Supporting Cyclic Garbage Collection

Python's support for detecting and collecting garbage which involves circular references requires support from object types which are "containers" for other objects which may also be containers. Types which do not store references to other objects, or which only store references to atomic types (such as numbers or strings), do not need to provide any explicit support for garbage collection.

To create a container type, the `tp_flags` field of the type object must include the `Py_TPFLAGS_HAVE_GC` and provide an implementation of the `tp_traverse` handler. If instances of the type are mutable, a `tp_clear` implementation must also be provided.

`Py_TPFLAGS_HAVE_GC`

Objects with a type with this flag set must conform with the rules documented here. For convenience these objects will be referred to as container objects.

Constructors for container types must conform to two rules:

1. The memory for the object must be allocated using `PyObject_GC_New()` or `PyObject_GC_NewVar()`.
2. Once all the fields which may contain references to other containers are initialized, it must call `PyObject_GC_Track()`.

TYPE* **`PyObject_GC_New`**(TYPE, PyTypeObject **type*)

Analogous to `PyObject_New()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

TYPE* **`PyObject_GC_NewVar`**(TYPE, PyTypeObject **type*, Py_ssize_t *size*)

Analogous to `PyObject_NewVar()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

TYPE* **`PyObject_GC_Resize`**(TYPE, PyVarObject **op*, Py_ssize_t *newsize*)

Resize an object allocated by `PyObject_NewVar()`. Returns the resized object or *NULL* on failure. *op* must not be tracked by the collector yet.

void **`PyObject_GC_Track`**(PyObject **op*)

Adds the object *op* to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the `tp_traverse` handler become valid, usually near the end of the constructor.

void **`_PyObject_GC_TRACK`**(PyObject **op*)

A macro version of PyObject_GC_Track(). It should not be used for extension modules.

Similarly, the deallocator for the object must conform to a similar pair of rules:

1. Before fields which refer to other containers are invalidated, PyObject_GC_UnTrack() must be called.
2. The object's memory must be deallocated using PyObject_GC_Del().

void **PyObject_GC_Del**(void **op*)

Releases memory allocated to an object using PyObject_GC_New() or PyObject_GC_NewVar().

void **PyObject_GC_UnTrack**(void **op*)

Remove the object *op* from the set of container objects tracked by the collector. Note that PyObject_GC_Track() can be called again on this object to add it back to the set of tracked objects. The deallocator (tp_dealloc handler) should call this for the object before any of the fields used by the tp_traverse handler become invalid.

void **_PyObject_GC_UNTRACK**(PyObject **op*)

A macro version of PyObject_GC_UnTrack(). It should not be used for extension modules.

The tp_traverse handler accepts a function parameter of this type:

int **(*visitproc)**(PyObject **object*, void **arg*)

Type of the visitor function passed to the tp_traverse handler. The function should be called with an object to traverse as *object* and the third parameter to the tp_traverse handler as *arg*. The Python core uses several visitor functions to implement cyclic garbage detection; it's not expected that users will need to write their own visitor functions.

The tp_traverse handler must have the following type:

int **(*traverseproc)**(PyObject **self*, visitproc *visit*, void **arg*)

Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a *NULL* object argument. If *visit* returns a non-zero value that value should be returned immediately.

To simplify writing tp_traverse handlers, a Py_VISIT() macro is provided. In order to use this macro, the tp_traverse implementation must name its arguments exactly *visit* and *arg*:

void **Py_VISIT**(PyObject *o)

If *o* is not *NULL*, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, `tp_traverse` handlers look like:

```c
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

The `tp_clear` handler must be of the `inquiry` type, or *NULL* if the object is immutable.

int **(*inquiry)**(PyObject *self)

Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don't just call `Py_DECREF()` on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.