COL1000: Introduction to Programming

Specifications & Debugging

Subodh Sharma | Lec 27 | Oct 21

Specifications (RECAP)

- Precise
- Observable
- Checkable
- Minimal + Complete

- UNIT TESTING
  - Typing hints, Function Signatures
  - Preconditions
  - Postconditions
  - State Invariants, Exceptions
- Mypy &

Clarification: Assertion vs Raise

- Assertion violation raises AssertionError exception object!
- Assertion can be disabled; but Raise can't be!

Conclusions: Specifications

- For complex problems, specification and solution go hand in hand
  - Problems are divided into parts, and each part is separately specified
  - Usually the solution of one part becomes the given input for another part
- Knowing the nature of problem can help specify problems and guide solutions
  - Decision Problems (Is there any solution with the given property?)
  - Search Problems (Find one or more solutions among all candidates with the property)
  - Counting Problems (number of candidates with the property)
  - Optimization Problems (best solution satisfying the property in terms of some metric) .

Conclusions: Specifications

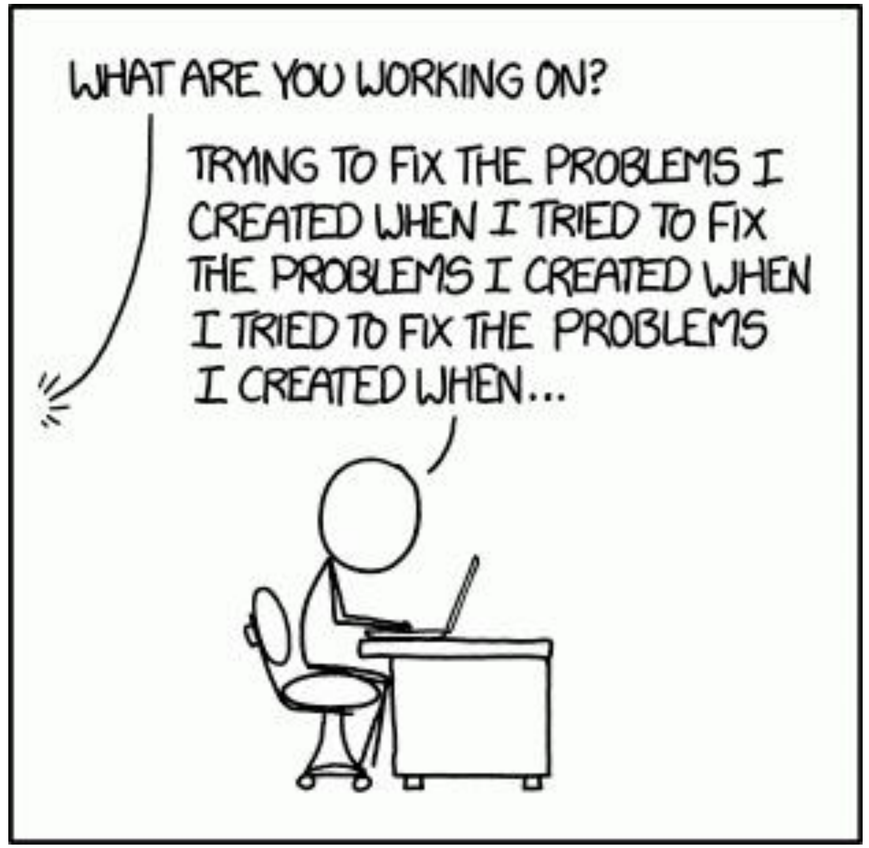
- Ensure that the input and output are clear: constraints the **input data type, size, or range**
- Establish that output for every possible input can be verified to meet the specification
  - If multiple correct solutions for the same input are possible, see if all must be produced
- Model the problem in mathematical terms, with numbers and symbols
  - Use only standard notation and commonly understood axioms

Conclusions: Specifications

- Provide sample examples of input/output
  - **How do we know the samples are ‘good’ (as in representative enough!)**
- Solution is a sample from a probability distribution
  - Or, satisfies a specified property with a high probability
- Usually divided into concretely specified sub-problems
  - e.g., Apply statistical methods on data

Debugging

"His debugging skills are exceptional."



Built-in Debuggers

pbd

- PBD — An interactive source-code debugger for Python programs
  - Allows one to set **breakpoints**
    - **Breakpoints —** points where execution will pause. Once paused, you can do the following:
      - Inspect values of variables
      - Execute code line by line
      - Step in and out of functions
      - See the full call stack

Built-in Debuggers

pbd

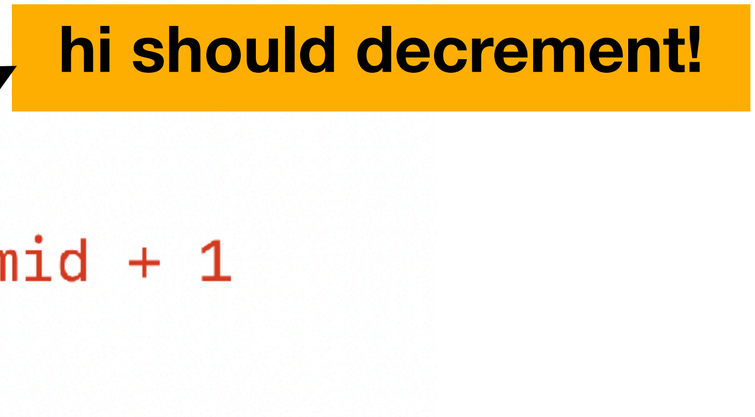
- Key Commands in PBD

Command	Alias	Description
`next`	`n`	Execute the current line and move to the **next line** in the same function.
`step`	`s`	Execute the current line and **step into** any function that is called.
`continue`	`c`	**Continue** execution until the script finishes or hits another breakpoint.
`list`	`l`	Show the source **code** around the current line.
`print <expr>`	`p`	**Print** the value of a variable or expression (e.g., `p my_variable`).
`where`	`w`	Show the **call stack** to see which function called which.
`return`	`r`	Continue execution until the current function **returns**.
`quit`	`q`	**Exit** the debugger and terminate the script immediately.
`args`	`a`	Print the **arguments** of the current function.

Debugging Buggy BinSearch

```
def binary_search(a: list[int], x: int) -> int:
    lo, hi = 0, len(a) - 1
    while lo <= hi: #Bug 1: should be <=
        mid = (lo + hi) // 2
        breakpoint()
        if a[mid] < x:
            lo = mid #Bug 2: should be mid + 1
        else:
            hi = mid
    return lo if lo < len(a) and a[lo] == x else -1

if __name__ == "__main__":
    print(binary_search([1, 3, 5, 7, 9], 8))
```



Debugging Recursive Merge

```
def merge_recursive(L, R):
    if not L:
        return R[:]
    if not R:
        return L[:]

    vL, vR = L[0], R[0]
    #breakpoint()
    if vL < vR:
        return [L[0]] + merge_recursive(L[1:], R)
    elif vL > vR:
        return [R[0]] + merge_recursive(L, R[1:])
    else: # when vL = vR
        return [L[0]] + merge_recursive(L[:], R)

if __name__ == "__main__":
    print(merge_recursive([1, 2, 2, 3], [2, 2, 4]))
```