# Beyond the Textbook (Zelle 3e - Chapter 7)

# Decision Structures

# Review Simple `if`

- See: *decisions_01_simple.py*

# Chaining Comparison Operators

- In the proper circumstances, chaining multiple comparison operators can lead to more readable code.
- See: *decisions_02_chaining_comparison_operators.py*

# Truth Value Testing

- All Python variables may be tested for **truthiness** regardless of their type.
- Empty and zero values evaluate to `False`.
- Non-empty and non-zero values evaluate to `True`.
- Some Python programmers believe that this leads to more readable code.
- See Tutorial Article.
- See: *decisions_03_truth_value_testing.py*

# Review Two-Way `if`

- See: *decisions_05_two_way.py*

# Review Multi-Way `if`

- When constructing a multi-way `if` that uses **inequalities**, you must test conditions **in order**.
- Ascending order is preferred.
- See: *decisions_10_multi_way.py*

# Extended Multi-Way `if`

- When constructing an extended multi-way `if` that uses **inequalities**, you must test conditions **in order**.
- Ascending order is preferred.
- See: *decisions_15_multi_way_extended.py*

# Using Multi-Way `if` For Lookups

- Inline lookups can be coded with a multi-way `if`.
- Refactoring the lookup into a function often leads to more readable code.
- When we get to Zelle 3e Chapter 11, we will learn how to do lookups using a Python `dictionary`.
- At this point in the course, we are learning how to do this without the `dictionary`.
- See:
  - *decisions_20_lookup.py*
  - *decisions_25_lookup_in_function.py*

# Using Nested `ifs` to Implement Complex Choices

- **Nested** `ifs` can be used to implement complex choices.
- Any code block can contain a simple, two-way, or multi-way `if`.
- Nesting `ifs` two levels deep is most common.
- Nesting `ifs` three levels deep is recommended only if it results in readable code.
- Nesting `ifs` more than three-levels deep is generally considered a bad practice.
- Refactoring a nested `if` into a function often leads to more readable code.
- See:
  - *decisions_30_nested_inline.py*
  - *decisions_35_nested_in_function.py*

# Using `try`/`except` Blocks

- `try`/`except` blocks allow recovery from anticipated program exceptions.
- Otherwise, exceptions cause a **stack trace** to print on the console and execution stops.
- The `try` block contains the code that might raise an exception.
- `except` blocks contain code that detects exceptions and handles them.
- The `finally` block allows for some code to run regardless of whether an exception was raised.
- See:
  - *decisions_40_try.py*
  - *decisions_43_try_with_called_code.py*

# Using `raise` to Signal an Error Condition

- We can `raise` exceptions in our own code as a way of signalling error conditions.
- This architecture allows called code to detect errors and calling code to handle them.
- Exceptions are implemented with Python classes.
- When raising exceptions, we often re-use the builtin Python exception classes. See Python Documentation.
- It is possible to create our own exception classes by creating a custom Python classes. See tutorial article.
- See *decisions_45_raise.py*

# Finding the Lowest (Highest, Longest, Shortest, Etc.)

- Finding the lowest (highest, etc.) value in a `list` is easily done with the builtin `min` and `max` functions.
- Finding the lowest (highest, etc) value in a file of entries is harder and requires that you follow a popular **design pattern**.
- This design pattern requires that the programmer know how to express very high and low values. Here are some references on how that is done for:
  - int
  - float
- See *decisions_50_find_lowest.py*

# Using `ifs` For Multi-Faceted Validation

- **Multi-faceted validation** can be implemented using a series of `if` statements.
- In this design pattern, we usually begin by assuming the the input is **valid**.
- Then, each facet is tested in turn.
- A failure of any one test, makes the input **invalid**.
- See:
  - *decisions_70_using_ifs_for_validation.py*
  - *decisions_75_validation_using_function.py*
  - *decisions_80_validation_using_function_and_messages.py*

# Extra Python Features (Syntactic Sugar)

See https://en.wikipedia.org/wiki/Syntactic_sugar

# Structural Pattern Matching

- This is a `switch` statement for Python. See Wikipedia article
- New in Python 3.10.
- We are covering it here in its simplest form: a substitute for the multi-way `if`.
- It also introduces a **pattern matching** mechanism that is potentially much more powerful than the multi-way `if`. See tutorial in Python documentation.
- See:
  - *decisions_90_lookup_using_structural_pattern_matching.py*
  - *decisions_92_lookup_in_function_using_structural_pattern_matching.py*

# Easier Message Formatting With Ternary `if`

- Sometimes we want to format an output message that is slightly different depending upon data values.
- A classic example is when we want the message to include plural or singluar terms based upon data values.
- This is possible using the two-way `if`.
- But, it may be easier to code using the **ternary** `if`.
- See:
  - *decisions_94_formatting_without_ternary_if.py*
  - *decisions_96_formatting_with_ternary_if.py*

**Last Revised 2022-09-18**