HPy

Release 0.9

HPy Collective

Sep 22, 2023

CONTENTS

HPy provides a new API for extending Python in C.

There are several advantages to writing C extensions in HPy:

- **Speed**: it runs much faster on PyPy, GraalPy, and at native speed on CPython
- **Deployment**: it is possible to compile a single binary which runs unmodified on all supported Python implementations and versions – think "stable ABI" on steroids
- **Simplicity**: it is simpler and more manageable than the `Python.h` API, both for the users and the Pythons implementing it
- **Debugging**: it provides an improved debugging experience. Debug mode can be turned on at runtime without the need to recompile the extension or the Python running it. HPy design is more suitable for automated checks.

The official Python/C API, also informally known as `#include <Python.h>`, is specific to the current implementation of CPython: it exposes a lot of internal details which makes it hard to:

- implement it for other Python implementations (e.g. PyPy, GraalPy, Jython, . . .)
- experiment with new approaches inside CPython itself, for example:
  - use a tracing garbage collection instead of reference counting
  - remove the global interpreter lock (GIL) to take full advantage of multicore architectures
  - use tagged pointers to reduce memory footprint

CHAPTER

ONE

WHERE TO GO NEXT:

- Show me the code:
  - *Quickstart*
  - *Simple documented HPy extension example*
  - *Tutorial: porting Python/C API extension to HPy*
- Details:
  - *HPy overview: motivation, goals, current status*
  - *HPy API concepts introduction*
  - *Python/C API to HPy Porting guide*
  - *HPy API reference*

CHAPTER

TWO

FULL TABLE OF CONTENTS:

2.1 HPy Quickstart

This section shows how to quickly get started with HPy by creating a simple HPy extension from scratch.

Install latest HPy release:

```
python3 -m pip install hpy
```

Alternatively, you can also install HPy from the development repository:

```
python3 -m pip install git+https://github.com/hpyproject/hpy.git#egg=hpy
```

Create a new directory for the new HPy extension. Location and name of the directory do not matter. Add the following two files:

```
// quickstart.c

// This header file is the entrypoint to the HPy API:
#include "hpy.h"

// HPy method: the HPyDef_METH macro generates some boilerplate code,
// the same code can be also written manually if desired
HPyDef_METH(say_hello, "say_hello", HPyFunc_NOARGS)
static HPy say_hello_impl(HPyContext *ctx, HPy self)
{
    // Methods take HPyContext, which must be passed as the first argument to
    // all HPy API functions. Other than that HPyUnicode_FromString does the
    // same thing as PyUnicode_FromString.
    //
    // HPy type represents a "handle" to a Python object, but may not be
    // a pointer to the object itself. It should be fully "opaque" to the
    // users. Try uncommenting the following two lines to see the difference
    // from PyObject*:
    //
    // if (self == self)
    //      HPyUnicode_FromString(ctx, "Surprise? Try HPy_Is(ctx, self, self)");

    return HPyUnicode_FromString(ctx, "Hello world");
}

static HPyDef *QuickstartMethods[] = {
    &say_hello, // 'say_hello' generated for us by the HPyDef_METH macro
    NULL,
```

(continues on next page)

(continued from previous page)

```
};

static HPyModuleDef quickstart_def = {
    .doc = "HPy Quickstart Example",
    .defines = QuickstartMethods,
};

// The Python interpreter will create the module for us from the
// HPyModuleDef specification. Additional initialization can be
// done in the HPy_mod_exec slot
HPy_MODINIT(quickstart, quickstart_def)
```

```
# setup.py

from setuptools import setup, Extension
from os import path

DIR = path.dirname(__file__)
setup(
    name="hpy-quickstart",
    hpy_ext_modules=[
        Extension('quickstart', sources=[path.join(DIR, 'quickstart.c')]),
    ],
    setup_requires=['hpy'],
)
```

Build the extension:

```
python3 setup.py --hpy-abi=universal develop
```

Try it out – start Python console in the same directory and type:

```
    import quickstart
    assert quickstart.say_hello() == "Hello world"
```

Notice the shared library that was created by running `setup.py`:

```
> ls *.so
quickstart.hpy0.so
```

It does not have Python version encoded in it. If you happen to have GraalPy or PyPy installation that supports given HPy version, you can try running the same extension on it. For example, start `$GRAALVM_HOME/bin/graalpy` in the same directory and type the same Python code: the extension should load and work just fine.

Where to go next?

- *Simple documented HPy extension example*
- *Tutorial: porting Python/C API extension to HPy*

2.2 HPy Overview

2.2.1 Motivation and goals

The superpower of the Python ecosystem is its libraries, which are developed by users. Over time, these libraries have grown in number, quality, and applicability. While it is possible to write python libraries entirely in python, many of them, especially in the scientific community, are written in C and exposed to Python using the Python.h API. The existence of these C extensions using the `Python.h` API leads to some issues:

1. Usually, alternative implementation of the Python programming language want to support C extensions. To do so, they must implement the same `Python.h` API or provide a compatibility layer.
2. CPython developers cannot experiment with new designs or refactoring without breaking compatibility with existing extensions.

Over the years, it has become evident that emulating `Python.h` in an efficient way is challenging, if not impossible. To summarize, it is mainly due to leaking of implementation details of CPython into the C/API - which makes it difficult to make different design choices than those made by CPython. As such - the main goal of HPy is to provide a **C API which makes as few assumptions as possible about the design decisions of any implementation of Python, allowing diverse implementations to support it efficiently and without compromise**. In particular, **reference counting is not part of the API**: we want a more generic way of managing resources that is possible to implement with different strategies, including the existing reference counting and/or with a moving *Garbage Collector* (like the ones used by PyPy, GraalPy or Java, for example). Moreover, each implementation can experiment with new memory layout of objects, add optimizations, etc. The following is a list of sub-goals.

Performance on CPython HPy is usable on CPython from day 1 with no performance impact compared to the existing `Python.h` API.

Incremental adoption It is possible to port existing C extensions piece by piece and to use the old and the new API side-by-side during the transition.

Easy migration It should be easy to migrate existing C extensions to HPy. Thanks to an appropriate and regular naming convention it should be obvious what the HPy equivalent of any existing `Python.h` API is. When a perfect replacement does not exist, the documentation explains what the alternative options are.

Better debugging In debug mode, you get early and precise errors and warnings when you make some specific kind of mistakes and/or violate the API rules and assumptions. For example, you get an error if you try to use a handle (see *Handles*) which has already been closed. It is possible to turn on the debug mode at startup time, *without needing to recompile*.

Simplicity The HPy API aims to be smaller and easier to study/use/manage than the existing `Python.h` API. Sometimes there is a trade-off between this goal and the others above, in particular *Performance on CPython* and *Easy migration*. The general approach is to have an API which is "as simple as possible" while not violating the other goals.

Universal binaries It is possible to compile extensions to a single binary which is ABI-compatible across multiple Python versions and/or multiple implementation. See *Target ABIs*.

Opt-in low level data structures Internal details might still be available, but in a opt-in way: for example, if Cython wants to iterate over a list of integers, it can ask if the implementation provides a direct low-level access to the content (e.g. in the form of a `int64_t[]` array) and use that. But at the same time, be ready to handle the generic fallback case.

2.2.2 API vs ABI

HPy defines *both* an API and an ABI. Before digging further into details, let's distinguish them:

- The **API** works at the level of source code: it is the set of functions, macros, types and structs which developers can use to write their own extension modules. For C programs, the API is generally made available through one or more header files (`*.h`).
- The **ABI** works at the level of compiled code: it is the interface between the host interpreter and the compiled DLL. Given a target CPU and operating system it defines things like the set of exported symbols, the precise memory layout of objects, the size of types, etc.

In general it is possible to compile the same source into multiple compiled libraries, each one targeting a different ABI. **PEP 3149** states that the filename of the compiled extension should contain the *ABI tag* to specify what the target ABI is. For example, if you compile an extension called `simple.c` on CPython 3.8, you get a DLL called `simple.cpython-38-x86_64-linux-gnu.so`:

- `cpython-38` is the ABI tag, in this case CPython 3.8
- `x86_64` is the CPU architecture
- `linux-gnu` is the operating system

The same source code compiled on PyPy3.6 7.2.0 results in a file called `simple.pypy38-pp73-x86_64-linux-gnu.so`:

- `pypy38-pp73` is the ABI tag, in this case "PyPy3.8", version "7.3.x"

The HPy C API is exposed to the user by including `hpy.h` and it is explained in its own section of the documentation.

2.2.3 Legacy and compatibility features

To allow an incremental transition to HPy, it is possible to use both `hpy.h` and `Python.h` API calls in the same extension. Using *HPy legacy features* you can:

- mix `Python.h` and HPy method defs in the same HPy module
- mix `Python.h` and HPy method defs and slots in the same HPy type
- convert `HPy` handles to and from `PyObject *` using `HPy_AsPyObject()` and `HPy_FromPyObject()`

Thanks to this, you can port your code to HPy one method and one type at a time, while keeping the extension fully functional during the transition period. See the *Porting Guide* for a concrete example.

Legacy features are available only if you target the CPython or HPy Hybrid ABIs, as explained in the next section.

2.2.4 Target ABIs

Depending on the compilation options, an HPy extension can target three different ABIs:

CPython ABI In this mode, HPy is implemented as a set of C macros and `static inline` functions which translate the HPy API into the CPython API at compile time. The result is a compiled extension which is indistinguishable from a "normal" one and can be distributed using all the standard tools and will run at the very same speed.

Legacy features are available.

The output filename is e.g. `simple.cpython-38-x86_64-linux-gnu.so`.

HPy Universal ABI As the name suggests, the HPy Universal ABI is designed to be loaded and executed by a variety of different Python implementations. Compiled extensions can be loaded unmodified on all the interpreters which support it. PyPy and GraalPy support it natively. CPython supports it by using the `hpy.universal` package, and there is a small speed penalty[1] compared to the CPython ABI.

Legacy features are **not** available and it is forbidden to `#include <Python.h>`.

The resulting filename is e.g. `simple.hpy0.so`.

HPy Hybrid ABI The HPy Hybrid ABI is essentially the same as the Universal ABI, with the big difference that it allows to `#include <Python.h>`, to use the legacy features and thus to allow incremental porting.

At the ABI level the resulting binary depends on *both* HPy and the specific Python implementation which was used to compile the extension. As the name suggests, this means that the binary is not "universal", thus negating some of the benefits of HPy. The main benefit of using the HPy Hybrid ABI instead of the CPython ABI is being able to use the *Debug Mode* on the HPy parts, and faster speed on alternative implementations.

Legacy features are available.

The resulting filename is e.g. `simple.hpy0-cp38.so`.

Moreover, each alternative Python implementation could decide to implement its own non-universal ABI if it makes sense for them. For example, a hypothetical project *DummyPython* could decide to ship its own `hpy.h` which implements the HPy API but generates a DLL which targets the DummyPython ABI.

This means that to compile an extension for CPython, you can choose whether to target the CPython ABI or the Universal ABI. The advantage of the former is that it runs at native speed, while the advantage of the latter is that you can distribute a single binary, although with a small speed penalty on CPython. Obviously, nothing stops you compiling and distributing both versions: this is very similar to what most projects are already doing, since they automatically compile and distribute extensions for many different CPython versions.

From the user point of view, extensions compiled for the CPython ABI can be distributed and installed as usual, while those compiled for the HPy Universal or HPy Hybrid ABIs require installing the `hpy.universal` package on CPython and have no further requirements on Pythons that support HPy natively.

2.2.5 Benefits for the Python ecosystem

The HPy project offers some benefits to the python ecosystem, both to Python users and to library developers.

- C extensions can achieve much better speed on alternative implementions, including PyPy and GraalPy: according to early *Early benchmarks*, an extension written in HPy can be ~3x faster than the equivalent extension written using `Python.h`.
- Improved debugging: when you load extensions in *Debug Mode*, many common mistakes are checked and reported automatically.
- Universal binaries: libraries can choose to distribute only Universal ABI binaries. By doing so, they can support all Python implementations and version of CPython (like PyPy, GraalPy, CPython 3.10, CPython 3.11, etc) for which an HPy loader exists, including those that do not yet exist! This currently comes with a small speed penalty on CPython, but for non-performance critical libraries it might still be a good tradeoff.
- Python environments: With general availability of universal ABI binaries for popular packages, users can create equivalent python environments that target different Python implementations. Thus, Python users can try their workload against different implementations and pick the one best suited for their usage.
- In a situation where most or all popular Python extensions target the universal ABI, it will be more feasible for CPython to make breaking changes to its C/API for performance or maintainability reasons.

[1] The reason for this minor performance penalty is a layer of pointer indirection. For instance, `ctx->HPyLong_FromLong` is called from the CPython extension, which in universal mode simply forwards the call to `PyLong_FromLong`. It is technically possible to implement a CPython universal module loader which edits the program's executable code at runtime to replace that call. Note that this is not at all trivial.

2.2.6 Cython extensions

If you use Cython, you can't use HPy directly. There is a work in progress to add Cython backend which emits HPy code instead of using `Python.h` code: once this is done, you will get the benefits of HPy automatically.

2.2.7 Extensions in other languages

On the API side, HPy is designed with C in mind, so it is not directly useful if you want to write an extension in a language other than C.

However, Python bindings for other languages could decide to target the *HPy Universal ABI* instead of the *CPython ABI*, and generate extensions which can be loaded seamlessly on all Python implementations which supports it. This is the route taken, for example, by Rust.

2.2.8 Benefits for alternative Python implementations

If you are writing an alternative Python implementation, there is a good chance that you already know how painful it is to support the `Python.h` API. HPy is designed to be both faster and easier to implement!

You have two choices:

- support the Universal ABI: in this case, you just need to export the needed functions and to add a hook to `dlopen()` the desired libraries
- use a custom ABI: in this case, you have to write your own replacement for `hpy.h` and recompile the C extensions with it.

2.2.9 Current status and roadmap

HPy left the early stages of development and already provides a noticeable set of features. As on April 2023, the following milestones have been reached:

- some prominent real-world Python packages have been ported to HPy API. There is a list of HPy-compatible packages we know about on the HPy website hpyproject.org.
- one can write extensions which expose module-level functions, with all the various kinds of calling conventions.
- there is support for argument parsing (i.e., the equivalents of `PyArg_ParseTuple` and `PyArg_ParseTupleAndKeywords`), and a convenient complex value building (i.e., the equivalent `Py_BuildValue`).
- one can implement custom types, whose struct may contain references to other Python objects using `HPyField`.
- there is a support for globally accessible Python object handles: `HPyGlobal`, which can still provide isolation for subinterpreters if needed.
- there is support for raising and catching exceptions.
- debug mode has been implemented and can be activated at run-time without recompiling. It can detect leaked handles or handles used after being closed.
- trace mode has been implemented and can be activated just like the debug mode. It helps analyzing the API usage (in particular wrt. performance).
- wheels can be built for HPy extensions with `python setup.py bdist_wheel` and can be installed with `pip install`.

- it is possible to choose between the *CPython ABI* and the *HPy Universal ABI* when compiling an extension module.
- extensions compiled with the CPython ABI work out of the box on CPython.
- it is possible to load HPy Universal extensions on CPython, thanks to the `hpy.universal` package.
- it is possible to load HPy Universal extensions on PyPy (using the PyPy hpy branch).
- it is possible to load HPy Universal extensions on GraalPy.
- there is support for multi-phase module initialization.
- support for metaclasses has been added.

However, there is still a long road before HPy is usable for the general public. In particular, the following features are on our roadmap but have not been implemented yet:

- many of the original `Python.h` functions have not been ported to HPy yet. Porting most of them is straightforward, so for now the priority is to test HPy with real-world Python packages and primarily resolve the "hard" features to prove that the HPy approach works.
- add C-level module state to complement the `HPyGlobal` approach. While `HPyGlobal` is easier to use, it will make the migration simpler for existing extensions that use CPython module state.
- the integration with Cython is work in progress
- it is not clear yet how to approach pybind11 and similar C++ bindings. They serve two use-cases:
  - As C++ wrappers for CPython API. HPy is fundamentally different in some ways, so fully compatible pybind11 port of this API to HPy does not make sense. There can be a similar or even partially pybind11 compatible C++ wrapper for HPy adhering to the HPy semantics and conventions (e.g., passing the HPyContext pointer argument around, no reference stealing, etc.).
  - Way to expose (or "bind") mostly pure C++ functions as Python functions where the C++ templating machinery takes care of the conversion between the Python world, i.e., `PyObject*`, and the C++ types. Porting this abstraction to HPy is possible and desired in the future. To determine the priority or such effort, we need to get more knowledge about existing pybind11 use-cases.

2.2.10 Early benchmarks

To validate our approach, we ported a simple yet performance critical module to HPy. We chose ultrajson because it is simple enough to require porting only a handful of API functions, but at the same time it is performance critical and performs many API calls during the parsing of a JSON file.

This blog post explains the results in more detail, but they can be summarized as follows:

- `ujson-hpy` compiled with the CPython ABI is as fast as the original `ujson`.
- A bit surprisingly, `ujson-hpy` compiled with the HPy Universal ABI is only 10% slower on CPython. We need more evidence than a single benchmark of course, but if the overhead of the HPy Universal ABI is only 10% on CPython, many projects may find it small enough that the benefits of distributing extensions using only the HPy Universal ABI out weight the performance costs.
- On PyPy, `ujson-hpy` runs 3x faster than the original `ujson`. Note the HPy implementation on PyPy is not fully optimized yet, so we expect even bigger speedups eventually.

2.2.11 Projects involved

HPy was born during EuroPython 2019, were a small group of people started to discuss the problems of the `Python.h` API and how it would be nice to have a way to fix them. Since then, it has gathered the attention and interest of people who are involved in many projects within the Python ecosystem. The following is a (probably incomplete) list of projects whose core developers are involved in HPy, in one way or the other. The mere presence in this list does not mean that the project as a whole endorse or recognize HPy in any way, just that some of the people involved contributed to the code/design/discussions of HPy:

- PyPy
- CPython
- Cython
- GraalPy
- RustPython
- rust-hpy (fork of the cpython crate)

2.2.12 Related work

A partial list of alternative implementations which offer a `Python.h` compatibility layer include:

- PyPy
- Jython
- IronPython
- GraalPy

2.3 HPy API Introduction

2.3.1 Handles

The "H" in HPy stands for **handle**, which is a central concept: handles are used to hold a C reference to Python objects, and they are represented by the C `HPy` type. They play the same role as `PyObject *` in the `Python.h` API, albeit with some important differences which are detailed below.

When they are no longer needed, handles must be closed by calling `HPy_Close`, which plays more or less the same role as `Py_DECREF`. Similarly, if you need a new handle for an existing object, you can duplicate it by calling `HPy_Dup`, which plays more or less the same role as `Py_INCREF`.

The HPy API strictly follows these rules:

- `HPy` handles returned by a function are **never borrowed**, i.e., the caller must either close or return it.
- `HPy` handles passed as function arguments are **never stolen**; if you receive a `HPy` handle argument from your caller, you should never close it.

These rules makes the code simpler to reason about. Moreover, no reference borrowing enables the Python implementations to use whatever internal representation they wish. For example, the object returned by `HPy_GetItem_i` may be created on demand from some compact internal representation, which does not need to convert itself to full blown representation in order to hold onto the borrowed object.

We strongly encourage the users of HPy to also internally follow these rules for their own internal APIs and helper functions. For the sake of simplicity and easier local reasoning and also because in the future, code adhering to those rules may be suitable target for some scalable and precise static analysis tool.

The concept of handles is certainly not unique to HPy. Other examples include Unix file descriptors, where you have `dup()` and `close()`, and Windows' `HANDLE`, where you have `DuplicateHandle()` and `CloseHandle()`.

Handles vs `PyObject *`

In order to fully understand the way HPy handles work, it is useful to discuss the `Pyobject *` pointer in `Python.h`. These pointers always point to the same object, and a python object's identity is completely given by its address in memory, and two pointers with the same address can be passed to `Python.h` API functions interchangeably. As a result, `Py_INCREF` and `Py_DECREF` can be called with any reference to an object as long as the total number of calls of `incref` is equal to the number of calls of `decref` at the end of the object lifetime.

Whereas using HPy API, each handle must be closed independently.

Thus, the following perfectly valid piece of code using `Python.h`:

```c
void foo(void)
{
    PyObject *x = PyLong_FromLong(42);  // implicit INCREF on x
    PyObject *y = x;
    Py_INCREF(y);                       // INCREF on y
    /* ... */
    Py_DECREF(x);
    Py_DECREF(x);                       // two DECREF on x
}
```

Becomes using HPy API:

```c
void foo(HPyContext *ctx)
{
    HPy x = HPyLong_FromLong(ctx, 42);
    HPy y = HPy_Dup(ctx, x);
    /* ... */
    // we need to close x and y independently
    HPy_Close(ctx, x);
    HPy_Close(ctx, y);
}
```

Calling any HPy function on a closed handle is an error. Calling `HPy_Close()` on the same handle twice is an error. Forgetting to call `HPy_Close()` on a handle results in a memory leak. When running in *Debug Mode*, HPy actively checks that you don't close a handle twice and that you don't forget to close any.

Note: Debug mode is a good example of how powerful it is to decouple the identity and therefore the lifetime of handles and those of objects. If you find a memory leak on CPython, you know that you are missing a `Py_DECREF` somewhere but the only way to find the corresponding `Py_INCREF` is to manually and carefully study the source code. On the other hand, if you forget to call `HPy_Close()`, debug mode is able to identify the precise code location which created the unclosed handle. Similarly, if you try to operate on a closed handle, it will identify the precise code locations which created and closed it. This is possible because handles are associated with a single call to a C/API function. As a result, given a handle that is leaked or used after freeing, it is possible to identify exactly the C/API function that produced it.

Remember that `Python.h` guarantees that multiple references to the same object results in the very same `PyObject *` pointer. Thus, it is possible to compare the pointer addresses to check whether they refer to the same object:

```
int is_same_object(PyObject *x, PyObject *y)
{
    return x == y;
}
```

On the other hand, in HPy, each handle is independent and it is common to have two different handles which point to the same underlying object, so comparing two handles directly is ill-defined. To prevent this kind of common error (especially when porting existing code to HPy), the `HPy` C type is opaque and the C compiler actively forbids comparisons between them. To check for identity, you can use `HPy_Is()`:

```
int is_same_object(HPyContext *ctx, HPy x, HPy y)
{
    // return x == y; // compilation error!
    return HPy_Is(ctx, x, y);
}
```

Note: The main benefit of opaque handle semantics is that implementations are allowed to use very different models of memory management. On CPython, implementing handles is trivial because `HPy` is basically `PyObject *` in disguise, and `HPy_Dup()` and `HPy_Close()` are just aliases for `Py_INCREF` and `Py_DECREF`.

Unlike CPython, PyPy does not use reference counting to manage memory: instead, it uses a *moving GC*, which means that the address of an object might change during its lifetime, and this makes it hard to implement semantics like `PyObject *`'s where the address *identifies* the object, and this is directly exposed to the user. HPy solves this problem: on PyPy, handles are integers which represent indices into a list, which is itself managed by the GC. When an address changes, the GC edits the list, without having to touch all the handles which have been passed to C.

2.3.2 HPyContext

All HPy function calls take an `HPyContext` as a first argument, which represents the Python interpreter all the handles belong to. Strictly speaking, it would be possible to design the HPy API without using `HPyContext`: after all, all HPy function calls are ultimately mapped to `Python.h` function call, where there is no notion of context.

One of the reasons to include `HPyContext` from the day one is to be future-proof: it is conceivable to use it to hold the interpreter or the thread state in the future, in particular when there will be support for sub-interpreters. Another possible usage could be to embed different versions or implementations of Python inside the same process. In addition, the `HPyContext` may also be extended by adding new functions to the end without breaking any extensions built against the current `HPyContext`.

Moreover, `HPyContext` is used by the *HPy Universal ABI* to contain a sort of virtual function table which is used by the C extensions to call back into the Python interpreter.

2.3.3 A simple example

In this section, we will see how to write a simple C extension using HPy. It is assumed that you are already familiar with the existing `Python.h` API, so we will underline the similarities and the differences with it.

We want to create a function named `myabs` and `double` which takes a single argument and computes its absolute value:

```
#include "hpy.h"

HPyDef_METH(myabs, "myabs", HPyFunc_O)
```

(continues on next page)

(continued from previous page)

```
static HPy myabs_impl(HPyContext *ctx, HPy self, HPy arg)
{
    return HPy_Absolute(ctx, arg);
}
```

There are a couple of points which are worth noting:

- We use the macro `HPyDef_METH` to declare we are going to define a HPy function called `myabs`.
- The function will be available under the name `"myabs"` in our Python module.
- The actual C function which implements `myabs` is called `myabs_impl` and is inferred by the macro. The macro takes the name and adds `_impl` to the end of it.
- It uses the `HPyFunc_O` calling convention. Like `METH_O` in `Python.h`, `HPyFunc_O` means that the function receives a single argument on top of `self`.
- `myabs_impl` takes two arguments of type `HPy`: handles for `self` and the argument, which are guaranteed to be valid. They are automatically closed by the caller, so there is no need to call `HPy_Close` on them.
- `myabs_impl` returns a handle, which has to be closed by the caller.
- `HPy_Absolute` is the equivalent of `PyNumber_Absolute` and computes the absolute value of the given argument.
- We also do not call `HPy_Close` on the result returned to the caller. We must return a valid handle.

Note: Among other things, the `HPyDef_METH` macro is needed to maintain compatibility with CPython. In CPython, C functions and methods have a C signature that is different to the one used by HPy: they don't receive an `HPyContext` and their arguments have the type `PyObject *` instead of `HPy`. The macro automatically generates a trampoline function whose signature is appropriate for CPython and which calls the `myabs_impl`. This trampoline is then used from both the CPython ABI and the CPython implementation of the universal ABI, but other implementations of the universal ABI will usually call directly the HPy function itself.

The second function definition is a bit different:

```
HPyDef_METH_IMPL(double_num, "double", double_impl, HPyFunc_O)
static HPy double_impl(HPyContext *ctx, HPy self, HPy arg)
{
    return HPy_Add(ctx, arg, arg);
}
```

This shows off the other way of creating functions.

- This example is much the same but the difference is that we use `HPyDef_METH_IMPL` to define a function named `double`.
- The difference between `HPyDef_METH_IMPL` and `HPyDef_METH` is that the former needs to be given a name for a the functions as the third argument.

Now, we can define our module:

```
static HPyDef *SimpleMethods[] = {
        &myabs,
        &double_num,
        NULL,
};

```

(continues on next page)

(continued from previous page)

```
static HPyModuleDef simple = {
        .doc = "HPy Example",
        .size = 0,
        .defines = SimpleMethods,
        .legacy_methods = NULL
};
```

This part is very similar to the one you would write with `Python.h`. Note that we specify `myabs` (and **not** `myabs_impl`) in the method table. There is also the `.legacy_methods` field, which allows to add methods that use the `Python.h` API, i.e., the value should be an array of `PyMethodDef`. This feature enables support for hybrid extensions in which some of the methods are still written using the `Python.h` API.

Note that the HPy module does not specify its name. HPy does not support the legacy single phase module initialization and the only module initialization approach is the multi-phase initialization (PEP 489). With multi-phase module initialization, the name of the module is always taken from the `ModuleSpec` (PEP 451) , i.e., most likely from the name used in the `import {{name}}` statement that imported your module.

This is the only difference stemming from multi-phase module initialization in this simple example. As long as there is no need for any further initialization, we can just "register" our module using the `HPy_MODINIT` convenience macro. The first argument is the name of the extension file and is needed for HPy, among other things, to be able to generate the entry point for CPython called `PyInit_{{name}}`. The second argument is the `HPyModuleDef` we just defined.

```
HPy_MODINIT(simple, simple)
```

Building the module

Let's write a `setup.py` to build our extension:

```
from setuptools import setup, Extension
from os import path

setup(
    name="hpy-simple-example",
    hpy_ext_modules=[
        Extension('simple', sources=[path.join(path.dirname(__file__), 'simple.c')]),
    ],
    setup_requires=['hpy'],
)
```

We can now build the extension by running `python setup.py build_ext -i`. On CPython, it will target the *CPython ABI* by default, so you will end up with a file named e.g. `simple.cpython-37m-x86_64-linux-gnu.so` which can be imported directly on CPython with no dependency on HPy.

To target the *HPy Universal ABI* instead, it is possible to pass the option `--hpy-abi=universal` to `setup.py`. The following command will produce a file called `simple.hpy.so` (note that you need to specify `--hpy-abi` **before** `build_ext`, since it is a global option):

```
python setup.py --hpy-abi=universal build_ext -i
```

Note: This command will also produce a Python file named `simple.py`, which loads the HPy module using the `universal.load` function from the `hpy` Python package.

VARARGS calling convention

If we want to receive more than a single arguments, we need the `HPy_METH_VARARGS` calling convention. Let's add a function `add_ints` which adds two integers:

```
HPyDef_METH(add_ints, "add_ints", HPyFunc_VARARGS)
static HPy add_ints_impl(HPyContext *ctx, HPy self, const HPy *args, size_t nargs)
{
    long a, b;
    if (!HPyArg_Parse(ctx, NULL, args, nargs, "ll", &a, &b))
        return HPy_NULL;
    return HPyLong_FromLong(ctx, a+b);
}
```

There are a few things to note:

- The C signature is different than the corresponding `Python.h` `METH_VARARGS`: in particular, instead of taking a tuple `PyObject *args`, we take an array of `HPy` and its size. This allows the call to happen more efficiently, because you don't need to create a tuple just to pass the arguments.
- We call `HPyArg_Parse` to parse the arguments. Contrarily to almost all the other HPy functions, this is **not** a thin wrapper around `PyArg_ParseTuple` because as stated above we don't have a tuple to pass to it, although the idea is to mimic its behavior as closely as possible. The parsing logic is implemented from scratch inside HPy, and as such there might be missing functionality during the early stages of HPy development.
- If an error occurs, we return `HPy_NULL`: we cannot simply `return NULL` because `HPy` is not a pointer type.

Once we have written our function, we can add it to the `SimpleMethods[]` table, which now becomes:

```
static HPyDef *SimpleMethods[] = {
        &myabs,
        &add_ints,
        NULL,
};
```

2.3.4 Creating types in HPy

Creating Python types in an HPy extension is again very similar to the C API with the difference that HPy only supports creating types from a specification. This is necessary because there is no such C-level type as `PyTypeObject` since that would expose the internal implementation.

Creating a simple type in HPy

This section assumes that the user wants to define a type that stores some data in a C-level structure. As an example, we will create a simple C structure `PointObject` that represents a two-dimensional point.

```
typedef struct {
    long x;
    long y;
} PointObject;
HPyType_HELPERS(PointObject)
```

The macro call `HPyType_HELPERS(PointObject)` generates useful helper facilities for working with the type. It generates a C enum `PointObject_SHAPE` and a helper function `PointObject_AsStruct`. The enum is used in the type specification. The helper function is used to efficiently retrieving the pointer `PointObject *` from an HPy handle to be able to access the C structure. We will use this helper function to implement the methods, get-set descriptors, and slots.

It makes sense to expose fields `PointObject.x` and `PointObject.y` as Python-level members. To do so, we need to define members by specifying their name, type, and location using HPy's convenience macro `HPyDef_MEMBER`:

```
HPyDef_MEMBER(Point_x, "x", HPyMember_LONG, offsetof(PointObject, x))
HPyDef_MEMBER(Point_y, "y", HPyMember_LONG, offsetof(PointObject, y))
```

The first argument of the macro is the name for the C glabal variable that will store the necessary information. We will need that later for registration of the type. The second, third, and fourth arguments are the Python-level name, the C type of the member, and the offset in the C structure, respectively.

Similarly, methods and get-set descriptors can be defined. For example, method `foo` is an instance method that takes no arguments (the self argument is, of course, implicit), does some computation with fields `x` and `y` and returns a Python `int`:

```
HPyDef_METH(Point_foo, "foo", HPyFunc_NOARGS)
static HPy Point_foo_impl(HPyContext *ctx, HPy self)
{
    PointObject *point = PointObject_AsStruct(ctx, self);
    return HPyLong_FromLong(ctx, point->x * 10 + point->y);
}
```

Get-set descriptors are also defined in a very similar way as methods. The following example defines a get-set descriptor for attribute `z` which is calculated from the `x` and `y` fields of the struct.

```
HPyDef_GETSET(Point_z, "z", .closure=(void *)1000)
static HPy Point_z_get(HPyContext *ctx, HPy self, void *closure)
{
    PointObject *point = PointObject_AsStruct(ctx, self);
    return HPyLong_FromLong(ctx, point->x*10 + point->y + (long)(HPy_ssize_t)closure);
}

static int Point_z_set(HPyContext *ctx, HPy self, HPy value, void *closure)
{
    PointObject *point = PointObject_AsStruct(ctx, self);
    long current = point->x*10 + point->y + (long)(HPy_ssize_t)closure;
    long target = HPyLong_AsLong(ctx, value);  // assume no exception
    point->y += target - current;
    return 0;
}
```

It is also possible to define a get-descriptor or a set-descriptor by using HPy's macros `HPyDef_GET` and `HPyDef_SET` in the same way.

HPy also supports type slots. In this example, we will define slot `HPy_tp_new` (which corresponds to magic method `__new__`) to initialize fields `x` and `y` when constructing the object:

```
HPyDef_SLOT(Point_new, HPy_tp_new)
static HPy Point_new_impl(HPyContext *ctx, HPy cls, const HPy *args,
        HPy_ssize_t nargs, HPy kw)
{
    long x, y;
    if (!HPyArg_Parse(ctx, NULL, args, nargs, "ll", &x, &y))
        return HPy_NULL;
    PointObject *point;
```

(continues on next page)

(continued from previous page)

```
    HPy h_point = HPy_New(ctx, cls, &point);
    if (HPy_IsNull(h_point))
        return HPy_NULL;
    point->x = x;
    point->y = y;
    return h_point;
}
```

After everything was defined, we need to create a list of all defines such that we are able to eventually register them to the type:

```
static HPyDef *Point_defines[] = {
    &Point_x,
    &Point_y,
    &Point_z,
    &Point_new,
    &Point_foo,
    NULL
};
```

Please note that it is required to terminate the list with `NULL`. We can now create the actual type specification by appropriately filling an `HPyType_Spec` structure:

```
static HPyType_Spec Point_spec = {
    .name = "simple_type.Point",
    .basicsize = sizeof(PointObject),
    .builtin_shape = PointObject_SHAPE,
    .defines = Point_defines
};
```

First, we need to define the name of the type by setting a C string to member `name`. Since this type has a C structure, we need to define the `basicsize` and best practice is to set it to `sizeof(PointObject)`. Also best practice is to set `builtin_shape` to `PointObject_SHAPE` where `PointObject_SHAPE` is generated by the previous usage of macro `HPyType_HELPERS(PointObject)`. Last but not least, we need to register the defines by setting field `defines` to the previously defined array `Point_defines`.

The type specification for the simple type `simple_type.Point` represented in C by structure `PointObject` is now complete. All that remains is to create the type object and add it to the module.

We will define a module execute slot, which is executed by the runtime right after the module is created. The purpose of the execute slot is to initialize the newly created module object. We can then add the type by using *`HPyHelpers_AddType()`*:

```
HPyDef_SLOT(simple_exec, HPy_mod_exec)
static int simple_exec_impl(HPyContext *ctx, HPy m) {
    if (!HPyHelpers_AddType(ctx, m, "Point", &Point_spec, NULL)) {
        return -1;
    }
    return 0; // success
}

static HPyDef *mod_defines[] = {
    &simple_exec, // 'simple_exec' is generated by the HPyDef_SLOT macro
    NULL,
};

```

(continues on next page)

(continued from previous page)

```
static HPyModuleDef moduledef = {
    .defines = mod_defines,
    // ...
```

Also look at the full example at: examples/hpytype-example/simple_type.

Legacy types

A type whose struct starts with `PyObject_HEAD` (either directly by embedding it in the type struct or indirectly by embedding another struct like `PyLongObject`) is a *legacy type*. A legacy type must set `.builtin_shape = HPyType_BuiltinShape_Legacy` in its `HPyType_Spec`. The counterpart (i.e. a non-legacy type) is called HPy pure type.

Legacy types are available to allow gradual porting of existing CPython extensions. It is possible to reuse existing `PyType_Slot` entities (i.e. slots, methods, members, and get/set descriptors). The idea is that you can then migrate one after each other while still running the tests.

The major restriction when using legacy types is that you cannot build a universal binary of your HPy extension (i.e. you cannot use *HPy Universal ABI*). The resulting binary will be specific to the Python interpreter used for building. Therefore, the goal should always be to fully migrate to HPy pure types.

A type with `.legacy_slots != NULL` is required to have `HPyType_BuiltinShape_Legacy` and to include `PyObject_HEAD` at the start of its struct. It would be easy to relax this requirement on CPython (where the `PyObject_HEAD` fields are always present) but a large burden on other implementations (e.g. PyPy, GraalPy) where a struct starting with `PyObject_HEAD` might not exist.

Types created via the old Python C API are automatically legacy types.

This section does not provide a dedicated example for how to create and use legacy types because the *Porting Example* already shows how that is useful during incremental migration to HPy.

Inherit from a built-in type

HPy also supports inheriting from following built-in types:

- `type`
- `int`
- `float`
- `unicode`
- `tuple`
- `list`

Inheriting from built-in types is straight forward if you don't have a C structure that represents your type. In other words, you can simply inherit from, e.g., `str` if the `basicsize` in your type specification is `0`. For example:

```
static HPyType_Spec Dummy_spec = {
    .name = "builtin_type.Dummy",
    .basicsize = 0
};
```

```
    HPyType_SpecParam param[] = {
        { HPyType_SpecParam_Base, ctx->h_UnicodeType },
        { (HPyType_SpecParam_Kind)0 }
    };
    if (!HPyHelpers_AddType(ctx, module, "Dummy", &Dummy_spec, param))
        return;
```

This case is simple because there is no `Dummy_AsStruct` since there is no associated C-level structure.

It is, however, more involved if your type also defines its own C structure (i.e. `basicsize > 0` in the type specification). In this case, it is strictly necessary to use the right *built-in shape*.

What is the right built-in shape?

This question is easy to answer: Each built-in shape (except of *`HPyType_BuiltinShape_Legacy`*) represents a built-in type. You need to use the built-in shape that fits to the specified base class. The mapping is described in *`HPyType_BuiltinShape`*.

Let's do an example. Assume we want to define a type that stores the natural language of a unicode string to the unicode object but the object should still just behave like a Python unicode object. So, we define struct `LanguageObject`:

```
typedef struct {
    char *language;
} LanguageObject;
HPyType_HELPERS(LanguageObject, HPyType_BuiltinShape_Unicode)
```

As you can see, we already specify the built-in shape here using `HPyType_HELPERS(LanguageObject, HPyType_BuiltinShape_Unicode)`. Then, in the type specification, we do:

```
static HPyType_Spec Language_spec = {
    .name = "builtin_type.Language",
    .basicsize = sizeof(LanguageObject),
    .builtin_shape = SHAPE(LanguageObject),
    .defines = Language_defines
};
```

In the last step, when actually creating the type from the specification, we need to define that its base class is `str` (aka. `UnicodeType`):

```
    HPyType_SpecParam param[] = {
        { HPyType_SpecParam_Base, ctx->h_UnicodeType },
        { (HPyType_SpecParam_Kind)0 }
    };
    if (!HPyHelpers_AddType(ctx, module, "Language", &Language_spec, param))
        return;
```

Function `LanguageObject_AsStruct` (which is generated by `HPyType_HELPERS`) will then return a pointer to `LanguageObject`.

To summarize this: Specifying a type that inherits from a built-in type needs to be considered in three places:

1. Pass the appropriate built-in shape to *`HPyType_HELPERS`*.
2. Assign `SHAPE(TYPE)` to *`HPyType_Spec.builtin_shape`*.
3. Specify the desired base class in the type specification parameters.

For more information about the built-in shape and for a technical explanation for why it is required, see *`HPyType_Spec.builtin_shape`* and *`HPyType_BuiltinShape`*.

2.3.5 More Examples

The *Porting Example* shows another complete example of HPy extension ported from Python/C API.

The HPy project space on GitHub contains forks of some popular Python extensions ported to HPy as a proof of concept/feasibility studies, such as the Kiwi solver. Note that those forks may not be up to date with their upstream projects or with the upstream HPy changes.

HPy unit tests

HPy usually has tests for each API function. This means that there is lots of examples available by looking at the tests. However, the test source uses many macros and is hard to read. To overcome this we supply a utility to export clean C sources for the tests. Since the HPy tests are not shipped by default, you need to clone the HPy repository from GitHub:

```
> git clone https://github.com/hpyproject/hpy.git
```

After that, install all test requirements and dump the sources:

```
> cd hpy
> python3 -m pip install pytest filelock
> python3 -m pytest --dump-dir=test_sources test/
```

This will dump the generated test sources into folder `test_sources`. Note, that the tests won't be executed but skipped with an appropriate message.

2.4 Porting Guide

2.4.1 Porting `PyObject *` to HPy API constructs

While in CPython one always uses `PyObject *` to reference to Python objects, in HPy there are several types of handles that should be used depending on the life-time of the handle: `HPy`, `HPyField`, and `HPyGlobal`.

- `HPy` represents short lived handles that live no longer than the duration of one call from Python to HPy extension function. Rule of thumb: use for local variables, arguments, and return values.
- `HPyField` represents handles that are Python object struct fields, i.e., live in native memory attached to some Python object.
- `HPyGlobal` represents handles stored in C global variables. `HPyGlobal` can provide isolation between subinterpreters.

Warning: Never use a local variable of type `HPyField`, for any reason! If the GC kicks in, it might become invalid and become a dangling pointer.

Warning: Never store *HPy* handles to a long-lived memory, for example: C global variables or Python object structs.

The `HPy`/`HPyField` dichotomy might seem arbitrary at first, but it is needed to allow Python implementations to use a moving GC, such as PyPy. It is easier to explain and understand the rules by thinking about how a moving GC interacts with the C code inside an HPy extension.

It is worth remembering that during the collection phase, a moving GC might move an existing object to another memory location, and in that case it needs to update all the places which store a pointer to it. In order to do so, it needs to *know* where the pointers are. If there is a local C variable which is unknown to the GC but contains a pointer to a GC-managed object, the variable will point to invalid memory as soon as the object is moved.

Back to `HPy` vs `HPyField` vs `HPyGlobal`:

- `HPy` handles must be used for all C local variables, function arguments and function return values. They are supposed to be short-lived and closed as soon as they are no longer needed. The debug mode will report a long-lived `HPy` as a potential memory leak.
- In PyPy and GraalPy, `HPy` handles are implemented using an indirection: they are indexes inside a big list of GC-managed objects: this big list is tracked by the GC, so when an object moves its pointer is correctly updated.
- `HPyField` is for long-lived references, and the GC must be aware of their location in memory. In PyPy, an `HPyField` is implemented as a direct pointer to the object, and thus we need a way to inform the GC where it is in memory, so that it can update its value upon moving: this job is done by `tp_traverse`, as explained in the next section.
- `HPyGlobal` is for long-lived references that are supposed to be closed implicitly when the module is unloaded (once module unloading is actually implemented). `HPyGlobal` provides indirection to isolate subinterpreters. Implementation wise, `HPyGlobal` will usually contain an index to a table with Python objects stored in the interpreter state.
- On CPython without subinterpreters support, `HPy`, `HPyGlobal`, and `HPyField` are implemented as `PyObject *`.
- On CPython with subinterpreters support, `HPyGlobal` will be implemented by an indirection through the interpreter state. Note that thanks to the HPy design, switching between this and the more efficient implementation without subinterpreter support will not require rebuilding of the extension (in HPy universal mode), nor rebuilding of CPython.

Note: If you write a custom type using `HPyField`, you **MUST** also write a `tp_traverse` slot. Note that this is different than the old `Python.h` API, where you need `tp_traverse` only under certain conditions. See the next section for more details.

Note: The contract of `tp_traverse` is that it must visit all members of type `HPyField` contained within given struct, or more precisely *owned* by given Python object (in the sense of the *owner* argument to `HPyField_Store`), and nothing more, nothing less. Some Python implementations may choose to not call the provided `tp_traverse` if they know how to visit all members of type `HPyField` by other means (for example, when they track them internally already). The debug mode will check this contract.

`tp_traverse`, `tp_clear`, `Py_TPFLAGS_HAVE_GC`

Let's quote the `Python.h` documentation about GC support

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

A good rule of thumb is that if your type contains `PyObject *` fields, you need to:

1. provide a `tp_traverse` slot;
2. provide a `tp_clear` slot;

3. add the `Py_TPFLAGS_GC` to the `tp_flags`.

However, if you know that your `PyObject *` fields will contain only “atomic” types, you can avoid these steps.

In HPy the rules are slightly different:

1. if you have a field of type `HPyField`, you always **MUST** provide a `tp_traverse`. This is needed so that a moving GC can track the relevant areas of memory. However, you **MUST NOT** rely on `tp_traverse` to be called;
2. `tp_clear` does not exist. On CPython, `HPy` automatically generates one for you, by using `tp_traverse` to know which are the fields to clear. Other implementations are free to ignore it, if it’s not needed;
3. `HPy_TPFLAGS_GC` is still needed, especially on CPython. If you don’t specify it, your type will not be tracked by CPython’s GC and thus it might cause memory leaks if it’s part of a reference cycle. However, other implementations are free to ignore the flag and track the objects anyway, if their GC implementation allows it.

`tp_dealloc` and `Py_DECREF`

Generally speaking, if you have one or more `PyObject *` fields in the old `Python.h`, you must provide a `tp_dealloc` slot where you `Py_DECREF` all of them. In HPy this is not needed and will be handled automatically by the system.

In particular, when running on top of CPython, HPy will automatically provide a `tp_dealloc` which decrefs all the fields listed by `tp_traverse`.

See also, *Deallocator slot Py_tp_dealloc*.

2.4.2 Direct C API to HPy mappings

In many cases, migrating to HPy is as easy as just replacing a certain C API function by the appropriate HPy API function. Table *Safe API function mapping* gives a mapping between C API and HPy API functions. This mapping is generated together with the code for the *CPython ABI* mode, so it is guaranteed to be correct.

Table 1: Safe API function mapping

C API function	HPY API function
PyBool_FromLong	*`HPyBool_FromLong()`*
PyBytes_AS_STRING	`HPyBytes_AS_STRING()`
PyBytes_AsString	`HPyBytes_AsString()`
PyBytes_Check	`HPyBytes_Check()`
PyBytes_FromString	`HPyBytes_FromString()`
PyBytes_GET_SIZE	`HPyBytes_GET_SIZE()`
PyBytes_Size	`HPyBytes_Size()`
PyCallable_Check	`HPyCallable_Check()`
PyCapsule_IsValid	`HPyCapsule_IsValid()`
PyContextVar_Get	`HPyContextVar_Get()`
PyContextVar_New	`HPyContextVar_New()`
PyContextVar_Set	`HPyContextVar_Set()`
PyDict_Check	*`HPyDict_Check()`*
PyDict_Copy	*`HPyDict_Copy()`*
PyDict_Keys	*`HPyDict_Keys()`*
PyDict_New	*`HPyDict_New()`*
PyErr_Clear	*`HPyErr_Clear()`*
PyErr_ExceptionMatches	*`HPyErr_ExceptionMatches()`*

continues on next page

Table 1 – continued from previous page

C API function	HPY API function
PyErr_NewException	*HPyErr_NewException()*
PyErr_NewExceptionWithDoc	*HPyErr_NewExceptionWithDoc()*
PyErr_NoMemory	*HPyErr_NoMemory()*
PyErr_SetFromErrnoWithFilename	*HPyErr_SetFromErrnoWithFilename()*
PyErr_SetFromErrnoWithFilenameObjects	*HPyErr_SetFromErrnoWithFilenameObjects()*
PyErr_SetObject	*HPyErr_SetObject()*
PyErr_SetString	*HPyErr_SetString()*
PyErr_WarnEx	*HPyErr_WarnEx()*
PyErr_WriteUnraisable	*HPyErr_WriteUnraisable()*
PyEval_EvalCode	*HPy_EvalCode()*
PyEval_RestoreThread	*HPy_ReenterPythonExecution()*
PyEval_SaveThread	*HPy_LeavePythonExecution()*
PyFloat_AsDouble	`HPyFloat_AsDouble()`
PyFloat_FromDouble	`HPyFloat_FromDouble()`
PyImport_ImportModule	`HPyImport_ImportModule()`
PyList_Append	`HPyList_Append()`
PyList_Check	`HPyList_Check()`
PyList_New	`HPyList_New()`
PyLong_AsDouble	`HPyLong_AsDouble()`
PyLong_AsLong	*HPyLong_AsLong()*
PyLong_AsLongLong	*HPyLong_AsLongLong()*
PyLong_AsSize_t	`HPyLong_AsSize_t()`
PyLong_AsSsize_t	`HPyLong_AsSsize_t()`
PyLong_AsUnsignedLong	*HPyLong_AsUnsignedLong()*
PyLong_AsUnsignedLongLong	*HPyLong_AsUnsignedLongLong()*
PyLong_AsUnsignedLongLongMask	*HPyLong_AsUnsignedLongLongMask()*
PyLong_AsUnsignedLongMask	*HPyLong_AsUnsignedLongMask()*
PyLong_AsVoidPtr	`HPyLong_AsVoidPtr()`
PyLong_FromLong	*HPyLong_FromLong()*
PyLong_FromLongLong	*HPyLong_FromLongLong()*
PyLong_FromSize_t	`HPyLong_FromSize_t()`
PyLong_FromSsize_t	`HPyLong_FromSsize_t()`
PyLong_FromUnsignedLong	*HPyLong_FromUnsignedLong()*
PyLong_FromUnsignedLongLong	*HPyLong_FromUnsignedLongLong()*
PyNumber_Absolute	`HPy_Absolute()`
PyNumber_Add	`HPy_Add()`
PyNumber_And	`HPy_And()`
PyNumber_Check	`HPyNumber_Check()`
PyNumber_Divmod	`HPy_Divmod()`
PyNumber_Float	`HPy_Float()`
PyNumber_FloorDivide	`HPy_FloorDivide()`
PyNumber_InPlaceAdd	`HPy_InPlaceAdd()`
PyNumber_InPlaceAnd	`HPy_InPlaceAnd()`
PyNumber_InPlaceFloorDivide	`HPy_InPlaceFloorDivide()`
PyNumber_InPlaceLshift	`HPy_InPlaceLshift()`
PyNumber_InPlaceMatrixMultiply	`HPy_InPlaceMatrixMultiply()`
PyNumber_InPlaceMultiply	`HPy_InPlaceMultiply()`
PyNumber_InPlaceOr	`HPy_InPlaceOr()`
PyNumber_InPlacePower	`HPy_InPlacePower()`

continues on next page

Table 1 – continued from previous page

C API function	HPY API function
PyNumber_InPlaceRemainder	HPy_InPlaceRemainder()
PyNumber_InPlaceRshift	HPy_InPlaceRshift()
PyNumber_InPlaceSubtract	HPy_InPlaceSubtract()
PyNumber_InPlaceTrueDivide	HPy_InPlaceTrueDivide()
PyNumber_InPlaceXor	HPy_InPlaceXor()
PyNumber_Index	HPy_Index()
PyNumber_Invert	HPy_Invert()
PyNumber_Long	HPy_Long()
PyNumber_Lshift	HPy_Lshift()
PyNumber_MatrixMultiply	HPy_MatrixMultiply()
PyNumber_Multiply	HPy_Multiply()
PyNumber_Negative	HPy_Negative()
PyNumber_Or	HPy_Or()
PyNumber_Positive	HPy_Positive()
PyNumber_Power	HPy_Power()
PyNumber_Remainder	HPy_Remainder()
PyNumber_Rshift	HPy_Rshift()
PyNumber_Subtract	HPy_Subtract()
PyNumber_TrueDivide	HPy_TrueDivide()
PyNumber_Xor	HPy_Xor()
PyObject_ASCII	*HPy_ASCII()*
PyObject_Bytes	*HPy_Bytes()*
PyObject_Call	*HPy_CallTupleDict()*
PyObject_DelItem	*HPy_DelItem()*
PyObject_GetAttr	*HPy_GetAttr()*
PyObject_GetAttrString	*HPy_GetAttr_s()*
PyObject_GetItem	*HPy_GetItem()*
PyObject_HasAttr	*HPy_HasAttr()*
PyObject_HasAttrString	*HPy_HasAttr_s()*
PyObject_Hash	*HPy_Hash()*
PyObject_IsTrue	*HPy_IsTrue()*
PyObject_Length	HPy_Length()
PyObject_Repr	*HPy_Repr()*
PyObject_RichCompare	*HPy_RichCompare()*
PyObject_RichCompareBool	*HPy_RichCompareBool()*
PyObject_SetAttr	*HPy_SetAttr()*
PyObject_SetAttrString	*HPy_SetAttr_s()*
PyObject_SetItem	*HPy_SetItem()*
PyObject_Str	*HPy_Str()*
PyObject_Type	*HPy_Type()*
PyObject_TypeCheck	*HPy_TypeCheck()*
PyObject_Vectorcall	*HPy_Call()*
PyObject_VectorcallMethod	*HPy_CallMethod()*
PySequence_Contains	HPy_Contains()
PySlice_AdjustIndices	*HPySlice_AdjustIndices()*
PySlice_Unpack	HPySlice_Unpack()
PyTuple_Check	HPyTuple_Check()
PyType_IsSubtype	*HPyType_IsSubtype()*
PyUnicode_AsASCIIString	HPyUnicode_AsASCIIString()

continues on next page

Table 1 – continued from previous page

C API function	HPY API function
PyUnicode_AsLatin1String	`HPyUnicode_AsLatin1String()`
PyUnicode_AsUTF8AndSize	`HPyUnicode_AsUTF8AndSize()`
PyUnicode_AsUTF8String	`HPyUnicode_AsUTF8String()`
PyUnicode_Check	`HPyUnicode_Check()`
PyUnicode_DecodeASCII	`HPyUnicode_DecodeASCII()`
PyUnicode_DecodeFSDefault	`HPyUnicode_DecodeFSDefault()`
PyUnicode_DecodeFSDefaultAndSize	`HPyUnicode_DecodeFSDefaultAndSize()`
PyUnicode_DecodeLatin1	`HPyUnicode_DecodeLatin1()`
PyUnicode_EncodeFSDefault	`HPyUnicode_EncodeFSDefault()`
PyUnicode_FromEncodedObject	`HPyUnicode_FromEncodedObject()`
PyUnicode_FromString	`HPyUnicode_FromString()`
PyUnicode_FromWideChar	`HPyUnicode_FromWideChar()`
PyUnicode_ReadChar	`HPyUnicode_ReadChar()`
PyUnicode_Substring	`HPyUnicode_Substring()`
Py_FatalError	*`HPy_FatalError()`*

2.4.3 Reference Counting `Py_INCREF` and `Py_DECREF`

The equivalents of `Py_INCREF` and `Py_DECREF` are essentially `HPy_Dup()` and `HPy_Close()`, respectively. The main difference is that `HPy_Dup()` gives you a *new handle* to the same object which means that the two handles may be different if comparing them with `memcmp` but still reference the same object. As a consequence, you may close a handle only once, i.e., you cannot call `HPy_Close()` twice on the same `HPy` handle, even if returned from `HPy_Dup`. For examples, see also sections *Handles* and *Handles vs PyObject **

2.4.4 Calling functions `PyObject_Call` and `PyObject_CallObject`

Both `PyObject_Call` and `PyObject_CallObject` are replaced by `HPy_CallTupleDict(callable, args, kwargs)` in which either or both of `args` and `kwargs` may be null handles.

`PyObject_Call(callable, args, kwargs)` becomes:

```
HPy result = HPy_CallTupleDict(ctx, callable, args, kwargs);
```

`PyObject_CallObject(callable, args)` becomes:

```
HPy result = HPy_CallTupleDict(ctx, callable, args, HPy_NULL);
```

If `args` is not a handle to a tuple or `kwargs` is not a handle to a dictionary, `HPy_CallTupleDict` will return `HPy_NULL` and raise a `TypeError`. This is different to `PyObject_Call` and `PyObject_CallObject` which may segfault instead.

2.4.5 Calling Protocol

Both the *tp_call* and *vectorcall* calling protocols are replaced by HPy's calling protocol. This is done by defining slot `HPy_tp_call`. HPy uses only one calling convention which is similar to the vectorcall calling convention. In the following example, we implement a call function for a simple Euclidean vector type. The function computes the dot product of two vectors.

```
typedef struct {
    long x;
    long y;
} EuclideanVectorObject;
HPyType_HELPERS(EuclideanVectorObject)
```

```
HPyDef_SLOT(call, HPy_tp_call)
static HPy
call_impl(HPyContext *ctx, HPy self, const HPy *args, size_t nargs,
          HPy kwnames)
{
    static const char *keywords[] = { "x1", "y1", NULL };
    long x1, y1;
    HPyTracker ht;
    if (!HPyArg_ParseKeywords(ctx, &ht, args, nargs, kwnames, "ll", keywords,
                &x1, &y1)) {
        return HPy_NULL;
    }
    EuclideanVectorObject *data = EuclideanVectorObject_AsStruct(ctx, self);
    return HPyLong_FromLong(ctx, data->x * x1 + data->y * y1);
}
```

Positional and keyword arguments are passed as C array `args`. Argument `nargs` specifies the number of positional arguments. Argument `kwnames` is a tuple containing the names of the keyword arguments. The keyword argument values are appended to positional arguments and start at `args[nargs]` (if there are any).

In the above example, function `call_impl` will be used by default to call all instances of the corresponding type. It is also possible to install (maybe specialized) call function implementations per instances by using function *`HPy_SetCallFunction()`*. This needs to be done in the constructor of an object. For example:

```
HPyDef_CALL_FUNCTION(special_call)
static HPy
special_call_impl(HPyContext *ctx, HPy self, const HPy *args, size_t nargs,
                  HPy kwnames)
{
    HPy tmp = call_impl(ctx, self, args, nargs, kwnames);
    HPy res = HPy_Negative(ctx, tmp);
    HPy_Close(ctx, tmp);
    return res;
}

HPyDef_SLOT(new, HPy_tp_new)
static HPy
new_impl(HPyContext *ctx, HPy cls, const HPy *args, HPy_ssize_t nargs, HPy kw)
{
    static const char *keywords[] = { "x", "y", "use_special_call", NULL };
    HPyTracker ht;
    long x, y;
    HPy use_special_call = ctx->h_False;
    if (!HPyArg_ParseKeywordsDict(ctx, &ht, args, nargs, kw, "ll|O", keywords,
```

(continues on next page)

(continued from previous page)

```
                &x, &y, &use_special_call)) {
        return HPy_NULL;
    }
    EuclideanVectorObject *vector;
    HPy h_point = HPy_New(ctx, cls, &vector);
    if (HPy_IsNull(h_point)) {
        HPyTracker_Close(ctx, ht);
        return HPy_NULL;
    }
    if (HPy_IsTrue(ctx, use_special_call) &&
            HPy_SetCallFunction(ctx, h_point, &special_call) < 0) {
        HPyTracker_Close(ctx, ht);
        HPy_Close(ctx, h_point);
        return HPy_NULL;
    }
    HPyTracker_Close(ctx, ht);
    vector->x = x;
    vector->y = y;
    return h_point;
}
```

Limitations

1. It is not possible to use slot `HPy_tp_call` for a *var object* (i.e. if `HPyType_Spec.itemsize` is greater `0`). Reason: HPy installs a hidden field in the object's data to store the call function pointer which is appended to everything else. In case of `EuclideanVectorObject`, a field is implicitly appended after member `y`. This is not possible for var objects because the variable part will also start after the fixed members.
2. It is also not possible to use slot `HPy_tp_call` with a legacy type that inherits the basicsize (i.e. if `HPyType_Spec.basicsize` is `0`) for the same reason as above.

To overcome these limitations, it is still possible to manually embed a field for the call function pointer in a type's C struct and tell HPy where this field is. In this case, it is always necessary to set the call function pointer using `HPy_SetCallFunction()` in the object's constructor. This procedure is less convenient than just using slot `HPy_tp_cal` but still not hard to use. Consider following example. We define a struct `FooObject` and declare field `HPyCallFunction call_func` which will be used to store the call function's pointer. We need to register the offset of that field with member `__vectorcalloffset__` and in the constructor `Foo_new`, we assign the call function `Foo_call_func`.

```
typedef struct {
    void *a;
    HPyCallFunction call_func;
    void *b;
} FooObject;
HPyType_HELPERS(FooObject)
```

```
HPyDef_MEMBER(Foo_call_func_offset, "__vectorcalloffset__", HPyMember_HPYSSIZET,
                  offsetof(FooObject, call_func), .readonly=1)

HPyDef_CALL_FUNCTION(Foo_call_func)
static HPy
Foo_call_func_impl(HPyContext *ctx, HPy self, const HPy *args, size_t nargs,
                   HPy kwnames)
{
    return HPyUnicode_FromString(ctx,
```

(continues on next page)

(continued from previous page)

```
            "hello manually initialized call function");
}

HPyDef_SLOT(Foo_new, HPy_tp_new)
static HPy Foo_new_impl(HPyContext *ctx, HPy cls, const HPy *args,
        HPy_ssize_t nargs, HPy kw)
{
    FooObject *data;
    HPy h_obj = HPy_New(ctx, cls, &data);
    if (HPy_IsNull(h_obj))
        return HPy_NULL;
    data->call_func = Foo_call_func;
    return h_obj;
}
```

Note: In contrast to CPython's vectorcall protocol, `nargs` will never have flag `PY_VECTORCALL_ARGUMENTS_OFFSET` set. It will **only** be the positional argument count.

Incremental Migration to HPy's Calling Protocol

In order to support incremental migration, HPy provides helper function *HPyHelpers_PackArgsAndKeywords()* that converts from HPy's calling convention to CPython's *tp_call* calling convention. Consider following example:

```
// function using legacy 'tp_call' calling convention
static HPy
Pack_call_legacy(HPyContext *ctx, HPy self, HPy args, HPy kwd)
{
    // use 'args' and 'kwd'
    return HPy_Dup(ctx, ctx->h_None);
}

// function using HPy calling convention
HPyDef_SLOT(Pack_call, HPy_tp_call)
static HPy
Pack_call_impl(HPyContext *ctx, HPy self, const HPy *args, size_t nargs,
          HPy kwnames)
{
    HPy args_tuple, kwd;
    HPy result;
    if (!HPyHelpers_PackArgsAndKeywords(ctx, args, nargs, kwnames,
             &args_tuple, &kwd)) {
        return HPy_NULL;
    }
    result = Pack_call_legacy(ctx, self, args_tuple, kwd);
    HPy_Close(ctx, args_tuple);
    HPy_Close(ctx, kwd);
    return result;
}
```

In this example, `args`, `nargs`, and `kwnames` are used to create a tuple of positional arguments `args_tuple` and a keyword arguments dictionary `kwd`.

2.4.6 PyModule_AddObject

`PyModule_AddObject` is replaced with a regular `HPy_SetAttr_s()`. There is no `HPyModule_AddObject` function because it has an unusual refcount behavior (stealing a reference but only when it returns `0`).

2.4.7 Deallocator slot `Py_tp_dealloc`

`Py_tp_dealloc` essentially becomes `HPy_tp_destroy`. The name intentionally differs because there are major differences: while the slot function of `Py_tp_dealloc` receives the full object (which makes it possible to resurrect it) and while there are no restrictions on what you may call in the C API deallocator, you must not do that in HPy's deallocator.

The two major restrictions apply to the slot function of `HPy_tp_destroy`:

1. The function must be **thread-safe**.
2. The function **must not** call into the interpreter.

The idea is, that `HPy_tp_destroy` just releases native resources (e.g. by using C lib's `free` function). Therefore, it only receives a pointer to the object's native data (and not a handle to the object) and it does not receive an `HPyContext` pointer argument.

For the time being, HPy will support the `HPy_tp_finalize` slot where those tight restrictions do not apply at the (significant) cost of performance.

2.4.8 Special slots `Py_tp_methods`, `Py_tp_members`, and `Py_tp_getset`

There is no direct replacement for C API slots `Py_tp_methods`, `Py_tp_members`, and `Py_tp_getset` because they are no longer needed. Methods, members, and get/set descriptors are specified *flatly* together with the other slots, using the standard mechanisms of `HPyDef_METH`, `HPyDef_MEMBER`, and `HPyDef_GETSET`. The resulting `HPyDef` structures are then accumulated in `HPyType_Spec.defines`.

2.4.9 Creating lists and tuples

The C API way of creating lists and tuples is to create an empty list or tuple object using `PyList_New(n)` or `PyTuple_New(n)`, respectively, and then to fill the empty object using `PyList_SetItem / PyList_SET_ITEM` or `PyTuple_SetItem / PyTuple_SET_ITEM`, respectively.

This is in particular problematic for tuples because they are actually immutable. HPy goes a different way and provides a dedicated *builder* API to avoid the (temporary) inconsistent state during object initialization.

Long story short, doing the same in HPy with builders is still very simple and straight forward. Following an example for creating a list:

```
PyObject *list = PyList_New(5);
if (list == NULL)
    return NULL; /* error */
PyList_SET_ITEM(list, 0, item0);
PyList_SET_ITEM(list, 1, item0);
...
PyList_SET_ITEM(list, 4, item0);
/* now 'list' is ready to use */
```

becomes

```
HPyListBuilder builder = HPyListBuilder_New(ctx, 5);
HPyListBuilder_Set(ctx, builder, 0, h_item0);
HPyListBuilder_Set(ctx, builder, 1, h_item1);
...
HPyListBuilder_Set(ctx, builder, 4, h_item4);
HPy h_list = HPyListBuilder_Build(ctx, builder);
if (HPy_IsNull(h_list))
    return HPy_NULL; /* error */
```

Note: In contrast to `PyList_SetItem`, `PyList_SET_ITEM`, `PyTuple_SetItem`, and `PyTuple_SET_ITEM`, the builder functions *`HPyListBuilder_Set()`* and *`HPyTupleBuilder_Set()`* are **NOT** stealing references. It is necessary to close the passed item handles (e.g. `h_item0` in the above example) if they are no longer needed.

If an error occurs during building the list or tuple, it is necessary to call *`HPyListBuilder_Cancel()`* or *`HPyTupleBuilder_Cancel()`*, respectively, to avoid memory leaks.

For details, see the API reference documentation *Building tuples and lists*.

2.4.10 Buffers

The buffer API in HPy is implemented using the `HPy_buffer` struct, which looks very similar to `Py_buffer` (refer to the CPython documentation for the meaning of the fields):

```
typedef struct {
    void *buf;
    HPy obj;
    HPy_ssize_t len;
    HPy_ssize_t itemsize;
    int readonly;
    int ndim;
    char *format;
    HPy_ssize_t *shape;
    HPy_ssize_t *strides;
    HPy_ssize_t *suboffsets;
    void *internal;
} HPy_buffer;
```

Buffer slots for HPy types are specified using slots `HPy_bf_getbuffer` and `HPy_bf_releasebuffer` on all supported Python versions, even though the matching PyType_Spec slots, `Py_bf_getbuffer` and `Py_bf_releasebuffer`, are only available starting from CPython 3.9.

2.4.11 Multi-phase Module Initialization

HPy supports only multi-phase module initialization (PEP 451). This means that the module object is typically created by interpreter from the `HPyModuleDef` specification and there is no “init” function. However, the module can define one or more `HPy_mod_exec` slots, which will be executed just after the module object is created. Inside the code of those slots, one can usually perform the same initialization as before.

Example of legacy single phase module initialization that uses Python/C API:

```
static struct PyModuleDef mod_def = {
    PyModuleDef_HEAD_INIT,
    .m_name = "legacyinit",
    .m_size = -1
};

PyMODINIT_FUNC
PyInit_legacyinit(void)
{
    PyObject *mod = PyModule_Create(&mod_def);
    if (mod == NULL) return NULL;

    // Some initialization: add types, constants, ...

    return mod;
}
```

The same code structure ported to HPy and multi-phase module initialization:

```
HPyDef_SLOT(my_exec, HPy_mod_exec)
int my_exec_impl(HPyContext *ctx, HPy mod) {

    // Some initialization: add types, constants, ...

    return 0; // success
}

static HPyDef *Methods[] = {
    &my_exec, // HPyDef_SLOT macro generated `my_exec` for us
    NULL,
};

static HPyModuleDef mod_def = {
    .defines = Methods
};

HPy_MODINIT(hpyinit, mod_def)
```

2.5 Porting Example

HPy supports *incrementally* porting an existing C extension from the original Python C API to the HPy API and to have the extension compile and run at each step along the way.

Here we walk through porting a small C extension that implements a Point type with some simple methods (a norm and a dot product). The Point type is minimal, but does contain additional C attributes (the x and y values of the point) and an attribute (obj) that contains a Python object (that we will need to convert from a `PyObject *` to an `HPyField`).

There is a separate C file illustrating each step of the incremental port:

- steps/step_00_c_api: The original C API version that we are going to port.
- steps/step_01_hpy_legacy: A possible first step where all methods still receive `PyObject *` arguments and may still cast them to `PyPointObject *` if they are instances of Point.
- steps/step_02_hpy_legacy: Shows how to transition some methods to HPy methods that receive `HPy` handles as arguments while still supporting legacy methods that receive `PyObject *` arguments.

- steps/step_03_hpy_final: The completed port to HPy where all methods receive `HPy` handles and `PyObject_HEAD` has been removed.

Take a moment to read through steps/step_00_c_api. Then, once you're ready, keep reading.

Each section below corresponds to one of the three porting steps above:

Note: The steps used here are one approach to porting a module. The specific steps are not required. They're just an example approach.

2.5.1 Step 01: Converting the module to a (legacy) HPy module

First for the easy bit – let's include `hpy.h`:

```c
#include <hpy.h>
```

We'd like to differentiate between references to `PyPointObject` that have been ported to HPy and those that haven't, so let's rename it to `PointObject` and alias `PyPointObject` to `PointObject`. We'll keep `PyPointObject` for the instances that haven't been ported yet (the legacy ones) and use `PointObject` where we have ported the references:

```c
typedef struct {
    // PyObject_HEAD is required while legacy_slots are still used
    // but can (and should) be removed once the port to HPy is completed.
    PyObject_HEAD
    double x;
    double y;
    PyObject *obj;
} PointObject;
```

```c
typedef PointObject PyPointObject;
```

For this step, all references will be to `PyPointObject` – we'll only start porting references in the next step.

Let's also call `HPyType_LEGACY_HELPERS` to define some helper functions for use with the `PointObject` struct:

```c
HPyType_LEGACY_HELPERS(PointObject)
```

Again, we won't use these helpers in this step – we're just setting things up for later.

Now for the big steps.

We need to replace `PyType_Spec` for the `Point` type with the equivalent `HPyType_Spec`:

```c
// HPy type methods and slots (no methods or slots have been ported yet)
static HPyDef *point_defines[] = {
    NULL
```

(continues on next page)

(continued from previous page)

```c
};

static HPyType_Spec Point_Type_spec = {
    .name = "point_hpy_legacy_1.Point",
    .basicsize = sizeof(PointObject),
    .itemsize = 0,
    .flags = HPy_TPFLAGS_DEFAULT,
    .builtin_shape = SHAPE(PointObject),
    .legacy_slots = Point_legacy_slots,
    .defines = point_defines,
};

// HPy supports only multiphase module initialization, so we must migrate the
// single phase initialization by extracting the code that populates the module
// object with attributes into a separate 'exec' slot. The module is not
// created manually by calling API like PyModule_Create, but the runtime creates
// the module for us from the specification in HPyModuleDef, and we can provide
// additional slots to populate the module before its initialization is finalized
HPyDef_SLOT(module_exec, HPy_mod_exec)
static int module_exec_impl(HPyContext *ctx, HPy mod)
{
    HPy point_type = HPyType_FromSpec(ctx, &Point_Type_spec, NULL);
    if (HPy_IsNull(point_type))
        return -1;
    HPy_SetAttr_s(ctx, mod, "Point", point_type);
    return 0;
}
```

Initially the list of ported methods in `point_defines` is empty and all of the methods are still in `Point_slots` which we have renamed to `Point_legacy_slots` for clarity.

`SHAPE(PointObject)` is a macro that retrieves the shape of `PointObject` as it was defined by the `HPyType_LEGACY_HELPERS` macro and will be set to `HPyType_BuiltinShape_Legacy` until we replace the legacy macro with the `HPyType_HELPERS` one. Any type with `legacy_slots` or that still includes `PyObject_HEAD` in its struct should have `.builtin_shape` set to `HPyType_BuiltinShape_Legacy`.

Similarly we replace `PyModuleDef` with `HPyModuleDef`:

```c
// Legacy module methods (the "dot" method is still a PyCFunction)
static PyMethodDef PointModuleLegacyMethods[] = {
    {"dot", (PyCFunction)dot, METH_VARARGS, "Dot product."},
    {NULL, NULL, 0, NULL}
};

// HPy module methods: no regular methods have been ported yet,
// but we add the module execute slot
static HPyDef *module_defines[] = {
    &module_exec,
    NULL
};

static HPyModuleDef moduledef = {
    // .name = "step_01_hpy_legacy",
    // ^-- .name is not needed for multiphase module initialization,
    // it is always taken from the ModuleSpec
    .doc = "Point module (Step 1; All legacy methods)",
    .size = 0,
```

(continues on next page)

(continued from previous page)

```c
    .legacy_methods = PointModuleLegacyMethods,
    .defines = module_defines,
};
```

Like the type, the list of ported methods in `module_defines` is initially almost empty: all the regular methods are still in `PointModuleMethods` which has been renamed to `PointModuleLegacyMethods`. However, because HPy supports only multiphase module initialization, we must convert our module initialization code to an "exec" slot on the module and add that slot to `module_defines`.

Now all that is left is to replace the module initialization function with one that uses `HPy_MODINIT`. The first argument is the name of the extension, i.e., what was `XXX` in `PyInit_XXX`, and the second argument is the `HPyModuleDef`.

```c
HPy_MODINIT(step_01_hpy_legacy, moduledef)
```

And we're done!

Instead of the `PyInit_XXX`, we now have an "exec" slot on the module. We implement it with a C function that that takes an `HPyContext *ctx` and `HPy mod` as arguments. The `ctx` must be forwarded as the first argument to calls to HPy API methods. The `mod` argument is a handle for the module object. The runtime creates the module for us from the provided `HPyModuleDef`. There is no need to call API like `PyModule_Create` explicitly.

Next step is to replace `PyType_FromSpec` by `HPyType_FromSpec`.

`HPy_SetAttr_s` is used to add the `Point` class to the module. HPy requires no special `PyModule_AddObject` method.

```c
HPyDef_SLOT(module_exec, HPy_mod_exec)
static int module_exec_impl(HPyContext *ctx, HPy mod)
{
    HPy point_type = HPyType_FromSpec(ctx, &Point_Type_spec, NULL);
    if (HPy_IsNull(point_type))
        return -1;
    HPy_SetAttr_s(ctx, mod, "Point", point_type);
    return 0;
}
```

2.5.2 Step 02: Transition some methods to HPy

In the previous step we put in place the type and module definitions required to create an HPy extension module. In this step we will port some individual methods.

Let us start by migrating `Point_traverse`. First we need to change `PyObject *obj` in the `PointObject` struct to `HPyField obj`:

```c
typedef struct {
    // PyObject_HEAD is required while legacy methods still access
    // PointObject and should be removed once the port to HPy is completed.
    PyObject_HEAD
    double x;
    double y;
    // HPy handles are shortlived to support all GC strategies
    // For that reason, PyObject* in C structs are replaced by HPyField
    HPyField obj;
} PointObject;
```

HPy handles can only be short-lived – i.e. local variables, arguments to functions or return values. HPyField is the way to store long-lived references to Python objects. For more information, please refer to the documentation of *HPyField*.

Now we can update Point_traverse:

```c
HPyDef_SLOT(Point_traverse, HPy_tp_traverse)
int Point_traverse_impl(void *self, HPyFunc_visitproc visit, void *arg)
{
    HPy_VISIT(&((PointObject*)self)->obj);
    return 0;
}
```

In the first line we used the HPyDef_SLOT macro to define a small structure that describes the slot being implemented. The first argument, Point_traverse, is the name to assign the structure to. By convention, the HPyDef_SLOT macro expects a function called Point_traverse_impl implementing the slot. The second argument, HPy_tp_traverse, specifies the kind of slot.

This is a change from how slots are defined in the old C API. In the old API, the kind of slot is only specified much lower down in Point_legacy_slots. In HPy the implementation and kind are defined in one place using a syntax reminiscent of Python decorators.

The implementation of traverse is now a bit simpler than in the old C API. We no longer need to visit Py_TYPE(self) and need only HPy_VISIT self->obj. HPy ensures that interpreter knows that the type of the instance is still referenced.

Only struct members of type HPyField can be visited with HPy_VISIT, which is why we needed to convert obj to an HPyField before we implemented the HPy traverse.

Next we must update Point_init to store the value of obj as an HPyField:

```c
HPyDef_SLOT(Point_init, HPy_tp_init)
int Point_init_impl(HPyContext *ctx, HPy self, const HPy *args,
        HPy_ssize_t nargs, HPy kw)
{
    static const char *kwlist[] = {"x", "y", "obj", NULL};
    PointObject *p = PointObject_AsStruct(ctx, self);
    p->x = 0.0;
    p->y = 0.0;
    HPy obj = HPy_NULL;
    HPyTracker ht;
    if (!HPyArg_ParseKeywordsDict(ctx, &ht, args, nargs, kw, "|ddO", kwlist,
                                  &p->x, &p->y, &obj))
        return -1;
    if (HPy_IsNull(obj))
        obj = ctx->h_None;
    /* INCREF not needed because HPyArg_ParseKeywordsDict does not steal a
       reference */
    HPyField_Store(ctx, self, &p->obj, obj);
    HPyTracker_Close(ctx, ht);
    return 0;
}
```

There are a few new HPy constructs used here:

- The kind of the slot passed to HPyDef_SLOT is HPy_tp_init.
- PointObject_AsStruct is defined by HPyType_LEGACY_HELPERS and returns an instance of the PointObject struct. Because we still include PyObject_HEAD at the start of the struct this is still a

valid `PyObject *` but once we finish the port the struct will no longer contain `PyObject_HEAD` and this will just be an ordinary C struct with no memory overhead!

- We use `HPyTracker` when parsing the arguments with `HPyArg_ParseKeywords`. The `HPyTracker` keeps track of open handles so that they can be closed easily at the end with `HPyTracker_Close`.
- `HPyArg_ParseKeywords` is the equivalent of `PyArg_ParseTupleAndKeywords`. Note that the HPy version does not steal a reference like the Python version.
- `HPyField_Store` is used to store a reference to `obj` in the struct. The arguments are the context (`ctx`), a handle to the object that owns the reference (`self`), the address of the `HPyField` (`&p->obj`), and the handle to the object (`obj`).

Note: An `HPyTracker` is not strictly needed for `HPyArg_ParseKeywords` in `Point_init`. The arguments `x` and `y` are C floats (so there are no handles to close) and the handle stored in `obj` was passed in to the `Point_init` as an argument and so should not be closed.

We showed the tracker here to demonstrate its use. You can read more about argument parsing in the *API docs*.

If a tracker is needed and one is not provided, `HPyArg_ParseKeywords` will return an error.

The last update we need to make for the change to `HPyField` is to migrate `Point_obj_get` which retrieves `obj` from the stored `HPyField`:

```c
HPyDef_GET(Point_obj, "obj", .doc="Associated object.")
HPy Point_obj_get(HPyContext *ctx, HPy self, void* closure)
{
    PointObject *p = PointObject_AsStruct(ctx, self);
    return HPyField_Load(ctx, self, p->obj);
}
```

Above we have used `PointObject_AsStruct` again, and then `HPyField_Load` to retrieve the value of `obj` from the `HPyField`.

We've now finished all of the changes needed by introducing `HPyField`. We could stop here, but let's migrate one ordinary method, `Point_norm`, to end off this stage of the port:

```c
HPyDef_METH(Point_norm, "norm", HPyFunc_NOARGS, .doc="Distance from origin.")
HPy Point_norm_impl(HPyContext *ctx, HPy self)
{
    PointObject *p = PointObject_AsStruct(ctx, self);
    double norm;
    norm = sqrt(p->x * p->x + p->y * p->y);
    return HPyFloat_FromDouble(ctx, norm);
}
```

To define a method we use `HPyDef_METH` instead of `HPyDef_SLOT`. `HPyDef_METH` creates a small structure defining the method. The first argument is the name to assign to the structure (`Point_norm`). The second is the Python name of the method (`norm`). The third specifies the method signature (`HPyFunc_NOARGS` – i.e. no additional arguments in this case). The last provides the docstring. The macro then expects a function named `Point_norm_impl` implementing the method.

The rest of the implementation remains similar, except that we use `HPyFloat_FromDouble` to create a handle to a Python float containing the result (i.e. the distance of the point from the origin).

Now we are done and just have to remove the old implementations from `Point_legacy_slots` and add them to `point_defines`:

```c
static HPyDef *point_defines[] = {
    &Point_init,
    &Point_norm,
    &Point_obj,
    &Point_traverse,
    NULL
};
```

2.5.3 Step 03: Complete the port to HPy

In this step we'll complete the port. We'll no longer include Python, remove `PyObject_HEAD` from the `PointObject` struct, and port the remaining methods.

First, let's remove the import of `Python.h`:

```c
// #include <Python.h>  // disallow use of the old C API
```

And `PyObject_HEAD` from the struct:

```c
typedef struct {
    // PyObject_HEAD is no longer available in PointObject. In CPython,
    // of course, it still exists but is inaccessible from HPy_AsStruct. In
    // other Python implementations (e.g. PyPy) it might no longer exist at
    // all.
    double x;
    double y;
    HPyField obj;
} PointObject;
```

And the typedef of `PointObject` to `PyPointObject`:

```c
// typedef PointObject PyPointObject;
```

Now any code that has not been ported should result in a compilation error.

We must also change the type helpers from `HPyType_LEGACY_HELPERS` to `HPyType_HELPERS` so that `PointObject_AsStruct` knows that `PyObject_HEAD` has been removed:

```c
HPyType_HELPERS(PointObject)
```

There is one more method to port, the `dot` method which is a module method that implements the dot product between two points:

```c
HPyDef_METH(dot, "dot", HPyFunc_VARARGS, .doc="Dot product.")
HPy dot_impl(HPyContext *ctx, HPy self, const HPy *args, size_t nargs)
{
    HPy point1, point2;
    if (!HPyArg_Parse(ctx, NULL, args, nargs, "OO", &point1, &point2))
        return HPy_NULL;
    PointObject *p1 = PointObject_AsStruct(ctx, point1);
    PointObject *p2 = PointObject_AsStruct(ctx, point2);
    double dp;
    dp = p1->x * p2->x + p1->y * p2->y;
    return HPyFloat_FromDouble(ctx, dp);
}
```

The changes are similar to those used in porting the `norm` method, except:

- We use HPyArg_Parse instead of HPyArg_ParseKeywordsDict.
- We opted not to use an HPyTracker by passing NULL as the pointer to the tracker when calling HPyArg_Parse. There is no reason not to use a tracker here, but the handles to the two points are passed in as arguments to dot_impl and thus there is no need to close them (and they should not be closed).

We use PointObject_AsStruct and HPyFloat_FromDouble as before.

Now that we have ported everything we can remove PointMethods, Point_legacy_slots and PointModuleLegacyMethods. The resulting type definition is much cleaner:

```c
static HPyDef *point_defines[] = {
    &Point_init,
    &Point_norm,
    &Point_obj,
    &Point_traverse,
    NULL
};

static HPyType_Spec Point_Type_spec = {
    .name = "point_hpy_final.Point",
    .doc = "Point (Step 03)",
    .basicsize = sizeof(PointObject),
    .itemsize = 0,
    .flags = HPy_TPFLAGS_DEFAULT,
    .defines = point_defines
};

HPyDef_SLOT(module_exec, HPy_mod_exec)
static int module_exec_impl(HPyContext *ctx, HPy mod)
{
    HPy point_type = HPyType_FromSpec(ctx, &Point_Type_spec, NULL);
    if (HPy_IsNull(point_type))
        return -1;
    HPy_SetAttr_s(ctx, mod, "Point", point_type);
    return 0;
}
```

and the module definition is simpler too:

```c
static HPyDef *module_defines[] = {
        &module_exec,
    &dot,
    NULL
};

static HPyModuleDef moduledef = {
    .doc = "Point module (Step 3; Porting complete)",
    .size = 0,
    .defines = module_defines,
};
```

Now that the port is complete, when we compile our extension in HPy universal mode, we obtain a built extension that depends only on the HPy ABI and not on the CPython ABI at all!

2.6 Debug Mode

HPy includes a debug mode which does useful run-time checks to ensure that C extensions use the API correctly. Its features include:

1. No special compilation flags are required: it is enough to compile the extension with the Universal ABI.
2. Debug mode can be activated at *import time*, and it can be activated per-extension.
3. You pay the overhead of debug mode only if you use it. Extensions loaded without the debug mode run at full speed.

This is possible because the whole of the HPy API is provided as part of the HPy context, so debug mode can pass in a special debugging context without affecting the performance of the regular context at all.

Note: The debug mode is only available if the module (you want to use it for) was built for *HPy Universal ABI*.

The debugging context can already check for:

- Leaked handles.
- Handles used after they are closed.
- Tuple and list builder used after they were *closed* (i.e. cancelled or the tuple/list was built).
- Reading from a memory which is no longer guaranteed to be still valid, for example, the buffer returned by `HPyUnicode_AsUTF8AndSize()`, `HPyBytes_AsString()`, and `HPyBytes_AS_STRING()`, after the corresponding `HPy` handle was closed.
- Writing to memory which should be read-only, for example the buffer returned by `HPyUnicode_AsUTF8AndSize()`, `HPyBytes_AsString()`, and `HPyBytes_AS_STRING()`

2.6.1 Activating Debug Mode

Debug mode works *only* for extensions built with HPy universal ABI.

To enable debug mode, use environment variable `HPY`. If `HPY=debug`, then all HPy modules are loaded with the trace context. Alternatively, it is also possible to specify the mode per module like this: `HPY=modA:debug,modB:debug`.

In order to verify that your extension is being loaded in debug mode, use environment variable `HPY_LOG`. If this variable is set, then all HPy extensions built in universal ABI mode print a message when loaded, such as:

```
> import snippets
Loading 'snippets' in HPy universal mode with a debug context
```

If the extension is built in CPython ABI mode, then the `HPY_LOG` environment variable has no effect.

An HPy extension module may be also explicitly loaded in debug mode using:

```
from hpy.universal import load, MODE_DEBUG
mod = load(module_name, so_filename, mode=MODE_DEBUG)
```

When loading HPy extensions explicitly, environment variables `HPY_LOG` and `HPY` have no effect for that extension.

2.6.2 Using Debug Mode

By default, when debug mode detects an error it will either abort the process (using *HPy_FatalError()*) or raise a fatal exception. This may sound very strict but in general, it is not safe to continue the execution.

When testing, aborting the process is unwanted. Module `hpy.debug` exposes the `LeakDetector` class to detect leaked `HPy` handles. For example:

```
def test_leak_detector():
    from hpy.debug.pytest import LeakDetector
    with LeakDetector() as ld:
        # add_ints is an HPy C function. If it forgets to close a handle,
        # LeakDetector will complain
        assert mixed.add_ints(40, 2) == 42
```

Additionally, the debug module also provides a pytest fixture, `hpy_debug`, that for the time being, enables the `LeakDetector`. In the future, it may also enable other useful debugging facilities.

```
from hpy.debug.pytest import hpy_debug
def test_that_uses_leak_detector_fixture(hpy_debug):
    # Run some HPy extension code
```

Warning: The usage of `LeakDetector` or `hpy_debug` by itself does not enable HPy debug mode! If debug mode is not enabled for any extension, then those features have no effect.

When dealing with handle leaks, it is useful to get a stack trace of the allocation of the leaked handle. This feature has large memory requirements and is therefore opt-in. It can be activated by:

```
    hpy.debug.set_handle_stack_trace_limit(16)
```

and disabled by:

```
    hpy.debug.disable_handle_stack_traces()
```

2.6.3 Example

Following HPy function leaks a handle:

```
HPyDef_METH(test_leak_stacktrace, "test_leak_stacktrace", HPyFunc_NOARGS)
static HPy test_leak_stacktrace_impl(HPyContext *ctx, HPy self)
{
    HPy num = HPyLong_FromLong(ctx, 42);
    if (HPy_IsNull(num)) {
        return HPy_NULL;
    }
    // No HPy_Close(ctx, num);
    return HPy_Dup(ctx, ctx->h_None);
}
```

When this script is executed in debug mode:

```
# Run with HPY=debug
import hpy.debug
import snippets
```

(continues on next page)

(continued from previous page)

```
hpy.debug.set_handle_stack_trace_limit(16)
from hpy.debug.pytest import LeakDetector
with LeakDetector() as ld:
    snippets.test_leak_stacktrace()
```

The output is:

```
Traceback (most recent call last):
  File "/path/to/hpy/docs/examples/debug-example.py", line 7, in <module>
    snippets.test_leak_stacktrace()
  File "/path/to/hpy/debug/leakdetector.py", line 43, in __exit__
    self.stop()
  File "/path/to/hpy/debug/leakdetector.py", line 36, in stop
    raise HPyLeakError(leaks)
hpy.debug.leakdetector.HPyLeakError: 1 unclosed handle:
    <DebugHandle 0x556bbcf907c0 for 42>
Allocation stacktrace:
/path/to/site-packages/hpy-0.0.4.dev227+gd7eeec6.d20220510-py3.8-linux-x86_64.egg/hpy/
↪universal.cpython-38d-x86_64-linux-gnu.so(debug_ctx_Long_FromLong+0x45)␣
↪[0x7f1d928c48c4]
/path/to/site-packages/hpy_snippets-0.0.0-py3.8-linux-x86_64.egg/snippets.hpy.
↪so(+0x122c) [0x7f1d921a622c]
/path/to/site-packages/hpy_snippets-0.0.0-py3.8-linux-x86_64.egg/snippets.hpy.
↪so(+0x14b1) [0x7f1d921a64b1]
/path/to/site-packages/hpy-0.0.4.dev227+gd7eeec6.d20220510-py3.8-linux-x86_64.egg/hpy/
↪universal.cpython-38d-x86_64-linux-gnu.so(debug_ctx_
↪CallRealFunctionFromTrampoline+0xca) [0x7f1d928bde1e]
/path/to/site-packages/hpy_snippets-0.0.0-py3.8-linux-x86_64.egg/snippets.hpy.
↪so(+0x129b) [0x7f1d921a629b]
/path/to/site-packages/hpy_snippets-0.0.0-py3.8-linux-x86_64.egg/snippets.hpy.
↪so(+0x1472) [0x7f1d921a6472]
/path/to/libpython3.8d.so.1.0(+0x10a022) [0x7f1d93807022]
/path/to/libpython3.8d.so.1.0(+0x1e986b) [0x7f1d938e686b]
/path/to/libpython3.8d.so.1.0(+0x2015e9) [0x7f1d938fe5e9]
/path/to/libpython3.8d.so.1.0(_PyEval_EvalFrameDefault+0x1008c) [0x7f1d938f875a]
/path/to/libpython3.8d.so.1.0(PyEval_EvalFrameEx+0x64) [0x7f1d938e86b8]
/path/to/libpython3.8d.so.1.0(_PyEval_EvalCodeWithName+0xfaa) [0x7f1d938fc8af]
/path/to/libpython3.8d.so.1.0(PyEval_EvalCodeEx+0x86) [0x7f1d938fca25]
/path/to/libpython3.8d.so.1.0(PyEval_EvalCode+0x4b) [0x7f1d938e862b]
```

For the time being, HPy uses the glibc `backtrace` and `backtrace_symbols` functions. Therefore all their caveats and limitations apply. Usual recommendations to get more symbols in the traces and not only addresses, such as `snippets.hpy.so(+0x122c)`, are:

- link your native code with `-rdynamic` flag (`LDFLAGS="-rdynamic"`)
- build your code without optimizations and with debug symbols (`CFLAGS="-O0 -g"`)
- use `addr2line` command line utility, e.g.: `addr2line -e /path/to/snippets.hpy.so -C -f +0x122c`

2.7 Trace Mode

HPy's trace mode allows you to analyze the usage of the HPy API. The two fundamental metrics are `call count` and `duration`. As the name already suggests, `call count` tells you how often a certain HPy API function was called and `duration` uses a monotonic clock to measure how much (accumulated) time was spent in a certain HPy API function. It is further possible to register custom *on-enter* and *on-exit* Python functions.

As with the debug mode, the trace mode can be activated at *import time*, so no recompilation is required.

2.7.1 Activating Trace Mode

Similar to how the *debug mode is activated*, use environment variable `HPY`. If `HPY=trace`, then all HPy modules are loaded with the trace context. Alternatively, it is also possible to specify the mode per module like this: `HPY=modA:trace,modB:trace`. Environment variable `HPY_LOG` also works.

2.7.2 Using Trace Mode

The trace mode can be accessed via the shipped module `hpy.trace`. It provides following functions:

- `get_call_counts()` returns a dict. The HPy API function names are used as keys and the corresponding call count is the value.
- `get_durations()` also returns a dict similar to `get_call_counts` but the value is the accumulated time spent in the corresponding HPy API function (in nanoseconds). Note, the used clock does not necessarily have a nanosecond resolution which means that the least significant digits may not be accurate.
- `set_trace_functions(on_enter=None, on_exit=None)` allows the user to register custom trace functions. The function provided for `on_enter` and `on_exit` functions will be executed before and after and HPy API function is and was executed, respectively. Passing `None` to any of the two arguments or omitting one will clear the corresponding function.
- `get_frequency()` returns the resolution of the used clock to measure the time in Hertz. For example, a value of `10000000` corresponds to `10 MHz`. In that case, the two least significant digits of the durations are inaccurate.

2.7.3 Example

Following HPy function uses `HPy_Add`:

```c
HPyDef_METH(add, "add", HPyFunc_VARARGS)
static HPy add_impl(HPyContext *ctx, HPy self, const HPy *args, size_t nargs)
{
    if (nargs != 2) {
        HPyErr_SetString(ctx, ctx->h_TypeError, "expected exactly two args");
        return HPy_NULL;
    }
    return HPy_Add(ctx, args[0], args[1]);
}
```

When this script is executed in trace mode:

```python
# Run with HPY=trace
from hpy.trace import get_call_counts
import snippets
```

(continues on next page)

(continued from previous page)

```
add_count_0 = get_call_counts()["ctx_Add"]
snippets.add(1, 2) == 3
add_count_1 = get_call_counts()["ctx_Add"]

print('get_call_counts()["ctx_Add"] == %d' % (add_count_1 - add_count_0))
```

The output is `get_call_counts()["ctx_Add"] == 1`.

2.8 API Reference

HPy's public API consists of three parts:

1. The **Core API** as defined in the *Public API Header*
2. **HPy Helper** functions
3. **Inline Helper** functions

2.8.1 Core API

The **Core API** consists of inline functions that call into the Python interpreter. Those functions will be implemented by each Python interpreter. In *CPython ABI* mode, many of these inline functions will just delegate to a C API functions. In *HPy Universal ABI* mode, they will call a function pointer from the HPy context. This is the source of the performance change between the modes.

HPy Core API Function Index

- `HPyBool_FromBool()`
- `HPyBytes_AS_STRING()`
- `HPyBytes_AsString()`
- `HPyBytes_Check()`
- `HPyBytes_FromString()`
- `HPyBytes_FromStringAndSize()`
- `HPyBytes_GET_SIZE()`
- `HPyBytes_Size()`
- `HPyCallable_Check()`
- `HPyCapsule_Get()`
- `HPyCapsule_IsValid()`
- `HPyCapsule_New()`
- `HPyCapsule_Set()`
- `HPyContextVar_Get()`
- `HPyContextVar_New()`
- `HPyContextVar_Set()`

- *HPyDict_Check()*
- *HPyDict_Copy()*
- *HPyDict_Keys()*
- *HPyDict_New()*
- *HPyErr_Clear()*
- *HPyErr_ExceptionMatches()*
- *HPyErr_NewException()*
- *HPyErr_NewExceptionWithDoc()*
- *HPyErr_NoMemory()*
- *HPyErr_Occurred()*
- *HPyErr_SetFromErrnoWithFilename()*
- *HPyErr_SetFromErrnoWithFilenameObjects()*
- *HPyErr_SetObject()*
- *HPyErr_SetString()*
- *HPyErr_WarnEx()*
- *HPyErr_WriteUnraisable()*
- *HPyField_Load()*
- *HPyField_Store()*
- HPyFloat_AsDouble()
- HPyFloat_FromDouble()
- *HPyGlobal_Load()*
- *HPyGlobal_Store()*
- HPyImport_ImportModule()
- *HPyListBuilder_Build()*
- *HPyListBuilder_Cancel()*
- *HPyListBuilder_New()*
- *HPyListBuilder_Set()*
- HPyList_Append()
- HPyList_Check()
- HPyList_New()
- HPyLong_AsDouble()
- HPyLong_AsInt32_t()
- HPyLong_AsInt64_t()
- HPyLong_AsSize_t()
- HPyLong_AsSsize_t()
- HPyLong_AsUInt32_t()

- HPyLong_AsUInt32_tMask()
- HPyLong_AsUInt64_t()
- HPyLong_AsUInt64_tMask()
- HPyLong_AsVoidPtr()
- HPyLong_FromInt32_t()
- HPyLong_FromInt64_t()
- HPyLong_FromSize_t()
- HPyLong_FromSsize_t()
- HPyLong_FromUInt32_t()
- HPyLong_FromUInt64_t()
- HPyNumber_Check()
- HPySlice_Unpack()
- HPyTracker_Add()
- HPyTracker_Close()
- HPyTracker_ForgetAll()
- HPyTracker_New()
- *HPyTupleBuilder_Build()*
- *HPyTupleBuilder_Cancel()*
- *HPyTupleBuilder_New()*
- *HPyTupleBuilder_Set()*
- HPyTuple_Check()
- HPyTuple_FromArray()
- *HPyType_FromSpec()*
- HPyType_GenericNew()
- *HPyType_GetName()*
- *HPyType_IsSubtype()*
- HPyUnicode_AsASCIIString()
- HPyUnicode_AsLatin1String()
- HPyUnicode_AsUTF8AndSize()
- HPyUnicode_AsUTF8String()
- HPyUnicode_Check()
- HPyUnicode_DecodeASCII()
- HPyUnicode_DecodeFSDefault()
- HPyUnicode_DecodeFSDefaultAndSize()
- HPyUnicode_DecodeLatin1()
- HPyUnicode_EncodeFSDefault()

- HPy_InPlaceAnd()
- HPy_InPlaceFloorDivide()
- HPy_InPlaceLshift()
- HPy_InPlaceMatrixMultiply()
- HPy_InPlaceMultiply()
- HPy_InPlaceOr()
- HPy_InPlacePower()
- HPy_InPlaceRemainder()
- HPy_InPlaceRshift()
- HPy_InPlaceSubtract()
- HPy_InPlaceTrueDivide()
- HPy_InPlaceXor()
- HPy_Index()
- HPy_Invert()
- *HPy_Is()*
- *HPy_IsTrue()*
- *HPy_LeavePythonExecution()*
- HPy_Length()
- HPy_Long()
- HPy_Lshift()
- HPy_MatrixMultiply()
- HPy_Multiply()
- HPy_Negative()
- HPy_Or()
- HPy_Positive()
- HPy_Power()
- *HPy_ReenterPythonExecution()*
- HPy_Remainder()
- *HPy_Repr()*
- *HPy_RichCompare()*
- *HPy_RichCompareBool()*
- HPy_Rshift()
- *HPy_SetAttr()*
- *HPy_SetAttr_s()*
- *HPy_SetCallFunction()*
- *HPy_SetItem()*

- *HPy_SetItem_i()*
- *HPy_SetItem_s()*
- *HPy_Str()*
- HPy_Subtract()
- HPy_TrueDivide()
- *HPy_Type()*
- *HPy_TypeCheck()*
- HPy_Xor()

HPy Context

The HPyContext structure is also part of the API since it provides handles for built-in objects. For a high-level description of the context, please also read *HPyContext*.

struct _HPyContext_s

> **const** char ***name**
>
> int **abi_version**
>
> HPy **h_None**
>
> HPy **h_True**
>
> HPy **h_False**
>
> HPy **h_NotImplemented**
>
> HPy **h_Ellipsis**
>
> HPy **h_BaseException**
>
> HPy **h_Exception**
>
> HPy **h_StopAsyncIteration**
>
> HPy **h_StopIteration**
>
> HPy **h_GeneratorExit**
>
> HPy **h_ArithmeticError**
>
> HPy **h_LookupError**
>
> HPy **h_AssertionError**
>
> HPy **h_AttributeError**
>
> HPy **h_BufferError**
>
> HPy **h_EOFError**
>
> HPy **h_FloatingPointError**
>
> HPy **h_OSError**
>
> HPy **h_ImportError**
>
> HPy **h_ModuleNotFoundError**
>
> HPy **h_IndexError**

HPy **h_KeyError**

HPy **h_KeyboardInterrupt**

HPy **h_MemoryError**

HPy **h_NameError**

HPy **h_OverflowError**

HPy **h_RuntimeError**

HPy **h_RecursionError**

HPy **h_NotImplementedError**

HPy **h_SyntaxError**

HPy **h_IndentationError**

HPy **h_TabError**

HPy **h_ReferenceError**

HPy **h_SystemError**

HPy **h_SystemExit**

HPy **h_TypeError**

HPy **h_UnboundLocalError**

HPy **h_UnicodeError**

HPy **h_UnicodeEncodeError**

HPy **h_UnicodeDecodeError**

HPy **h_UnicodeTranslateError**

HPy **h_ValueError**

HPy **h_ZeroDivisionError**

HPy **h_BlockingIOError**

HPy **h_BrokenPipeError**

HPy **h_ChildProcessError**

HPy **h_ConnectionError**

HPy **h_ConnectionAbortedError**

HPy **h_ConnectionRefusedError**

HPy **h_ConnectionResetError**

HPy **h_FileExistsError**

HPy **h_FileNotFoundError**

HPy **h_InterruptedError**

HPy **h_IsADirectoryError**

HPy **h_NotADirectoryError**

HPy **h_PermissionError**

HPy **h_ProcessLookupError**

HPy **h_TimeoutError**

HPy **h_Warning**

HPy **h_UserWarning**

HPy **h_DeprecationWarning**

HPy **h_PendingDeprecationWarning**

HPy **h_SyntaxWarning**

HPy **h_RuntimeWarning**

HPy **h_FutureWarning**

HPy **h_ImportWarning**

HPy **h_UnicodeWarning**

HPy **h_BytesWarning**

HPy **h_ResourceWarning**

HPy **h_BaseObjectType**

HPy **h_TypeType**

HPy **h_BoolType**

HPy **h_LongType**

HPy **h_FloatType**

HPy **h_UnicodeType**

HPy **h_TupleType**

HPy **h_ListType**

HPy **h_ComplexType**

HPy **h_BytesType**

HPy **h_MemoryViewType**

HPy **h_CapsuleType**

HPy **h_SliceType**

HPy **h_Builtins**

HPy Object

int **HPy_IsTrue** (HPyContext **ctx*, HPy *h*)

HPy **HPy_GetAttr** (HPyContext **ctx*, HPy *obj*, HPy *name*)

HPy **HPy_GetAttr_s** (HPyContext **ctx*, HPy *obj*, **const** char **utf8_name*)

int **HPy_HasAttr** (HPyContext **ctx*, HPy *obj*, HPy *name*)

int **HPy_HasAttr_s** (HPyContext **ctx*, HPy *obj*, **const** char **utf8_name*)

int **HPy_SetAttr** (HPyContext **ctx*, HPy *obj*, HPy *name*, HPy *value*)

int **HPy_SetAttr_s** (HPyContext **ctx*, HPy *obj*, **const** char **utf8_name*, HPy *value*)

HPy **HPy_GetItem** (HPyContext **ctx*, HPy *obj*, HPy *key*)

HPy **`HPy_GetItem_s`** (HPyContext **ctx*, HPy *obj*, **`const`** char **utf8_key*)

HPy **`HPy_GetItem_i`** (HPyContext **ctx*, HPy *obj*, HPy_ssize_t *idx*)

int **`HPy_SetItem`** (HPyContext **ctx*, HPy *obj*, HPy *key*, HPy *value*)

int **`HPy_SetItem_s`** (HPyContext **ctx*, HPy *obj*, **`const`** char **utf8_key*, HPy *value*)

int **`HPy_SetItem_i`** (HPyContext **ctx*, HPy *obj*, HPy_ssize_t *idx*, HPy *value*)

int **`HPy_DelItem`** (HPyContext **ctx*, HPy *obj*, HPy *key*)

int **`HPy_DelItem_s`** (HPyContext **ctx*, HPy *obj*, **`const`** char **utf8_key*)

int **`HPy_DelItem_i`** (HPyContext **ctx*, HPy *obj*, HPy_ssize_t *idx*)

HPy **`HPy_Type`** (HPyContext **ctx*, HPy *obj*)

Returns the type of the given object `obj`.

On failure, raises `SystemError` and returns `HPy_NULL`. This is equivalent to the Python expression``type(obj)``.

Parameters

- **`ctx`** – The execution context.
- **`obj`** – a Python object (must not be `HPy_NULL`)

Returns The type of `obj` or `HPy_NULL` in case of errors.

int **`HPy_TypeCheck`** (HPyContext **ctx*, HPy *obj*, HPy *type*)

Checks if `ob` is an instance of `type` or any subtype of `type`.

Parameters

- **`ctx`** – The execution context.
- **`obj`** – a Python object (must not be `HPy_NULL`)
- **`type`** – A Python type object. This argument must not be `HPy_NULL` and must be a type (i.e. it must inherit from Python `type`). If this is not the case, the behavior is undefined (verification of the argument is only done in debug mode).

Returns Non-zero if object `obj` is an instance of type `type` or an instance of a subtype of `type`, and `0` otherwise.

int **`HPy_Is`** (HPyContext **ctx*, HPy *obj*, HPy *other*)

HPy **`HPy_Repr`** (HPyContext **ctx*, HPy *obj*)

HPy **`HPy_Str`** (HPyContext **ctx*, HPy *obj*)

HPy **`HPy_ASCII`** (HPyContext **ctx*, HPy *obj*)

HPy **`HPy_Bytes`** (HPyContext **ctx*, HPy *obj*)

HPy **`HPy_RichCompare`** (HPyContext **ctx*, HPy *v*, HPy *w*, int *op*)

int **`HPy_RichCompareBool`** (HPyContext **ctx*, HPy *v*, HPy *w*, int *op*)

HPy_hash_t **`HPy_Hash`** (HPyContext **ctx*, HPy *obj*)

int **`HPy_SetCallFunction`** (HPyContext **ctx*, HPy *h*, HPyCallFunction **func*)

Set the call function for the given object.

By defining slot `HPy_tp_call` for some type, instances of this type will be callable objects. The specified call function will be used by default for every instance. This should account for the most common case (every instance of an object uses the same call function) but to still provide the necessary flexibility, function

`HPy_SetCallFunction` allows to set different (maybe specialized) call functions for each instance. This must be done in the constructor of an object.

A more detailed description on how to use that function can be found in section *Calling Protocol*.

Parameters

- **ctx** – The execution context.
- **h** – A handle to an object implementing the call protocol, i.e., the object's type must have slot `HPy_tp_call`. Otherwise, a `TypeError` will be raised. This argument must not be `HPy_NULL`.
- **def** – A pointer to the call function definition to set (must not be `NULL`). The definition is usually created using *`HPyDef_CALL_FUNCTION`*

Returns `0` in case of success and `-1` in case of an error.

HPy Types and Modules

Types, modules and their attributes (i.e. methods, members, slots, get-set descriptors) are defined in a similar way. Section *HPy Type* documents the type-specific and *HPy Module* documents the module-specific part. Section *HPy Definition* documents how to define attributes for both, types and modules.

HPy Type

Definition

struct HPyType_Spec

const char ***name**
The Python name of type (UTF-8 encoded)

int **basicsize**
The size in bytes of the types associated native structure. Usually, you define some C structure, e.g., `typedef struct { int a; } MyObject;`, and then this field is set to `sizeof(MyObject)`.

int **itemsize**
The size of embedded elements (currently not supported).

unsigned long **flags**
Type flags (see *`HPy_TPFLAGS_DEFAULT`*, *`HPy_TPFLAGS_BASETYPE`*, *`HPy_TPFLAGS_HAVE_GC`*, and others if available).

HPyType_BuiltinShape **builtin_shape**
The internal *shape* of the type. The shape gives the necessary hint to compute the offset to the data pointer of the object's underlying struct that should be returned when calling `MyObject_AsStruct`. **ATTENTION**: It is also necessary to specify the right base class in the type's specification parameters (see *`HPyType_SpecParam`*). Assuming that the type's C structure is called `MyObject`, this field should be initialized with `.builtin_shape = SHAPE(MyObject)`. Note: This requires that you use *`HPyType_HELPERS`* or *`HPyType_LEGACY_HELPERS`*. Some more explanation: It would be possible to reduce this information to a Boolean that specifies if the type is a *legacy* type or not. Everything else could be determined by looking at the base classes. However, with this information it is possible to do the data pointer computation statically and thus is performance critical. Types that do not define a struct of their own, should set the value of `.builtin_shape` to the same value as the type they inherit from. If they inherit from a built-in type, they must set the corresponding `.builtin_shape`.

void ***legacy_slots**
: Pointer to a `NULL`-terminated array of legacy (i.e. `PyType_Slot`) slots. A type with `.legacy_slots != NULL` is required to have `HPyType_BuiltinShape_Legacy` and to include `PyObject_HEAD` at the start of its struct. It would be easy to relax this requirement on CPython (where the `PyObject_HEAD` fields are always present) but a large burden on other implementations (e.g. PyPy, GraalPy) where a struct starting with `PyObject_HEAD` might not exist.

int ****defines**
: Pointer to a `NULL`-terminated array of pointers to HPy defines (i.e. `HPyDef *`).

const char ***doc**
: Docstring of the type (UTF-8 encoded; may be `NULL`)

enum HPyType_BuiltinShape

enumerator HPyType_BuiltinShape_Legacy = -1
: A type whose struct starts with `PyObject_HEAD` or equivalent is a legacy type. A legacy type must set `.builtin_shape = HPyType_BuiltinShape_Legacy` in its *HPyType_Spec*. A type is a non-legacy type, also called HPy pure type, if its struct does not include `PyObject_HEAD`. Using pure types should be preferred. Legacy types are available to allow gradual porting of existing CPython extensions. A type with `.legacy_slots != NULL` (see *HPyType_Spec.legacy_slots*) is required to have `HPyType_BuiltinShape_Legacy` and to include `PyObject_HEAD` at the start of its struct. It would be easy to relax this requirement on CPython (where the `PyObject_HEAD` fields are always present) but a large burden on other implementations (e.g. PyPy, GraalPy) where a struct starting with `PyObject_HEAD` might not exist. Types created via the old Python C API are automatically legacy types.

enumerator HPyType_BuiltinShape_Object = 0
: The type inherits from built-in type `object` (default).

enumerator HPyType_BuiltinShape_Type = 1
: The type inherits from built-in type `type`. This can be used to create metaclasses. If using this shape, you need to specify base class `ctx->h_TypeType`.

enumerator HPyType_BuiltinShape_Long = 2
: The type inherits from built-in type `int` (aka. long object). If using this shape, you need to specify base class `ctx->h_LongType`.

enumerator HPyType_BuiltinShape_Float = 3
: The type inherits from built-in type `float`. If using this shape, you need to specify base class `ctx->h_FloatType`.

enumerator HPyType_BuiltinShape_Unicode = 4
: The type inherits from built-in type `str` (aka. unicode object). If using this shape, you need to specify base class `ctx->h_UnicodeType`.

enumerator HPyType_BuiltinShape_Tuple = 5
: The type inherits from built-in type `tuple`. If using this shape, you need to specify base class `ctx->h_TupleType`.

enumerator HPyType_BuiltinShape_List = 6
: The type inherits from built-in type `list`. If using this shape, you need to specify base class `ctx->h_ListType`.

struct HPyType_SpecParam

HPyType_SpecParam_Kind **kind**
: The kind of the type spec param.

int **`object`**

The value of the type spec param (an HPy handle).

`enum HPyType_SpecParam_Kind`

`enumerator HPyType_SpecParam_Base` = 1

Specify a base class. This parameter may be repeated but cannot be used together with *`HPyType_SpecParam_Kind.HPyType_SpecParam_BasesTuple`*.

`enumerator HPyType_SpecParam_BasesTuple` = 2

Specify a tuple of base classes. Cannot be used together with *`HPyType_SpecParam_Kind.HPyType_SpecParam_Base`*

`enumerator HPyType_SpecParam_Metaclass` = 3

Specify a meta class for the type.

`HPyType_HELPERS``()`

A macro for creating (static inline) helper functions for custom types.

Two versions of the helper exist. One for legacy types and one for pure HPy types.

Example for a pure HPy custom type:

```
HPyType_HELPERS(PointObject)
```

It is also possible to inherit from some built-in types. The list of available built-in base types is given in enum *HPyTupe_BuiltinShape*. In case you want to inherit from one of those, it is necessary to specify the base built-in type in the *HPyType_HELPERS* macro. Here is an example for a pure HPy custom type inheriting from a built-in type 'tuple':

```
HPyType_HELPERS(PointObject, HPyType_BuiltinShape_Tuple)
```

This would generate the following:

- `PointObject * PointObject_AsStruct(HPyContext *ctx, HPy h)`: a static inline function that uses HPy_AsStruct to return the PointObject struct associated with a given handle. The behaviour is undefined if *h* is associated with an object that is not an instance of PointObject. However, debug mode will catch an incorrect usage.
- `SHAPE(PointObject)`: a macro that is meant to be used as static initializer in the corresponding HPyType_Spec. It is recommended to write `.builtin_shape = SHAPE(PointObject)` such that you don't have to remember to update the spec when the helpers used changes.

Example for a legacy custom type:

```
HPyType_LEGACY_HELPERS(PointObject)
```

This would generate the same functions and constants as above, except:

- `_HPy_AsStruct_Legacy` is used instead of `_HPy_AsStruct_Object`.
- `SHAPE(PointObject)` would be `HPyType_BuiltinShape_Legacy`.

Parameters

- **`STRUCT`** – The C structure of the HPy type.
- **`SHAPE`** – Optional. The built-in shape of the type. This defaults to *`HPyType_BuiltinShape_Object`*. Possible values are all enumerators of *`HPyType_BuiltinShape`*.

HPyType_LEGACY_HELPERS(*TYPE*)
Convenience macro which is equivalent to: `HPyType_HELPERS(TYPE, HPyType_BuiltinShape_Legacy)`

HPy_TPFLAGS_DEFAULT
Default type flags for HPy types.

HPy_TPFLAGS_BASETYPE
Set if the type allows subclassing

HPy_TPFLAGS_HAVE_GC
If set, the object will be tracked by CPython's GC. Probably irrelevant for GC-based alternative implementations.

Construction and More

HPy **HPyType_FromSpec**(HPyContext **ctx*, *HPyType_Spec* **spec*, *HPyType_SpecParam* **params*)
Create a type from a `HPyType_Spec` and an additional list of specification parameters.

Parameters

- **ctx** – The execution context.
- **spec** – The type spec to use to create the type.
- **params** – A 0-terminated list of type specification parameters or `NULL`.

Returns a handle of the created type on success, `HPy_NULL` on failure.

const char ***HPyType_GetName**(HPyContext **ctx*, HPy *type*)
Return the type's name.

Equivalent to getting the type's `__name__` attribute. If you want to retrieve the type's name as a handle that refers to a `str`, then just use `HPy_GetAttr_s(ctx, type, "__name__")`.

Parameters

- **ctx** – The execution context.
- **type** – A Python type object. This argument must not be `HPy_NULL` and must be a type (i.e. it must inherit from Python `type`). If this is not the case, the behavior is undefined (verification of the argument is only done in debug mode).

Returns The name of the type as C string (UTF-8 encoded) or `NULL` in case of an error. The returned pointer is read-only and guaranteed to be valid as long as the handle `type` is valid.

int **HPyType_IsSubtype**(HPyContext **ctx*, HPy *sub*, HPy *type*)
Checks if `sub` is a subtype of `type`.

This function only checks for actual subtypes, which means that `__subclasscheck__()` is not called on `type`.

Parameters

- **ctx** – The execution context.
- **sub** – A Python type object. This argument must not be `HPy_NULL` and must be a type (i.e. it must inherit from Python `type`). If this is not the case, the behavior is undefined (verification of the argument is only done in debug mode).
- **type** – A Python type object. This argument must not be `HPy_NULL` and must be a type (i.e. it must inherit from Python `type`). If this is not the case, the behavior is undefined (verification of the argument is only done in debug mode).

Returns Non-zero if `sub` is a subtype of `type`.

HPy Module

`HPY_EMBEDDED_MODULES`

If `HPY_EMBEDDED_MODULES` is defined, this means that there will be several embedded HPy modules (and so, several `HPy_MODINIT` usages) in the same binary. In this case, some restrictions apply:

1. all of the module's methods/member/slots/. . . must be defined in the same file
2. the embedder **MUST** declare the module to be *embeddable* by using macro *`HPY_MOD_EMBEDDABLE`*.

`HPY_MOD_EMBEDDABLE`(*modname*)

Declares a module to be *embeddable* which means that it and its members can be compiled/linked into a binary together with other embeddable HPy modules.

You may declare a module to be *embeddable* if all of its member definitions are in the same file.

`struct HPyModuleDef`

Definition of a Python module. Pointer to this struct is returned from the HPy initialization function `HPyInit_{extname}` and the Python interpreter creates a Python module from it. HPy supports only the multi-phase module initialization approach (PEP 451).

There is no HPy API to create a Python module manually, i.e., equivalent of `PyModule_Create` or `PyModule_FromDefAndSpec`, for the time being, but may be added if a use-case arises.

Note: unlike Python/C API, HPy module definition does not specify module name. The name if always taken from the ModuleSpec, which is also the case in multi-phase module initialization on Python/C API.

`const` char ***`doc`**

Docstring of the type (UTF-8 encoded; may be `NULL`)

int **`size`**

The size (in bytes) of the module state structure. If set to zero, then the module will not get allocated and assigned any HPy module state. Negative size, unlike in Python/C API, does not have any specific meaning and will produce a runtime error.

int ***`legacy_methods`**

`NULL`-terminated list of legacy module-level methods. In order to enable incremental migration from C API to HPy, it is possible to still add *legacy* method definitions. Those methods have a C API signature which means that they still receive `PyObject *` and similar arguments. If legacy methods are defined, you cannot create a *universal binary* (i.e. a binary that will run on all Python engines).

int ****`defines`**

Pointer to a `NULL`-terminated array of pointers to HPy defines (i.e. `HPyDef *`). Note, that some kinds of HPy definitions don't make sense for a module. In particular, anything else than methods.

int ****`globals`**

Pointer to a `NULL`-terminated array of pointers to `HPyGlobal` variables. For details, see *HPyGlobal*.

`HPy_MODINIT`(*ext_name*, *mod_def*)

Convenience macro for generating the module initialization code. This will generate three functions that are used by to verify an initialize the module when loading:

`get_required_hpy_major_version_<modname>` The HPy major version this module was built with.

`get_required_hpy_minor_version_<modname>` The HPy minor version this module was built with.

HPyModuleDef* HPyInit_<extname> The init function that will be called by the interpreter. This function does not have an access to HPyContext and thus cannot call any HPy APIs. The purpose of this function is to return a pointer to a HPyModuleDef structure that will serve as a specification of the module that should be created by the interpreter. HPy supports only multi-phase module initialization (PEP 451). Any module initialization code can be added to the HPy_mod_exec slot of the module if needed.

Example:

```
HPy_MODINIT(myextension_shared_library_filename, my_hpy_module_def)
```

HPy Definition

Defining slots, methods, members, and get-set descriptors for types and modules is done with HPy definition (represented by C struct `HPyDef`).

struct HPyDef
Generic structure of an HPy definition.

This struct can be used to define a slot, method, member, or get/set descriptor. For details, see embedded structures `HPySlot`, `HPyMeth`, `HPyMember`, or `HPyGetSet`.

HPyDef_Kind **kind**
The kind of this definition. The value of this field determines which one of the embedded members `slot`, `meth`, `member`, or `getset` is used. Since those are combined in a union, only one can be used at a time.

enum HPyDef_Kind
Enum to identify an HPy definition's kind.

enumerator HPyDef_Kind_Slot

enumerator HPyDef_Kind_Meth

enumerator HPyDef_Kind_Member

enumerator HPyDef_Kind_GetSet

struct HPySlot
C structure to define an HPy slot.

It is perfectly fine to fill this structure manually. However, the recommended and easier way is to use macro `HPyDef_SLOT`.

int **slot**
The slot to fill.

HPyCFunction **impl**
Function pointer to the slot's implementation

int **cpy_trampoline**
Function pointer to the CPython trampoline function which is used by CPython to call the actual HPy function `impl`.

struct HPyMeth
C structure to define an HPy method.

It is perfectly fine to fill this structure manually. However, the recommended and easier way is to use macro `HPyDef_METH`.

const char ***name**
The name of Python attribute (UTF-8 encoded)

HPyCFunction **`impl`**
: Function pointer of the C function implementation

int **`cpy_trampoline`**
: Function pointer to the CPython trampoline function which is used by CPython to call the actual HPy function `impl`.

int **`signature`**
: Indicates the C function's expected signature

`const` char ***`doc`**
: Docstring of the method (UTF-8 encoded; may be `NULL`)

`enum HPyMember_FieldType`
Describes the type (and therefore also the size) of an HPy member.

`enumerator HPyMember_SHORT`

`enumerator HPyMember_INT`

`enumerator HPyMember_LONG`

`enumerator HPyMember_FLOAT`

`enumerator HPyMember_DOUBLE`

`enumerator HPyMember_STRING`

`enumerator HPyMember_OBJECT`

`enumerator HPyMember_CHAR`

`enumerator HPyMember_BYTE`

`enumerator HPyMember_UBYTE`

`enumerator HPyMember_USHORT`

`enumerator HPyMember_UINT`

`enumerator HPyMember_ULONG`

`enumerator HPyMember_STRING_INPLACE`

`enumerator HPyMember_BOOL`

`enumerator HPyMember_OBJECT_EX`

`enumerator HPyMember_LONGLONG`

`enumerator HPyMember_ULONGLONG`

`enumerator HPyMember_HPYSSIZET`

`enumerator HPyMember_NONE`

`struct HPyMember`
C structure to define an HPy member.

It is perfectly fine to fill this structure manually. However, the recommended and easier way is to use macro *`HPyDef_MEMBER`*.

`const` char ***`name`**
: The name of Python attribute (UTF-8 encoded)

HPyMember_FieldType **`type`**
: The type of the HPy member (see enum `HPyMember_FieldType`).

int **offset**
    The location (byte offset) of the member. Usually computed with `offsetof(type, field)`.

int **readonly**
    Flag indicating if the member is read-only

const char ***doc**
    Docstring of the member (UTF-8 encoded; may be `NULL`)

struct HPyGetSet
    C structure to define an HPy get/set descriptor.

    It is perfectly fine to fill this structure manually. However, the recommended and easier way is to use macros *`HPyDef_GET`* (to create a get descriptor only), *`HPyDef_SET`* (to create a set descriptor only), or *`HPyDef_GETSET`* (to create both).

const char ***name**
    The name of Python attribute (UTF-8 encoded)

HPyCFunction **getter_impl**
    Function pointer of the C getter function (may be `NULL`)

HPyCFunction **setter_impl**
    Function pointer of the C setter function (may be `NULL`)

int **getter_cpy_trampoline**
    Function pointer to the CPython trampoline function for the getter (may be `NULL` if (and only if) `getter_impl == NULL`)

int **setter_cpy_trampoline**
    Function pointer to the CPython trampoline function for the setter (may be `NULL` if (and only if) `setter_impl == NULL`)

const char ***doc**
    Docstring of the get/set descriptor (UTF-8 encoded; may be `NULL`)

void ***closure**
    A value that will be passed to the `getter_impl`/`setter_impl` functions.

HPyDef_SLOT(*SYM*, *SLOT*)
    A convenience macro and recommended way to create a definition for an HPy slot.

    The macro generates a C global variable and an appropriate CPython trampoline function. It will fill an *`HPyDef`* structure appropriately and store it in the global variable.

    This macro expects a C function `SYM_impl` that will be used as the implementing slot function.

Parameters

- **SYM** – A C symbol name of the resulting global variable that will contain the generated HPy definition. The variable is defined as `static`.
- **SLOT** – The HPy slot identifier.

HPyDef_METH(*SYM*, *NAME*, *SIG*)
    A convenience macro and recommended way to create a definition for an HPy method.

    The macro generates a C global variable and an appropriate CPython trampoline function. It will fill an *`HPyDef`* structure appropriately and store it in the global variable.

    This macro expects a C function `SYM_impl` that will be used as the implementing C function.

Parameters

- **SYM** – A C symbol name of the resulting global variable that will contain the generated HPy definition. The variable is defined as `static`.
- **NAME** – The Python attribute name (UTF-8 encoded).
- **SIG** – The implementation's C signature (see `HPyFunc_Signature`).

HPyDef_MEMBER(*SYM*, *NAME*, *TYPE*, *OFFSET*)

A convenience macro and recommended way to create a definition for an HPy member.

The macro generates a C global variable. It will fill an *HPyDef* structure appropriately and store it in the global variable.

Parameters

- **SYM** – A C symbol name of the resulting global variable that will contain the generated HPy definition. The variable is defined as `static`.
- **NAME** – The Python attribute name (UTF-8 encoded).
- **TYPE** – The implementation's C signature (see `HPyFunc_Signature`).
- **OFFSET** – The Python attribute name (UTF-8 encoded).
- **.readonly** – Optional flag indicating if the member is read-only.
- **.doc** – Optional docstring (UTF-8 encoded).

HPyDef_GET(*SYM*, *NAME*)

A convenience macro and recommended way to create a definition for an HPy get descriptor.

The macro generates a C global variable. It will fill an *HPyDef* structure appropriately and store it in the global variable.

Parameters

- **SYM** – A C symbol name of the resulting global variable that will contain the generated HPy definition. The variable is defined as `static`.
- **NAME** – The Python attribute name (UTF-8 encoded).
- **.doc** – Optional docstring (UTF-8 encoded).
- **.closure** – Optional pointer, providing additional data for the getter.

HPyDef_SET(*SYM*, *NAME*)

A convenience macro and recommended way to create a definition for an HPy set descriptor.

The macro generates a C global variable. It will fill an *HPyDef* structure appropriately and store it in the global variable.

Parameters

- **SYM** – A C symbol name of the resulting global variable that will contain the generated HPy definition. The variable is defined as `static`.
- **NAME** – The Python attribute name (UTF-8 encoded).
- **.doc** – Optional docstring (UTF-8 encoded).
- **.closure** – Optional pointer, providing additional data for the setter.

HPyDef_GETSET(*SYM*, *NAME*)

A convenience macro and recommended way to create a definition for an HPy get/set descriptor.

The macro generates a C global variable. It will fill an *HPyDef* structure appropriately and store it in the global variable.

Parameters

- **`SYM`** – A C symbol name of the resulting global variable that will contain the generated HPy definition. The variable is defined as `static`.
- **`NAME`** – The Python attribute name (UTF-8 encoded).
- **`.doc`** – Optional docstring (UTF-8 encoded).
- **`.closure`** – Optional pointer, providing additional data for the getter and setter.

`HPyDef_CALL_FUNCTION`(*SYM*)

A convenience macro and the recommended way to create a call function definition.

The macro generates a C global variable with name `SYM`. It will fill an `HPyCallFunction` structure appropriately and store it in the global variable.

This macro expects a C function `SYM_impl` that will be used as the implementing C function.

Parameters

- **`SYM`** – A C symbol name of the resulting global variable that will contain the generated call function definition. The variable is defined as `static`.

HPy Call API

HPy **`HPy_Call`**(HPyContext **ctx*, HPy *callable*, **`const`** HPy **args*, size_t *nargs*, HPy *kwnames*)

Call a Python object.

Parameters

- **`ctx`** – The execution context.
- **`callable`** – A handle to the Python object to call (must not be `HPy_NULL`).
- **`args`** – A pointer to an array of positional and keyword arguments. This argument must not be `NULL` if `nargs > 0` or `HPy_Length(ctx, kwnames) > 0`.
- **`nargs`** – The number of positional arguments in `args`.
- **`kwnames`** – A handle to the tuple of keyword argument names (may be `HPy_NULL`). The values of the keyword arguments are also passed in `args` appended to the positional arguments. Argument `nargs` does not include the keyword argument count.

Returns The result of the call on success, or `HPy_NULL` in case of an error.

HPy **`HPy_CallMethod`**(HPyContext **ctx*, HPy *name*, **`const`** HPy **args*, size_t *nargs*, HPy *kwnames*)

Call a method of a Python object.

Parameters

- **`ctx`** – The execution context.
- **`name`** – A handle to the name (a Unicode object) of the method. Must not be `HPy_NULL`.
- **`args`** – A pointer to an array of the arguments. The receiver is `args[0]`, and the positional and keyword arguments are starting at `args[1]`. This argument must not be `NULL` since a receiver is always required.
- **`nargs`** – The number of positional arguments in `args` including the receiver at `args[0]` (therefore, `nargs` must be at least `1`).
- **`kwnames`** – A handle to the tuple of keyword argument names (may be `HPy_NULL`). The values of the keyword arguments are also passed in `args` appended to the positional arguments. Argument `nargs` does not include the keyword argument count.

Returns The result of the call on success, or `HPy_NULL` in case of an error.

HPy **`HPy_CallTupleDict`** (HPyContext **ctx*, HPy *callable*, HPy *args*, HPy *kw*)
Call a Python object.

Parameters

- **`ctx`** – The execution context.
- **`callable`** – A handle to the Python object to call (must not be `HPy_NULL`).
- **`args`** – A handle to a tuple containing the positional arguments (must not be `HPy_NULL` but can, of course, be empty).
- **`kw`** – A handle to a Python dictionary containing the keyword arguments (may be `HPy_NULL`).

Returns The result of the call on success, or `HPy_NULL` in case of an error.

HPyField

HPy **`HPyField_Load`** (HPyContext **ctx*, HPy *source_object*, HPyField *source_field*)

void **`HPyField_Store`** (HPyContext **ctx*, HPy *target_object*, HPyField **target_field*, HPy *h*)
HPyFields should be used ONLY in parts of memory which is known to the GC, e.g. memory allocated by HPy_New:

- NEVER declare a local variable of type HPyField
- NEVER use HPyField on a struct allocated by e.g. malloc()

CPython's note: contrary to PyObject*, you don't need to manually manage refcounting when using HPyField: if you use HPyField_Store to overwrite an existing value, the old object will be automatically decrefed. This means that you CANNOT use HPyField_Store to write memory which contains uninitialized values, because it would try to decref a dangling pointer.

Note that HPy_New automatically zeroes the memory it allocates, so everything works well out of the box. In case you are using manually allocated memory, you should initialize the HPyField to HPyField_NULL.

Note the difference:

- `obj->f = HPyField_NULL`: this should be used only to initialize uninitialized memory. If you use it to overwrite a valid HPyField, you will cause a memory leak (at least on CPython)
- HPyField_Store(ctx, &obj->f, HPy_NULL): this does the right thing and decref the old value. However, you CANNOT use it if the memory is not initialized.

Note: target_object and source_object are there in case an implementation needs to add write and/or read barriers on the objects. They are ignored by CPython but e.g. PyPy needs a write barrier.

HPyGlobal

void **`HPyGlobal_Store`** (HPyContext **ctx*, HPyGlobal **global*, HPy *h*)
HPyGlobal is an alternative to module state. HPyGlobal must be a statically allocated C global variable registered in HPyModuleDef.globals array. A HPyGlobal can be used only after the HPy module where it is registered was created using HPyModule_Create.

HPyGlobal serves as an identifier of a Python object that should be globally available per one Python interpreter. Python objects referenced by HPyGlobals are destroyed automatically on the interpreter exit (not necessarily the process exit).

HPyGlobal instance does not allow anything else but loading and storing a HPy handle using a HPyContext. Even if the HPyGlobal C variable may be shared between threads or different interpreter instances within one process, the API to load and store a handle from HPyGlobal is thread-safe (but like any other HPy API must not be called in HPy_LeavePythonExecution blocks).

Given that a handle to object X1 is stored to HPyGlobal using HPyContext of Python interpreter I1, then loading a handle from the same HPyGlobal using HPyContext of Python interpreter I1 should give a handle to the same object X1. Another Python interpreter I2 running within the same process and using the same HPyGlobal variable will not be able to load X1 from it, it will have its own view on what is stored in the given HPyGlobal.

Python interpreters may use indirection to isolate different interpreter instances, but alternative techniques such as copy-on-write or immortal objects can be used to avoid that indirection (even selectively on per object basis using tagged pointers).

CPython HPy implementation may even provide configuration option that switches between a faster version that directly stores PyObject* to HPyGlobal but does not support subinterpreters, or a version that supports subinterpreters. For now, CPython HPy always stores PyObject* directly to HPyGlobal.

While the standard implementation does not fully enforce the documented contract, the HPy debug mode will enforce it (not implemented yet).

Implementation notes: All Python interpreters running in one process must be compatible, because they will share all HPyGlobal C level variables. The internal data stored in HPyGlobal are specific for each HPy implementation, each implementation is also responsible for handling thread-safety when initializing the internal data in HPyModule_Create. Note that HPyModule_Create may be called concurrently depending on the semantics of the Python implementation (GIL vs no GIL) and also depending on the whether there may be multiple instances of given Python interpreter running within the same process. In the future, HPy ABI may include a contract that internal data of each HPyGlobal must be initialized to its address using atomic write and HPy implementations will not be free to choose what to store in HPyGlobal, however, this will allow multiple different HPy implementations within one process. This contract may also be activated only by some runtime option, letting the HPy implementation use more optimized HPyGlobal implementation otherwise.

HPy **`HPyGlobal_Load`**(HPyContext **ctx*, HPyGlobal *global*)

HPy Dict

int **`HPyDict_Check`**(HPyContext **ctx*, HPy *h*)

Tests if an object is an instance of a Python dict.

Parameters

- **`ctx`** – The execution context.
- **`h`** – A handle to an arbitrary object (must not be `HPy_NULL`).

Returns Non-zero if object `h` is an instance of type `dict` or an instance of a subtype of `dict`, and `0` otherwise.

HPy **`HPyDict_New`**(HPyContext **ctx*)

Creates a new empty Python dictionary.

Parameters

- **`ctx`** – The execution context.

Returns A handle to the new and empty Python dictionary or `HPy_NULL` in case of an error.

HPy **`HPyDict_Keys`**(HPyContext **ctx*, HPy *h*)

Returns a list of all keys from the dictionary.

Note: This function will directly access the storage of the dict object and therefore ignores if method `keys` was overwritten.

Parameters

- **`ctx`** – The execution context.
- **`h`** – A Python dict object. If this argument is `HPy_NULL` or not an instance of a Python dict, a `SystemError` will be raised.

Returns A Python list object containing all keys of the given dictionary or `HPy_NULL` in case of an error.

HPy **`HPyDict_Copy`**(HPyContext **ctx*, HPy *h*)

Creates a copy of the provided Python dict object.

Parameters

- **`ctx`** – The execution context.
- **`h`** – A Python dict object. If this argument is `HPy_NULL` or not an instance of a Python dict, a `SystemError` will be raised.

Returns Return a new dictionary that contains the same key-value pairs as `h` or `HPy_NULL` in case of an error.

Leave/enter Python execution (GIL)

HPyThreadState **`HPy_LeavePythonExecution`**(HPyContext **ctx*)

void **`HPy_ReenterPythonExecution`**(HPyContext **ctx*, HPyThreadState *state*)

Leaving Python execution: for releasing GIL and other use-cases.

In most situations, users should prefer using convenience macros: HPy_BEGIN_LEAVE_PYTHON(context)/HPy_END_LEAVE_PYTHON(context)

HPy extensions may leave Python execution when running Python independent code: long-running computations or blocking operations. When an extension has left the Python execution it must not call any HPy API other than HPy_ReenterPythonExecution. It can access pointers returned by HPy API, e.g., HPyUnicode_AsUTF8String, provided that they are valid at the point of calling HPy_LeavePythonExecution.

Python execution must be reentered on the same thread as where it was left. The leave/enter calls must not be nested. Debug mode will, in the future, enforce these constraints.

Python implementations may use this knowledge however they wish. The most obvious use case is to release the GIL, in which case the HPy_BEGIN_LEAVE_PYTHON/HPy_END_LEAVE_PYTHON becomes equivalent to Py_BEGIN_ALLOW_THREADS/Py_END_ALLOW_THREADS.

Exception Handling

HPy **`HPyErr_SetFromErrnoWithFilename`**(HPyContext **ctx*, HPy *h_type*, **`const`** char **filename_fsencoded*)

Similar to *`HPyErr_SetFromErrnoWithFilenameObjects()`* but takes one filename (a C string) that will be decoded using `HPyUnicode_DecodeFSDefault()`.

Parameters

- **`ctx`** – The execution context.
- **`h_type`** – The exception type to raise.
- **`filename_fsencoded`** – a filename; may be `NULL`

Returns always returns `HPy_NULL`

HPy **`HPyErr_SetFromErrnoWithFilenameObjects`**(HPyContext **ctx*, HPy *h_type*, HPy *filename1*, HPy *filename2*)

A convenience function to raise an exception when a C library function has returned an error and set the C variable `errno`. It constructs an instance of the provided exception type `h_type` by calling `h_type(errno, strerror(errno), filename1, 0, filename2)`. The exception instance is then raised.

Parameters

- **`ctx`** – The execution context.
- **`h_type`** – The exception type to raise.
- **`filename1`** – A filename; may be `HPy_NULL`. In the case of `h_type` is the `OSError` exception, this is used to define the filename attribute of the exception instance.
- **`filename2`** – another filename argument; may be `HPy_NULL`

Returns always returns `HPy_NULL`

void **`HPy_FatalError`**(HPyContext **ctx*, **const** char **message*)

HPy **`HPyErr_SetString`**(HPyContext **ctx*, HPy *h_type*, **const** char **utf8_message*)

HPy **`HPyErr_SetObject`**(HPyContext **ctx*, HPy *h_type*, HPy *h_value*)

int **`HPyErr_Occurred`**(HPyContext **ctx*)

int **`HPyErr_ExceptionMatches`**(HPyContext **ctx*, HPy *exc*)

HPy **`HPyErr_NoMemory`**(HPyContext **ctx*)

void **`HPyErr_Clear`**(HPyContext **ctx*)

HPy **`HPyErr_NewException`**(HPyContext **ctx*, **const** char **utf8_name*, HPy *base*, HPy *dict*)

HPy **`HPyErr_NewExceptionWithDoc`**(HPyContext **ctx*, **const** char **utf8_name*, **const** char **utf8_doc*, HPy *base*, HPy *dict*)

int **`HPyErr_WarnEx`**(HPyContext **ctx*, HPy *category*, **const** char **utf8_message*, HPy_ssize_t *stack_level*)

void **`HPyErr_WriteUnraisable`**(HPyContext **ctx*, HPy *obj*)

Building tuples and lists

HPyTupleBuilder **`HPyTupleBuilder_New`**(HPyContext **ctx*, HPy_ssize_t *size*)

Create a new tuple builder for `size` elements. The builder is then able to take at most `size` elements. This function does not raise any exception (even if running out of memory).

Parameters

- **`ctx`** – The execution context.
- **`size`** – The number of elements to hold.

void **`HPyTupleBuilder_Set`**(HPyContext **ctx*, HPyTupleBuilder *builder*, HPy_ssize_t *index*, HPy *h_item*)

Assign an element to a certain index of the builder. Valid indices are in range `0 <= index < size` where `size` is the value passed to *`HPyTupleBuilder_New()`*. This function does not raise * any exception.

Parameters

- **`ctx`** – The execution context.
- **`builder`** – A tuple builder handle.

- **`index`** – The index to assign the object to.
- **`h_item`** – An HPy handle of the object to store or `HPy_NULL`. Please note that HPy **never** steals handles and so, `h_item` needs to be closed by the caller.

HPy **`HPyTupleBuilder_Build`**(HPyContext **ctx*, HPyTupleBuilder *builder*)

Build a tuple from a tuple builder.

Parameters

- **`ctx`** – The execution context.
- **`builder`** – A tuple builder handle.

Returns An HPy handle to a tuple containing the values inserted with *`HPyTupleBuilder_Set()`* or `HPy_NULL` in case an error occurred during building or earlier when creating the builder or setting the items.

void **`HPyTupleBuilder_Cancel`**(HPyContext **ctx*, HPyTupleBuilder *builder*)

Cancel building of a tuple and free any acquired resources. This function ignores if any error occurred previously when using the tuple builder.

Parameters

- **`ctx`** – The execution context.
- **`builder`** – A tuple builder handle.

HPyListBuilder **`HPyListBuilder_New`**(HPyContext **ctx*, HPy_ssize_t *size*)

Create a new list builder for `size` elements. The builder is then able to take at most `size` elements. This function does not raise any exception (even if running out of memory).

Parameters

- **`ctx`** – The execution context.
- **`size`** – The number of elements to hold.

void **`HPyListBuilder_Set`**(HPyContext **ctx*, HPyListBuilder *builder*, HPy_ssize_t *index*, HPy *h_item*)

Assign an element to a certain index of the builder. Valid indices are in range `0 <= index < size` where `size` is the value passed to *`HPyListBuilder_New()`*. This function does not raise any exception.

Parameters

- **`ctx`** – The execution context.
- **`builder`** – A list builder handle.
- **`index`** – The index to assign the object to.
- **`h_item`** – An HPy handle of the object to store or `HPy_NULL`. Please note that HPy **never** steals handles and so, `h_item` needs to be closed by the caller.

HPy **`HPyListBuilder_Build`**(HPyContext **ctx*, HPyListBuilder *builder*)

Build a list from a list builder.

Parameters

- **`ctx`** – The execution context.
- **`builder`** – A list builder handle.

Returns An HPy handle to a list containing the values inserted with *`HPyListBuilder_Set()`* or `HPy_NULL` in case an error occurred during building or earlier when creating the builder or setting the items.

void **HPyListBuilder_Cancel** (HPyContext **ctx*, HPyListBuilder *builder*)

Cancel building of a tuple and free any acquired resources. This function ignores if any error occurred previously when using the tuple builder.

Parameters

- **ctx** – The execution context.
- **builder** – A tuple builder handle.

HPy Eval

enum HPy_SourceKind

An enumeration of the different kinds of source code strings.

enumerator HPy_SourceKind_Expr = 0

Parse isolated expressions (e.g. `a + b`).

enumerator HPy_SourceKind_File = 1

Parse sequences of statements as read from a file or other source. This is the symbol to use when compiling arbitrarily long Python source code.

enumerator HPy_SourceKind_Single = 2

Parse a single statement. This is the mode used for the interactive interpreter loop.

HPy **HPy_Compile_s** (HPyContext **ctx*, **const** char **utf8_source*, **const** char **utf8_filename*, *HPy_SourceKind* *kind*)

Parse and compile the Python source code.

Parameters

- **ctx** – The execution context.
- **utf8_source** – Python source code given as UTF-8 encoded C string (must not be `NULL`).
- **utf8_filename** – The filename (UTF-8 encoded C string) to use for construction of the code object. It may appear in tracebacks or in `SyntaxError` exception messages.
- **kind** – The source kind which tells the parser if a single expression, statement, or a whole file should be parsed (see enum `HPy_SourceKind`).

Returns A Python code object resulting from the parsed and compiled Python source code or `HPy_NULL` in case of errors.

HPy **HPy_EvalCode** (HPyContext **ctx*, HPy *code*, HPy *globals*, HPy *locals*)

Evaluate a precompiled code object.

Code objects can be compiled from a string using `HPy_Compile_s()`.

Parameters

- **ctx** – The execution context.
- **code** – The code object to evaluate.
- **globals** – A Python dictionary defining the global variables for the evaluation.
- **locals** – A mapping object defining the local variables for the evaluation.

Returns The result produced by the executed code. May be `HPy_NULL` in case of errors.

Public API Header

The core API is defined in public_api.h:

```
/* HPy public API */

/*
 * IMPORTANT: In order to ensure backwards compatibility of HPyContext, it is
 * necessary to define the order of the context members. To do so, use macro
 * 'HPy_ID(idx)' for context handles and functions. When adding members, it
 * doesn't matter where they are located in this file. It's just important that
 * the maximum context index is incremented by exactly one.
 */

#ifdef AUTOGEN

/* Constants */
HPy_ID(0) HPy h_None;
HPy_ID(1) HPy h_True;
HPy_ID(2) HPy h_False;
HPy_ID(3) HPy h_NotImplemented;
HPy_ID(4) HPy h_Ellipsis;

/* Exceptions */
HPy_ID(5) HPy h_BaseException;
HPy_ID(6) HPy h_Exception;
HPy_ID(7) HPy h_StopAsyncIteration;
HPy_ID(8) HPy h_StopIteration;
HPy_ID(9) HPy h_GeneratorExit;
HPy_ID(10) HPy h_ArithmeticError;
HPy_ID(11) HPy h_LookupError;
HPy_ID(12) HPy h_AssertionError;
HPy_ID(13) HPy h_AttributeError;
HPy_ID(14) HPy h_BufferError;
HPy_ID(15) HPy h_EOFError;
HPy_ID(16) HPy h_FloatingPointError;
HPy_ID(17) HPy h_OSError;
HPy_ID(18) HPy h_ImportError;
HPy_ID(19) HPy h_ModuleNotFoundError;
HPy_ID(20) HPy h_IndexError;
HPy_ID(21) HPy h_KeyError;
HPy_ID(22) HPy h_KeyboardInterrupt;
HPy_ID(23) HPy h_MemoryError;
HPy_ID(24) HPy h_NameError;
HPy_ID(25) HPy h_OverflowError;
HPy_ID(26) HPy h_RuntimeError;
HPy_ID(27) HPy h_RecursionError;
HPy_ID(28) HPy h_NotImplementedError;
HPy_ID(29) HPy h_SyntaxError;
HPy_ID(30) HPy h_IndentationError;
HPy_ID(31) HPy h_TabError;
HPy_ID(32) HPy h_ReferenceError;
HPy_ID(33) HPy h_SystemError;
HPy_ID(34) HPy h_SystemExit;
HPy_ID(35) HPy h_TypeError;
HPy_ID(36) HPy h_UnboundLocalError;
HPy_ID(37) HPy h_UnicodeError;
HPy_ID(38) HPy h_UnicodeEncodeError;
```

(continues on next page)

(continued from previous page)

```
HPy_ID(39) HPy h_UnicodeDecodeError;
HPy_ID(40) HPy h_UnicodeTranslateError;
HPy_ID(41) HPy h_ValueError;
HPy_ID(42) HPy h_ZeroDivisionError;
HPy_ID(43) HPy h_BlockingIOError;
HPy_ID(44) HPy h_BrokenPipeError;
HPy_ID(45) HPy h_ChildProcessError;
HPy_ID(46) HPy h_ConnectionError;
HPy_ID(47) HPy h_ConnectionAbortedError;
HPy_ID(48) HPy h_ConnectionRefusedError;
HPy_ID(49) HPy h_ConnectionResetError;
HPy_ID(50) HPy h_FileExistsError;
HPy_ID(51) HPy h_FileNotFoundError;
HPy_ID(52) HPy h_InterruptedError;
HPy_ID(53) HPy h_IsADirectoryError;
HPy_ID(54) HPy h_NotADirectoryError;
HPy_ID(55) HPy h_PermissionError;
HPy_ID(56) HPy h_ProcessLookupError;
HPy_ID(57) HPy h_TimeoutError;
// EnvironmentError, IOError and WindowsError are intentionally omitted (they
// are all aliases of OSError since Python 3.3).

/* Warnings */
HPy_ID(58) HPy h_Warning;
HPy_ID(59) HPy h_UserWarning;
HPy_ID(60) HPy h_DeprecationWarning;
HPy_ID(61) HPy h_PendingDeprecationWarning;
HPy_ID(62) HPy h_SyntaxWarning;
HPy_ID(63) HPy h_RuntimeWarning;
HPy_ID(64) HPy h_FutureWarning;
HPy_ID(65) HPy h_ImportWarning;
HPy_ID(66) HPy h_UnicodeWarning;
HPy_ID(67) HPy h_BytesWarning;
HPy_ID(68) HPy h_ResourceWarning;

/* Types */
HPy_ID(69) HPy h_BaseObjectType;   /* built-in 'object' */
HPy_ID(70) HPy h_TypeType;         /* built-in 'type' */
HPy_ID(71) HPy h_BoolType;         /* built-in 'bool' */
HPy_ID(72) HPy h_LongType;         /* built-in 'int' */
HPy_ID(73) HPy h_FloatType;        /* built-in 'float' */
HPy_ID(74) HPy h_UnicodeType;      /* built-in 'str' */
HPy_ID(75) HPy h_TupleType;        /* built-in 'tuple' */
HPy_ID(76) HPy h_ListType;         /* built-in 'list' */
HPy_ID(238) HPy h_ComplexType;     /* built-in 'complex' */
HPy_ID(239) HPy h_BytesType;       /* built-in 'bytes' */
HPy_ID(240) HPy h_MemoryViewType;  /* built-in 'memoryview' */
HPy_ID(241) HPy h_CapsuleType;     /* built-in 'capsule' */
HPy_ID(242) HPy h_SliceType;       /* built-in 'slice' */

/* Reflection */
HPy_ID(243) HPy h_Builtins;        /* dict of builtins */

#endif

HPy_ID(77)
HPy HPy_Dup(HPyContext *ctx, HPy h);
```

(continues on next page)

(continued from previous page)

```
HPy_ID(78)
void HPy_Close(HPyContext *ctx, HPy h);

HPy_ID(79)
HPy HPyLong_FromInt32_t(HPyContext *ctx, int32_t value);
HPy_ID(80)
HPy HPyLong_FromUInt32_t(HPyContext *ctx, uint32_t value);
HPy_ID(81)
HPy HPyLong_FromInt64_t(HPyContext *ctx, int64_t v);
HPy_ID(82)
HPy HPyLong_FromUInt64_t(HPyContext *ctx, uint64_t v);
HPy_ID(83)
HPy HPyLong_FromSize_t(HPyContext *ctx, size_t value);
HPy_ID(84)
HPy HPyLong_FromSsize_t(HPyContext *ctx, HPy_ssize_t value);

HPy_ID(85)
int32_t HPyLong_AsInt32_t(HPyContext *ctx, HPy h);
HPy_ID(86)
uint32_t HPyLong_AsUInt32_t(HPyContext *ctx, HPy h);
HPy_ID(87)
uint32_t HPyLong_AsUInt32_tMask(HPyContext *ctx, HPy h);
HPy_ID(88)
int64_t HPyLong_AsInt64_t(HPyContext *ctx, HPy h);
HPy_ID(89)
uint64_t HPyLong_AsUInt64_t(HPyContext *ctx, HPy h);
HPy_ID(90)
uint64_t HPyLong_AsUInt64_tMask(HPyContext *ctx, HPy h);
HPy_ID(91)
size_t HPyLong_AsSize_t(HPyContext *ctx, HPy h);
HPy_ID(92)
HPy_ssize_t HPyLong_AsSsize_t(HPyContext *ctx, HPy h);
HPy_ID(93)
void* HPyLong_AsVoidPtr(HPyContext *ctx, HPy h);
HPy_ID(94)
double HPyLong_AsDouble(HPyContext *ctx, HPy h);

HPy_ID(95)
HPy HPyFloat_FromDouble(HPyContext *ctx, double v);
HPy_ID(96)
double HPyFloat_AsDouble(HPyContext *ctx, HPy h);

HPy_ID(97)
HPy HPyBool_FromBool(HPyContext *ctx, bool v);


/* abstract.h */
HPy_ID(98)
HPy_ssize_t HPy_Length(HPyContext *ctx, HPy h);

HPy_ID(99)
int HPyNumber_Check(HPyContext *ctx, HPy h);
HPy_ID(100)
HPy HPy_Add(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(101)
HPy HPy_Subtract(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(102)
```

(continues on next page)

(continued from previous page)

```
HPy HPy_Multiply(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(103)
HPy HPy_MatrixMultiply(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(104)
HPy HPy_FloorDivide(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(105)
HPy HPy_TrueDivide(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(106)
HPy HPy_Remainder(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(107)
HPy HPy_Divmod(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(108)
HPy HPy_Power(HPyContext *ctx, HPy h1, HPy h2, HPy h3);
HPy_ID(109)
HPy HPy_Negative(HPyContext *ctx, HPy h1);
HPy_ID(110)
HPy HPy_Positive(HPyContext *ctx, HPy h1);
HPy_ID(111)
HPy HPy_Absolute(HPyContext *ctx, HPy h1);
HPy_ID(112)
HPy HPy_Invert(HPyContext *ctx, HPy h1);
HPy_ID(113)
HPy HPy_Lshift(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(114)
HPy HPy_Rshift(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(115)
HPy HPy_And(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(116)
HPy HPy_Xor(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(117)
HPy HPy_Or(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(118)
HPy HPy_Index(HPyContext *ctx, HPy h1);
HPy_ID(119)
HPy HPy_Long(HPyContext *ctx, HPy h1);
HPy_ID(120)
HPy HPy_Float(HPyContext *ctx, HPy h1);

HPy_ID(121)
HPy HPy_InPlaceAdd(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(122)
HPy HPy_InPlaceSubtract(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(123)
HPy HPy_InPlaceMultiply(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(124)
HPy HPy_InPlaceMatrixMultiply(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(125)
HPy HPy_InPlaceFloorDivide(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(126)
HPy HPy_InPlaceTrueDivide(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(127)
HPy HPy_InPlaceRemainder(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(128)
HPy HPy_InPlacePower(HPyContext *ctx, HPy h1, HPy h2, HPy h3);
HPy_ID(129)
HPy HPy_InPlaceLshift(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(130)
```

(continues on next page)

(continued from previous page)

```
HPy HPy_InPlaceRshift(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(131)
HPy HPy_InPlaceAnd(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(132)
HPy HPy_InPlaceXor(HPyContext *ctx, HPy h1, HPy h2);
HPy_ID(133)
HPy HPy_InPlaceOr(HPyContext *ctx, HPy h1, HPy h2);

HPy_ID(134)
int HPyCallable_Check(HPyContext *ctx, HPy h);

/**
 * Call a Python object.
 *
 * :param ctx:
 *     The execution context.
 * :param callable:
 *     A handle to the Python object to call (must not be ``HPy_NULL``).
 * :param args:
 *     A handle to a tuple containing the positional arguments (must not be
 *     ``HPy_NULL`` but can, of course, be empty).
 * :param kw:
 *     A handle to a Python dictionary containing the keyword arguments (may be
 *     ``HPy_NULL``).
 *
 * :returns:
 *     The result of the call on success, or ``HPy_NULL`` in case of an error.
 */
HPy_ID(135)
HPy HPy_CallTupleDict(HPyContext *ctx, HPy callable, HPy args, HPy kw);

/**
 * Call a Python object.
 *
 * :param ctx:
 *     The execution context.
 * :param callable:
 *     A handle to the Python object to call (must not be ``HPy_NULL``).
 * :param args:
 *     A pointer to an array of positional and keyword arguments. This argument
 *     must not be ``NULL`` if ``nargs > 0`` or
 *     ``HPy_Length(ctx, kwnames) > 0``.
 * :param nargs:
 *     The number of positional arguments in ``args``.
 * :param kwnames:
 *     A handle to the tuple of keyword argument names (may be ``HPy_NULL``).
 *     The values of the keyword arguments are also passed in ``args`` appended
 *     to the positional arguments. Argument ``nargs`` does not include the
 *     keyword argument count.
 *
 * :returns:
 *     The result of the call on success, or ``HPy_NULL`` in case of an error.
 */
HPy_ID(261)
HPy HPy_Call(HPyContext *ctx, HPy callable, const HPy *args, size_t nargs, HPy␣
↪kwnames);

```

(continues on next page)

(continued from previous page)

```
/**
 * Call a method of a Python object.
 *
 * :param ctx:
 *     The execution context.
 * :param name:
 *     A handle to the name (a Unicode object) of the method. Must not be
 *     ``HPy_NULL``.
 * :param args:
 *     A pointer to an array of the arguments. The receiver is ``args[0]``, and
 *     the positional and keyword arguments are starting at ``args[1]``. This
 *     argument must not be ``NULL`` since a receiver is always required.
 * :param nargs:
 *     The number of positional arguments in ``args`` including the receiver at
 *     ``args[0]`` (therefore, ``nargs`` must be at least ``1``).
 * :param kwnames:
 *     A handle to the tuple of keyword argument names (may be ``HPy_NULL``).
 *     The values of the keyword arguments are also passed in ``args`` appended
 *     to the positional arguments. Argument ``nargs`` does not include the
 *     keyword argument count.
 *
 * :returns:
 *     The result of the call on success, or ``HPy_NULL`` in case of an error.
 */
HPy_ID(262)
HPy HPy_CallMethod(HPyContext *ctx, HPy name, const HPy *args, size_t nargs, HPy␣
↪kwnames);

/* pyerrors.h */
HPy_ID(136)
void HPy_FatalError(HPyContext *ctx, const char *message);
HPy_ID(137)
HPy HPyErr_SetString(HPyContext *ctx, HPy h_type, const char *utf8_message);
HPy_ID(138)
HPy HPyErr_SetObject(HPyContext *ctx, HPy h_type, HPy h_value);

/**
 * Similar to :c:func:`HPyErr_SetFromErrnoWithFilenameObjects` but takes one
 * filename (a C string) that will be decoded using
 * :c:func:`HPyUnicode_DecodeFSDefault`.
 *
 * :param ctx:
 *     The execution context.
 * :param h_type:
 *     The exception type to raise.
 * :param filename_fsencoded:
 *     a filename; may be ``NULL``
 *
 * :return:
 *     always returns ``HPy_NULL``
 */
HPy_ID(139)
HPy HPyErr_SetFromErrnoWithFilename(HPyContext *ctx, HPy h_type, const char *filename_
↪fsencoded);

/**
 * A convenience function to raise an exception when a C library function has
```

(continues on next page)

(continued from previous page)

```
 * returned an error and set the C variable ``errno``. It constructs an
 * instance of the provided exception type ``h_type`` by calling
 * ``h_type(errno, strerror(errno), filename1, 0, filename2)``. The exception
 * instance is then raised.
 *
 * :param ctx:
 *     The execution context.
 * :param h_type:
 *     The exception type to raise.
 * :param filename1:
 *     A filename; may be ``HPy_NULL``. In the case of ``h_type`` is the
 *     ``OSError`` exception, this is used to define the filename attribute of
 *     the exception instance.
 * :param filename2:
 *     another filename argument; may be ``HPy_NULL``
 *
 * :return:
 *     always returns ``HPy_NULL``
 */
HPy_ID(140)
HPy HPyErr_SetFromErrnoWithFilenameObjects(HPyContext *ctx, HPy h_type, HPy filename1,
↪ HPy filename2);
/* note: HPyErr_Occurred() returns a flag 0-or-1, instead of a 'PyObject *' */
HPy_ID(141)
int HPyErr_Occurred(HPyContext *ctx);
HPy_ID(142)
int HPyErr_ExceptionMatches(HPyContext *ctx, HPy exc);
HPy_ID(143)
HPy HPyErr_NoMemory(HPyContext *ctx);
HPy_ID(144)
void HPyErr_Clear(HPyContext *ctx);
HPy_ID(145)
HPy HPyErr_NewException(HPyContext *ctx, const char *utf8_name, HPy base, HPy dict);
HPy_ID(146)
HPy HPyErr_NewExceptionWithDoc(HPyContext *ctx, const char *utf8_name, const char␣
↪*utf8_doc, HPy base, HPy dict);
HPy_ID(147)
int HPyErr_WarnEx(HPyContext *ctx, HPy category, const char *utf8_message, HPy_ssize_
↪t stack_level);
HPy_ID(148)
void HPyErr_WriteUnraisable(HPyContext *ctx, HPy obj);

/* object.h */
HPy_ID(149)
int HPy_IsTrue(HPyContext *ctx, HPy h);

/**
 * Create a type from a :c:struct:`HPyType_Spec` and an additional list of
 * specification parameters.
 *
 * :param ctx:
 *     The execution context.
 * :param spec:
 *     The type spec to use to create the type.
 * :param params:
 *     A 0-terminated list of type specification parameters or ``NULL``.
 *
```

(continues on next page)

(continued from previous page)

```
 * :returns: a handle of the created type on success, ``HPy_NULL`` on failure.
 */
HPy_ID(150)
HPy HPyType_FromSpec(HPyContext *ctx, HPyType_Spec *spec,
                     HPyType_SpecParam *params);
HPy_ID(151)
HPy HPyType_GenericNew(HPyContext *ctx, HPy type, const HPy *args, HPy_ssize_t nargs,␣
↪HPy kw);

HPy_ID(152)
HPy HPy_GetAttr(HPyContext *ctx, HPy obj, HPy name);
HPy_ID(153)
HPy HPy_GetAttr_s(HPyContext *ctx, HPy obj, const char *utf8_name);

HPy_ID(154)
int HPy_HasAttr(HPyContext *ctx, HPy obj, HPy name);
HPy_ID(155)
int HPy_HasAttr_s(HPyContext *ctx, HPy obj, const char *utf8_name);

HPy_ID(156)
int HPy_SetAttr(HPyContext *ctx, HPy obj, HPy name, HPy value);
HPy_ID(157)
int HPy_SetAttr_s(HPyContext *ctx, HPy obj, const char *utf8_name, HPy value);

HPy_ID(158)
HPy HPy_GetItem(HPyContext *ctx, HPy obj, HPy key);
HPy_ID(159)
HPy HPy_GetItem_i(HPyContext *ctx, HPy obj, HPy_ssize_t idx);
HPy_ID(160)
HPy HPy_GetItem_s(HPyContext *ctx, HPy obj, const char *utf8_key);

HPy_ID(161)
int HPy_Contains(HPyContext *ctx, HPy container, HPy key);

HPy_ID(162)
int HPy_SetItem(HPyContext *ctx, HPy obj, HPy key, HPy value);
HPy_ID(163)
int HPy_SetItem_i(HPyContext *ctx, HPy obj, HPy_ssize_t idx, HPy value);
HPy_ID(164)
int HPy_SetItem_s(HPyContext *ctx, HPy obj, const char *utf8_key, HPy value);

HPy_ID(235)
int HPy_DelItem(HPyContext *ctx, HPy obj, HPy key);
HPy_ID(236)
int HPy_DelItem_i(HPyContext *ctx, HPy obj, HPy_ssize_t idx);
HPy_ID(237)
int HPy_DelItem_s(HPyContext *ctx, HPy obj, const char *utf8_key);

/**
 * Returns the type of the given object ``obj``.
 *
 * On failure, raises ``SystemError`` and returns ``HPy_NULL``. This is
 * equivalent to the Python expression``type(obj)``.
 *
 * :param ctx:
 *     The execution context.
 * :param obj:
```

(continues on next page)

(continued from previous page)

```
 *     a Python object (must not be ``HPy_NULL``)
 *
 * :returns:
 *     The type of ``obj`` or ``HPy_NULL`` in case of errors.
 */
HPy_ID(165)
HPy HPy_Type(HPyContext *ctx, HPy obj);

/**
 * Checks if ``ob`` is an instance of ``type`` or any subtype of ``type``.
 *
 * :param ctx:
 *     The execution context.
 * :param obj:
 *     a Python object (must not be ``HPy_NULL``)
 * :param type:
 *     A Python type object. This argument must not be ``HPy_NULL`` and must be
 *     a type (i.e. it must inherit from Python ``type``). If this is not the
 *     case, the behavior is undefined (verification of the argument is only
 *     done in debug mode).
 *
 * :returns:
 *     Non-zero if object ``obj`` is an instance of type ``type`` or an instance
 *     of a subtype of ``type``, and ``0`` otherwise.
 */
HPy_ID(166)
int HPy_TypeCheck(HPyContext *ctx, HPy obj, HPy type);

/**
 * Return the type's name.
 *
 * Equivalent to getting the type's ``__name__`` attribute. If you want to
 * retrieve the type's name as a handle that refers to a ``str``, then just use
 * ``HPy_GetAttr_s(ctx, type, "__name__")``.
 *
 * :param ctx:
 *     The execution context.
 * :param type:
 *     A Python type object. This argument must not be ``HPy_NULL`` and must be
 *     a type (i.e. it must inherit from Python ``type``). If this is not the
 *     case, the behavior is undefined (verification of the argument is only
 *     done in debug mode).
 *
 * :returns:
 *     The name of the type as C string (UTF-8 encoded) or ``NULL`` in case of
 *     an error. The returned pointer is read-only and guaranteed to be valid as
 *     long as the handle ``type`` is valid.
 */
HPy_ID(253)
const char *HPyType_GetName(HPyContext *ctx, HPy type);

/**
 * Checks if ``sub`` is a subtype of ``type``.
 *
 * This function only checks for actual subtypes, which means that
 * ``__subclasscheck__()`` is not called on ``type``.
 *
```

(continues on next page)

(continued from previous page)

```
 * :param ctx:
 *     The execution context.
 * :param sub:
 *     A Python type object. This argument must not be ``HPy_NULL`` and must be
 *     a type (i.e. it must inherit from Python ``type``). If this is not the
 *     case, the behavior is undefined (verification of the argument is only
 *     done in debug mode).
 * :param type:
 *     A Python type object. This argument must not be ``HPy_NULL`` and must be
 *     a type (i.e. it must inherit from Python ``type``). If this is not the
 *     case, the behavior is undefined (verification of the argument is only
 *     done in debug mode).
 *
 * :returns:
 *     Non-zero if ``sub`` is a subtype of ``type``.
 */
HPy_ID(254)
int HPyType_IsSubtype(HPyContext *ctx, HPy sub, HPy type);

HPy_ID(167)
int HPy_Is(HPyContext *ctx, HPy obj, HPy other);

HPy_ID(168)
void* _HPy_AsStruct_Object(HPyContext *ctx, HPy h);
HPy_ID(169)
void* _HPy_AsStruct_Legacy(HPyContext *ctx, HPy h);
HPy_ID(228)
void* _HPy_AsStruct_Type(HPyContext *ctx, HPy h);
HPy_ID(229)
void* _HPy_AsStruct_Long(HPyContext *ctx, HPy h);
HPy_ID(230)
void* _HPy_AsStruct_Float(HPyContext *ctx, HPy h);
HPy_ID(231)
void* _HPy_AsStruct_Unicode(HPyContext *ctx, HPy h);
HPy_ID(232)
void* _HPy_AsStruct_Tuple(HPyContext *ctx, HPy h);
HPy_ID(233)
void* _HPy_AsStruct_List(HPyContext *ctx, HPy h);
HPy_ID(234)
HPyType_BuiltinShape _HPyType_GetBuiltinShape(HPyContext *ctx, HPy h_type);

HPy_ID(170)
HPy _HPy_New(HPyContext *ctx, HPy h_type, void **data);

HPy_ID(171)
HPy HPy_Repr(HPyContext *ctx, HPy obj);
HPy_ID(172)
HPy HPy_Str(HPyContext *ctx, HPy obj);
HPy_ID(173)
HPy HPy_ASCII(HPyContext *ctx, HPy obj);
HPy_ID(174)
HPy HPy_Bytes(HPyContext *ctx, HPy obj);

HPy_ID(175)
HPy HPy_RichCompare(HPyContext *ctx, HPy v, HPy w, int op);
HPy_ID(176)
int HPy_RichCompareBool(HPyContext *ctx, HPy v, HPy w, int op);
```

(continues on next page)

(continued from previous page)

```

HPy_ID(177)
HPy_hash_t HPy_Hash(HPyContext *ctx, HPy obj);

/* bytesobject.h */
HPy_ID(178)
int HPyBytes_Check(HPyContext *ctx, HPy h);
HPy_ID(179)
HPy_ssize_t HPyBytes_Size(HPyContext *ctx, HPy h);
HPy_ID(180)
HPy_ssize_t HPyBytes_GET_SIZE(HPyContext *ctx, HPy h);
HPy_ID(181)
const char* HPyBytes_AsString(HPyContext *ctx, HPy h);
HPy_ID(182)
const char* HPyBytes_AS_STRING(HPyContext *ctx, HPy h);
HPy_ID(183)
HPy HPyBytes_FromString(HPyContext *ctx, const char *bytes);
HPy_ID(184)
HPy HPyBytes_FromStringAndSize(HPyContext *ctx, const char *bytes, HPy_ssize_t len);

/* unicodeobject.h */
HPy_ID(185)
HPy HPyUnicode_FromString(HPyContext *ctx, const char *utf8);
HPy_ID(186)
int HPyUnicode_Check(HPyContext *ctx, HPy h);
HPy_ID(187)
HPy HPyUnicode_AsASCIIString(HPyContext *ctx, HPy h);
HPy_ID(188)
HPy HPyUnicode_AsLatin1String(HPyContext *ctx, HPy h);
HPy_ID(189)
HPy HPyUnicode_AsUTF8String(HPyContext *ctx, HPy h);
HPy_ID(190)
const char* HPyUnicode_AsUTF8AndSize(HPyContext *ctx, HPy h, HPy_ssize_t *size);
HPy_ID(191)
HPy HPyUnicode_FromWideChar(HPyContext *ctx, const wchar_t *w, HPy_ssize_t size);
HPy_ID(192)
HPy HPyUnicode_DecodeFSDefault(HPyContext *ctx, const char *v);
HPy_ID(193)
HPy HPyUnicode_DecodeFSDefaultAndSize(HPyContext *ctx, const char *v, HPy_ssize_t␣
↪size);
HPy_ID(194)
HPy HPyUnicode_EncodeFSDefault(HPyContext *ctx, HPy h);
HPy_ID(195)
HPy_UCS4 HPyUnicode_ReadChar(HPyContext *ctx, HPy h, HPy_ssize_t index);
HPy_ID(196)
HPy HPyUnicode_DecodeASCII(HPyContext *ctx, const char *ascii, HPy_ssize_t size,␣
↪const char *errors);
HPy_ID(197)
HPy HPyUnicode_DecodeLatin1(HPyContext *ctx, const char *latin1, HPy_ssize_t size,␣
↪const char *errors);

/**
 * Decode a bytes-like object to a Unicode object.
 *
 * The bytes of the bytes-like object are decoded according to the given
 * encoding and using the error handling defined by ``errors``.
 *
```

(continues on next page)

(continued from previous page)

```
 * :param ctx:
 *     The execution context.
 * :param obj:
 *     A bytes-like object. This can be, for example, Python *bytes*,
 *     *bytearray*, *memoryview*, *array.array* and objects that support the
 *     Buffer protocol. If this argument is `HPy_NULL``, a ``SystemError`` will
 *     be raised. If the argument is not a bytes-like object, a ``TypeError``
 *     will be raised.
 * :param encoding:
 *     The name (UTF-8 encoded C string) of the encoding to use. If the encoding
 *     does not exist, a ``LookupError`` will be raised. If this argument is
 *     ``NULL``, the default encoding ``UTF-8`` will be used.
 * :param errors:
 *     The error handling (UTF-8 encoded C string) to use when decoding. The
 *     possible values depend on the used encoding. This argument may be
 *     ``NULL`` in which case it will default to ``"strict"``.
 *
 * :returns:
 *     A handle to a ``str`` object created from the decoded bytes or
 *     ``HPy_NULL`` in case of errors.
 */
HPy_ID(255)
HPy HPyUnicode_FromEncodedObject(HPyContext *ctx, HPy obj, const char *encoding,␣
↪const char *errors);

/**
 * Return a substring of ``str``, from character index ``start`` (included) to
 * character index ``end`` (excluded).
 *
 * Indices ``start`` and ``end`` must not be negative, otherwise an
 * ``IndexError`` will be raised. If ``start >= len(str)`` or if
 * ``end < start``, an empty string will be returned. If ``end > len(str)`` then
 * ``end == len(str)`` will be assumed.
 *
 * :param ctx:
 *     The execution context.
 * :param str:
 *     A Python Unicode object (must not be ``HPy_NULL``). Otherwise, the
 *     behavior is undefined (verification of the argument is only done in
 *     debug mode).
 * :param start:
 *     The non-negative start index (inclusive).
 * :param end:
 *    The non-negative end index (exclusive).
 *
 * :returns:
 *     The requested substring or ``HPy_NULL`` in case of an error.
 */
HPy_ID(256)
HPy HPyUnicode_Substring(HPyContext *ctx, HPy str, HPy_ssize_t start, HPy_ssize_t␣
↪end);

/* listobject.h */
HPy_ID(198)
int HPyList_Check(HPyContext *ctx, HPy h);
HPy_ID(199)
HPy HPyList_New(HPyContext *ctx, HPy_ssize_t len);
```

(continues on next page)

(continued from previous page)

```
HPy_ID(200)
int HPyList_Append(HPyContext *ctx, HPy h_list, HPy h_item);

/* dictobject.h */

/**
 * Tests if an object is an instance of a Python dict.
 *
 * :param ctx:
 *     The execution context.
 * :param h:
 *     A handle to an arbitrary object (must not be ``HPy_NULL``).
 *
 * :returns:
 *     Non-zero if object ``h`` is an instance of type ``dict`` or an instance
 *     of a subtype of ``dict``, and ``0`` otherwise.
 */
HPy_ID(201)
int HPyDict_Check(HPyContext *ctx, HPy h);

/**
 * Creates a new empty Python dictionary.
 *
 * :param ctx:
 *     The execution context.
 *
 * :returns:
 *     A handle to the new and empty Python dictionary or ``HPy_NULL`` in case
 *     of an error.
 */
HPy_ID(202)
HPy HPyDict_New(HPyContext *ctx);

/**
 * Returns a list of all keys from the dictionary.
 *
 * Note: This function will directly access the storage of the dict object and
 * therefore ignores if method ``keys`` was overwritten.
 *
 * :param ctx:
 *     The execution context.
 * :param h:
 *     A Python dict object. If this argument is ``HPy_NULL`` or not an
 *     instance of a Python dict, a ``SystemError`` will be raised.
 *
 * :returns:
 *     A Python list object containing all keys of the given dictionary or
 *     ``HPy_NULL`` in case of an error.
 */
HPy_ID(257)
HPy HPyDict_Keys(HPyContext *ctx, HPy h);

/**
 * Creates a copy of the provided Python dict object.
 *
 * :param ctx:
 *     The execution context.
```

(continues on next page)

(continued from previous page)

```
 * :param h:
 *     A Python dict object. If this argument is ``HPy_NULL`` or not an
 *     instance of a Python dict, a ``SystemError`` will be raised.
 *
 * :returns:
 *     Return a new dictionary that contains the same key-value pairs as ``h``
 *     or ``HPy_NULL`` in case of an error.
 */
HPy_ID(258)
HPy HPyDict_Copy(HPyContext *ctx, HPy h);

/* tupleobject.h */
HPy_ID(203)
int HPyTuple_Check(HPyContext *ctx, HPy h);
HPy_ID(204)
HPy HPyTuple_FromArray(HPyContext *ctx, HPy items[], HPy_ssize_t n);
// note: HPyTuple_Pack is implemented as a macro in common/macros.h

/* sliceobject.h */

/**
 * Extract the start, stop and step data members from a slice object as C
 * integers.
 *
 * The slice members may be arbitrary int-like objects. If they are not Python
 * int objects, they will be coerced to int objects by calling their
 * ``__index__`` method.
 *
 * If a slice member value is out of bounds, it will be set to the maximum value
 * of ``HPy_ssize_t`` if the member was a positive number, or to the minimum
 * value of ``HPy_ssize_t`` if it was a negative number.
 *
 * :param ctx:
 *     The execution context.
 * :param slice:
 *     A handle to a Python slice object. This argument must be a slice object
 *     and must not be ``HPy_NULL``. Otherwise, behavior is undefined.
 * :param start:
 *     A pointer to a variable where to write the unpacked slice start. Must not
 *     be ``NULL``.
 * :param end:
 *     A pointer to a variable where to write the unpacked slice end. Must not
 * :param step:
 *     A pointer to a variable where to write the unpacked slice step. Must not
 *     be ``NULL``.
 *
 * :returns:
 *     ``-1`` on error, ``0`` on success
 */
HPy_ID(259)
int HPySlice_Unpack(HPyContext *ctx, HPy slice, HPy_ssize_t *start, HPy_ssize_t *stop,
↪ HPy_ssize_t *step);

/* import.h */
HPy_ID(205)
HPy HPyImport_ImportModule(HPyContext *ctx, const char *utf8_name);

```

(continues on next page)

(continued from previous page)

```
/* pycapsule.h */
HPy_ID(244)
HPy HPyCapsule_New(HPyContext *ctx, void *pointer, const char *utf8_name, HPyCapsule_
↪Destructor *destructor);
HPy_ID(245)
void* HPyCapsule_Get(HPyContext *ctx, HPy capsule, _HPyCapsule_key key, const char␣
↪*utf8_name);
HPy_ID(246)
int HPyCapsule_IsValid(HPyContext *ctx, HPy capsule, const char *utf8_name);
HPy_ID(247)
int HPyCapsule_Set(HPyContext *ctx, HPy capsule, _HPyCapsule_key key, void *value);

/* integration with the old CPython API */
HPy_ID(206)
HPy HPy_FromPyObject(HPyContext *ctx, cpy_PyObject *obj);
HPy_ID(207)
cpy_PyObject *HPy_AsPyObject(HPyContext *ctx, HPy h);

/* internal helpers which need to be exposed to modules for practical reasons :( */
HPy_ID(208)
void _HPy_CallRealFunctionFromTrampoline(HPyContext *ctx,
                                         HPyFunc_Signature sig,
                                         HPyCFunction func,
                                         void *args);

/* Builders */

/**
 * Create a new list builder for ``size`` elements. The builder is then able to
 * take at most ``size`` elements. This function does not raise any
 * exception (even if running out of memory).
 *
 * :param ctx:
 *     The execution context.
 * :param size:
 *     The number of elements to hold.
 */
HPy_ID(209)
HPyListBuilder HPyListBuilder_New(HPyContext *ctx, HPy_ssize_t size);

/**
 * Assign an element to a certain index of the builder. Valid indices are in
 * range ``0 <= index < size`` where ``size`` is the value passed to
 * :c:func:`HPyListBuilder_New`. This function does not raise any exception.
 *
 * :param ctx:
 *     The execution context.
 * :param builder:
 *     A list builder handle.
 * :param index:
 *     The index to assign the object to.
 * :param h_item:
 *     An HPy handle of the object to store or ``HPy_NULL``. Please note that
 *     HPy **never** steals handles and so, ``h_item`` needs to be closed by
 *     the caller.
 */
HPy_ID(210)
```

(continues on next page)

(continued from previous page)

```
void HPyListBuilder_Set(HPyContext *ctx, HPyListBuilder builder,
                        HPy_ssize_t index, HPy h_item);

/**
 * Build a list from a list builder.
 *
 * :param ctx:
 *     The execution context.
 * :param builder:
 *     A list builder handle.
 *
 * :returns:
 *     An HPy handle to a list containing the values inserted with
 *     :c:func:`HPyListBuilder_Set` or ``HPy_NULL`` in case an error occurred
 *     during building or earlier when creating the builder or setting the
 *     items.
 */
HPy_ID(211)
HPy HPyListBuilder_Build(HPyContext *ctx, HPyListBuilder builder);

/**
 * Cancel building of a tuple and free any acquired resources.
 * This function ignores if any error occurred previously when using the tuple
 * builder.
 *
 * :param ctx:
 *     The execution context.
 * :param builder:
 *     A tuple builder handle.
 */
HPy_ID(212)
void HPyListBuilder_Cancel(HPyContext *ctx, HPyListBuilder builder);

/**
 * Create a new tuple builder for ``size`` elements. The builder is then able
 * to take at most ``size`` elements. This function does not raise any
 * exception (even if running out of memory).
 *
 * :param ctx:
 *     The execution context.
 * :param size:
 *     The number of elements to hold.
 */
HPy_ID(213)
HPyTupleBuilder HPyTupleBuilder_New(HPyContext *ctx, HPy_ssize_t size);

/**
 * Assign an element to a certain index of the builder. Valid indices are in
 * range ``0 <= index < size`` where ``size`` is the value passed to
 * :c:func:`HPyTupleBuilder_New`. This function does not raise * any exception.
 *
 * :param ctx:
 *     The execution context.
 * :param builder:
 *     A tuple builder handle.
 * :param index:
 *     The index to assign the object to.
```

(continues on next page)

(continued from previous page)

```c
 * :param h_item:
 *     An HPy handle of the object to store or ``HPy_NULL``. Please note that
 *     HPy **never** steals handles and so, ``h_item`` needs to be closed by
 *     the caller.
 */
HPy_ID(214)
void HPyTupleBuilder_Set(HPyContext *ctx, HPyTupleBuilder builder,
                         HPy_ssize_t index, HPy h_item);

/**
 * Build a tuple from a tuple builder.
 *
 * :param ctx:
 *     The execution context.
 * :param builder:
 *     A tuple builder handle.
 *
 * :returns:
 *     An HPy handle to a tuple containing the values inserted with
 *     :c:func:`HPyTupleBuilder_Set` or ``HPy_NULL`` in case an error occurred
 *     during building or earlier when creating the builder or setting the
 *     items.
 */
HPy_ID(215)
HPy HPyTupleBuilder_Build(HPyContext *ctx, HPyTupleBuilder builder);

/**
 * Cancel building of a tuple and free any acquired resources.
 * This function ignores if any error occurred previously when using the tuple
 * builder.
 *
 * :param ctx:
 *     The execution context.
 * :param builder:
 *     A tuple builder handle.
 */
HPy_ID(216)
void HPyTupleBuilder_Cancel(HPyContext *ctx, HPyTupleBuilder builder);

/* Helper for correctly closing handles */

HPy_ID(217)
HPyTracker HPyTracker_New(HPyContext *ctx, HPy_ssize_t size);
HPy_ID(218)
int HPyTracker_Add(HPyContext *ctx, HPyTracker ht, HPy h);
HPy_ID(219)
void HPyTracker_ForgetAll(HPyContext *ctx, HPyTracker ht);
HPy_ID(220)
void HPyTracker_Close(HPyContext *ctx, HPyTracker ht);

/**
 * HPyFields should be used ONLY in parts of memory which is known to the GC,
 * e.g. memory allocated by HPy_New:
 *
 *   - NEVER declare a local variable of type HPyField
 *   - NEVER use HPyField on a struct allocated by e.g. malloc()
 *
```

(continues on next page)

(continued from previous page)

```
 * **CPython's note**: contrary to PyObject*, you don't need to manually
 * manage refcounting when using HPyField: if you use HPyField_Store to
 * overwrite an existing value, the old object will be automatically decrefed.
 * This means that you CANNOT use HPyField_Store to write memory which
 * contains uninitialized values, because it would try to decref a dangling
 * pointer.
 *
 * Note that HPy_New automatically zeroes the memory it allocates, so
 * everything works well out of the box. In case you are using manually
 * allocated memory, you should initialize the HPyField to HPyField_NULL.
 *
 * Note the difference:
 *
 *   - ``obj->f = HPyField_NULL``: this should be used only to initialize
 *     uninitialized memory. If you use it to overwrite a valid HPyField, you
 *     will cause a memory leak (at least on CPython)
 *
 *   - HPyField_Store(ctx, &obj->f, HPy_NULL): this does the right thing and
 *     decref the old value. However, you CANNOT use it if the memory is not
 *     initialized.
 *
 * Note: target_object and source_object are there in case an implementation
 * needs to add write and/or read barriers on the objects. They are ignored by
 * CPython but e.g. PyPy needs a write barrier.
*/
HPy_ID(221)
void HPyField_Store(HPyContext *ctx, HPy target_object, HPyField *target_field, HPy␣
↪h);
HPy_ID(222)
HPy HPyField_Load(HPyContext *ctx, HPy source_object, HPyField source_field);

/**
 * Leaving Python execution: for releasing GIL and other use-cases.
 *
 * In most situations, users should prefer using convenience macros:
 * HPy_BEGIN_LEAVE_PYTHON(context)/HPy_END_LEAVE_PYTHON(context)
 *
 * HPy extensions may leave Python execution when running Python independent
 * code: long-running computations or blocking operations. When an extension
 * has left the Python execution it must not call any HPy API other than
 * HPy_ReenterPythonExecution. It can access pointers returned by HPy API,
 * e.g., HPyUnicode_AsUTF8String, provided that they are valid at the point
 * of calling HPy_LeavePythonExecution.
 *
 * Python execution must be reentered on the same thread as where it was left.
 * The leave/enter calls must not be nested. Debug mode will, in the future,
 * enforce these constraints.
 *
 * Python implementations may use this knowledge however they wish. The most
 * obvious use case is to release the GIL, in which case the
 * HPy_BEGIN_LEAVE_PYTHON/HPy_END_LEAVE_PYTHON becomes equivalent to
 * Py_BEGIN_ALLOW_THREADS/Py_END_ALLOW_THREADS.
*/
HPy_ID(223)
void HPy_ReenterPythonExecution(HPyContext *ctx, HPyThreadState state);
HPy_ID(224)
HPyThreadState HPy_LeavePythonExecution(HPyContext *ctx);
```

(continues on next page)

(continued from previous page)

```

/**
 * HPyGlobal is an alternative to module state. HPyGlobal must be a statically
 * allocated C global variable registered in HPyModuleDef.globals array.
 * A HPyGlobal can be used only after the HPy module where it is registered was
 * created using HPyModule_Create.
 *
 * HPyGlobal serves as an identifier of a Python object that should be globally
 * available per one Python interpreter. Python objects referenced by HPyGlobals
 * are destroyed automatically on the interpreter exit (not necessarily the
 * process exit).
 *
 * HPyGlobal instance does not allow anything else but loading and storing
 * a HPy handle using a HPyContext. Even if the HPyGlobal C variable may
 * be shared between threads or different interpreter instances within one
 * process, the API to load and store a handle from HPyGlobal is thread-safe (but
 * like any other HPy API must not be called in HPy_LeavePythonExecution blocks).
 *
 * Given that a handle to object X1 is stored to HPyGlobal using HPyContext of
 * Python interpreter I1, then loading a handle from the same HPyGlobal using
 * HPyContext of Python interpreter I1 should give a handle to the same object
 * X1. Another Python interpreter I2 running within the same process and using
 * the same HPyGlobal variable will not be able to load X1 from it, it will have
 * its own view on what is stored in the given HPyGlobal.
 *
 * Python interpreters may use indirection to isolate different interpreter
 * instances, but alternative techniques such as copy-on-write or immortal
 * objects can be used to avoid that indirection (even selectively on per
 * object basis using tagged pointers).
 *
 * CPython HPy implementation may even provide configuration option that
 * switches between a faster version that directly stores PyObject* to
 * HPyGlobal but does not support subinterpreters, or a version that supports
 * subinterpreters. For now, CPython HPy always stores PyObject* directly
 * to HPyGlobal.
 *
 * While the standard implementation does not fully enforce the documented
 * contract, the HPy debug mode will enforce it (not implemented yet).
 *
 * **Implementation notes:**
 * All Python interpreters running in one process must be compatible, because
 * they will share all HPyGlobal C level variables. The internal data stored
 * in HPyGlobal are specific for each HPy implementation, each implementation
 * is also responsible for handling thread-safety when initializing the
 * internal data in HPyModule_Create. Note that HPyModule_Create may be called
 * concurrently depending on the semantics of the Python implementation (GIL vs
 * no GIL) and also depending on the whether there may be multiple instances of
 * given Python interpreter running within the same process. In the future, HPy
 * ABI may include a contract that internal data of each HPyGlobal must be
 * initialized to its address using atomic write and HPy implementations will
 * not be free to choose what to store in HPyGlobal, however, this will allow
 * multiple different HPy implementations within one process. This contract may
 * also be activated only by some runtime option, letting the HPy implementation
 * use more optimized HPyGlobal implementation otherwise.
*/
HPy_ID(225)
void HPyGlobal_Store(HPyContext *ctx, HPyGlobal *global, HPy h);
```

(continues on next page)

(continued from previous page)

```
HPy_ID(226)
HPy HPyGlobal_Load(HPyContext *ctx, HPyGlobal global);

/* Debugging helpers */
HPy_ID(227)
void _HPy_Dump(HPyContext *ctx, HPy h);

/* Evaluating Python statements/expressions */

/**
 * Parse and compile the Python source code.
 *
 * :param ctx:
 *     The execution context.
 * :param utf8_source:
 *     Python source code given as UTF-8 encoded C string (must not be ``NULL``).
 * :param utf8_filename:
 *     The filename (UTF-8 encoded C string) to use for construction of the code
 *     object. It may appear in tracebacks or in ``SyntaxError`` exception
 *     messages.
 * :param kind:
 *     The source kind which tells the parser if a single expression, statement,
 *     or a whole file should be parsed (see enum :c:enum:`HPy_SourceKind`).
 *
 * :returns:
 *     A Python code object resulting from the parsed and compiled Python source
 *     code or ``HPy_NULL`` in case of errors.
 */
HPy_ID(248)
HPy HPy_Compile_s(HPyContext *ctx, const char *utf8_source, const char *utf8_filename,
↪ HPy_SourceKind kind);

/**
 * Evaluate a precompiled code object.
 *
 * Code objects can be compiled from a string using :c:func:`HPy_Compile_s`.
 *
 * :param ctx:
 *     The execution context.
 * :param code:
 *     The code object to evaluate.
 * :param globals:
 *     A Python dictionary defining the global variables for the evaluation.
 * :param locals:
 *     A mapping object defining the local variables for the evaluation.
 *
 * :returns:
 *     The result produced by the executed code. May be ``HPy_NULL`` in case of
 *     errors.
 */
HPy_ID(249)
HPy HPy_EvalCode(HPyContext *ctx, HPy code, HPy globals, HPy locals);
HPy_ID(250)
HPy HPyContextVar_New(HPyContext *ctx, const char *name, HPy default_value);
HPy_ID(251)
int32_t HPyContextVar_Get(HPyContext *ctx, HPy context_var, HPy default_value, HPy␣
↪*result);
```

(continues on next page)

(continued from previous page)

```
HPy_ID(252)
HPy HPyContextVar_Set(HPyContext *ctx, HPy context_var, HPy value);

/**
 * Set the call function for the given object.
 *
 * By defining slot ``HPy_tp_call`` for some type, instances of this type will
 * be callable objects. The specified call function will be used by default for
 * every instance. This should account for the most common case (every instance
 * of an object uses the same call function) but to still provide the necessary
 * flexibility, function ``HPy_SetCallFunction`` allows to set different (maybe
 * specialized) call functions for each instance. This must be done in the
 * constructor of an object.
 *
 * A more detailed description on how to use that function can be found in
 * section :ref:`porting-guide:calling protocol`.
 *
 * :param ctx:
 *     The execution context.
 * :param h:
 *     A handle to an object implementing the call protocol, i.e., the object's
 *     type must have slot ``HPy_tp_call``. Otherwise, a ``TypeError`` will be
 *     raised. This argument must not be ``HPy_NULL``.
 * :param def:
 *     A pointer to the call function definition to set (must not be
 *     ``NULL``). The definition is usually created using
 *     :c:macro:`HPyDef_CALL_FUNCTION`
 *
 * :returns:
 *     ``0`` in case of success and ``-1`` in case of an error.
 */
HPy_ID(260)
int HPy_SetCallFunction(HPyContext *ctx, HPy h, HPyCallFunction *func);

/* *******
   hpyfunc
   *******

   These typedefs are used to generate the various macros used by
   include/common/hpyfunc.h
*/
typedef HPy (*HPyFunc_noargs)(HPyContext *ctx, HPy self);
typedef HPy (*HPyFunc_o)(HPyContext *ctx, HPy self, HPy arg);
typedef HPy (*HPyFunc_varargs)(HPyContext *ctx, HPy self, const HPy *args, size_t␣
↪nargs);
typedef HPy (*HPyFunc_keywords)(HPyContext *ctx, HPy self, const HPy *args,
                                size_t nargs, HPy kwnames);

typedef HPy (*HPyFunc_unaryfunc)(HPyContext *ctx, HPy);
typedef HPy (*HPyFunc_binaryfunc)(HPyContext *ctx, HPy, HPy);
typedef HPy (*HPyFunc_ternaryfunc)(HPyContext *ctx, HPy, HPy, HPy);
typedef int (*HPyFunc_inquiry)(HPyContext *ctx, HPy);
typedef HPy_ssize_t (*HPyFunc_lenfunc)(HPyContext *ctx, HPy);
typedef HPy (*HPyFunc_ssizeargfunc)(HPyContext *ctx, HPy, HPy_ssize_t);
typedef HPy (*HPyFunc_ssizessizeargfunc)(HPyContext *ctx, HPy, HPy_ssize_t, HPy_ssize_
↪t);
typedef int (*HPyFunc_ssizeobjargproc)(HPyContext *ctx, HPy, HPy_ssize_t, HPy);
```

(continues on next page)

(continued from previous page)

```
typedef int (*HPyFunc_ssizessizeobjargproc)(HPyContext *ctx, HPy, HPy_ssize_t, HPy_
↪ssize_t, HPy);
typedef int (*HPyFunc_objobjargproc)(HPyContext *ctx, HPy, HPy, HPy);
typedef void (*HPyFunc_freefunc)(HPyContext *ctx, void *);
typedef HPy (*HPyFunc_getattrfunc)(HPyContext *ctx, HPy, char *);
typedef HPy (*HPyFunc_getattrofunc)(HPyContext *ctx, HPy, HPy);
typedef int (*HPyFunc_setattrfunc)(HPyContext *ctx, HPy, char *, HPy);
typedef int (*HPyFunc_setattrofunc)(HPyContext *ctx, HPy, HPy, HPy);
typedef HPy (*HPyFunc_reprfunc)(HPyContext *ctx, HPy);
typedef HPy_hash_t (*HPyFunc_hashfunc)(HPyContext *ctx, HPy);
typedef HPy (*HPyFunc_richcmpfunc)(HPyContext *ctx, HPy, HPy, HPy_RichCmpOp);
typedef HPy (*HPyFunc_getiterfunc)(HPyContext *ctx, HPy);
typedef HPy (*HPyFunc_iternextfunc)(HPyContext *ctx, HPy);
typedef HPy (*HPyFunc_descrgetfunc)(HPyContext *ctx, HPy, HPy, HPy);
typedef int (*HPyFunc_descrsetfunc)(HPyContext *ctx, HPy, HPy, HPy);
typedef int (*HPyFunc_initproc)(HPyContext *ctx, HPy self,
                                const HPy *args, HPy_ssize_t nargs, HPy kw);
typedef HPy (*HPyFunc_newfunc)(HPyContext *ctx, HPy type, const HPy *args,
                               HPy_ssize_t nargs, HPy kw);
typedef HPy (*HPyFunc_getter)(HPyContext *ctx, HPy, void *);
typedef int (*HPyFunc_setter)(HPyContext *ctx, HPy, HPy, void *);
typedef int (*HPyFunc_objobjproc)(HPyContext *ctx, HPy, HPy);
typedef int (*HPyFunc_getbufferproc)(HPyContext *ctx, HPy, HPy_buffer *, int);
typedef void (*HPyFunc_releasebufferproc)(HPyContext *ctx, HPy, HPy_buffer *);
typedef int (*HPyFunc_traverseproc)(void *object, HPyFunc_visitproc visit, void *arg);
typedef void (*HPyFunc_destructor)(HPyContext *ctx, HPy);

typedef void (*HPyFunc_destroyfunc)(void *);

// Note: separate type, because we need a different trampoline
typedef HPy (*HPyFunc_mod_create)(HPyContext *ctx, HPy);


/* ~~~ HPySlot_Slot ~~~

   The following enum is used to generate autogen_hpyslot.h, which contains:

     - The real definition of the enum HPySlot_Slot

     - the macros #define _HPySlot_SIGNATURE_*

*/

// NOTE: if you uncomment/enable a slot below, make sure to write a corresponding
// test in test_slots.py

/* Note that the magic numbers are the same as CPython */
typedef enum {
    HPy_bf_getbuffer = SLOT(1, HPyFunc_GETBUFFERPROC),
    HPy_bf_releasebuffer = SLOT(2, HPyFunc_RELEASEBUFFERPROC),
    HPy_mp_ass_subscript = SLOT(3, HPyFunc_OBJOBJARGPROC),
    HPy_mp_length = SLOT(4, HPyFunc_LENFUNC),
    HPy_mp_subscript = SLOT(5, HPyFunc_BINARYFUNC),
    HPy_nb_absolute = SLOT(6, HPyFunc_UNARYFUNC),
    HPy_nb_add = SLOT(7, HPyFunc_BINARYFUNC),
    HPy_nb_and = SLOT(8, HPyFunc_BINARYFUNC),
    HPy_nb_bool = SLOT(9, HPyFunc_INQUIRY),
```

(continues on next page)

(continued from previous page)

```
    HPy_nb_divmod = SLOT(10, HPyFunc_BINARYFUNC),
    HPy_nb_float = SLOT(11, HPyFunc_UNARYFUNC),
    HPy_nb_floor_divide = SLOT(12, HPyFunc_BINARYFUNC),
    HPy_nb_index = SLOT(13, HPyFunc_UNARYFUNC),
    HPy_nb_inplace_add = SLOT(14, HPyFunc_BINARYFUNC),
    HPy_nb_inplace_and = SLOT(15, HPyFunc_BINARYFUNC),
    HPy_nb_inplace_floor_divide = SLOT(16, HPyFunc_BINARYFUNC),
    HPy_nb_inplace_lshift = SLOT(17, HPyFunc_BINARYFUNC),
    HPy_nb_inplace_multiply = SLOT(18, HPyFunc_BINARYFUNC),
    HPy_nb_inplace_or = SLOT(19, HPyFunc_BINARYFUNC),
    HPy_nb_inplace_power = SLOT(20, HPyFunc_TERNARYFUNC),
    HPy_nb_inplace_remainder = SLOT(21, HPyFunc_BINARYFUNC),
    HPy_nb_inplace_rshift = SLOT(22, HPyFunc_BINARYFUNC),
    HPy_nb_inplace_subtract = SLOT(23, HPyFunc_BINARYFUNC),
    HPy_nb_inplace_true_divide = SLOT(24, HPyFunc_BINARYFUNC),
    HPy_nb_inplace_xor = SLOT(25, HPyFunc_BINARYFUNC),
    HPy_nb_int = SLOT(26, HPyFunc_UNARYFUNC),
    HPy_nb_invert = SLOT(27, HPyFunc_UNARYFUNC),
    HPy_nb_lshift = SLOT(28, HPyFunc_BINARYFUNC),
    HPy_nb_multiply = SLOT(29, HPyFunc_BINARYFUNC),
    HPy_nb_negative = SLOT(30, HPyFunc_UNARYFUNC),
    HPy_nb_or = SLOT(31, HPyFunc_BINARYFUNC),
    HPy_nb_positive = SLOT(32, HPyFunc_UNARYFUNC),
    HPy_nb_power = SLOT(33, HPyFunc_TERNARYFUNC),
    HPy_nb_remainder = SLOT(34, HPyFunc_BINARYFUNC),
    HPy_nb_rshift = SLOT(35, HPyFunc_BINARYFUNC),
    HPy_nb_subtract = SLOT(36, HPyFunc_BINARYFUNC),
    HPy_nb_true_divide = SLOT(37, HPyFunc_BINARYFUNC),
    HPy_nb_xor = SLOT(38, HPyFunc_BINARYFUNC),
    HPy_sq_ass_item = SLOT(39, HPyFunc_SSIZEOBJARGPROC),
    HPy_sq_concat = SLOT(40, HPyFunc_BINARYFUNC),
    HPy_sq_contains = SLOT(41, HPyFunc_OBJOBJPROC),
    HPy_sq_inplace_concat = SLOT(42, HPyFunc_BINARYFUNC),
    HPy_sq_inplace_repeat = SLOT(43, HPyFunc_SSIZEARGFUNC),
    HPy_sq_item = SLOT(44, HPyFunc_SSIZEARGFUNC),
    HPy_sq_length = SLOT(45, HPyFunc_LENFUNC),
    HPy_sq_repeat = SLOT(46, HPyFunc_SSIZEARGFUNC),
    //HPy_tp_alloc = SLOT(47, HPyFunc_X),      NOT SUPPORTED
    //HPy_tp_base = SLOT(48, HPyFunc_X),
    //HPy_tp_bases = SLOT(49, HPyFunc_X),
    HPy_tp_call = SLOT(50, HPyFunc_KEYWORDS),
    //HPy_tp_clear = SLOT(51, HPyFunc_X),      NOT SUPPORTED, use tp_traverse
    //HPy_tp_dealloc = SLOT(52, HPyFunc_X),    NOT SUPPORTED
    //HPy_tp_del = SLOT(53, HPyFunc_X),
    //HPy_tp_descr_get = SLOT(54, HPyFunc_X),
    //HPy_tp_descr_set = SLOT(55, HPyFunc_X),
    //HPy_tp_doc = SLOT(56, HPyFunc_X),
    //HPy_tp_getattr = SLOT(57, HPyFunc_X),
    //HPy_tp_getattro = SLOT(58, HPyFunc_X),
    HPy_tp_hash = SLOT(59, HPyFunc_HASHFUNC),
    HPy_tp_init = SLOT(60, HPyFunc_INITPROC),
    //HPy_tp_is_gc = SLOT(61, HPyFunc_X),
    //HPy_tp_iter = SLOT(62, HPyFunc_X),
    //HPy_tp_iternext = SLOT(63, HPyFunc_X),
    //HPy_tp_methods = SLOT(64, HPyFunc_X),    NOT SUPPORTED
    HPy_tp_new = SLOT(65, HPyFunc_NEWFUNC),
    HPy_tp_repr = SLOT(66, HPyFunc_REPRFUNC),
```

(continues on next page)

(continued from previous page)

```
    HPy_tp_richcompare = SLOT(67, HPyFunc_RICHCMPFUNC),
    //HPy_tp_setattr = SLOT(68, HPyFunc_X),
    //HPy_tp_setattro = SLOT(69, HPyFunc_X),
    HPy_tp_str = SLOT(70, HPyFunc_REPRFUNC),
    HPy_tp_traverse = SLOT(71, HPyFunc_TRAVERSEPROC),
    //HPy_tp_members = SLOT(72, HPyFunc_X),    NOT SUPPORTED
    //HPy_tp_getset = SLOT(73, HPyFunc_X),     NOT SUPPORTED
    //HPy_tp_free = SLOT(74, HPyFunc_X),       NOT SUPPORTED
    HPy_nb_matrix_multiply = SLOT(75, HPyFunc_BINARYFUNC),
    HPy_nb_inplace_matrix_multiply = SLOT(76, HPyFunc_BINARYFUNC),
    //HPy_am_await = SLOT(77, HPyFunc_X),
    //HPy_am_aiter = SLOT(78, HPyFunc_X),
    //HPy_am_anext = SLOT(79, HPyFunc_X),
    HPy_tp_finalize = SLOT(80, HPyFunc_DESTRUCTOR),

    /* extra HPy slots */
    HPy_tp_destroy = SLOT(1000, HPyFunc_DESTROYFUNC),

    /**
     * Module create slot: the function receives loader spec and should
     * return an HPy handle representing the module. Currently, creating
     * real module objects cannot be done by user code, so the only other
     * useful thing that this slot can do is to create another object that
     * can work as a module, such as SimpleNamespace.
     */
    HPy_mod_create = SLOT(2000, HPyFunc_MOD_CREATE),
    /**
     * Module exec slot: the function receives module object that was created
     * by the runtime from HPyModuleDef. This slot can do any initialization
     * of the module, such as adding types. There can be multiple exec slots
     * and they will be executed in the declaration order.
     */
    HPy_mod_exec = SLOT(2001, HPyFunc_INQUIRY),

} HPySlot_Slot;
```

2.8.2 HPy Helper Functions

HPy Helper functions are functions (written in C) that will be compiled together with the HPy extension's sources. The appropriate source files are automatically added to the extension sources. The helper functions will, of course, use the core API to interact with the interpreter. The main reason for having the helper functions in the HPy extension is to avoid compatibility problems due to different compilers.

Argument Parsing

Implementation of HPyArg_Parse and HPyArg_ParseKeywords.

Note: those functions are runtime helper functions, i.e., they are not part of the HPy context, but are available to HPy extensions to incorporate at compile time.

HPyArg_Parse parses positional arguments and replaces PyArg_ParseTuple. HPyArg_ParseKeywords parses positional and keyword arguments and replaces PyArg_ParseTupleAndKeywords.

HPy intends to only support the simpler format string types (numbers, bools) and handles. More complex types (e.g. buffers) should be retrieved as handles and then processed further as needed.

Supported Formatting Strings

Numbers

`b (int) [unsigned char]` Convert a nonnegative Python integer to an unsigned tiny int, stored in a C unsigned char.

`B (int) [unsigned char]` Convert a Python integer to a tiny int without overflow checking, stored in a C unsigned char.

`h (int) [short int]` Convert a Python integer to a C short int.

`H (int) [unsigned short int]` Convert a Python integer to a C unsigned short int, without overflow checking.

`i (int) [int]` Convert a Python integer to a plain C int.

`I (int) [unsigned int]` Convert a Python integer to a C unsigned int, without overflow checking.

`l (int) [long int]` Convert a Python integer to a C long int.

`k (int) [unsigned long]` Convert a Python integer to a C unsigned long without overflow checking.

`L (int) [long long]` Convert a Python integer to a C long long.

`K (int) [unsigned long long]` Convert a Python integer to a C unsigned long long without overflow checking.

`n (int) [HPy_ssize_t]` Convert a Python integer to a C HPy_ssize_t.

`f (float) [float]` Convert a Python floating point number to a C float.

`d (float) [double]` Convert a Python floating point number to a C double.

Strings and buffers

These formats allow accessing an object as a contiguous chunk of memory. You don't have to provide raw storage for the returned unicode or bytes area.

In general, when a format sets a pointer to a buffer, the pointer is valid only until the corresponding HPy handle is closed.

`s (unicode) [const char*]`

Convert a Unicode object to a C pointer to a character string. A pointer to an existing string is stored in the character pointer variable whose address you pass. The C string is NUL-terminated. The Python string must not contain embedded null code points; if it does, a *ValueError* exception is raised. Unicode objects are converted to C strings using 'utf-8' encoding. If this conversion fails, a *UnicodeError* is raised.

Note: This format does not accept bytes-like objects and is therefore not suitable for filesystem paths.

Handles (Python Objects)

`O (object) [HPy]` Store a handle pointing to a generic Python object.

When using O with HPyArg_ParseKeywords, an HPyTracker is created and returned via the parameter *ht*. If HPyArg_ParseKeywords returns successfully, you must call HPyTracker_Close on *ht* once the returned handles are no longer needed. This will close all the handles created during argument parsing. There is no need to call *HPyTracker_Close* on failure – the argument parser does this for you.

Miscellaneous

`p (bool) [int]` Tests the value passed in for truth (a boolean predicate) and converts the result to its equivalent C true/false integer value. Sets the int to 1 if the expression was true and 0 if it was false. This accepts any valid Python value. See Truth Value Testing for more information about how Python tests values for truth.

Options

`|` Indicates that the remaining arguments in the argument list are optional. The C variables corresponding to optional arguments should be initialized to their default value — when an optional argument is not specified, the contents of the corresponding C variable is not modified.

`$` HPyArg_ParseKeywords() only: Indicates that the remaining arguments in the argument list are keyword-only. Currently, all keyword-only arguments must also be optional arguments, so | must always be specified before $ in the format string.

`:` The list of format units ends here; the string after the colon is used as the function name in error messages. : and ; are mutually exclusive and whichever occurs first takes precedence.

`;` The list of format units ends here; the string after the semicolon is used as the error message instead of the default error message. : and ; are mutually exclusive and whichever occurs first takes precedence.

Argument Parsing API

int **`HPyArg_Parse`** (HPyContext **ctx*, HPyTracker **ht*, **`const`** HPy **args*, size_t *nargs*, **`const`** char **fmt*, ...)

Parse positional arguments.

Parameters

- **`ctx`** – The execution context.
- **`ht`** – An optional pointer to an HPyTracker. If the format string never results in new handles being created, `ht` may be `NULL`. Currently only the `O` formatting option to this function requires an HPyTracker.
- **`args`** – The array of positional arguments to parse.
- **`nargs`** – The number of elements in `args`.
- **`fmt`** – The format string to use to parse the arguments.
- **`...`** – A va_list of references to variables in which to store the parsed arguments. The number and types of the arguments should match the the format string, `fmt`.

Returns `0` on failure, `1` on success.

If a `NULL` pointer is passed to `ht` and an HPyTracker is required by the format string, a `SystemError` will be raised.

If a pointer is provided to `ht`, the HPyTracker will always be created and must be closed with `HPyTracker_Close` if parsing succeeds (after all handles returned are no longer needed). If parsing fails, this function will close the HPyTracker automatically.

Examples:

Using `HPyArg_Parse` without an HPyTracker:

```
long a, b;
if (!HPyArg_Parse(ctx, NULL, args, nargs, "ll", &a, &b))
    return HPy_NULL;
...
```

Using `HPyArg_Parse` with an HPyTracker:

```
long a, b;
HPyTracker ht;
if (!HPyArg_Parse(ctx, &ht, args, nargs, "ll", &a, &b))
    return HPy_NULL;
...
HPyTracker_Close(ctx, ht);
...
```

Note: Currently `HPyArg_Parse` never requires the use of an HPyTracker. The option exists only to support releasing temporary storage used by future format string codes (e.g. for character strings).

int **`HPyArg_ParseKeywords`**(HPyContext **ctx*, HPyTracker **ht*, **`const`** HPy **args*, size_t *nargs*, HPy *kwnames*, **`const`** char **fmt*, **`const`** char **keywords*[], ...)

Parse positional and keyword arguments.

Parameters

- **`ctx`** – The execution context.
- **`ht`** – An optional pointer to an HPyTracker. If the format string never results in new handles being created, `ht` may be `NULL`. Currently only the `O` formatting option to this function requires an HPyTracker.
- **`args`** – The array of positional arguments to parse.
- **`nargs`** – The number of elements in `args`.
- **`kwnames`** – A handle to the tuple of keyword argument names (may be `HPy_NULL`). The values of the keyword arguments are appended to `args`. Argument `nargs` does not include the keyword argument count.
- **`fmt`** – The format string to use to parse the arguments.
- **`keywords`** – A `NULL`-terminated array of argument names. The number of names should match the format string provided. Positional only arguments should have the name `""` (i.e. the null-terminated empty string). Positional only arguments must preceded all other arguments.
- **`...`** – A va_list of references to variables in which to store the parsed arguments. The number and types of the arguments should match the the format string, `fmt`.

Returns `0` on failure, `1` on success.

If a NULL pointer is passed to ht and an HPyTracker is required by the format string, a SystemError will be raised.

If a pointer is provided to ht, the HPyTracker will always be created and must be closed with HPyTracker_Close if parsing succeeds (after all handles returned are no longer needed). If parsing fails, this function will close the HPyTracker automatically.

Examples:

Using *HPyArg_ParseKeywords* without an *HPyTracker*:

```
long a, b;
if (!HPyArg_ParseKeywords(ctx, NULL, args, nargs, kwnames, "ll", &a, &b))
    return HPy_NULL;
...
```

Using *HPyArg_ParseKeywords* with an *HPyTracker*:

```
HPy a, b;
HPyTracker ht;
if (!HPyArg_ParseKeywords(ctx, &ht, args, nargs, kwnames, "OO", &a, &b))
    return HPy_NULL;
...
HPyTracker_Close(ctx, ht);
...
```

Note: Currently HPyArg_ParseKeywords only requires the use of an HPyTracker when the O format is used. In future other new format string codes (e.g. for character strings) may also require it.

int **HPyArg_ParseKeywordsDict** (HPyContext **ctx*, HPyTracker **ht*, **const** HPy **args*, HPy_ssize_t *nargs*, HPy *kw*, **const** char **fmt*, **const** char **keywords*[], ...)

Parse positional arguments and keyword arguments in a dict.

Parameters

- **ctx** – The execution context.
- **ht** – An optional pointer to an HPyTracker. If the format string never results in new handles being created, ht may be NULL. Currently only the O formatting option to this function requires an HPyTracker.
- **args** – The array of positional arguments to parse.
- **nargs** – The number of elements in args.
- **kw** – A handle to the dictionary of keyword arguments (may be HPy_NULL).
- **fmt** – The format string to use to parse the arguments.
- **keywords** – A NULL-terminated array of argument names. The number of names should match the format string provided. Positional only arguments should have the name "" (i.e. the null-terminated empty string). Positional only arguments must preceded all other arguments.
- **...** – A va_list of references to variables in which to store the parsed arguments. The number and types of the arguments should match the the format string, fmt.

Returns 0 on failure, 1 on success.

If a NULL pointer is passed to ht and an HPyTracker is required by the format string, a SystemError will be raised.

If a pointer is provided to `ht`, the HPyTracker will always be created and must be closed with `HPyTracker_Close` if parsing succeeds (after all handles returned are no longer needed). If parsing fails, this function will close the HPyTracker automatically.

For examples, see *`HPyArg_ParseKeywords()`*.

Building Complex Python Objects

Implementation of HPy_BuildValue.

Note: *`HPy_BuildValue()`* is a runtime helper functions, i.e., it is not a part of the HPy context, but is available to HPy extensions to incorporate at compile time.

`HPy_BuildValue` creates a new value based on a format string from the values passed in variadic arguments. Returns `HPy_NULL` in case of an error and raises an exception.

`HPy_BuildValue` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size `0` or one, parenthesize the format string.

Building complex values with `HPy_BuildValue` is more convenient than the equivalent code that uses more granular APIs with proper error handling and cleanup. Moreover, `HPy_BuildValue` provides straightforward way to port existing code that uses `Py_BuildValue`.

`HPy_BuildValue` always returns a new handle that will be owned by the caller. Even an artificial example `HPy_BuildValue(ctx, "O", h)` does not simply forward the value stored in `h` but duplicates the handle.

Supported Formatting Strings

Numbers

`i (int) [int]` Convert a plain C int to a Python integer object.

`l (int) [long int]` Convert a C long int to a Python integer object.

`I (int) [unsigned int]` Convert a C unsigned int to a Python integer object.

`k (int) [unsigned long]` Convert a C unsigned long to a Python integer object.

`L (int) [long long]` Convert a C long long to a Python integer object.

`K (int) [unsigned long long]` Convert a C unsigned long long to a Python integer object.

`n (int) [HPy_ssize_t]` Convert a C HPy_ssize_t to a Python integer object.

`f (float) [float]` Convert a C float to a Python floating point number.

`d (float) [double]` Convert a C double to a Python floating point number.

Collections

`(items) (tuple) [matching-items]` Convert a sequence of C values to a Python tuple with the same number of items.

`[items] (list) [matching-items]` Convert a sequence of C values to a Python list with the same number of items.

`{key:value} (dict) [matching-items]` Convert a sequence of C values to a Python dict with the same number of items.

Misc

`O (Python object) [HPy]` Pass an untouched Python object represented by the handle.

If the object passed in is a HPy_NULL, it is assumed that this was caused because the call producing the argument found an error and set an exception. Therefore, HPy_BuildValue will also immediately stop and return HPy_NULL but will not raise any new exception. If no exception has been raised yet, SystemError is set.

Any HPy handle passed to HPy_BuildValue is always owned by the caller. HPy_BuildValue never closes the handle nor transfers its ownership. If the handle is used, then HPy_BuildValue creates a duplicate of the handle.

`S (Python object) [HPy]` Alias for 'O'.

API

HPy **`HPy_BuildValue`** (HPyContext **ctx*, **`const`** char **fmt*, ...)

Creates a new value based on a format string from the values passed in variadic arguments.

Parameters

- **`ctx`** – The execution context.
- **`fmt`** – The format string (ASCII only; must not be `NULL`). For details, see *Supported Formatting Strings*.
- **`...`** – Variable arguments according to the provided format string.

Returns A handle to the built Python value or `HPy_NULL` in case of errors.

String Formatting Helpers

HPy string formatting helpers.

Note: these functions are runtime helper functions, i.e., they are not part of the HPy context ABI, but are available to HPy extensions to incorporate at compile time.

The formatting helper functions are: `HPyUnicode_FromFormat`, `HPyUnicode_FromFormatV`, and `HPyErr_Format`.

Supported Formatting Units

`%%` - The literal % character.

Compatible with C (s)printf:

`%c [int]`

`%d [int]`

`%u [unsigned int]`

`%ld [long]`

`%li [long]`

`%lu [unsigned long]`

`%lld [long long]`

`%lli [long long]`

`%llu [unsigned long long]`

`%zd [HPy_ssize_t]`

`%zi [HPy_ssize_t]`

`%zu [size_t]`

`%i [int]`

`%x [int]`

`%s [const char*]`

`%p [const void*]` Guaranteed to start with the literal ‘0x’ regardless of what the platform’s printf yields. However, there is no guarantee for zero-padding after the ‘0x’ prefix. Some systems pad the pointer to 32 or 64 digits depending on the architecture, some do not zero pad at all. Moreover, there is no guarantee whether the letters will be capitalized or not.

Python specific:

`%A [HPy]` The result of calling `HPy_Ascii`.

`%U [HPy]` A Unicode object.

`%V [HPy, const char*]` A Unicode object (which may be `HPy_NULL`) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is `HPy_NULL`).

`%S [HPy]` The result of calling `HPy_Str`.

`%R [HPy]` The result of calling `HPy_Repr`.

Additional flags:

The format is `%[0]{width}.{precision}{formatting-unit}`.

The `precision` flag for numbers gives the minimal number of digits (i.e., excluding the minus sign). Shorter numbers are padded with zeros. For strings it gives the maximum number of characters, i.e., the string may be shortened if it is longer than `precision`.

The `width` determines how many characters should be output. If the formatting result with `width` flag applied is shorter, then it is padded from left with spaces. If it is longer, the result will *not* be shortened.

The `0` flag is supported only for numeric units: if present, the number is padded to desired `width` with zeros instead of spaces. Unlike with spaces padding, the minus sign is shifted to the leftmost position with zero padding.

The `width` formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes for `%s` and `%V` (if the HPy argument is `HPy_NULL`), and a number of characters for `%A`, `%U`, `%S`, `%R` and `%V` (if the HPy argument is not `HPy_NULL`).

Compatibility with CPython API

HPy is more strict in these cases:

CPython API ignores width, precision, zero-padding flag for formatting units that do not support them: `%c` and `%p`, but HPy raises a system error in such cases.

CPython API ignores zero-padding for “string” formatting units, while HPy raises a system error in such cases.

Note: users should not rely on these system errors, as HPy may choose to support some of those flags in the future.

Struct Sequences

Struct sequences are subclasses of tuple. Similar to the API for creating tuples, HPy provides an API to create struct sequences. This is a builder API such that the struct sequence is guaranteed not to be written after it is created.

Note: There is no specific getter function for struct sequences. Just use one of *`HPy_GetItem()`*, *`HPy_GetItem_i()`*, or *`HPy_GetItem_s()`*.

`struct HPyStructSequence_Field`
Describes a field of a struct sequence.

`const` char ***`name`**
Name (UTF-8 encoded) for the field or `NULL` to end the list of named fields. Set the name to *`HPyStructSequence_UnnamedField`* to leave it unnamed.

`const` char ***`doc`**
Docstring of the field (UTF-8 encoded); may be `NULL`.

`struct HPyStructSequence_Desc`
Contains the meta information of a struct sequence type to create. Struct sequences are subclasses of tuple. The index in the *`fields`* array of the descriptor determines which field of the struct sequence is described.

`const` char ***`name`**
Name of the struct sequence type (UTF-8 encoded; must not be `NULL`).

`const` char ***`doc`**
Docstring of the type (UTF-8 encoded); may be `NULL`.

HPyStructSequence_Field ***fields**

Pointer to `NULL`-terminated array with field names of the new type (must not be `NULL`).

extern const char ***const HPyStructSequence_UnnamedField**

A marker that can be used as struct sequence field name to indicate that a field should be anonymous (i.e. cannot be accessed by a name but only by numeric index).

HPy **HPyStructSequence_NewType**(HPyContext **ctx*, *HPyStructSequence_Desc* **desc*)

Create a new struct sequence type from a descriptor. Instances of the resulting type can be created with `HPyStructSequence_New()`.

Parameters

- **ctx** – The execution context.
- **desc** – The descriptor of the struct sequence type to create (must not be `NULL`):

Returns A handle to the new struct sequence type or `HPy_NULL` in case of errors.

HPy **HPyStructSequence_New**(HPyContext **ctx*, HPy *type*, HPy_ssize_t *nargs*, HPy **args*)

Creates a new instance of `type` initializing it with the given arguments.

Since struct sequences are immutable objects, they need to be initialized at instantiation. This function will create a fresh instance of the provided struct sequence type. The type must have been created with `HPyStructSequence_NewType()`.

Parameters

- **ctx** – The execution context.
- **type** – A struct sequence type (must not be `HPy_NULL`). If the passed object is not a type, the behavior is undefined. If the given type is not appropriate, a `TypeError` will be raised.
- **nargs** – The number of arguments in `args`. If this argument is not exactly the number of fields of the struct sequence, a `TypeError` will be raised.
- **args** – An array of HPy handles to Python objects to be used for initializing the struct sequence. If `nargs > 0` then this argument must not be `NULL`.

Returns A new instance of `type` or `HPy_NULL` if an error occurred.

Misc Helpers

int **HPyHelpers_AddType**(HPyContext **ctx*, HPy *obj*, **const** char **name*, *HPyType_Spec* **hpyspec*, *HPyType_SpecParam* **params*)

Create a type and add it as an attribute on the given object. The type is created using `HPyType_FromSpec()`. The object is often a module that the type is being added to.

Parameters

- **ctx** – The execution context.
- **obj** – A handle to the object the type is being added to (often a module).
- **name** – The name of the attribute on the object to assign the type to.
- **hpyspec** – The type spec to use to create the type.
- **params** – The type spec parameters to use to create the type.

Returns `0` on failure, `1` on success.

Examples:

Using `HPyHelpers_AddType` without any `HPyType_SpecParam` parameters:

```
if (!HPyHelpers_AddType(ctx, module, "MyType", hpyspec, NULL))
    return HPy_NULL;
...
```

Using *HPyHelpers_AddType* with *HPyType_SpecParam* parameters:

```
HPyType_SpecParam params[] = {
    { HPyType_SpecParam_Base, ctx->h_LongType },
    { 0 }
};

if (!HPyHelpers_AddType(ctx, module, "MyType", hpyspec, params))
    return HPy_NULL;
...
```

int **`HPyHelpers_PackArgsAndKeywords`** (HPyContext **ctx*, **`const`** HPy **args*, size_t *nargs*, HPy *kwnames*, HPy **out_pos_args*, HPy **out_kwd*)

Convert positional/keyword argument vector to argument tuple and keywords dictionary.

This helper function is useful to convert arguments from HPy's calling convention to the legacy CPython *tp_call* calling convention. HPy's calling convention is similar to CPython's fastcall/vectorcall calling convention where positional and keyword arguments are passed as a C array, the number of positional arguments is explicitly given by an argument and the names of the keyword arguments are provided in a tuple.

For an example on how to use this function, see section *Incremental Migration to HPy's Calling Protocol*.

Parameters

- **`ctx`** – The execution context.
- **`args`** – A pointer to an array of positional and keyword arguments. This argument must not be `NULL` if `nargs > 0` or `HPy_Length(ctx, kwnames) > 0`.
- **`nargs`** – The number of positional arguments in `args`.
- **`kwnames`** – A handle to the tuple of keyword argument names (may be `HPy_NULL`). The values of the keyword arguments are also passed in `args` appended to the positional arguments. Argument `nargs` does not include the keyword argument count.
- **`out_pos_args`** – A pointer to a variable where to write the created positional arguments tuple to. If there are no positional arguments (i.e. `nargs == 0`), then `HPy_NULL` will be written. The pointer will not be used if any error occurs during conversion.
- **`out_kwd`** – A pointer to a variable where to write the created keyword arguments dictionary to. If there are not keyword arguments (i.e. `HPy_Length(ctx, kwnames) == 0`), then `HPy_NULL` will be written. The pointer will not be used if any error occurs during conversion.

Returns `0` on failure, `1` on success.

2.8.3 Inline Helper Functions

Inline Helper functions are `static inline` functions (written in C). Those functions are usually small convenience functions that everyone could write but in order to avoid duplicated effort, they are defined by HPy.

Inline Helpers

Inline Helper functions are `static inline` functions (written in C). Those functions are usually small convenience functions that everyone could write but in order to avoid duplicated effort, they are defined by HPy.

One category of inline helpers are functions that convert the commonly used but not fixed width C types, such as `int`, or `long long`, to HPy API. The HPy API always uses well-defined fixed width types like `int32` or `unsigned int8`.

`HPY_INLINE_HELPERS_H`

HPy **`HPyErr_SetFromErrno`** (HPyContext **ctx*, HPy *h_type*)

Same as *`HPyErr_SetFromErrnoWithFilenameObjects()`* but passes `HPy_NULL` to the optional arguments.

Parameters

- **`ctx`** – The execution context.
- **`h_type`** – The exception type to raise.

Returns always returns `HPy_NULL`

HPy **`HPyErr_SetFromErrnoWithFilenameObject`** (HPyContext **ctx*, HPy *h_type*, HPy *filename*)

Same as *`HPyErr_SetFromErrnoWithFilenameObjects()`* but passes `HPy_NULL` to the last (optional) argument.

Parameters

- **`ctx`** – The execution context.
- **`h_type`** – The exception type to raise.
- **`filename`** – a filename; may be `HPy_NULL`

Returns always returns `HPy_NULL`

HPy **`HPyTuple_Pack`** (HPyContext **ctx*, HPy_ssize_t *n*, ...)

Create a tuple from arguments.

A convenience function that will allocate a temporary array of `HPy` elements and use `HPyTuple_FromArray()` to create a tuple.

Parameters

- **`ctx`** – The execution context.
- **`n`** – The number of elements to pack into a tuple.
- **`...`** – Variable number of `HPy` arguments.

Returns A new tuple with `n` elements or `HPy_NULL` in case of an error occurred.

int **`HPy_DelAttr`** (HPyContext **ctx*, HPy *obj*, HPy *name*)

Delete an attribute.

This is the equivalent of the Python statement `del o.attr_name`.

Parameters

- **ctx** – The execution context.
- **obj** – The object with the attribute.
- **name** – The name (an unicode object) of the attribute.

Returns `0` on success; `-1` in case of an error.

int **HPy_DelAttr_s**(HPyContext **ctx*, HPy *obj*, **const** char **utf8_name*)
Delete an attribute.

This is the equivalent of the Python statement `del o.attr_name`.

Parameters

- **ctx** – The execution context.
- **obj** – The object with the attribute.
- **utf8_name** – The name (an UTF-8 encoded C string) of the attribute.

Returns `0` on success; `-1` in case of an error.

HPy **HPyLong_FromLong**(HPyContext **ctx*, long *l*)
Create a Python long object from a C `long` value.

Parameters

- **ctx** – The execution context.
- **l** – A C long value.

Returns A Python long object with the value of `l` or `HPy_NULL` on failure.

HPy **HPyLong_FromUnsignedLong**(HPyContext **ctx*, unsigned long *l*)
Create a Python long object from a C `unsigned long` value.

Parameters

- **ctx** – The execution context.
- **l** – A C `unsigned long` value.

Returns A Python long object with the value of `l` or `HPy_NULL` on failure.

HPy **HPyLong_FromLongLong**(HPyContext **ctx*, long long *l*)
Create a Python long object from a C `long long` value.

Parameters

- **ctx** – The execution context.
- **l** – A C `long long` value.

Returns A Python long object with the value of `l` or `HPy_NULL` on failure.

HPy **HPyLong_FromUnsignedLongLong**(HPyContext **ctx*, unsigned long long *l*)
Create a Python long object from a C `unsigned long long` value.

Parameters

- **ctx** – The execution context.
- **l** – A C `unsigned long long` value.

Returns A Python long object with the value of `l` or `HPy_NULL` on failure.

long **HPyLong_AsLong**(HPyContext **ctx*, HPy *h*)

Return a C `long` representation of the given Python long object. If the object is not an instance of Python long, the object's `__index__` method (if present) will be used to convert it to a Python long object.

This function will raise an `OverflowError` if the value of the object is out of range for a C `long`.

This function will raise a `TypeError` if:

- The object is neither an instance of Python long nor it provides an `__index__` method.
- If the `__index__` method does not return an instance of Python long.

Parameters

- **ctx** – The execution context.
- **h** – Either an instance of Python long or an object that provides an `__index__` method (which returns a Python long).

Returns A C `long` value. Errors will be indicated with return value `-1`. In this case, use *`HPyErr_Occurred()`* to disambiguate.

unsigned long **HPyLong_AsUnsignedLong**(HPyContext **ctx*, HPy *h*)

Return a C `unsigned long` representation of the given Python long object.

This function will raise a `TypeError` if the object is not an instance of Python long and it will raise an `OverflowError` if the object's value is negative or out of range for a C `unsigned long`.

Parameters

- **ctx** – The execution context.
- **h** – The object to convert to C `unsigned long` (must be an instance of Python long).

Returns A C `unsigned long` value. Errors will be indicated with return value `(unsigned long)-1`. In this case, use *`HPyErr_Occurred()`* to disambiguate.

unsigned long **HPyLong_AsUnsignedLongMask**(HPyContext **ctx*, HPy *h*)

Return a C `unsigned long` representation of the given Python long object. If the object is not an instance of Python long, the object's `__index__` method (if present) will be used to convert it to a Python long object.

If the object's value is out of range for an `unsigned long`, return the reduction of that value modulo `ULONG_MAX + 1`. Therefore, this function will **NOT** raise an `OverflowError` if the value of the object is out of range for a C `unsigned long`.

Parameters

- **ctx** – The execution context.
- **h** – Either an instance of Python long or an object that provides an `__index__` method (which returns a Python long).

Returns A C `unsigned long` value. Errors will be indicated with return value `(unsigned long)-1`. In this case, use *`HPyErr_Occurred()`* to disambiguate.

long long **HPyLong_AsLongLong**(HPyContext **ctx*, HPy *h*)

Return a C `long long` representation of the given Python long object. If the object is not an instance of Python long, the object's `__index__` method (if present) will be used to convert it to a Python long object.

This function will raise an `OverflowError` if the value of the object is out of range for a C `long long`.

This function will raise a `TypeError` if:

- The object is neither an instance of Python long nor it provides an `__index__` method.

- If the `__index__` method does not return an instance of Python long.

  Parameters

  - **ctx** – The execution context.
  - **h** – Either an instance of Python long or an object that provides an `__index__` method (which returns a Python long).

  Returns A C `long long` value. Errors will be indicated with return value `-1`. In this case, use *HPyErr_Occurred()* to disambiguate.

unsigned long long **HPyLong_AsUnsignedLongLong**(HPyContext **ctx*, HPy *h*)

Return a C `unsigned long long` representation of the given Python long object.

This function will raise a `TypeError` if the object is not an instance of Python long and it will raise an `OverflowError` if the object's value is negative or out of range for a C `unsigned long`.

Parameters

- **ctx** – The execution context.
- **h** – The object to convert to C `unsigned long long` (must be an instance of Python long).

Returns A C `unsigned long long` value. Errors will be indicated with return value `(unsigned long long)-1`. In this case, use *HPyErr_Occurred()* to disambiguate.

unsigned long long **HPyLong_AsUnsignedLongLongMask**(HPyContext **ctx*, HPy *h*)

Return a C `unsigned long long` representation of the given Python long object. If the object is not an instance of Python long, the object's `__index__` method (if present) will be used to convert it to a Python long object.

If the object's value is out of range for an `unsigned long long`, return the reduction of that value modulo `ULLONG_MAX + 1`. Therefore, this function will **NOT** raise an `OverflowError` if the value of the object is out of range for a C `unsigned long long`.

Parameters

- **ctx** – The execution context.
- **h** – Either an instance of Python long or an object that provides an `__index__` method (which returns a Python long).

Returns A C `unsigned long` value. Errors will be indicated with return value `(unsigned long long)-1`. In this case, use *HPyErr_Occurred()* to disambiguate.

HPy **HPyBool_FromLong**(HPyContext **ctx*, long *v*)

Returns Python `True` or `False` depending on the truth value of `v`.

Parameters

- **ctx** – The execution context.
- **v** – A C `long` value.

Returns Python `True` if `v != 0`; Python `False` otherwise.

HPy_ssize_t **HPySlice_AdjustIndices**(HPyContext **_HPy_UNUSED_ARG*) ctx

, HPy_ssize_t *length*, HPy_ssize_t **start*, HPy_ssize_t **stop*, HPy_ssize_t *step*Adjust start/end slice indices assuming a sequence of the specified length.

Out of bounds indices are clipped in a manner consistent with the handling of normal slices. This function cannot fail and does not call interpreter routines.

Parameters

- **`ctx`** – The execution context.
- **`length`** – The length of the sequence that should be assumed for adjusting the indices.
- **`start`** – Pointer to the start value (must not be `NULL`).
- **`stop`** – Pointer to the stop value (must not be `NULL`).
- **`step`** – The step value of the slice (must not be `0`)

Returns Return the length of the slice. Always successful. Doesn't call Python code.

HPy **`HPy_CallMethodTupleDict_s`**(HPyContext **ctx*, **const** char **utf8_name*, HPy *receiver*, HPy *args*, HPy *kw*)

Call a method of a Python object.

This is a convenience function for calling a method. It uses *`HPy_GetAttr_s()`* and *`HPy_CallTupleDict()`* to perform the method call.

Parameters

- **`ctx`** – The execution context.
- **`utf8_name`** – The name (UTF-8 encoded C string) of the method. Must not be `NULL`.
- **`receiver`** – A handle to the receiver of the call (i.e. the `self`). Must not be `HPy_NULL`.
- **`args`** – A handle to a tuple containing the positional arguments (must not be `HPy_NULL` but can, of course, be empty).
- **`kw`** – A handle to a Python dictionary containing the keyword arguments (may be `HPy_NULL`).

Returns The result of the call on success, or `HPy_NULL` in case of an error.

HPy **`HPy_CallMethodTupleDict`**(HPyContext **ctx*, HPy *name*, HPy *receiver*, HPy *args*, HPy *kw*)

Call a method of a Python object.

This is a convenience function for calling a method. It uses *`HPy_GetAttr()`* and *`HPy_CallTupleDict()`* to perform the method call.

Parameters

- **`ctx`** – The execution context.
- **`name`** – A handle to the name (a Unicode object) of the method. Must not be `HPy_NULL`.
- **`receiver`** – A handle to the receiver of the call (i.e. the `self`). Must not be `HPy_NULL`.
- **`args`** – A handle to a tuple containing the positional arguments (must not be `HPy_NULL` but can, of course, be empty).
- **`kw`** – A handle to a Python dictionary containing the keyword arguments (may be `HPy_NULL`).

Returns The result of the call on success, or `HPy_NULL` in case of an error.

2.9 Contributing

2.9.1 Getting Started

TBD

2.9.2 Adding New API

1. Add the function to `hpy/tools/autogen/public_api.h`. If the CPython equivalent function name is not the same (after removing the leading `H`, add an appropriate CPython function mapping in `hpy/tools/autogen/conf.py`. If the signature is complicated or there is no clear equivalent function, the mapping should be `None`, and follow the directions in the next step. Otherwise all the needed functions will be autogenerated.
2. If the function cannot be autogenerated (i.e. the mapping does not exist), you must write the wrapper by hand. Add the function to `NO_WRAPPER` in `hpy/tools/autogen/debug.py`, and add a `ctx_fname` function to `hyp/devel/src/runtime/*.c` (possibly adding the new file to `setup.py`), add a debug wrapper to `hpy/debug/src/debug_ctx.c`, add a implementation that uses the `ctx` variant to `hpy/devel/include/hpy/cpython/misc.h` and add the declaration to `hpy/devel/include/hpy/runtime/ctx_funcs.h`.
3. Run `make autogen` which will turn the mapping into autogenerated functions
4. Add a test for the functionality
5. Build with `python setup.py build_ext`. After that works, build with `python -m pip install -e .`, then run the test with `python -m pytest ...`.

2.10 Misc Notes

2.10.1 Embedding HPy modules

There might be cases where it is beneficial or even necessary to embed multiple HPy modules into one library. HPy itself already makes use of that. The debug and the trace module do not have individual libraries but are embedded into the universal module.

To achieve that, the embedder will use the macro *`HPy_MODINIT`* several times. Unfortunately, this macro defines global state and cannot repeatedly be used by default. In order to correctly embed several HPy modules into one library, the embedder needs to consider following:

- The modules must be compiled with preprocessor macro *`HPY_EMBEDDED_MODULES`* defined to enable this feature.
- There is one major restriction: All HPy-specific module pieces must be in the same compilation unit. *HPy-specific pieces* are things like the module's init function (`HPy_MODINIT`) and all slots, members, methods of the module or any type of it (`HPyDef_*`). The implementation functions (usually the `*_impl` functions) of the slots, members, methods, etc. and any helper functions may still be in different compilation units. The reason for this is that the global state induced by `HPy_MODINIT` is, of course, made local (e.g. using C modifier `static`).
- It is also necessary to use macro *`HPY_MOD_EMBEDDABLE`* before the first usage of any `HPyDef_*` macro.

Also refer to the API reference *HPy Module*.

Example

```
// compile with -DHPY_EMBEDDED_MODULES

HPY_MOD_EMBEDDABLE(hpymodA)

HPyDef_METH(foo, /* ... */)
static HPy foo_impl(/* ... */)
{
    // ...
}

HPy_MODINIT(extension_name, hpymodA)
```

2.11 Changelog

2.11.1 Version 0.9 (April 25th, 2023)

This release adds numerous major features and indicates the end of HPy's *alhpa* phase. We've migrated several key packages to HPy (for a list, see our website https://hpyproject.org) and we are now confident that HPy is mature enough for being used as serious extension API. We also plan that the next major release will be `1.0`.

Major new features

Support subclasses of built-in types It is now possible to create pure HPy types that inherit from built-in types like `type` or `float`. This was already possible before but in a very limited way, i.e., by setting *`HPyType_Spec.basicsize`* to `0`. In this case, the type implicitly inherited the basic size of the supertype but that also means that you cannot have a custom C struct. It is now possible inherit from a built-in type **AND** have a custom C struct. For further reference, see *`HPyType_Spec.builtin_shape`* and *`HPyType_BuiltinShape`*.

Support for metaclasses HPy now supports creating types with metaclasses. This can be done by passing type specification parameter with kind *`HPyType_SpecParam_Metaclass`* when calling *`HPyType_FromSpec()`*.

HPy Hybrid ABI In addition to *CPython ABI* and *HPy Universal ABI*, we now introduced the Hybrid ABI. The major difference is that whenever you use a legacy API like `HPy_AsPyObject()` or `HPy_FromPyObject()`, the prdouced binary will then be specific to one interpreter. This was necessary to ensure that universal binaries are really portable and can be used on any HPy-capable interpreter.

Trace Mode Similar to the *Debug Mode*, HPy now provides the Trace Mode that can be enabled at runtime and helps analyzing API usage and identifying performance issues.

Multi-phase Module Initialization HPy now support multi-phase module initialization which is an important feature in particular needed for two important use cases: (1) module state support (which is planned to be introduced in the next major release), and (2) subinterpreters. We decided to drop support for single-phase module initialization since this makes the API cleaner and easier to use.

HPy *Calling Protocol* This was a big missing piece and is now eventually available. It enables slot `HPy_tp_call`, which can now be used in the HPy type specification. We decided to use a calling convention similar to CPython's vectorcall calling convention. This is: the arguments are passed in a C array and the keyword argument names are provided as a Python tuple. Before this release, the only way to create a callable type was to set the special method `__call__`. However, this has several disadvantages. In particlar, poor performance on CPython (and maybe other implementations) and it was not possible to have specialized call function implementations per object (see *`HPy_SetCallFunction()`*)

Added APIs

Deleting attributes and items *HPy_DelAttr()*, *HPy_DelAttr_s()*, *HPy_DelItem()*, *HPy_DelItem_i()*, *HPy_DelItem_s()*

Capsule API HPyCapsule_New(), HPyCapsule_IsValid(), HPyCapsule_Get(), HPyCapsule_Set()

Eval API *HPy_Compile_s()* and *HPy_EvalCode()*

Formatting helpers HPyUnicode_FromFormat() and HPyErr_Format()

Contextvar API HPyContextVar_New(), HPyContextVar_Get(), HPyContextVar_Set()

Unicode API HPyUnicode_FromEncodedObject() and HPyUnicode_Substring()

Dict API *HPyDict_Keys()* and *HPyDict_Copy()*

Type API *HPyType_GetName()* and *HPyType_IsSubtype()*

Slice API HPySlice_Unpack() and *HPySlice_AdjustIndices()*

Structseq API *HPyStructSequence_NewType()*, *HPyStructSequence_New()*

Call API *HPy_Call()*, *HPy_CallMethod()*, *HPy_CallMethodTupleDict()*, *HPy_CallMethodTupleDict_s()*

HPy call protocol *HPy_SetCallFunction()*

Debug mode

- Detect closing and returning (without dup) of context handles
- Detect invalid usage of stored HPyContext * pointer
- Detect invalid usage of tuple and list builders
- Added Windows support for checking invalid use of raw data pointers (e.g HPyUnicode_AsUTF8AndSize) after handle was closed.
- Added support for backtrace on MacOS

Documentation

- Added incremental *Porting Example*
- Added *HPy Quickstart* guide
- Extended *API Reference*
- Added *HPy Core API Function Index*
- Added possiblity to generate examples from tests with argument --dump-dir (see *HPy unit tests*)
- Added initial *Contributing* docs

Incompatible changes to version 0.0.4

- Simplified `HPyDef_*` macros
- Changed macro *`HPy_MODINIT`* because of multi-phase module init support.
- Replace environment variable `HPY_DEBUG` by `HPY` (see *Debug Mode* or *Trace Mode*).
- Changed signature of `HPyFunc_VARARGS` and `HPyFunc_ KEYWORDS` to align with HPy's call protocol calling convention.

Supported Python versions

- Added Python 3.11 support
- Preliminary Python 3.12 support
- Dropped Python 3.6 support (since EOL)
- Dropped Python 3.7 support (since EOL by June 2023)

Misc

- Ensure deterministic auto-generation
- Ensure ABI backwards compatibility
  - Explicitly define slot within HPyContext of function pointers and handles
  - Compile HPy ABI version into binary and verify at load time
- Added proper support for object members `HPyMember_OBJECT`
- Changed `HPyBytes_AsString()` and `HPyBytes_AS_STRING()` to return `const char *`
- Use fixed-width integers in context functions

2.11.2 Version 0.0.4 (May 25th, 2022)

New Features/API:

- HPy headers are C++ compliant
- Python 3.10 support
- HPyField: References to Python objects that can be stored in raw native memory owned by Python objects.
  - New API functions: `HPyField_Load`, `HPyField_Store`
- HPyGlobal: References to Python objects that can be stored into a C global variable.
  - New API functions: `HPyGlobal_Load`, `HPyGlobal_Store`
  - Note: `HPyGlobal` does not allow to share Python objects between (sub)interpreters
- GIL support - New API functions: `HPy_ReenterPythonExecution`, `HPy_LeavePythonExecution`
- Value building support (`HPy_BuildValue`)
- New type slots
  - `HPy_mp_ass_subscript`, `HPy_mp_length`, `HPy_mp_subscript`
  - `HPy_tp_finalize`

- Other new API functions
  - `HPyErr_SetFromErrnoWithFilename`, `HPyErr_SetFromErrnoWithFilenameObjects`
  - `HPyErr_ExceptionMatches`
  - `HPyErr_WarnEx`
  - `HPyErr_WriteUnraisable`
  - `HPy_Contains`
  - `HPyLong_AsVoidPtr`
  - `HPyLong_AsDouble`
  - `HPyUnicode_AsASCIIString`, `HPyUnicode_DecodeASCII`
  - `HPyUnicode_AsLatin1String`, `HPyUnicode_DecodeLatin1`
  - `HPyUnicode_DecodeFSDefault`, `HPyUnicode_DecodeFSDefaultAndSize`
  - `HPyUnicode_ReadChar`

Debug mode:

- Support activation of debug mode via environment variable `HPY_DEBUG`
- Support capturing stack traces of handle allocations
- Check for invalid use of raw data pointers (e.g `HPyUnicode_AsUTF8AndSize`) after handle was closed.
- Detect invalid handles returned from extension functions
- Detect incorrect closing of handles passed as arguments

Misc Changes:

- Removed unnecessary prefix `"m_"` from fields of `HPyModuleDef` (incompatible change)
- For HPy implementors: new pytest mark for HPy tests assuming synchronous GC

2.11.3 Version 0.0.3 (September 22nd, 2021)

This release adds various new API functions (see below) and extends the debug mode with the ability to track closed handles. The default ABI mode now is 'universal' for non-CPython implementations. Also, the type definition of `HPyContext` was changed and it's no longer a pointer type. The name of the HPy dev package was changed to 'hpy' (formerly: 'hpy.devel'). Macro HPy_CAST was replaced by HPy_AsStruct.

New features:

- Added helper HPyHelpers_AddType for creating new types
- Support format specifier 's' in HPyArg_Parse
- Added API functions: HPy_Is, HPy_AsStructLegacy (for legacy types), HPyBytes_FromStringAndSize, HPyErr_NewException, HPyErr_NewExceptionWithDoc, HPyUnicode_AsUTF8AndSize, HPyUnicode_DecodeFSDefault, HPyImport_ImportModule
- Debug mode: Implemented tracking of closed handles
- Debug mode: Add hook for invalid handle access

Bug fixes:

- Distinguish between pure and legacy types
- Fix Sphinx doc errors

CHAPTER THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX