Python Style Guide

- TLDR: Run pylint over your code [https://google.github.io/styleguide/pylintrc]
- **Imports**:
  - Use *import x* for importing packages and modules.
  - Use *from x import y* where x is the package prefix and y is the module name with no prefix.
  - Use *from x import y as z* if two modules named y are to be imported, if y conflicts with a top-level name defined in the current module, or if y is an inconveniently long name.
  - Use import y as z only when z is a standard abbreviation (e.g., np for numpy).
- **Packages**:
  - Avoids conflicts in module names or incorrect imports due to the module search path not being what the author expected. Makes it easier to find modules.
- **Exceptions**:
  - Make use of built-in exception classes when it makes sense.
  - Exception names should end in Error and should not introduce repetition (foo.FooError).
  - Never use catch-all except unless you are reraising the exception or you are creating an isolation point where exceptions are not propagated but are recorded and supressed instead.
  - Minimize the amount of try/except block.
  - Use finally to execute code whether or not an exception is raised in try block (for clean-up)
- **Global Variables:**
  - Try avoiding global variables.
  - If needed, global variables should be declared at the module level and made internal to the module by prepending an _ to the name
  - While module-level constants are technically variables, they are permitted and encouraged. For example: MAX_HOLY_HANDGRENADE_COUNT = 3. Constants must be named using all caps with underscores.

- **Nested/Local/Inner Classes and Functions**:
    - They are fine with some caveats. Avoid nested functions or classes except when closing over a local value other than self or cls. Do not nest a function just to hide it from users of a module. Instead, prefix its name with an _ at the module level so that it can still be accessed by tests
- **Comprehensions and Generator Expressions**:
    - Okay to use for simple cases. Each portion must fit on one line: mapping expression, for clause, filter expression. Multiple for clauses or filter expressions are not permitted. Use loops instead when things get more complicated.
- **Default Iterators and Operators**:
    - Use default iterators and operators for types that support them, like lists, dictionaries, and files.
        - Yes: `for key in adict:`
        - No: `for key in adict.keys()`
- **Generators**:
    - Use generators as needed.
    - Simpler code, because the state of local variables and control flow are preserved for each call. A generator uses less memory than a function that creates an entire list of values at once.
- **Lambda Functions**:
    - Okay for one-line functions
    - Okay to use them for one-liners. If the code inside the lambda function is longer than 60-80 chars, it's probably better to define it as a regular nested function.
- **Conditional Expressions:**
    - Okay for simple cases
- **Default Argument Values:**
    - Okay to use with following caveat:
        - Do not use mutable objects as default values in the function or method definition.

- **Properties:**
  - Properties are allowed, but, like operator overloading, should only be used when necessary and match the expectations of typical attribute access; follow the getters and setters rules otherwise.
  - Properties should be created with the @property decorator.
  - Do not use properties to implement computations a subclass may ever want to override and extend
- **True/False Evaluations**:
  - Use the "implicit" false, if possible,
    - e.g., if foo: rather than if foo != []
  - Always use if foo is None: (or is not None) to check for a None value.
  - Never compare a Boolean variable to False using ==. Use if not x: instead.
  - For sequences (strings, lists, tuples), use the fact that empty sequences are false, so if seq: and if not seq: are preferable to if len(seq): and if not len(seq): respectively.
  - When handling integers, implicit false may involve more risk than benefit (i.e., accidentally handling None as 0).
- **Lexical Scoping:**
  - A nested Python function can refer to variables defined in enclosing functions, but cannot assign to them. Variable bindings are resolved using lexical scoping, that is, based on the static program text.
  - An example of the use of this feature is:
    - ```
      def get_adder(summand1: float) -> Callable[[float], float]:
          """Returns a function that adds numbers to a given number."""
          def adder(summand2: float) -> float:
              return summand1 + summand2

          return adder
      ```
  - Can lead to confusing bugs. Such as this example based on PEP-0227
      ```

- **Function and Method Decorators:**
  - Use decorators judiciously when there is a clear advantage. Avoid static-method and limit use of class-method.
  - Avoid external dependencies in the decorator itself (e.g., don't rely on files, sockets, database connections, etc.),
  - Decorators are a special case of "top level code"
  - Never use static-method unless forced to in order to integrate with an API defined in an existing library. Write a module level function instead
  - Use class-method only when writing a named constructor or a class-specific routine that modifies necessary global state such as a process-wide cache.
- **Threading:**
  - Do not rely on the atomicity of built-in types.
  - Use the Queue module's Queue data type as the preferred way to communicate data between threads. Otherwise, use the threading module and its locking primitives. Prefer condition variables and threading. Condition instead of using lower-level locks.
- **Power Features:**
  - Avoid These
  - Standard library modules and classes that internally use these features are okay to use (for example, abc. ABCMeta, dataclasses, and Enum).
- **Modern Python:**
  - Use of from __future__ import statements is encouraged. It allows a given source file to start using more modern Python syntax features today. Once you no longer need to run on a version where the features are hidden behind a __future__ import, feel free to remove those lines.
- **Type Annotated Code:**
  - You can annotate Python code with type hints according to PEP-484, and type-check the code at build time with a type checking tool like pytype.
  - You are strongly encouraged to enable Python type analysis when updating code. When adding or modifying public APIs, include

type annotations and enable checking via pytype in the build system.

- **Python Style Rules:**
  - Do not terminate your lines with semicolons, and do not use semicolons to put two statements on the same line.
  - Maximum line length is *80 characters*.
    - Long import statements.
    - URLs, pathnames, or long flags in comments.
    - Long string module level constants not containing whitespace that would be inconvenient to split across lines such as URLs or pathnames.
    - Do not use backslash line continuation except for with statements requiring three or more context managers.
    - Use parentheses sparingly. Do not use them in return statements or conditional statements unless using parentheses for implied line continuation or to indicate a tuple.
    - Indent your code blocks with *4 spaces*.
    - Trailing commas in sequences of items are recommended only when the closing container token],), or} does not appear on the same line as the final element.
    - Two blank lines between top-level definitions, be they function or class definitions. One blank line between method definitions and between the class line and the first method. No blank line following a def line. Use single blank lines as you judge appropriate within functions or methods.
    - No whitespace inside parentheses, brackets or braces.
    - Never use spaces around = when passing keyword arguments or defining a default parameter value, with one exception: when a type annotation is present, *do* use spaces around the = for the default parameter value.
    - Most .py files do not need to start with a #! line. Start the main file of a program with #!/usr/bin/env python3 (to support virtualenvs) or #!/usr/bin/python3 per PEP-394.
    - Be sure to use the right style for module, function, method docstrings and inline comments.

        - A function must have a docstring, unless it meets all of the following criteria:
            - not externally visible
            - very short
            - Obvious
        - Classes should have a docstring below the class definition describing the class. If your class has public attributes, they should be documented here in an Attributes section and follow the same formatting as a function's Args section.
        - All class docstrings should start with a one-line summary that describes what the class instance represents.
        - The final place to have comments is in tricky parts of the code. If you're going to have to explain it at the next code review, you should comment it now. Complicated operations get a few lines of comments before the operations commence. Non-obvious ones get comments at the end of the line.
        - Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.
        - Use an f-string, the % operator, or the format method for formatting strings, even when the parameters are all strings. Use your best judgment to decide between + and string formatting.
        - Be consistent with your choice of string quote character within a file.
- **Logging:**
    - For logging functions that expect a pattern-string (with %-placeholders) as their first argument: Always call them with a string literal (not an f-string!) as their first argument with pattern-parameters as subsequent arguments.

- **Error Message:**
  - The messages need to precisely match the actual error condition.
  - Interpolated pieces need to always be clearly identifiable as such
  - They should allow simple automated processing (e.g., grepping)
- **Files Sockets and Similar Stateful Resources:**
  - Explicitly close files and sockets when done with them.
  - The preferred way to manage files and similar resources is using with statement.
- **TODOs:**
  - Use TODO comments for code that is temporary, a short-term solution, or good-enough but not perfect.
- **Getters and Setters:**
  - Getter and setter functions (also called accessors and mutators) should be used when they provide a meaningful role or behaviour for getting or setting a variable's value.
  - In particular, they should be used when getting or setting the variable is complex or the cost is significant, either currently or in a reasonable future.
  - Should follow the Naming guidelines such as get_foo() and set_foo()
- **Naming:**
  - Function names, variable names, and filenames should be descriptive; eschew abbreviation. In particular, do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

- **Main:**
    - In Python, pydoc as well as unit tests require modules to be importable. If a file is meant to be used as an executable, its main functionality should be in a main() function, and your code should always check if __name__ == '__main__' before executing your main program, so that it is not executed when the module is imported.

- **Function length:**
    - Prefer Small and Focused functions.
    - We recognize that long functions are sometimes appropriate, so no hard limit is placed on function length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

- **BE CONSISTENT**