pythoncapi Documentation

Release 0.1

Victor Stinner

Apr 14, 2021

Contents

Subtitle: “Make the stable API usable”.

PyPy and *Gilectomy* projects are using different approaches to optimize Python, but their performance are limited by the *current C API* on C extensions. The C API should be “fixed” to unlock raw performances!

To be able to introduce backward incompatible changes to the C API without breaking too many C extensions, this project proposes two things:

- design a *helper layer* providing *removed functions*;
- a new *Python runtime* which is only usable with C extensions compiled with the new stricter and smaller C API (and the new *stable ABI*) for Python 3.8 and newer, whereas the existing “regular python” becomes the “regular runtime” which provides maximum backward compatibility with Python 3.7 and older.

The current C API has multiple issues:

- The Python lifecycle is shorter than the *lifecycle of some operating systems*: how to get the latest Python on an “old” but stable operating system?
- *Python debug build* is currently mostly unusable in practice, making development of C extension harder, especially debugging.

Existing C extensions will still be supported and will not have to be modified. The *old C API* is not deprecated and there is no plan penalize users of the old C API.

CHAPTER 1

Pages

1.1 Fix the Python C API to optimize Python

CPython cannot be optimized and other Python implementations see their performances limited by the C API. The relationship between the C API and performance is not obvious, and even can be counterintuitive.

1.1.1 Optimizations

Faster object allocation

CPython requires to allocate all Python objects on the heap memory and objects cannot be moved during their life cycle.

It would be more efficient to allow to allocate temporary objects on the stack, implement nurseries of young objects and compact memory to remove "holes" when many objects are deallocated.

Faster and incremental garbage collection

CPython relies on reference counting to collect garbage. Reference counting does not scale for parallelism with multithreading.

A tracing and moving garbage collector would be more efficient. The garbage collection could be done in multiple steps in separated thread rather than having long delays causing by the CPython blocking stop-the-world garbage collector.

The ability to deference pointers like `PyObject*` makes the implementation of a moving garbarge collector more complicated. Only using handles would make the implementation simpler.

Run Python threads in parallel

CPython uses a GIL for objects consistency and to ease the implementation of the C API. The GIL has many convenient advantages to simplify the implementation. But it basically limits CPython to a single thread to run CPU-bound workload distributed in multiple threads.

Per-object locks would allow to help scaling threads on multiple CPU.

More efficient data structures (boxing/unboxing)

CPython requires builtin types like list and dict to only contain `PyObject*`.

PyPy implements a list strategy for integers: integers are stored directly as integers, not as objects. Integers are only boxed on demand.

1.1.2 Reasons why the C API prevents to optimize Python

Structures are part of the public C API (make them opaque)

Core C structures like `PyObject` are part of the public C API and so every Python implementations must implement exactly this structure.

The C API directly or indirectly access structure members. For example, the `Py_INCREF()` function modifies directly `PyObject.ob_refcnt` and so makes the assumption that objects have a reference counter. Another example is `PyTuple_GET_ITEM()` which reads directly the `PyTupleObject.ob_item` member and so requires `PyTupleObject` to only store `PyObject*` objects.

The C API should be modified to abstract accesses to objects through function calls rather than using macros which access directly structure members.

Structures must be excluded from the public C API: become "opaque".

PyObject* type can be dereferenced (use handles)

Since structures are public, it is possible to deference pointers to access structure members. For example, access directly to `PyObject.ob_type` member from a `PyObject*` pointer, or access directly to `PyTupleObject.ob_type[index]` from a `PyTupleObject*` pointer.

Using opaque **handles** like HPy does would prevent that.

Borrowed references (avoid them)

Many C API functions like `PyDict_GetItem()` or `PyTuple_GetItem()` return a borrowed reference. They make the assumption that all objects are actual objects. For example, if tagged pointers are implemented, a `PyObject*` does not point to a concrete object: the value must be boxed to get a `PyObject*`. The problem with borrowed reference is to decide when it is safe to destroy the temporary `PyObject*` object. One heuristic is to consider that it must remain alive as long as its container (ex: a list) remains alive.

PyObject must be allocated on the stack

In CPython, all objects must be allocated on the stack. Using reference counting, when an object is passed to a function, the function can store it in another container and so the object remains alive after the function completes.

The caller cannot destroy the object, since it does not take care of the object lifecycle. The object can only be destroyed when the last strong reference to the object is deleted.

Pseudo-code:

```
void func(void)
{
    PyObject *x = PyLong_FromLong(1);
    func(x);
    Py_DECREF(x);
    // if func() creates a new strong reference to x,
    // x is still alive at this point.
}
```

HPy uses a different strategy: if a function wants to create a new reference to a handle, `HPy_Dup()` function must be called. `HPy_Dup()` can continue to use the same object, but it can also duplicate an immutable object.

PyObject cannot be moved in memory

Since `PyObject*` is a direct memory address to a `PyObject`, moving a `PyObject` requires to change all `PyObject*` values pointing to it. Using handles, there is not such issue.

Other C API functions give a direct memory address into an object content with no API to "release" the resource. For example, `PyBytes_AsString()` gives a direct access into the bytes string, there is no way for the object to know when the caller no longer needs this pointer. The string cannot be moved in memory.

Functions using `PyObject**` type (array of `PyObject*` pointers) have a similar issue. Example: `&PyTuple_GET_ITEM()` is used to get `&PyTupleObject.ob_item`.

The `PyObject_GetBuffer()` is a sane API: it requires the caller to call `PuBuffer_Release()` to release the `Py_buffer` object. Memory can be copied if needed to allow moving the object while the buffer is used.

1.2 Rationale to change the Python C API

To be able to introduce backward incompatible changes to the C API without breaking too many C extensions, this project proposes two things:

- design a *helper layer* providing *removed functions*;
- a new *Python runtime* which is only usable with C extensions compiled with the new stricter and smaller C API (and the new *stable ABI*) for Python 3.8 and newer, whereas the existing "regular python" becomes the "regular runtime" which provides maximum backward compatibility with Python 3.7 and older.

The current C API has multiple issues:

- The Python lifecycle is shorter than the *lifecycle of some operating systems*: how to get the latest Python on an "old" but stable operating system?
- *Python debug build* is currently mostly unusable in practice, making development of C extension harder, especially debugging.

1.2.1 PyPy cpyext is slow

The *PyPy cpyext* is slow because there is no efficient way to write a correct implementation of the *current C API*. For example, *borrowed references* is hard to optimize since the runtime cannot track the lifetime of a borrowed object.

1.2.2 The stable ABI is not usable

The current Python C API produces an ABI which is far from *stable*. The API leaks many implementation details into the ABI.

For example, using the `PyTuple_GET_ITEM()` macro produces machine code which uses an **hardcoded** offset. If the PyObject structure changes, the offset changes, and so the **ABI** changes.

See *Relationship between the C API and the ABI*.

Issues with an unstable ABI:

- **Packaging.** Developers have to publish one binary package per Python version (3.6, 3.7, 3.8, etc.). Upgrading Python 3 to a newer version is very hard because of this issue: each binary package must be distributed in many versions, one per Python version. A stable ABI would allow to build a package for Python 3.6 and use the same binary for Python 3.7, 3.8, 3.9, etc.
- **Experiment issue.** It is not possible to change the base Python structures like PyObject or PyTupleObject to reduce the memory footprint or implement new optimizations. Experiment such changes requires to rebuild C extensions.
- **Debug builds.** Most Linux distributions provide a debug build of Python (`python3-dbg`), but such build has a different ABI and so all C extensions must be recompiled to be usable on such flavor of Python. Being able to use debug builds would ease debugging since the debug builds add many debug checks which are too expensive (CPU/memory) to be enabled by default in a release build.

1.2.3 Performance problem: How can we make Python 2x faster?

"Leaking" implementation details into the ABI prevents many optimizations opportunities. See *Optimization ideas*.

1.2.4 Keep backward compatibility

Existing C extensions will still be supported and will not have to be modified. The *old C API* is not deprecated and there is no plan penalize users of the old C API.

See *Backward compatibility*.

1.2.5 The Big Carrot

Changing the C API means that authors of C extensions have to do something. To justify these changes, we need a big carrot. Examples:

- faster Python if you pick new API? faster PyPy *cpyext*?
- less bugs? Bugs caused by borrow references are hard to debug.
- new features?

1.3 Roadmap for a new Python C API

1.3.1 Roadmap

- Step 1: Identify *Bad C API* and list functions that should be modified or even removed

- Step 2: Add an **opt-in** *new C API* with these cleanups. Test *popular C extensions* to measure how much code is broken. Start to fix these C extensions by making them **forward** compatible. Slowly involve more and more players into the game.
- Step 3: *Remove more functions*. Maybe replace *Py_INCREF() macro* with a function call. Finish to hide all C structures especially `PyObject.ob_refcnt`. Measure the performance. Decide what to do.
- Step 4: if step 3 gone fine and most people are still ok to continue, make the *new C API* as the default in CPython and add an option for **opt-out** to stick with the *old C API*.

1.3.2 Open questions

- Remove or deprecate APIs using borrowed references? If `PyTuple_GetItem()` must be replaced with `PyTuple_GetItemRef()`, how do we provide `PyTuple_GetItemRef()` for Python 3.7 and older? See *Backward compatibility*.

1.3.3 Status

- 2020-04-10: PEP: Modify the C API to hide implementation details sent to python-dev.
- 2019-05-01: Status of the stable API and ABI in Python 3.8, slides of Victor Stinner's lightning talk at the Language Summit (during Pycon US 2019)
- 2019-02-22: [capi-sig] Update on CPython header files reorganization
- 2018-09-04: Creation of CPython fork to experiment a new incompatible C API excluding borrowed references and not access directly structure members.
- 2018-07-29: pythoncapi project created on GitHub
- 2018-06: capi-sig mailing list migrated to Mailman 3
- 2017-12-21: It's an idea. There is an old PEP draft, but no implementation, the PEP has no number and was not accepted yet (nor really proposed).
- 2017-11: Idea proposed on python-dev, [Python-Dev] Make the stable API-ABI usable
- 2017-09: Blog post: A New C API for CPython
- 2017-09: Idea discussed at the CPython sprint at Instagram (California). Liked by all core developers. The expected performance slowdown is likely to be accepted.
- 2017-07-11: [Python-ideas] PEP: Hide implementation details in the C API
- 2017-07: Idea proposed on python-ideas. [Python-ideas] PEP: Hide implementation details in the C API
- 2017-05: Idea proposed at the Python Language Summit, during PyCon US 2017. My "Python performance" slides (PDF). LWN article: Keeping Python competitive.

1.3.4 Players

- CPython: Victor Stinner
- PyPy *cpyext*: Ronan Lamy
- *Cython*: Stefan Behnel

Unknown status:

- RustPython

- MicroPython?
- IronPython?
- Jython?
- Pyjion
- Pyston
- any other?

1.4 Bad C API

The first step to change the Python C API is to define what is a good and a bad C API. The goal is to hide *implementation details*. The *new C API* must not leak implementation details anymore.

The Python C API is just too big. For performance reasons, CPython calls internally directly the implementation of a function instead of using the abstract API. For example, `PyDict_GetItem()` is preferred over `PyObject_GetItem()`. Inside, CPython, such optimization is fine. But exposing so many functions is an issue: CPython has to keep backward compatibility, PyPy has to implement all these functions, etc. Third party C extensions should call abstract functions like `PyObject_GetItem()`.

1.4.1 Borrowed references

Problem caused by borrowed references

A borrowed reference is a pointer which doesn't "hold" a reference. If the object is destroyed, the borrowed reference becomes a dangling pointer: point to freed memory which might be reused by a new object. Borrowed references can lead to bugs and crashes when misused. Recent example of CPython bug: bpo-25750: crash in type_getattro().

Borrowed references are a problem whenever there is no reference to borrow: they assume that a referenced object already exists (and thus have a positive refcount), so that it is just borrowed.

Tagged pointers are an example of this: since there is no concrete `PyObject*` to represent the integer, it cannot easily be manipulated.

PyPy has a similar problem with list strategies: if there is a list containing only integers, it is stored as a compact C array of longs, and the W_IntObject is only created when an item is accessed (most of the time the W_IntObject is optimized away by the JIT, but this is another story).

But for *cpyext*, this is a problem: `PyList_GetItem()` returns a borrowed reference, but there is no any concrete `PyObject*` to return! The current `cpyext` solution is very bad: basically, the first time `PyList_GetItem()` is called, the *whole* list is converted to a list of `PyObject*`, just to have something to return: see cpyext get_list_storage().

See also the *Specialized list for small integers* optimization: same optimization applied to CPython. This optimization is incompatible with borrowed references, since the runtime cannot guess when the temporary object should be destroyed.

If `PyList_GetItem()` returned a strong reference, the `PyObject*` could just be allocated on the fly and destroy it when the user decref it. Basically, by putting borrowed references in the API, we are fixing in advance the data structure to use!

C API using borrowed references

CPython 3.7 has many functions and macros which return or use borrowed references. For example, `PyTuple_GetItem()` returns a borrowed reference, whereas `PyTuple_SetItem()` stores a borrowed reference (store an item into a tuple without increasing the reference counter).

CPython contains `Doc/data/refcounts.dat` (file is edited manually) which documents how functions handle reference count.

See also *functions steal references*.

Functions

- `PyDict_GetItem()`
- `PyDict_GetItemWithError()`
- `PyDict_GetItemString()`
- `PyDict_SetDefault()`
- `PyErr_Occurred()`
- `PyEval_GetBuiltins()`
- **`PyEval_GetFuncName()`: return the internal `const char*` inside a** borrowed reference to a function `__name__`.
- `PyFile_Name()`
- `PyFunction_GetClosure()`
- `PyFunction_GetCode()`
- `PyFunction_GetDefaults()`
- `PyFunction_GetGlobals()`
- `PyFunction_GetModule()`
- `Py_InitModule()`
- `Py_InitModule3()`
- `Py_InitModule4()`
- `PyImport_GetModuleDict()`
- `PyList_GetItem()`
- `PyList_SetItem()`
- `PyMethod_Class()`
- `PyMethod_Function()`
- `PyMethod_Self()`
- `PyModule_GetDict()`
- `PyNumber_Check()`
- `PyObject_Init()`
- `PySys_GetObject()`
- `PySys_GetXOptions()`

- `PyThreadState_GetDict()`
- `PyTuple_GetItem()`
- `PyTuple_SetItem()`
- `PyWeakref_GetObject()`: see https://mail.python.org/pipermail/python-dev/2016-October/146604.html

Macros

- `PyCell_GET()`
- `PyList_GET_ITEM()`
- `PyList_SET_ITEM()`
- `PyMethod_GET_CLASS()`
- `PyMethod_GET_FUNCTION()`
- `PyMethod_GET_SELF()`
- `PySequence_Fast_GET_ITEM()`
- `PyTuple_GET_ITEM()`
- `PyTuple_SET_ITEM()`
- `PyWeakref_GET_OBJECT()`

Border line

- `Py_SETREF()`, `Py_XSETREF()`: the caller has to manually increment the reference counter of the new value
- `N` format of `Py_BuildValue()`?

Py_TYPE() corner case

Technically, `Py_TYPE()` returns a borrowed reference to a `PyTypeObject*`. In practice, for heap types, an instance holds already a strong reference to the type in `PyObject.ob_type`. For static types, instances use a borrowed reference, but static types are never destroyed.

Hugh Fisher summarized:

> It don't think it is worth forcing every C extension module to be rewritten, and incur a performance hit, to eliminate a rare bug from badly written code.

Discussions:

- [Python-Dev] bpo-34595: How to format a type name? (Sept 2018)
- capi-sig: Open questions about borrowed reference. (Sept 2018).

See also *Opaque PyObject structure*.

1.4.2 Duplicated functions

- `PyEval_CallObjectWithKeywords()`: almost duplicate `PyObject_Call()`, except that *args* (tuple of positional arguments) can be `NULL`

- `PyObject_CallObject()`: almost duplicate `PyObject_Call()`, except that *args* (tuple of positional arguments) can be `NULL`

1.4.3 Only keep abstract functions?

Good: abstract functions. Examples:

- `PyObject_GetItem()`, `PySequence_GetItem()`

Bad? implementations for concrete types. Examples:

- `PyObject_GetItem()`, `PySequence_GetItem()`:
  - `PyList_GetItem()`
  - `PyTuple_GetItem()`
  - `PyDict_GetItem()`

Implementations for concrete types don't *have to* be part of the C API. Moreover, using directly them introduce bugs when the caller pass a subtype. For example, PyDict_GetItem() **must not** be used on a dict subtype, since `__getitem__()` be be overridden for good reasons.

1.4.4 Functions kept for backward compatibility

- `PyEval_CallFunction()`: a comment says *"PyEval_CallFunction is exact copy of PyObject_CallFunction. This function is kept for backward compatibility."*
- `PyEval_CallMethod()`: a comment says *"PyEval_CallMethod is exact copy of PyObject_CallMethod. This function is kept for backward compatibility."*

1.4.5 No public C functions if it can't be done in Python

There should not be C APIs that do something that you can't do in Python.

Example: the C buffer protocol, the Python `memoryview` type only expose a subset of `buffer` features.

1.4.6 Array of pointers to Python objects (`PyObject**`)

`PyObject**` must not be exposed: `PyObject** PySequence_Fast_ITEMS(ob)` has to go.

1.4.7 PyDict_GetItem()

The `PyDict_GetItem()` API is one of the most commonly called function but it has multiple flaws:

- it returns a *borrowed reference*
- it ignores any kind of error: it calls `PyErr_Clear()`

The dictionary lookup is surrounded by `PyErr_Fetch()` and `PyErr_Restore()` to ignore any exception.

If hash(key) raises an exception, it clears the exception and just returns `NULL`.

Enjoy the comment from the C code:

```
/* Note that, for historical reasons, PyDict_GetItem() suppresses all errors
 * that may occur (originally dicts supported only string keys, and exceptions
 * weren't possible).  So, while the original intent was that a NULL return
 * meant the key wasn't present, in reality it can mean that, or that an error
 * (suppressed) occurred while computing the key's hash, or that some error
 * (suppressed) occurred when comparing keys in the dict's internal probe
 * sequence.  A nasty example of the latter is when a Python-coded comparison
 * function hits a stack-depth error, which can cause this to return NULL
 * even if the key is present.
 */
```

Functions implemented with `PyDict_GetItem()`:

- `PyDict_GetItemString()`
- `_PyDict_GetItemId()`

There is `PyDict_GetItemWithError()` which doesn't ignore all errors: it only ignores `KeyError` if the key doesn't exist. Sadly, the function still returns a borrowed references.

1.4.8 C structures

Don't leak the structures like `PyObject` or `PyTupleObject` to not access directly fields, to not use fixed offset at the ABI level. Replace macros with functions calls. PyPy already does this in its C API (`cpyext`).

Example of macros:

- `PyCell_GET()`: access directly `PyCellObject.ob_ref`
- `PyList_GET_ITEM()`: access directly `PyListObject.ob_item`
- `PyMethod_GET_FUNCTION()`: access directly `PyMethodObject.im_func`
- `PyMethod_GET_SELF()`: access directly `PyMethodObject.im_self`
- `PySequence_Fast_GET_ITEM()`: use `PyList_GET_ITEM()` or `PyTuple_GET_ITEM()`
- `PyTuple_GET_ITEM()`: access directly `PyTupleObject.ob_item`
- `PyWeakref_GET_OBJECT()`: access directly `PyWeakReference.wr_object`

1.4.9 PyType_Ready() and setting directly PyTypeObject fields

- `PyTypeObject` structure should become opaque
- `PyType_Ready()` should be removed

See *Implement a PyTypeObject in C* for the rationale.

1.4.10 Integer overflow

`PyLong_AsUnsignedLongMask()` ignores integer overflow.

`k` format of `PyArg_ParseTuple()` calls `PyLong_AsUnsignedLongMask()`.

See also `PyLong_AsLongAndOverflow()`.

1.4.11 Functions stealing references

- `PyContext_Exit()`: *ctx*
- `PyContextVar_Reset()`: *token*
- `PyErr_Restore()`: *type*, *value*, *traceback*
- `PySet_Discard()`: *key*, no effect if key not found
- `PyString_ConcatAndDel()`: *newpart*
- `Py_DECREF()`: *o*
- `Py_XDECREF()`: *o*, if *o* is not NULL
- `PyModule_AddObject()`: *o* on success, no change on error!

See also *borrowed references*.

1.4.12 Open questions

Reference counting

Should we do something for reference counting, Py_INCREF and Py_DECREF? Replace them with function calls at least?

See *Change the garbage collector* and *Py_INCREF*.

`PyObject_CallFunction("O")`

Weird `PyObject_CallFunction()` API: bpo-28977. Fix the API or document it?

PyPy requests

Finalizer API

Deprecate finalizer API: PyTypeObject.tp_finalize of PEP 442. Too specific to the CPython garbage collector? Destructors (`__del__()`) are not deterministic in PyPy because of their garbage collector: context manager must be used (ex: `with file:`), or resources must be explicitly released (ex: `file.close()`).

Cython uses `_PyGC_FINALIZED()`, see:

- https://github.com/cython/cython/issues/2721
- https://bugs.python.org/issue35081#msg330045
- Cython generate_dealloc_function().

Compact Unicode API

Deprecate Unicode API introduced by the PEP 393, compact strings, like `PyUnicode_4BYTE_DATA(str_obj)`.

PyArg_ParseTuple

The family of `PyArg_Parse*()` functions like `PyArg_ParseTuple()` support a wide range of argument formats, but some of them leak implementation details:

- `O`: returns a borrowed reference
- `s`: returns a pointer to internal storage

Is it an issue? Should we do something?

1.4.13 For internal use only

Public but not documented and not part of Python.h:

- `PyFrame_FastToLocalsWithError()`
- `PyFrame_FastToLocals()`
- `PyFrame_LocalsToFast()`

These functions should be made really private and removed from the C API.

1.5 A new C API for Python

1.5.1 Design goals

- Reducing the number of *backward incompatible changes* to be able to use the new C API on the maximum number of existing C extensions which use directly the C API.
- Hide most CPython implementation details. The exact list has to be written. One required change is to replace macros with functions calls. The option question remains if it will be possible to replace `Py_INCREF()` and `Py_DECREF()` with function calls without killing performances.
- Reduce the size of the C API to reduce the maintenance burden of *Python implementations other than CPython*: remove functions.
- Reduce the size of the ABI, especially export less symbols.

The *backward compatibility* issue is partially solved by keeping the existing *old C API* available as an opt-in option: see the *Regular runtime*.

1.5.2 Remove functions and macros

Removed functions and macros because they use *borrowed references*:

- `Py_TYPE()`
- `PyTuple_GET_ITEM()`
- `PyTuple_GetItem()`
- `PyTuple_SetItem()`
- `PyTuple_SET_ITEM()`

1.5.3 New functions

XXX the following functions have been added to the current WORK-IN-PROGRESS implementation of the new C API. Maybe they will go away later. It's just a small step to move away from borrowed references. Maybe existing `PyObject_GetItem()` and `PyObject_SetItem()` are already good enough.

- `PyTuple_GetItemRef()`: similar to `PyTuple_GetItem()` but returns a strong reference, rather than a borrowed reference
- `PyList_GetItemRef()`: similar to `PyList_GetItem()` but returns a strong reference, rather than a borrowed reference
- `PyTuple_SetItemRef()`: similar to `PyTuple_SetItem()` but uses a strong reference on the item
- `PySequence_Fast_GetItemRef()`
- `PyStructSequence_SetItemRef()`

XXX private functions:

- `_Py_SET_TYPE()`: see *Implement a PyTypeObject in C*
- `_Py_SET_SIZE()`

If we decide that *Py_TYPE()* should go away, 3 more functions/features are needed:

- `Py_GetType()`: similar to `Py_TYPE()` but returns a strong reference
- `Py_TYPE_IS(ob, type)`: equivalent to `Py_TYPE(ob) == type`
- `%T` format for `PyUnicode_FromFormat()`

1.5.4 Non-goal

- *Cython* and cffi must be preferred to write new C extensions: there is no intent to replace Cython. Moreover, there is no intent to make Cython development harder. Cython will still be able to access directly the full C API which includes implementation details and low-level "private" APIs.

1.5.5 Hide implementation details

See also *Bad C API*.

What are implementation details?

"Implementation details" is not well specified at this point, but maybe hiding implementation can be done incrementally.

The PEP 384 "Defining a Stable ABI" is a very good stable to find the borders between the public C API and implementation details: see *Stable ABI*.

Replace macros with function calls

Replacing macros with functions calls is one part of the practical solution. For example:

```
#define PyList_GET_ITEM(op, i) ((PyListObject *)op)->ob_item[i]
```

would become:

```
#define PyList_GET_ITEM(op, i) PyList_GetItem(op, i)
```

or maybe even:

```
PyObject* PyList_GET_ITEM(PyObject *op, PyObject *i) { return PyList_GetItem(op, i); }
```

Adding a **new** `PyList_GET_ITEM()` **function** would make the ABI larger, whereas the ABI should become smaller.

This change remains backward compatible in term of **C API**. Moreover, using function calls helps to make C extension backward compatible at the **ABI level** as well.

Problem: it's no longer possible to use `Py_TYPE()` and `Py_SIZE()` as l-value:

```
Py_SIZE(obj) = size;
Py_TYPE(obj) = type;
```

XXX in the current implementation, `_Py_SET_SIZE()` and `_Py_SET_TYPE()` macros have been added for such use case. For the type, see also *Implement a PyTypeObject in C*.

Py_INCREF()

The open question remains if it will be possible to replace `Py_INCREF()` and `Py_DECREF()` with function calls without killing performances.

See *Reference counting* and *Change the garbage collector*.

Hide C structures

The most backward incompatible change is to hide fields of C structures, up to PyObject. To final goal will be able to hide `PyObject.ob_refcnt` from the public C API.

C extensions must be modified to use functions to access fields.

In the worst case, there will be no way to access to hidden field from the public C API. For these users, the only option will be to stick at the *old C API* which remains backward compatible and still expose implementation details like C structure fields.

1.6 Python runtimes

To be able to implement *backward compatibility*, the plan is to provide multiple Python runtimes, at least two. An "old" runtime with maximum backward compatibility, and one "new" runtime which includes experimental new changes but requires using the newer C API.

Welcome into the bright world of multiple new cool and compatible Python runtimes!

1.6.1 Regular Python: /usr/bin/python3

- Python compiled in release mode
- This runtime still provides `Py_INCREF()` macro: modify `PyObject.ob_refcnt` at the ABI level.
- Should be fully compatible with *old C API (Python 3.7 C API)*

- Should be fully compatible with Python 3.7 *stable ABI* (it may become incompatible with the Python 3.7 full ABI).

Compared to Python 3.7 regular runtime, this runtime no longer check its arguments for performance reasons. The debug runtime should now be preferred to develop C extensions and to run tests.

Example of Python 3.7 code:

```
int
PyList_Append(PyObject *op, PyObject *newitem)
{
    if (PyList_Check(op) && (newitem != NULL))
        return app1((PyListObject *)op, newitem);
    PyErr_BadInternalCall();
    return -1;
}
```

The `if (PyList_Check(op) && (newitem != NULL))` belongs to the debug runtime and should be removed from the regular Python.

1.6.2 Debug runtime: /usr/bin/python3-dbg

- Compatible with Python 3.7 C **API**.
- Compatible with regular runtime 3.8 **ABI**, but **not compatible** with regular runtime 3.7 ABI.
- CPython compiled with `./configure --with-pydebug`
- Provide `sys.gettotalrefcount()` which allows to check for reference leaks.
- C function calls check most arguments: check type, value range, etc.
- Runtime compiled with C assertion: crash (kill itself with SIGABRT signal) if a C assertion fails (`assert(...);`).
- Use the debug hooks on memory allocators by default, as `PYTHONDEBUG=debug` environment variable: detect memory under- and overflow and misuse of memory allocators.
- Compiled without compiler optimizations (`-Og` or even `-O0`) to be usable with a debugger like `gdb`: python-gdb.py should work perfectly. However, the regular runtime is unusable with gdb since most variables and function arguments are stored in registers, and so gdb fails with the "<optimized out>" message.

For example, the debug runtime can check that the GIL is held by the caller.

See also

- *Remove debug checks*
- Rejected idea: Check index in PyTuple_GET_ITEM/PyTuple_SET_ITEM in debug mode and PyTuple_SET_ITEM could check bounds in debug mode. Issues:
  - Serhiy Storchaka: "I think we can break this only after adding public API for accessing internal storage of a tuple: PyTuple_ITEMS()."
  - Stefan Krah: "I'm using &PyTuple_GET_ITEM(args, 0), so Serhiy's concern is not theoretical."
  - Stefan Behnel: "If this is really just about debugging, then I would suggest to not break existing code at all."

1.6.3 New experimental runtime: python3-exp

- Loading a C extension compiled with Python 3.7 must fail.
- Loading a C extension compiled with the Python C API 3.8 in the backward compatible mode must fail.
- Only C extensions compiled with the **new** Python C API 3.8 can be loaded. You have to **opt-in** for this runtime.
- Not compatible with Python 3.7 API: PyDict_GetItem() is gone, PyDict_GetItemRef() must be used instead.
- Not compatible with Python 3.8 ABI: using Py_INCREF() macro uses `PyObject.ob_refcnt` at the ABI level, whereas this field must **not** be access at the ABI level.
- `Py_GC` header and `PyObject` structure can be very different from the one used by the regular and debug runtimes.

Technically, this experimental runtime can be a opt-in compilation mode of the upstream CPython code base.

See *Optimization ideas*.

1.6.4 Other Python implementations

Last 10 years, CPython has been forked multiple times to attempt different CPython enhancements:

- Unladen Swallow: add a JIT compiler based on LLVM
- Pyston: add a JIT compiler based on LLVM (CPython 2.7 fork)
- Pyjion: add a JIT compiler based on Microsoft CLR
- Gilectomy: remove the Global Interpreter Lock nicknamed "GIL"

Sadly, none is this project has been merged back into CPython. Unladen Swallow lost its funding from Google, Pyston lost its funding from Dropbox, Pyjion is developed in the limited spare time of two Microsoft employees.

Other Python implementations written from scratch:

- PyPy
- RustPython
- MicroPython
- Jython
- IronPython

Since the *C API will be smaller* and the *stable ABI will become more usable*, you can imagine that Python implementations other than CPython will be able to more easily have a **full and up-to-date support** of the latest full C API.

1.6.5 Put your CPython fork here!

Since a *stable ABI* have been designed, if all your C extensions have opt-in for the *new C API*: you are now allowed to fork CPython and experiment your own flavor CPython. Do whatever you want: C extensions only calls your runtime through function calls.

See *Optimization ideas*.

1.7 Old C API

The "current" or "old" C API is the Python 3.7 API which "leaks" implementation details like `PyObject.ob_refcnt` through *Py_INCREF()* macro.

With the new C API, the old C API will remain available thanks to the *regular runtime*, for CPython internals, for specific use cases like *Cython* (for best performances) and *debugging tools*, but also for the long tail of C extensions on PyPI.

See also *Calling conventions*.

1.7.1 What is the Python C API?

- Python objects
  - Protocol, Abstract
  - Types, Classes
- Memory Allocators
- Python initialization and configuration
- Control flow
  - Generator
  - Exception: `PyErr_SetString()`, `PyErr_Clear()`

1.7.2 Current Python C API

- CPython: headers of the Include/ directory
- PyPy *cpyext*: pypy/module/cpyext/ (cpyext/stubs.py)

1.8 Implement a PyTypeObject in C

1.8.1 Old C API

Truncated example of the `PyUnicode_Type` (Python `str`):

```
PyTypeObject PyUnicode_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "str",                          /* tp_name */
    sizeof(PyUnicodeObject),        /* tp_basicsize */
    0,                              /* tp_itemsize */
    /* Slots */
    (destructor)unicode_dealloc,    /* tp_dealloc */
    0,                              /* tp_print */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_reserved */
    unicode_repr,                   /* tp_repr */
    &unicode_as_number,             /* tp_as_number */
    &unicode_as_sequence,           /* tp_as_sequence */
```

(continues on next page)

(continued from previous page)

```
    &unicode_as_mapping,          /* tp_as_mapping */
    (hashfunc) unicode_hash,      /* tp_hash*/
    (...)
    0,                            /* tp_init */
    0,                            /* tp_alloc */
    unicode_new,                  /* tp_new */
    PyObject_Del,                 /* tp_free */
};
```

The type must then be initialized once by calling `PyType_Ready()`:

```
if (PyType_Ready(&PyUnicode_Type) < 0) { /* handle the error */ }
```

This API has an obvious flaw: it rely on the exact implementation of `PyTypeObject`, the developer has to know all fields.

Variant using C99 syntax:

```
static PyTypeObject _abc_data_type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "_abc_data",                        /*tp_name*/
    sizeof(_abc_data),                  /*tp_size*/
    .tp_dealloc = (destructor)abc_data_dealloc,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_alloc = PyType_GenericAlloc,
    .tp_new = abc_data_new,
};
```

1.8.2 PyType_FromSpec()

Python 3.1 introduced a new function:

```
PyObject* PyType_FromSpec(PyType_Spec *spec)
```

Documentation:

> Creates and returns a heap type object from the *spec* passed to the function.

There are two additional **private** functions (excluded from the *Stable ABI*):

```
PyObject* PyType_FromSpecWithBases(PyType_Spec*, PyObject*);
void* PyType_GetSlot(PyTypeObject*, int);
```

`PyType_GetSlot()` expects a slot number which comes from Include/typeslots.inc: see `slotoffsets` array. The file contains the warning:

> Do not renumber the file; these numbers are **part of the stable ABI**.

Some slots have been disabled (bpo-10181):

```
/* Disabled, see #10181 */
#undef Py_bf_getbuffer
#undef Py_bf_releasebuffer
```

Examples of slots:

```
#define Py_nb_add 7

#define Py_tp_alloc 47

#define Py_tp_call 50

#define Py_tp_clear 51

#define Py_tp_doc 56

#define Py_tp_getattr 57
```

Example of type:

```
static PyType_Slot PyCursesPanel_Type_slots[] = {
    {Py_tp_dealloc, PyCursesPanel_Dealloc},
    {Py_tp_methods, PyCursesPanel_Methods},
    {0, 0},
};

static PyType_Spec PyCursesPanel_Type_spec = {
    "_curses_panel.panel",
    sizeof(PyCursesPanelObject),
    0,
    Py_TPFLAGS_DEFAULT,
    PyCursesPanel_Type_slots
};
```

Later initialized by:

```
PyObject *v = PyType_FromSpec(&PyCursesPanel_Type_spec);
if (v == NULL)
    goto fail;
((PyTypeObject *)v)->tp_new = NULL;
_curses_panelstate(m)->PyCursesPanel_Type = v;
```

Missing fields:

- To provide a __dict__ in a defined C type, tp_dict_offset slot must be set, but this slot is missing from the stable ABI.
- Same issue with tp_weaklistoffset

1.8.3 Remove cross-version binary compatibility

See bpo-32388.

1.8.4 PyStructSequence_InitType()

There are two public APIs for structure sequences that are wrappers around `PyType_Ready()` for initializing a `PyTypeObject` in-place:

- `PyStructSequence_InitType()`
- `PyStructSequence_InitType2()`

A third-party developer should prefer `PyStructSequence_NewType()` to create a type object from an array of `PyStructSequence_Desc`.

1.9 Optimization ideas

Once the *new C API* will succeed to hide implementation details, it becomes possible to experiment radical changes in CPython to implement new optimizations.

See *Experimental runtime*.

1.9.1 Remove debug checks

See *Regular runtime*: since a debug runtime will be provided by default, many sanity checks can be removed from the release build.

1.9.2 Change the garbage collector and remove reference counting: unlikely

CPython 3.7 garbage collector (GC) uses "stop-the-world" which is a big issue for realtime applications like games and can be major issue more generally for performance critical applications. There is a desire to use a GC which doesn't need to "stop the world". PyPy succeeded to use an incremental GC.

There are discussing to use a tracing garbage collector for CPython, but this idea remains highly hypothetical since it very likely require deep changes in the C API, which is out of the scope of the *new C API project*. The main risk is to break too many C extensions which would make this idea unusable in practice.

It may be possible to emulate reference counting for the C API. Py_INCREF() and Py_DECREF() would be re-implemented using an hash table: object => reference counter.

Larry Hastings consider to use a tracing garbage collector for *Gilectomy*.

See also *Reference counting*.

1.9.3 Remove the GIL: unlikely

Removing the Global Interpreter Lock (GIL) from CPython, or at least being able to use one GIL per Python interpreter (when using multiple interpreters per process) is an old dream. It means replacing a single big lock with many smaller locks, maybe one lock per Python object.

Jython has not GIL.

Reference counting remains a good and convenient API for C extension. Maybe this design can be kept for the public C API for external C extensions, but CPython internals can be modified to avoid reference counting, like using a tracing garbage collector for example. Once the C API stops leaking implementation details, many new options become possible.

Gilectomy project is CPython 3.6 fork which tries to remove the GIL. In 2017, the project did not succeed yet to scale linearly performances with the number of threads. It seems like **reference counting is a performance killer** for multi-threading.

By the way, using atomic operations to access (increase in `Py_INCREF()`, decrease and test in `Py_DECREF()`) the reference count has been proposed, but experiment showed a slowdown of 20% on single threaded micro-benchmarks.

1.9.4 Tagged pointers: doable

See Wikipedia: Tagged pointer.

Tagged pointers are used by MicroPython to reduce the memory footprint.

Using tagged pointers is a common optimization technique to reduce the boxing/unboxing cost and to reduce the memory consumption.

Currently, it's not possible to implement such optimization, since most of the C API rely on real pointer values for direct access to Python objects.

Note: ARM64 was recently extended its address space to 48 bits, causing issue in LuaJIT: 47 bit address space restriction on ARM64.

Neil Schemenauer PoC (Sept 2018): https://mail.python.org/archives/list/capi-sig@python.org/thread/EGAY55ZWMF2WSEMP7VAZSFZCZ4VARU7L/#EGAY55ZWMF2WSEMP7VAZSFZCZ4VARU7L

1.9.5 Copy-on-Write (CoW): doable

Copy-on-Write (CoW). Instagram is using prefork with Django but has memory usage issues caused by reference counting. Accessing a Python object modifies its reference counter and so copies the page which was created a COW in the forked child process. Python 3.7 added gc.freeze() workaround.

- Replace `Py_ssize_t ob_refcnt;` (integer) with `Py_ssize_t *ob_refcnt;` (pointer to an integer).
- Same change for the GC header?
- Store all reference counters in a separated memory block (or maybe multiple memory blocks)

Expected advantage: smaller memory footprint when using fork() on UNIX which is implemented with Copy-On-Write on physical memory pages.

See also Dismissing Python Garbage Collection at Instagram.

1.9.6 Transactional Memory: highly experimental

PyPy experimented Software Transactional Memory (STM) but the project has been abandoned, PyPy STM.

1.9.7 Specialized list for small integers

If C extensions don't access structure members anymore, it becomes possible to modify the memory layout.

For example, it's possible to design a specialized implementation of `PyListObject` for small integers:

```
typedef struct {
    PyVarObject ob_base;
    int use_small_int;
    PyObject **pyobject_array;
    int32_t *small_int_array;   // <-- new compact C array for integers
    Py_ssize_t allocated;
} PyListObject;

PyObject* PyList_GET_ITEM(PyObject *op, Py_ssize_t index)
{
    PyListObject *list = (PyListObject *)op;
    if (list->use_small_int) {
```

(continues on next page)

(continued from previous page)

```
        int32_t item = list->small_int_array[index];
        /* create a new object at each call */
        return PyLong_FromLong(item);
    }
    else {
        return list->pyobject_array[index];
    }
}
```

Each call to `PyList_GET_ITEM()` of this example creates a new temporary object which leads the memory leak (reference leak). This is one concrete example of issue with borrowed references.

List specialized for numbers is just a example easy to understand to show that it becomes possible to modify PyObject structures. The main benefit of the memory footprint, but the overall on performances is unknown at this point.

1.9.8 O(1) conversion of bytearray to bytes

TODO: find a better method name :-)

Problem: memory copy, memory usage

When a function produces a bytes string but the output length is enough, using a temporary bytearray object is recommended to use the efficient `bytearray += bytes` pattern (bytearray overallocates its internal buffer and so reduce the number of reallocations). Problem: if the result type must be bytes, the bytearray must be converted to bytes... and this operation currently requires to copy the memory. For example, _pyio.FileIO.readall() copies the full content of a binary file and doubles the memory usage.

In Python 3.7, a bytes object always use a single memory block: content follows the object header, whereas a bytearray uses two memory blocks. It's not possible to transfer data from bytearray to bytes to implement a O(1) conversion.

Solution: support bytes using two memory blocks

If the bytes type is modified to also support storing data in a second memory block, it becomes possible to implement O(1) conversion of bytearray to bytes. The bytearray would pass its memory block to the bytes object and then "loose its content" (becomes an empty buffer).

1.9.9 And more!

Insert your new cool idea here!

1.10 Backward compatibility

To reduce the risk of failure, *changing the C API* should be as much as possible compatible with the *old C API (Python 3.7 C API)*. One solution for that is to provide a backward compatible header file and/or library.

1.10.1 Backward compatibility with Python 3.7 and older

For example, if `PyDict_GetItem()` is removed because it returns a borrowed reference, a new function `PyDict_GetItemRef()` which increases the reference counter can be added in the new API. But to make it backward compatible, a macro can be used in Python 3.7 using `PyDict_GetItem()` and `Py_XINCREF()`. Pseudocode:

```
static PyObject*
PyDict_GetItemRef(PyObject *dict, PyObject *key)
{
    PyObject *value = PyDict_GetItem(dict, key);
    Py_XINCREF(value);
    return value;
}
```

Option questions:

- Should the backward compatibility layer be only a header file? Should it be a C library?
- Should we support Python 2.7? Technically, supporting Python 2 shouldn't be hard since the many functions of the C API are the same between Python 2 and Python 3.

1.10.2 Forward compatibility with Python 3.8 and newer

C extensions have to be modified to become compatible with the *new C API*, because of *removed functions* for example.

1.10.3 Open question: support Python 2?

Would it be possible to provide a basic support of Python 2 in the *new C API*?

1.10.4 Open question: how to install the compatibility layers

pip install something?

1.11 Supporting multiple Python versions per operating system release

Supporting multiple minor Python versions, like Python 3.6 and 3.7, requires more work for operating system vendors, like Linux vendors. To reduce the maintenance burden, Linux vendors chose to only support one minor Python version. For example, even if Fedora 28 provides multiple Python binaries (ex: 2.7, 3.5, 3.6 and 3.7), only packages for Python 3.6 are available. Only providing a binary is easy. Providing the full chain of dependencies to get a working Django application is something different.

Issues:

- Each Python minor version introduces subtle minor behaviour changes which requires to sometimes to fix issues in Python modules and applications. This issue is not solved by the new C API.
- Each C extension must be recompiled once per Python minor version.
- The QA team has to test each Python package: having two packages per Python module doubles the work.

Time scale:

- A Python release is supported upstream for 5 years.
- A Fedora release is supported for less than one year.
- Ubuntu LTS releases are supported for 5 years.
- Red Hat Enterprise Linux (RHEL) is supported for 10 years, and customers can subscribe to an extended support up to 15 years.

In 2018, the latest macOS release still only provides Python 2.7 which will reach its end-of-life (EOL) at January 1, 2020 (in less than 2 years).

1.12 C API calling conventions

1.12.1 CPython 3.7 calling conventions

- `METH_NOARGS`: `PyObject* func(PyObject *module)`
- `METH_O`: `PyObject* func(PyObject *module, PyObject *arg)`
- `METH_VARARGS`: `PyObject* func(PyObject *module, PyObject *args)`, *args* type must be `tuple` (subclasses are not supported)
- `METH_VARARGS | METH_KEYWORDS`: `PyObject* func(PyObject *module, PyObject *args, PyObject *kwargs)`, *args* type must be `tuple` (subclasses are not supported) and *kwargs* type must be `dict` (subclasses are not supported), *kwargs* can be NULL
- `METH_FASTCALL`: `PyObject* func(PyObject *module, PyObject *const *args, Py_ssize_t nargs)`, nargs must be greater or equal than `0`
- `METH_FASTCALL | METH_KEYWORDS`: `PyObject* func(PyObject *module, PyObject *const *args, Py_ssize_t nargs, PyObject *kwnames)`, nargs must be greater or equal than `0` and *kwnames* must be a Python `tuple` of Python `str`.

1.12.2 Argument Clinic

CPython contains a tool called "Argument Clinic" to generate the boilerplate to parse arguments and convert the result.

Read the CPython documentation: Argument Clinic How-To.

Example with the builtin `compile()` function:

```
/*[clinic input]
compile as builtin_compile

    source: object
    filename: object(converter="PyUnicode_FSDecoder")
    mode: str
    flags: int = 0
    dont_inherit: bool(accept={int}) = False
    optimize: int = -1

Compile source into a code object that can be executed by exec() or eval().

(...)
[clinic start generated code]*/
```

(continues on next page)

(continued from previous page)

```
static PyObject *
builtin_compile_impl(PyObject *module, PyObject *source, PyObject *filename,
                     const char *mode, int flags, int dont_inherit,
                     int optimize)
/*[clinic end generated code: output=1fa176e33452bb63 input=0ff726f595eb9fcd]*/
{
    /* ... */
}
```

Generated code:

```
#define BUILTIN_COMPILE_METHODDEF    \
    {"compile", (PyCFunction)builtin_compile, METH_FASTCALL|METH_KEYWORDS, builtin_
↪compile__doc__},

static PyObject *
builtin_compile_impl(PyObject *module, PyObject *source, PyObject *filename,
                     const char *mode, int flags, int dont_inherit,
                     int optimize);

static PyObject *
builtin_compile(PyObject *module, PyObject *const *args, Py_ssize_t nargs, PyObject␣
↪*kwnames)
{
    PyObject *return_value = NULL;
    static const char * const _keywords[] = {"source", "filename", "mode", "flags",
↪"dont_inherit", "optimize", NULL};
    static _PyArg_Parser _parser = {"OO&s|iii:compile", _keywords, 0};
    PyObject *source;
    PyObject *filename;
    const char *mode;
    int flags = 0;
    int dont_inherit = 0;
    int optimize = -1;

    if (!_PyArg_ParseStackAndKeywords(args, nargs, kwnames, &_parser,
        &source, PyUnicode_FSDecoder, &filename, &mode, &flags, &dont_inherit, &
↪optimize)) {
        goto exit;
    }
    return_value = builtin_compile_impl(module, source, filename, mode, flags, dont_
↪inherit, optimize);

exit:
    return return_value;
}
```

1.12.3 CPython PyArg_ParseTuple() and Py_BuildValue(), getargs.c

Read the CPython documentation: Parsing arguments and building values.

Example:

```
static PyObject *
array_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
```

(continues on next page)

(continued from previous page)

```
{
    int c;
    PyObject *initial = NULL, *it = NULL;

    if (!PyArg_ParseTuple(args, "C|O:array", &c, &initial))
        return NULL;
    ...
}
```

1.12.4 Summer 2018: 3 PEPs

- PEP 576 – Rationalize Built-in function classes by Mark Shannon
- PEP 579 – Refactoring C functions and methods by by Jeroen Demeyer
- PEP 580 – The C call protocol by Jeroen Demeyer

1.12.5 New calling conventions?

New calling conventions means more work for everybody? Benefit? Avoid boxing/unboxing? Avoid temporary expensive Python objects?

Pass C types like `char`, `int` and `double` rather than `PyObject*`?

Use case: call "C function" from a "C function".

Two entry points? Regular `PyObject*` entry point, but efficient "C" entry point as well?

PyPy wants this, *Cython* would benefit as well.

1.13 Python stable ABI?

1.13.1 Links

- Petr Viktorin's Python Stable ABI improvement notes
- PEP 489 – Multi-phase extension module initialization
  - bpo-1635741: Py_Finalize() doesn't clear all Python objects at exit (convert extensions to multi-phase init PEP 489)
- PEP 620 – Hide implementation details from the C API (Victor Stinner)
  - Move the default C API towards the limited C API
  - Make structures opaque:
    - PyObject: https://bugs.python.org/issue39573
    - PyTypeObject: https://bugs.python.org/issue40170
    - PyFrameObject: https://bugs.python.org/issue40421
    - PyThreadState: https://bugs.python.org/issue39947
    - PyInterpreterState: DONE in Python 3.8!
    - PyGC_Head: DONE in Python 3.9!

  - bpo-40989: Remove _Py_NewReference() and _Py_ForgetReference() from the public C API
  - bpo-41078: Convert PyTuple_GET_ITEM() macro to a static inline function
  - bpo-40601: Hide static types from the limited C API
- PEP 573 – Module State Access from C Extension Methods
  - Python 3.10 has a new `_PyType_GetModuleByDef()` function
  - Python 3.9 added:
    * New PyType_FromModuleAndSpec()
    * New PyType_GetModuleState()
    * New METH_METHOD calling convention flag
    * New PyCMethod function signature
    * New `defining_class` type in Argument Clinic
- PEP 630 – Isolating Extension Modules (Petr Viktorin)
  - bpo-40077: Convert static types to heap types: use PyType_FromSpec()
- bpo-41111: Convert a few stdlib extensions to the limited C API
- HPy project

1.13.2 Relationship between the C API and the ABI

Here is a short explanation. For a longer explanation, read A New C API for CPython (September 2017) by Victor Stinner.

Given the following code:

```
typedef struct {
    PyVarObject ob_base;
    PyObject **ob_item;   // <-- pointer to the array of list items
    Py_ssize_t allocated;
} PyListObject;

#define PyList_GET_ITEM(op, i) ((PyListObject *)op)->ob_item[i]
```

And the following C code:

```
PyObject *item = PyList_GET_ITEM(list, 0);
```

On a 64-bit machine, the machine code of a release build becomes something like:

```
PyObject **items = (PyObject **)(((char*)op) + 24);
PyObject *item = items[0];
```

whereas a debug build uses an offset of **40** instead of **24**, because `PyVarObject` contains additional fields for debugging purpose:

```
PyObject **items = (PyObject **)(((char*)op) + 40);
PyObject *item = items[0];
```

As a consequence, the compiled C extension is incompatible at the ABI level: a C extension has to be build twice, once in release mode and once in debug mode.

To reduce the maintenance burden, *Linux vendors* only provide C extensions compiled in release mode, making the *debug mode* mostly unusable on Linux in practice.

1.13.3 CPython Py_LIMITED_API

- CPython documentation: Stable Application Binary Interface
- Who uses it?
- PEP 384 – Defining a Stable ABI by Martin v. Löwis: implemented in CPython 3.2 (2011)

1.13.4 Check for ABI changes

- https://abi-laboratory.pro/tracker/timeline/python/
- https://bugs.python.org/issue21142
- https://sourceware.org/libabigail/

1.14 Consumers of the Python C API

1.14.1 Popular C extensions using the C API

- numpy
- pandas
- *Cython*
- Pillow
- lxml

1.14.2 Popular modules using Cython

- uvloop

See *Cython*.

1.14.3 Debugging tools

Need to access implementation details:

- faulthandler
- vmprof

1.15 PyPy cpyext module

cpyext is the implementation of the C API in PyPy.

See Ronan Lamy's talk Adventures in compatibility emulating CPython's C API in PyPy (YouTube video) at EuroPython 2018.

1.15.1 Source

See pypy/module/cpyext/ and cpyext/stubs.py.

cpyext has unit tests written in Python.

1.15.2 Performance issue

PyPy with cpyext remains slower than CPython.

XXX how much?

1.15.3 Issue with borrowed references

See *Borrowed references*.

1.15.4 Replace macros with functions

Already done in cpyext.

1.16 Cython

An alternative to using the C API directly is to rewrite a C extension using Cython which generates C code using the C API.

Cython allows you to wrap existing C code, but if you want to use Python objects in C or details which are not exposed at the Python level, you still need the Python C API to work with those.

The code that Cython generates is usually faster than the "obvious" C-API functions, so *not* using the C-API explicitly in Cython has many advantages, such as not relying on CPython version details, simplifying your code, and speeding it up.

Cython is not just a tool to "wrap existing C code". That's only one of its major use cases. Speeding up Python code, and writing C code without writing C code, are just as important.

Whether to use Cython or not depends on your use case. It's not always the obvious choice (otherwise, those other tools would not exist).

XXX write a better rationale why migrating to Cython!

Questions:

- How many popular Python modules use Cython? See *Consumers of the Python C API*.
- How long would it take to rewrite a C extension with `Cython`?

- What is the long-term transition plan to reach the "no C API" goal? At least, CPython will continue to use its own C API internally.
- How to deal with *backward compatibility*?

Small practical issue: `Cython` is not part of the Python 3.7 standard library yet.

See also *cffi* and *Remove the C API*.

1.17 cffi

An alternative to the C API is to rewrite a C extension using cffi.

XXX write a better rationale why migrating to cffi!

Questions:

- Is it easy to distribute binaries generated by cffi to avoid to require C headers and a C compiler? (Windows, macOS, Linux?)
- How many popular Python modules use the C API? See *Consumers of the Python C API*.
- How long would it take to rewrite a C extension with `cffi`?
- What is the long-term transition plan to reach the "no C API" goal? At least, CPython will continue to use its own C API internally.
- How to deal with *backward compatibility*?

Small practical issue: `cffi` is not part of the Python 3.7 standard library yet. Previous attempt to add it, in 2013: [Python-Dev] cffi in stdlib.

See also *Cython* and *Remove the C API*.

1.18 Gilectomy

Gilectomy is Larry Hastings's project to attempt to remove the GIL from CPython. It a fork on CPython which uses lock per object rather than using a Global Interpreter Lock (GIL).

Gilectomy has multiple issues, but the two main issues are:

- The *current C API*: "CPython doesn't use multiple cores and Gilectomy 1.0 is not high performance, which leads him to **consider breaking the C API**".
- Reference counting: "With his complicated buffered-reference-count approach he was able to get his "gilectomized" interpreter to reach performance parity with CPython—except that his interpreter was running on around seven cores to keep up with CPython on one."

For "Gilectomy 2.0", Hastings will be looking at using a tracing garbage collector (GC), rather than the CPython GC that is based on reference counts. Tracing GCs are more multi-core friendly, but he doesn't know anything about them. He also would rather not write his own GC.

- https://github.com/larryhastings/gilectomy
- May 2018: https://lwn.net/Articles/754577/
- PyCon US 2017: https://speakerdeck.com/pycon2017/larry-hastings-the-gilectomy-hows-it-going

1.19 Remove the C API

See Test the next Python to estimate how many C extension modules would be broken by a C API change, like removing a function.

One proposed alternative to a new better C API is no C API at all. The reasoning is that since existing solutions are already available, complete and reliable:

- *Cython*
- *cffi*

We do not need to have one for python itself.

However, this approach has lots of open questions without satisfying answers:

What about the long tail of C extensions on PyPI which still use the C extension? Would it mean a new Python without all these C extensions on PyPI?

Moreover, lots of project do not use those solution, and the C API is part of Python success. For example, there would be no numpy without the C API, and one can look at *Consumers of the Python C API* to see others examples.

Removing it would negatively impact those projects, so this doesn't sound like a workable solution.

1.20 Performance

The C API exposes implementation details for historical reasons (there was no design for the public C API, the public C API is just the private API made public), but also for performance. Macros are designed for best performances, but should be reserved to developers who have a good understanding of CPython internals.

1.20.1 Performance slowdown

Hiding implementation details is likely to make tiny loops slower, since it adds function calls instead of directly accessing the memory.

The performance slowdown is expected to be negligible, but has to be measured once a concrete implementation will be written.

Question: would it be acceptable to have a new better C API if the average slowdown is around 10%? What if the slowdown is up to 25%? Or even 50%?

Right now, the project is too young to guess anything or to bet. Performances will be carefully measured using the Python benchmark suite pyperformance, but only once the design of the new C API is complete.

1.21 Split Include/ directory

Currently, the stable API (Py_LIMITED_API), the private functions (`_Py` prefix), functions that must only be used in CPython core (`Py_BUILD_CORE`) and other functions (regular C API) are all defined in the same file. The 3 API levels:

- Py_BUILD_CORE: API only intended to be used by CPython internals
- Py_LIMITED_API: API for the stable ABI
- other is the *current C API*

In the past, many functions have been added to the wrong API level, just because everything is at the same place. To prevent such mistakes, headers files should be reorganized with clearly separated files.

- https://bugs.python.org/issue35134
- https://bugs.python.org/issue35081

1.22 PyHandle

1.22.1 Email

Copy of Armin Rigo's email (Nov 2018):

https://mail.python.org/pipermail/python-dev/2018-November/155806.html

FWIW, a "handle" is typically something that users of an API store and pass around, and which can be used to do all operations on some object. It is whatever a specific implementation needs to describe references to an object. In the CPython C API, this is `PyObject*`. I think that using "handle" for something more abstract is just going to create confusion.

Also FWIW, my own 2 cents on the topic of changing the C API: let's entirely drop `PyObject *` and instead use more opaque handles—like a `PyHandle` that is defined as a pointer-sized C type but is not actually directly a pointer. The main difference this would make is that the user of the API cannot dereference anything from the opaque handle, nor directly compare handles with each other to learn about object identity. They would work exactly like Windows handles or POSIX file descriptors. These handles would be returned by C API calls, and would need to be closed when no longer used. Several different handles may refer to the same object, which stays alive for at least as long as there are open handles to it. Doing it this way would untangle the notion of objects from their actual implementation. In CPython objects would internally use reference counting, a handle is really just a PyObject pointer in disguise, and closing a handle decreases the reference counter. In PyPy we'd have a global table of "open objects", and a handle would be an index in that table; closing a handle means writing NULL into that table entry. No emulated reference counting needed: we simply use the existing GC to keep alive objects that are referenced from one or more table entries. The cost is limited to a single indirection.

The C API would change a lot, so it's not reasonable to do that in the CPython repo. But it could be a third-party project, attempting to define an API like this and implement it well on top of both CPython and PyPy. IMHO this might be a better idea than just changing the API of functions defined long ago to make them more regular (e.g. stop returning borrowed references); by now this would mostly mean creating more work for the PyPy team to track and adapt to the changes, with no real benefits.

1.22.2 POSIX and Windows API

POSIX uses file descriptors, `int` type:

- open() creates a file descriptor
- dup() duplicates a file descriptor
- close() closes a file descriptor

Windows uses an oquaque `HANDLE` type:

- CreateFile() creates a handle
- DuplicateHandle() duplicates a handle
- CloseHandle() closes a handle

1.23 Deprecate old APIs

CPython is old, the code evolved. Some functions became useless and so should be removed. But backward compatibility matters in Python, so we need a transition period with a deprecation process.

The deprecation can be:

- Emitted at runtime: `DeprecationWarning`
- At the compilation: `Py_DEPRECATED()`
- In the documentation

Functions that should deprecated:

- Unicode functions using `Py_UNICODE` type
- `Py_VA_COPY()`: use directly standard `va_copy()`, see email
- `Py_MEMCPY()`: kept for backwards compatibility

See:

- https://bugs.python.org/issue19569
-

1.24 Python Intermediate representation (IR)

Status: no one is working on the implementation, it's an idea.

Nathaniel Smith's (njs) IR idea for numpy:

- video: Inside NumPy: how it works and how we can make it better (starting at ~26:00).
- slides

Python Compilers Workshop (2016) where the idea was proposed. The notes from the meeting.

1.25 Reorganize Python "runtime"

Starter point: PEP 554 – Multiple Interpreters in the Stdlib.

1.25.1 Goal

The goal is to support running multiple Python interpreters in parallel with one lock per interpreter (no more "Global Interpreter Lock", but one "Interpreter Lock" per interpreter). An interpreter would only be able to run one Python thread holding the interpreter lock at the same time, but multiple Python threads which released the interpreter lock (ex: to call a system call like `read()`) can be run in parallel.

1.25.2 What do we need?

To maximize performances, shared states between interpreters must be minimized. Each share state must be carefully protected by a lock, which prevent to run code in parallel.

1.25.3 Current state of the code (2019-05-24)

During Python 3.7 and 3.8 dev cycle, Eric Snow moved scattered core global variables into a _PyRuntimeState structure which has a single global and shared instance: `_PyRuntime`.

Most functions access directly to `_PyRuntime`, directly or indirectly:

- `PyThreadState *tstate = _PyThreadState_GET();` access implicitly `_PyRuntime`.
- `PyThreadState *tstate = _PyRuntimeState_GetThreadState(&_PyRuntime);` gets access explicitly `_PyRuntime`. Get `runtime->gilstate.tstate_current`.

`_PyRuntimeState` fields:

- `ceval`
- `exitfuncs`, `nexitfuncs`
- `finalizing`
- `gc`
- `gilstate`
- `interpreters`
- `main_thread`
- `open_code_hook`, `open_code_userdata`, `audit_hook_head`
- `pre_initialized`, `core_initialized`, `initialized`
- `preconfig`
- `xidregistry`

1.25.4 TODO

- Move `_PyRuntimeState.gilstate` to `PyInterpreterState`:
  - Remove `_PyRuntimeState_GetThreadState()`
  - Update `_PyThreadState_GET()`
- Move most `_PyRuntimeState` fields into `PyInterpreterState`
- Pass the "context" to private C functions: the context can be `_PyRuntime`, a field of `_PyRuntime`, the Python thread state (`tstate`), etc.

1.25.5 Out of the scope

- Functions of public C API must not be modified at this stage to add new "context" parameters. Only the internal C API can be modified.

1.25.6 Roots

- Get the current Python thread: `_PyRuntimeState_GetThreadState(&_PyRuntime)`. WIP: `gilstate` must move to `PyInterpreterState`
- Get the current interpreter: `tstate->interp`.

1.25.7 Status (2019-05-24)

- `PyInterpreterState` moved to the internal C API
- `_PyRuntimeState` structure and `_PyRuntime` variable created

1.25.8 Links

- https://bugs.python.org/issue36710
- https://bugs.python.org/issue36876
- https://bugs.python.org/issue36877
- https://mail.python.org/archives/list/capi-sig@python.org/thread/RBLU35OUT2KDFCABK32VNOH4UKSKEUWW/
- https://twitter.com/VictorStinner/status/1125887394220269568

1.26 Opaque PyObject structure

A blocker issue for many *optimization ideas* is that the `PyObject` structure fields are exposed in the public C API. Example:

```
PyObject *
PyUnicode_FromObject(PyObject *obj)
{
    ...
    PyErr_Format(PyExc_TypeError,
                 "Can't convert '%.100s' object to str implicitly",
                 Py_TYPE(obj)->tp_name);
    return NULL;
}
```

with:

```
#define Py_TYPE(ob)             (_PyObject_CAST(ob)->ob_type)
#define _PyObject_CAST(op)      ((PyObject*)(op))
```

The issue is that `obj->ob_type` is accessed directly. It prevents to implement *Tagged pointers* for example.

By the way, `Py_TYPE()` returns a *borrowed reference* which is another kind of problem. See *Py_TYPE() corner case*.

In the long term, `PyObject` structure should be opaque. Accessing `ob_refcnt` and `ob_type` fields should always go through functions.

XXX which functions?

XXX how to convert old code to these new functions?

1.27 Statistics on the Python C API

1.27.1 Line numbers

Number of C API line numbers per Python version:

Python	Public	CPython	Internal	Total
2.7	12686 (100%)	0	0	12686
3.6	16011 (100%)	0	0	16011
3.7	16517 (96%)	0	705 (4%)	17222
3.8	13160 (70%)	3417 (18%)	2230 (12%)	18807
3.9	12264 (62%)	4343 (22%)	3066 (16%)	19673
3.10	10305 (52%)	4513 (23%)	5092 (26%)	19910

Comands:

- public: `wc -l Include/*.h`
- cpython: `wc -l Include/cpython/*.h`
- internal: `wc -l Include/internal/*.h`

1.27.2 Symbols

Symbols exported with PyAPI_FUNC() and PyAPI_DATA():

Python	Symbols
2.7	1098
3.6	1460
3.7	1547 (+87)
3.8	1561 (+14)
3.9	1552 (-9)
3.10	1495 (-57)

Command:

```
grep -E 'PyAPI_(FUNC|DATA)' Include/*.h Include/cpython/*.h|wc -l
```

Since Python 3.9, Python is now built with `-fvisibility=hidden` to avoid exporting symbols which are not **explicitly** exported.

The `make smelly` command checks for public symbols of libpython and C extension which are prefixed by `Py` or `_Py`. See `Tools/scripts/smelly.py` script.

CHAPTER 2

Links

- Python C API (this documentation)
- CPython fork implementing Py_NEWCAPI
- pythoncapi GitHub project (this documentation can be found in the `doc/` subdirectory).
- capi-sig mailing list
- py3c (py3c on GitHub): A Python 2/3 compatibility layer for C extensions.

CHAPTER 3

Table of Contents