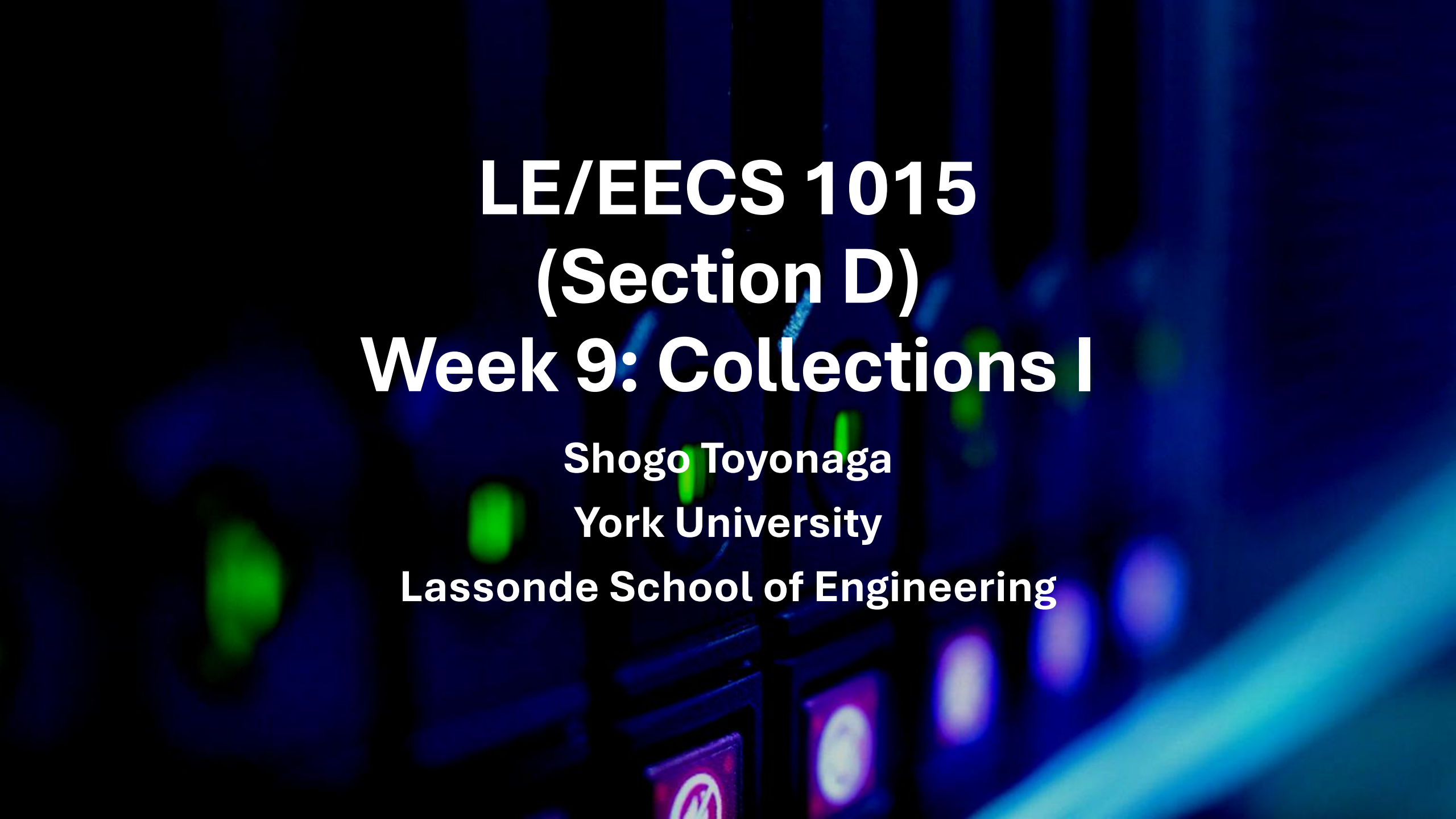

LE/EECS 1015 (Section D) Week 9: Collections I

Shogo Toyonaga

York University

Lassonde School of Engineering

This Week...

1. Data Structures
    - Tuple
    - List
    - Set
    - Dictionary
2. Docstring

Goals of Lab 7

1. Using data structures from the built-in collections module to implement and/or debug problems.
2. Writing concise doctest for functions that use built-in collections

Lab 7 – What You Do....

Task	Points
Follow the Steps (Calculate Average)	30
Debugging (Fruit Quantity)	30
Implementation (Inventory Catalogue)	10
Implementation (Intersection of Chars)	10
Implementation (Remove List Item)	5
Implementation (Remove List Index)	5
Implementation (Remove Tuple Index)	10

Lab 7 – Useful Resources

- Python Documentation: Collections
- Python Documentation: Type Hinting (Doctest)
- Python lists, sets, and tuples explained (BroCode)
- Python dictionaries are easy (BroCode)
- Practice Problems (w3schools)

Tuples

- Represents an immutable ordered collection of data which is indexed by integers.
- As a best practice, you tend to declare a tuple by providing a comma separated sequence of data types inside of round brackets: (a_1, a_2, a_3, a_4).
- Basic Operators follow similar behaviours from Strings (e.g., Indexing & Slicing):
  - '+' → Concatenation
  - '*' → Duplication
  - Comparisons → Element-by-Element using Lexicographic Ordering. The compared elements must be of the same data type.

Tuples

- Note that unpacking allows you to quickly assign multiple variables with values from a tuple. The number of elements on the left and right side must be balanced.
- Tuple Methods:
  - $count(self, value, /)$: Returns the number of occurrences of, “value”
  - $index(self, value, start, stop, /)$: Returns the first index of, “value” if it exists in the Tuple; Otherwise, it raises a ValueError.
- Important Functions
  - “a in b” → Checks if ‘a’ is in the collection (e.g., tuple), ‘b’

Tuples (Homework)

1. Write a function which uses Tuple Packing. Check that all the elements are integers. If so, return the mean (as a float).
2. Use the $help()$ function to learn about $enumerate()$ and $zip()$.
3. Implement a function, $unzip()$ which effectively undoes $zip()$ for a Tuple.

Lists

- Represents a mutable ordered collection of data which is indexed by integers.
- As a best practice, you tend to declare a list by providing a comma separated sequence of data types inside of square brackets: $[a_1, a_2, a_3, a_4]$.
- Basic Operators follow similar behaviours from Strings (e.g., Indexing & Slicing):
  - '+' → Concatenation
  - '*' → Duplication
  - Comparisons → Element-by-Element using Lexicographic Ordering. The compared elements must be of the same data type.

Lists

- Unpacking and the ‘in’ operator are also supported with Lists.
- (Some) List Methods:
  - $append(self, object,/)$: Append object to the end of the list.
  - $index(self, value, start, stop,/)$: Returns first index of a value if its in the list; Otherwise, raises a ValueError.
  - $remove(self, value,/)$: Remove first occurrence of value, otherwise, return a ValueError.
  - $pop(self, index,/)$: Removes and returns the item at index. Raises IndexError if list is empty or index is out of range.
- For more information, refer to $help(list)$

Sets

- Represents a mutable unordered collection of data which does not support indexing. The elements in a set must be immutable.
  - As there is no indexing, slicing is not supported…
  - Repeated (Duplicate) variables are omitted from sets; they only contain unique values.
  - '+' and '*' are not supported (like for Lists and Tuples)
- As a best practice, you tend to declare a set by providing a comma separated sequence of data types inside of curly brackets: $\{a_1, a_2, a_3, a_4\}$.
- For comparison operators, $a < b$ refers to a strict subset
- For comparison operators, $a > b$ refers to a strict superset

Sets

- **Unpacking and the ‘in’ operator are also supported with Sets.**
- **Most of if not all the set methods are inspired from their mathematical counterparts:**
  - $A - B = set.difference()$
  - $A \cap B = set.intersection()$
  - $A \cup B = set.union()$

Dictionary (HashTable)

- Represents a mutable unordered collection of data. It relies on using key-value pairs for retrieving and storing data.
- Keys must be immutable
- As a best practice, you tend to declare a dictionary by providing the key-value pairs between curly brackets like so:

$$eg = \{k_1 : v_1, k_2 : v_2, \dots, k_n : v_n\}$$

- Comparison and arithmetic operators are not supported by default.

Dictionary (HashTable)

- The ‘in’ operator works, but only for checking if a key exists in the dictionary.
- <u>(Some) Dictionary Methods:</u>
  - $get(self, key, default,/)$
  - $items()$: Returns a set-like object providing a view of the dictionary items (key-value pairs).

Dictionary (HashTable)

- The hash function consistently maps a name to the same index.
- Collisions occur when two keys are assigned to the same index. This is bad because the, “solution” (e.g., Linked List Approach) greatly reduces runtime efficiency.
- In the average case, a HashTable takes $O(1)$ for searching, inserting, and deleting elements in the collection. Multiple collisions will have it take $O(n)$ time in the worst-case.

Hash Functions

- You can tell if a hash function is good or bad based on the load factor: $LF = \frac{\#(Items \in HT)}{\#(Slots)}$
- If $LF > 1$, this means that you have more items than available slots (indexes). You need to add more slots!
- You should resize the HashTable if $LD > 0.7$ to avoid collisions. This typically involves doubling the size by $\times 2$ and re-inserting the key-value pairs into the new HashTable.
- A good hash function distributes values in the array evenly whereas a bad one groups things together.

Time Complexity: Overview (Average Case)

Data Structure	Access	Search	Insert	Delete
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$
HashMap		$O(1)$	$O(1)$	$O(1)$

Putting It All Together...

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that:

- Each input would have exactly one solution.
- You may not use the same element twice.
- You can return the answer in any order.

Constraints:

- $2 \leq nums.length \leq 10^4$
- $-10^9 \leq nums[i] \leq 10^9$
- $-10^9 \leq target \leq 10^9$

Putting It All Together...

```
# Keep track of candidate solutions
memo = {}
for i in range(len(nums)):
    if nums[i] in memo:
        return [memo[nums[i]], i]
    # Save <k,v>: the key represents the second integer required to sum to target; the value is the index of the first integer
    memo[target - nums[i]] = i
```

Putting It All Together...

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

Putting It All Together...

```
# Keep track of the required closing brackets (ordered....)
stack = []
# Tell us what kind of bracket to expect (after reading an opening one....)
mapping = {
    '(' : ')',
    '{' : '}',
    '[' : ']'
}

# Iterate through each character of the input (s)
for c in s:
    # If we read an open bracket, push the corresponding closing bracket onto the stack
    if c in mapping:
        stack.append(mapping[c])
    # If the stack is empty or the popped element is not the correct corresponding closing bracket, return False (Bad input)
    elif len(stack) == 0 or stack.pop() != c:
        return False
return not stack
```

Thank You!

Shogo Toyonaga
Lassonde School of Engineering