Object Protocol

PyObject* **Py_NotImplemented**

The `NotImplemented` singleton, used to signal that an operation is not implemented for the given type combination.

Py_RETURN_NOTIMPLEMENTED

Properly handle returning `Py_NotImplemented` from within a C function (that is, increment the reference count of NotImplemented and return it).

int **PyObject_Print**(PyObject **o*, FILE **fp*, int *flags*)

Print an object *o*, on file *fp*. Returns `-1` on error. The flags argument is used to enable certain printing options. The only option currently supported is `Py_PRINT_RAW`; if given, the `str()` of the object is written instead of the `repr()`.

int **PyObject_HasAttr**(PyObject **o*, PyObject **attr_name*)

Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. This function always succeeds.

int **PyObject_HasAttrString**(PyObject **o*, const char **attr_name*)

Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. This function always succeeds.

PyObject* **PyObject_GetAttr**(PyObject **o*, PyObject **attr_name*)

Return value: New reference.

Retrieve an attribute named *attr_name* from object *o*. Returns the attribute value on success, or *NULL* on failure. This is the equivalent of the Python expression `o.attr_name`.

PyObject* **PyObject_GetAttrString**(PyObject **o*, const char **attr_name*)

Return value: New reference.

Retrieve an attribute named *attr_name* from object *o*. Returns the attribute value on success, or *NULL* on failure. This is the equivalent of the Python expression `o.attr_name`.

PyObject* **PyObject_GenericGetAttr**(PyObject **o*, PyObject **name*)

Generic attribute getter function that is meant to be put into a type object's `tp_getattro` slot. It looks for a descriptor in the dictionary of classes in the object's MRO as well as an attribute in the object's `__dict__` (if present). As outlined in Implementing Descriptors, data descriptors take preference over in-

stance attributes, while non-data descriptors don't. Otherwise, an `AttributeError` is raised.

int **PyObject_SetAttr**(PyObject *o, PyObject *attr_name, PyObject *v)

Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

If *v* is *NULL*, the attribute is deleted, however this feature is deprecated in favour of using `PyObject_DelAttr()`.

int **PyObject_SetAttrString**(PyObject *o, const char *attr_name, PyObject *v)

Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

If *v* is *NULL*, the attribute is deleted, however this feature is deprecated in favour of using `PyObject_DelAttrString()`.

int **PyObject_GenericSetAttr**(PyObject *o, PyObject *name, PyObject *value)

Generic attribute setter and deleter function that is meant to be put into a type object's `tp_setattro` slot. It looks for a data descriptor in the dictionary of classes in the object's MRO, and if found it takes preference over setting or deleting the attribute in the instance dictionary. Otherwise, the attribute is set or deleted in the object's `__dict__` (if present). On success, `0` is returned, otherwise an `AttributeError` is raised and `-1` is returned.

int **PyObject_DelAttr**(PyObject *o, PyObject *attr_name)

Delete attribute named *attr_name*, for object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.

int **PyObject_DelAttrString**(PyObject *o, const char *attr_name)

Delete attribute named *attr_name*, for object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.

PyObject* **PyObject_GenericGetDict**(PyObject *o, void *context)

A generic implementation for the getter of a `__dict__` descriptor. It creates the dictionary if necessary.

New in version 3.3.

int **PyObject_GenericSetDict**(PyObject *o, void *context)

A generic implementation for the setter of a `__dict__` descriptor. This implementation does not allow the dictionary to be deleted.

New in version 3.3.

PyObject* **PyObject_RichCompare**(PyObject *o1, PyObject *o2, int *opid*)

Return value: New reference.

Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, or `Py_GE`, corresponding to `<`, `<=`, `==`, `!=`, `>`, or `>=` respectively. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to *opid*. Returns the value of the comparison on success, or *NULL* on failure.

int **PyObject_RichCompareBool**(PyObject *o1, PyObject *o2, int *opid*)

Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, or `Py_GE`, corresponding to `<`, `<=`, `==`, `!=`, `>`, or `>=` respectively. Returns `-1` on error, `0` if the result is false, `1` otherwise. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to *opid*.

Note: If *o1* and *o2* are the same object, `PyObject_RichCompareBool()` will always return `1` for `Py_EQ` and `0` for `Py_NE`.

PyObject* **PyObject_Repr**(PyObject *o)

Return value: New reference.

Compute a string representation of object *o*. Returns the string representation on success, *NULL* on failure. This is the equivalent of the Python expression `repr(o)`. Called by the `repr()` built-in function.

Changed in version 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

PyObject* **PyObject_ASCII**(PyObject *o)

As `PyObject_Repr()`, compute a string representation of object *o*, but escape the non-ASCII characters in the string returned by `PyObject_Repr()` with `\x`, `\u` or `\U` escapes. This generates a string similar to that returned by `PyObject_Repr()` in Python 2. Called by the `ascii()` built-in function.

PyObject* **PyObject_Str**(PyObject *o)

Return value: New reference.

Compute a string representation of object *o*. Returns the string representation on success, *NULL* on failure. This is the equivalent of the Python expression `str(o)`. Called by the `str()` built-in function and, therefore, by the `print()` function.

Changed in version 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

PyObject* **PyObject_Bytes**(PyObject *o)

Compute a bytes representation of object *o*. *NULL* is returned on failure and a bytes object on success. This is equivalent to the Python expression `bytes(o)`, when *o* is not an integer. Unlike `bytes(o)`, a TypeError is raised when *o* is an integer instead of a zero-initialized bytes object.

int **PyObject_IsSubclass**(PyObject *derived, PyObject *cls)

Return `1` if the class *derived* is identical to or derived from the class *cls*, otherwise return `0`. In case of an error, return `-1`.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`.

If *cls* has a `__subclasscheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *derived* is a subclass of *cls* if it is a direct or indirect subclass, i.e. contained in `cls.__mro__`.

Normally only class objects, i.e. instances of `type` or a derived class, are considered classes. However, objects can override this by having a `__bases__` attribute (which must be a tuple of base classes).

int **PyObject_IsInstance**(PyObject *inst, PyObject *cls)

Return `1` if *inst* is an instance of the class *cls* or a subclass of *cls*, or `0` if not. On error, returns `-1` and sets an exception.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`.

If *cls* has a `__instancecheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *inst* is an instance of *cls* if its class is a subclass of *cls*.

An instance *inst* can override what is considered its class by having a `__class__` attribute.

An object *cls* can override if it is considered a class, and what its base classes are, by having a `__bases__` attribute (which must be a tuple of base classes).

int **PyCallable_Check**(PyObject *o)

Determine if the object *o* is callable. Return `1` if the object is callable and `0` otherwise. This function always succeeds.

PyObject* **PyObject_Call**(PyObject **callable_object*, PyObject **args*, PyObject **kw*)

Return value: New reference.

Call a callable Python object *callable_object*, with arguments given by the tuple *args*, and named arguments given by the dictionary *kw*. If no named arguments are needed, *kw* may be *NULL*. *args* must not be *NULL*, use an empty tuple if no arguments are needed. Returns the result of the call on success, or *NULL* on failure. This is the equivalent of the Python expression `callable_object(*args, **kw)`.

PyObject* **PyObject_CallObject**(PyObject **callable_object*, PyObject **args*)

Return value: New reference.

Call a callable Python object *callable_object*, with arguments given by the tuple *args*. If no arguments are needed, then *args* may be *NULL*. Returns the result of the call on success, or *NULL* on failure. This is the equivalent of the Python expression `callable_object(*args)`.

PyObject* **PyObject_CallFunction**(PyObject **callable*, const char **format*, ...)

Return value: New reference.

Call a callable Python object *callable*, with a variable number of C arguments. The C arguments are described using a `Py_BuildValue()` style format string. The format may be *NULL*, indicating that no arguments are provided. Returns the result of the call on success, or *NULL* on failure. This is the equivalent of the Python expression `callable(*args)`. Note that if you only pass `PyObject *` args, `PyObject_CallFunctionObjArgs()` is a faster alternative.

Changed in version 3.4: The type of *format* was changed from `char *`.

PyObject* **PyObject_CallMethod**(PyObject **o*, const char **method*, const char **format*, ...)

Return value: New reference.

Call the method named *method* of object *o* with a variable number of C arguments. The C arguments are described by a `Py_BuildValue()` format string that should produce a tuple. The format may be *NULL*, indicating that no arguments are provided. Returns the result of the call on success, or *NULL* on failure. This is the equivalent of the Python expression `o.method(args)`. Note that if you only pass `PyObject *` args, `PyObject_CallMethodObjArgs()` is a faster alternative.

Changed in version 3.4: The types of *method* and *format* were changed from `char *`.

PyObject* **PyObject_CallFunctionObjArgs**(PyObject *callable, ..., NULL)

Return value: New reference.

Call a callable Python object *callable*, with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by *NULL*. Returns the result of the call on success, or *NULL* on failure.

PyObject* **PyObject_CallMethodObjArgs**(PyObject *o, PyObject *name, ..., NULL)

Return value: New reference.

Calls a method of the object *o*, where the name of the method is given as a Python string object in *name*. It is called with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by *NULL*. Returns the result of the call on success, or *NULL* on failure.

Py_hash_t **PyObject_Hash**(PyObject *o)

Compute and return the hash value of an object *o*. On failure, return `-1`. This is the equivalent of the Python expression `hash(o)`.

Changed in version 3.2: The return type is now Py_hash_t. This is a signed integer the same size as Py_ssize_t.

Py_hash_t **PyObject_HashNotImplemented**(PyObject *o)

Set a `TypeError` indicating that `type(o)` is not hashable and return `-1`. This function receives special treatment when stored in a `tp_hash` slot, allowing a type to explicitly indicate to the interpreter that it is not hashable.

int **PyObject_IsTrue**(PyObject *o)

Returns `1` if the object *o* is considered to be true, and `0` otherwise. This is equivalent to the Python expression `not not o`. On failure, return `-1`.

int **PyObject_Not**(PyObject *o)

Returns `0` if the object *o* is considered to be true, and `1` otherwise. This is equivalent to the Python expression `not o`. On failure, return `-1`.

PyObject* **PyObject_Type**(PyObject *o)

Return value: New reference.

When *o* is non-*NULL*, returns a type object corresponding to the object type of object *o*. On failure, raises `SystemError` and returns *NULL*. This is equivalent to the Python expression `type(o)`. This function increments the reference count of the return value. There's really no reason to use this function instead of the common expression `o->ob_type`, which returns a pointer of type `PyTypeObject*`, except when the incremented reference count is needed.

int **PyObject_TypeCheck**(PyObject *o, PyTypeObject *type)

Return true if the object *o* is of type *type* or a subtype of *type*. Both parameters must be non-*NULL*.

Py_ssize_t **PyObject_Size**(PyObject **o*)
Py_ssize_t **PyObject_Length**(PyObject **o*)

Return the length of object *o*. If the object *o* provides either the sequence and mapping protocols, the sequence length is returned. On error, `-1` is returned. This is the equivalent to the Python expression `len(o)`.

Py_ssize_t **PyObject_LengthHint**(PyObject **o*, Py_ssize_t *default*)

Return an estimated length for the object *o*. First try to return its actual length, then an estimate using `__length_hint__()`, and finally return the default value. On error return `-1`. This is the equivalent to the Python expression `operator.length_hint(o, default)`.

New in version 3.4.

PyObject* **PyObject_GetItem**(PyObject **o*, PyObject **key*)

Return value: New reference.

Return element of *o* corresponding to the object *key* or *NULL* on failure. This is the equivalent of the Python expression `o[key]`.

int **PyObject_SetItem**(PyObject **o*, PyObject **key*, PyObject **v*)

Map the object *key* to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[key] = v`.

int **PyObject_DelItem**(PyObject **o*, PyObject **key*)

Remove the mapping for the object *key* from the object *o*. Return `-1` on failure. This is equivalent to the Python statement `del o[key]`.

PyObject* **PyObject_Dir**(PyObject **o*)

Return value: New reference.

This is equivalent to the Python expression `dir(o)`, returning a (possibly empty) list of strings appropriate for the object argument, or *NULL* if there was an error. If the argument is *NULL*, this is like the Python `dir()`, returning the names of the current locals; in this case, if no execution frame is active then *NULL* is returned but `PyErr_Occurred()` will return false.

PyObject* **PyObject_GetIter**(PyObject **o*)

Return value: New reference.

This is equivalent to the Python expression `iter(o)`. It returns a new iterator for the object argument, or the object itself if the object is already an iterator. Raises `TypeError` and returns *NULL* if the object cannot be iterated.