23.3.2 Python API

You can get quick online help for GDB's Python API by issuing the command *python help (gdb)*.

Functions and methods which have two or more optional arguments allow them to be specified using keyword syntax. This allows passing some optional arguments while skipping others. Example: `gdb.some_function ('foo', bar = 1, baz = 2)`.

23.3.2.1 Basic Python

At startup, GDB overrides Python's `sys.stdout` and `sys.stderr` to print using GDB's output-paging streams. A Python program which outputs to one of these streams may have its output interrupted by the user (see Section 22.4 [Screen Size], page 344). In this situation, a Python `KeyboardInterrupt` exception is thrown.

Some care must be taken when writing Python code to run in GDB. Two things worth noting in particular:

- GDB install handlers for `SIGCHLD` and `SIGINT`. Python code must not override these, or even change the options using `sigaction`. If your program changes the handling of these signals, GDB will most likely stop working correctly. Note that it is unfortunately common for GUI toolkits to install a `SIGCHLD` handler.
- GDB takes care to mark its internal file descriptors as close-on-exec. However, this cannot be done in a thread-safe way on all platforms. Your Python programs should be aware of this and should both create new file descriptors with the close-on-exec flag set and arrange to close unneeded file descriptors before starting a child process.

GDB introduces a new Python module, named `gdb`. All methods and classes added by GDB are placed in this module. GDB automatically `import`s the `gdb` module for use in all scripts evaluated by the `python` command.

Some types of the `gdb` module come with a textual representation (accessible through the `repr` or `str` functions). These are offered for debugging purposes only, expect them to change over time.

`gdb.PYTHONDIR` [Variable]

A string containing the python directory (see Section 23.3 [Python], page 371).

`gdb.execute` (*command* [, *from_tty* [, *to_string*]]) [Function]

Evaluate *command*, a string, as a GDB CLI command. If a GDB exception happens while *command* runs, it is translated as described in Section 23.3.2.2 [Exception Handling], page 377.

The *from_tty* flag specifies whether GDB ought to consider this command as having originated from the user invoking it interactively. It must be a boolean value. If omitted, it defaults to `False`.

By default, any output produced by *command* is sent to GDB's standard output (and to the log output if logging is turned on). If the *to_string* parameter is `True`, then output will be collected by `gdb.execute` and returned as a string. The default is `False`, in which case the return value is `None`. If *to_string* is `True`, the GDB virtual terminal will be temporarily set to unlimited width and height, and its pagination will be disabled; see Section 22.4 [Screen Size], page 344.

`gdb.breakpoints ()` [Function]

Return a sequence holding all of GDB's breakpoints. See Section 23.3.2.30 [Breakpoints In Python], page 444, for more information. In GDB version 7.11 and earlier, this function returned `None` if there were no breakpoints. This peculiarity was subsequently fixed, and now `gdb.breakpoints` returns an empty sequence in this case.

`gdb.rbreak` (*regex* [, *minsyms* [, *throttle*, [, *symtabs*]]]) [Function]

Return a Python list holding a collection of newly set `gdb.Breakpoint` objects matching function names defined by the *regex* pattern. If the *minsyms* keyword is `True`, all system functions (those not explicitly defined in the inferior) will also be included in the match. The *throttle* keyword takes an integer that defines the maximum number of pattern matches for functions matched by the *regex* pattern. If the number of matches exceeds the integer value of *throttle*, a `RuntimeError` will be raised and no breakpoints will be created. If *throttle* is not defined then there is no imposed limit on the maximum number of matches and breakpoints to be created. The *symtabs* keyword takes a Python iterable that yields a collection of `gdb.Symtab` objects and will restrict the search to those functions only contained within the `gdb.Symtab` objects.

`gdb.parameter` (*parameter*) [Function]

Return the value of a GDB *parameter* given by its name, a string; the parameter name string may contain spaces if the parameter has a multi-part name. For example, '`print object`' is a valid parameter name.

If the named parameter does not exist, this function throws a `gdb.error` (see Section 23.3.2.2 [Exception Handling], page 377). Otherwise, the parameter's value is converted to a Python value of the appropriate type, and returned.

`gdb.history` (*number*) [Function]

Return a value from GDB's value history (see Section 10.11 [Value History], page 161). The *number* argument indicates which history element to return. If *number* is negative, then GDB will take its absolute value and count backward from the last element (i.e., the most recent element) to find the value to return. If *number* is zero, then GDB will return the most recent element. If the element specified by *number* doesn't exist in the value history, a `gdb.error` exception will be raised.

If no exception is raised, the return value is always an instance of `gdb.Value` (see Section 23.3.2.3 [Values From Inferior], page 378).

`gdb.convenience_variable` (*name*) [Function]

Return the value of the convenience variable (see Section 10.12 [Convenience Vars], page 162) named *name*. *name* must be a string. The name should not include the '`$`' that is used to mark a convenience variable in an expression. If the convenience variable does not exist, then `None` is returned.

`gdb.set_convenience_variable` (*name*, *value*) [Function]

Set the value of the convenience variable (see Section 10.12 [Convenience Vars], page 162) named *name*. *name* must be a string. The name should not include the '`$`' that is used to mark a convenience variable in an expression. If *value* is `None`, then the convenience variable is removed. Otherwise, if *value* is not a `gdb.Value` (see Section 23.3.2.3 [Values From Inferior], page 378), it is is converted using the `gdb.Value` constructor.

gdb.parse_and_eval (*expression*) [Function]

Parse *expression*, which must be a string, as an expression in the current language, evaluate it, and return the result as a `gdb.Value`.

This function can be useful when implementing a new command (see Section 23.3.2.20 [Commands In Python], page 421), as it provides a way to parse the command's argument as an expression. It is also useful simply to compute values.

gdb.find_pc_line (*pc*) [Function]

Return the `gdb.Symtab_and_line` object corresponding to the *pc* value. See Section 23.3.2.28 [Symbol Tables In Python], page 441. If an invalid value of *pc* is passed as an argument, then the `symtab` and `line` attributes of the returned `gdb.Symtab_and_line` object will be `None` and 0 respectively. This is identical to `gdb.current_progspace().find_pc_line(pc)` and is included for historical compatibility.

gdb.post_event (*event*) [Function]

Put *event*, a callable object taking no arguments, into GDB's internal event queue. This callable will be invoked at some later point, during GDB's event processing. Events posted using `post_event` will be run in the order in which they were posted; however, there is no way to know when they will be processed relative to other events inside GDB.

GDB is not thread-safe. If your Python program uses multiple threads, you must be careful to only call GDB-specific functions in the GDB thread. `post_event` ensures this. For example:

```
(gdb) python
>import threading
>
>class Writer():
> def __init__(self, message):
>        self.message = message;
> def __call__(self):
>        gdb.write(self.message)
>
>class MyThread1 (threading.Thread):
> def run (self):
>        gdb.post_event(Writer("Hello "))
>
>class MyThread2 (threading.Thread):
> def run (self):
>        gdb.post_event(Writer("World\n"))
>
>MyThread1().start()
>MyThread2().start()
>end
(gdb) Hello World
```

gdb.write (*string* [, *stream*]) [Function]

Print a string to GDB's paginated output stream. The optional *stream* determines the stream to print to. The default stream is GDB's standard output stream. Possible stream values are:

`gdb.STDOUT`
: GDB's standard output stream.

`gdb.STDERR`
: GDB's standard error stream.

`gdb.STDLOG`
: GDB's log stream (see Section 2.4 [Logging Output], page 20).

Writing to `sys.stdout` or `sys.stderr` will automatically call this function and will automatically direct the output to the relevant stream.

gdb.flush () [Function]

Flush the buffer of a GDB paginated stream so that the contents are displayed immediately. GDB will flush the contents of a stream automatically when it encounters a newline in the buffer. The optional *stream* determines the stream to flush. The default stream is GDB's standard output stream. Possible stream values are:

`gdb.STDOUT`
: GDB's standard output stream.

`gdb.STDERR`
: GDB's standard error stream.

`gdb.STDLOG`
: GDB's log stream (see Section 2.4 [Logging Output], page 20).

Flushing `sys.stdout` or `sys.stderr` will automatically call this function for the relevant stream.

gdb.target_charset () [Function]

Return the name of the current target character set (see Section 10.21 [Character Sets], page 177). This differs from `gdb.parameter('target-charset')` in that '`auto`' is never returned.

gdb.target_wide_charset () [Function]

Return the name of the current target wide character set (see Section 10.21 [Character Sets], page 177). This differs from `gdb.parameter('target-wide-charset')` in that '`auto`' is never returned.

gdb.solib_name (*address*) [Function]

Return the name of the shared library holding the given *address* as a string, or `None`. This is identical to `gdb.current_progspace().solib_name(address)` and is included for historical compatibility.

gdb.decode_line ([*expression*]) [Function]

Return locations of the line specified by *expression*, or of the current line if no argument was given. This function returns a Python tuple containing two elements. The first element contains a string holding any unparsed section of *expression* (or `None` if the expression has been fully parsed). The second element contains either `None` or another tuple that contains all the locations that match the expression represented as `gdb.Symtab_and_line` objects (see Section 23.3.2.28 [Symbol Tables In Python], page 441). If *expression* is provided, it is decoded the way that GDB's inbuilt `break` or `edit` commands do (see Section 9.2 [Specify Location], page 120).

`gdb.prompt_hook` (*current_prompt*) [Function]

If *prompt_hook* is callable, GDB will call the method assigned to this operation before a prompt is displayed by GDB.

The parameter `current_prompt` contains the current GDB prompt. This method must return a Python string, or `None`. If a string is returned, the GDB prompt will be set to that string. If `None` is returned, GDB will continue to use the current prompt.

Some prompts cannot be substituted in GDB. Secondary prompts such as those used by readline for command input, and annotation related prompts are prohibited from being changed.

23.3.2.2 Exception Handling

When executing the `python` command, Python exceptions uncaught within the Python code are translated to calls to GDB error-reporting mechanism. If the command that called `python` does not handle the error, GDB will terminate it and print an error message containing the Python exception name, the associated value, and the Python call stack backtrace at the point where the exception was raised. Example:

```
(gdb) python print foo
Traceback (most recent call last):
  File "<string>", line 1, in <module>
NameError: name 'foo' is not defined
```

GDB errors that happen in GDB commands invoked by Python code are converted to Python exceptions. The type of the Python exception depends on the error.

`gdb.error`

This is the base class for most exceptions generated by GDB. It is derived from `RuntimeError`, for compatibility with earlier versions of GDB.

If an error occurring in GDB does not fit into some more specific category, then the generated exception will have this type.

`gdb.MemoryError`

This is a subclass of `gdb.error` which is thrown when an operation tried to access invalid memory in the inferior.

`KeyboardInterrupt`

User interrupt (via *C-c* or by typing *q* at a pagination prompt) is translated to a Python `KeyboardInterrupt` exception.

In all cases, your exception handler will see the GDB error message as its value and the Python call stack backtrace at the Python statement closest to where the GDB error occured as the traceback.

When implementing GDB commands in Python via `gdb.Command`, or functions via `gdb.Function`, it is useful to be able to throw an exception that doesn't cause a traceback to be printed. For example, the user may have invoked the command incorrectly. GDB provides a special exception class that can be used for this purpose.

`gdb.GdbError`

When thrown from a command or function, this exception will cause the command or function to fail, but the Python stack will not be displayed. GDB does not throw this exception itself, but rather recognizes it when thrown from user Python code. Example:

```
(gdb) python
>class HelloWorld (gdb.Command):
>  """Greet the whole world."""
>  def __init__ (self):
>    super (HelloWorld, self).__init__ ("hello-world", gdb.COMMAND_USER)
>  def invoke (self, args, from_tty):
>    argv = gdb.string_to_argv (args)
>    if len (argv) != 0:
>      raise gdb.GdbError ("hello-world takes no arguments")
>    print ("Hello, World!")
>HelloWorld ()
>end
(gdb) hello-world 42
hello-world takes no arguments
```

23.3.2.3 Values From Inferior

GDB provides values it obtains from the inferior program in an object of type `gdb.Value`. GDB uses this object for its internal bookkeeping of the inferior's values, and for fetching values when necessary.

Inferior values that are simple scalars can be used directly in Python expressions that are valid for the value's data type. Here's an example for an integer or floating-point value `some_val`:

```
bar = some_val + 2
```

As result of this, `bar` will also be a `gdb.Value` object whose values are of the same type as those of `some_val`. Valid Python operations can also be performed on `gdb.Value` objects representing a `struct` or `class` object. For such cases, the overloaded operator (if present), is used to perform the operation. For example, if `val1` and `val2` are `gdb.Value` objects representing instances of a `class` which overloads the `+` operator, then one can use the `+` operator in their Python script as follows:

```
val3 = val1 + val2
```

The result of the operation `val3` is also a `gdb.Value` object corresponding to the value returned by the overloaded `+` operator. In general, overloaded operators are invoked for the following operations: `+` (binary addition), `-` (binary subtraction), `*` (multiplication), `/`, `%`, `<<`, `>>`, `|`, `&`, `^`.

Inferior values that are structures or instances of some class can be accessed using the Python *dictionary syntax*. For example, if `some_val` is a `gdb.Value` instance holding a structure, you can access its `foo` element with:

```
bar = some_val['foo']
```

Again, `bar` will also be a `gdb.Value` object. Structure elements can also be accessed by using `gdb.Field` objects as subscripts (see Section 23.3.2.4 [Types In Python], page 384, for more information on `gdb.Field` objects). For example, if `foo_field` is a `gdb.Field` object corresponding to element `foo` of the above structure, then `bar` can also be accessed as follows:

```
bar = some_val[foo_field]
```

A `gdb.Value` that represents a function can be executed via inferior function call. Any arguments provided to the call must match the function's prototype, and must be provided in the order specified by that prototype.

For example, `some_val` is a `gdb.Value` instance representing a function that takes two integers as arguments. To execute this function, call it like so:

```
result = some_val (10,20)
```

Any values returned from a function call will be stored as a `gdb.Value`.

The following attributes are provided:

`Value.address` [Variable]

If this object is addressable, this read-only attribute holds a `gdb.Value` object representing the address. Otherwise, this attribute holds `None`.

`Value.is_optimized_out` [Variable]

This read-only boolean attribute is true if the compiler optimized out this value, thus it is not available for fetching from the inferior.

`Value.type` [Variable]

The type of this `gdb.Value`. The value of this attribute is a `gdb.Type` object (see Section 23.3.2.4 [Types In Python], page 384).

`Value.dynamic_type` [Variable]

The dynamic type of this `gdb.Value`. This uses the object's virtual table and the C++ run-time type information (RTTI) to determine the dynamic type of the value. If this value is of class type, it will return the class in which the value is embedded, if any. If this value is of pointer or reference to a class type, it will compute the dynamic type of the referenced object, and return a pointer or reference to that type, respectively. In all other cases, it will return the value's static type.

Note that this feature will only work when debugging a C++ program that includes RTTI for the object in question. Otherwise, it will just return the static type of the value as in *ptype foo* (see Chapter 16 [Symbols], page 247).

`Value.is_lazy` [Variable]

The value of this read-only boolean attribute is `True` if this `gdb.Value` has not yet been fetched from the inferior. GDB does not fetch values until necessary, for efficiency. For example:

```
myval = gdb.parse_and_eval ('somevar')
```

The value of `somevar` is not fetched at this time. It will be fetched when the value is needed, or when the `fetch_lazy` method is invoked.

The following methods are provided:

`Value.__init__ (`*val*`)` [Function]

Many Python values can be converted directly to a `gdb.Value` via this object initializer. Specifically:

Python boolean
: A Python boolean is converted to the boolean type from the current language.

Python integer
: A Python integer is converted to the C `long` type for the current architecture.

Python long

A Python long is converted to the C `long long` type for the current architecture.

Python float

A Python float is converted to the C `double` type for the current architecture.

Python string

A Python string is converted to a target string in the current target language using the current target encoding. If a character cannot be represented in the current target encoding, then an exception is thrown.

`gdb.Value`

If `val` is a `gdb.Value`, then a copy of the value is made.

`gdb.LazyString`

If `val` is a `gdb.LazyString` (see Section 23.3.2.32 [Lazy Strings In Python], page 448), then the lazy string's `value` method is called, and its result is used.

`Value.__init__` (*val*, *type*) [Function]

This second form of the `gdb.Value` constructor returns a `gdb.Value` of type *type* where the value contents are taken from the Python buffer object specified by *val*. The number of bytes in the Python buffer object must be greater than or equal to the size of *type*.

`Value.cast` (*type*) [Function]

Return a new instance of `gdb.Value` that is the result of casting this instance to the type described by *type*, which must be a `gdb.Type` object. If the cast cannot be performed for some reason, this method throws an exception.

`Value.dereference` () [Function]

For pointer data types, this method returns a new `gdb.Value` object whose contents is the object pointed to by the pointer. For example, if `foo` is a C pointer to an `int`, declared in your C program as

```
int *foo;
```

then you can use the corresponding `gdb.Value` to access what `foo` points to like this:

```
bar = foo.dereference ()
```

The result `bar` will be a `gdb.Value` object holding the value pointed to by `foo`.

A similar function `Value.referenced_value` exists which also returns `gdb.Value` objects corresponding to the values pointed to by pointer values (and additionally, values referenced by reference values). However, the behavior of `Value.dereference` differs from `Value.referenced_value` by the fact that the behavior of `Value.dereference` is identical to applying the C unary operator `*` on a given value. For example, consider a reference to a pointer `ptrref`, declared in your C++ program as

```
typedef int *intptr;
...
int val = 10;
intptr ptr = &val;
```

```
intptr &ptrref = ptr;
```

Though `ptrref` is a reference value, one can apply the method `Value.dereference` to the `gdb.Value` object corresponding to it and obtain a `gdb.Value` which is identical to that corresponding to `val`. However, if you apply the method `Value.referenced_value`, the result would be a `gdb.Value` object identical to that corresponding to `ptr`.

```
py_ptrref = gdb.parse_and_eval ("ptrref")
py_val = py_ptrref.dereference ()
py_ptr = py_ptrref.referenced_value ()
```

The `gdb.Value` object `py_val` is identical to that corresponding to `val`, and `py_ptr` is identical to that corresponding to `ptr`. In general, `Value.dereference` can be applied whenever the C unary operator `*` can be applied to the corresponding C value. For those cases where applying both `Value.dereference` and `Value.referenced_value` is allowed, the results obtained need not be identical (as we have seen in the above example). The results are however identical when applied on `gdb.Value` objects corresponding to pointers (`gdb.Value` objects with type code `TYPE_CODE_PTR`) in a C/C++ program.

`Value.referenced_value ()` [Function]

For pointer or reference data types, this method returns a new `gdb.Value` object corresponding to the value referenced by the pointer/reference value. For pointer data types, `Value.dereference` and `Value.referenced_value` produce identical results. The difference between these methods is that `Value.dereference` cannot get the values referenced by reference values. For example, consider a reference to an `int`, declared in your C++ program as

```
int val = 10;
int &ref = val;
```

then applying `Value.dereference` to the `gdb.Value` object corresponding to `ref` will result in an error, while applying `Value.referenced_value` will result in a `gdb.Value` object identical to that corresponding to `val`.

```
py_ref = gdb.parse_and_eval ("ref")
er_ref = py_ref.dereference ()       # Results in error
py_val = py_ref.referenced_value ()  # Returns the referenced value
```

The `gdb.Value` object `py_val` is identical to that corresponding to `val`.

`Value.reference_value ()` [Function]

Return a `gdb.Value` object which is a reference to the value encapsulated by this instance.

`Value.const_value ()` [Function]

Return a `gdb.Value` object which is a `const` version of the value encapsulated by this instance.

`Value.dynamic_cast` (*type*) [Function]

Like `Value.cast`, but works as if the C++ `dynamic_cast` operator were used. Consult a C++ reference for details.

`Value.reinterpret_cast` (*type*) [Function]

Like `Value.cast`, but works as if the C++ `reinterpret_cast` operator were used. Consult a C++ reference for details.

`Value.format_string (...)` [Function]

Convert a `gdb.Value` to a string, similarly to what the `print` command does. Invoked with no arguments, this is equivalent to calling the `str` function on the `gdb.Value`. The representation of the same value may change across different versions of GDB, so you shouldn't, for instance, parse the strings returned by this method.

All the arguments are keyword only. If an argument is not specified, the current global default setting is used.

`raw` `True` if pretty-printers (see Section 10.10 [Pretty Printing], page 158) should not be used to format the value. `False` if enabled pretty-printers matching the type represented by the `gdb.Value` should be used to format it.

`pretty_arrays`
`True` if arrays should be pretty printed to be more convenient to read, `False` if they shouldn't (see `set print array` in Section 10.9 [Print Settings], page 148).

`pretty_structs`
`True` if structs should be pretty printed to be more convenient to read, `False` if they shouldn't (see `set print pretty` in Section 10.9 [Print Settings], page 148).

`array_indexes`
`True` if array indexes should be included in the string representation of arrays, `False` if they shouldn't (see `set print array-indexes` in Section 10.9 [Print Settings], page 148).

`symbols` `True` if the string representation of a pointer should include the corresponding symbol name (if one exists), `False` if it shouldn't (see `set print symbol` in Section 10.9 [Print Settings], page 148).

`unions` `True` if unions which are contained in other structures or unions should be expanded, `False` if they shouldn't (see `set print union` in Section 10.9 [Print Settings], page 148).

`address` `True` if the string representation of a pointer should include the address, `False` if it shouldn't (see `set print address` in Section 10.9 [Print Settings], page 148).

`deref_refs`
`True` if C++ references should be resolved to the value they refer to, `False` (the default) if they shouldn't. Note that, unlike for the `print` command, references are not automatically expanded when using the `format_string` method or the `str` function. There is no global `print` setting to change the default behaviour.

`actual_objects`
`True` if the representation of a pointer to an object should identify the *actual* (derived) type of the object rather than the *declared* type, using the virtual function table. `False` if the *declared* type should be used. (See `set print object` in Section 10.9 [Print Settings], page 148).

`static_members`
: `True` if static members should be included in the string representation of a C++ object, `False` if they shouldn't (see `set print static-members` in Section 10.9 [Print Settings], page 148).

`max_elements`
: Number of array elements to print, or `0` to print an unlimited number of elements (see `set print elements` in Section 10.9 [Print Settings], page 148).

`max_depth`
: The maximum depth to print for nested structs and unions, or `-1` to print an unlimited number of elements (see `set print max-depth` in Section 10.9 [Print Settings], page 148).

`repeat_threshold`
: Set the threshold for suppressing display of repeated array elements, or `0` to represent all elements, even if repeated. (See `set print repeats` in Section 10.9 [Print Settings], page 148).

`format`
: A string containing a single character representing the format to use for the returned string. For instance, `'x'` is equivalent to using the GDB command `print` with the `/x` option and formats the value as a hexadecimal number.

`Value.string ([`*encoding*`[, `*errors*`[, `*length*`]]])` [Function]

If this `gdb.Value` represents a string, then this method converts the contents to a Python string. Otherwise, this method will throw an exception.

Values are interpreted as strings according to the rules of the current language. If the optional length argument is given, the string will be converted to that length, and will include any embedded zeroes that the string may contain. Otherwise, for languages where the string is zero-terminated, the entire string will be converted.

For example, in C-like languages, a value is a string if it is a pointer to or an array of characters or ints of type `wchar_t`, `char16_t`, or `char32_t`.

If the optional *encoding* argument is given, it must be a string naming the encoding of the string in the `gdb.Value`, such as `"ascii"`, `"iso-8859-6"` or `"utf-8"`. It accepts the same encodings as the corresponding argument to Python's `string.decode` method, and the Python codec machinery will be used to convert the string. If *encoding* is not given, or if *encoding* is the empty string, then either the `target-charset` (see Section 10.21 [Character Sets], page 177) will be used, or a language-specific encoding will be used, if the current language is able to supply one.

The optional *errors* argument is the same as the corresponding argument to Python's `string.decode` method.

If the optional *length* argument is given, the string will be fetched and converted to the given length.

`Value.lazy_string ([`*encoding* `[, `*length*`]])` [Function]

If this `gdb.Value` represents a string, then this method converts the contents to a `gdb.LazyString` (see Section 23.3.2.32 [Lazy Strings In Python], page 448). Otherwise, this method will throw an exception.

If the optional *encoding* argument is given, it must be a string naming the encoding of the `gdb.LazyString`. Some examples are: '`ascii`', '`iso-8859-6`' or '`utf-8`'. If the *encoding* argument is an encoding that GDB does recognize, GDB will raise an error.

When a lazy string is printed, the GDB encoding machinery is used to convert the string during printing. If the optional *encoding* argument is not provided, or is an empty string, GDB will automatically select the encoding most suitable for the string type. For further information on encoding in GDB please see Section 10.21 [Character Sets], page 177.

If the optional *length* argument is given, the string will be fetched and encoded to the length of characters specified. If the *length* argument is not provided, the string will be fetched and encoded until a null of appropriate width is found.

`Value.fetch_lazy` () [Function]

If the `gdb.Value` object is currently a lazy value (`gdb.Value.is_lazy` is `True`), then the value is fetched from the inferior. Any errors that occur in the process will produce a Python exception.

If the `gdb.Value` object is not a lazy value, this method has no effect.

This method does not return a value.

23.3.2.4 Types In Python

GDB represents types from the inferior using the class `gdb.Type`.

The following type-related functions are available in the `gdb` module:

`gdb.lookup_type` (*name* [, *block*]) [Function]

This function looks up a type by its *name*, which must be a string.

If *block* is given, then *name* is looked up in that scope. Otherwise, it is searched for globally.

Ordinarily, this function will return an instance of `gdb.Type`. If the named type cannot be found, it will throw an exception.

If the type is a structure or class type, or an enum type, the fields of that type can be accessed using the Python *dictionary syntax*. For example, if `some_type` is a `gdb.Type` instance holding a structure type, you can access its `foo` field with:

```
bar = some_type['foo']
```

`bar` will be a `gdb.Field` object; see below under the description of the `Type.fields` method for a description of the `gdb.Field` class.

An instance of `Type` has the following attributes:

`Type.alignof` [Variable]

The alignment of this type, in bytes. Type alignment comes from the debugging information; if it was not specified, then GDB will use the relevant ABI to try to determine the alignment. In some cases, even this is not possible, and zero will be returned.

`Type.code` [Variable]

The type code for this type. The type code will be one of the `TYPE_CODE_` constants defined below.

`Type.dynamic` [Variable]

A boolean indicating whether this type is dynamic. In some situations, such as Rust `enum` types or Ada variant records, the concrete type of a value may vary depending on its contents. That is, the declared type of a variable, or the type returned by `gdb.lookup_type` may be dynamic; while the type of the variable's value will be a concrete instance of that dynamic type.

For example, consider this code:

```
int n;
int array[n];
```

Here, at least conceptually (whether your compiler actually does this is a separate issue), examining `gdb.lookup_symbol("array", ...).type` could yield a `gdb.Type` which reports a size of `None`. This is the dynamic type.

However, examining `gdb.parse_and_eval("array").type` would yield a concrete type, whose length would be known.

`Type.name` [Variable]

The name of this type. If this type has no name, then `None` is returned.

`Type.sizeof` [Variable]

The size of this type, in target `char` units. Usually, a target's `char` type will be an 8-bit byte. However, on some unusual platforms, this type may have a different size. A dynamic type may not have a fixed size; in this case, this attribute's value will be `None`.

`Type.tag` [Variable]

The tag name for this type. The tag name is the name after `struct`, `union`, or `enum` in C and C++; not all languages have this concept. If this type has no tag name, then `None` is returned.

`Type.objfile` [Variable]

The `gdb.Objfile` that this type was defined in, or `None` if there is no associated objfile.

The following methods are provided:

`Type.fields ()` [Function]

Return the fields of this type. The behavior depends on the type code:

- For structure and union types, this method returns the fields.
- Range types have two fields, the minimum and maximum values.
- Enum types have one field per enum constant.
- Function and method types have one field per parameter. The base types of C++ classes are also represented as fields.
- Array types have one field representing the array's range.
- If the type does not fit into one of these categories, a `TypeError` is raised.

Each field is a `gdb.Field` object, with some pre-defined attributes:

`bitpos` This attribute is not available for `enum` or `static` (as in C++) fields. The value is the position, counting in bits, from the start of the containing type. Note that, in a dynamic type, the position of a field may not be constant. In this case, the value will be `None`. Also, a dynamic type may have fields that do not appear in a corresponding concrete type.

`enumval` This attribute is only available for `enum` fields, and its value is the enumeration member's integer representation.

`name` The name of the field, or `None` for anonymous fields.

`artificial`
This is `True` if the field is artificial, usually meaning that it was provided by the compiler and not the user. This attribute is always provided, and is `False` if the field is not artificial.

`is_base_class`
This is `True` if the field represents a base class of a C++ structure. This attribute is always provided, and is `False` if the field is not a base class of the type that is the argument of `fields`, or if that type was not a C++ class.

`bitsize` If the field is packed, or is a bitfield, then this will have a non-zero value, which is the size of the field in bits. Otherwise, this will be zero; in this case the field's size is given by its type.

`type` The type of the field. This is usually an instance of `Type`, but it can be `None` in some situations.

`parent_type`
The type which contains this field. This is an instance of `gdb.Type`.

`Type.array` (*n1* [, *n2*]) [Function]
Return a new `gdb.Type` object which represents an array of this type. If one argument is given, it is the inclusive upper bound of the array; in this case the lower bound is zero. If two arguments are given, the first argument is the lower bound of the array, and the second argument is the upper bound of the array. An array's length must not be negative, but the bounds can be.

`Type.vector` (*n1* [, *n2*]) [Function]
Return a new `gdb.Type` object which represents a vector of this type. If one argument is given, it is the inclusive upper bound of the vector; in this case the lower bound is zero. If two arguments are given, the first argument is the lower bound of the vector, and the second argument is the upper bound of the vector. A vector's length must not be negative, but the bounds can be.

The difference between an `array` and a `vector` is that arrays behave like in C: when used in expressions they decay to a pointer to the first element whereas vectors are treated as first class values.

`Type.const` () [Function]
Return a new `gdb.Type` object which represents a `const`-qualified variant of this type.

`Type.volatile` () [Function]
Return a new `gdb.Type` object which represents a `volatile`-qualified variant of this type.

`Type.unqualified ()` [Function]

Return a new `gdb.Type` object which represents an unqualified variant of this type. That is, the result is neither `const` nor `volatile`.

`Type.range ()` [Function]

Return a Python `Tuple` object that contains two elements: the low bound of the argument type and the high bound of that type. If the type does not have a range, GDB will raise a `gdb.error` exception (see Section 23.3.2.2 [Exception Handling], page 377).

`Type.reference ()` [Function]

Return a new `gdb.Type` object which represents a reference to this type.

`Type.pointer ()` [Function]

Return a new `gdb.Type` object which represents a pointer to this type.

`Type.strip_typedefs ()` [Function]

Return a new `gdb.Type` that represents the real type, after removing all layers of typedefs.

`Type.target ()` [Function]

Return a new `gdb.Type` object which represents the target type of this type.

For a pointer type, the target type is the type of the pointed-to object. For an array type (meaning C-like arrays), the target type is the type of the elements of the array. For a function or method type, the target type is the type of the return value. For a complex type, the target type is the type of the elements. For a typedef, the target type is the aliased type.

If the type does not have a target, this method will throw an exception.

`Type.template_argument` (*n* [, *block*]) [Function]

If this `gdb.Type` is an instantiation of a template, this will return a new `gdb.Value` or `gdb.Type` which represents the value of the *n*th template argument (indexed starting at 0).

If this `gdb.Type` is not a template type, or if the type has fewer than *n* template arguments, this will throw an exception. Ordinarily, only C++ code will have template types.

If *block* is given, then *name* is looked up in that scope. Otherwise, it is searched for globally.

`Type.optimized_out ()` [Function]

Return `gdb.Value` instance of this type whose value is optimized out. This allows a frame decorator to indicate that the value of an argument or a local variable is not known.

Each type has a code, which indicates what category this type falls into. The available type categories are represented by constants defined in the `gdb` module:

`gdb.TYPE_CODE_PTR`

The type is a pointer.

`gdb.TYPE_CODE_ARRAY`
: The type is an array.

`gdb.TYPE_CODE_STRUCT`
: The type is a structure.

`gdb.TYPE_CODE_UNION`
: The type is a union.

`gdb.TYPE_CODE_ENUM`
: The type is an enum.

`gdb.TYPE_CODE_FLAGS`
: A bit flags type, used for things such as status registers.

`gdb.TYPE_CODE_FUNC`
: The type is a function.

`gdb.TYPE_CODE_INT`
: The type is an integer type.

`gdb.TYPE_CODE_FLT`
: A floating point type.

`gdb.TYPE_CODE_VOID`
: The special type `void`.

`gdb.TYPE_CODE_SET`
: A Pascal set type.

`gdb.TYPE_CODE_RANGE`
: A range type, that is, an integer type with bounds.

`gdb.TYPE_CODE_STRING`
: A string type. Note that this is only used for certain languages with language-defined string types; C strings are not represented this way.

`gdb.TYPE_CODE_BITSTRING`
: A string of bits. It is deprecated.

`gdb.TYPE_CODE_ERROR`
: An unknown or erroneous type.

`gdb.TYPE_CODE_METHOD`
: A method type, as found in C++.

`gdb.TYPE_CODE_METHODPTR`
: A pointer-to-member-function.

`gdb.TYPE_CODE_MEMBERPTR`
: A pointer-to-member.

`gdb.TYPE_CODE_REF`
: A reference type.

`gdb.TYPE_CODE_RVALUE_REF`
: A C++11 rvalue reference type.

`gdb.TYPE_CODE_CHAR`
A character type.

`gdb.TYPE_CODE_BOOL`
A boolean type.

`gdb.TYPE_CODE_COMPLEX`
A complex float type.

`gdb.TYPE_CODE_TYPEDEF`
A typedef to some other type.

`gdb.TYPE_CODE_NAMESPACE`
A C++ namespace.

`gdb.TYPE_CODE_DECFLOAT`
A decimal floating point type.

`gdb.TYPE_CODE_INTERNAL_FUNCTION`
A function internal to GDB. This is the type used to represent convenience functions.

Further support for types is provided in the `gdb.types` Python module (see Section 23.3.4.2 [gdb.types], page 453).

23.3.2.5 Pretty Printing API

A pretty-printer is just an object that holds a value and implements a specific interface, defined here. An example output is provided (see Section 10.10 [Pretty Printing], page 158).

`pretty_printer.children` (*self*) [Function]

GDB will call this method on a pretty-printer to compute the children of the pretty-printer's value.

This method must return an object conforming to the Python iterator protocol. Each item returned by the iterator must be a tuple holding two elements. The first element is the "name" of the child; the second element is the child's value. The value can be any Python object which is convertible to a GDB value.

This method is optional. If it does not exist, GDB will act as though the value has no children.

For efficiency, the `children` method should lazily compute its results. This will let GDB read as few elements as necessary, for example when various print settings (see Section 10.9 [Print Settings], page 148) or `-var-list-children` (see Section 27.17 [GDB/MI Variable Objects], page 575) limit the number of elements to be displayed.

Children may be hidden from display based on the value of '`set print max-depth`' (see Section 10.9 [Print Settings], page 148).

`pretty_printer.display_hint` (*self*) [Function]

The CLI may call this method and use its result to change the formatting of a value. The result will also be supplied to an MI consumer as a '`displayhint`' attribute of the variable being printed.

This method is optional. If it does exist, this method must return a string or the special value `None`.

Some display hints are predefined by GDB:

'`array`' Indicate that the object being printed is "array-like". The CLI uses this to respect parameters such as `set print elements` and `set print array`.

'`map`' Indicate that the object being printed is "map-like", and that the children of this value can be assumed to alternate between keys and values.

'`string`' Indicate that the object being printed is "string-like". If the printer's `to_string` method returns a Python string of some kind, then GDB will call its internal language-specific string-printing function to format the string. For the CLI this means adding quotation marks, possibly escaping some characters, respecting `set print elements`, and the like.

The special value `None` causes GDB to apply the default display rules.

`pretty_printer.to_string` (*self*) [Function]

GDB will call this method to display the string representation of the value passed to the object's constructor.

When printing from the CLI, if the `to_string` method exists, then GDB will prepend its result to the values returned by `children`. Exactly how this formatting is done is dependent on the display hint, and may change as more hints are added. Also, depending on the print settings (see Section 10.9 [Print Settings], page 148), the CLI may print just the result of `to_string` in a stack trace, omitting the result of `children`.

If this method returns a string, it is printed verbatim.

Otherwise, if this method returns an instance of `gdb.Value`, then GDB prints this value. This may result in a call to another pretty-printer.

If instead the method returns a Python value which is convertible to a `gdb.Value`, then GDB performs the conversion and prints the resulting value. Again, this may result in a call to another pretty-printer. Python scalars (integers, floats, and booleans) and strings are convertible to `gdb.Value`; other types are not.

Finally, if this method returns `None` then no further operations are peformed in this method and nothing is printed.

If the result is not one of these types, an exception is raised.

GDB provides a function which can be used to look up the default pretty-printer for a `gdb.Value`:

`gdb.default_visualizer` (*value*) [Function]

This function takes a `gdb.Value` object as an argument. If a pretty-printer for this value exists, then it is returned. If no such printer exists, then this returns `None`.

23.3.2.6 Selecting Pretty-Printers

GDB provides several ways to register a pretty-printer: globally, per program space, and per objfile. When choosing how to register your pretty-printer, a good rule is to register it with the smallest scope possible: that is prefer a specific objfile first, then a program space, and only register a printer globally as a last resort.

`gdb.pretty_printers` [Variable]

The Python list `gdb.pretty_printers` contains an array of functions or callable objects that have been registered via addition as a pretty-printer. Printers in this list are called `global` printers, they're available when debugging all inferiors.

Each `gdb.Progspace` contains a `pretty_printers` attribute. Each `gdb.Objfile` also contains a `pretty_printers` attribute.

Each function on these lists is passed a single `gdb.Value` argument and should return a pretty-printer object conforming to the interface definition above (see Section 23.3.2.5 [Pretty Printing API], page 389). If a function cannot create a pretty-printer for the value, it should return `None`.

GDB first checks the `pretty_printers` attribute of each `gdb.Objfile` in the current program space and iteratively calls each enabled lookup routine in the list for that `gdb.Objfile` until it receives a pretty-printer object. If no pretty-printer is found in the objfile lists, GDB then searches the pretty-printer list of the current program space, calling each enabled function until an object is returned. After these lists have been exhausted, it tries the global `gdb.pretty_printers` list, again calling each enabled function until an object is returned.

The order in which the objfiles are searched is not specified. For a given list, functions are always invoked from the head of the list, and iterated over sequentially until the end of the list, or a printer object is returned.

For various reasons a pretty-printer may not work. For example, the underlying data structure may have changed and the pretty-printer is out of date.

The consequences of a broken pretty-printer are severe enough that GDB provides support for enabling and disabling individual printers. For example, if `print frame-arguments` is on, a backtrace can become highly illegible if any argument is printed with a broken printer.

Pretty-printers are enabled and disabled by attaching an `enabled` attribute to the registered function or callable object. If this attribute is present and its value is `False`, the printer is disabled, otherwise the printer is enabled.

23.3.2.7 Writing a Pretty-Printer

A pretty-printer consists of two parts: a lookup function to detect if the type is supported, and the printer itself.

Here is an example showing how a `std::string` printer might be written. See Section 23.3.2.5 [Pretty Printing API], page 389, for details on the API this class must provide.

```
class StdStringPrinter(object):
    "Print a std::string"

    def __init__(self, val):
        self.val = val

    def to_string(self):
        return self.val['_M_dataplus']['_M_p']

    def display_hint(self):
        return 'string'
```

And here is an example showing how a lookup function for the printer example above might be written.

```
def str_lookup_function(val):
    lookup_tag = val.type.tag
    if lookup_tag == None:
        return None
    regex = re.compile("^std::basic_string<char,.*>$")
    if regex.match(lookup_tag):
        return StdStringPrinter(val)
    return None
```

The example lookup function extracts the value's type, and attempts to match it to a type that it can pretty-print. If it is a type the printer can pretty-print, it will return a printer object. If not, it returns `None`.

We recommend that you put your core pretty-printers into a Python package. If your pretty-printers are for use with a library, we further recommend embedding a version number into the package name. This practice will enable GDB to load multiple versions of your pretty-printers at the same time, because they will have different names.

You should write auto-loaded code (see Section 23.3.3 [Python Auto-loading], page 452) such that it can be evaluated multiple times without changing its meaning. An ideal auto-load file will consist solely of `import`s of your printer modules, followed by a call to a register pretty-printers with the current objfile.

Taken as a whole, this approach will scale nicely to multiple inferiors, each potentially using a different library version. Embedding a version number in the Python package name will ensure that GDB is able to load both sets of printers simultaneously. Then, because the search for pretty-printers is done by objfile, and because your auto-loaded code took care to register your library's printers with a specific objfile, GDB will find the correct printers for the specific version of the library used by each inferior.

To continue the `std::string` example (see Section 23.3.2.5 [Pretty Printing API], page 389), this code might appear in `gdb.libstdcxx.v6`:

```
def register_printers(objfile):
    objfile.pretty_printers.append(str_lookup_function)
```

And then the corresponding contents of the auto-load file would be:

```
import gdb.libstdcxx.v6
gdb.libstdcxx.v6.register_printers(gdb.current_objfile())
```

The previous example illustrates a basic pretty-printer. There are a few things that can be improved on. The printer doesn't have a name, making it hard to identify in a list of installed printers. The lookup function has a name, but lookup functions can have arbitrary, even identical, names.

Second, the printer only handles one type, whereas a library typically has several types. One could install a lookup function for each desired type in the library, but one could also have a single lookup function recognize several types. The latter is the conventional way this is handled. If a pretty-printer can handle multiple data types, then its *subprinters* are the printers for the individual data types.

The `gdb.printing` module provides a formal way of solving these problems (see Section 23.3.4.1 [gdb.printing], page 453). Here is another example that handles multiple types.

These are the types we are going to pretty-print:

```
struct foo { int a, b; };
struct bar { struct foo x, y; };
```

Here are the printers:

```
class fooPrinter:
    """Print a foo object."""

    def __init__(self, val):
        self.val = val

    def to_string(self):
        return ("a=<" + str(self.val["a"]) +
                "> b=<" + str(self.val["b"]) + ">")

class barPrinter:
    """Print a bar object."""

    def __init__(self, val):
        self.val = val

    def to_string(self):
        return ("x=<" + str(self.val["x"]) +
                "> y=<" + str(self.val["y"]) + ">")
```

This example doesn't need a lookup function, that is handled by the `gdb.printing` module. Instead a function is provided to build up the object that handles the lookup.

```
import gdb.printing

def build_pretty_printer():
    pp = gdb.printing.RegexpCollectionPrettyPrinter(
        "my_library")
    pp.add_printer('foo', '^foo$', fooPrinter)
    pp.add_printer('bar', '^bar$', barPrinter)
    return pp
```

And here is the autoload support:

```
import gdb.printing
import my_library
gdb.printing.register_pretty_printer(
    gdb.current_objfile(),
    my_library.build_pretty_printer())
```

Finally, when this printer is loaded into GDB, here is the corresponding output of '`info pretty-printer`':

```
(gdb) info pretty-printer
my_library.so:
  my_library
    foo
    bar
```

23.3.2.8 Type Printing API

GDB provides a way for Python code to customize type display. This is mainly useful for substituting canonical typedef names for types.

A *type printer* is just a Python object conforming to a certain protocol. A simple base class implementing the protocol is provided; see Section 23.3.4.2 [gdb.types], page 453. A type printer must supply at least:

`enabled` [Instance Variable of `type_printer`]

A boolean which is True if the printer is enabled, and False otherwise. This is manipulated by the `enable type-printer` and `disable type-printer` commands.

`name` [Instance Variable of `type_printer`]

The name of the type printer. This must be a string. This is used by the `enable type-printer` and `disable type-printer` commands.

`instantiate` (*self*) [Method on `type_printer`]

This is called by GDB at the start of type-printing. It is only called if the type printer is enabled. This method must return a new object that supplies a `recognize` method, as described below.

When displaying a type, say via the `ptype` command, GDB will compute a list of type recognizers. This is done by iterating first over the per-objfile type printers (see Section 23.3.2.24 [Objfiles In Python], page 430), followed by the per-progspace type printers (see Section 23.3.2.23 [Progspaces In Python], page 428), and finally the global type printers.

GDB will call the `instantiate` method of each enabled type printer. If this method returns `None`, then the result is ignored; otherwise, it is appended to the list of recognizers.

Then, when GDB is going to display a type name, it iterates over the list of recognizers. For each one, it calls the recognition function, stopping if the function returns a non-`None` value. The recognition function is defined as:

`recognize` (*self, type*) [Method on `type_recognizer`]

If *type* is not recognized, return `None`. Otherwise, return a string which is to be printed as the name of *type*. The *type* argument will be an instance of `gdb.Type` (see Section 23.3.2.4 [Types In Python], page 384).

GDB uses this two-pass approach so that type printers can efficiently cache information without holding on to it too long. For example, it can be convenient to look up type information in a type printer and hold it for a recognizer's lifetime; if a single pass were done then type printers would have to make use of the event system in order to avoid holding information that could become stale as the inferior changed.

23.3.2.9 Filtering Frames

Frame filters are Python objects that manipulate the visibility of a frame or frames when a backtrace (see Section 8.2 [Backtrace], page 108) is printed by GDB.

Only commands that print a backtrace, or, in the case of GDB/MI commands (see Chapter 27 [GDB/MI], page 525), those that return a collection of frames are affected. The commands that work with frame filters are:

`backtrace` (see [The backtrace command], page 108), `-stack-list-frames` (see [The -stack-list-frames command], page 572), `-stack-list-variables` (see [The -stack-list-variables command], page 574), `-stack-list-arguments` see [The -stack-list-arguments command], page 570) and `-stack-list-locals` (see [The -stack-list-locals command], page 574).

A frame filter works by taking an iterator as an argument, applying actions to the contents of that iterator, and returning another iterator (or, possibly, the same iterator it was provided in the case where the filter does not perform any operations). Typically, frame filters utilize tools such as the Python's `itertools` module to work with and create new iterators from the source iterator. Regardless of how a filter chooses to apply actions, it must not alter the underlying GDB frame or frames, or attempt to alter the call-stack within GDB. This preserves data integrity within GDB. Frame filters are executed on a priority basis and care should be taken that some frame filters may have been executed before, and that some frame filters will be executed after.

An important consideration when designing frame filters, and well worth reflecting upon, is that frame filters should avoid unwinding the call stack if possible. Some stacks can run very deep, into the tens of thousands in some cases. To search every frame when a frame filter executes may be too expensive at that step. The frame filter cannot know how many frames it has to iterate over, and it may have to iterate through them all. This ends up duplicating effort as GDB performs this iteration when it prints the frames. If the filter can defer unwinding frames until frame decorators are executed, after the last filter has executed, it should. See Section 23.3.2.10 [Frame Decorator API], page 396, for more information on decorators. Also, there are examples for both frame decorators and filters in later chapters. See Section 23.3.2.11 [Writing a Frame Filter], page 399, for more information.

The Python dictionary `gdb.frame_filters` contains key/object pairings that comprise a frame filter. Frame filters in this dictionary are called `global` frame filters, and they are available when debugging all inferiors. These frame filters must register with the dictionary directly. In addition to the `global` dictionary, there are other dictionaries that are loaded with different inferiors via auto-loading (see Section 23.3.3 [Python Auto-loading], page 452). The two other areas where frame filter dictionaries can be found are: `gdb.Progspace` which contains a `frame_filters` dictionary attribute, and each `gdb.Objfile` object which also contains a `frame_filters` dictionary attribute.

When a command is executed from GDB that is compatible with frame filters, GDB combines the `global`, `gdb.Progspace` and all `gdb.Objfile` dictionaries currently loaded. All of the `gdb.Objfile` dictionaries are combined, as several frames, and thus several object files, might be in use. GDB then prunes any frame filter whose `enabled` attribute is `False`. This pruned list is then sorted according to the `priority` attribute in each filter.

Once the dictionaries are combined, pruned and sorted, GDB creates an iterator which wraps each frame in the call stack in a `FrameDecorator` object, and calls each filter in order. The output from the previous filter will always be the input to the next filter, and so on.

Frame filters have a mandatory interface which each frame filter must implement, defined here:

`FrameFilter.filter` (*iterator*) [Function]

GDB will call this method on a frame filter when it has reached the order in the priority list for that filter.

For example, if there are four frame filters:

```
Name         Priority

Filter1      5
Filter2      10
Filter3      100
Filter4      1
```

The order that the frame filters will be called is:

```
Filter3 -> Filter2 -> Filter1 -> Filter4
```

Note that the output from `Filter3` is passed to the input of `Filter2`, and so on.

This `filter` method is passed a Python iterator. This iterator contains a sequence of frame decorators that wrap each `gdb.Frame`, or a frame decorator that wraps another frame decorator. The first filter that is executed in the sequence of frame filters will receive an iterator entirely comprised of default `FrameDecorator` objects. However, after each frame filter is executed, the previous frame filter may have wrapped some or all of the frame decorators with their own frame decorator. As frame decorators must also conform to a mandatory interface, these decorators can be assumed to act in a uniform manner (see Section 23.3.2.10 [Frame Decorator API], page 396).

This method must return an object conforming to the Python iterator protocol. Each item in the iterator must be an object conforming to the frame decorator interface. If a frame filter does not wish to perform any operations on this iterator, it should return that iterator untouched.

This method is not optional. If it does not exist, GDB will raise and print an error.

`FrameFilter.name` [Variable]

The `name` attribute must be Python string which contains the name of the filter displayed by GDB (see Section 8.6 [Frame Filter Management], page 116). This attribute may contain any combination of letters or numbers. Care should be taken to ensure that it is unique. This attribute is mandatory.

`FrameFilter.enabled` [Variable]

The `enabled` attribute must be Python boolean. This attribute indicates to GDB whether the frame filter is enabled, and should be considered when frame filters are executed. If `enabled` is `True`, then the frame filter will be executed when any of the backtrace commands detailed earlier in this chapter are executed. If `enabled` is `False`, then the frame filter will not be executed. This attribute is mandatory.

`FrameFilter.priority` [Variable]

The `priority` attribute must be Python integer. This attribute controls the order of execution in relation to other frame filters. There are no imposed limits on the range of `priority` other than it must be a valid integer. The higher the `priority` attribute, the sooner the frame filter will be executed in relation to other frame filters. Although `priority` can be negative, it is recommended practice to assume zero is the lowest priority that a frame filter can be assigned. Frame filters that have the same priority are executed in unsorted order in that priority slot. This attribute is mandatory. 100 is a good default priority.

23.3.2.10 Decorating Frames

Frame decorators are sister objects to frame filters (see Section 23.3.2.9 [Frame Filter API], page 394). Frame decorators are applied by a frame filter and can only be used in conjunction with frame filters.

The purpose of a frame decorator is to customize the printed content of each `gdb.Frame` in commands where frame filters are executed. This concept is called decorating a frame. Frame decorators decorate a `gdb.Frame` with Python code contained within each API call.

This separates the actual data contained in a `gdb.Frame` from the decorated data produced by a frame decorator. This abstraction is necessary to maintain integrity of the data contained in each `gdb.Frame`.

Frame decorators have a mandatory interface, defined below.

GDB already contains a frame decorator called `FrameDecorator`. This contains substantial amounts of boilerplate code to decorate the content of a `gdb.Frame`. It is recommended that other frame decorators inherit and extend this object, and only to override the methods needed.

`FrameDecorator` is defined in the Python module `gdb.FrameDecorator`, so your code can import it like:

```
from gdb.FrameDecorator import FrameDecorator
```

`FrameDecorator.elided` (*self*) [Function]

The `elided` method groups frames together in a hierarchical system. An example would be an interpreter, where multiple low-level frames make up a single call in the interpreted language. In this example, the frame filter would elide the low-level frames and present a single high-level frame, representing the call in the interpreted language, to the user.

The `elided` function must return an iterable and this iterable must contain the frames that are being elided wrapped in a suitable frame decorator. If no frames are being elided this function may return an empty iterable, or `None`. Elided frames are indented from normal frames in a `CLI` backtrace, or in the case of `GDB/MI`, are placed in the `children` field of the eliding frame.

It is the frame filter's task to also filter out the elided frames from the source iterator. This will avoid printing the frame twice.

`FrameDecorator.function` (*self*) [Function]

This method returns the name of the function in the frame that is to be printed.

This method must return a Python string describing the function, or `None`.

If this function returns `None`, GDB will not print any data for this field.

`FrameDecorator.address` (*self*) [Function]

This method returns the address of the frame that is to be printed.

This method must return a Python numeric integer type of sufficient size to describe the address of the frame, or `None`.

If this function returns a `None`, GDB will not print any data for this field.

`FrameDecorator.filename` (*self*) [Function]

This method returns the filename and path associated with this frame.

This method must return a Python string containing the filename and the path to the object file backing the frame, or `None`.

If this function returns a `None`, GDB will not print any data for this field.

`FrameDecorator.line` (*self*): [Function]

This method returns the line number associated with the current position within the function addressed by this frame.

This method must return a Python integer type, or `None`.

If this function returns a `None`, GDB will not print any data for this field.

`FrameDecorator.frame_args` (*self*) [Function]

This method must return an iterable, or `None`. Returning an empty iterable, or `None` means frame arguments will not be printed for this frame. This iterable must contain objects that implement two methods, described here.

This object must implement a `symbol` method which takes a single `self` parameter and must return a `gdb.Symbol` (see Section 23.3.2.27 [Symbols In Python], page 438), or a Python string. The object must also implement a `value` method which takes a single `self` parameter and must return a `gdb.Value` (see Section 23.3.2.3 [Values From Inferior], page 378), a Python value, or `None`. If the `value` method returns `None`, and the `argument` method returns a `gdb.Symbol`, GDB will look-up and print the value of the `gdb.Symbol` automatically.

A brief example:

```
class SymValueWrapper():

    def __init__(self, symbol, value):
        self.sym = symbol
        self.val = value

    def value(self):
        return self.val

    def symbol(self):
        return self.sym

class SomeFrameDecorator()
...
...
    def frame_args(self):
        args = []
        try:
            block = self.inferior_frame.block()
        except:
            return None

        # Iterate over all symbols in a block.  Only add
        # symbols that are arguments.
        for sym in block:
            if not sym.is_argument:
                continue
            args.append(SymValueWrapper(sym,None))

        # Add example synthetic argument.
        args.append(SymValueWrapper(``foo'', 42))

        return args
```

`FrameDecorator.frame_locals` (*self*) [Function]

This method must return an iterable or `None`. Returning an empty iterable, or `None` means frame local arguments will not be printed for this frame.

The object interface, the description of the various strategies for reading frame locals, and the example are largely similar to those described in the `frame_args` function, (see [The frame filter frame_args function], page 398). Below is a modified example:

```
class SomeFrameDecorator()
...
...
    def frame_locals(self):
        vars = []
        try:
            block = self.inferior_frame.block()
        except:
            return None

        # Iterate over all symbols in a block.  Add all
        # symbols, except arguments.
        for sym in block:
            if sym.is_argument:
                continue
            vars.append(SymValueWrapper(sym,None))

        # Add an example of a synthetic local variable.
        vars.append(SymValueWrapper(``bar'', 99))

        return vars
```

`FrameDecorator.inferior_frame` (*self*): [Function]

This method must return the underlying `gdb.Frame` that this frame decorator is decorating. GDB requires the underlying frame for internal frame information to determine how to print certain values when printing a frame.

23.3.2.11 Writing a Frame Filter

There are three basic elements that a frame filter must implement: it must correctly implement the documented interface (see Section 23.3.2.9 [Frame Filter API], page 394), it must register itself with GDB, and finally, it must decide if it is to work on the data provided by GDB. In all cases, whether it works on the iterator or not, each frame filter must return an iterator. A bare-bones frame filter follows the pattern in the following example.

```
import gdb

class FrameFilter():

    def __init__(self):
        # Frame filter attribute creation.
        #
        # 'name' is the name of the filter that GDB will display.
        #
        # 'priority' is the priority of the filter relative to other
        # filters.
        #
        # 'enabled' is a boolean that indicates whether this filter is
        # enabled and should be executed.

        self.name = "Foo"
        self.priority = 100
        self.enabled = True
```

```
            # Register this frame filter with the global frame_filters
            # dictionary.
            gdb.frame_filters[self.name] = self

        def filter(self, frame_iter):
            # Just return the iterator.
            return frame_iter
```

The frame filter in the example above implements the three requirements for all frame filters. It implements the API, self registers, and makes a decision on the iterator (in this case, it just returns the iterator untouched).

The first step is attribute creation and assignment, and as shown in the comments the filter assigns the following attributes: `name`, `priority` and whether the filter should be enabled with the `enabled` attribute.

The second step is registering the frame filter with the dictionary or dictionaries that the frame filter has interest in. As shown in the comments, this filter just registers itself with the global dictionary `gdb.frame_filters`. As noted earlier, `gdb.frame_filters` is a dictionary that is initialized in the `gdb` module when GDB starts. What dictionary a filter registers with is an important consideration. Generally, if a filter is specific to a set of code, it should be registered either in the `objfile` or `progspace` dictionaries as they are specific to the program currently loaded in GDB. The global dictionary is always present in GDB and is never unloaded. Any filters registered with the global dictionary will exist until GDB exits. To avoid filters that may conflict, it is generally better to register frame filters against the dictionaries that more closely align with the usage of the filter currently in question. See Section 23.3.3 [Python Auto-loading], page 452, for further information on auto-loading Python scripts.

GDB takes a hands-off approach to frame filter registration, therefore it is the frame filter's responsibility to ensure registration has occurred, and that any exceptions are handled appropriately. In particular, you may wish to handle exceptions relating to Python dictionary key uniqueness. It is mandatory that the dictionary key is the same as frame filter's `name` attribute. When a user manages frame filters (see Section 8.6 [Frame Filter Management], page 116), the names GDB will display are those contained in the `name` attribute.

The final step of this example is the implementation of the `filter` method. As shown in the example comments, we define the `filter` method and note that the method must take an iterator, and also must return an iterator. In this bare-bones example, the frame filter is not very useful as it just returns the iterator untouched. However this is a valid operation for frame filters that have the `enabled` attribute set, but decide not to operate on any frames.

In the next example, the frame filter operates on all frames and utilizes a frame decorator to perform some work on the frames. See Section 23.3.2.10 [Frame Decorator API], page 396, for further information on the frame decorator interface.

This example works on inlined frames. It highlights frames which are inlined by tagging them with an "[inlined]" tag. By applying a frame decorator to all frames with the Python `itertools imap` method, the example defers actions to the frame decorator. Frame decorators are only processed when GDB prints the backtrace.

This introduces a new decision making topic: whether to perform decision making operations at the filtering step, or at the printing step. In this example's approach, it does not perform any filtering decisions at the filtering step beyond mapping a frame decorator to each frame. This allows the actual decision making to be performed when each frame is printed. This is an important consideration, and well worth reflecting upon when designing a frame filter. An issue that frame filters should avoid is unwinding the stack if possible. Some stacks can run very deep, into the tens of thousands in some cases. To search every frame to determine if it is inlined ahead of time may be too expensive at the filtering step. The frame filter cannot know how many frames it has to iterate over, and it would have to iterate through them all. This ends up duplicating effort as GDB performs this iteration when it prints the frames.

In this example decision making can be deferred to the printing step. As each frame is printed, the frame decorator can examine each frame in turn when GDB iterates. From a performance viewpoint, this is the most appropriate decision to make as it avoids duplicating the effort that the printing step would undertake anyway. Also, if there are many frame filters unwinding the stack during filtering, it can substantially delay the printing of the backtrace which will result in large memory usage, and a poor user experience.

```
class InlineFilter():

    def __init__(self):
        self.name = "InlinedFrameFilter"
        self.priority = 100
        self.enabled = True
        gdb.frame_filters[self.name] = self

    def filter(self, frame_iter):
        frame_iter = itertools.imap(InlinedFrameDecorator,
                                    frame_iter)
        return frame_iter
```

This frame filter is somewhat similar to the earlier example, except that the `filter` method applies a frame decorator object called `InlinedFrameDecorator` to each element in the iterator. The `imap` Python method is light-weight. It does not proactively iterate over the iterator, but rather creates a new iterator which wraps the existing one.

Below is the frame decorator for this example.

```
class InlinedFrameDecorator(FrameDecorator):

    def __init__(self, fobj):
        super(InlinedFrameDecorator, self).__init__(fobj)

    def function(self):
        frame = fobj.inferior_frame()
        name = str(frame.name())

        if frame.type() == gdb.INLINE_FRAME:
            name = name + " [inlined]"

        return name
```

This frame decorator only defines and overrides the `function` method. It lets the supplied `FrameDecorator`, which is shipped with GDB, perform the other work associated with printing this frame.

The combination of these two objects create this output from a backtrace:

```
#0  0x004004e0 in bar () at inline.c:11
```

```
#1  0x00400566 in max [inlined] (b=6, a=12) at inline.c:21
#2  0x00400566 in main () at inline.c:31
```

So in the case of this example, a frame decorator is applied to all frames, regardless of whether they may be inlined or not. As GDB iterates over the iterator produced by the frame filters, GDB executes each frame decorator which then makes a decision on what to print in the `function` callback. Using a strategy like this is a way to defer decisions on the frame content to printing time.

Eliding Frames

It might be that the above example is not desirable for representing inlined frames, and a hierarchical approach may be preferred. If we want to hierarchically represent frames, the `elided` frame decorator interface might be preferable.

This example approaches the issue with the `elided` method. This example is quite long, but very simplistic. It is out-of-scope for this section to write a complete example that comprehensively covers all approaches of finding and printing inlined frames. However, this example illustrates the approach an author might use.

This example comprises of three sections.

```
class InlineFrameFilter():

    def __init__(self):
        self.name = "InlinedFrameFilter"
        self.priority = 100
        self.enabled = True
        gdb.frame_filters[self.name] = self

    def filter(self, frame_iter):
        return ElidingInlineIterator(frame_iter)
```

This frame filter is very similar to the other examples. The only difference is this frame filter is wrapping the iterator provided to it (`frame_iter`) with a custom iterator called `ElidingInlineIterator`. This again defers actions to when GDB prints the backtrace, as the iterator is not traversed until printing.

The iterator for this example is as follows. It is in this section of the example where decisions are made on the content of the backtrace.

```
class ElidingInlineIterator:
    def __init__(self, ii):
        self.input_iterator = ii

    def __iter__(self):
        return self

    def next(self):
        frame = next(self.input_iterator)

        if frame.inferior_frame().type() != gdb.INLINE_FRAME:
            return frame

        try:
            eliding_frame = next(self.input_iterator)
        except StopIteration:
            return frame
        return ElidingFrameDecorator(eliding_frame, [frame])
```

This iterator implements the Python iterator protocol. When the `next` function is called (when GDB prints each frame), the iterator checks if this frame decorator, `frame`, is wrapping an inlined frame. If it is not, it returns the existing frame decorator untouched. If it is wrapping an inlined frame, it assumes that the inlined frame was contained within the next oldest frame, `eliding_frame`, which it fetches. It then creates and returns a frame decorator, `ElidingFrameDecorator`, which contains both the elided frame, and the eliding frame.

```
class ElidingInlineDecorator(FrameDecorator):

    def __init__(self, frame, elided_frames):
        super(ElidingInlineDecorator, self).__init__(frame)
        self.frame = frame
        self.elided_frames = elided_frames

    def elided(self):
        return iter(self.elided_frames)
```

This frame decorator overrides one function and returns the inlined frame in the `elided` method. As before it lets `FrameDecorator` do the rest of the work involved in printing this frame. This produces the following output.

```
#0  0x004004e0 in bar () at inline.c:11
#2  0x00400529 in main () at inline.c:25
    #1  0x00400529 in max (b=6, a=12) at inline.c:15
```

In that output, `max` which has been inlined into `main` is printed hierarchically. Another approach would be to combine the `function` method, and the `elided` method to both print a marker in the inlined frame, and also show the hierarchical relationship.

23.3.2.12 Unwinding Frames in Python

In GDB terminology "unwinding" is the process of finding the previous frame (that is, caller's) from the current one. An unwinder has three methods. The first one checks if it can handle given frame ("sniff" it). For the frames it can sniff an unwinder provides two additional methods: it can return frame's ID, and it can fetch registers from the previous frame. A running GDB mantains a list of the unwinders and calls each unwinder's sniffer in turn until it finds the one that recognizes the current frame. There is an API to register an unwinder.

The unwinders that come with GDB handle standard frames. However, mixed language applications (for example, an application running Java Virtual Machine) sometimes use frame layouts that cannot be handled by the GDB unwinders. You can write Python code that can handle such custom frames.

You implement a frame unwinder in Python as a class with which has two attributes, `name` and `enabled`, with obvious meanings, and a single method `__call__`, which examines a given frame and returns an object (an instance of `gdb.UnwindInfo class)` describing it. If an unwinder does not recognize a frame, it should return `None`. The code in GDB that enables writing unwinders in Python uses this object to return frame's ID and previous frame registers when GDB core asks for them.

An unwinder should do as little work as possible. Some otherwise innocuous operations can cause problems (even crashes, as this code is not not well-hardened yet). For example, making an inferior call from an unwinder is unadvisable, as an inferior call will reset GDB's stack unwinding process, potentially causing re-entrant unwinding.

Unwinder Input

An object passed to an unwinder (a `gdb.PendingFrame` instance) provides a method to read frame's registers:

`PendingFrame.read_register` (*reg*) [Function]

This method returns the contents of the register *reg* in the frame as a `gdb.Value` object. For a description of the acceptable values of *reg* see [Frame.read_register], page 435. If *reg* does not name a register for the current architecture, this method will throw an exception.

Note that this method will always return a `gdb.Value` for a valid register name. This does not mean that the value will be valid. For example, you may request a register that an earlier unwinder could not unwind—the value will be unavailable. Instead, the `gdb.Value` returned from this method will be lazy; that is, its underlying bits will not be fetched until it is first used. So, attempting to use such a value will cause an exception at the point of use.

The type of the returned `gdb.Value` depends on the register and the architecture. It is common for registers to have a scalar type, like `long long`; but many other types are possible, such as pointer, pointer-to-function, floating point or vector types.

It also provides a factory method to create a `gdb.UnwindInfo` instance to be returned to GDB:

`PendingFrame.create_unwind_info` (*frame_id*) [Function]

Returns a new `gdb.UnwindInfo` instance identified by given *frame_id*. The argument is used to build GDB's frame ID using one of functions provided by GDB. *frame_id*'s attributes determine which function will be used, as follows:

`sp, pc`
: The frame is identified by the given stack address and PC. The stack address must be chosen so that it is constant throughout the lifetime of the frame, so a typical choice is the value of the stack pointer at the start of the function—in the DWARF standard, this would be the "Call Frame Address".

  This is the most common case by far. The other cases are documented for completeness but are only useful in specialized situations.

`sp, pc, special`
: The frame is identified by the stack address, the PC, and a "special" address. The special address is used on architectures that can have frames that do not change the stack, but which are still distinct, for example the IA-64, which has a second stack for registers. Both *sp* and *special* must be constant throughout the lifetime of the frame.

`sp`
: The frame is identified by the stack address only. Any other stack frame with a matching *sp* will be considered to match this frame. Inside gdb, this is called a "wild frame". You will never need this.

Each attribute value should be an instance of `gdb.Value`.

PendingFrame.architecture () [Function]

Return the gdb.Architecture (see Section 23.3.2.33 [Architectures In Python], page 449) for this gdb.PendingFrame. This represents the architecture of the particular frame being unwound.

Unwinder Output: UnwindInfo

Use PendingFrame.create_unwind_info method described above to create a gdb.UnwindInfo instance. Use the following method to specify caller registers that have been saved in this frame:

gdb.UnwindInfo.add_saved_register (*reg, value*) [Function]

reg identifies the register, for a description of the acceptable values see [Frame.read_register], page 435. *value* is a register value (a gdb.Value object).

Unwinder Skeleton Code

GDB comes with the module containing the base Unwinder class. Derive your unwinder class from it and structure the code as follows:

```
from gdb.unwinders import Unwinder

class FrameId(object):
    def __init__(self, sp, pc):
        self.sp = sp
        self.pc = pc


class MyUnwinder(Unwinder):
    def __init__(....):
        super(MyUnwinder, self).__init___(<expects unwinder name argument>)

    def __call__(pending_frame):
        if not <we recognize frame>:
            return None
        # Create UnwindInfo.  Usually the frame is identified by the stack
        # pointer and the program counter.
        sp = pending_frame.read_register(<SP number>)
        pc = pending_frame.read_register(<PC number>)
        unwind_info = pending_frame.create_unwind_info(FrameId(sp, pc))

        # Find the values of the registers in the caller's frame and
        # save them in the result:
        unwind_info.add_saved_register(<register>, <value>)
        ....

        # Return the result:
        return unwind_info
```

Registering a Unwinder

An object file, a program space, and the GDB proper can have unwinders registered with it.

The gdb.unwinders module provides the function to register a unwinder:

`gdb.unwinder.register_unwinder` (*locus, unwinder, replace=False*) [Function]

locus is specifies an object file or a program space to which *unwinder* is added. Passing `None` or `gdb` adds *unwinder* to the GDB's global unwinder list. The newly added *unwinder* will be called before any other unwinder from the same locus. Two unwinders in the same locus cannot have the same name. An attempt to add a unwinder with already existing name raises an exception unless *replace* is `True`, in which case the old unwinder is deleted.

Unwinder Precedence

GDB first calls the unwinders from all the object files in no particular order, then the unwinders from the current program space, and finally the unwinders from GDB.

23.3.2.13 Xmethods In Python

Xmethods are additional methods or replacements for existing methods of a C++ class. This feature is useful for those cases where a method defined in C++ source code could be inlined or optimized out by the compiler, making it unavailable to GDB. For such cases, one can define an xmethod to serve as a replacement for the method defined in the C++ source code. GDB will then invoke the xmethod, instead of the C++ method, to evaluate expressions. One can also use xmethods when debugging with core files. Moreover, when debugging live programs, invoking an xmethod need not involve running the inferior (which can potentially perturb its state). Hence, even if the C++ method is available, it is better to use its replacement xmethod if one is defined.

The xmethods feature in Python is available via the concepts of an *xmethod matcher* and an *xmethod worker*. To implement an xmethod, one has to implement a matcher and a corresponding worker for it (more than one worker can be implemented, each catering to a different overloaded instance of the method). Internally, GDB invokes the `match` method of a matcher to match the class type and method name. On a match, the `match` method returns a list of matching *worker* objects. Each worker object typically corresponds to an overloaded instance of the xmethod. They implement a `get_arg_types` method which returns a sequence of types corresponding to the arguments the xmethod requires. GDB uses this sequence of types to perform overload resolution and picks a winning xmethod worker. A winner is also selected from among the methods GDB finds in the C++ source code. Next, the winning xmethod worker and the winning C++ method are compared to select an overall winner. In case of a tie between a xmethod worker and a C++ method, the xmethod worker is selected as the winner. That is, if a winning xmethod worker is found to be equivalent to the winning C++ method, then the xmethod worker is treated as a replacement for the C++ method. GDB uses the overall winner to invoke the method. If the winning xmethod worker is the overall winner, then the corresponding xmethod is invoked via the `__call__` method of the worker object.

If one wants to implement an xmethod as a replacement for an existing C++ method, then they have to implement an equivalent xmethod which has exactly the same name and takes arguments of exactly the same type as the C++ method. If the user wants to invoke the C++ method even though a replacement xmethod is available for that method, then they can disable the xmethod.

See Section 23.3.2.14 [Xmethod API], page 407, for API to implement xmethods in Python. See Section 23.3.2.15 [Writing an Xmethod], page 408, for implementing xmethods in Python.

23.3.2.14 Xmethod API

The GDB Python API provides classes, interfaces and functions to implement, register and manipulate xmethods. See Section 23.3.2.13 [Xmethods In Python], page 406.

An xmethod matcher should be an instance of a class derived from `XMethodMatcher` defined in the module `gdb.xmethod`, or an object with similar interface and attributes. An instance of `XMethodMatcher` has the following attributes:

`name` [Variable]

The name of the matcher.

`enabled` [Variable]

A boolean value indicating whether the matcher is enabled or disabled.

`methods` [Variable]

A list of named methods managed by the matcher. Each object in the list is an instance of the class `XMethod` defined in the module `gdb.xmethod`, or any object with the following attributes:

`name` Name of the xmethod which should be unique for each xmethod managed by the matcher.

`enabled` A boolean value indicating whether the xmethod is enabled or disabled.

The class `XMethod` is a convenience class with same attributes as above along with the following constructor:

`XMethod.__init__` (*self, name*) [Function]

Constructs an enabled xmethod with name *name*.

The `XMethodMatcher` class has the following methods:

`XMethodMatcher.__init__` (*self, name*) [Function]

Constructs an enabled xmethod matcher with name *name*. The `methods` attribute is initialized to `None`.

`XMethodMatcher.match` (*self, class_type, method_name*) [Function]

Derived classes should override this method. It should return a xmethod worker object (or a sequence of xmethod worker objects) matching the *class_type* and *method_name*. *class_type* is a `gdb.Type` object, and *method_name* is a string value. If the matcher manages named methods as listed in its `methods` attribute, then only those worker objects whose corresponding entries in the `methods` list are enabled should be returned.

An xmethod worker should be an instance of a class derived from `XMethodWorker` defined in the module `gdb.xmethod`, or support the following interface:

`XMethodWorker.get_arg_types` (*self*) [Function]

This method returns a sequence of `gdb.Type` objects corresponding to the arguments that the xmethod takes. It can return an empty sequence or `None` if the xmethod does not take any arguments. If the xmethod takes a single argument, then a single `gdb.Type` object corresponding to it can be returned.

`XMethodWorker.get_result_type` (*self, *args*) [Function]

This method returns a `gdb.Type` object representing the type of the result of invoking this xmethod. The *args* argument is the same tuple of arguments that would be passed to the `__call__` method of this worker.

`XMethodWorker.__call__` (*self, *args*) [Function]

This is the method which does the *work* of the xmethod. The *args* arguments is the tuple of arguments to the xmethod. Each element in this tuple is a gdb.Value object. The first element is always the `this` pointer value.

For GDB to lookup xmethods, the xmethod matchers should be registered using the following function defined in the module `gdb.xmethod`:

`register_xmethod_matcher` (*locus, matcher, replace=False*) [Function]

The `matcher` is registered with `locus`, replacing an existing matcher with the same name as `matcher` if `replace` is `True`. `locus` can be a `gdb.Objfile` object (see Section 23.3.2.24 [Objfiles In Python], page 430), or a `gdb.Progspace` object (see Section 23.3.2.23 [Progspaces In Python], page 428), or `None`. If it is `None`, then `matcher` is registered globally.

23.3.2.15 Writing an Xmethod

Implementing xmethods in Python will require implementing xmethod matchers and xmethod workers (see Section 23.3.2.13 [Xmethods In Python], page 406). Consider the following C++ class:

```
class MyClass
{
public:
  MyClass (int a) : a_(a) { }

  int geta (void) { return a_; }
  int operator+ (int b);

private:
  int a_;
};

int
MyClass::operator+ (int b)
{
  return a_ + b;
}
```

Let us define two xmethods for the class `MyClass`, one replacing the method `geta`, and another adding an overloaded flavor of `operator+` which takes a `MyClass` argument (the C++ code above already has an overloaded `operator+` which takes an `int` argument). The xmethod matcher can be defined as follows:

```
class MyClass_geta(gdb.xmethod.XMethod):
```

```
    def __init__(self):
        gdb.xmethod.XMethod.__init__(self, 'geta')

    def get_worker(self, method_name):
        if method_name == 'geta':
            return MyClassWorker_geta()


class MyClass_sum(gdb.xmethod.XMethod):
    def __init__(self):
        gdb.xmethod.XMethod.__init__(self, 'sum')

    def get_worker(self, method_name):
        if method_name == 'operator+':
            return MyClassWorker_plus()


class MyClassMatcher(gdb.xmethod.XMethodMatcher):
    def __init__(self):
        gdb.xmethod.XMethodMatcher.__init__(self, 'MyClassMatcher')
        # List of methods 'managed' by this matcher
        self.methods = [MyClass_geta(), MyClass_sum()]

    def match(self, class_type, method_name):
        if class_type.tag != 'MyClass':
            return None
        workers = []
        for method in self.methods:
            if method.enabled:
                worker = method.get_worker(method_name)
                if worker:
                    workers.append(worker)

        return workers
```

Notice that the `match` method of `MyClassMatcher` returns a worker object of type `MyClassWorker_geta` for the `geta` method, and a worker object of type `MyClassWorker_plus` for the `operator+` method. This is done indirectly via helper classes derived from `gdb.xmethod.XMethod`. One does not need to use the `methods` attribute in a matcher as it is optional. However, if a matcher manages more than one xmethod, it is a good practice to list the xmethods in the `methods` attribute of the matcher. This will then facilitate enabling and disabling individual xmethods via the `enable/disable` commands. Notice also that a worker object is returned only if the corresponding entry in the `methods` attribute of the matcher is enabled.

The implementation of the worker classes returned by the matcher setup above is as follows:

```
class MyClassWorker_geta(gdb.xmethod.XMethodWorker):
    def get_arg_types(self):
        return None

    def get_result_type(self, obj):
        return gdb.lookup_type('int')

    def __call__(self, obj):
        return obj['a_']
```

```
class MyClassWorker_plus(gdb.xmethod.XMethodWorker):
    def get_arg_types(self):
        return gdb.lookup_type('MyClass')

    def get_result_type(self, obj):
        return gdb.lookup_type('int')

    def __call__(self, obj, other):
        return obj['a_'] + other['a_']
```

For GDB to actually lookup a xmethod, it has to be registered with it. The matcher defined above is registered with GDB globally as follows:

```
gdb.xmethod.register_xmethod_matcher(None, MyClassMatcher())
```

If an object `obj` of type `MyClass` is initialized in C++ code as follows:

```
MyClass obj(5);
```

then, after loading the Python script defining the xmethod matchers and workers into `GDBN`, invoking the method `geta` or using the operator `+` on `obj` will invoke the xmethods defined above:

```
(gdb) p obj.geta()
$1 = 5

(gdb) p obj + obj
$2 = 10
```

Consider another example with a C++ template class:

```
template <class T>
class MyTemplate
{
public:
  MyTemplate () : dsize_(10), data_ (new T [10]) { }
  ~MyTemplate () { delete [] data_; }

  int footprint (void)
  {
    return sizeof (T) * dsize_ + sizeof (MyTemplate<T>);
  }

private:
  int dsize_;
  T *data_;
};
```

Let us implement an xmethod for the above class which serves as a replacement for the `footprint` method. The full code listing of the xmethod workers and xmethod matchers is as follows:

```
class MyTemplateWorker_footprint(gdb.xmethod.XMethodWorker):
    def __init__(self, class_type):
        self.class_type = class_type

    def get_arg_types(self):
        return None

    def get_result_type(self):
        return gdb.lookup_type('int')
```

```
    def __call__(self, obj):
        return (self.class_type.sizeof +
                obj['dsize_'] *
                self.class_type.template_argument(0).sizeof)


class MyTemplateMatcher_footprint(gdb.xmethod.XMethodMatcher):
    def __init__(self):
        gdb.xmethod.XMethodMatcher.__init__(self, 'MyTemplateMatcher')

    def match(self, class_type, method_name):
        if (re.match('MyTemplate<[ \t\n]*[_a-zA-Z][ _a-zA-Z0-9]*>',
                     class_type.tag) and
            method_name == 'footprint'):
            return MyTemplateWorker_footprint(class_type)
```

Notice that, in this example, we have not used the `methods` attribute of the matcher as the matcher manages only one xmethod. The user can enable/disable this xmethod by enabling/disabling the matcher itself.

23.3.2.16 Inferiors In Python

Programs which are being run under GDB are called inferiors (see Section 4.9 [Inferiors Connections and Programs], page 40). Python scripts can access information about and manipulate inferiors controlled by GDB via objects of the `gdb.Inferior` class.

The following inferior-related functions are available in the `gdb` module:

`gdb.inferiors ()` [Function]
: Return a tuple containing all inferior objects.

`gdb.selected_inferior ()` [Function]
: Return an object representing the current inferior.

A `gdb.Inferior` object has the following attributes:

`Inferior.num` [Variable]
: ID of inferior, as assigned by GDB.

`Inferior.connection_num` [Variable]
: ID of inferior's connection as assigned by GDB, or None if the inferior is not connected to a target. See Section 4.9 [Inferiors Connections and Programs], page 40.

`Inferior.pid` [Variable]
: Process ID of the inferior, as assigned by the underlying operating system.

`Inferior.was_attached` [Variable]
: Boolean signaling whether the inferior was created using 'attach', or started by GDB itself.

`Inferior.progspace` [Variable]
: The inferior's program space. See Section 23.3.2.23 [Progspaces In Python], page 428.

A `gdb.Inferior` object has the following methods:

`Inferior.is_valid ()` [Function]

Returns `True` if the `gdb.Inferior` object is valid, `False` if not. A `gdb.Inferior` object will become invalid if the inferior no longer exists within GDB. All other `gdb.Inferior` methods will throw an exception if it is invalid at the time the method is called.

`Inferior.threads ()` [Function]

This method returns a tuple holding all the threads which are valid when it is called. If there are no valid threads, the method will return an empty tuple.

`Inferior.architecture ()` [Function]

Return the `gdb.Architecture` (see Section 23.3.2.33 [Architectures In Python], page 449) for this inferior. This represents the architecture of the inferior as a whole. Some platforms can have multiple architectures in a single address space, so this may not match the architecture of a particular frame (see Section 23.3.2.25 [Frames In Python], page 432).

`Inferior.read_memory` (*address, length*) [Function]

Read *length* addressable memory units from the inferior, starting at *address*. Returns a buffer object, which behaves much like an array or a string. It can be modified and given to the `Inferior.write_memory` function. In Python 3, the return value is a `memoryview` object.

`Inferior.write_memory` (*address, buffer* [, *length*]) [Function]

Write the contents of *buffer* to the inferior, starting at *address*. The *buffer* parameter must be a Python object which supports the buffer protocol, i.e., a string, an array or the object returned from `Inferior.read_memory`. If given, *length* determines the number of addressable memory units from *buffer* to be written.

`Inferior.search_memory` (*address, length, pattern*) [Function]

Search a region of the inferior memory starting at *address* with the given *length* using the search pattern supplied in *pattern*. The *pattern* parameter must be a Python object which supports the buffer protocol, i.e., a string, an array or the object returned from `gdb.read_memory`. Returns a Python `Long` containing the address where the pattern was found, or `None` if the pattern could not be found.

`Inferior.thread_from_handle` (*handle*) [Function]

Return the thread object corresponding to *handle*, a thread library specific data structure such as `pthread_t` for pthreads library implementations.

The function `Inferior.thread_from_thread_handle` provides the same functionality, but use of `Inferior.thread_from_thread_handle` is deprecated.

23.3.2.17 Events In Python

GDB provides a general event facility so that Python code can be notified of various state changes, particularly changes that occur in the inferior.

An *event* is just an object that describes some state change. The type of the object and its attributes will vary depending on the details of the change. All the existing events are described below.

In order to be notified of an event, you must register an event handler with an *event registry*. An event registry is an object in the `gdb.events` module which dispatches particular events. A registry provides methods to register and unregister event handlers:

`EventRegistry.connect` (*object*) [Function]
: Add the given callable *object* to the registry. This object will be called when an event corresponding to this registry occurs.

`EventRegistry.disconnect` (*object*) [Function]
: Remove the given *object* from the registry. Once removed, the object will no longer receive notifications of events.

Here is an example:

```
def exit_handler (event):
    print ("event type: exit")
    print ("exit code: %d" % (event.exit_code))

gdb.events.exited.connect (exit_handler)
```

In the above example we connect our handler `exit_handler` to the registry `events.exited`. Once connected, `exit_handler` gets called when the inferior exits. The argument *event* in this example is of type `gdb.ExitedEvent`. As you can see in the example the `ExitedEvent` object has an attribute which indicates the exit code of the inferior.

The following is a listing of the event registries that are available and details of the events they emit:

`events.cont`

Emits `gdb.ThreadEvent`.

Some events can be thread specific when GDB is running in non-stop mode. When represented in Python, these events all extend `gdb.ThreadEvent`. Note, this event is not emitted directly; instead, events which are emitted by this or other modules might extend this event. Examples of these events are `gdb.BreakpointEvent` and `gdb.ContinueEvent`.

`ThreadEvent.inferior_thread` [Variable]
: In non-stop mode this attribute will be set to the specific thread which was involved in the emitted event. Otherwise, it will be set to `None`.

Emits `gdb.ContinueEvent` which extends `gdb.ThreadEvent`.

This event indicates that the inferior has been continued after a stop. For inherited attribute refer to `gdb.ThreadEvent` above.

`events.exited`

Emits `events.ExitedEvent` which indicates that the inferior has exited. `events.ExitedEvent` has two attributes:

`ExitedEvent.exit_code` [Variable]
: An integer representing the exit code, if available, which the inferior has returned. (The exit code could be unavailable if, for example, GDB detaches from the inferior.) If the exit code is unavailable, the attribute does not exist.

`ExitedEvent.inferior` [Variable]
A reference to the inferior which triggered the `exited` event.

`events.stop`
Emits `gdb.StopEvent` which extends `gdb.ThreadEvent`.

Indicates that the inferior has stopped. All events emitted by this registry extend StopEvent. As a child of `gdb.ThreadEvent`, `gdb.StopEvent` will indicate the stopped thread when GDB is running in non-stop mode. Refer to `gdb.ThreadEvent` above for more details.

Emits `gdb.SignalEvent` which extends `gdb.StopEvent`.

This event indicates that the inferior or one of its threads has received as signal. `gdb.SignalEvent` has the following attributes:

`SignalEvent.stop_signal` [Variable]
A string representing the signal received by the inferior. A list of possible signal values can be obtained by running the command `info signals` in the GDB command prompt.

Also emits `gdb.BreakpointEvent` which extends `gdb.StopEvent`.

`gdb.BreakpointEvent` event indicates that one or more breakpoints have been hit, and has the following attributes:

`BreakpointEvent.breakpoints` [Variable]
A sequence containing references to all the breakpoints (type `gdb.Breakpoint`) that were hit. See Section 23.3.2.30 [Breakpoints In Python], page 444, for details of the `gdb.Breakpoint` object.

`BreakpointEvent.breakpoint` [Variable]
A reference to the first breakpoint that was hit. This function is maintained for backward compatibility and is now deprecated in favor of the `gdb.BreakpointEvent.breakpoints` attribute.

`events.new_objfile`
Emits `gdb.NewObjFileEvent` which indicates that a new object file has been loaded by GDB. `gdb.NewObjFileEvent` has one attribute:

`NewObjFileEvent.new_objfile` [Variable]
A reference to the object file (`gdb.Objfile`) which has been loaded. See Section 23.3.2.24 [Objfiles In Python], page 430, for details of the `gdb.Objfile` object.

`events.clear_objfiles`
Emits `gdb.ClearObjFilesEvent` which indicates that the list of object files for a program space has been reset. `gdb.ClearObjFilesEvent` has one attribute:

`ClearObjFilesEvent.progspace` [Variable]
A reference to the program space (`gdb.Progspace`) whose objfile list has been cleared. See Section 23.3.2.23 [Progspaces In Python], page 428.

`events.inferior_call`
Emits events just before and after a function in the inferior is called by GDB. Before an inferior call, this emits an event of type `gdb.InferiorCallPreEvent`, and after an inferior call, this emits an event of type `gdb.InferiorCallPostEvent`.

`gdb.InferiorCallPreEvent`
Indicates that a function in the inferior is about to be called.

`InferiorCallPreEvent.ptid` [Variable]
The thread in which the call will be run.

`InferiorCallPreEvent.address` [Variable]
The location of the function to be called.

`gdb.InferiorCallPostEvent`
Indicates that a function in the inferior has just been called.

`InferiorCallPostEvent.ptid` [Variable]
The thread in which the call was run.

`InferiorCallPostEvent.address` [Variable]
The location of the function that was called.

`events.memory_changed`
Emits `gdb.MemoryChangedEvent` which indicates that the memory of the inferior has been modified by the GDB user, for instance via a command like `set *addr = value`. The event has the following attributes:

`MemoryChangedEvent.address` [Variable]
The start address of the changed region.

`MemoryChangedEvent.length` [Variable]
Length in bytes of the changed region.

`events.register_changed`
Emits `gdb.RegisterChangedEvent` which indicates that a register in the inferior has been modified by the GDB user.

`RegisterChangedEvent.frame` [Variable]
A gdb.Frame object representing the frame in which the register was modified.

`RegisterChangedEvent.regnum` [Variable]
Denotes which register was modified.

`events.breakpoint_created`
This is emitted when a new breakpoint has been created. The argument that is passed is the new `gdb.Breakpoint` object.

`events.breakpoint_modified`
This is emitted when a breakpoint has been modified in some way. The argument that is passed is the new `gdb.Breakpoint` object.

`events.breakpoint_deleted`
: This is emitted when a breakpoint has been deleted. The argument that is passed is the `gdb.Breakpoint` object. When this event is emitted, the `gdb.Breakpoint` object will already be in its invalid state; that is, the `is_valid` method will return `False`.

`events.before_prompt`
: This event carries no payload. It is emitted each time GDB presents a prompt to the user.

`events.new_inferior`
: This is emitted when a new inferior is created. Note that the inferior is not necessarily running; in fact, it may not even have an associated executable.

  The event is of type `gdb.NewInferiorEvent`. This has a single attribute:

  `NewInferiorEvent.inferior` [Variable]
  : The new inferior, a `gdb.Inferior` object.

`events.inferior_deleted`
: This is emitted when an inferior has been deleted. Note that this is not the same as process exit; it is notified when the inferior itself is removed, say via `remove-inferiors`.

  The event is of type `gdb.InferiorDeletedEvent`. This has a single attribute:

  `NewInferiorEvent.inferior` [Variable]
  : The inferior that is being removed, a `gdb.Inferior` object.

`events.new_thread`
: This is emitted when GDB notices a new thread. The event is of type `gdb.NewThreadEvent`, which extends `gdb.ThreadEvent`. This has a single attribute:

  `NewThreadEvent.inferior_thread` [Variable]
  : The new thread.

23.3.2.18 Threads In Python

Python scripts can access information about, and manipulate inferior threads controlled by GDB, via objects of the `gdb.InferiorThread` class.

The following thread-related functions are available in the `gdb` module:

`gdb.selected_thread ()` [Function]
: This function returns the thread object for the selected thread. If there is no selected thread, this will return `None`.

To get the list of threads for an inferior, use the `Inferior.threads()` method. See Section 23.3.2.16 [Inferiors In Python], page 411.

A `gdb.InferiorThread` object has the following attributes:

`InferiorThread.name` [Variable]

The name of the thread. If the user specified a name using `thread name`, then this returns that name. Otherwise, if an OS-supplied name is available, then it is returned. Otherwise, this returns `None`.

This attribute can be assigned to. The new value must be a string object, which sets the new name, or `None`, which removes any user-specified thread name.

`InferiorThread.num` [Variable]

The per-inferior number of the thread, as assigned by GDB.

`InferiorThread.global_num` [Variable]

The global ID of the thread, as assigned by GDB. You can use this to make Python breakpoints thread-specific, for example (see [The Breakpoint.thread attribute], page 446).

`InferiorThread.ptid` [Variable]

ID of the thread, as assigned by the operating system. This attribute is a tuple containing three integers. The first is the Process ID (PID); the second is the Lightweight Process ID (LWPID), and the third is the Thread ID (TID). Either the LWPID or TID may be 0, which indicates that the operating system does not use that identifier.

`InferiorThread.inferior` [Variable]

The inferior this thread belongs to. This attribute is represented as a `gdb.Inferior` object. This attribute is not writable.

A `gdb.InferiorThread` object has the following methods:

`InferiorThread.is_valid ()` [Function]

Returns `True` if the `gdb.InferiorThread` object is valid, `False` if not. A `gdb.InferiorThread` object will become invalid if the thread exits, or the inferior that the thread belongs is deleted. All other `gdb.InferiorThread` methods will throw an exception if it is invalid at the time the method is called.

`InferiorThread.switch ()` [Function]

This changes GDB's currently selected thread to the one represented by this object.

`InferiorThread.is_stopped ()` [Function]

Return a Boolean indicating whether the thread is stopped.

`InferiorThread.is_running ()` [Function]

Return a Boolean indicating whether the thread is running.

`InferiorThread.is_exited ()` [Function]

Return a Boolean indicating whether the thread is exited.

`InferiorThread.handle ()` [Function]

Return the thread object's handle, represented as a Python `bytes` object. A `gdb.Value` representation of the handle may be constructed via `gdb.Value(bufobj, type)` where *bufobj* is the Python `bytes` representation of the handle and *type* is a `gdb.Type` for the handle type.

23.3.2.19 Recordings In Python

The following recordings-related functions (see Chapter 7 [Process Record and Replay], page 99) are available in the `gdb` module:

`gdb.start_recording` ([*method*], [*format*]) [Function]

Start a recording using the given *method* and *format*. If no *format* is given, the default format for the recording method is used. If no *method* is given, the default method will be used. Returns a `gdb.Record` object on success. Throw an exception on failure.

The following strings can be passed as *method*:

- `"full"`
- `"btrace"`: Possible values for *format*: `"pt"`, `"bts"` or leave out for default format.

`gdb.current_recording` () [Function]

Access a currently running recording. Return a `gdb.Record` object on success. Return `None` if no recording is currently active.

`gdb.stop_recording` () [Function]

Stop the current recording. Throw an exception if no recording is currently active. All record objects become invalid after this call.

A `gdb.Record` object has the following attributes:

`Record.method` [Variable]

A string with the current recording method, e.g. `full` or `btrace`.

`Record.format` [Variable]

A string with the current recording format, e.g. `bt`, `pts` or `None`.

`Record.begin` [Variable]

A method specific instruction object representing the first instruction in this recording.

`Record.end` [Variable]

A method specific instruction object representing the current instruction, that is not actually part of the recording.

`Record.replay_position` [Variable]

The instruction representing the current replay position. If there is no replay active, this will be `None`.

`Record.instruction_history` [Variable]

A list with all recorded instructions.

`Record.function_call_history` [Variable]

A list with all recorded function call segments.

A `gdb.Record` object has the following methods:

`Record.goto` (*instruction*) [Function]

Move the replay position to the given *instruction*.

The common `gdb.Instruction` class that recording method specific instruction objects inherit from, has the following attributes:

`Instruction.pc` [Variable]
: An integer representing this instruction's address.

`Instruction.data` [Variable]
: A buffer with the raw instruction data. In Python 3, the return value is a `memoryview` object.

`Instruction.decoded` [Variable]
: A human readable string with the disassembled instruction.

`Instruction.size` [Variable]
: The size of the instruction in bytes.

Additionally `gdb.RecordInstruction` has the following attributes:

`RecordInstruction.number` [Variable]
: An integer identifying this instruction. `number` corresponds to the numbers seen in `record instruction-history` (see Chapter 7 [Process Record and Replay], page 99).

`RecordInstruction.sal` [Variable]
: A `gdb.Symtab_and_line` object representing the associated symtab and line of this instruction. May be `None` if no debug information is available.

`RecordInstruction.is_speculative` [Variable]
: A boolean indicating whether the instruction was executed speculatively.

If an error occured during recording or decoding a recording, this error is represented by a `gdb.RecordGap` object in the instruction list. It has the following attributes:

`RecordGap.number` [Variable]
: An integer identifying this gap. `number` corresponds to the numbers seen in `record instruction-history` (see Chapter 7 [Process Record and Replay], page 99).

`RecordGap.error_code` [Variable]
: A numerical representation of the reason for the gap. The value is specific to the current recording method.

`RecordGap.error_string` [Variable]
: A human readable string with the reason for the gap.

A `gdb.RecordFunctionSegment` object has the following attributes:

`RecordFunctionSegment.number` [Variable]
: An integer identifying this function segment. `number` corresponds to the numbers seen in `record function-call-history` (see Chapter 7 [Process Record and Replay], page 99).

`RecordFunctionSegment.symbol` [Variable]
: A `gdb.Symbol` object representing the associated symbol. May be `None` if no debug information is available.

RecordFunctionSegment.level [Variable]

An integer representing the function call's stack level. May be `None` if the function call is a gap.

RecordFunctionSegment.instructions [Variable]

A list of `gdb.RecordInstruction` or `gdb.RecordGap` objects associated with this function call.

RecordFunctionSegment.up [Variable]

A `gdb.RecordFunctionSegment` object representing the caller's function segment. If the call has not been recorded, this will be the function segment to which control returns. If neither the call nor the return have been recorded, this will be `None`.

RecordFunctionSegment.prev [Variable]

A `gdb.RecordFunctionSegment` object representing the previous segment of this function call. May be `None`.

RecordFunctionSegment.next [Variable]

A `gdb.RecordFunctionSegment` object representing the next segment of this function call. May be `None`.

The following example demonstrates the usage of these objects and functions to create a function that will rewind a record to the last time a function in a different file was executed. This would typically be used to track the execution of user provided callback functions in a library which typically are not visible in a back trace.

```
def bringback ():
    rec = gdb.current_recording ()
    if not rec:
        return

    insn = rec.instruction_history
    if len (insn) == 0:
        return

    try:
        position = insn.index (rec.replay_position)
    except:
        position = -1
    try:
        filename = insn[position].sal.symtab.fullname ()
    except:
        filename = None

    for i in reversed (insn[:position]):
try:
            current = i.sal.symtab.fullname ()
except:
            current = None

        if filename == current:
            continue

        rec.goto (i)
        return
```

Another possible application is to write a function that counts the number of code executions in a given line range. This line range can contain parts of functions or span across several functions and is not limited to be contiguous.

```
def countrange (filename, linerange):
    count = 0

    def filter_only (file_name):
        for call in gdb.current_recording ().function_call_history:
            try:
                if file_name in call.symbol.symtab.fullname ():
                    yield call
            except:
                pass

    for c in filter_only (filename):
        for i in c.instructions:
            try:
                if i.sal.line in linerange:
                    count += 1
                    break;
            except:
                    pass

    return count
```

23.3.2.20 Commands In Python

You can implement new GDB CLI commands in Python. A CLI command is implemented using an instance of the `gdb.Command` class, most commonly using a subclass.

`Command.__init__` (*name*, *command_class* [, *completer_class* [, *prefix*]]) [Function]

The object initializer for `Command` registers the new command with GDB. This initializer is normally invoked from the subclass' own `__init__` method.

name is the name of the command. If *name* consists of multiple words, then the initial words are looked for as prefix commands. In this case, if one of the prefix commands does not exist, an exception is raised.

There is no support for multi-line commands.

command_class should be one of the '`COMMAND_`' constants defined below. This argument tells GDB how to categorize the new command in the help system.

completer_class is an optional argument. If given, it should be one of the '`COMPLETE_`' constants defined below. This argument tells GDB how to perform completion for this command. If not given, GDB will attempt to complete using the object's `complete` method (see below); if no such method is found, an error will occur when completion is attempted.

prefix is an optional argument. If `True`, then the new command is a prefix command; sub-commands of this command may be registered.

The help text for the new command is taken from the Python documentation string for the command's class, if there is one. If no documentation string is provided, the default value "This command is not documented." is used.

Command.dont_repeat () [Function]

By default, a GDB command is repeated when the user enters a blank line at the command prompt. A command can suppress this behavior by invoking the `dont_repeat` method. This is similar to the user command `dont-repeat`, see Section 23.1.1 [Define], page 361.

Command.invoke (*argument, from_tty*) [Function]

This method is called by GDB when this command is invoked.

argument is a string. It is the argument to the command, after leading and trailing whitespace has been stripped.

from_tty is a boolean argument. When true, this means that the command was entered by the user at the terminal; when false it means that the command came from elsewhere.

If this method throws an exception, it is turned into a GDB `error` call. Otherwise, the return value is ignored.

To break *argument* up into an argv-like string use `gdb.string_to_argv`. This function behaves identically to GDB's internal argument lexer `buildargv`. It is recommended to use this for consistency. Arguments are separated by spaces and may be quoted. Example:

```
print gdb.string_to_argv ("1 2\ \\\"3 '4 \"5' \"6 '7\"")
['1', '2 "3', '4 "5', "6 '7"]
```

Command.complete (*text, word*) [Function]

This method is called by GDB when the user attempts completion on this command. All forms of completion are handled by this method, that is, the `TAB` and `M-?` key bindings (see Section 3.3 [Completion], page 24), and the `complete` command (see Section 3.5 [Help], page 28).

The arguments *text* and *word* are both strings; *text* holds the complete command line up to the cursor's location, while *word* holds the last word of the command line; this is computed using a word-breaking heuristic.

The `complete` method can return several values:

- If the return value is a sequence, the contents of the sequence are used as the completions. It is up to `complete` to ensure that the contents actually do complete the word. A zero-length sequence is allowed, it means that there were no completions available. Only string elements of the sequence are used; other elements in the sequence are ignored.
- If the return value is one of the '`COMPLETE_`' constants defined below, then the corresponding GDB-internal completion function is invoked, and its result is used.
- All other results are treated as though there were no available completions.

When a new command is registered, it must be declared as a member of some general class of commands. This is used to classify top-level commands in the on-line help system; note that prefix commands are not listed under their own category but rather that of their top-level command. The available classifications are represented by constants defined in the `gdb` module:

`gdb.COMMAND_NONE`
: The command does not belong to any particular class. A command in this category will not be displayed in any of the help categories.

`gdb.COMMAND_RUNNING`
: The command is related to running the inferior. For example, `start`, `step`, and `continue` are in this category. Type *`help running`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_DATA`
: The command is related to data or variables. For example, `call`, `find`, and `print` are in this category. Type *`help data`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_STACK`
: The command has to do with manipulation of the stack. For example, `backtrace`, `frame`, and `return` are in this category. Type *`help stack`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_FILES`
: This class is used for file-related commands. For example, `file`, `list` and `section` are in this category. Type *`help files`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_SUPPORT`
: This should be used for "support facilities", generally meaning things that are useful to the user when interacting with GDB, but not related to the state of the inferior. For example, `help`, `make`, and `shell` are in this category. Type *`help support`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_STATUS`
: The command is an '`info`'-related command, that is, related to the state of GDB itself. For example, `info`, `macro`, and `show` are in this category. Type *`help status`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_BREAKPOINTS`
: The command has to do with breakpoints. For example, `break`, `clear`, and `delete` are in this category. Type *`help breakpoints`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_TRACEPOINTS`
: The command has to do with tracepoints. For example, `trace`, `actions`, and `tfind` are in this category. Type *`help tracepoints`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_TUI`
: The command has to do with the text user interface (see Chapter 25 [TUI], page 515). Type *`help tui`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_USER`
: The command is a general purpose command for the user, and typically does not fit in one of the other categories. Type *`help user-defined`* at the GDB

prompt to see a list of commands in this category, as well as the list of gdb macros (see Section 23.1 [Sequences], page 361).

`gdb.COMMAND_OBSCURE`
: The command is only used in unusual circumstances, or is not of general interest to users. For example, `checkpoint`, `fork`, and `stop` are in this category. Type *`help obscure`* at the GDB prompt to see a list of commands in this category.

`gdb.COMMAND_MAINTENANCE`
: The command is only useful to GDB maintainers. The `maintenance` and `flushregs` commands are in this category. Type *`help internals`* at the GDB prompt to see a list of commands in this category.

A new command can use a predefined completion function, either by specifying it via an argument at initialization, or by returning it from the `complete` method. These predefined completion constants are all defined in the `gdb` module:

`gdb.COMPLETE_NONE`
: This constant means that no completion should be done.

`gdb.COMPLETE_FILENAME`
: This constant means that filename completion should be performed.

`gdb.COMPLETE_LOCATION`
: This constant means that location completion should be done. See Section 9.2 [Specify Location], page 120.

`gdb.COMPLETE_COMMAND`
: This constant means that completion should examine GDB command names.

`gdb.COMPLETE_SYMBOL`
: This constant means that completion should be done using symbol names as the source.

`gdb.COMPLETE_EXPRESSION`
: This constant means that completion should be done on expressions. Often this means completing on symbol names, but some language parsers also have support for completing on field names.

The following code snippet shows how a trivial CLI command can be implemented in Python:

```
class HelloWorld (gdb.Command):
  """Greet the whole world."""

  def __init__ (self):
    super (HelloWorld, self).__init__ ("hello-world", gdb.COMMAND_USER)

  def invoke (self, arg, from_tty):
    print ("Hello, World!")

HelloWorld ()
```

The last line instantiates the class, and is necessary to trigger the registration of the command with GDB. Depending on how the Python code is read into GDB, you may need to import the `gdb` module explicitly.

23.3.2.21 Parameters In Python

You can implement new GDB parameters using Python. A new parameter is implemented as an instance of the `gdb.Parameter` class.

Parameters are exposed to the user via the `set` and `show` commands. See Section 3.5 [Help], page 28.

There are many parameters that already exist and can be set in GDB. Two examples are: `set follow fork` and `set charset`. Setting these parameters influences certain behavior in GDB. Similarly, you can define parameters that can be used to influence behavior in custom Python scripts and commands.

`Parameter.__init__` (*name*, *command-class*, *parameter-class* [, *enum-sequence*]) [Function]

The object initializer for `Parameter` registers the new parameter with GDB. This initializer is normally invoked from the subclass' own `__init__` method.

name is the name of the new parameter. If *name* consists of multiple words, then the initial words are looked for as prefix parameters. An example of this can be illustrated with the `set print` set of parameters. If *name* is `print foo`, then `print` will be searched as the prefix parameter. In this case the parameter can subsequently be accessed in GDB as `set print foo`.

If *name* consists of multiple words, and no prefix parameter group can be found, an exception is raised.

command-class should be one of the '`COMMAND_`' constants (see Section 23.3.2.20 [Commands In Python], page 421). This argument tells GDB how to categorize the new parameter in the help system.

parameter-class should be one of the '`PARAM_`' constants defined below. This argument tells GDB the type of the new parameter; this information is used for input validation and completion.

If *parameter-class* is `PARAM_ENUM`, then *enum-sequence* must be a sequence of strings. These strings represent the possible values for the parameter.

If *parameter-class* is not `PARAM_ENUM`, then the presence of a fourth argument will cause an exception to be thrown.

The help text for the new parameter is taken from the Python documentation string for the parameter's class, if there is one. If there is no documentation string, a default value is used.

`Parameter.set_doc` [Variable]

If this attribute exists, and is a string, then its value is used as the help text for this parameter's `set` command. The value is examined when `Parameter.__init__` is invoked; subsequent changes have no effect.

`Parameter.show_doc` [Variable]

If this attribute exists, and is a string, then its value is used as the help text for this parameter's `show` command. The value is examined when `Parameter.__init__` is invoked; subsequent changes have no effect.

`Parameter.value` [Variable]

The `value` attribute holds the underlying value of the parameter. It can be read and assigned to just as any other attribute. GDB does validation when assignments are made.

There are two methods that may be implemented in any `Parameter` class. These are:

`Parameter.get_set_string` (*self*) [Function]

If this method exists, GDB will call it when a *parameter*'s value has been changed via the `set` API (for example, *set foo off*). The `value` attribute has already been populated with the new value and may be used in output. This method must return a string. If the returned string is not empty, GDB will present it to the user.

If this method raises the `gdb.GdbError` exception (see Section 23.3.2.2 [Exception Handling], page 377), then GDB will print the exception's string and the `set` command will fail. Note, however, that the `value` attribute will not be reset in this case. So, if your parameter must validate values, it should store the old value internally and reset the exposed value, like so:

```
class ExampleParam (gdb.Parameter):
   def __init__ (self, name):
      super (ExampleParam, self).__init__ (name,
                   gdb.COMMAND_DATA,
                   gdb.PARAM_BOOLEAN)
      self.value = True
      self.saved_value = True
   def validate(self):
      return False
   def get_set_string (self):
      if not self.validate():
        self.value = self.saved_value
        raise gdb.GdbError('Failed to validate')
      self.saved_value = self.value
      return ""
```

`Parameter.get_show_string` (*self*, *svalue*) [Function]

GDB will call this method when a *parameter*'s `show` API has been invoked (for example, *show foo*). The argument `svalue` receives the string representation of the current value. This method must return a string.

When a new parameter is defined, its type must be specified. The available types are represented by constants defined in the `gdb` module:

`gdb.PARAM_BOOLEAN`
: The value is a plain boolean. The Python boolean values, `True` and `False` are the only valid values.

`gdb.PARAM_AUTO_BOOLEAN`
: The value has three possible states: true, false, and '`auto`'. In Python, true and false are represented using boolean constants, and '`auto`' is represented using `None`.

`gdb.PARAM_UINTEGER`
: The value is an unsigned integer. The value of 0 should be interpreted to mean "unlimited".

`gdb.PARAM_INTEGER`
: The value is a signed integer. The value of 0 should be interpreted to mean "unlimited".

`gdb.PARAM_STRING`
: The value is a string. When the user modifies the string, any escape sequences, such as '`\t`', '`\f`', and octal escapes, are translated into corresponding characters and encoded into the current host charset.

`gdb.PARAM_STRING_NOESCAPE`
: The value is a string. When the user modifies the string, escapes are passed through untranslated.

`gdb.PARAM_OPTIONAL_FILENAME`
: The value is a either a filename (a string), or `None`.

`gdb.PARAM_FILENAME`
: The value is a filename. This is just like `PARAM_STRING_NOESCAPE`, but uses file names for completion.

`gdb.PARAM_ZINTEGER`
: The value is an integer. This is like `PARAM_INTEGER`, except 0 is interpreted as itself.

`gdb.PARAM_ZUINTEGER`
: The value is an unsigned integer. This is like `PARAM_INTEGER`, except 0 is interpreted as itself, and the value cannot be negative.

`gdb.PARAM_ZUINTEGER_UNLIMITED`
: The value is a signed integer. This is like `PARAM_ZUINTEGER`, except the special value -1 should be interpreted to mean "unlimited". Other negative values are not allowed.

`gdb.PARAM_ENUM`
: The value is a string, which must be one of a collection string constants provided when the parameter is created.

23.3.2.22 Writing new convenience functions

You can implement new convenience functions (see Section 10.12 [Convenience Vars], page 162) in Python. A convenience function is an instance of a subclass of the class `gdb.Function`.

`Function.__init__` (*name*) [Function]

The initializer for `Function` registers the new function with GDB. The argument *name* is the name of the function, a string. The function will be visible to the user as a convenience variable of type `internal function`, whose name is the same as the given *name*.

The documentation for the new function is taken from the documentation string for the new class.

`Function.invoke` (**args*) [Function]

When a convenience function is evaluated, its arguments are converted to instances of `gdb.Value`, and then the function's `invoke` method is called. Note that GDB does not predetermine the arity of convenience functions. Instead, all available arguments are passed to `invoke`, following the standard Python calling convention. In particular, a convenience function can have default values for parameters without ill effect.

The return value of this method is used as its value in the enclosing expression. If an ordinary Python value is returned, it is converted to a `gdb.Value` following the usual rules.

The following code snippet shows how a trivial convenience function can be implemented in Python:

```
class Greet (gdb.Function):
  """Return string to greet someone.
Takes a name as argument."""

  def __init__ (self):
    super (Greet, self).__init__ ("greet")

  def invoke (self, name):
    return "Hello, %s!" % name.string ()

Greet ()
```

The last line instantiates the class, and is necessary to trigger the registration of the function with GDB. Depending on how the Python code is read into GDB, you may need to import the `gdb` module explicitly.

Now you can use the function in an expression:

```
(gdb) print $greet("Bob")
$1 = "Hello, Bob!"
```

23.3.2.23 Program Spaces In Python

A program space, or *progspace*, represents a symbolic view of an address space. It consists of all of the objfiles of the program. See Section 23.3.2.24 [Objfiles In Python], page 430. See Section 4.9 [Inferiors Connections and Programs], page 40, for more details about program spaces.

The following progspace-related functions are available in the `gdb` module:

`gdb.current_progspace ()` [Function]

This function returns the program space of the currently selected inferior. See Section 4.9 [Inferiors Connections and Programs], page 40. This is identical to `gdb.selected_inferior().progspace` (see Section 23.3.2.16 [Inferiors In Python], page 411) and is included for historical compatibility.

`gdb.progspaces ()` [Function]

Return a sequence of all the progspaces currently known to GDB.

Each progspace is represented by an instance of the `gdb.Progspace` class.

`Progspace.filename` [Variable]

The file name of the progspace as a string.

`Progspace.pretty_printers` [Variable]

The `pretty_printers` attribute is a list of functions. It is used to look up pretty-printers. A `Value` is passed to each function in order; if the function returns `None`, then the search continues. Otherwise, the return value should be an object which is used to format the value. See Section 23.3.2.5 [Pretty Printing API], page 389, for more information.

`Progspace.type_printers` [Variable]
The `type_printers` attribute is a list of type printer objects. See Section 23.3.2.8 [Type Printing API], page 393, for more information.

`Progspace.frame_filters` [Variable]
The `frame_filters` attribute is a dictionary of frame filter objects. See Section 23.3.2.9 [Frame Filter API], page 394, for more information.

A program space has the following methods:

`Progspace.block_for_pc` (*pc*) [Function]
Return the innermost `gdb.Block` containing the given *pc* value. If the block cannot be found for the *pc* value specified, the function will return `None`.

`Progspace.find_pc_line` (*pc*) [Function]
Return the `gdb.Symtab_and_line` object corresponding to the *pc* value. See Section 23.3.2.28 [Symbol Tables In Python], page 441. If an invalid value of *pc* is passed as an argument, then the `symtab` and `line` attributes of the returned `gdb.Symtab_and_line` object will be `None` and 0 respectively.

`Progspace.is_valid` () [Function]
Returns `True` if the `gdb.Progspace` object is valid, `False` if not. A `gdb.Progspace` object can become invalid if the program space file it refers to is not referenced by any inferior. All other `gdb.Progspace` methods will throw an exception if it is invalid at the time the method is called.

`Progspace.objfiles` () [Function]
Return a sequence of all the objfiles referenced by this program space. See Section 23.3.2.24 [Objfiles In Python], page 430.

`Progspace.solib_name` (*address*) [Function]
Return the name of the shared library holding the given *address* as a string, or `None`.

One may add arbitrary attributes to `gdb.Progspace` objects in the usual Python way. This is useful if, for example, one needs to do some extra record keeping associated with the program space.

In this contrived example, we want to perform some processing when an objfile with a certain symbol is loaded, but we only want to do this once because it is expensive. To achieve this we record the results with the program space because we can't predict when the desired objfile will be loaded.

```
(gdb) python
def clear_objfiles_handler(event):
    event.progspace.expensive_computation = None
def expensive(symbol):
    """A mock routine to perform an "expensive" computation on symbol."""
    print ("Computing the answer to the ultimate question ...")
    return 42
```

```
def new_objfile_handler(event):
    objfile = event.new_objfile
    progspace = objfile.progspace
    if not hasattr(progspace, 'expensive_computation') or \
            progspace.expensive_computation is None:
        # We use 'main' for the symbol to keep the example simple.
        # Note: There's no current way to constrain the lookup
        # to one objfile.
        symbol = gdb.lookup_global_symbol('main')
        if symbol is not None:
            progspace.expensive_computation = expensive(symbol)
gdb.events.clear_objfiles.connect(clear_objfiles_handler)
gdb.events.new_objfile.connect(new_objfile_handler)
end
(gdb) file /tmp/hello
Reading symbols from /tmp/hello...
Computing the answer to the ultimate question ...
(gdb) python print gdb.current_progspace().expensive_computation
42
(gdb) run
Starting program: /tmp/hello
Hello.
[Inferior 1 (process 4242) exited normally]
```

23.3.2.24 Objfiles In Python

GDB loads symbols for an inferior from various symbol-containing files (see Section 18.1 [Files], page 273). These include the primary executable file, any shared libraries used by the inferior, and any separate debug info files (see Section 18.3 [Separate Debug Files], page 282). GDB calls these symbol-containing files *objfiles*.

The following objfile-related functions are available in the `gdb` module:

`gdb.current_objfile` () [Function]

When auto-loading a Python script (see Section 23.3.3 [Python Auto-loading], page 452), GDB sets the "current objfile" to the corresponding objfile. This function returns the current objfile. If there is no current objfile, this function returns `None`.

`gdb.objfiles` () [Function]

Return a sequence of objfiles referenced by the current program space. See Section 23.3.2.24 [Objfiles In Python], page 430, and Section 23.3.2.23 [Progspaces In Python], page 428. This is identical to `gdb.selected_inferior().progspace.objfiles()` and is included for historical compatibility.

`gdb.lookup_objfile` (*name* [, *by_build_id*]) [Function]

Look up *name*, a file name or build ID, in the list of objfiles for the current program space (see Section 23.3.2.23 [Progspaces In Python], page 428). If the objfile is not found throw the Python `ValueError` exception.

If *name* is a relative file name, then it will match any source file name with the same trailing components. For example, if *name* is '`gcc/expr.c`', then it will match source file name of `/build/trunk/gcc/expr.c`, but not `/build/trunk/libcpp/expr.c` or `/build/trunk/gcc/x-expr.c`.

If *by_build_id* is provided and is `True` then *name* is the build ID of the objfile. Otherwise, *name* is a file name. This is supported only on some operating systems, notably those which use the ELF format for binary files and the GNU Binutils. For more details about this feature, see the description of the `--build-id` command-line option in Section "Command Line Options" in *The GNU Linker*.

Each objfile is represented by an instance of the `gdb.Objfile` class.

`Objfile.filename` [Variable]
The file name of the objfile as a string, with symbolic links resolved.

The value is `None` if the objfile is no longer valid. See the `gdb.Objfile.is_valid` method, described below.

`Objfile.username` [Variable]
The file name of the objfile as specified by the user as a string.

The value is `None` if the objfile is no longer valid. See the `gdb.Objfile.is_valid` method, described below.

`Objfile.owner` [Variable]
For separate debug info objfiles this is the corresponding `gdb.Objfile` object that debug info is being provided for. Otherwise this is `None`. Separate debug info objfiles are added with the `gdb.Objfile.add_separate_debug_file` method, described below.

`Objfile.build_id` [Variable]
The build ID of the objfile as a string. If the objfile does not have a build ID then the value is `None`.

This is supported only on some operating systems, notably those which use the ELF format for binary files and the GNU Binutils. For more details about this feature, see the description of the `--build-id` command-line option in Section "Command Line Options" in *The GNU Linker*.

`Objfile.progspace` [Variable]
The containing program space of the objfile as a `gdb.Progspace` object. See Section 23.3.2.23 [Progspaces In Python], page 428.

`Objfile.pretty_printers` [Variable]
The `pretty_printers` attribute is a list of functions. It is used to look up pretty-printers. A `Value` is passed to each function in order; if the function returns `None`, then the search continues. Otherwise, the return value should be an object which is used to format the value. See Section 23.3.2.5 [Pretty Printing API], page 389, for more information.

`Objfile.type_printers` [Variable]
The `type_printers` attribute is a list of type printer objects. See Section 23.3.2.8 [Type Printing API], page 393, for more information.

`Objfile.frame_filters` [Variable]
The `frame_filters` attribute is a dictionary of frame filter objects. See Section 23.3.2.9 [Frame Filter API], page 394, for more information.

One may add arbitrary attributes to `gdb.Objfile` objects in the usual Python way. This is useful if, for example, one needs to do some extra record keeping associated with the objfile.

In this contrived example we record the time when GDB loaded the objfile.

```
(gdb) python
import datetime
def new_objfile_handler(event):
    # Set the time_loaded attribute of the new objfile.
    event.new_objfile.time_loaded = datetime.datetime.today()
gdb.events.new_objfile.connect(new_objfile_handler)
end
(gdb) file ./hello
Reading symbols from ./hello...
(gdb) python print gdb.objfiles()[0].time_loaded
2014-10-09 11:41:36.770345
```

A `gdb.Objfile` object has the following methods:

`Objfile.is_valid` () [Function]

Returns `True` if the `gdb.Objfile` object is valid, `False` if not. A `gdb.Objfile` object can become invalid if the object file it refers to is not loaded in GDB any longer. All other `gdb.Objfile` methods will throw an exception if it is invalid at the time the method is called.

`Objfile.add_separate_debug_file` (*file*) [Function]

Add *file* to the list of files that GDB will search for debug information for the objfile. This is useful when the debug info has been removed from the program and stored in a separate file. GDB has built-in support for finding separate debug info files (see Section 18.3 [Separate Debug Files], page 282), but if the file doesn't live in one of the standard places that GDB searches then this function can be used to add a debug info file from a different place.

`Objfile.lookup_global_symbol` (*name* [, *domain*]) [Function]

Search for a global symbol named *name* in this objfile. Optionally, the search scope can be restricted with the *domain* argument. The *domain* argument must be a domain constant defined in the `gdb` module and described in Section 23.3.2.27 [Symbols In Python], page 438. This function is similar to `gdb.lookup_global_symbol`, except that the search is limited to this objfile.

The result is a `gdb.Symbol` object or `None` if the symbol is not found.

`Objfile.lookup_static_symbol` (*name* [, *domain*]) [Function]

Like `Objfile.lookup_global_symbol`, but searches for a global symbol with static linkage named *name* in this objfile.

23.3.2.25 Accessing inferior stack frames from Python

When the debugged program stops, GDB is able to analyze its call stack (see Section 8.1 [Stack frames], page 107). The `gdb.Frame` class represents a frame in the stack. A `gdb.Frame` object is only valid while its corresponding frame exists in the inferior's stack. If you try to use an invalid frame object, GDB will throw a `gdb.error` exception (see Section 23.3.2.2 [Exception Handling], page 377).

Two `gdb.Frame` objects can be compared for equality with the `==` operator, like:

```
(gdb) python print gdb.newest_frame() == gdb.selected_frame ()
True
```

The following frame-related functions are available in the `gdb` module:

`gdb.selected_frame ()` [Function]

Return the selected frame object. (see Section 8.3 [Selecting a Frame], page 111).

`gdb.newest_frame ()` [Function]

Return the newest frame object for the selected thread.

`gdb.frame_stop_reason_string (`*reason*`)` [Function]

Return a string explaining the reason why GDB stopped unwinding frames, as expressed by the given *reason* code (an integer, see the `unwind_stop_reason` method further down in this section).

`gdb.invalidate_cached_frames` [Function]

GDB internally keeps a cache of the frames that have been unwound. This function invalidates this cache.

This function should not generally be called by ordinary Python code. It is documented for the sake of completeness.

A `gdb.Frame` object has the following methods:

`Frame.is_valid ()` [Function]

Returns true if the `gdb.Frame` object is valid, false if not. A frame object can become invalid if the frame it refers to doesn't exist anymore in the inferior. All `gdb.Frame` methods will throw an exception if it is invalid at the time the method is called.

`Frame.name ()` [Function]

Returns the function name of the frame, or `None` if it can't be obtained.

`Frame.architecture ()` [Function]

Returns the `gdb.Architecture` object corresponding to the frame's architecture. See Section 23.3.2.33 [Architectures In Python], page 449.

`Frame.type ()` [Function]

Returns the type of the frame. The value can be one of:

`gdb.NORMAL_FRAME`
: An ordinary stack frame.

`gdb.DUMMY_FRAME`
: A fake stack frame that was created by GDB when performing an inferior function call.

`gdb.INLINE_FRAME`
: A frame representing an inlined function. The function was inlined into a `gdb.NORMAL_FRAME` that is older than this one.

`gdb.TAILCALL_FRAME`
: A frame representing a tail call. See Section 11.2 [Tail Call Frames], page 184.

`gdb.SIGTRAMP_FRAME`
: A signal trampoline frame. This is the frame created by the OS when it calls into a signal handler.

`gdb.ARCH_FRAME`
: A fake stack frame representing a cross-architecture call.

`gdb.SENTINEL_FRAME`
: This is like `gdb.NORMAL_FRAME`, but it is only used for the newest frame.

`Frame.unwind_stop_reason ()` [Function]

Return an integer representing the reason why it's not possible to find more frames toward the outermost frame. Use `gdb.frame_stop_reason_string` to convert the value returned by this function to a string. The value can be one of:

`gdb.FRAME_UNWIND_NO_REASON`
: No particular reason (older frames should be available).

`gdb.FRAME_UNWIND_NULL_ID`
: The previous frame's analyzer returns an invalid result. This is no longer used by GDB, and is kept only for backward compatibility.

`gdb.FRAME_UNWIND_OUTERMOST`
: This frame is the outermost.

`gdb.FRAME_UNWIND_UNAVAILABLE`
: Cannot unwind further, because that would require knowing the values of registers or memory that have not been collected.

`gdb.FRAME_UNWIND_INNER_ID`
: This frame ID looks like it ought to belong to a NEXT frame, but we got it for a PREV frame. Normally, this is a sign of unwinder failure. It could also indicate stack corruption.

`gdb.FRAME_UNWIND_SAME_ID`
: This frame has the same ID as the previous one. That means that unwinding further would almost certainly give us another frame with exactly the same ID, so break the chain. Normally, this is a sign of unwinder failure. It could also indicate stack corruption.

`gdb.FRAME_UNWIND_NO_SAVED_PC`
: The frame unwinder did not find any saved PC, but we needed one to unwind further.

`gdb.FRAME_UNWIND_MEMORY_ERROR`
: The frame unwinder caused an error while trying to access memory.

`gdb.FRAME_UNWIND_FIRST_ERROR`
: Any stop reason greater or equal to this value indicates some kind of error. This special value facilitates writing code that tests for errors in unwinding in a way that will work correctly even if the list of the other values is modified in future GDB versions. Using it, you could write:

```
reason = gdb.selected_frame().unwind_stop_reason ()
```

```
reason_str =  gdb.frame_stop_reason_string (reason)
if reason >=  gdb.FRAME_UNWIND_FIRST_ERROR:
    print ("An error occured: %s" % reason_str)
```

`Frame.pc` () [Function]
Returns the frame's resume address.

`Frame.block` () [Function]
Return the frame's code block. See Section 23.3.2.26 [Blocks In Python], page 436. If the frame does not have a block – for example, if there is no debugging information for the code in question – then this will throw an exception.

`Frame.function` () [Function]
Return the symbol for the function corresponding to this frame. See Section 23.3.2.27 [Symbols In Python], page 438.

`Frame.older` () [Function]
Return the frame that called this frame.

`Frame.newer` () [Function]
Return the frame called by this frame.

`Frame.find_sal` () [Function]
Return the frame's symtab and line object. See Section 23.3.2.28 [Symbol Tables In Python], page 441.

`Frame.read_register` (*register*) [Function]
Return the value of *register* in this frame. Returns a `Gdb.Value` object. Throws an exception if *register* does not exist. The *register* argument must be one of the following:

1. A string that is the name of a valid register (e.g., `'sp'` or `'rax'`).
2. A `gdb.RegisterDescriptor` object (see Section 23.3.2.34 [Registers In Python], page 450).
3. A GDB internal, platform specific number. Using these numbers is supported for historic reasons, but is not recommended as future changes to GDB could change the mapping between numbers and the registers they represent, breaking any Python code that uses the platform-specific numbers. The numbers are usually found in the corresponding `*platform*-tdep.h` file in the GDB source tree.

Using a string to access registers will be slightly slower than the other two methods as GDB must look up the mapping between name and internal register number. If performance is critical consider looking up and caching a `gdb.RegisterDescriptor` object.

`Frame.read_var` (*variable* [, *block*]) [Function]
Return the value of *variable* in this frame. If the optional argument *block* is provided, search for the variable from that block; otherwise start at the frame's current block (which is determined by the frame's current program counter). The *variable* argument must be a string or a `gdb.Symbol` object; *block* must be a `gdb.Block` object.

`Frame.select ()` [Function]

Set this frame to be the selected frame. See Chapter 8 [Examining the Stack], page 107.

23.3.2.26 Accessing blocks from Python

In GDB, symbols are stored in blocks. A block corresponds roughly to a scope in the source code. Blocks are organized hierarchically, and are represented individually in Python as a `gdb.Block`. Blocks rely on debugging information being available.

A frame has a block. Please see Section 23.3.2.25 [Frames In Python], page 432, for a more in-depth discussion of frames.

The outermost block is known as the *global block*. The global block typically holds public global variables and functions.

The block nested just inside the global block is the *static block*. The static block typically holds file-scoped variables and functions.

GDB provides a method to get a block's superblock, but there is currently no way to examine the sub-blocks of a block, or to iterate over all the blocks in a symbol table (see Section 23.3.2.28 [Symbol Tables In Python], page 441).

Here is a short example that should help explain blocks:

```
/* This is in the global block.  */
int global;

/* This is in the static block.  */
static int file_scope;

/* 'function' is in the global block, and 'argument' is
   in a block nested inside of 'function'.  */
int function (int argument)
{
  /* 'local' is in a block inside 'function'.  It may or may
     not be in the same block as 'argument'.  */
  int local;

  {
     /* 'inner' is in a block whose superblock is the one holding
        'local'.  */
     int inner;

     /* If this call is expanded by the compiler, you may see
        a nested block here whose function is 'inline_function'
        and whose superblock is the one holding 'inner'.  */
     inline_function ();
  }
}
```

A `gdb.Block` is iterable. The iterator returns the symbols (see Section 23.3.2.27 [Symbols In Python], page 438) local to the block. Python programs should not assume that a specific block object will always contain a given symbol, since changes in GDB features and infrastructure may cause symbols move across blocks in a symbol table. You can also use Python's *dictionary syntax* to access variables in this block, e.g.:

```
symbol = some_block['variable']  # symbol is of type gdb.Symbol
```

The following block-related functions are available in the `gdb` module:

`gdb.block_for_pc` (*pc*) [Function]

Return the innermost `gdb.Block` containing the given *pc* value. If the block cannot be found for the *pc* value specified, the function will return `None`. This is identical to `gdb.current_progspace().block_for_pc(pc)` and is included for historical compatibility.

A `gdb.Block` object has the following methods:

`Block.is_valid` () [Function]

Returns `True` if the `gdb.Block` object is valid, `False` if not. A block object can become invalid if the block it refers to doesn't exist anymore in the inferior. All other `gdb.Block` methods will throw an exception if it is invalid at the time the method is called. The block's validity is also checked during iteration over symbols of the block.

A `gdb.Block` object has the following attributes:

`Block.start` [Variable]

The start address of the block. This attribute is not writable.

`Block.end` [Variable]

One past the last address that appears in the block. This attribute is not writable.

`Block.function` [Variable]

The name of the block represented as a `gdb.Symbol`. If the block is not named, then this attribute holds `None`. This attribute is not writable.

For ordinary function blocks, the superblock is the static block. However, you should note that it is possible for a function block to have a superblock that is not the static block – for instance this happens for an inlined function.

`Block.superblock` [Variable]

The block containing this block. If this parent block does not exist, this attribute holds `None`. This attribute is not writable.

`Block.global_block` [Variable]

The global block associated with this block. This attribute is not writable.

`Block.static_block` [Variable]

The static block associated with this block. This attribute is not writable.

`Block.is_global` [Variable]

`True` if the `gdb.Block` object is a global block, `False` if not. This attribute is not writable.

`Block.is_static` [Variable]

`True` if the `gdb.Block` object is a static block, `False` if not. This attribute is not writable.

23.3.2.27 Python representation of Symbols

GDB represents every variable, function and type as an entry in a symbol table. See Chapter 16 [Examining the Symbol Table], page 247. Similarly, Python represents these symbols in GDB with the `gdb.Symbol` object.

The following symbol-related functions are available in the `gdb` module:

`gdb.lookup_symbol` (*name* [, *block* [, *domain*]]) [Function]

This function searches for a symbol by name. The search scope can be restricted to the parameters defined in the optional domain and block arguments.

name is the name of the symbol. It must be a string. The optional *block* argument restricts the search to symbols visible in that *block*. The *block* argument must be a `gdb.Block` object. If omitted, the block for the current frame is used. The optional *domain* argument restricts the search to the domain type. The *domain* argument must be a domain constant defined in the `gdb` module and described later in this chapter.

The result is a tuple of two elements. The first element is a `gdb.Symbol` object or `None` if the symbol is not found. If the symbol is found, the second element is `True` if the symbol is a field of a method's object (e.g., `this` in C++), otherwise it is `False`. If the symbol is not found, the second element is `False`.

`gdb.lookup_global_symbol` (*name* [, *domain*]) [Function]

This function searches for a global symbol by name. The search scope can be restricted to by the domain argument.

name is the name of the symbol. It must be a string. The optional *domain* argument restricts the search to the domain type. The *domain* argument must be a domain constant defined in the `gdb` module and described later in this chapter.

The result is a `gdb.Symbol` object or `None` if the symbol is not found.

`gdb.lookup_static_symbol` (*name* [, *domain*]) [Function]

This function searches for a global symbol with static linkage by name. The search scope can be restricted to by the domain argument.

name is the name of the symbol. It must be a string. The optional *domain* argument restricts the search to the domain type. The *domain* argument must be a domain constant defined in the `gdb` module and described later in this chapter.

The result is a `gdb.Symbol` object or `None` if the symbol is not found.

Note that this function will not find function-scoped static variables. To look up such variables, iterate over the variables of the function's `gdb.Block` and check that `block.addr_class` is `gdb.SYMBOL_LOC_STATIC`.

There can be multiple global symbols with static linkage with the same name. This function will only return the first matching symbol that it finds. Which symbol is found depends on where GDB is currently stopped, as GDB will first search for matching symbols in the current object file, and then search all other object files. If the application is not yet running then GDB will search all object files in the order they appear in the debug information.

gdb.lookup_static_symbols (*name* [, *domain*]) [Function]

Similar to gdb.lookup_static_symbol, this function searches for global symbols with static linkage by name, and optionally restricted by the domain argument. However, this function returns a list of all matching symbols found, not just the first one.

name is the name of the symbol. It must be a string. The optional *domain* argument restricts the search to the domain type. The *domain* argument must be a domain constant defined in the gdb module and described later in this chapter.

The result is a list of gdb.Symbol objects which could be empty if no matching symbols were found.

Note that this function will not find function-scoped static variables. To look up such variables, iterate over the variables of the function's gdb.Block and check that block.addr_class is gdb.SYMBOL_LOC_STATIC.

A gdb.Symbol object has the following attributes:

Symbol.type [Variable]

The type of the symbol or None if no type is recorded. This attribute is represented as a gdb.Type object. See Section 23.3.2.4 [Types In Python], page 384. This attribute is not writable.

Symbol.symtab [Variable]

The symbol table in which the symbol appears. This attribute is represented as a gdb.Symtab object. See Section 23.3.2.28 [Symbol Tables In Python], page 441. This attribute is not writable.

Symbol.line [Variable]

The line number in the source code at which the symbol was defined. This is an integer.

Symbol.name [Variable]

The name of the symbol as a string. This attribute is not writable.

Symbol.linkage_name [Variable]

The name of the symbol, as used by the linker (i.e., may be mangled). This attribute is not writable.

Symbol.print_name [Variable]

The name of the symbol in a form suitable for output. This is either name or linkage_name, depending on whether the user asked GDB to display demangled or mangled names.

Symbol.addr_class [Variable]

The address class of the symbol. This classifies how to find the value of a symbol. Each address class is a constant defined in the gdb module and described later in this chapter.

Symbol.needs_frame [Variable]

This is True if evaluating this symbol's value requires a frame (see Section 23.3.2.25 [Frames In Python], page 432) and False otherwise. Typically, local variables will require a frame, but other symbols will not.

Symbol.is_argument [Variable]
: `True` if the symbol is an argument of a function.

Symbol.is_constant [Variable]
: `True` if the symbol is a constant.

Symbol.is_function [Variable]
: `True` if the symbol is a function or a method.

Symbol.is_variable [Variable]
: `True` if the symbol is a variable.

A `gdb.Symbol` object has the following methods:

Symbol.is_valid () [Function]
: Returns `True` if the `gdb.Symbol` object is valid, `False` if not. A `gdb.Symbol` object can become invalid if the symbol it refers to does not exist in GDB any longer. All other `gdb.Symbol` methods will throw an exception if it is invalid at the time the method is called.

Symbol.value ([*frame*]) [Function]
: Compute the value of the symbol, as a `gdb.Value`. For functions, this computes the address of the function, cast to the appropriate type. If the symbol requires a frame in order to compute its value, then *frame* must be given. If *frame* is not given, or if *frame* is invalid, then this method will throw an exception.

The available domain categories in `gdb.Symbol` are represented as constants in the `gdb` module:

`gdb.SYMBOL_UNDEF_DOMAIN`
: This is used when a domain has not been discovered or none of the following domains apply. This usually indicates an error either in the symbol information or in GDB's handling of symbols.

`gdb.SYMBOL_VAR_DOMAIN`
: This domain contains variables, function names, typedef names and enum type values.

`gdb.SYMBOL_STRUCT_DOMAIN`
: This domain holds struct, union and enum type names.

`gdb.SYMBOL_LABEL_DOMAIN`
: This domain contains names of labels (for gotos).

`gdb.SYMBOL_MODULE_DOMAIN`
: This domain contains names of Fortran module types.

`gdb.SYMBOL_COMMON_BLOCK_DOMAIN`
: This domain contains names of Fortran common blocks.

The available address class categories in `gdb.Symbol` are represented as constants in the `gdb` module:

`gdb.SYMBOL_LOC_UNDEF`
: If this is returned by address class, it indicates an error either in the symbol information or in GDB's handling of symbols.

gdb.SYMBOL_LOC_CONST
: Value is constant int.

gdb.SYMBOL_LOC_STATIC
: Value is at a fixed address.

gdb.SYMBOL_LOC_REGISTER
: Value is in a register.

gdb.SYMBOL_LOC_ARG
: Value is an argument. This value is at the offset stored within the symbol inside the frame's argument list.

gdb.SYMBOL_LOC_REF_ARG
: Value address is stored in the frame's argument list. Just like `LOC_ARG` except that the value's address is stored at the offset, not the value itself.

gdb.SYMBOL_LOC_REGPARM_ADDR
: Value is a specified register. Just like `LOC_REGISTER` except the register holds the address of the argument instead of the argument itself.

gdb.SYMBOL_LOC_LOCAL
: Value is a local variable.

gdb.SYMBOL_LOC_TYPEDEF
: Value not used. Symbols in the domain `SYMBOL_STRUCT_DOMAIN` all have this class.

gdb.SYMBOL_LOC_BLOCK
: Value is a block.

gdb.SYMBOL_LOC_CONST_BYTES
: Value is a byte-sequence.

gdb.SYMBOL_LOC_UNRESOLVED
: Value is at a fixed address, but the address of the variable has to be determined from the minimal symbol table whenever the variable is referenced.

gdb.SYMBOL_LOC_OPTIMIZED_OUT
: The value does not actually exist in the program.

gdb.SYMBOL_LOC_COMPUTED
: The value's address is a computed location.

gdb.SYMBOL_LOC_COMMON_BLOCK
: The value's address is a symbol. This is only used for Fortran common blocks.

23.3.2.28 Symbol table representation in Python

Access to symbol table data maintained by GDB on the inferior is exposed to Python via two objects: `gdb.Symtab_and_line` and `gdb.Symtab`. Symbol table and line data for a frame is returned from the `find_sal` method in `gdb.Frame` object. See Section 23.3.2.25 [Frames In Python], page 432.

For more information on GDB's symbol table management, see Chapter 16 [Examining the Symbol Table], page 247, for more information.

A `gdb.Symtab_and_line` object has the following attributes:

Symtab_and_line.symtab [Variable]
The symbol table object (gdb.Symtab) for this frame. This attribute is not writable.

Symtab_and_line.pc [Variable]
Indicates the start of the address range occupied by code for the current source line. This attribute is not writable.

Symtab_and_line.last [Variable]
Indicates the end of the address range occupied by code for the current source line. This attribute is not writable.

Symtab_and_line.line [Variable]
Indicates the current line number for this object. This attribute is not writable.

A gdb.Symtab_and_line object has the following methods:

Symtab_and_line.is_valid () [Function]
Returns True if the gdb.Symtab_and_line object is valid, False if not. A gdb.Symtab_and_line object can become invalid if the Symbol table and line object it refers to does not exist in GDB any longer. All other gdb.Symtab_and_line methods will throw an exception if it is invalid at the time the method is called.

A gdb.Symtab object has the following attributes:

Symtab.filename [Variable]
The symbol table's source filename. This attribute is not writable.

Symtab.objfile [Variable]
The symbol table's backing object file. See Section 23.3.2.24 [Objfiles In Python], page 430. This attribute is not writable.

Symtab.producer [Variable]
The name and possibly version number of the program that compiled the code in the symbol table. The contents of this string is up to the compiler. If no producer information is available then None is returned. This attribute is not writable.

A gdb.Symtab object has the following methods:

Symtab.is_valid () [Function]
Returns True if the gdb.Symtab object is valid, False if not. A gdb.Symtab object can become invalid if the symbol table it refers to does not exist in GDB any longer. All other gdb.Symtab methods will throw an exception if it is invalid at the time the method is called.

Symtab.fullname () [Function]
Return the symbol table's source absolute file name.

Symtab.global_block () [Function]
Return the global block of the underlying symbol table. See Section 23.3.2.26 [Blocks In Python], page 436.

Symtab.static_block () [Function]

Return the static block of the underlying symbol table. See Section 23.3.2.26 [Blocks In Python], page 436.

Symtab.linetable () [Function]

Return the line table associated with the symbol table. See Section 23.3.2.29 [Line Tables In Python], page 443.

23.3.2.29 Manipulating line tables using Python

Python code can request and inspect line table information from a symbol table that is loaded in GDB. A line table is a mapping of source lines to their executable locations in memory. To acquire the line table information for a particular symbol table, use the `linetable` function (see Section 23.3.2.28 [Symbol Tables In Python], page 441).

A `gdb.LineTable` is iterable. The iterator returns `LineTableEntry` objects that correspond to the source line and address for each line table entry. `LineTableEntry` objects have the following attributes:

LineTableEntry.line [Variable]

The source line number for this line table entry. This number corresponds to the actual line of source. This attribute is not writable.

LineTableEntry.pc [Variable]

The address that is associated with the line table entry where the executable code for that source line resides in memory. This attribute is not writable.

As there can be multiple addresses for a single source line, you may receive multiple `LineTableEntry` objects with matching `line` attributes, but with different `pc` attributes. The iterator is sorted in ascending `pc` order. Here is a small example illustrating iterating over a line table.

```
symtab = gdb.selected_frame().find_sal().symtab
linetable = symtab.linetable()
for line in linetable:
   print ("Line: "+str(line.line)+" Address: "+hex(line.pc))
```

This will have the following output:

```
Line: 33 Address: 0x4005c8L
Line: 37 Address: 0x4005caL
Line: 39 Address: 0x4005d2L
Line: 40 Address: 0x4005f8L
Line: 42 Address: 0x4005ffL
Line: 44 Address: 0x400608L
Line: 42 Address: 0x40060cL
Line: 45 Address: 0x400615L
```

In addition to being able to iterate over a `LineTable`, it also has the following direct access methods:

LineTable.line (*line*) [Function]

Return a Python `Tuple` of `LineTableEntry` objects for any entries in the line table for the given *line*, which specifies the source code line. If there are no entries for that source code *line*, the Python `None` is returned.

`LineTable.has_line` (*line*) [Function]

Return a Python `Boolean` indicating whether there is an entry in the line table for this source line. Return `True` if an entry is found, or `False` if not.

`LineTable.source_lines` () [Function]

Return a Python `List` of the source line numbers in the symbol table. Only lines with executable code locations are returned. The contents of the `List` will just be the source line entries represented as Python `Long` values.

23.3.2.30 Manipulating breakpoints using Python

Python code can manipulate breakpoints via the `gdb.Breakpoint` class.

A breakpoint can be created using one of the two forms of the `gdb.Breakpoint` constructor. The first one accepts a string like one would pass to the `break` (see Section 5.1.1 [Setting Breakpoints], page 56) and `watch` (see Section 5.1.2 [Setting Watchpoints], page 63) commands, and can be used to create both breakpoints and watchpoints. The second accepts separate Python arguments similar to Section 9.2.2 [Explicit Locations], page 121, and can only be used to create breakpoints.

`Breakpoint.__init__` (*spec* [, *type*][, *wp_class*][, *internal*][, *temporary*][, *qualified*]) [Function]

Create a new breakpoint according to *spec*, which is a string naming the location of a breakpoint, or an expression that defines a watchpoint. The string should describe a location in a format recognized by the `break` command (see Section 5.1.1 [Setting Breakpoints], page 56) or, in the case of a watchpoint, by the `watch` command (see Section 5.1.2 [Setting Watchpoints], page 63).

The optional *type* argument specifies the type of the breakpoint to create, as defined below.

The optional *wp_class* argument defines the class of watchpoint to create, if *type* is `gdb.BP_WATCHPOINT`. If *wp_class* is omitted, it defaults to `gdb.WP_WRITE`.

The optional *internal* argument allows the breakpoint to become invisible to the user. The breakpoint will neither be reported when created, nor will it be listed in the output from `info breakpoints` (but will be listed with the `maint info breakpoints` command).

The optional *temporary* argument makes the breakpoint a temporary breakpoint. Temporary breakpoints are deleted after they have been hit. Any further access to the Python breakpoint after it has been hit will result in a runtime error (as that breakpoint has now been automatically deleted).

The optional *qualified* argument is a boolean that allows interpreting the function passed in `spec` as a fully-qualified name. It is equivalent to `break`'s `-qualified` flag (see Section 9.2.1 [Linespec Locations], page 120, and Section 9.2.2 [Explicit Locations], page 121).

`Breakpoint.__init__` ([*source*][, *function*][, *label*][, *line*],][*internal*][, *temporary*][, *qualified*]) [Function]

This second form of creating a new breakpoint specifies the explicit location (see Section 9.2.2 [Explicit Locations], page 121) using keywords. The new breakpoint will be created in the specified source file *source*, at the specified *function*, *label* and *line*.

internal, *temporary* and *qualified* have the same usage as explained previously.

The available types are represented by constants defined in the `gdb` module:

`gdb.BP_BREAKPOINT`
: Normal code breakpoint.

`gdb.BP_HARDWARE_BREAKPOINT`
: Hardware assisted code breakpoint.

`gdb.BP_WATCHPOINT`
: Watchpoint breakpoint.

`gdb.BP_HARDWARE_WATCHPOINT`
: Hardware assisted watchpoint.

`gdb.BP_READ_WATCHPOINT`
: Hardware assisted read watchpoint.

`gdb.BP_ACCESS_WATCHPOINT`
: Hardware assisted access watchpoint.

The available watchpoint types are represented by constants defined in the `gdb` module:

`gdb.WP_READ`
: Read only watchpoint.

`gdb.WP_WRITE`
: Write only watchpoint.

`gdb.WP_ACCESS`
: Read/Write watchpoint.

`Breakpoint.stop` (*self*) [Function]

The `gdb.Breakpoint` class can be sub-classed and, in particular, you may choose to implement the `stop` method. If this method is defined in a sub-class of `gdb.Breakpoint`, it will be called when the inferior reaches any location of a breakpoint which instantiates that sub-class. If the method returns `True`, the inferior will be stopped at the location of the breakpoint, otherwise the inferior will continue.

If there are multiple breakpoints at the same location with a `stop` method, each one will be called regardless of the return status of the previous. This ensures that all `stop` methods have a chance to execute at that location. In this scenario if one of the methods returns `True` but the others return `False`, the inferior will still be stopped.

You should not alter the execution state of the inferior (i.e., step, next, etc.), alter the current frame context (i.e., change the current active frame), or alter, add or delete any breakpoint. As a general rule, you should not alter any data within GDB or the inferior at this time.

Example `stop` implementation:

```
class MyBreakpoint (gdb.Breakpoint):
      def stop (self):
```

```
inf_val = gdb.parse_and_eval("foo")
if inf_val == 3:
  return True
return False
```

`Breakpoint.is_valid ()` [Function]

Return `True` if this `Breakpoint` object is valid, `False` otherwise. A `Breakpoint` object can become invalid if the user deletes the breakpoint. In this case, the object still exists, but the underlying breakpoint does not. In the cases of watchpoint scope, the watchpoint remains valid even if execution of the inferior leaves the scope of that watchpoint.

`Breakpoint.delete ()` [Function]

Permanently deletes the GDB breakpoint. This also invalidates the Python `Breakpoint` object. Any further access to this object's attributes or methods will raise an error.

`Breakpoint.enabled` [Variable]

This attribute is `True` if the breakpoint is enabled, and `False` otherwise. This attribute is writable. You can use it to enable or disable the breakpoint.

`Breakpoint.silent` [Variable]

This attribute is `True` if the breakpoint is silent, and `False` otherwise. This attribute is writable.

Note that a breakpoint can also be silent if it has commands and the first command is `silent`. This is not reported by the `silent` attribute.

`Breakpoint.pending` [Variable]

This attribute is `True` if the breakpoint is pending, and `False` otherwise. See Section 5.1.1 [Set Breaks], page 56. This attribute is read-only.

`Breakpoint.thread` [Variable]

If the breakpoint is thread-specific, this attribute holds the thread's global id. If the breakpoint is not thread-specific, this attribute is `None`. This attribute is writable.

`Breakpoint.task` [Variable]

If the breakpoint is Ada task-specific, this attribute holds the Ada task id. If the breakpoint is not task-specific (or the underlying language is not Ada), this attribute is `None`. This attribute is writable.

`Breakpoint.ignore_count` [Variable]

This attribute holds the ignore count for the breakpoint, an integer. This attribute is writable.

`Breakpoint.number` [Variable]

This attribute holds the breakpoint's number — the identifier used by the user to manipulate the breakpoint. This attribute is not writable.

`Breakpoint.type` [Variable]

This attribute holds the breakpoint's type — the identifier used to determine the actual breakpoint type or use-case. This attribute is not writable.

Breakpoint.visible [Variable]

This attribute tells whether the breakpoint is visible to the user when set, or when the 'info breakpoints' command is run. This attribute is not writable.

Breakpoint.temporary [Variable]

This attribute indicates whether the breakpoint was created as a temporary breakpoint. Temporary breakpoints are automatically deleted after that breakpoint has been hit. Access to this attribute, and all other attributes and functions other than the is_valid function, will result in an error after the breakpoint has been hit (as it has been automatically deleted). This attribute is not writable.

Breakpoint.hit_count [Variable]

This attribute holds the hit count for the breakpoint, an integer. This attribute is writable, but currently it can only be set to zero.

Breakpoint.location [Variable]

This attribute holds the location of the breakpoint, as specified by the user. It is a string. If the breakpoint does not have a location (that is, it is a watchpoint) the attribute's value is None. This attribute is not writable.

Breakpoint.expression [Variable]

This attribute holds a breakpoint expression, as specified by the user. It is a string. If the breakpoint does not have an expression (the breakpoint is not a watchpoint) the attribute's value is None. This attribute is not writable.

Breakpoint.condition [Variable]

This attribute holds the condition of the breakpoint, as specified by the user. It is a string. If there is no condition, this attribute's value is None. This attribute is writable.

Breakpoint.commands [Variable]

This attribute holds the commands attached to the breakpoint. If there are commands, this attribute's value is a string holding all the commands, separated by newlines. If there are no commands, this attribute is None. This attribute is writable.

23.3.2.31 Finish Breakpoints

A finish breakpoint is a temporary breakpoint set at the return address of a frame, based on the finish command. gdb.FinishBreakpoint extends gdb.Breakpoint. The underlying breakpoint will be disabled and deleted when the execution will run out of the breakpoint scope (i.e. Breakpoint.stop or FinishBreakpoint.out_of_scope triggered). Finish breakpoints are thread specific and must be create with the right thread selected.

FinishBreakpoint.__init__ ([*frame*] [, *internal*]) [Function]

Create a finish breakpoint at the return address of the gdb.Frame object *frame*. If *frame* is not provided, this defaults to the newest frame. The optional *internal* argument allows the breakpoint to become invisible to the user. See Section 23.3.2.30 [Breakpoints In Python], page 444, for further details about this argument.

`FinishBreakpoint.out_of_scope` (*self*) [Function]

In some circumstances (e.g. `longjmp`, C++ exceptions, GDB `return` command, . . .), a function may not properly terminate, and thus never hit the finish breakpoint. When GDB notices such a situation, the `out_of_scope` callback will be triggered.

You may want to sub-class `gdb.FinishBreakpoint` and override this method:

```
class MyFinishBreakpoint (gdb.FinishBreakpoint)
    def stop (self):
        print ("normal finish")
        return True

    def out_of_scope ():
        print ("abnormal finish")
```

`FinishBreakpoint.return_value` [Variable]

When GDB is stopped at a finish breakpoint and the frame used to build the `gdb.FinishBreakpoint` object had debug symbols, this attribute will contain a `gdb.Value` object corresponding to the return value of the function. The value will be `None` if the function return type is `void` or if the return value was not computable. This attribute is not writable.

23.3.2.32 Python representation of lazy strings

A *lazy string* is a string whose contents is not retrieved or encoded until it is needed.

A `gdb.LazyString` is represented in GDB as an `address` that points to a region of memory, an `encoding` that will be used to encode that region of memory, and a `length` to delimit the region of memory that represents the string. The difference between a `gdb.LazyString` and a string wrapped within a `gdb.Value` is that a `gdb.LazyString` will be treated differently by GDB when printing. A `gdb.LazyString` is retrieved and encoded during printing, while a `gdb.Value` wrapping a string is immediately retrieved and encoded on creation.

A `gdb.LazyString` object has the following functions:

`LazyString.value ()` [Function]

Convert the `gdb.LazyString` to a `gdb.Value`. This value will point to the string in memory, but will lose all the delayed retrieval, encoding and handling that GDB applies to a `gdb.LazyString`.

`LazyString.address` [Variable]

This attribute holds the address of the string. This attribute is not writable.

`LazyString.length` [Variable]

This attribute holds the length of the string in characters. If the length is -1, then the string will be fetched and encoded up to the first null of appropriate width. This attribute is not writable.

`LazyString.encoding` [Variable]

This attribute holds the encoding that will be applied to the string when the string is printed by GDB. If the encoding is not set, or contains an empty string, then GDB will select the most appropriate encoding when the string is printed. This attribute is not writable.

`LazyString.type` [Variable]

This attribute holds the type that is represented by the lazy string's type. For a lazy string this is a pointer or array type. To resolve this to the lazy string's character type, use the type's `target` method. See Section 23.3.2.4 [Types In Python], page 384. This attribute is not writable.

23.3.2.33 Python representation of architectures

GDB uses architecture specific parameters and artifacts in a number of its various computations. An architecture is represented by an instance of the `gdb.Architecture` class.

A `gdb.Architecture` class has the following methods:

`Architecture.name ()` [Function]

Return the name (string value) of the architecture.

`Architecture.disassemble (`*`start_pc`* `[, `*`end_pc`* `[, `*`count`*`]])` [Function]

Return a list of disassembled instructions starting from the memory address *start_pc*. The optional arguments *end_pc* and *count* determine the number of instructions in the returned list. If both the optional arguments *end_pc* and *count* are specified, then a list of at most *count* disassembled instructions whose start address falls in the closed memory address interval from *start_pc* to *end_pc* are returned. If *end_pc* is not specified, but *count* is specified, then *count* number of instructions starting from the address *start_pc* are returned. If *count* is not specified but *end_pc* is specified, then all instructions whose start address falls in the closed memory address interval from *start_pc* to *end_pc* are returned. If neither *end_pc* nor *count* are specified, then a single instruction at *start_pc* is returned. For all of these cases, each element of the returned list is a Python `dict` with the following string keys:

`addr` The value corresponding to this key is a Python long integer capturing the memory address of the instruction.

`asm` The value corresponding to this key is a string value which represents the instruction with assembly language mnemonics. The assembly language flavor used is the same as that specified by the current CLI variable `disassembly-flavor`. See Section 9.6 [Machine Code], page 128.

`length` The value corresponding to this key is the length (integer value) of the instruction in bytes.

`Architecture.registers ([ `*`reggroup`*` ])` [Function]

Return a `gdb.RegisterDescriptorIterator` (see Section 23.3.2.34 [Registers In Python], page 450) for all of the registers in *reggroup*, a string that is the name of a register group. If *reggroup* is omitted, or is the empty string, then the register group '`all`' is assumed.

`Architecture.register_groups ()` [Function]

Return a `gdb.RegisterGroupsIterator` (see Section 23.3.2.34 [Registers In Python], page 450) for all of the register groups available for the `gdb.Architecture`.

23.3.2.34 Registers In Python

Python code can request from a `gdb.Architecture` information about the set of registers available (see [`Architecture.registers`], page 449). The register information is returned as a `gdb.RegisterDescriptorIterator`, which is an iterator that in turn returns `gdb.RegisterDescriptor` objects.

A `gdb.RegisterDescriptor` does not provide the value of a register (see [`Frame.read_register`], page 435, for reading a register's value), instead the `RegisterDescriptor` is a way to discover which registers are available for a particular architecture.

A `gdb.RegisterDescriptor` has the following read-only properties:

`RegisterDescriptor.name` [Variable]

The name of this register.

It is also possible to lookup a register descriptor based on its name using the following `gdb.RegisterDescriptorIterator` function:

`RegisterDescriptorIterator.find` (*name*) [Function]

Takes *name* as an argument, which must be a string, and returns a `gdb.RegisterDescriptor` for the register with that name, or `None` if there is no register with that name.

Python code can also request from a `gdb.Architecture` information about the set of register groups available on a given architecture (see [`Architecture.register_groups`], page 449).

Every register can be a member of zero or more register groups. Some register groups are used internally within GDB to control things like which registers must be saved when calling into the program being debugged (see Section 17.5 [Calling Program Functions], page 265). Other register groups exist to allow users to easily see related sets of registers in commands like `info registers` (see [`info registers` *reggroup*], page 168).

The register groups information is returned as a `gdb.RegisterGroupsIterator`, which is an iterator that in turn returns `gdb.RegisterGroup` objects.

A `gdb.RegisterGroup` object has the following read-only properties:

`RegisterGroup.name` [Variable]

A string that is the name of this register group.

23.3.2.35 Implementing new TUI windows

New TUI (see Chapter 25 [TUI], page 515) windows can be implemented in Python.

`gdb.register_window_type` (*name*, *factory*) [Function]

Because TUI windows are created and destroyed depending on the layout the user chooses, new window types are implemented by registering a factory function with GDB.

name is the name of the new window. It's an error to try to replace one of the built-in windows, but other window types can be replaced.

function is a factory function that is called to create the TUI window. This is called with a single argument of type `gdb.TuiWindow`, described below. It should return an object that implements the TUI window protocol, also described below.

As mentioned above, when a factory function is called, it is passed an object of type `gdb.TuiWindow`. This object has these methods and attributes:

`TuiWindow.is_valid ()` [Function]

This method returns `True` when this window is valid. When the user changes the TUI layout, windows no longer visible in the new layout will be destroyed. At this point, the `gdb.TuiWindow` will no longer be valid, and methods (and attributes) other than `is_valid` will throw an exception.

When the TUI is disabled using `tui disable` (see Section 25.4 [tui disable], page 518) the window is hidden rather than destroyed, but `is_valid` will still return `False` and other methods (and attributes) will still throw an exception.

`TuiWindow.width` [Variable]

This attribute holds the width of the window. It is not writable.

`TuiWindow.height` [Variable]

This attribute holds the height of the window. It is not writable.

`TuiWindow.title` [Variable]

This attribute holds the window's title, a string. This is normally displayed above the window. This attribute can be modified.

`TuiWindow.erase ()` [Function]

Remove all the contents of the window.

`TuiWindow.write (`*`string`*`)` [Function]

Write *string* to the window. *string* can contain ANSI terminal escape styling sequences; GDB will translate these as appropriate for the terminal.

The factory function that you supply should return an object conforming to the TUI window protocol. These are the method that can be called on this object, which is referred to below as the "window object". The methods documented below are optional; if the object does not implement one of these methods, GDB will not attempt to call it. Additional new methods may be added to the window protocol in the future. GDB guarantees that they will begin with a lower-case letter, so you can start implementation methods with upper-case letters or underscore to avoid any future conflicts.

`Window.close ()` [Function]

When the TUI window is closed, the `gdb.TuiWindow` object will be put into an invalid state. At this time, GDB will call `close` method on the window object.

After this method is called, GDB will discard any references it holds on this window object, and will no longer call methods on this object.

`Window.render ()` [Function]

In some situations, a TUI window can change size. For example, this can happen if the user resizes the terminal, or changes the layout. When this happens, GDB will call the `render` method on the window object.

If your window is intended to update in response to changes in the inferior, you will probably also want to register event listeners and send output to the `gdb.TuiWindow`.

`Window.hscroll (num)` [Function]

This is a request to scroll the window horizontally. *num* is the amount by which to scroll, with negative numbers meaning to scroll right. In the TUI model, it is the viewport that moves, not the contents. A positive argument should cause the viewport to move right, and so the content should appear to move to the left.

`Window.vscroll (num)` [Function]

This is a request to scroll the window vertically. *num* is the amount by which to scroll, with negative numbers meaning to scroll backward. In the TUI model, it is the viewport that moves, not the contents. A positive argument should cause the viewport to move down, and so the content should appear to move up.

23.3.3 Python Auto-loading

When a new object file is read (for example, due to the `file` command, or because the inferior has loaded a shared library), GDB will look for Python support scripts in several ways: `objfile-gdb.py` and `.debug_gdb_scripts` section. See Section 23.5 [Auto-loading extensions], page 507.

The auto-loading feature is useful for supplying application-specific debugging commands and scripts.

Auto-loading can be enabled or disabled, and the list of auto-loaded scripts can be printed.

`set auto-load python-scripts [on|off]`
: Enable or disable the auto-loading of Python scripts.

`show auto-load python-scripts`
: Show whether auto-loading of Python scripts is enabled or disabled.

`info auto-load python-scripts [regexp]`
: Print the list of all Python scripts that GDB auto-loaded.

Also printed is the list of Python scripts that were mentioned in the `.debug_gdb_scripts` section and were either not found (see Section 23.5.2 [dotdebug_gdb_scripts section], page 508) or were not auto-loaded due to `auto-load safe-path` rejection (see Section 22.8 [Auto-loading], page 348). This is useful because their names are not printed when GDB tries to load them and fails. There may be many of them, and printing an error message for each one is problematic.

If *regexp* is supplied only Python scripts with matching names are printed.

Example:

```
(gdb) info auto-load python-scripts
Loaded Script
Yes    py-section-script.py
       full name: /tmp/py-section-script.py
No     my-foo-pretty-printers.py
```

When reading an auto-loaded file or script, GDB sets the *current objfile*. This is available via the `gdb.current_objfile` function (see Section 23.3.2.24 [Objfiles In Python], page 430). This can be useful for registering objfile-specific pretty-printers and frame-filters.