Beyond the Textbook (Zelle 3e - Chapter 6)

Python Functions

Why Use Functions

- The naïve structure for a program is to put all code into `main()`.
- The better practice for larger programs is to break the code up into `main()` and an appropriate number of sub-functions.
- `main()` becomes the orchestrating function.
- This approach leads to code that is:
  - Easier to write.
  - Easier to read.
  - Easier to test.
  - Easier to debug.
  - Easier to modify.

Refactoring (and Factoring)

- **Refactoring** is changing a program that has already been written to make it more readable, testable, or maintainable without changing its function. See Fowler.
- **Factoring** would be organizing your code as you write it so that it is easy to read, easy to test, and easy to maintain.
- Most people don't use the term *factoring*. Instead, they just call it **program design**.
- Refactoring often calls for removing code from a function and creating a sub-function (**method extraction**).
- Sometimes, refactoring calls for the opposite (**method inlining**).
- Refactoring principles are as important when you are writing the original code as they are when you are improving it after the fact.

Design Principles For Functions

- The functions in well-designed programs should have **high cohesion** and **low coupling**.
- Functions that have high cohesion are all about the same thing. See Wikipedia article.
- One sign of high cohesion is that the function is easy to name.
- Names for Python functions should be *verb-noun phrases* like `print_heading`.
- Functions that have low coupling know as little as possible about the inner workings of other functions. See Wikipedia article.
- Functions that have low coupling only know about the interface of the called function: the name, the expected arguments, and the expected return values (if any).

Factoring Code Into Sub-Functions

4 Common Use Cases

Use Case 1: Bags of Code

- There is too much code to fit into the higher-level function.
- Good function size is about a page or less.
- In this use case, code is divided up into multiple functions that meet size requirements.
- See:
  - *_01_bags_of_code_original.py*
  - *_02_bags_of_code_refactored.py*

Use Case 2: Reusable Parts (DRY)

- There is common code that appears 2 or more times.
- Good practice is to factor this code into a common function that can be called from wherever the common code appears.
- This saves authoring time, testing time, debugging time, and maintenance time.
- Don't repeat yourself (**DRY**).
- See:
  - *_03_reusable_parts_original.py*
  - *_04_reusable_parts_refactored.py*

Use Case 3: Parameterized Tools (DRY)

- There are multiple pieces of code that are not exactly alike.
- But, they are sufficiently similar to be addressed by a function that takes parameters.
- This saves authoring time, testing time, debugging time, and maintenance time.
- Don't repeat yourself (**DRY**).
- See:
  - *_05_parameterized_tools_original.py*
  - *_06_parameterized_tools_refactored.py*

Use Case 4: Hiding Places For Assumptions

- There are multiple pieces of code that do similar work.
- Replacing them with function calls **may not save** any lines of code.
- Yet, we **do replace** these code pieces with calls to a common function.
- Our rationale is to isolate some assumption regarding how we solve the problem to just one function.
- This saves maintenance time should the assumption change in the future.
- See:
  - *_07_hiding_places_original.py*
  - *_08_hiding_places_refactored.py*

All Python Functions Return a Single Value

- If there is no `return` statement, the `None` value is returned.
- If the `return` statement returns one value, then that value is returned.
- If the `return` statement appears to return more than one value, then a tuple is created and the tuple is returned.
- Note that this is not an excuse to design functions that lack cohesion. All values in the returned tuple should be about the same thing.
- See:
  - *_09_use_function_returns_more_than_one_value.py*
  - *_10_create_function_returns_more_than_one_value.py*

Positional vs Keyword Parameters

- With positional parameters, arguments in the calling code are matched up with formal parameters in the called code by position.
- Python also has **keyword parameters**. These are matched by name.
- Positional parameters must be coded before keyword parameters.
- Keyword parameters provide **default values**.
- See:
  - *_12_use_function_takes_keyword_parameter.py*
  - *_15_create_function_takes_keyword_parameter.py*

Resources

- Refactoring home page. Refactoring. (n.d.). Retrieved September 11, 2022, from https://refactoring.com/
- Wikimedia Foundation. (2022, August 29). Cohesion (Computer Science). Wikipedia. Retrieved September 11, 2022, from https://en.wikipedia.org/wiki/Cohesion_(computer_science)
- Wikimedia Foundation. (2022, May 6). Coupling (computer programming). Wikipedia. Retrieved September 11, 2022, from https://en.wikipedia.org/wiki/Coupling_(computer_programming)

Last Revised 2022-09-11