Atom selection syntax

The PDB hierarchy enables fast selection lookup based on a simple syntax. Many programs in Phenix make heavy use of these selections to handle user input, but they are equally important internally, where they simplify the task of pulling specific atoms out of the hierarchy.

The syntax allows for selection of atomic properties such as atom name, residue number, chain ID, or B-factor, which can be combined by the use of simple boolean Examples of the syntax:

All atoms

```
all
```

All atoms with H in the name (* is a wildcard character)

```
name *H*
```

Atoms names with * (backslash disables wildcard function)

```
name o2\*
```

Atom names with spaces

```
name 'O 1'
```

Atom names with primes don't necessarily have to be quoted

```
name o2'
```

Boolean "and", "or", and "not"

```
resname ALA and (name ca or name c or name n or name o)
chain a and not altid b
resid 120 and icode c and model 2
segid a and element c and charge 2+ and anisou
```

Note that the `and`, `or`, and `not` operators have equal priority, so parentheses may be required to indicate which clauses go together.
The first example above selects for all atoms with the specified names in alanine residues, but if the parentheses are omitted:

```
resname ALA and name ca or name c or name n or name o
```

it will instead select C-alpha atoms in ALA, plus all atoms named C, N, or O regardless of residue name.

A single residue by number

```
resseq 188
```

Note that if there are several chains containing residue number 188, all of them will be selected. To be more specific and select residue 188 in particular chain:

```
chain A and resid 188
```

this will select residue 188 only in chain A.

A range of residues

```
resseq 2:10
```

This selects all residue numbers falling within this range, independent of their order in the PDB file. The numbering must be ascending; ``resseq 10:2`` will not work.

Insertion codes

`resid` combines the residue number (`resseq`) with the insertion code (`icode`); thus the following selections are identical

```
resid 188
resseq 188 and icode ' '
```

as are these:

```
resid 27C
resseq 27 and icode 'C'
```

Specifying ordered ranges of residues

For some purposes, such as specifying TLS groups, it may be impractical to specify a numerical range of `resseq` attributes. This is especially the case when the residue numbering is not continuous and ascending, which is found in some PDB entries, or when the insertion code is non-blank. An alternative is to define a series of residues by their combined number and insertion code, using the `through` keyword:

```
chain A and resid 1000 through 17J
```

In this case, the residues will be examined in the order that they appear in the PDB file, and every atom falling between 1000 and 17J in chain A will be included. This is the only situation where the ordering of atoms is important in atom selections.

B-factors and occupancies

You may also select atoms based on their numerical properties:

```
bfactor > 80
bfactor = 0
occupancy < 1
occupancy = 0
```

Other

A few additional keywords are supported:

```
element H
water
pepnames
hetero
```

These examples specify hydrogen atoms, water molecules (detected by residue name), common amino acid residues, and atoms labeled as HETATM, respectively.

Selecting no atoms

```
(not all)
```

libtbx.phil - Python-based hierarchical interchange language

Phil overview

Phil (Python-based hierarchical interchange language) is a module for the management of parameters and inputs. Many applications use command-line options as a user interface (e.g. based on Python's optparse, a.k.a Optik). This approach works well for small applications, but has it limitations for more complex applications with a large set of parameters.

A simple Phil file as presented to the user may look like this:

```
minimization.input {
  file_name = experiment.dat
  label = set2
}
minimization.output {
  model_file = final.mdl
  plot_file = None
}
minimization.parameters {
  method = *bfgs conjugate_gradient
  max_iterations = 10
}
```

Phil is designed with a minimal syntax. An important goal is to enable users to get started just by looking at defaults and examples, without having to read documentation.

The Phil syntax has only two main elements, `phil.definition` (e.g. `max_iterations = 10`) and `phil.scope` (e.g. `minimization.input { }`). To make this syntax as user-friendly as possible, strings do not have to be quoted and, unlike Python, indentation is not syntactically significant. E.g. this:

```
minimization.input {
file_name="experiment.dat"
labels="set2"
}
```

is equivalent to the corresponding definitions above.

Scopes can be nested recursively. The number of nesting levels is limited only by Python's recursion limit (default 1000). To maximize convenience, nested scopes can be defined in two equivalent ways. For example:

```
minimization {
  input {
```

```
  }
}
```
is equivalent to:

```
minimization.input {
}
```

Features of Phil

Phil is much more than just a parser for a very simple, user-friendly syntax. Major Phil features are:

- The concepts of `master files` and `user files`. The syntax for the two types of Phil files is identical, but the processed Phil files are used in different ways. I.e. the concepts exist only at the semantical level. The "look and feel" of the files is uniform.
- Interpretation of command-line arguments as Phil definitions.
- Merging of (multiple) Phil files and (multiple) Phil definitions derived from command-line arguments.
- Automatic conversion of Phil files to Python objects which are essentially independent of the Phil system. I.e. core algorithms using Phil-derived parameter objects do not actually have to depend on Phil.
- The reverse conversion of (potentially modified) Python objects back to Phil files. This could also be viewed as a Phil pretty printer.
- Shell-like variable substitution using $var and ${var} syntax.
- `include` syntax to merge Phil files at the parser level, or to import Phil objects from other Python scripts.

Master files

Master files are written by the software developer and include "attributes" for each parameter, such as the type (integer, floating-point, string, etc.) and support information for graphical interfaces. For example:

```
minimization.parameters
  .help = "Selection and tuning of minimization algorithm."
  .expert_level = 0
{
  method = *bfgs conjugate_gradient
    .type = choice
  max_iterations = 10
    .type = int
    .input_size = 8
}
```

The is the last part of the output of this command:

```
libtbx.phil --show-some-attributes example.params
```

Run this command with `--show-all-attributes` to see the full set of `definition` and `scope`attributes. This output tends to get very long, but end-users don't have to be aware of this, and even programmers only have to deal with the attributes they want to change.

User files

User files are typically generated by the application. For example:

```
minimization.quick --show_defaults
```

will process its master file and show only the most relevant parameters, classified by the software developer as `.expert_level = 0` (default). E.g.
the `minimization.parameters` scope in the example above is not shown. The attributes are also not shown. Therefore the output is much shorter compared to the `libtbx.phil --show-some-attributes` output above:

```
minimization.parameters {
  method = *bfgs conjugate_gradient
  max_iterations = 10
}
```

Command-line arguments

In theory, the user could save and edit the generated parameter files. However, in many practical situations this manual step can be avoided. Phil is designed with the idea that the application inspects all input files and uses the information found to fill in the blanks automatically. This is not only convenient, but also eliminates the possiblity of typing errors. In addition, the user can specify parameters directly on the command line, and this information is also use to fill in the blanks.

Command-line arguments that are not file names or options prefixed with '--' (like '--show_defaults' above) should be given to Phil for examination. E.g., this is a possible command:

```
minimization.quick experiment.dat output.plot_file=plot.pdf
```

First the application should check if an argument is the name of a file that can be opened. Assume this succeeds for the first argument, so the processing of this argument is finished. Assume further that a file with the name 'output.plot_file=plot.pdf' does not exist. This argument will therefore be interpreted with Phil. The next section presents an example.

fetch: merging of Phil objects

The Phil parser converts master files, user files and command line arguments to uniform Phil objects which can be merged to generate a combined set of "working" parameters used in running the application. We demonstrate this by way of a simple, self-contained Python script with embedded Phil syntax:

```
from libtbx.phil import parse

master_phil = parse("""
  minimization.input {
    file_name = None
      .type = path
    label = None
      .type = str
  }
  """)

user_phil = parse("""
  minimization.input {
    file_name = experiment.dat
  }
  """)

command_line_phil = parse(
  "minimization.input.label=set2")

working_phil = master_phil.fetch(
  sources=[user_phil, command_line_phil])
working_phil.show()
```

`master_phil` defines all available parameters including the type information. `user_phil` overrides the default 'file_name' assignment but leaves the 'labels' undefined. These are defined by a (fake) command-line argument. All inputs are merged via `master_phil.fetch()`.
`working_phil.show()` produces:

```
minimization.input {
  file_name = experiment.dat
  label = set2
}
```

Having to type in fully qualified parameter names (e.g. `minimization.input.labels`) can be very inconvenient. Therefore Phil includes support for matching parameter names of command-line arguments as substrings to the parameter names in the master files:

```
argument_interpreter = master_phil.command_line_argument_interpreter(
  home_scope="minimization")

command_line_phil = argument_interpreter.process(
```

```
  arg="minimization.input.label=set2")
```
This works even if the user writes just 'label=set2' or even 'put.lab=x1 x2'. The only requirement is that the substring leads to a unique match in the master file. Otherwise Phil produces a helpful error message. For example:

```
argument_interpreter.process("a=set2")
```
leads to:

```
Sorry: Ambiguous parameter definition: a = set2
Best matches:
  minimization.input.file_name
  minimization.input.label
```
The user can cut-and-paste the desired parameter into the command line for another trial to run the application.

extract: conversion of Phil objects to Python objects

The Phil parser produces objects that preserve most information generated in the parsing process, such as line numbers and parameter attributes. While this information is very useful for pretty printing (e.g. to archive the working parameters) and the automatic generation of graphical user interfaces, it is only a burden in the context of core algorithms. Therefore Phil supports "extraction" of light-weight Python objects from the Phil objects. Based on the example above, this can be achieved with just one line:

```
working_params = working_phil.extract()
```
We can now use the extracted objects in the context of Python:

```
print working_params.minimization.input.file_name
print working_params.minimization.input.label
```
Output:

```
experiment.dat
set2
```
'file_name' and 'label' are now a simple Python strings.

format: conversion of Python objects to Phil objects

Phil also supports the reverse conversion compared to the previous section, from Python objects to Phil objects. For example, to change the label:

```
working_params.minimization.input.label = "set3"
modified_phil = master_phil.format(python_object=working_params)
modified_phil.show()
```
Output:

```
minimization.input {
  file_name = "experiment.dat"
  label = "set3"
}
```

We need to bring in `master_phil` again because all the meta information was lost in the `working_phil.extract()` step that produced `working_params`. A type-specific converter is used to produce a string for each Python object (see the Extending Phil section below).

.multiple = True

Both `phil.definition` and `phil.scope` support the `.multiple = True` attribute. For the sake of simplicity, in the following "multiple definition" and "multiple scope" means a master definition or scope with `.multiple = True`. Please note the distinction between this and multiple *values* given in a user file. For example, this is a multiple definition in a master file:

```
master_phil = parse("""
  minimization.input {
    file_name = None
      .type = path
      .multiple = True
  }
  """)
```

And these are multiple values for this definition in a user file:

```
user_phil = parse("""
  minimization.input {
    file_name = experiment1.dat
    file_name = experiment2.dat
    file_name = experiment3.dat
  }
  """)
```

I.e. multiple values are simply specified by repeated definitions. Without the `.multiple = True` in the master file, `.fetch()` retains only the *last* definition found in the master and all user files or command-line arguments. `.multiple = True` directs Phil to keep all values. `.extract()` then returns a list of all these values converted to Python objects. For example, given the user file above:

```
working_params = master_phil.fetch(source=user_phil).extract()
print working_params.minimization.input.file_name
```

will show this Python list:

```
['experiment1.dat', 'experiment2.dat', 'experiment3.dat']
```

Multiple scopes work similarly, for example:

```
master_phil = parse("""
```

```
  minimization {
    input
      .multiple = True
    {
      file_name = None
        .type = path
      label = None
        .type = str
    }
  }
  """)
```

A corresponding user file may look this this:

```
user_phil = parse("""
  minimization {
    input {
      file_name = experiment1.dat
      label = set2
    }
    input {
      file_name = experiment2.dat
      label = set1
    }
  }
  """)
```

The result of the usual fetch-extract sequence is:

```
working_params = master_phil.fetch(source=user_phil).extract()
for input in working_params.minimization.input:
  print input.file_name
  print input.label
```

Output:

```
experiment1.dat
set2
experiment2.dat
set1
```

Definitions and scopes may be nested with any combination of `.multiple = False` or `.multiple = True`. For example, this would be a plausible master file:

```
master_phil = parse("""
  minimization {
    input
      .multiple = True
    {
      file_name = None
        .type = path
      label = None
        .type = str
        .multiple = True
    }
  }
  """)
```

This is a possible corresponding user file:

```
user_phil = parse("""
  minimization {
    input {
      file_name = experiment1.dat
      label = set1
      label = set2
      label = set3
    }
    input {
      file_name = experiment2.dat
      label = set2
      label = set3
    }
  }
  """)
```

The fetch-extract sequence is the same as before:

```
working_params = master_phil.fetch(source=user_phil).extract()
  for input in working_params.minimization.input:
    print input.file_name
    print input.label
```

but the output shows lists of strings for `label` instead of just one Python string:

```
experiment1.dat
['set1', 'set2', 'set3']
experiment2.dat
['set2', 'set3']
```

fetch_diff: difference between master_phil and working_phil

The `.fetch()` method introduced above produces a complete copy of the Phil master with all user definitions and scopes merged in. If the Phil master is large, the output of `working_phil.show()` will therefore also be large. It may be difficult to see which definitions still have default values, and which definitions are changed. To get just the difference between the master and the working Phil objects, the `.fetch_diff()` method is available. For example:

```
master_phil = parse("""
  minimization.parameters {
    method = *bfgs conjugate_gradient
      .type = choice
    max_iterations = 10
      .type = int
  }
  """)

user_phil = parse("""
  minimization.parameters {
```

```
    method = bfgs *conjugate_gradient
  }
  """)

working_phil = master_phil.fetch(source=user_phil)
diff_phil = master_phil.fetch_diff(source=working_phil)
diff_phil.show()
```

Output:

```
minimization.parameters {
  method = bfgs *conjugate_gradient
}
```

Here the minimization method was changed from 'bfgs' to 'conjugate_gradient' but the number of iterations is unchanged. Therefore the latter does not appear in the output. `.fetch_diff()` is completely general and works for any combination of definitions and scopes with `.multiple = False` or `.multiple = True`.

Includes

Phil also supports merging of files at the parsing level. For example:

```
include file general.params

minimization.parameters {
  include file specific.params
}
```

Another option for building master files from a library of building blocks is based on Python's import mechanism. For example:

```
include file general.params

minimization.parameters {
  include scope app.module.master_phil
}
```

When encountering the `include scope`, the Phil parser automatically imports `app.module`(equivalent to `import app.module` in a Python script). The `master_phil` object in the imported module must be a pre-parsed Phil scope or a plain Phil string. The content of the `master_phil`scope is inserted into the scope of the `include scope` statement.

`include` directives enable hierarchical building of master files without the need to copy-and-paste large fragments explicitly. Duplication appears only in automatically generated user files. I.e. the programmer is well served because a system of master files can be kept free of large-scale redundancies that are difficult to maintain. At the same time the end user is well served because the indirections are resolved automatically and all parameters are presented in one uniform view.

Variable substitution

Phil supports variable substitution using $var and $(var) syntax. A few examples say more than many words:

```
var_phil = parse("""
  root_name = peak
  file_name = $root_name.mtz
  full_path = $HOME/$file_name
  related_file_name = $(root_name)_data.mtz
  message = "Reading $file_name"
  as_is = ' $file_name '
  """)
var_phil.fetch(source=var_phil).show()
```

Output:

```
root_name = peak
file_name = "peak.mtz"
full_path = "/net/cci/rwgk/peak.mtz"
related_file_name = "peak_data.mtz"
message = "Reading peak.mtz"
as_is = ' $file_name '
```

Note that the variable substitution does not happen during parsing. The output of `params.show()` is identical to the input. In the example above, variables are substituted by the `.fetch()` method that we introduced earlier to merge user files given a master file.

Extending Phil

Phil comes with a number of predefined converters used by `.extract()` and `.format()` to convert to and from pure Python objects. These are:

```
.type =
  words     retains the "words" produced by the parser
  strings   list of Python strings (also used for .type = None)
  str       combines all words into one string
  path      path name (same as str_converters)
  key       database key (same as str_converters)
  bool      Python bool
  int       Python int
  float     Python float
  choice    string selected from a pre-defined list
```

It is possible to extend Phil with user-defined converters. For example:

```
import libtbx.phil
from libtbx.phil import tokenizer

class upper_converters:
```

```
  phil_type = "upper"

  def __str__(self): return self.phil_type

  def from_words(self, words, master):
    s = libtbx.phil.str_from_words(words=words)
    if (s is None): return None
    return s.upper()

  def as_words(self, python_object, master):
    if (python_object is None):
      return [tokenizer.word(value="None")]
    return [tokenizer.word(value=python_object.upper())]

converter_registry = libtbx.phil.extended_converter_registry(
  additional_converters=[upper_converters])
```

The extended `converter_registry` is passed as an additional argument to Phil's `parse` function:

```
master_phil = parse("""
  value = None
    .type = upper
  """,
    converter_registry=converter_registry)
user_phil = parse("value = extracted")
working_params = master_phil.fetch(source=user_phil).extract()
print(working_params.value)
```

The `print` statement at the end writes "EXTRACTED". It also goes the other way, starting with a lower-case Python value:

```
working_params.value = "formatted"
working_phil = master_phil.format(python_object=working_params)
working_phil.show()
```

The output of the `.show()` call is "value = FORMATTED".

Arbitrary new types can be added to Phil by defining similar converters. If desired, the pre-defined converters for the basic types can even be replaced. All converters have to have `__str__()`, `from_words()` and `as_words()` methods. More complex converters may optionally have a non-trivial `__init__()` method (an example is the `choice_converters` class in `libtbx/phil/__init__.py`).

Additional domain-specific converters are best defined in a separate module, along with a corresponding parse() function using the extended converter registry as the default. See, for example, `iotbx/phil.py` in the same `cctbx` project that also hosts `libtbx`.

Details

.type = ints and .type = floats

The built-in `ints` and `floats` converters handle lists of integer and floating point numbers, respectively. For example:

```
master_phil = parse("""
  random_integers = None
    .type = ints
  euler_angles = None
    .type = floats(size=3)
  unit_cell_parameters = None
    .type = floats(size_min=1, size_max=6)
  rotation_part = None
    .type = ints(size=9, value_min=-1, value_max=1)
  """)

user_phil = parse("""
  random_integers = 3 18 5
  euler_angles = 10 -20 30
  unit_cell_parameters = 10,20,30
  rotation_part = "1,0,0;0,-1,0;0,0,-1"
  """)

working_phil = master_phil.fetch(source=user_phil)
working_phil.show()
print
working_params = working_phil.extract()
print working_params.random_integers
print working_params.euler_angles
print working_params.unit_cell_parameters
print working_params.rotation_part
print
working_phil = master_phil.format(python_object=working_params)
working_phil.show()
```

Output:

```
random_integers = 3 18 5
euler_angles = 10 -20 30
unit_cell_parameters = 10,20,30
rotation_part = "1,0,0;0,-1,0;0,0,-1"

[3, 18, 5]
[10.0, -20.0, 30.0]
[10.0, 20.0, 30.0]
[1, 0, 0, 0, -1, 0, 0, 0, -1]

random_integers = 3 18 5
euler_angles = 10 -20 30
unit_cell_parameters = 10 20 30
rotation_part = 1 0 0 0 -1 0 0 0 -1
```

The list of `random_integers` can have arbitrary size and arbitrary values.

For `euler_angles`, exactly three values must be given.

For `unit_cell_parameters`, one to six values are acceptable.

The list of values for `rotation_part` must have nine integer elements, with values {-1,0,1}.

All keywords are optional and can be used in any combination, except if `size` is given, `size_min`and `size_max` cannot also be given.

Lists of values can optionally use commas or semicolons as separators between values. In this context, both characters are equivalent to a white-space. `.format()` always uses spaces as separators, i.e. commas and semicolons are not preserved in an `.extract()`-`.format()` cycle. (Note that lists using semicolons as separators must be quoted; see the "Semicolon syntax" section below.)

.type = choice

The built-in `choice` converters support single and multi choices. Here are two examples, a single choice `gender` and a multi choice `favorite_sweets`:

```
master_phil = parse("""
  gender = male female
    .type = choice
  favorite_sweets = ice_cream chocolate candy_cane cookies
    .type = choice(multi=True)
  """)

jims_choices = parse("""
  gender = *male female
  favorite_sweets = *ice_cream chocolate candy_cane *cookies
  """)

jims_phil = master_phil.fetch(source=jims_choices)
jims_phil.show()
jims_params = jims_phil.extract()
print jims_params.gender, jims_params.favorite_sweets
```

Selected items are marked with a star '*'. The `.extract()` method returns either a string with the selected value (single choice) or a list of strings with all selected values (multi choice). The output of the example is:

```
gender = *male female
favorite_sweets = *ice_cream chocolate candy_cane *cookies
male ['ice_cream', 'cookies']
```

To maximize convenience, especially for choices specified via the command-line, the '*' is optional if only one value is given. For example, the following two definitions are equivalent:

```
gender = female
gender = male *female
```

If the `.optional` attribute is not defined, it defaults to `True` and this is possible:

```
ignorant_choices = parse("""
  gender = male female
  favorite_sweets = ice_cream chocolate candy_cane cookies
  """)

ignorant_params = master_phil.fetch(source=ignorant_choices).extract()
print ignorant_params.gender, ignorant_params.favorite_sweets
```

Output:

```
None []
```

In this case the application has to deal with the `None` and the empty list. If `.optional = False`, `.extract()` will lead to informative error messages. The application will never receive `None` or an empty list.

If a value in the user file is not a possible choice, `.extract()` leads to an error message listing all possible choices, for example:

```
Sorry: Not a possible choice for favorite_sweets: icecream
  Possible choices are:
    ice_cream
    chocolate
    candy_cane
    cookies
```

This message is designed to aid users in recovering from mis-spelled choices typed in at the command-line. Command-line choices are further supported by this syntax:

```
greedy_choices = parse("""
  favorite_sweets=ice_cream+chocolate+cookies
  """)

greedy_params = master_phil.fetch(source=greedy_choices).extract()
print greedy_params.favorite_sweets
```

Ouput:

```
['ice_cream', 'chocolate', 'cookies']
```

Finally, if the `.optional` attribute is not specified or `True`, `None` can be assigned:

```
no_thanks_choices = parse("""
  favorite_sweets=None
  """)

no_thanks_params = master_phil.fetch(source=no_thanks_choices).extract()
print no_thanks_params.favorite_sweets
```

Output:

```
[]
```

scope_extract

The result of `scope.extract()` is a `scope_extract` instance with attributes corresponding to the embedded definitions and sub-scopes. For example:

```
master_phil = parse("""
  minimization.input {
    file_name = None
      .type = path
  }
  minimization.parameters {
    max_iterations = 10
      .type = int
  }
  """)

user_phil = parse("""
  minimization.input.file_name = experiment.dat
  minimization.parameters.max_iterations = 5
  """)

working_params = master_phil.fetch(source=user_phil).extract()
print working_params
print working_params.minimization.input.file_name
print working_params.minimization.parameters.max_iterations
```

Output:

```
<libtbx.phil.scope_extract object at 0x2ad50bae7550&mt;
experiment.dat
5
```

This just repeats what was shown several times before, but `scope_extract` includes a few additional, special features that are worth knowing. The first special feature is the `.__phil_path__()` method:

```
print working_params.minimization.input.__phil_path__()
print working_params.minimization.parameters.__phil_path__()
```

Output:

```
minimization.input
minimization.parameters
```

This feature is most useful for formatting informative error messages without having to hard-wire the fully-qualified parameter names. Use `.__phil_path__()` to ensure that the names are automatically correct even if the master file is changed in major ways. Note that the`.__phil_path__()` method is available only for extracted scopes, not for extracted definitions since it would be very cumbersome to implement. However, the fully-qualified name of a definition can be obtained via `.__phil_path__(object_name="max_iterations")`; usually the `object_name` is readily available in the contexts in which the fully-qualified name is needed. There is

also `.__phil_path_and_value__(object_name)` which returns a 2-tuple of the fully-qualified path and the extracted value, ready to be used for formatting error messages.

The next important feature is a safety guard: assignment to a non-existing attribute leads to an exception. For example, if the attribute is mis-spelled:

```
working_params.minimization.input.filename = "other.dat"
```

Result:

```
AttributeError: Assignment to non-existing attribute
"minimization.input.filename"
  Please correct the attribute name, or to create
  a new attribute use: obj.__inject__(name, value)
```

In addition to trivial spelling errors, the safety guard traps overlooked dependencies related to changes in the master file.

In some (unusual) situations it may be useful to attach attributes to an extracted scope that have no correspondence in the master file. Use the `.__inject__(name, value)` method for this purpose to by-pass the safety-guard. As a side-effect of this design, injected attributes are easily pin-pointed in the source code (simply search for `__inject__)`, which can be a big help in maintaining a large code base.

Multiple definitions and scopes

All Phil attributes of multiple definitions or scopes are determined by the first occurrence in the master file. All following instances in the master file are defaults. Any instances in user files (merged via `.fetch()`) are added to the default instances in the master file. For example:

```
master_phil = parse("""
  plot
    .multiple = True
  {
    style = line bar pie_chart
      .type=choice
    title = None
      .type = str
  }
  plot {
    style = line
    title = Line plot (default in master)
  }
  """)

user_phil = parse("""
  plot {
    style = bar
```

```
    title = Bar plot (provided by user)
  }
  """)

working_phil = master_phil.fetch(source=user_phil)
working_phil.show()
```

Output:

```
plot {
  style = *line bar pie_chart
  title = Line plot (default in master)
}
plot {
  style = line *bar pie_chart
  title = Bar plot (provided by user)
}
```

`.extract()` will produce a list with two elements:

```
working_params = working_phil.extract()
print working_params.plot
```

Output:

```
[<libtbx.phil.scope_extract object at 0x2b1ccb5b1910&mt;,
 <libtbx.phil.scope_extract object at 0x2b1ccb5b1c10&mt;]
```

Note that the first (i.e. master) occurrence of the scope is not extracted. In practice this is usually the desired behavior, but it can be changed by setting the `plot` scope attribute `.optional = False`. For example:

```
master_phil = parse("""
  plot
    .multiple = True
    .optional = False
  {
    style = line bar pie_chart
      .type=choice
    title = None
      .type = str
  }
  plot {
    style = line
    title = Line plot (default in master)
  }
  """)
```

With the `user_phil` as before, `.show()` and `.extract()` now produce three entries each:

```
working_phil = master_phil.fetch(source=user_phil)
working_phil.show()
print working_phil.extract().plot
```

Output:

```
plot {
  style = line bar pie_chart
  title = None
}
plot {
  style = *line bar pie_chart
  title = Line plot (default in master)
}
plot {
  style = line *bar pie_chart
  title = Bar plot (provided by user)
}
[<libtbx.phil.scope_extract object at 0x2af4c307bcd0&mt;,
 <libtbx.phil.scope_extract object at 0x2af4c307bd50&mt;,
 <libtbx.phil.scope_extract object at 0x2af4c307be10&mt;]
```

With `.optional = True`, the master of a multiple definition or scope is *never* extracted. With `.optional = False`, it is *always* extracted, and always first in the list.

The "always first in the list" rule for multiple master objects is special. Other instances of multiple scopes are shown and extracted in the order in which they appear in the master file and the merged user file(s), with all *exact* duplicates removed. If duplicates are detected, the earlier copy is removed, unless it is the master.

These rules are designed to produce easily predictable results in situations where multiple Phil files are merged (via `.fetch()`), including complete copies of the master file.

fetch option: track_unused_definitions

The default behavior of `.fetch()` is to simply ignore user definitions that don't match anything in the master file. It it is possible to request a complete list of all user definitions ignored by`.fetch()`. For example:

```
master_phil = parse("""
  input {
    file_name = None
      .type = path
  }
  """)

user_phil = parse("""
  input {
    file_name = experiment.dat
    label = set1
    lable = set2
  }
  """)

working_phil, unused = master_phil.fetch(
```

```
  source=user_phil, track_unused_definitions=True)
working_phil.show()
for object_locator in unused:
  print "unused:", object_locator
```

Output:

```
input {
  file_name = experiment.dat
}
unused: input.label (input line 4)
unused: input.lable (input line 5)
```

To catch spelling errors, or to alert users to changes in the master file, it is good practice to set `track_unused_definitions=True` and to show warnings or errors.

Semicolon syntax

In all the examples above, line breaks act as syntactical elements delineating the end of definitions. This is most obvious, but for convenience, Phil also supports using the semicolon ';' instead. For example:

```
phil_scope = parse("""
   quick .multiple=true;.optional=false{and=very;.type=str;dirty=use only on
command-lines, please!;.type=str}
   """)

phil_scope.show(attributes_level=2)
```

Clearly, the output looks much nicer:

```
quick
  .optional = False
  .multiple = True
{
  and = very
    .type = str
  dirty = use only on command-lines, please!
    .type = str
}
```

Master files generally shouldn't make use of the semicolon syntax, even though it is possible. In user files it is more acceptable, but the main purpose is to support passing parameters from the command line.

Note that the Phil output methods (`.show()`, `.as_str()`) never make use of the semicolon syntax.

Phil comments

Phil supports two types of comments:

- Simple one-line comments starting with a hash '#'. All following characters through the end of the line are ignored.
- Syntax-aware comments starting with an exclamation mark '!'.

The exclamation mark can be used to easily comment out entire syntactical constructs, for example a complete scope including all attributes:

```
master_phil = parse("""
  !input {
    file_name = None
      .type = path
      .multiple = True
  }
  """)
master_phil.show()
```

Output:

```
!input {
  file_name = None
}
```

As is evident from the output, Phil keeps the content "in mind", but the scope is not actually used by `.fetch()`:

```
user_phil = parse("""
  input.file_name = experiment.dat
  """)
print len(master_phil.fetch(source=user_phil).as_str())
```

Output:

```
0
```

I.e. the `.fetch()` method ignored the user definition because the corresponding master is commented out.

Quotes and backslash

Similar to Python, Phil supports single quotes, double quotes, and triple quotes (three single or three double quotes). Unlike Python, quotes can often be omitted, and single quotes and double quotes have different semantics, similar to that of Unix shells: '$' variables are expanded if embedded in double quotes, but not if embedded in single quotes. See the variable substitution section above for examples.

The backslash can be used in the usual way (Python, Unix shells) to "escape" line breaks, quotes, and a second backslash.

For convenience, a line starting with quotes is automatically treated as a continuation of a definition on the previous line(s). The trailing backslash on the previous line may be omitted.

The exact rules for quoting and backslash escapes are intricate. A significant effort was made to mimic the familiar behavior of Python and Unix shells where possible, but nested constructs of quotes and backslashes are still prone to cause surprises. In unusual situations, probably the fastest method to obtain the desired result is trial and error (as opposed to studying the intricate rules).

scope.show(attributes_level=3)

In this document, the `scope.show()` method is used extensively in the examples. With the defaults for the method parameters, it only shows the Phil scope or definition names and and associated values. It is also possible to include some or all Phil scope or definition attributes in the `.show()` output, as directed by the `attributes_level` parameter:

```
attributes_level=0: shows only names and values
                 1: also shows the .help attribute
                 2: shows all attributes which are not None
                 3: shows all attributes
```

`scope.show(attributes_level=2)` can be used to pretty-print master files without any loss of information. `attributes_level=3` is useful to obtain a full listing of all available attributes, but all information is preserved with the usually much less verbose `attributes_level=2`. This is illustrated by the following example:

```
master_phil = parse("""
  minimization {
    input
      .help = "File names and data labels."
      .multiple = True
    {
      file_name = None
        .type = path
      label = None
        .help = "A unique substring of the data label is sufficient."
        .type = str
    }
  }
  """)

for attributes_level in range(4):
  master_phil.show(attributes_level=attributes_level)
```

Output with `attributes_level=0` (the default):

```
minimization {
```

```
  input {
    file_name = None
    label = None
  }
}
```

Output with `attributes_level=1`:

```
minimization {
  input
    .help = "File names and data labels."
  {
    file_name = None
    label = None
      .help = "A unique substring of the data label is sufficient."
  }
}
```

Output with `attributes_level=2`:

```
minimization {
  input
    .help = "File names and data labels."
    .multiple = True
  {
    file_name = None
      .type = path
    label = None
      .help = "A unique substring of the data label is sufficient."
      .type = str
  }
}
```

Output with `attributes_level=3`:

```
minimization
  .style = None
  .help = None
  .caption = None
  .short_caption = None
  .optional = None
  .call = None
  .multiple = None
  .sequential_format = None
  .disable_add = None
  .disable_delete = None
  .expert_level = None
{
  input
    .style = None
    .help = "File names and data labels."
    .caption = None
    .short_caption = None
    .optional = None
    .call = None
    .multiple = True
```

```
    .sequential_format = None
    .disable_add = None
    .disable_delete = None
    .expert_level = None
  {
    file_name = None
      .help = None
      .caption = None
      .short_caption = None
      .optional = None
      .type = path
      .multiple = None
      .input_size = None
      .expert_level = None
    label = None
      .help = "A unique substring of the data label is sufficient."
      .caption = None
      .short_caption = None
      .optional = None
      .type = str
      .multiple = None
      .input_size = None
      .expert_level = None
  }
}
```