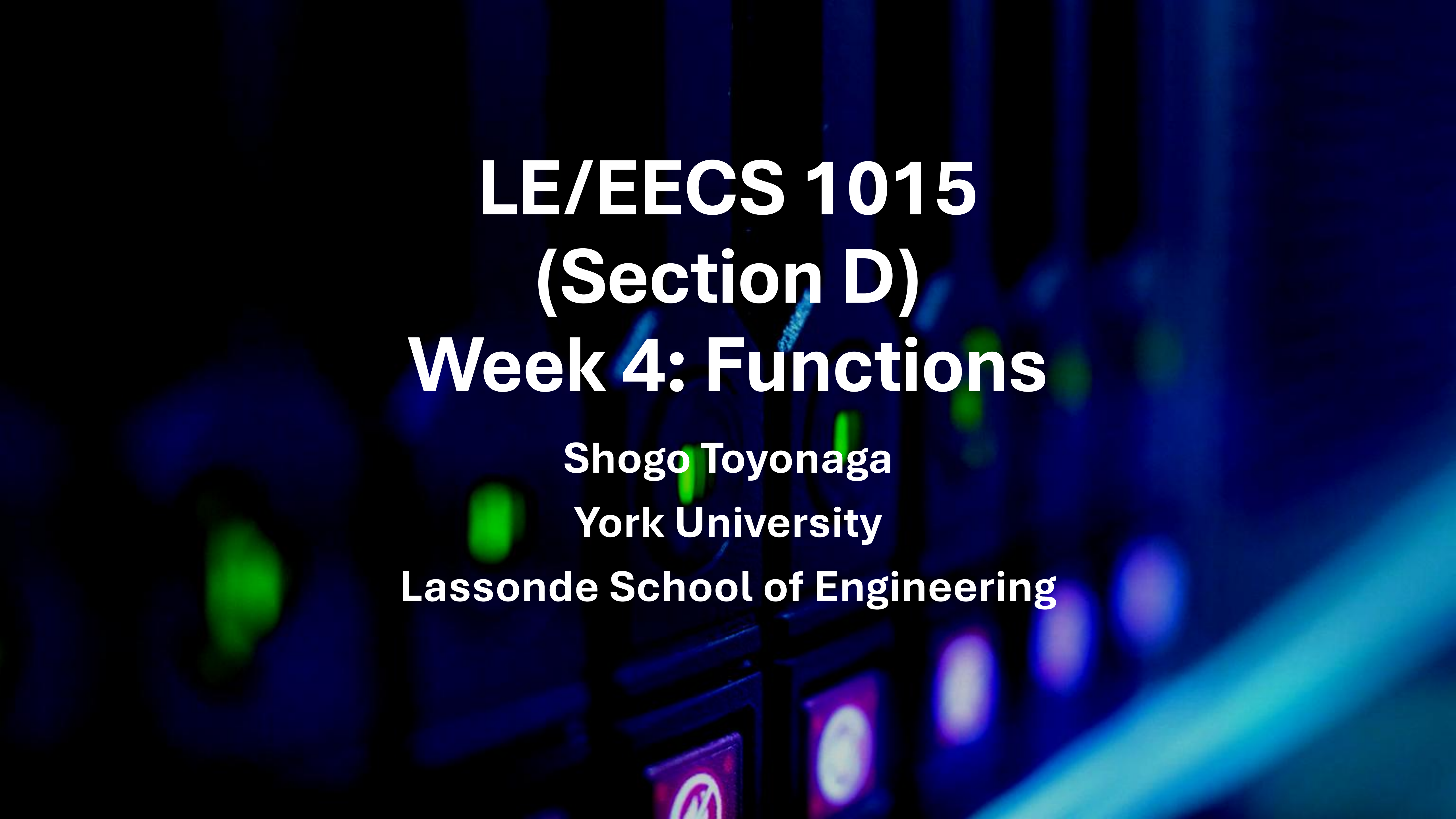

LE/EECS 1015 (Section D) Week 4: Functions

Shogo Toyonaga

York University

Lassonde School of Engineering

This Week…

1. String Formatting
   - Concatenation $(+)$
   - Using $\%$ with Placeholders: $\%\boldsymbol{s}, \%\boldsymbol{d}, \%\boldsymbol{f}, \%\boldsymbol{r}$
   - Str.format
   - F-Strings
2. Functions
   - Design Recipe
3. Test-Driven Development (TDD)
   - Doctest
   - Unittest

Goals of Lab 4

1. Writing functions with the design recipe

2. Debugging functions through Test-Driven Development (doctest, unittest)

Lab 4 – What You Do….

Task	Points
Follow the Steps (Fruit Prices)	30
Debugging ($x \# y \equiv x^2 - y^2$)	30
Implementation (Count Wheels)	10
Implementation (Child Tickets)	10
Implementation ($x \oplus y$)	10
Implementation ($x \# y \equiv x^2 - y^2$)	10

Lab 4 – Useful Resources

- f-string Examples
- Python string format 💬 (Bro Code)
- Format specifiers in Python are awesome 💬 (Bro Code)
- Functions in Python are easy 📞(Bro Code)
- Testing code examples in docstrings with Python's doctest (redshiftzero)
    - Python Documentation
- Unit testing | Intro to CS - Python | Khan Academy
    - Python Documentation

String Formatting

1. Concatenation
   - Both operands must be of type str or else a TypeError will be thrown
2. ‘%’ with Placeholders
   - Uses common placeholders such as, “$\%s\ (str), \%f\ (float), \%d\ (int), \%r\ (bool)$” to substitute values in order
3. Built-In String Formatting
   - Uses $\{\}$ as the placeholder; advantageous in the sense that you don’t need to declare specific data types
   - For n substitutions, you can define your own ordering by providing indexes between the parentheses from $[0, n)$
4. f-Strings
   - Allows you to directly substitute variable names into the string surrounded by $\{\}$

String Formatting

- Choosing the method to format your strings is largely up to preference, however:
  - Concatenation can become very complex with multiple variables. You also need to manage whitespaces properly. Sometimes using str.join() may be more advantageous
  - Some strategies (e.g., % with placeholders) offer better readability but are more cumbersome (verbose) to write out for very simple tasks
- I personally prefer using f-strings!

Functions

- Functions reduce the amount of time to re-write and re-use the same code with minor modifications (arguments)
- Supports modular design which also helps to increase readability
- Uses the, “def” keyword followed by the function name, parameters (with data types), and return type to specify the header.
  - Recommended to follow PEP-8 naming convention; lowercase words separated by underscores.
- Variables defined in a function are local and can only be accessed inside of the function (More on this much later….)
- Function calls can be nested (Remember Lab 2!)

Functions (The Design Recipe 🍴)

- Used to define the roadmap for designing and evaluating functions through Test-Driven Development (TDD)
- Some prerequisite terminology to cover:
    - Pre-Condition: A, “promise” that if the functions requirements are met (e.g., about the parameters), then the code will behave as expected.
        - If the pre-condition is violated, there are no explicit guarantees on how the function will operate.
    - Post-Condition: A behaviour or, “promise” that the program guarantees after execution if the pre-condition was maintained.

Functions (The Design Recipe 🍴)

1. Define the Header
    - Function Name
    - Parameters (and Types)
    - Return Type
2. Consider the Contract $(Pre \rightarrow Post)$
    - We use $assertions$ to implement checks for the pre-condition!
    - Read about how to use them in interactive mode: $help("assert")$
3. Write DocString
    - You should still be writing comments in your function body; DocString is not all that you need (😉) to maintain excellent code readability.
4. Write Test-Cases
5. Implement the Function Body
6. Debug for Correctness & Performance

Putting it all together…

1. Write a function $generate\_symmetric\_pyramid(height: int)$, that returns a string-representation of a symmetric pyramid with asterisks. The height should be a positive integer.
2. Write a function $generate\_right\_angle\_triangle(height: int)$, that returns a string-representation of a left-leaning right-angle triangle with asterisks. The height should be a positive integer.
3. Write a function $generate\_sum\_of\_even\_integers(numbers: List[int])$, that returns the sum of even integers from the list, numbers.
4. Write a function $convert\_time\_format(seconds: int)$, which returns a string formatted as $hh: mm: ss$. Don't worry about overflow (for now).

Unit Testing

Unit Testing with Brevity 😉

- The three types of testing equivalence classes:
    1. **Happy:** A well-defined test case using known inputs that execute and produce the expected output. It does not guarantee handling of error conditions.
    2. **Boundary:** Synonymous with, "edge / corner" cases. It tests the application under an extreme (min/max) operating parameter.
    3. **Exceptional:** Used to test the application under contract-breaking violation(s). We want to test the robustness of the applications error handling.
- At a minimum, you must implement one test-case per equivalence class.

Thank You!

Shogo Toyonaga
Lassonde School of Engineering