cctbx **Developer Guidance**

The Computational Crystallography Toolbox (cctbx) is a large code base under active development by several groups worldwide. There are more than 1 million lines of code, 500 commits per month, and 20 active developers. It is therefore very important that all contributors follow some basic guidelines. Keep in mind that the intention is for the *cctbx* to provide a fully featured code base for crystallographic calculations while also remaining lightweight and straightforward to compile and install.

1. No new dependencies without discussion with the other developers.
    a. This is important to keep the *cctbx* easy to compile and install. The inclusion of third-party packages that have their own dependencies is therefore strongly discouraged. This applies especially if a dependency on a new compiler language is introduced.
2. Don't introduce new dependencies if there is code in *cctbx* to provide the needed functionality
    a. It is often the case that the required functionality already exists in the *cctbx*. Developers are encouraged to check first with other *cctbx* developers and review the code base prior to implementing new functionality.
3. Avoid code duplication
    a. Add new context independent functions into appropriate modules, not into specialized code. Example: a place for the function that calculates distance between two points is *scitbx*; however, it may be very tempting to inline this function into a specialized code as needed thus creating code duplication.
    b. Use constants from a central place. Example: use math.pi instead of defining pi=3.14 every time it is needed. Note: there are plenty of other constants available, such as scitbx::constants::two_pi_sq; add more as needed.
    (Make sure not to use an OS-dependent constant)
4. Use an appropriate coding style for *cctbx*
    a. There is a Python Style guide that is generally useful
    Note however that where *cctbx* deviates from PEP8, follow *cctbx* (for example using 2 spaces to indent instead of 4).
    b. Constructs should be used to make subsequent use and testing as easy to debug as possible (i.e. standard out and standard error output)
    c. Printing output
        i. Use "print >> log, bla" and not "print bla" .
        ii. Use "show" function to print a summary or result of code execution instead of in-lining print statements directly into the code.
        iii. Avoid unconditional printing.

d. Consistency. If editing an existing file, follow the code style of that file.

e. Code clarity. Ideally, clearly written code does not need documentation (however, even clearly written code may benefit from documentation!). Having this in mind:

   i. Use self-explicable function/class/argument/file names. A clear long name is better than a short cryptic name.

   ii. While valid exceptions exist, generally if a function does not fit a page then it is likely to benefit from being broken into smaller functions.

f. Common modern computation courses emphasize the importance of in-code documentation

   i. Since what is clear to one person might not so clear to another (or to the same person after several months or years).

   ii. Auto documentation creation (like sphinx) use the in-code documentation strings.

   iii. Time saving: a developer can read the documentation to understand what a function does, what are the arguments' types and formats and what it returns. This is much better than having to read and figure out what a function needs and what it does.

   iv. Longevity: Since developers and collaborators change over time and since the *cctbx* and related systems such as *Phenix* and DIALS are quite complex software, uncommented code might be a barrier for continuous development.

   v. Update documentation to reflect code changes

g. Developers are encouraged to correct ~~severe~~ deviations from coding styles found anywhere in codebase.

h. Developers are encouraged to modify code comments when unclear, outdated or in a format that does not render well in the SPHINX automated documentation.

5. Run find clutter before commits

   a. libtbx.find_clutter primarily checks for a few common errors:

      i. No 'from __future__ import division'. Use libtbx.add_from_future_import_division to fix. Important: always use the // operator to indicate integer division; the / operator is exclusively for floating point division.

      ii. Tabs or trailing whitespace: use libtbx.clean_clutter to fix.

      iii. Unused imports: use libtbx.find_unused_imports_crude to find them and remove them.

iv. The use of bare except statements is prohibited, since it causes the try...except to catch KeyboardInterrupt. At a minimum use 'except Exception'.

b. Before submitting your code, be sure to test it again after fixing problems by running find_clutter.

6. Note that some IDEs have Code inspection tools and Style formatting tools, that can help maintain the style and avoid other code pitfalls (for example: pycharm)
7. Use the *cctbx* mailing list to ask for guidance from other developers, and locate specific features in the current code base.
    a. The other *cctbx* developers are an invaluable resource that can be used to help with getting started in *cctbx* development.
    b. Link to the mailing list: http://www.phenix-online.org/mailman/listinfo/cctbxbb
8. Send email to the mailing list stating the intent to submit a new code tree within the *cctbx*.
    a. There are several sub projects within the *cctbx*, often embodied in the form of code trees within the main *cctbx* project. New code trees should not be introduced without good reason, and if abandoned the code tree should be removed. This reduces the accumulation of distracting code in the *cctbx* over time.
    b. Periodically, unused code trees may be removed from the *cctbx* to minimize clutter.
9. Developers are encouraged to subscribe to svn check-in alerts sent via email (code changes alerts) and review the diffs.
    a. This will minimize surprises later when someone changes someone's code.
    b. Any inefficiencies or bugs spotted in diffs should be pointed out to a respective contributor or fixed by anyone who found them first.
10. Python do's and don'ts:
    a. Prefer xrange() and xreadlines() to range() and readlines() for performance reasons; especially for large lists & arrays.
    b. Use inheritance to specialize classes whenever possible to avoid the duplication of code.
    c. Put most imports inside the method whenever possible, thus avoiding imports within the global space of a module. Nested imports create huge runtime overhead, particularly as the code base has grown so large over time.
    d. Never use import *, thus it should always be clear within a module where a name comes from, i.e., from math import sin, cos, pi. The only exception is within the __init__.py module of a package, where it is permissible to import all items from a C++ extension module.

e. Never use isinstance(): a method should not be forced to inquire what type an argument is in order to know how to perform. Instead, the method is entitled to expect arguments to conform to an interface specification; for example if the method prints an object, it should be expected (or documented) that the object should have a __str__() method. More details of this discussion may be found at http://www.canonical.org/~kragen/isinstance

11. Tests (see more below)
    a. Any newly added functionality requires a unit test.
    b. Developers are encouraged to add unit tests to any existing functionality found to have no unit test.
    c. Any bug fix requires a regression test.

Guidance for Developing Tests

The *cctbx* tests are to ensure the code base is always functional. Tests preserve designed functionality ensuring it always performs as expected. Also, tests are a great learning resource as they exemplify most of the available functionality. For many developers they substitute for documentation. Note: not all tests available in the code base are good examples to follow. When adding a new test please follow the guidelines below. Ask questions on the *cctbx* mailing list.

1. Place:
    a. Each module has a directory called regression. This is the place for all tests.
    b. Each module has a file called run_tests.py that runs all tests listed in it (that includes all tests in that module).
    c. Historically, many test files were added next to the actual implementation (in the same directory). Those should eventually be moved to the regression directory.
2. Name:
    a. The general name template for a test file is tst_xxx.py, where "xxx" may be the name of functionality or file being tested. Example: tst_miller.py.
3. Run time (per each tst_xxx.py):
    a. The faster, the better. Generally execution time should be well under 30 seconds, in exceptional cases 60 seconds should be the absolute max.
4. Adding a test involves three steps:
    a. Create a file tst_xxx.py
    b. Place it into the module/regression directory
    c. Edit module/run_tests.py file (otherwise the test will not be run).
5. Miscellaneous:

a. Tests should be focused, clear and exercise one functionality at a time. A general template is shown in the inset below.

b. Ideally, a test code ("exercise" in the example on the right) should not exceed a page in length so it can be quickly read through and understood (and, if it fails, be fixed by anyone). One file may contain several tests. A brief doc string should state the test objective, means and expected result. If a test fails the failure should be clear by showing the full traceback (no swallowing tracebacks with printing "TEST FAILED").

```
from __future__ import division
from libtbx.utils import format_cpu_times
from libtbx.test_utils import approx_equal

def exercise():
  """
  Make sure 2*2 is 4.
  """
  x=2.
  result=x*x
  assert approx_equal(result, 4.)

if(__name__ == "__main__"):
  exercise()
  print format_cpu_times()
  print "OK"
```

c. Use tools from libtbx.test_utils as much as possible. Add more as needed. Example: use approx_equal to assert the expected result.

d. Inputs that can be generated at run time should be used as much as possible (as opposed to storing input files with models and data). Examples:

   i. If an atomic model is needed use random_structure or make the PDB records as short as possible and inline them into the body of the test file.

   ii. Diffraction data can be calculated from an atomic model.

e. If an auxiliary functionality is identified that is repeatedly used across multiple tests and is deemed to be useful for future tests, it may be abstracted and placed in specialized locations such as cctbx/development. An example of such functionality is cctbx/development/random_structure.py that generates a random atomic model.

f. It is best to keep the structure and style of tests as similar as possible, so that anyone (and not only the test author) can fix a broken test if necessary. Remember, fixing broken tests is not a pleasant exercise and often is time consuming. Therefore, any test design that can make this task easier is greatly appreciated; one is keeping tests similar in structure and style.

g. Tests should be robust w.r.t. platform, compiler and rounding errors.

h. Broken tests stop others from committing their code. Therefore fixing a failed test is the highest priority.

i. Avoid using libtbx.easy_run.fully_buffered because it hides the output and the traceback. Use libtbx.easy_run.call instead. Print the command that is about to run to standard output.

j. Output actual values in a failed assert statement:

- `assert a>b, "%f > %f assertion failed" % (a,b)`