Coding Standards and Guidelines for the Gemini Data Processing Software

Kathleen Labrie, Craig Allen, and Emma Hogan

Science Users Support Department

V2.1 – 03 October 2018

Revision History

V1.0 – 27 March 2013	Kathleen Labrie
V1.1 – 13 June 2014	Kathleen Labrie
V1.2 – 28 February 2015	Kathleen Labrie
V2.0 – 25 January 2018	Kathleen Labrie
V2.1 – 03 October 2018	Kathleen Labrie

Document ID: DPSG-STD-102_CodingStandards

Document Purpose

This document defines the coding standards and guidelines to apply to the implementation of the Gemini data processing software suite.

Intended Audience

This document is intended primarily to the developers of the Gemini data processing software, whether they are on the Data Processing Software Group or not. This document is to be used by anyone reviewing code against the standards.

Table of Contents

1. Policy

The guidelines stated in this document are to be adhered to when developing software for the Gemini data processing software suite. Code will be reviewed against these guidelines. The objectives are to ensure quality, robustness, maintainability, and re-usability.

2. References

DPSG-STD-101_ConfMgmtNomenclature.docx
PIPE-TEMP-101_Primitive.py
PIPE-USER-102_DatasetNameSuffixes.docx

3. Definitions

3.1 Guidelines Priority Definitions

Each guideline is given a priority level. The levels are:

1. CRITICAL
2. INTERFACE
3. STANDARD
4. SUGGESTED
5. INFORMATIVE

3.1.1 CRITICAL

CRITICAL guidelines are those that are crucial to the functioning of the software. If they are not followed the software will not work. For example, if a recipe filename does not respect the format "recipe.*", it will not be recognized as a recipe by the Recipe System.

3.1.2 INTERFACE

Guidelines marked as INTERFACE are generally those that are related to interfaces or user/programmer experience. Those guidelines must be adhered to during development.

3.1.3 STANDARD

Guidelines marked STANDARD priority are required and often related to the perceived cleanliness of the code. They express a general team consensus on best practices. Deviation from those guidelines need to be justified and approved. (KL: what is the process for such approval?)

3.1.4 SUGGESTED

SUGGESTED guidelines are for points that are left to the individual discretion of the developer but upon which the team has agreed are often arguably best practices, though there may be exceptions and application depends on the case.

3.1.5 INFORMATIVE

Guidelines labeled INFORMATIVE are suggestions to help convey some regular practices that can be valuable but are not critical in any way, for examples practices that promote forward-compatibility.

4. Codebase Management Standards

The Codebase Management standards are defined in the document DPSG-STD-103_CodebaseMgmt. The document provides information about revision control for both the Python codebase and the IRAF codebase. [KL. see http://gdpsg.wikis-internal.gemini.edu/index.php/GDPSG-CodebaseManagement when it's time to write that document.]

5. Code Review Guidelines

The Code Review process and guidelines are defined in the document DPSG-PROCESS-101_CodeReview. The document provides information about the whole code review process, from objectives of the reviews to their duration, for example. [KL. see http://gdpsg.wikis-internal.gemini.edu/index.php/GDPSG-CodebaseManagement when it's time to write that document.]

6. Glossary – GP-Py-Glos

GP-Py-Glos-1 **[STANDARD]**	**DRAGONS**
	The name of the Python data reduction software suite. It contains Astrodata (astrodata package) and the RecipeSystem (recipe_system package), as well as the official Gemini add-ons, gemini_instruments and geminidr, and finally the gempy package.
GP-Py-Glos-2 **[STANDARD]**	**AstroData**
	The base class for all the data specific AstroData classes.

GP-Py-Glos-3	**Astrodata Tag**
[STANDARD]	A tag defined in an Astrodata add-on package. A set of tags describes the type of data loaded in the AstroData object.
GP-Py-Glos-4	**Descriptor**
[STANDARD]	A metadata concept appropriate for any dataset.
GP-Py-Glos-5	**Astrodata Add-on**
[STANDARD]	A package containing the AstroData subclasses, the tags, and the descriptors for one or more instruments. Eg. gemini_instruments, octocam_instrument.
GP-Py-Glos-6	**RecipeSystem Add-on**
[STANDARD]	A package containing the recipes, primitives and data reduction algorithms for one or more instruments. Eg. geminidr, octocamdr.
GP-Py-Glos-7	**Primitive**
[STANDARD]	A step that performs a "scientifically meaningful" transformation of the dataset or a measurement. A primitive is a special function that can be called in a Recipe.
GP-Py-Glos-8	**Recipe**
[STANDARD]	A list of sequential instructions formed of the names of primitives and other recipes. A recipe is run by the Recipe System.
GP-Py-Glos-9	**"scientifically meaningful"**
[STANDARD]	This term describes the most fine-grained unit of transformation a primitive should perform.
	This term is of course difficult to define in detail. However, we use it as it gives a direction to some of the statements of goals in the Gemini data processing software. The objective is to avoid creating primitives that are too fine-grained as we aimed to have recipes that can be read as a story rather than as code.
GP-Py-Glos-10	**Recipe System**
[STANDARD]	The system that runs the recipes. It is part of the DRAGONS package and requires add-on's do to any data processing. The Recipe System is the automation layer of the pipeline.

7. Guidelines for the DRAGONS Codebase

ID [Priority]	Name
	Statement
	Note

7.1 Guiding Principles for the Python Codebase

1) **PEP8**
   The "Style Guide for Python Code" (PEP8) shall be adopted, unless otherwise specified. (http://www.python.org/dev/peps/pep-0008/)
2) **Clarity**
   Source code and comments effectively communicates information to readers only to the extent that its contents are made comprehensible. Clarity is enhanced through writing comments in complete sentences consisting of correctly spelled words. Placing declarations and directives so that they are easy to find, using clean and simple logic, and using meaningful names.
3) **Visual Coherence**
   Software placing related information in close visual proximity, and visually separating unrelated information, is more readable than code which does not.
4) **Consistency**
   Requires that all software within a project conform to a single set of syntactic and stylistic conventions.
5) **Integrity**
   The most obvious form of integrity is physical integrity. Source files must be editable, viewable, and printable, using standard utilities available on a variety of platforms.
6) **Modularity**
   Maintenance of software is easier when the software is composed of pieces which may be rearranged and/or modified independently of each other.
7) **Efficiency**
   At Gemini we are data bound, therefore it is important to keep code as efficient as possible without compromising portability.

7.2 Verification

The developer should run pylint on the code during development and certainly before submitting the code for review. A pylintrc for the DRAGONS software is available in DRAGONS/gempy/support_files. To use, just copy pylintrc to $HOME/.pylintrc.

7.3 Python Source Code Formatting – GP-Py-Format

GP-Py-Format-1 [STANDARD]	**Line Length Rule**
	Line length shall be kept to 79 characters or less.
GP-Py-Format-2	**Line Wrapping**
	Long lines shall be wrapped using Python's implied line continuation inside

[SUGGESTION]	parentheses, brackets, and braces, and using a backslash elsewhere.
	Since newline and indentation is important in Python source code, it can be difficult to break lines up appropriately. It is preferred that it not be done with line continuation characters ("\"). Instead, new lines inside parentheses and reliance on the python string concatenation features should be relied upon if possible.
GP-Py-Format-3 **[STANDARD]**	**Indentation Rule**
	Indentation shall consist of four (4) spaces. Tabs are NOT allowed.
	It is therefore required that code only be edited with an editor that respects the uses of spaces (ie. does not convert whitespaces to tabs). Examples of editors known to be configurable: emacs, vi, PyCharm, and the eclipse IDE.
GP-Py-Format-4 **[SUGGESTION]**	**White Spaces**

	1) Inside parentheses, elements should be separated with one blank.
	spam(ham[1], {egg:2}, 'this string')
	2) In statements, put one blank after comma "," and semicolon ";".
	print(x, y); x, y = y, x
	3) In statements, equal signs "=" are flanked on both sides by a blank.
	if x == 4:
	4) In function argument lists, do not flank equal signs "=" with blanks.
	def boo(arg1=defa, arg2=0.0):
	5) In lists, do not use blanks when indexing or slicing.
	dict['key'] = list[index]

GP-Py-Format-5 **[SUGGESTION]**	**Import Statement**
	Imports should be on separate lines and should avoid using “*”.
	Instead of "import os, sys": import os import sys However, this is allowed: from numpy import shape, where, size Not allowed (see exception in GP-Py-Const-4): from numpy import *
GP-Py-Format-6 **[STANDARD]**	**Import Sorting Rule**
	Imports shall appear at the top of modules and be ordered as described in the notes below. Exception for modules being "lazy loaded" for a specific reason (to be documented in place with comments).
	Imports shall appear in the following order: 1. Major Python Packages, such as sys, and os 2. Standard Python Packages 3. Third Party Dependency Packages 4. DRAGONS Package imports 5. Utility Imports "Lazy loading" refers to putting an import statement inside code where the package is needed so rather than being loaded when the calling module is itself imported, it will be imported only when it is going to be needed.

7.4 Constants – GP-Py-Const

GP-Py-Const-1	**Constant Name**
[STANDARD]	Name constants in ALLCAPS.
	Python does not support the idea of actual constants. Instead, global scope variables are used, thus the need for clear identification of variables intended to be constants.
GP-Py-Const-2	**Respect for constants**
[STANDARD]	Do not change the value of a constant.
	Python does not support the idea of actual constants, and will let such variable be modified. Therefore, is important to never, under any circumstances, change a value of a variable we identify as a constant. If the value really needs changing, then it is not constant.
GP-Py-Const-3	**No hardcode**
[STANDARD]	Do not hardcode code statements.
	Instead of hardcoding, use constants properly, or use clearly assigned variables.
GP-Py-Const-4	**Groups of related constants**
[STANDARD]	A group of related constants shall be stored together in their own module and imported.
	"from xyz import *" is valid in this case.
GP-Py-Const-5	**Look-up tables for constant in configuration space**
[STANDARD]	Any constant associated with an with an AstroData tag or descriptor and used in more than once should be stored in a Python look-up table, in octocam_instrument/octocam/lookup.py. The look-up table should be a Python file, for example storing a Python dictionary or list. Similarly, the look-up tables used in the primitives should be Python files, dictionary or list, and stored in octocamdr/octocam/lookups/, the name of the file representing the content.
	See DRAGONS' gemini_instruments/gmos/lookup.py and geminidr/gmos/lookups/ for examples.
GP-Py-Const-6	**Constant definition location**
[STANDARD]	Constants appear at the top of modules, class, or function definitions.

7.5 Naming – GP-Py-Name

GP-Py-Name-1	**Variable names internal to a function, a module, a class, etc.**
[STANDARD]	Variable names shall start with a lowercase letter and use underscores ("_") to separate terms for readability.
	This is in accordance to PEP8. Normally, the variable name is all lower case, not just the first letter.
GP-Py-Name-2	**Meaning of variables**
[SUGGESTED]	Variables can be named however suits the developer so long as they are not misleading. Sensible terms or abbreviations, and consistency throughout the code is expected.
GP-Py-Name-3	**Constant names. See GP-Py-Const-1**
[STANDARD]	
GP-Py-Name-4	**Function names**
[STANDARD]	Function names shall be all lowercase letter and use underscores ("_") to separate terms for readability.
	This is in accordance to PEP8. (Primitives have a special status and thus a special naming convention; see GP-Py-Name-25.)
GP-Py-Name-5	**Class names**
[STANDARD]	Class names shall begin with an uppercase letter and use CamelCaps convention for additional words.
	This is in accordance to PEP8.
GP-Py-Name-6	**Module names**
[STANDARD]	Modules shall have short, all-lowercase names. Underscores can be used in the module name if it improves readability.
	This is in accordance to PEP8.
GP-Py-Name-7	**Public method names and instance variables**
[STANDARD]	Public method names and instance variables shall be all lowercase with words separated by underscores.
	This is in accordance to PEP8.
GP-Py-Name-8	**Private method names and instance variables**
[STANDARD]	Private method names and instance variables shall start with an underscore and be all lowercase with words separated by underscores.
	This is in accordance to PEP8.
GP-Py-Name-9	**Import alias for numpy**
[STANDARD]	The numpy package shall be imported as np.
	import numpy as np

GP-Py-Name-10 **[STANDARD]**	**Import for astropy.io.fits**
	The astropy fits package shall be imported as follow: from astropy.io import fits
	from astropy.io import fits
GP-Py-Name-11 **[STANDARD]**	**Import alias for matplotlib**
	The matplotlib.pyplot modules shall be imported as plt.
	import matplotlib.pyplot as plt
GP-Py-Name-12 **[INTERFACE]**	**Reserved argument names**
	The following are reserved argument names: **ad** : contains an AstroData object. **adinput** : AstroData object serving as input to function **pretty**: boolean argument for Descriptors. pretty is a flag that controls string output, indicating that a human readable string should be returned. **stripID** : boolean argument for Descriptors. stripID is a flag that controls string output, indicating that the component ID should be removed from the returned value.
GP-Py-Name-13 **[STANDARD]**	**Astrodata add-on package name**
	The name of the add-on package/directory should end with the string "_instrument", preceded by a string that uniquely identify the add-on.
	This is simply a standard we have adopted. Using the format for new instruments developed by third-party teams will simplify reviews and the future integration of the new package into the main gemini_instruments AstroData configuration package. For example, for OCTOCAM, the package should be named octocam_instrument.
GP-Py-Name-14 **[STANDARD]**	**Astrodata add-on configuration file name**
	The configuration file containing the AstroData tags and descriptor definitions is named adclass.py and is located in <adpkg>_instrument/<instrumentname>/.
	This is simply a standard we have adopted. Using that name and structure will help integration of third-party software (eg. Outsourced DR software) and help with long-term maintenance.
GP-Py-Name-15 **[CRITICAL]**	**Astrodata tag functions**
	AstroData tags are defined in the instrument's AstroData class and must be decorated with the @astro_data_tag decorator. The name of the function shall start with "_tag_" followed by an appropriately descriptive string.
	The decorator differentiates the tags from the descriptors. The tag functions are private functions of the class, the name reflects that. Eg. def _tag_dark(): Note that a tag function can return more than one tag.
GP-Py-Name-16 **[CRITICAL]**	**Astrodata descriptor functions**
	AstroData descriptors are defined in the instrument's AstroData class and must be decorated with the @astro_data_descriptor decorator. The name of the function is the name of the descriptor. See GP-Py-Name-18 DESCRIPTOR NAME.

	The decorator differentiates the tags from the descriptors. The descriptor functions are the access to the descriptor values (unlike the tags). The function name is the descriptor name.
GP-Py-Name-17 **[INTERFACE]**	**Astrodata tag name**
	An Astrodata tag name shall be written in uppercase, with words or abbreviations separated with hyphens.
	E.g. GMOS, IMAGE, LONGSLIT, IFU-B.
GP-Py-Name-18 **[INTERFACE]**	**Astrodata descriptor name**
	Descriptor names shall be all lowercase with terms separated with underscores. Common abbreviations and acronyms can be used when the alternative, spelling it out, is unreasonably long. Acronyms should be capitalized. The descriptor name must be instrument agnostic and refer to a particular concept describing the data.
GP-Py-Name-19 **[STANDARD]**	**RecipeSystem data reduction package name**
	The name of the importable package containing all the data reduction recipes and primitives, and their associated data reduction algorithms and look-up tables should be named <descriptive>dr.
	For example, all the production Gemini data reduction code is provided by the geminidr package, and installed in the PYTHONPATH. When work is outsourced, the name of or abbreviation for the instrument is preferably used, eg. octocamdr.
GP-Py-Name-20 **[CRITICAL]**	**Location of add-on recipes**
	The recipes must be located in <drpkg>/<instrument>/recipes/<mode>, all lower case within <drpkg>, eg. geminidr/gmos/recipes/sq/. The allowed reduction modes are Science Quality (sq), Quality Assessment (qa), and QuickLook (ql).
	The recipe location is semi-hardcoded for now. The path is defined in the code as <drpkg>/ad.instrument(generic=True).lower()/recipes/<mode>/ The recipes within that location will be matched to the data using the tags.
GP-Py-Name-21 **[STANDARD]**	**Recipe module name**
	The recipe file shall be named recipes_<data_type>.py, recipes_FLAT_IMAGE.py. The
	While there are rules applying to the location of the recipe files, there are no critical-level rules for the name of the recipe files, they can be named anything. However, for readability, we have decided to use recipes_<data_type>.py
GP-Py-Name-22 **[CRITICAL]**	**Primitive sets location**
	Primitive set modules must be located in <drpkg>/<lowercase_instrument>/ , eg. geminidr/gmos/
	The recipe system mapper will use the name of the instrument to import the relevant primitive sets and build the one most appropriate to the input AstroData object.
GP-Py-Name-23	**Primitive set file name format**
	Primitive set files should be name primitives_<primsetid>.py, with <primsetid>

[STANDARD]	being a descriptive string related either to the data type or the type of algorithm it contains. Eg. primitives_gmos_mos.py, primitives_photometry.py
	This is simply a standard. There no hardcoded restriction on the file name.
GP-Py-Name-24	**Primitive parameters definition file name**
[STANDARD]	The parameter definition file name shall be using the format parameters_<primsetid>.py where <primsetid> must match the string used in the associated primitive file.
GP-Py-Name-25	**Primitive name**
[INTERFACE]	Primitives shall be named using camelCaps with an initial lowercase letter. Common abbreviations and acronyms can be used when the alternative, spelling it out, is unreasonably long. Acronyms should be capitalized.
GP-Py-Name-26	**Recipe name**
[INTERFACE]	Recipe functions, located in recipe modules, shall be named using lowercase_underscore syntax. Common abbreviations and acronyms can be used when the alternative, spelling it out, is unreasonably long. Acronyms should be capitalized.
GP-Py-Name-27	**Primitive parameter name**
[INTERFACE]	Primitive parameter names shall be kept down to a short single word in lowercase. When multiple words are absolutely necessary, underscores are to be used.
	The developer should try really hard to keep the parameter name to a short single word before resorting to the underscore, multiword option.
GP-Py-Name-28	**Output file name syntax**
[INTERFACE]	Each primitive shall be assigned a default suffix to be added to the root filename when written to disk. The suffix must be representative of the transformation done by the primitive.
	The suffixes used in the Gemini package are defined in PIPE-USER-102_DatasetNameSuffixes
GP-Py-Name-29	**Double underscore**
[STANDARD]	Double leading and trailing underscores shall not be used for variable or function names.
	Use only the documented double-underscore names (eg. __init__, __import__, __file__, etc.). Do not invent your own.
GP-Py-Name-30	**Acronyms in names**
[SUGGESTION]	Acronyms in function names, including primitives, shall be capitalized.
	E.g. ADUToElectrons.
GP-Py-Name-31	**The lowercase letter "l"**
	The lowercase letter "l" shall be avoided as variable name. If absolutely

[SUGGESTION]	needed, use "L".
	It is too easy to confuse 'el' with and 'i' or a '1', or even a pipe '|'.
GP-Py-Name-32	**Reserved EXTNAME**
	The following EXTNAME are reserved: • SCI: Science pixel extensions • VAR: Variance extensions • DQ: Data quality extensions • MDF: Mask Definition File extensions (FITS table) • PHU: Primary Header Unit • OBJCAT: FITS table of sources detected in the data • REFCAT: FITS table of sources from catalog that are in or near the field of view • OBJMASK: FITS table of sources to be masked.

7.6 Programming – GP-Py-Prog

GP-Py-Prog-1	**Comparison to None**
[STANDARD]	Comparison to None shall always be done with "is" or "is not", and never with the equality operators.
	Also, beware of writing "if x" when you really mean "if x is not None" – e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type (such as a container) that could be false in a boolean context.
GP-Py-Prog-2	**Class-based Exceptions**
[STANDARD]	Use class-based exceptions. String exceptions shall not be used.
	String exceptions have been removed since Python 2.6. In Python 3, exceptions must subclass BaseException.
GP-Py-Prog-3	**Raising Exceptions**
[STANDARD]	When raising an exception, use "raise ValueError('message' ".
	Do not use the older form "raise ValueError, 'message'".
GP-Py-Prog-4	**Catching Exceptions**
[STANDARD]	When catching exceptions, use specific exceptions whenever possible instead of bare except clause.

	For example:
	try:
	  import platform_specific_module
	except ImportError:
	  platform_specific_module = None

GP-Py-Prog-5	**String slicing for prefixes and suffixes**
[STANDARD]	The functions ".startwith()" and ".endswith()" shall be used to check for prefixes and suffixes instead of string slicing.

	For example:
	  if foo.startswith('bar'):
	instead of
	  if foo[:3] == 'bar':

GP-Py-Prog-6 **[STANDARD]**	**Object type comparison**
	Object type comparison shall be done with "isinstance()".

	For example:
	  if isinstance(obj, int):
	instead of
	  if type(obj) is type(1):

GP-Py-Prog-7 **[STANDARD]**	**Check for empty sequences**
	For sequences like strings, lists, tuples, use the fact that empty sequences are false.

	For example:
	  if not seq:
	instead of
	  if len(seq):

7.7 Classes – GP-Py-Class

GP-Py-Class-1 **[STANDARD]**	**Member declaration location**
	All class members shall be declared in the class scope at the top of the class.
	Specifically, do not declare members only in the __init__ function.
GP-Py-Class-2 **[STANDARD]**	**Order of member declaration**

	Members shall be ordered in the following order:
	
	1. data members
	  a. class scope members
	  b. private members
	  c. public members
	2. function members
	  a. class members (e.g. classmember decorated)
	  b. private member functions
	  c. public member functions

GP-Py-Class-3 **[STANDARD]**	**Mutable members initialized to None**
	Class members that are intended for mutable types (ie. dictionaries and lists) should be initialized to None in the class scope and initialized in the constructor.
	Setting the members to None avoids accidental use of a list or dictionary that is shared among all instances. If this is the desired effect then the code should be commented.
GP-Py-Class-4	**Member initialization**

[STANDARD]	Members can be declared to default starting values, but if the real starting value is set in __init__ then the member should be initialized to None.

7.8 Documentation – GP-Py-Doc

This section specifically applies, and is limited to, documentation associated with code. See also the Storage section of the Configuration Management Nomenclature document (DPSG-STD-101_ConfMgmtNomenclature) for a wider perspective.

GP-Py-Doc-1	**Use of docstrings**
[INTERFACE]	All functions, classes, and modules shall be documented with docstrings. The docstring format follows the astropy and numpy docstring standard.
	http://astropy.readthedocs.org/en/latest/development/docrules.html
GP-Py-Doc-2	**Use of Sphinx**
[STANDARD]	Documentation, especially user and programmer manuals shall be produced with Sphinx.
	Any other document for which Sphinx is appropriate should indeed make use of Sphinx. The SRS template, for example, is Sphinx-based.
GP-Py-Doc-3	**Use of reStructured Text**
[STANDARD]	ReStructured Text appropriate for use with Sphinx shall be used to document parameters and other special formatting in docstrings, and as a general format in Sphinx manuals.
GP-Py-Doc-4	**Output format for documentation**
[INTERFACE]	The output format of the documentation shall be PDF and HTML.
GP-Py-Doc-5	**Basic documentation requirements**
[STANDARD]	Basic documentation includes: • in-code comments • docstrings • programmer's manual • user's manual Basic documentation shall be kept up-to-date.
GP-Py-Doc-6	**Storage of manuals**
[STANDARD]	Manuals intended for distribution shall be stored with the code base.
	While a lot of development documents are stored in our internal DPSGdocuments repository, the documentation intended to be distributed, like user or programmer manuals, must be kept with the code such that it can be included during packaging.
GP-Py-Doc-7	**Document sources preservation**

[STANDARD]	The document sources, for all documents, shall be checked in the repository and available to edit.
	In other words, it must be possible to edit any document. For example, PDF only check-ins are not allowed.
GP-Py-Doc-8 **[STANDARD]**	**Use of the team's wiki (internal only)**
	The team's wiki shall be used for quick documentation, discussions, notes, and for often-used procedures. http://gdpsg.wikis-internal.gemini.edu/index.php/Main_Page
	The goal is for information and notes to be shared easily and rapidly. The wiki can be the initial location for documentation. Anything truly important though should be formally written and added to configuration management.
GP-Py-Doc-9 **[STANDARD]**	**Accuracy of comments**
	Comments shall agree with the code and be up-to-date.
GP-Py-Doc-10 **[SUGGESTION]**	**Comment style**
	• Comments shall be explanatory, clear, and concise. • Comments shall be written in English. • Comments shall not state the obvious. [Bad] x = x + 1 # Increment x [Good] x = x + 1 # Compensate for border
	Don't forget appropriate punctuation on long comments.
GP-Py-Doc-11 **[STANDARD]**	**Inline comments**
	Inline comments shall be used sparingly. Full line comments are preferred.
GP-Py-Doc-12 **[STANDARD]**	**docstring delimiters**
	The """ that ends a multiline docstring shall be on a line by itself and preceded by a blank line. One line docstrings can have the closing """ on the same line.

7.9 Tests – GP-Py-Test

This section describes the standards related to testing the Python software suite.

GP-Py-Test-1	**Unit tests for public methods**
	The API must have a thorough set of unit tests associated with it.

[STANDARD]	
GP-Py-Test-2 **[SUGGESTION]**	**Unit tests for private methods**
	It is **strongly** recommended to write unit tests for all private methods
GP-Py-Test-3 **[SUGGESTION]**	**Unit tests for functions**
	All function should have a unit test suite, especially functions likely to be called from outside their home module.
GP-Py-Test-4 **[STANDARD]**	**Unit test software**
	All unit tests use pytest.
GP-Py-Test-5 **[STANDARD]**	**Unit tests location**
	The unit tests are stored in the subdirectory named “tests” located at the same level as the module being tested.
GP-Py-Test-6 **[STANDARD]**	**Unit test filenames**
	The unit test filenames use this format “test_<descriptive_name>”, where the descriptive name should include the name of the module or class being tested, as applicable.
GP-Py-Test-7 **[STANDARD]**	**Regression tests for primitives**
	Each primitive must be tested with all relevant types of inputs using the Regression Test Framework.

7.10 Astrodata – GP-Py-AD

GP-Py-AD-1 **[STANDARD]**	**Astrodata dependencies**
	1. Depends on fits, numpy, astropy. 2. Shall not depend on pyraf/iraf 3. Astrodata shall not depend on the Recipe System.
	The purpose here is to ensure that people without pyraf/iraf, or without the need of the automation system (Recipe System) can still make use of the Astrodata features to manipulate FITS files.
GP-Py-AD-2 **[STANDARD]**	**Lazy loading**
	Dependencies that are time consuming to load and not always required should be lazy-loaded.

7.11 Astrodata Tag – GP-Py-ADTag

GP-Py-ADTag-1 **[STANDARD]**	**Use PHU only**
	The Astrodata tags shall be determined from the Primary Header Unit only.
	This obviously is dependent on the content of the PHU. The intent of this guideline is to promote improvement of the PHU when a new system is delivered, such that this guideline can be applied.
GP-Py-ADTag-2 **[STANDARD]**	**Astrodata tags are defined in <adpkg>_instrument/<instrument>/adclass.py**
	The Astrodata tags shall be defined in the adclass.py module located in the instrument subdirectory of the Astrodata add-on package. Eg. gemini_instrument/gmos/adclass.py

7.12 DRAGONS Add-on – GP-Py-DRadd

GP-Py-DRadd-1 **[STANDARD]**	**Provides Astrodata add-on and RecipeSystem add-on**
	A DRAGONS add-on package shall provide an Astrodata add-on package to support the new data, and a RecipeSystem add-on package that will provide the recipes, the primitives and the data reduction algorithms.
GP-Py-DRadd-2 **[STANDARD]**	**Astrodata add-on basic structure**
	<adpkgID>_instrument/ doc/ <name_of_document>/ (sphinx files, conf.py, intro.rst, etc) <instrument_name>/ __init__.py| adclass.py lookups.py (any other support file for that instrument) tests/ test_*.py __init__.py
	Also see: GP-Py-Name-13, GP-Py-Name-14, GP-Py-Test-5, GP-Py-Test-6, GP-Py-Const-5
GP-Py-DRadd-3 **[STANDARD]** **[CRITICAL]**	**RecipeSystem add-on basic structure**
	<drpkgID>dr/ doc/ progmanual/ (sphinx files, conf.py, intro.rst, etc) usermanual/

	(sphinx files, conf.py, intro.rst, etc) <instrument_name>/ lookups/ BPM/ __init__.py *.fits (bad pixel masks) MDF/ __init__.py *.fits (mask definition files) __init__.py (look-up tables as .py files) recipes/ qa/ __init__.py recipes_*.py ql/ __init__.py recipes_*.py sq/ __init__.py recipes_*.py __init__.py tests/ test_*.py __init__.py parameters_*.py primitives_*.py __init__.py
	WARNING: Some of that directory structure is a critical requirement. GP-Py-Name-19, GP-Py-Name-20, GP-Py-Name-22, GP-Py-Test-5, GP-Py-Test-6, GP-Py-Const-5

7.13 Recipe Set – GP-Py-Recipe

GP-Py-Recipe-1 **[CRITICAL]**	**Recipe module contents**
	A recipe module shall have • a top level docstring that explains the general purpose of the recipe set. • a variable named recipe_tags holding a set of Astroaata tags valid for this module. • A variable named default at the bottom setting the recipe to run if this module is selected by the Recipe Mapper.
	The first bullet is a non-critical standard, the other two are critical.
GP-Py-Recipe-2 **[STANDARD]**	**Recipe contents**
	A recipe is a function consisting of a list of calls to members of the selected primitive set. The primitive set is the sole argument of a recipe function. Flow logic is technically allowed but discouraged to keep the

	recipe readable to the scientist.
GP-Py-Recipe-3	**Recipe completeness**
[STANDARD]	A recipe shall perform a "complete" and "scientifically meaningful" transformation.
	This refers mostly to the notion that a recipe must perform something sizeable and meaningful to an astronomer.

7.14 Primitives – GP-Py-Prim

GP-Py-Prim-1	**Use of primitive template**
[STANDARD]	A primitive shall follow the template named PIPE-TEMP-101_Primitive.
	The template is available as a .py file in the Templates section of the DPSGdocuments repository.
GP-Py-Prim-2	**Primitive names. See GP-Py-Name-25.**
[STANDARD]	
GP-Py-Prim-3	**Primitives use the Gemini logger**
[STANDARD]	A primitive shall use the Gemini logger located in gempy/utils and the log message utilities in gempy/gemini/gemini_tools.
GP-Py-Prim-4	**Primitives timestamp the dataset**
[STANDARD]	A primitive shall timestamp the dataset header to indicate that it has been run on that dataset.
	A list of timestamp keywords is available in the geminidr/gemini/lookups/ directory (timestamp_keywords.py). A third-party package might have to complement that list with an additional lookup table in their add-on package.
GP-Py-Prim-5	**Primitives that should not be run twice on a dataset must be able to tell if they have been run already**
[STANDARD]	A primitive that should not be run twice on a dataset shall check for timestamp or other indicator and skip the processing, exiting gracefully.
GP-Py-Prim-6	**Inputs processed one at a time**
[STANDARD]	The inputs shall be processed as AstroData objects, one at a time, in a for-loop. Of course, if the inputs are to be combined this does not apply.
GP-Py-Prim-7	**Primitives use the External Task Interface**

[STANDARD]	A primitive calling an external (e.g. an IRAF task, SExtractor) shall use the External Task Interface (ETI).
GP-Py-Prim-8	**Primitives ordered alphabetically.**
[STANDARD]	Primitives in a primitive set shall be ordered alphabetically.

7.15 Descriptors – GP-Py-Descrip

GP-Py-Descrip-1	**Descriptor return value independent of processing status**
[INTERFACE]	A descriptor shall return the correct value, regardless of the data processing status of the AstroData object.
	For example, the value returned for an 'unprepared' dataset should be the same as for when the data has been 'prepared'.
GP-Py-Descrip-2	**Descriptors do not add or update keywords**
[STANDARD]	A descriptor shall not add or update entries to the headers of the AstroData object. A descriptor's value(s) can be written to the HISTORY, but for information only.
	The value of a descriptor should always be obtained from the call to the descriptor, not by looking for updated values in the headers.
GP-Py-Descrip-3	**Descriptor return value**
[INTERFACE]	Descriptors applying tp the whole array, and normally derived from information in the PHU, should always return a single value, while descriptors representing values specific to the extensions should return a list of values, even if there is only one value, unless the descriptor was specifically called on a single extension. If the descriptor function fails to calculated a value, it shall return None.
	Eg. ad.gain() returns a list, even for F2 which has only one extension, but ad[0].gain() returns a single value because it is called on a specific extension.
GP-Py-Descrip-4	**Keyword access in descriptor functions**
[STANDARD]	A descriptor function shall use the phu.get() and hdr.get() AstroData member functions to access keywords in the header of an AstroData object.
GP-Py-Descrip-5	**No logging in descriptors**
[INTERFACE]	A descriptor shall not log any messages.
GP-Py-Descrip-6	**Return None upon failure to calculate value**

[STANDARD]	A descriptor function shall return None if its value cannot be determined for whatever reason.
	It turns out that bad headers are too frequent to raise errors.
GP-Py-Descrip-7	**Descriptor names. See GP-Py-Name-18**
GP-Py-Descrip-8	**Standard descriptor arguments.**
[STANDARD]	Descriptor functions can only accept the pretty and stripID arguments.

7.16 External Modules – GP-Py-Ext

GP-Py-Ext-1	**IRAF dependency**
[STANDARD]	The gemini data reduction software is allowed to depend on IRAF. However, for the Python code base, it is strongly discouraged.
	If DRAGONS software must depend on IRAF, it should be justified and discussion with the SUSD members.
GP-Py-Ext-2	**PyRAF dependency and compatibility**
[STANDARD]	All IRAF scripts shall be PyRAF compatible. The Python code base is allowed to depend on the pyraf module, however, it is strongly discouraged.
	If DRAGONS software must depend on PyRAF, it should be justified and discussion with the SUSD members.
GP-Py-Ext-3	**IRAF external package dependencies**
[STANDARD]	The public package is allowed to depend only the following external packages: • fitsutil • stsdas • tables
GP-Py-Ext-4	**Allowed Python dependencies**
[STANDARD]	Gemini data reduction software is allowed to have the packages and modules included in AstroConda as dependencies.
	Any modules from the Python Standard Library is obviously allowed too (not really dependencies since they come with Python). See http://docs.python.org/2/library/ for a list.
GP-Py-Ext-5	**SExtractor dependencies**
[STANDARD]	SExtractor is an allowed dependency.
	It is also part of AstroConda.

7.17 GUI – GP-Py-GUI

***** TODO *****

GP-Py-GUI-1 **[STANDARD]**	**Tkinter**
GP-Py-GUI-2 **[STANDARD]**	**json**
GP-Py-GUI-3 **[STANDARD]**	**javascript**
GP-Py-GUI-4 **[STANDARD]**	
GP-Py-GUI-5 **[STANDARD]**	
GP-Py-GUI-6 **[STANDARD]**	
GP-Py-GUI-7	
GP-Py-GUI-8	

8. Guidelines for the gemini_IRAF Codebase

ID [Priority]	Name
	Statement
	Note

8.1 IRAF CL Source Code Formatting – GI-CL-Format

GI-CL-Format-1 [STANDARD]	**Line Length Rule**
	Line length shall be kept to 79 characters or less.
GI-CL-Format-2 **[STANDARD]**	**Line Wrapping**
	Long lines shall be wrapped using the backslash “\”
GI-CL-Format-3 **[STANDARD]**	**Indentation Rule**
	Indentation shall consist of four (4) spaces. Tabs are NOT allowed.
	It is therefore required that code only be edited with an editor that respects the uses of spaces (ie. does not convert whitespaces to tabs). Example of editors known to be configurable emacs, vi, PyCharm, and the eclipse IDE.
GI-CL-Format-4 **[SUGGESTION]**	**White Spaces**
	6) In statements, put one blank after comma "," . print (tmpinlist, > todel_list) 7) In statements, operators are flanked on both sides by a blank. counter = counter + 1 8) In function argument lists, do not flank equal signs "=" with blanks. sections (“@”//outlist, option=”nolist”)

8.2 IRAF CL Constants – GI-CL-Const

GI-CL-Const-1 [STANDARD]	**Constant Name**
	Name constants in ALLCAPS.
	This improves readability.
GI-CL-Const-2 **[STANDARD]**	**Respect for constants**
	Do not change the value of a constant.
GI-CL-Const-3 **[STANDARD]**	**No hardcode**
	Do not hardcode code statements.
	Instead of hardcoding, use constants properly, or use clearly assigned variables.

GI-CL-Const-4 **[STANDARD]**	**Look-up tables for constant in configuration space**
	Any values mostly stable but likely to change when changes to the instrument hardware is made should be stored in a look-up tables. The look-up tables are normally stored in the data$ directory of the instrument package.
GP-CL-Const-5 **[STANDARD]**	**Constant definition location**
	Constants appear at the top of modules, class, or function definitions.

8.3 IRAF CL Naming – GI-CL-Name

GI-CL-Name-1 **[STANDARD]**	**Variable names internal to a script.**
	Variable names shall start with a lower case letter and use underscores ("_") to separate terms for readability.
GI-CL-Name-2 **[SUGGESTED]**	**Meaning of variables**
	Variables can be named however suits the developer so long as they are not misleading. Sensible terms or abbreviations, and consistency throughout the code is expected.
GI-CL-Name-3 **[STANDARD]**	**Constant names. See GI-CL-Const-1**
GI-CL-Name-4 **[STANDARD]**	**Local variables mapping input parameters**
	Local variables mapping input parameters shall start with l_ followed by the name of the parameter. Eg. l_rawpath for parameter 'rawpath'.
GI-CL-Name-5 **[STANDARD]**	**Length of variables**
	The length of a variable name must be greater than 1 and, the uniqueness of the variable name must be guaranteed with the first 8 characters.
	Exceptions to the greater than one character name are allowed for obvious counters, eg. 'i'. Because the code is translated to Fortran, variables must have a unique name once converted. Names longer than 8 characters are allowed but care must be taken to ensure uniqueness post-convertion.
GI-CL-Name-6 **[INTERFACE]**	**Reserved parameter names**
	The following are reserved argument names: **fl_<something>** : The prefix fl_ is reserved for booleans. **inimages :** Used for input image lists **outimages :** Used for output image lists

	outprefix : Used for prefix to use for output filenames when outimages is not specified. **rawpath :** Used for path to raw data **fl_vardq :** Control propagation of VAR and DQ planes **logfile :** Name of the logfile **verbose :** Control verbose mode **status :** Exit status of the script
GI-CL-Name-7 **[INTERFACE]**	**Reduction scripts default output filename**
	The default naming of output files shall be a one-letter prefix and shall be specified by the parameter “outprefix”.
	TODO: List all the prefixes currently in use.
GI-CL-Name-8 **[SUGGESTION]**	**Acronyms in names**
	Acronyms in function names shall be capitalized.
	Note however that IRAF CL does not distinguish between upper case and lower case, “adu” is the same as “ADU”. This guideline is for code readability purpose only.
GI-CL-Name-9 **[SUGGESTION]**	**The lowercase letter "l"**
	The lowercase letter "l" shall be avoided as variable name. If absolutely needed, use "L".
	It is too easy to confuse 'el' with and 'i' or a '1', or even a pipe '|'.

8.4 IRAF CL Programming – GI-CL-Prog

GI-CL-Prog-1 **[STANDARD]**	**Full parameter and task names**
	Use full parameter and task names in code.
	For example, use “verbose” instead of “ver” or “verb”, “delete” instead of “del”. While IRAF CL will in many cases be able to do the completion, it might pick the wrong completion too, or simply crash if it cannot figure it out.
GI-CL-Prog-2 **[STANDARD]**	**Updating Headers**
	Use gemhedit when making updates to any headers.
GI-CL-Prog-3 **[STANDARD]**	**Calls to other IRAF tasks**
	Specify all parameters in calls to other IRAF tasks.
	This will prevent IRAF from retrieving erroneous cached parameter values from the uparm.
GI-CL-Prog-4	**If statement**
	One-line “if” statement are not allowed.

[STANDARD]	
GI-CL-Prog-5 **[STANDARD]**	**Length of script**
	Keep the length of the script as short as possible. Create utility scripts if necessary.
GI-CL-Prog-6 **[STANDARD]**	**Data Tables**
	Data tables must be FITS binary tables.
	The only exception is if sexagesimal format absolutely needs to be used, then the ".tab" format can be considered.
GI-CL-Prog-7 **[STANDARD]**	**End of line**
	Each line must end with a "\n" character with no white spaces between it and the previous character.
	In other words, no trailing white spaces, and simple Unix end-of-line character.

8.5 IRAF CL Documentation – GI-CL-Doc

GI-CL-Doc-1 **[STANDARD]**	**Help pages**
	Each task must have an IRAF help page.
	The help pages are normally stored in the doc$ directory of the instrument package.
GI-CL-Doc-2 **[STANDARD]**	**Content of help pages**
	A help page must contain the following sections: • Header (.help line) • NAME • USAGE • PARAMETERS • DESCRIPTION • EXAMPLES • TIME REQUIREMENTS (optional for non computer intensive tasks) • BUGS AND LIMITATIONS • SEE ALSO
GI-CL-Doc-3 **[STANDARD]**	**Formatting of help page text**
	IRAF task names are written in uppercase letters. Parameter names in descriptive text are highlighted (\fIparameter\fR).
GI-CL-Doc-4 **[STANDARD]**	**Basic documentation requirements**
	Basic documentation includes: • in-code comments

	• help page • data reduction example script for each mode Basic documentation shall be kept up-to-date.
GI-CL-Doc-5 **[STANDARD]**	**Storage of manuals**
	Manuals intended for distribution shall be stored with the code base in the doc$ directory of the instrument package.
GI-CL-Doc-6 **[STANDARD]**	**Document sources preservation**
	The document sources, for all documents, shall be checked in the repository and available to edit.
	In other words, it must be possible to edit any document. For example, PDF only check-ins are not allowed.
GI-CL-Doc-7 **[STANDARD]**	**Use of the team's wiki**
	The team's wiki shall be used for quick documentation, discussions, notes, and for often-used procedures. http://gdpsg.wikis-internal.gemini.edu/index.php/Main_Page
	The goal is for information and notes to be shared easily and rapidly. The wiki can be the initial location for documentation. Anything truly important though should be formally written and added to configuration management.
GI-CL-Doc-8 **[STANDARD]**	**Accuracy of comments**
	Comments shall agree with the code and be up-to-date.
GP-CL-Doc-9 **[SUGGESTION]**	**Comment style**
	1) Comments shall be explanatory, clear, and concise. 2) Comments shall be written in English. 3) Comments shall not state the obvious. [Bad] x = x + 1 # Increment x [Good] x = x + 1 # Compensate for border
	Don't forget appropriate punctuation on long comments.
GI-CL-Doc-10 **[STANDARD]**	**Inline comments**
	Inline comments shall be used sparingly. Full line comments shall be preferred.

8.6 IRAF CL Tests – GI-CL-Test

GI-CL-Test-1 **[STANDARD]**	**Regression tests for tasks**
	Each task must be tested with all relevant inputs using the Regression Test Framework.

8.7 IRAF CL External Dependencies – GI-CL-Ext

GI-CL-Ext-1 **[STANDARD]**	**IRAF external package dependencies**
	The allowed dependencies on external IRAF packages are: • stsdas • tables • fitsutil
GI-CL-Ext-2 **[STANDARD]**	**IRAF and PyRAF compatibility**
	All scripts written of the GEMINI IRAF package shall be both IRAF and PyRAF compatible.
	In other words, no Python code can be distributed as part of the GEMINI IRAF package.

9. Detailed Revision History

v1.0	27 March 2013	Kathleen Labrie
		Initial revision. Based on information written by Craig Allen on the wiki, with feedback and corrections from Emma Hogan. Added to official configuration management.
V1.1	16 June 2014	Kathleen Labrie
		Reviewed. Added section on tests. Added IRAF coding standards.
V2.0	25 January 2018	Kathleen Labrie
		Major update to match DRAGONS. Any reference to the old gemini_python removed.
V2.1	3 October 2018	Kathleen Labrie
		Move the glossary to the top following a suggestion from Bruno Quint. I also moved the Python requirements ahead of the IRAF ones.