**Guidelines for Code Refactoring**

In the realm of software development, code refactoring serves as a fundamental practice to enhance the maintainability, readability, and overall quality of our codebase. As we embark on this journey of refining our software, it is crucial to establish clear and comprehensive guidelines for code refactoring. These guidelines will not only streamline the process but also ensure that the team adopts a consistent and collaborative approach, fostering a codebase that is resilient to change and conducive to long-term success. Here are the guidelines to adhere to when undertaking code refactoring endeavors:

1. **Indentation**  
   Indentation refers to the spaces at the beginning of a code line. While indentation in other programming languages is only a resource for readability, indentation in Python is mandatory. Python uses indentation to open a block of code. More precisely, 4 consecutive spaces/1 tab per indentation level (We will prefer 1 tab instead of 4 spaces), as shown in the following code:

for number in [1,2,3,4]:

if number % 2 == 0:

print(number)

else:

continue

1. **Follow the PEP 8 Style Guide for Python Code**  
   The PEP 8 style guide for Python Code, also known as PEP8 or PEP-8, is a comprehensive guide that provides reminders on how to write impeccable Python code. Published in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan, this manual has become the holy book for writing Python code that’s readable and consistent. This style guide makes practicing Python coding effortless.
2. **Document and Comment Your Code Properly**  
   A critical part of Python best practices is documenting and commenting on all code properly. This point is significant if multiple parties are working on the same piece of code at any time. Without clear documentation and concise comments, confusion can set in and leave everyone scratching their heads as they try to contribute their code.
3. **Python commenting best practices**  
   Comments are very important to make annotations for future readers of our code. Although it is difficult to define how the code should be commented on, there are certain guidelines that we can follow:

* Any comment that contradicts the code is worse than no comment. That is why it is very important that we update the code, and do not forget to update the comments to avoid creating inconsistencies.
* Comments must be complete sentences, with the first letter capitalized.
* Try to write comments in English. Although everyone is free to write their comments in the language they consider appropriate, it is recommended to do so in English.
* Ensure that your comments are clear and easily understandable to other speakers of the language you are writing in.
* There are two types of comments: block comments and inline comments.
  + A block comment explains the code that follows it. Typically, you indent a block comment at the same level as the code block. Each line of a block comment starts with a # and a single space, as follows:

# This is a block comment  
print('Welcome to Course5!")

* + Inline comments appear at the same level as code. Inline comments should be separated by at least two spaces from the statement. Like a block comment, an inline comment begins with a single hash sign (#) and is followed by a space and a text string.

1. **Write Readable Code**  
   It’s vital that when it comes to Python coding, every line should be easily readable. Ultimately, each piece of code is highly focused, giving a clear function, class, or module to that line of code. Readable code results in teams being able to contribute their code without unnecessary delays and a beautifully designed final project.
2. **Write Object-Oriented Code**  
   Object-Oriented Programming (OOP) is the ingredient that properly structures software design around different data points or objects. By writing OOP, any pieces of code written become reusable for future projects and stop the possibility of redundancies, causing significant project delays.
3. **Use One Statement of Code per Line**  
   A typical saying in Python programming is “do a lot with a little,” which every developer should remember when writing code. A common practice when writing Python code is to write a single piece of code per line, with the new line marking the end of the first piece of code. Ultimately, this means that every part of Python code is way more potent and prevents unnecessary code from being written.
4. **Use a Proper Naming Convention**When writing Python code, using a proper naming method can save time. With different variables, functions, classes, packages, and more, picking a logical naming convention will help streamline a project timeline and make it easier for team members to add their code.
   1. **Variables & Functions:** use all lowercase letters and separate words with underscores.
   2. **Classes:** use CapWords/CamelCase, using a capital letter for each new word but not separating the words by spaces or underscores.
   3. **Constants:** use all caps.
   4. **Modules**: use all lowercase. Add underscores between words if it improves readability.
   5. **Methods:** use the function naming rules but indicate internal use methods by adding an underscore before the name.
   6. **Single-Character Names:** avoid using these altogether.
   7. **Built-In Names:** avoid using the Python built-in names, such as naming a variable “list.”
5. **Blank lines**  
   Surround top-level function and class definitions with two blank lines. Method definitions inside a class are surrounded by a single blank line. Extra blank lines may be used (sparingly) to separate groups of related functions. Finally, use blank lines in functions, (sparingly) to indicate logical sections.
6. **Python docstring best practices**A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Typically, you use a documentation string to automatically generate the code documentation. As such, a docstring can be accessed at run-time using the obj.\_\_doc\_\_ special attribute of that object.

For consistency, docstrings are always enclosed in triple double quotes (""").

There are two forms of docstrings: one-liners and multi-line docstrings. One-liners are for really obvious cases. They should really fit on one line. The following example illustrates a one-line docstring in the multiply() function:

def multiply(x, y):

""" Return the product of two numbers """

result = x \* y

return result

On the other hand, a multi-line docstring can span multiple lines. Such a docstring should document the script’s function and command line syntax, environment variables, and files. Usage messages can be fairly elaborate and should be sufficient for a new user to use the command properly.

def calculate\_salary(hours, price=20):

""" Return the salary according to the hours worked

Arguments:

hours: total hours invested

price: price of each hours worked. Minimum price is 20 dollars

"""

salary = hours \* price

return salary

1. **Adhere to SOLID Principles:**  
   Ensure that refactored code aligns with the SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) to maintain modularity, flexibility, and extensibility.
2. **Follow the DRY Principle (Don't Repeat Yourself):**

Eliminate duplicate code by creating reusable abstractions, such as functions, methods, or classes, to promote maintainability and reduce the risk of inconsistencies.

1. **Encapsulate Responsibilities:**Each class or module should have a clear and singular responsibility. Refactor code to encapsulate specific functionalities within well-defined classes, promoting a modular and organized codebase.
2. **Use Descriptive Naming Conventions:**Choose meaningful and self-explanatory names for classes, methods, and variables to enhance code readability. Avoid ambiguous or overly concise names that may lead to confusion.
3. **Ensure Proper Inheritance Usage:**Review class hierarchies to ensure proper use of inheritance. Refactor if necessary, favoring composition over inheritance when appropriate to avoid tight coupling.
4. **Apply Design Patterns:**Identify and implement relevant design patterns to solve common problems in a consistent and proven way. This can enhance the maintainability and scalability of the codebase.
5. **Unit Test Refactored Code:**Write or update unit tests to validate the correctness of the refactored code. Ensure that existing functionality is preserved and that the refactoring process does not introduce new bugs.
6. **Refactor Large Methods:**Break down large methods into smaller, more focused ones. This not only improves readability but also allows for better code organization and easier maintenance.
7. **Continuous Refactoring:**  
   Incorporate refactoring as an ongoing process rather than a one-time activity. Regularly review and improve the codebase to prevent the accumulation of technical debt.