REPORT ON OpenDreamKit DELIVERABLE D4.13

Refactorisation of SAGE's SPHINX documentation system

JEROEN DEMEYER

Due on	31/08/2017 (M24)
Delivered on	31/08/2018
Lead	Ghent University (UGent)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/87`	

CONTENTS

1. INTRODUCTION

The SAGE computational system, like PYTHON and many other Python projects, uses the SPHINX documentation system. Similar in principle to javadoc or doxygen, it generates structured documentation in various formats (HTML, PDF, ...). The documentation is generated either from `.rst` files or from so-called *docstrings*, that is snippets of documentation interspersed in the code it documents.

One specific challenge is scaling to the sheer size of the SAGE documentation (13k pages just for the reference manual): building it requires a lot of time and memory. In addition – due to those particularly stringent needs and a long legacy – SAGE was (at the time of writing the proposal) using an outdated version of SPHINX, crippled by many layers of customisation and adaptations that had accumulated over the years. This was becoming impossible to maintain, and impeding SAGE's long term sustainability.

Task **T4.4** tackles this situation with, broadly speaking, two main goals:

(1) a critically needed deep refactorisation to foster sustainability by outsourcing back to SPHINX all generic aspects (parallel compilation, index generation, etc);
(2) optimizing resource usage for the documentation build.

Although a lot has already been achieved with more than twenty two contributions to SAGE or upstream (see `https://github.com/OpenDreamKit/OpenDreamKit/issues/87` for an up to date list), progress has been significantly slower than originally planned for. In large part, this is because many upstream projects are involved: SPHINX, DOCUTILS, PYGMENTS, CYTHON, and PYTHON. We are thus dependent on their own independent schedule whenever an upstream contribution is needed; an highlight of such sensitive contribution is PEP 580, a Python Enhancement Proposal for the core language implementation (see § 2.2). Such a situation was foreseen in our Risk Management Plan and its impact on the overall project is negligible: on the one hand, no other task depends on **T4.4**; on the other hand, it was easy to shuffle around some of our internal work plan, and reserve PMs for this task during OpenDreamKit's final year, at Ghent University and Université Paris-Sud. Also, knowing that this was a long term endeavour and that the timeline was not critical, we decided on several occasions to invest in long term solutions that would benefit not only SAGE but the community as a whole (see § 2.2).

We report here on the current achievements, and detail the work plan for the final year. Formally speaking, this deliverable report concerns solely the first goal of **T4.4**; however, as the its goals are closely interrelated, reporting on both is required to explain the big picture.

2. REMOVING SAGE CUSTOMISATIONS

SAGE uses a highly customised version of SPHINX. The word "customised" can mean any of the following: an actual patch to the source code, a monkey-patch (a patch at runtime), a fork of some plug-in, or just a highly specialised configuration.

One of the goals of this deliverable is to remove these customisations, allowing SAGE to use a standard SPHINX. This can be achieved in two ways: either contributing back the customization to upstream SPHINX to make it a standard feature, or changing SAGE (or some other project like CYTHON or PYTHON) to remove the need for the customisation.

This has multiple advantages: first of all, having less custom code makes SAGE more maintainable. Second the customization contributed upstream benefit everybody and not just SAGE. Last, but not least, it helps packaging SAGE (see **T3.3**), as packagers need not any produce and maintain a customized SPHINX package in addition to the standard one.

2.1. General clean up and minor fixes

We did quite a lot of minor clean up. This is hard to list in detail. In particular, various fixes were made to the build system for the documentation, including using a more standard SPHINX configuration, as well as removing several irrelevant customisations to `autodoc`, the extension for generating documentation from docstrings. We also made building the SAGE documentation a lot more reliable: in the past, it often failed to build incrementally after an upgrade, which required a resource expensive full rebuild. This was a burden for developers but also a barrier for efficient continuous integration (see D3.8: "Continuous integration platform for multi-platform build/test.").

2.2. Support for Cython functions

SAGE uses a combination of Python and Cython modules. Cython is a programming language which mostly uses the Python syntax, but which is compiled to C instead of interpreted like Python. By default, functions in Cython become Python *built-in* functions. These are a special kind of optimised function, implemented in C.

Unfortunately, built-in functions do not work well with SPHINX, as they do not have all the needed introspection capabilities. In particular, SPHINX cannot determine the input arguments of such functions. Since the arguments are an important part of the documentation of a function, this is a big problem. SAGE and CYTHON have various hacks to make it work anyway. These are quite ugly, so it would not make much sense to support those in SPHINX.

Fixing this properly requires adding support in PYTHON for fast user-defined function classes. For user-defined function classes, CYTHON can add whatever attributes are needed to support SPHINX and other use cases that require deep introspection of functions. The problem is that such user-defined function classes are necessarily slower than built-in functions. This is not acceptable for SAGE where speed is very important. In June 2018, we submitted PEP 580 (see Appendix A), a Python Enhancement Proposal to fix this. At the May 2018 OpenDreamKit developers workshop in Cernay (see D2.11: "Community building: Impact of development workshops, dissemination and training activities, year 2 and 3"), we already discussed an earlier variant of the PEP with the core CYTHON developers and they were very positive about it. Both before and after submitting the PEP, we discussed it online with the Python community, leading to substantial improvements. Unfortunately, the process of getting a PEP accepted is quite slow: at the time of writing this report, the PEP has not been accepted (nor rejected).

If PEP 580 is accepted, CYTHON can be updated to produce functions which are equally fast as built-in functions and which support all documentation features that standard Python functions support.

2.3. Parallel build of the reference manual as a single document

The SAGE reference manual is currently not built as one monolithic document, but in several pieces: one for each mathematical topic such as combinatorics or matrices. The original motivations were twofold: it makes it easy to build the various pieces in parallel and it lowers the memory requirements. Unfortunately, these pieces are not completely independent: we still want to end up with one consistent document, so things like the index and references need to merged. This required a custom SAGE `multidocs` extension for SPHINX.

In addition, the natural granularity of the pieces at the semantic level never quite matched a proper granularity for parallelisation; the existence of a few long chapters like on combinatorics led to annoyingly long compilation.

Finally the parallelisation itself required a bespoke parallel docbuilder, as SPHINX did not have parallel build features at the time. This is no more the case: SPHINX now has native parallel support, even within single documents.

Hence it is desirable to decouple parallelisation from documentation splitting and work on memory requirements, to be able to build again the reference manual as a single document.

At the OpenDreamKit organized Sage Days 77 workshop in April 2016 (see D2.11), we had a coding sprint with with Robert Lehmann, one of the main SPHINX developers, where we implemented a proof-of-concept showing the feasibility of this plan. Once the memory requirements will be tackled (see § 4) it will be little work to finalize that plan.

2.4. Custom syntax highlighting

SAGE currently ships with a slightly patched version of PYGMENTS in order to support highlighting of the `sage:` prompt in addition to the standard Python >>> prompts. This is a burden for packagers. We need to investigate whether we can configure PYGMENTS without

patching it, or push support for SAGE upstream to PYGMENTS. Either way, this should be quick and straightforward.

3. UPGRADING SPHINX

In tandem with removing customisations, it is obvious that we should use the most recent release of SPHINX. This is not as trivial as it sounds as we started with a very old version of SPHINX in which the custom code integrated tightly with SPHINX. Thanks to an incremental upgrade in a series of 6 tickets, SAGE is now using a reasonably recent version of SPHINX (1.7.6 as of now). Allowing for smoother future upgrades is yet another reason for removing the customisations mentioned in the previous sections.

4. HIGH RESOURCE USAGE

Building the full SAGE documentation takes a lot of resources: it needs about 3 gigabytes of memory and about 1.5 hours on an averge personal computer to build the documentation from scratch. This makes it the most resource-intensive part of building the complete SAGE computational system. Because of this, SAGE users who wish to build from source are often suggested to skip building the documentation. It is also a problem for automatic builds in continuous integration setups, which typically only have limited resources.

During the aforementioned Sage Days 77 coding sprint with a SPHINX developer, we have investigated where the high memory usage comes from. It turns out that it is a combination of factors and that there is not a single obvious cause. Some of it is caching of data which is no longer needed, which should be easy to fix. Some of it is data structures using basic Python primitives such as a `dict` instead of more specialized (and space-efficient) structures.

The SAGE customizations to SPHINX (in particular what is mentioned in § 2.3 and § 2.3) may cloud this analysis somewhat as they might make things better or worse. Therefore, it would make most sense to tackle this problem after removing or at least reducing those customizations.

Fixing this will need to be tackled at the level of DOCUTILS or SPHINX. Therefore, this will benefit all projects using SPHINX.

5. RELATED WORK

It is beneficial for SAGE packages to use the same theme and configuration for their documentation as SAGE. We simplified the process by implementing the `sage-package` utility (which contains among other things a copy of the Sphinx configuration for SAGE). Its usage is illustrated in the `sage_sample` template for SAGE packages. We are investigating how to avoid the aforementioned copy. In any case, it is our goal to use as much as possible a standard SPHINX setup, so this should be limited to some basic configuration and style options.

In D4.7: "Full featured JUPYTER interface for GAP, PARI/GP, Singular", we report on ThebeLab, a JupyterLab based JavaScript library that turns static code examples in an HTML page into live code cells that the user can edit and execute. We configured SPHINX to enable ThebeLab in the documentation of SAGE and SAGE packages.

Finally, we mention two solutions to turn SPHINX documentation into JUPYTER notebooks: OpenDreamKit members, with some external contributions, developed early on a stand-alone `rst2ipynb` program for converting a single `.rst` file to a Jupyter notebook. More recently a SPHINX plugin `sphinxcontrib-jupyter` was developed by the community, with some OpenDreamKit contributions. This plugin integrates directly with SPHINX, operating on a collection of documents instead of a single document. As such, it supports crosslinks, which makes it a lot more suitable for converting large documents such as books (see D2.9: "Demonstrator: interactive books on Linear Algebra and Nonlinear Processes in Biology") or the Sage

manuals. We are currently in the process of adding support for this in the Sage docbuilder, see `https://trac.sagemath.org/ticket/25909`

APPENDIX A. PEP: 580

Title: The C call protocol
Author: Jeroen Demeyer <`J.Demeyer@UGent.be`>
Status: Draft
Type: Standards Track
URL: `https://www.python.org/dev/peps/pep-0580/`
Content-Type: text/x-rst
Created: 14-Jun-2018
Python-Version: 3.8
Post-History: 20-Jun-2018, 22-Jun-2018, 16-Jul-2018

A.1. Abstract

A new "C call" protocol is proposed. It is meant for classes representing functions or methods which need to implement fast calling. The goal is to generalize existing optimizations for built-in functions to arbitrary extension types.

In the reference implementation, this new protocol is used for the existing classes `builtin_function_or_m` and `method_descriptor`. However, in the future, more classes may implement it.

NOTE: This PEP deals only with the Python/C API, it does not affect the Python language or standard library.

A.2. Motivation

Currently, the Python bytecode interpreter has various optimizations for calling instances of `builtin_function_or_method`, `method_descriptor`, `method` and `function`. However, none of these classes is subclassable. Therefore, these optimizations are not available to user-defined extension types.

If this PEP is implemented, then the checks for `builtin_function_or_method` and `method_descriptor` could be replaced by simply checking for and using the C call protocol. This simplifies existing code.

We also design the C call protocol such that it can easily be extended with new features in the future.

For more background and motivation, see PEP 579.

A.3. Basic idea

Currently, CPython has multiple optimizations for fast calling for a few specific function classes. Calling instances of these classes using a plain `tp_call` is slower than using the optimizations. The basic idea of this PEP is to allow user-defined extension types (not Python classes) to use these optimizations also, both as caller and as callee.

The existing class `builtin_function_or_method` and a few others use a `PyMethodDef` structure for describing the underlying C function and its signature. The first concrete change is that this is replaced by a new structure `PyCCallDef`. This stores some of the same information as a `PyMethodDef`, but with one important addition: the "parent" of the function (the class or module where it is defined). Note that `PyMethodDef` arrays are still used to construct functions/methods but no longer for calling them.

Second, we want that every class can use such a `PyCCallDef` for optimizing calls, so the `PyTypeObject` structure gains a `tp_ccalloffset` field giving an offset to a `PyCCallDef` $*$ in the object structure and a flag `Py_TPFLAGS_HAVE_CCALL` indicating that `tp_ccalloffset` is valid.

Third, since we want to deal efficiently with unbound and bound methods too (as opposed to only plain functions), we need to handle `__self__` too: after the `PyCCallDef *` in the object structure, there is a `PyObject *self` field. These two fields together are referred to as a `PyCCallRoot` structure.

The new protocol for efficiently calling objects using these new structures is called the "C call protocol".

A.4. New data structures

The `PyTypeObject` structure gains a new field `Py_ssize_t tp_ccalloffset` and a new flag `Py_TPFLAGS_HAVE_CCALL`. If this flag is set, then `tp_ccalloffset` is assumed to be a valid offset inside the object structure (similar to `tp_weaklistoffset`). It must be a strictly positive integer. At that offset, a `PyCCallRoot` structure appears:

```
typedef struct {
    PyCCallDef *cr_ccall;
    PyObject   *cr_self;     /* __self__ argument for methods */
} PyCCallRoot;
```

The `PyCCallDef` structure contains everything needed to describe how the function can be called:

```
typedef struct {
    uint32_t  cc_flags;
    PyCFunc   cc_func;    /* C function to call */
    PyObject *cc_parent;  /* class or module */
} PyCCallDef;
```

The reason for putting `__self__` outside of `PyCCallDef` is that `PyCCallDef` is not meant to be changed after creating the function. A single `PyCCallDef` can be shared by an unbound method and multiple bound methods. This wouldn't work if we would put `__self__` inside that structure.

NOTE: unlike `tp_dictoffset` we do not allow negative numbers for `tp_ccalloffset` to mean counting from the end. There does not seem to be a use case for it and it would only complicate the implementation.

A.4.1. *Parent.* The `cc_parent` field (accessed for example by a `__parent__` or `__objclass__` descriptor from Python code) can be any Python object. For methods of extension types, this is set to the class. For functions of modules, this is set to the module. However, custom classes are free to set `cc_parent` to whatever they want, it can also be `NULL`. It is only used by the C call protocol if the `CCALL_OBJCLASS` flag is set.

The parent serves multiple purposes: for methods of extension types (more precisely, when the flag `CCALL_OBJCLASS` is set), it is used for type checks like the following:

```
>>> list.append({}, "x")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor 'append' requires a 'list' object but received a 'd
```

PEP 573 specifies that every function should have access to the module in which it is defined. It is recommended to use the parent for this, which works directly for functions of a module. For methods, this works indirectly through the class, assuming that the class has a pointer to the module.

The parent would also typically be used to implement `__qualname__`. The new C API function `PyCCall_GenericGetQualname()` does exactly that.

NOTE: for functions of modules, the parent is exactly the same as `__self__`. However, using `__self__` for the module is a quirk of the current implementation: in the future, we want to allow

functions which use `__self__` in the normal way, for implementing methods. Such functions can still use `cc_parent` instead to refer to the module.

A.4.2. *Using tp_print.* We propose to replace the existing unused field `tp_print` by `tp_ccalloffset`. Since `Py_TPFLAGS_HAVE_CCALL` would *not* be added to `Py_TPFLAGS_DEFAULT`, this ensures full backwards compatibility for existing extension modules setting `tp_print`. It also means that we can require that `tp_ccalloffset` is a valid offset when `Py_TPFLAGS_HAVE_CCALL` is specified: we do not need to check `tp_ccalloffset != 0`. In future Python versions, we may decide that `tp_print` becomes `tp_ccalloffset` unconditionally, drop the `Py_TPFLAGS_HAVE_CCALL` flag and instead check for `tp_ccalloffset != 0`.

A.5. The C call protocol

We say that a class implements the C call protocol if it has the `Py_TPFLAGS_HAVE_CCALL` flag set (as explained above, it must then set `tp_ccalloffset > 0`). Such a class must implement `__call__` as described in this subsection (in practice, this just means setting `tp_call` to `PyCCall_Call`).

The `cc_func` field is a C function pointer, which plays the same role as the existing `ml_meth` field of `PyMethodDef`. Its precise signature depends on flags. Below are the possible values for `cc_flags & CCALL_SIGNATURE` together with the arguments that the C function takes. The return value is always `PyObject *`. The following are analogous to the existing `PyMethodDef` signature flags:

- `CCALL_VARARGS`: `cc_func(PyObject *self, PyObject *args)`
- `CCALL_VARARGS | CCALL_KEYWORDS`: `cc_func(PyObject *self, PyObject *args, PyObject *kwds)`
- `CCALL_FASTCALL`: `cc_func(PyObject *self, PyObject *const *args, Py_ssize_t nargs)`
- `CCALL_FASTCALL | CCALL_KEYWORDS`: `cc_func(PyObject *self, PyObject *const *args, Py_ssize_t nargs, PyObject *kwnames)` (`kwnames` is either `NULL` or a non-empty tuple of keyword names)
- `CCALL_NOARGS`: `cc_func(PyObject *self, PyObject *unused)` (second argument is always `NULL`)
- `CCALL_O`: `cc_func(PyObject *self, PyObject *arg)`

The flag `CCALL_FUNCARG` may be combined with any of these. If so, the C function takes an additional argument as first argument before `self`. This argument is used to pass the function object (the `self` in `__call__` but see NOTE below). For example, we have the following signature:

- `CCALL_FUNCARG | CCALL_VARARGS`: `cc_func(PyObject *func, PyObject *self, PyObject *args)`

One exception is `CCALL_FUNCARG | CCALL_NOARGS`: the `unused` argument is dropped, so the signature becomes

- `CCALL_FUNCARG | CCALL_NOARGS`: `cc_func(PyObject *func, PyObject *self)`

NOTE: in the case of bound methods, it is currently unspecified whether the "function object" in the paragraph above refers to the bound method or the original function (which is wrapped by the bound method). In the reference implementation, the bound method is passed. In the future, this may change to the wrapped function. Despite this ambiguity, the implementation of bound methods guarantees that `PyCCall_CCALLDEF(func)` points to the `PyCCallDef` of the original function.

NOTE: unlike the existing `METH_...` flags, the `CCALL_...` constants do not necessarily represent single bits. So checking `(cc_flags & CCALL_VARARGS) == 0` is not a valid

way for checking the signature. There are also no guarantees of binary compatibility between Python versions for these flags.

A.5.1. *Checking __objclass.* If the `CCALL_OBJCLASS` flag is set and if `cr_self` is NULL (this is the case for unbound methods of extension types), then a type check is done: the function must be called with at least one positional argument and the first (typically called `self`) must be an instance of `cc_parent` (which must be a class). If not, a `TypeError` is raised.

A.5.2. *Self slicing.* If `cr_self` is not NULL or if the flag `CCALL_SELFARG` is not set in `cc_flags`, then the argument passed as `self` is simply `cr_self`.

If `cr_self` is NULL and the flag `CCALL_SELFARG` is set, then the first positional argument is removed from `args` and instead passed as first argument to the C function. Effectively, the first positional argument is treated as `__self__`. If there are no positional arguments, `TypeError` is raised.

This process is called "self slicing" and a function is said to have self slicing if `cr_self` is NULL and `CCALL_SELFARG` is set.

Note that a `CCALL_NOARGS` function with self slicing effectively has one argument, namely `self`. Analogously, a `CCALL_O` function with self slicing has two arguments.

A.5.3. *Descriptor behavior.* Classes supporting the C call protocol must implement the descriptor protocol in a specific way. This is required for an efficient implementation of bound methods: it allows sharing the `PyCCallDef` structure between bound and unbound methods. It is also needed for a correct implementation of `_PyObject_GetMethod` which is used by the `LOAD_METHOD`/`CALL_METHOD` optimization. First of all, if `func` supports the C call protocol, then `func.__set__` must not be implemented.

Second, `func.__get__` must behave as follows:

- If `cr_self` is not NULL, then `__get__` must be a no-op in the sense that `func.__get__(obj, cls)(*args, **kwds)` behaves exactly the same as `func(*args, **kwds)`. It is also allowed for `__get__` to be not implemented at all.
- If `cr_self` is NULL, then `func.__get__(obj, cls)(*args, **kwds)` (with `obj` not None) must be equivalent to `func(obj, *args, **kwds)`. In particular, `__get__` must be implemented in this case. Note that this is unrelated to self slicing: `obj` may be passed as `self` argument to the C function or it may be the first positional argument.
- If `cr_self` is NULL, then `func.__get__(None, cls)(*args, **kwds)` must be equivalent to `func(*args, **kwds)`.

There are no restrictions on the object `func.__get__(obj, cls)`. The latter is not required to implement the C call protocol for example. It only specifies what `func.__get__(obj, cls).__call__` does.

For classes that do not care about `__self__` and `__get__` at all, the easiest solution is to assign `cr_self = Py_None` (or any other non-NULL value).

The C call protocol requires that the function has a `__name__` attribute which is of type `str` (not a subclass).

Furthermore, this must be idempotent in the sense that getting the `__name__` attribute twice in a row must return exactly the same Python object. This implies that it cannot be a temporary object, it must be stored somewhere. This is required because `PyEval_GetFuncName` uses a borrowed reference to the `__name__` attribute.

A.5.4. *Generic API functions.* This subsection lists the new public API functions or macros dealing with the C call protocol.

- `int PyCCall_Check(PyObject *op)`: return true if `op` implements the C call protocol.

All the functions and macros below apply to any instance supporting the C call protocol. In other words, `PyCCall_Check(func)` must be true.

- `PyObject *PyCCall_Call(PyObject *func, PyObject *args, PyObject *kwds)`: call `func` with positional arguments `args` and keyword arguments `kwds` (`kwds` may be NULL). This function is meant to be put in the `tp_call` slot.
- `PyObject *PyCCall_FASTCALL(PyObject *func, PyObject *const *args, Py_ssize_t nargs, PyObject *kwds)`: call `func` with `nargs` positional arguments given by `args[0]`, ..., `args[nargs-1]`. The parameter `kwds` can be NULL (no keyword arguments), a dict with `name:value` items or a tuple with keyword names. In the latter case, the keyword values are stored in the `args` array, starting at `args[nargs]`.

Macros to access the `PyCCallRoot` and `PyCCallDef` structures:

- `PyCCallRoot *PyCCall_CCALLROOT(PyObject *func)`: pointer to the `PyCCallRoot` structure inside `func`.
- `PyCCallDef *PyCCall_CCALLDEF(PyObject *func)`: shorthand for `PyCCall_CCALLRO`
- `PyCCallDef *PyCCall_FLAGS(PyObject *func)`: shorthand for `PyCCall_CCALLROOT(`
- `PyObject *PyCCall_SELF(PyOject *func)`: shorthand for `PyCCall_CCALLROOT(func`

Generic getters, meant to be put into the `tp_getset` array:

- `PyObject *PyCCall_GenericGetParent(PyObject *func, void *closure)`: return `cc_parent`. Raise `AttributeError` if `cc_parent` is NULL.
- `PyObject *PyCCall_GenericGetQualname(PyObject *func, void *closure)`: return a string suitable for using as `__qualname__`. This uses the `__qualname__` of `cc_parent` if possible. It also uses the `__name__` attribute.

A.5.5. *Profiling.* The profiling events `c_call`, `c_return` and `c_exception` are only generated when calling actual instances of `builtin_function_or_method` or `method_descriptor`. This is done for simplicity and also for backwards compatibility (such that the profile function does not receive objects that it does not recognize). In a future PEP, we may extend C-level profiling to arbitrary classes implementing the C call protocol.

A.6. **Changes to built-in functions and methods**

The reference implementation of this PEP changes the existing classes `builtin_function_or_method` and `method_descriptor` to use the C call protocol. In fact, those two classes are almost merged: the implementation becomes very similar, but they remain separate classes (mostly for backwards compatibility). The `PyCCallDef` structure is simply stored as part of the object structure. Both classes use `PyCFunctionObject` as object structure. This is the new layout for both classes:

```
typedef struct {
    PyObject_HEAD
    PyCCallDef  *m_ccall;
    PyObject    *m_self;          /* Passed as 'self' arg to the C functio
    PyCCallDef   _ccalldef;       /* Storage for m_ccall */
    PyObject    *m_name;          /* __name__; str object (not NULL) */
    PyObject    *m_module;        /* __module__; can be anything */
    const char  *m_doc;           /* __text_signature__ and __doc__ */
```

```
    PyObject    *m_weakreflist;  /* List of weak references */
} PyCFunctionObject;
```

For functions of a module and for unbound methods of extension types, `m_ccall` points to the `_ccalldef` field. For bound methods, `m_ccall` points to the `PyCCallDef` of the unbound method.

NOTE: the new layout of `method_descriptor` changes it such that it no longer starts with `PyDescr_COMMON`. This is purely an implementation detail and it should cause few (if any) compatibility problems.

A.6.1. *C API functions.* The following function is added (also to the stable ABI[1]):

- `PyObject * PyCFunction_ClsNew(PyTypeObject *cls, PyMethodDef *ml, PyObject *self, PyObject *module, PyObject *parent)`: create a new object with object structure `PyCFunctionObject` and class `cls`. The entries of the `PyMethodDef` structure are used to construct the new object, but the pointer to the `PyMethodDef` structure is not stored. The flags for the C call protocol are automatically determined in terms of `ml->ml_flags`, `self` and `parent`.

The existing functions `PyCFunction_New`, `PyCFunction_NewEx` and `PyDescr_NewMethod` are implemented in terms of `PyCFunction_ClsNew`.

The undocumented functions `PyCFunction_GetFlags` and `PyCFunction_GET_FLAGS` are removed because it would be non-trivial to support them in a backwards-compatible way.

A.7. **Inheritance**

Extension types inherit the type flag `Py_TPFLAGS_HAVE_CCALL` and the value `tp_ccalloffset` from the base class, provided that they implement `tp_call` and `tp_descr_get` the same way as the base class. Heap types never inherit the C call protocol because that would not be safe (heap types can be changed dynamically).

A.8. **Performance**

This PEP should not impact the performance of existing code (in the positive or negative sense). It is meant to allow efficient new code to be written, not to make existing code faster.

Here are a few pointers to the `python-dev` mailing list where performance improvements are discussed:

- `https://mail.python.org/pipermail/python-dev/2018-July/154571.html`
- `https://mail.python.org/pipermail/python-dev/2018-July/154740.html`
- `https://mail.python.org/pipermail/python-dev/2018-July/154775.html`

A.9. **Stable ABI**

None of the functions, structures or constants dealing with the C call protocol are added to the stable ABI[2].

There are two reasons for this: first of all, the most useful feature of the C call protocol is probably the `METH_FASTCALL` calling convention. Given that this is not even part of the public API (see also PEP 579, issue 6), it would be strange to add anything else from the C call protocol to the stable ABI.

Second, we want the C call protocol to be extensible in the future. By not adding anything to the stable ABI, we are free to do that without restrictions.

[1] Löwis, PEP 384 – Defining a Stable ABI, `https://www.python.org/dev/peps/pep-0384/`

[2] Löwis, PEP 384 – Defining a Stable ABI, `https://www.python.org/dev/peps/pep-0384/`

A.10. **Backwards compatibility**

There is no difference at all for the Python interface, nor for the documented C API (in the sense that all functions remain supported with the same functionality).

The removed function `PyCFunction_GetFlags`, is officially part of the stable ABI[3]. However, this is probably an oversight: first of all, it is not even documented. Second, the flag `METH_FASTCALL` is not part of the stable ABI but it is very common (because of Argument Clinic). So, if one cannot support `METH_FASTCALL`, it is hard to imagine a use case for `PyCFunction_GetFlags`. The fact that `PyCFunction_GET_FLAGS` and `PyCFunction_GetFlags` are not used at all by CPython outside of `Objects/call.c` further shows that these functions are not particularly useful.

Concluding: the only potential breakage is with C code which accesses the internals of `PyCFunctionObject` and `PyMethodDescrObject`. We expect very few problems because of this.

A.11. **Rationale**

A.11.1. *Why is this better than PEP 575?* One of the major complaints of PEP 575 was that is was coupling functionality (the calling and introspection protocol) with the class hierarchy: a class could only benefit from the new features if it was a subclass of `base_function`. It may be difficult for existing classes to do that because they may have other constraints on the layout of the C object structure, coming from an existing base class or implementation details. For example, `functools.lru_cache` cannot implement PEP 575 as-is.

It also complicated the implementation precisely because changes were needed both in the implementation details and in the class hierarchy.

The current PEP does not have these problems.

A.11.2. *Why store the function pointer in the instance?* The actual information needed for calling an object is stored in the instance (in the `PyCCallDef` structure) instead of the class. This is different from the `tp_call` slot or earlier attempts at implementing a `tp_fastcall` slot[4].

The main use case is built-in functions and methods. For those, the C function to be called does depend on the instance.

Note that the current protocol makes it easy to support the case where the same C function is called for all instances: just use a single static `PyCCallDef` structure for every instance.

A.11.3. *Why CCALL_OBJCLASS?.* The flag `CCALL_OBJCLASS` is meant to support various cases where the class of a `self` argument must be checked, such as:

```
>>> list.append({}, None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: append() requires a 'list' object but received a 'dict'

>>> list.__len__({})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__len__' requires a 'list' object but received a '

>>> float.__dict__["fromhex"](list, "0xff")
Traceback (most recent call last):
```

[3] Löwis, PEP 384 – Defining a Stable ABI, `https://www.python.org/dev/peps/pep-0384/`

[4] Add tp_fastcall to PyTypeObject: support FASTCALL calling convention for all callable objects, `https://bugs.python.org/issue29259`

```
  File "<stdin>", line 1, in <module>
TypeError: descriptor 'fromhex' for type 'float' doesn't apply to type 'l
```

In the reference implementation, only the first of these uses the new code. The other examples show that these kind of checks appear in multiple places, so it makes sense to add generic support for them.

A.11.4. *Why CCALL_SELFARG?.* The flag `CCALL_SELFARG` and the concept of self slicing are needed to support methods: the C function should not care whether it is called as unbound method or as bound method. In both cases, there should be a `self` argument and this is simply the first positional argument of an unbound method call.

For example, `list.append` is a `METH_O` method. Both the calls `list.append([], 42)` and `[].append(42)` should translate to the C call `list_append([], 42)`.

Thanks to the proposed C call protocol, we can support this in such a way that both the unbound and the bound method share a `PyCCallDef` structure (with the `CCALL_SELFARG` flag set).

Concluding, `CCALL_SELFARG` has two advantages: there is no extra layer of indirection for calling and constructing bound methods does not require setting up a `PyCCallDef` structure.

A.11.5. *Replacing tp_print.* We repurpose `tp_print` as `tp_ccalloffset` because this makes it easier for external projects to backport the C call protocol to earlier Python versions. In particular, the Cython project has shown interest in doing that (see `https://mail.python.org/pipermail/python-dev/2018-June/153927.html`).

A.12. **Alternative suggestions**

PEP 576 is an alternative approach to solving the same problem as this PEP. See `https://mail.python.org/pipermail/python-dev/2018-July/154238.html` for comments on the difference between PEP 576 and PEP 580.

A.13. **Reference implementation**

The reference implementation can be found at `https://github.com/jdemeyer/cpython/tree/pep580`

A.14. **Copyright**

This document has been placed in the public domain.