# Analytics Products Coding Standards

|  |  |  |
| --- | --- | --- |
| **ID** | **Description** | **Purpose** |
| C1 | Develop code to be correct first, perform and scale second, and be maintainable third. (But make coding maintainable code a habit in general.) | Separate concerns to reduce task complexity, which on the whole improves velocity and quality. |
| C2 | Build a standard API into each tool. The API should be robust enough to support automated testing, and to let sophisticated customers invoke the tool through the API. | Support test automation, and support sophisticated end users wanting to invoke our tools from outside the Alteryx UI. |
| C3 | All magic numbers and string shall be managed via the UI, API, and/or configuration files or (database) tables. | Avoid hard-coding model design decisions. |
| C4 | Don’t break programming interfaces.   1. Don’t rename arguments, classes, functions, etc. 2. Don’t change the semantics of inputs, outputs, etc. 3. Don’t change a function or module’s behavior in a way that reverses a pre-existing behavior, unless the pre-existing behavior is provably defective.   Instead, extend interfaces or create new ones.   1. Create a wrapper for a pre-existing function, class, etc., with the wrapper having the naming you want. 2. Add new inputs or outputs having the semantics you want. 3. Add behavior by adding optional arguments and features, or by writing new functions, modules, classes, etc.   Every time you touch a pre-existing piece of code, ask yourself if you can imagine a way that the changes you plan to make could result in a dependent program or dataflow change its behavior.  *The only changes of this kind that are acceptable, within AP, are changes that measurably improve model quality without otherwise affecting the behavior of dependent code or dataflows.* Ideally, the measurable improvement should be Pareto optimal, in the sense that the changes should improve one or more model-quality metrics on some test cases without degrading any model-quality metrics on any test cases.  If the change does not achieve Pareto optimality, we should study the change carefully before deciding to release it to a customer.  Unfortunately, not all of the code others write honors this norm.  So we sometimes must undertake defensive programming to account for changes to tools we use.  In such cases the best practice is:  When a tool we code against changes its interface or behavior across versions in a way that breaks our code, create a wrapper for the tool that has a consistent interface and behavior across versions.  (Within the wrapper, branch on the tool version to handle the change.)  Then, code to the wrapper instead of the tool proper.  (If the tool’s author persists in breaking interfaces, consider replacing the tool to eliminate the dependency on an unreliable tool author!) |  |
| C5 | [How do we handle the question of using unofficial R and Python libraries (e.g. R packages on GitHub that are not on CRAN)?] |  |
| C6 | [How do we handle using open-source code?] |  |
| C7 | Model components such as transformations, variable-selection methods, null-replacement techniques, fitness functions, fitting algorithms, and model-quality metrics shall be designed and implemented to support re-use, and shall be re-used, wherever possible. |  |
| C8 | Drafting technical writing is a development activity on par with writing code. Technical writing must be drafted before code review can occur, and code review must include a writing review. |  |
| C9 | When you work around a bug, please put a comment above the workaround, like this:  # WORKAROUND  # The assign() statement below moves the token ‘some\_token’ to the global environment, where the lm() function can find it.  # Otherwise, something inside lm() isn’t finding ‘some\_token’ on its environment search path.  assign(x = ‘some\_token’, blah blah blah)  lm(blah blah blah , some\_token)  Note that the first line in the comment is ‘# WORKAROUND’. This will let us find all of our workarounds when doing PI planning, to try to eliminate them and the technical debt they represent. |  |
| C9 | Here’s a sample help instruction for a *notional* select-list input in a *notional* OLS linear-regression tool:  Select the variable you want to predict. This variable is often termed the dependent, target, response, outcome, or predicted variable. (See <https://en.wikipedia.org/wiki/Dependent_and_independent_variables#Statistics_synonyms> for more synonyms. See <https://en.wikipedia.org/wiki/Linear_regression#Introduction> for an explanation of this variable’s role in ordinary least squares (OLS) linear regression. See <https://stat.ethz.ch/R-manual/R-devel/library/stats/html/lm.html> for a description of the R lm() function called by this tool. The present input maps to the response variable in the formula passed to the lm() function’s formula argument.)  The structure is,   1. a terse imperative telling the user what to do with the field 2. another terse sentence explaining jargon or defining terms 3. pointers to relevant online literature, especially Wikipedia and R function/package documentation; and verbiage identifying the input with its R counterpart.   Note that the help entry does not try to explain what a linear model is or how it works. Instead, it points the reader to relevant online explanations. |  |