

# SOFTWARE ENGINEERING CS 487

## Homework – 3

Name : Satyam Rajput (A20537375)

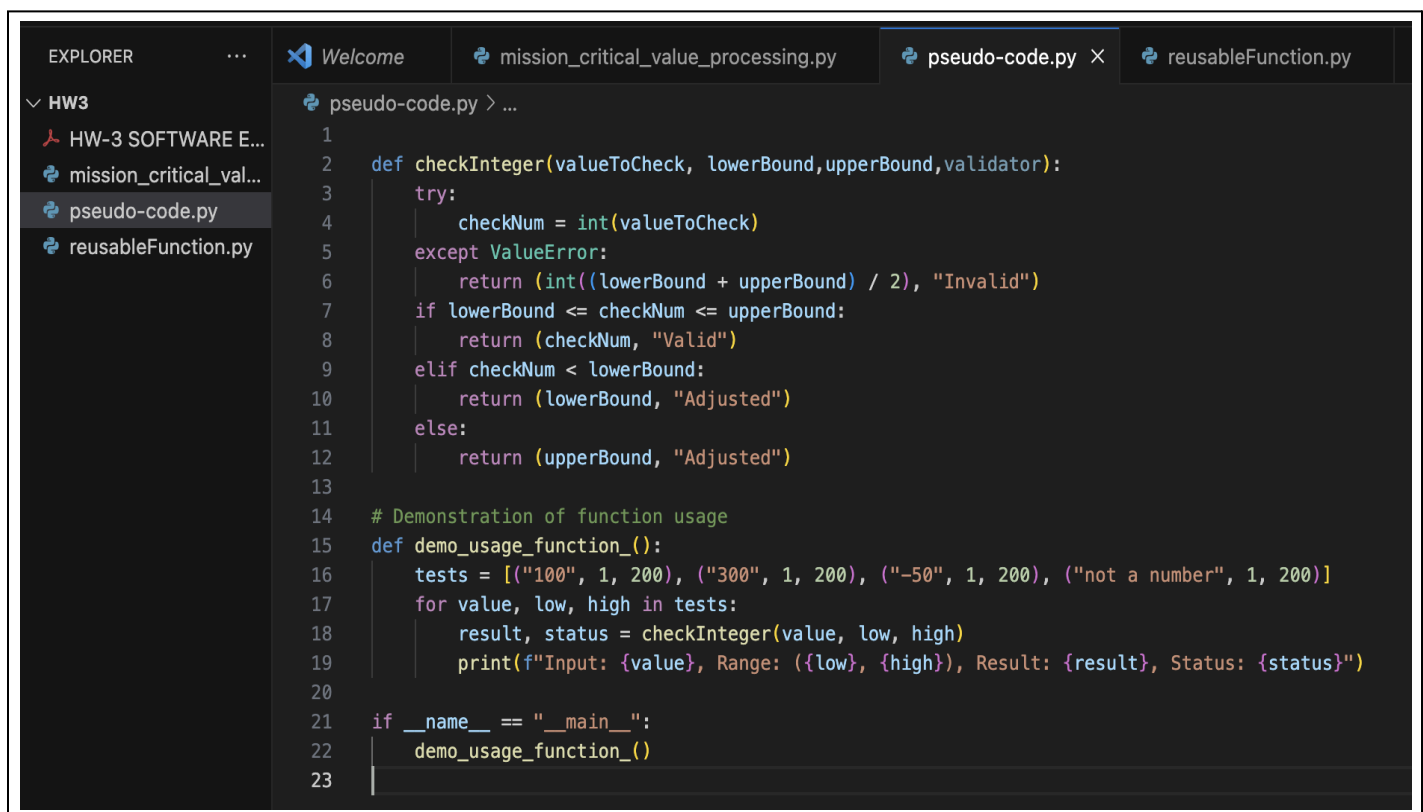
[srajput3@hawk.iit.edu](mailto:srajput3@hawk.iit.edu)

### Homework - 3

**Design a component which validates input values.**

1. Show pseudo-code for a module which must assess an integer against a valid range. It must return a valid value as close as possible to the input and an indicator of the returned value's reliability.

- The function shown in the pseudo-code validates if a given value is within a specific range.  
If the value is not a valid integer  
or lies outside the given range, Adjusts the value and also provides an indicator of the value's reliability.



```
EXPLORER  ...  Welcome  mission_critical_value_processing.py  pseudo-code.py X  reusableFunction.py

HW3
  HW-3 SOFTWARE E...
  mission_critical_val...
  pseudo-code.py
  reusableFunction.py

pseudo-code.py > ...
1
2 def checkInteger(valueToCheck, lowerBound, upperBound, validator):
3     try:
4         checkNum = int(valueToCheck)
5     except ValueError:
6         return (int((lowerBound + upperBound) / 2), "Invalid")
7     if lowerBound <= checkNum <= upperBound:
8         return (checkNum, "Valid")
9     elif checkNum < lowerBound:
10        return (lowerBound, "Adjusted")
11    else:
12        return (upperBound, "Adjusted")
13
14 # Demonstration of function usage
15 def demo_usage_function():
16     tests = [("100", 1, 200), ("300", 1, 200), ("-50", 1, 200), ("not a number", 1, 200)]
17     for value, low, high in tests:
18         result, status = checkInteger(value, low, high)
19         print(f"Input: {value}, Range: ({low}, {high}), Result: {result}, Status: {status}")
20
21 if __name__ == "__main__":
22     demo_usage_function()
23
```

The function also tags the output with a status indicating whether the value was valid, adjusted, or invalid.

- It takes the Parameters:
  - valueToCheck: The value to be evaluated.
  - lowerBound: The minimum valid value.
  - upperBound: The maximum valid value.
- And Returns:
  - A tuple containing the validated (or adjusted) value and a string indicating the reliability of this value ("Valid", "Adjusted", or "Invalid").

### Input Validation and Range Checking :

- The function **checkInteger** takes an input value along with a minimum (**lowerBound**) and maximum(**upperBound**) value to define the valid range.

### Value Correction and Reliability Indicator :

- It attempts to convert the input value to an integer. If this conversion fails (the input is not a valid integer), it returns a default value (the midpoint of the range) marked as "**Invalid**".
- If the conversion succeeds, it checks if the number falls within the specified range. If so, it returns the number marked as "**Valid**".
- If the number is outside the range, it **adjusts** the number to the nearest boundary of the range and marks it as "**Adjusted**".
- The function returns a tuple containing the validated (or adjusted) value and a **reliability indicator**.

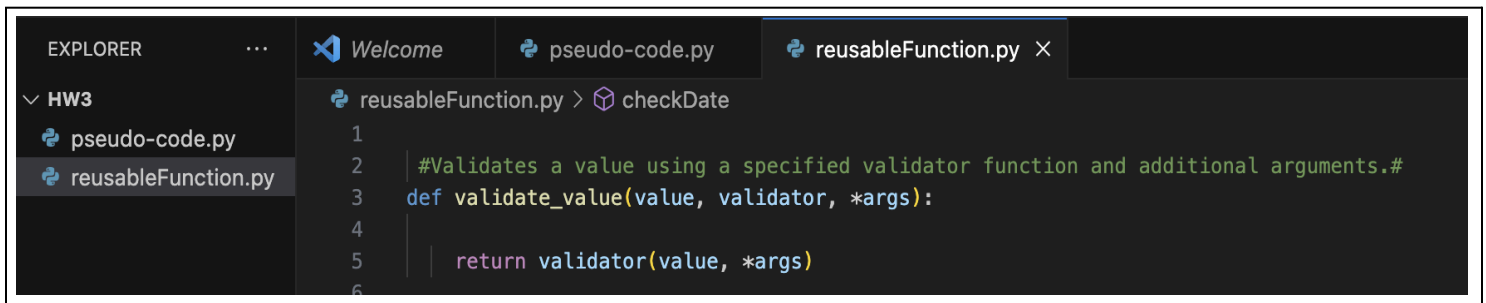
### Robustness and Error Handling:

- By incorporating helper functions the component is made more robust. It can handle inputs that might otherwise cause errors or exceptions, ensuring the system's stability.

2. Make it reusable – identify the code segments that can remain as is and those which need to be modified such that the component can be used to process other data types (e.g., date and string values)

### A) Define a Generic Validation Function

- The generic validation function will identify the input's data type and apply the proper validation logic by that type.



The screenshot shows a code editor with a dark theme. The Explorer panel on the left shows a project named 'HW3' with two files: 'pseudo-code.py' and 'reusableFunction.py'. The main editor area shows the 'reusableFunction.py' file with the following code:

```
1  
2 | #Validates a value using a specified validator function and additional arguments.  
3 | def validate_value(value, validator, *args):  
4 |  
5 |     return validator(value, *args)  
6 |
```

- Parameters
  - value: The value to be validated.
  - validator: A function that will perform the validation. This function is expected to return a tuple of (processed\_value, status).
  - args: Additional arguments required by the validator function.
- Returns
  - A tuple of the processed value and a status tag ('Valid', 'Adjusted', or 'Invalid')

### B) Implementing Type specific validation

- Implement a validator function (**checkInteger**) for each type of data (integer, date, string, etc.). The **checkInteger** range function, which is currently in use, can serve as the integer validator if its interface is slightly modified to conform to the generic validator pattern.

```

7  # Integer Validator (Existing Function Modified)
8  def checkInteger(valueToCheck, lowerBound, upperBound, validator):
9      try:
10         checkNum = int(valueToCheck)
11     except ValueError:
12         return (int((lowerBound + upperBound) / 2), "Invalid")
13     if lowerBound <= checkNum <= upperBound:
14         return (checkNum, "Valid")
15     elif checkNum < lowerBound:
16         return (lowerBound, "Adjusted")
17     else:
18         return (upperBound, "Adjusted")
19
20  # Date Validator
21  def checkDate(valueToCheck, min_date, max_date):
22      # Date-specific validation logic here
23      pass
24
25  # String Validator
26  def checkString(valueToCheck, pattern):
27      pass
28

```

### C) Reusable Function Usage

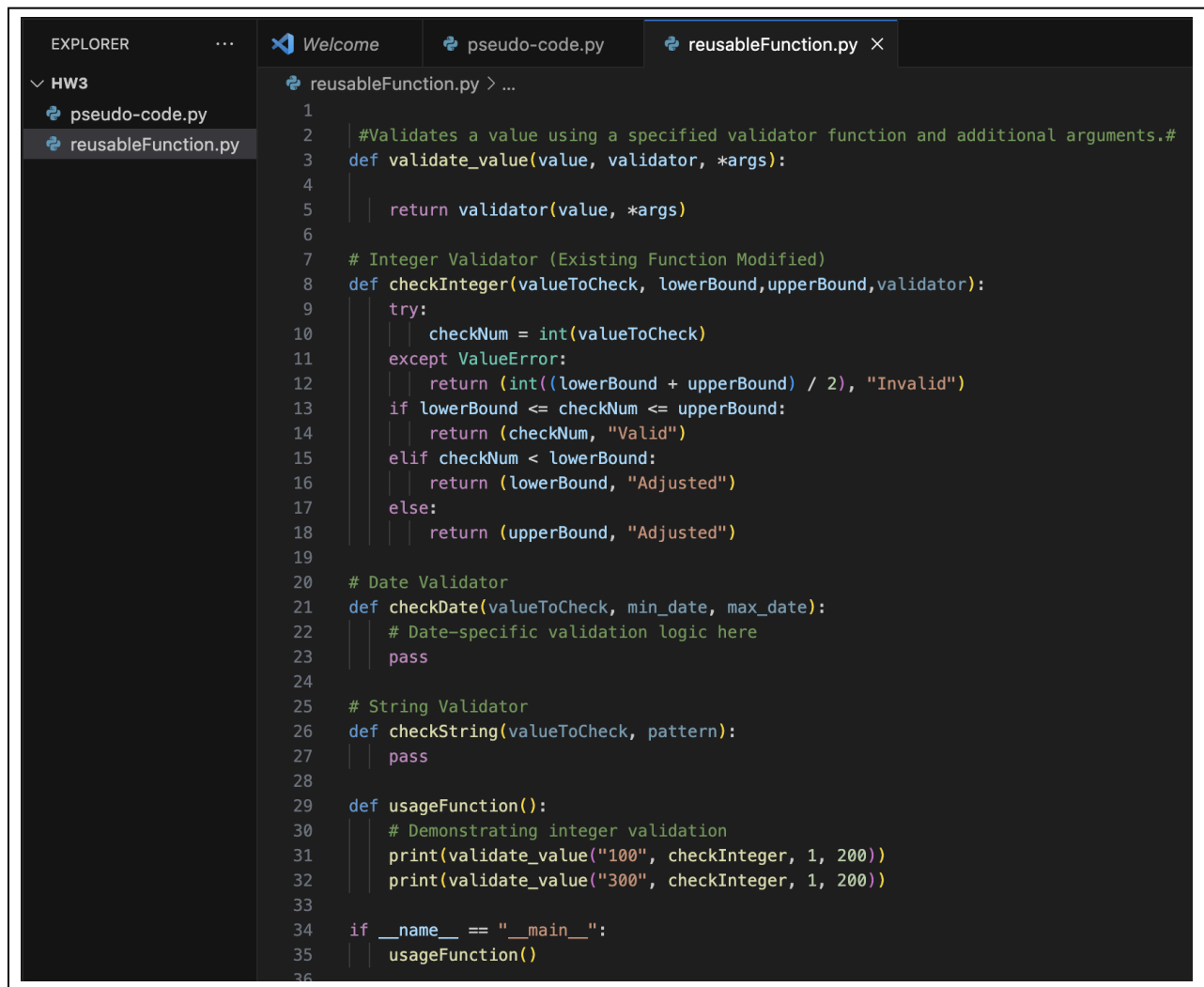
- The function `validate_value` can be used with different validators for various data types.

```

29  def usageFunction():
30      # Demonstrating integer validation
31      print(validate_value("100", checkInteger, 1, 200))
32      print(validate_value("300", checkInteger, 1, 200))
33
34  if __name__ == "__main__":
35      usageFunction()
36

```

## Complete Code :

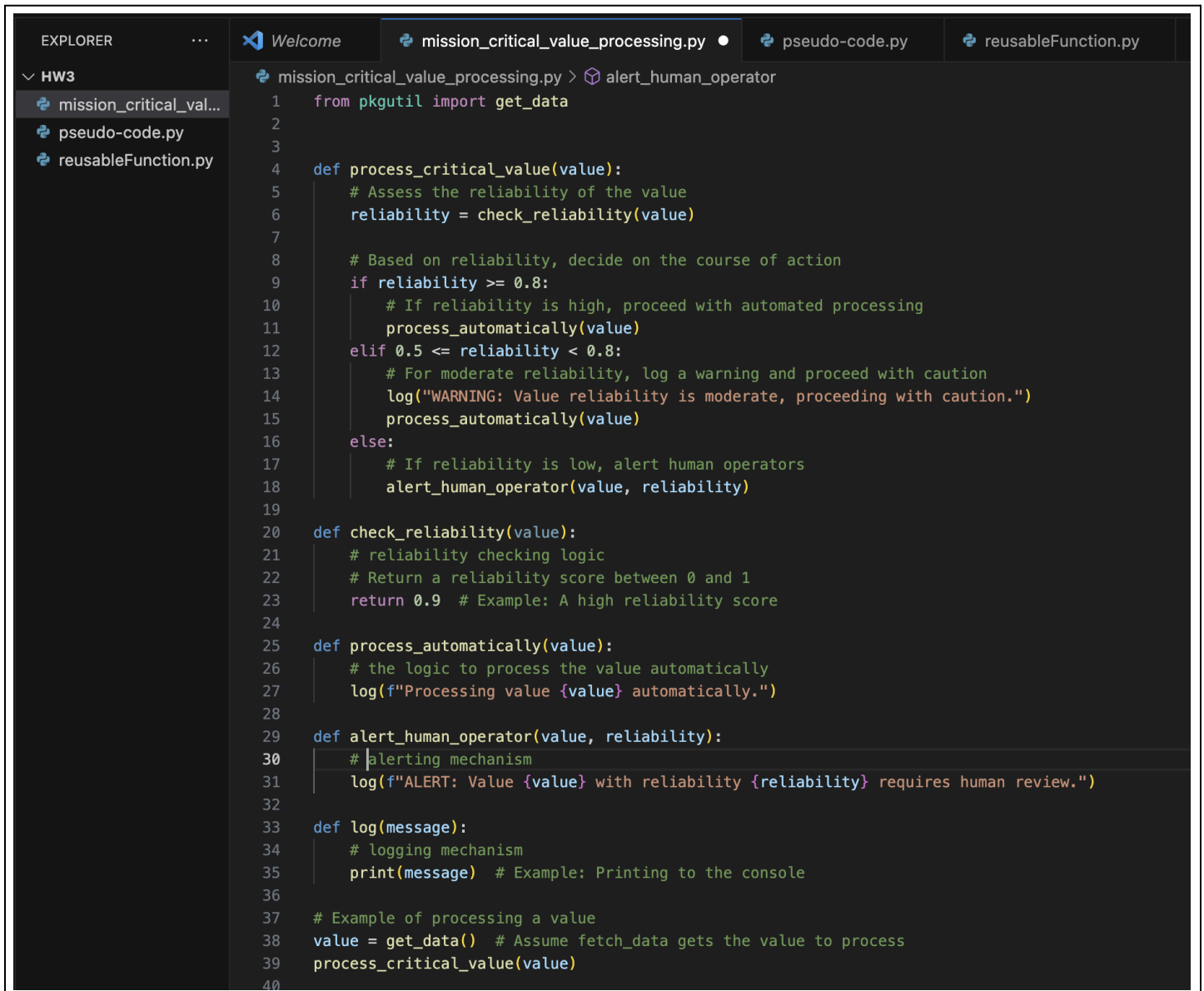


```
1
2  #Validates a value using a specified validator function and additional arguments.
3  def validate_value(value, validator, *args):
4
5      return validator(value, *args)
6
7  # Integer Validator (Existing Function Modified)
8  def checkInteger(valueToCheck, lowerBound, upperBound, validator):
9      try:
10         checkNum = int(valueToCheck)
11     except ValueError:
12         return (int((lowerBound + upperBound) / 2), "Invalid")
13     if lowerBound <= checkNum <= upperBound:
14         return (checkNum, "Valid")
15     elif checkNum < lowerBound:
16         return (lowerBound, "Adjusted")
17     else:
18         return (upperBound, "Adjusted")
19
20 # Date Validator
21 def checkDate(valueToCheck, min_date, max_date):
22     # Date-specific validation logic here
23     pass
24
25 # String Validator
26 def checkString(valueToCheck, pattern):
27     pass
28
29 def usageFunction():
30     # Demonstrating integer validation
31     print(validate_value("100", checkInteger, 1, 200))
32     print(validate_value("300", checkInteger, 1, 200))
33
34 if __name__ == "__main__":
35     usageFunction()
36
```

3. Use pseudo-code to show how an automated mission-critical system would manage the exception of a “less-than-perfectly-reliable” value. Alert the humans if automated processing is too risky

- **Reliability Check:** The function `check_reliability` simulates evaluating the reliability of the value and produces an **index ranging from 0 (completely unreliable) to 1**. This function would carry out particular checks to assess dependability in a real-world situation.

- **Automated Processing Decision:** `process_critical_value` determines whether to process the value automatically, warn and proceed, or notify human operators based on the reliability score.
- **High reliability** ( $\geq 0.8$ ) results in processing that is easily automated. A warning is generated by moderate reliability ( $0.5 \leq \text{reliability} < 0.8$ ), but automated processing is still carried out—possibly with extra security.
- **Reduced dependability** (less than 0.5) necessitates human examination, alerting operators.
- **Logging and Alerting:** The `log` and `alert_human_operator` functions stand for human alerting and logging, which are essential for mission-critical system documentation and intervention.



The screenshot shows a code editor with a dark theme. The Explorer panel on the left shows a project named 'HW3' with three files: 'mission\_critical\_val...', 'pseudo-code.py', and 'reusableFunction.py'. The main editor window displays the file 'mission\_critical\_value\_processing.py'. The code defines several functions: `process_critical_value`, `check_reliability`, `process_automatically`, `alert_human_operator`, and `log`. The `process_critical_value` function uses a conditional structure to handle different reliability levels. The `check_reliability` function returns a score of 0.9 as an example. The `process_automatically` function logs the processing of a value. The `alert_human_operator` function logs an alert message. The `log` function prints a message to the console. At the bottom, an example of processing a value is shown, where `get_data()` is used to fetch a value and `process_critical_value` is called with that value.

```

1  from pkgutil import get_data
2
3
4  def process_critical_value(value):
5      # Assess the reliability of the value
6      reliability = check_reliability(value)
7
8      # Based on reliability, decide on the course of action
9      if reliability >= 0.8:
10         # If reliability is high, proceed with automated processing
11         process_automatically(value)
12     elif 0.5 <= reliability < 0.8:
13         # For moderate reliability, log a warning and proceed with caution
14         log("WARNING: Value reliability is moderate, proceeding with caution.")
15         process_automatically(value)
16     else:
17         # If reliability is low, alert human operators
18         alert_human_operator(value, reliability)
19
20 def check_reliability(value):
21     # reliability checking logic
22     # Return a reliability score between 0 and 1
23     return 0.9 # Example: A high reliability score
24
25 def process_automatically(value):
26     # the logic to process the value automatically
27     log(f"Processing value {value} automatically.")
28
29 def alert_human_operator(value, reliability):
30     # alerting mechanism
31     log(f"ALERT: Value {value} with reliability {reliability} requires human review.")
32
33 def log(message):
34     # logging mechanism
35     print(message) # Example: Printing to the console
36
37 # Example of processing a value
38 value = get_data() # Assume fetch_data gets the value to process
39 process_critical_value(value)
40

```

## Automated mission-critical system factors :

1. **Evaluate Reliability:** The system begins by evaluating the input value's reliability. This could be accomplished in a number of ways, including by validating the information, comparing it to established patterns, or examining the source.
2. **Making Decisions:** The system makes decisions about whether to continue with automated processing, stop and notify a human, or issue a warning and continue based on the reliability score.
3. **Automated Processing:** The value is processed automatically by the system if it is considered reliable enough. This could include any processing related to the mission-critical system, including calculations and data transformation.
4. **Logging and Alerts:** The system logs a warning for values that are not entirely dependable. It notifies human operators if the reliability of the value is too low for safe automated processing. The alert system may consist of text messages, emails, or alarms that appear on a dashboard for monitoring.