
HPy

Release 0.0.1

HPy Collective

Jun 02, 2022

CONTENTS

1	HPy overview	3
1.1	Motivation and goals	3
1.2	API vs ABI	4
1.3	Target ABIs	4
1.4	C extensions	5
1.5	Cython extensions	5
1.6	Extensions in other languages	5
1.7	Benefits for alternative Python implementations	6
1.8	Current status and roadmap	6
1.9	Early benchmarks	7
1.10	Projects involved	7
1.11	Related work	8
2	HPy API introduction	9
2.1	Handles	9
2.2	HPyContext	11
2.3	A simple example	11
3	Porting guide	15
3.1	Porting PyObject* to HPy API constructs	15
3.2	PyModule_AddObject	17
3.3	Py_tp_dealloc	17
3.4	Py_tp_methods, Py_tp_members and Py_tp_getset	17
3.5	PyList_New/PyList_SET_ITEM	17
3.6	PyObject_Call and PyObject_CallObject	18
3.7	Buffers	18
4	Debug Mode	19
4.1	Activating Debug Mode	19
4.2	Using Debug Mode	20
4.3	Example	20
5	API Reference	23
5.1	HPyField	23
5.2	HPyGlobal	24
5.3	Leave/enter Python execution (GIL)	24
5.4	Argument Parsing	25
5.5	Building complex Python objects	29
5.6	Runtime Helpers	30
6	Misc notes	33

6.1	bytes/str building API	33
7	Changelog	39
7.1	Version 0.0.3 (September 22nd, 2021)	39
8	Indices and tables	41
	Index	43

HPy provides a new API for extending Python in C.

The official [Python/C API](#) 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, GraalPython, Jython, IronPython, etc.)
- to experiment with new things inside CPython itself: e.g. using a GC instead of refcounting, or to remove the GIL.

There are several advantages to write your C extension in HPy:

- it runs much faster on PyPy, GraalPython, and at native speed on CPython
- it is possible to compile a single binary which runs unmodified on all supported Python implementations and versions
- it is simpler and more manageable than the Python/C API
- it provides an improved debugging experience: in “debug mode”, HPy actively checks for many common mistakes such as reference leaks and invalid usage of objects after they have been deleted. It is possible to turn the “debug mode” on at startup time, without needing to recompile Python or the extension itself

HPY OVERVIEW

1.1 Motivation and goals

The biggest quality of the Python ecosystem is to have a huge number of high quality libraries for all kind of jobs. Many of them, especially in the scientific community, are written in C and exposed to Python using the [Python/C API](#). However, the Python/C API exposes a number of CPython’s implementation details and low-level data structure layouts. This has two important consequences:

1. any alternative implementation which wants to support C extensions needs to either follow the same low-level layout or to 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 the Python/C API in an efficient way is [challenging, if not impossible](#). The main goal of HPy is provide a **C API which is possible to implement in an efficient way on a number of very diverse implementations**. 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/C 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/C 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*.

Hide internal details The API is designed to allow a lot of flexibility for Python implementations, allowing the possibility to explore different choices to the ones used by CPython. In particular, **reference counting is not part of the API**: we want a more generic way of managing resources which 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, GraalPython or Java, for example).

Moreover, we want to avoid exposing internal details of a specific implementation, so that each implementation can experiment with new memory layout of objects, add optimizations, etc.

Simplicity The HPy API aims to be smaller and easier to study/use/manage than the existing Python/C 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.

1.2 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 3.7, you get a DLL called `simple.cpython-37m-x86_64-linux-gnu.so`:

- `cpython-37m` is the ABI tag, in this case CPython 3.7
- `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.pypy3-72-x86_64-linux-gnu.so`:

- `pypy3-72` is the ABI tag, in this case “PyPy3.x”, version “7.2.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.

1.3 Target ABIs

Depending on the compilation options, and 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. The ABI tag is defined by the version of CPython which is used to compile it (e.g., `cpython-37m`),

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 supports it. PyPy and GraalPython support it natively. CPython supports it by using the `hpy.universal` package, and there is a small speed penalty compared to the CPython ABI. The ABI tag has not been formally defined yet.

HPy Hybrid ABI To allow an incremental transition to HPy, it is possible to use both HPy and Python/C API calls in the same extension. In this case, it is not possible to target the Universal ABI because the resulting compiled library also needs to be compatible with a specific CPython version. The ABI tag will be something like `hpy-1_cpython-37m`. Note: the tag is not implemented yet. Currently, the approach to use HPy in hybrid

mode is to build the extension in HPy universal mode, which, for now, still allows mixing the HPy and CPython APIs. Extensions mixing the HPy and CPython APIs will not work on Pythons that do not support the hybrid ABI.

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.

1.4 C extensions

If you are writing a Python extension in C, you are a consumer of the HPy API. There are three big advantages in using HPy instead of the old Python/C API:

- Speed on PyPy, GraalPython and other alternative implementations: according to early *Early benchmarks*, an extension written in HPy can be ~3x faster than the equivalent extension written in Python/C.
- Improved debugging: when you load extensions in *Debug Mode*, many common mistakes are checked and reported automatically.
- Universal binaries: you can choose to distribute only Universal ABI binaries. This comes with a small speed penalty on CPython, but for non-performance critical libraries it might still be a good tradeoff.

1.5 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 Python/C code: once this is done, you will get the benefits of HPy automatically.

1.6 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*.

1.7 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/C 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.

1.8 Current status and roadmap

HPy is still in the early stages of development, but many big pieces are already in place. As on April 2022, the following milestones have been reached:

- some real-world Python packages have been ported to HPy API. The ports will be published soon.
- 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.
- wheels can be build 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 [GraalPython](#).

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/C 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 HPy-Context 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.

1.9 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 outweigh 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.

1.10 Projects involved

HPy was born during EuroPython 2019, where a small group of people started to discuss the problems of the Python/C 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 endorses or recognizes HPy in any way, just that some of the people involved contributed to the code/design/discussions of HPy:

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

1.11 Related work

A partial list of alternative implementations which offer a Python/C compatibility layer include:

- [PyPy](#)
- [Jython](#)
- [IronPython](#)
- [GraalPython](#)

HPY API INTRODUCTION

Warning: HPy is still in the early stages of development and the API may change.

2.1 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/C 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()`.

2.1.1 Handles vs `PyObject *`

In the old Python/C API, multiple `PyObject *` references to the same object are completely equivalent to each other. Therefore they can be passed to Python/C 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 Python/C code:

```
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:

```
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: The debug mode is a good example of how powerful it is to decouple the lifetime of handles and the lifetime of an 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()`, the HPy debug mode is able to tell the precise code location which created the unclosed handle. Similarly, if you try to operate on a closed handle, it will tell you the precise code locations which created and closed it.

The other important difference is that Python/C guarantees that multiple references to the same object results in the very same `PyObject *` pointer. Thus, it is possible to compare C pointers by equality to check whether they point 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 the semantics of handles is that it allows implementations 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 for memory management: 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 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 object moves, the GC fixes the address in the list, without having to touch all the handles which have been passed to C.

2.2 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/C 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.

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 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/C API, so we will underline the similarities and the differences with it.

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

```
#include "hpy.h"

HPyDef_METH(myabs, "myabs", myabs_impl, HPyFunc_O)
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`.
- It uses the `HPyFunc_O` calling convention. Like `METH_O` in Python/C API, `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.

Now, we can define our module:

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

static HPyModuleDef simple = {
    .name = "simple",
    .doc = "HPy Example",
    .size = -1,
    .defines = SimpleMethods,
    .legacy_methods = NULL
};
```

This part is very similar to the one you would write in Python/C. 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/C 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/C API.

Finally, `HPyModuleDef` is basically the same as the old `PyModuleDef`:

```
HPy_MODINIT(simple)
HPy init_simple_impl(HPyContext *ctx) {
    return HPyModule_Create(ctx, &simple);
}
```

2.3.1 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.

2.3.2 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", add_ints_impl, HPyFunc_VARARGS)
static HPy add_ints_impl(HPyContext *ctx, HPy self, HPy *args, HPy_ssize_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/C `METH_VARARGS`: in particular, instead of taking a `PyObject *args`, we take an array of `HPy` and its size. This allows e.g. `PyPy` to do a call 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,
};
```


PORTING GUIDE

3.1 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 GraalPython, *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 move 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 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.

IMPORTANT: if you write a custom type having `HPyFields`, you **MUST** also write a `tp_traverse` slot. Note that this is different than the old Python/C API, where you need `tp_traverse` only under certain conditions. See the next section for more details.

IMPORTANT: the contract of `tp_traverse` is that it must visit all the `HPyFields` 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 the `HPyFields` by other means (for example, when they track them internally already). The debug mode will check this contract.

3.1.1 `tp_traverse`, `tp_clear`, `Py_TPFLAGS_HAVE_GC`

Let’s quote the Python/C 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.

3.1.2 `tp_dealloc` and `Py_DECREF`

Generally speaking, if you have one or more `PyObject *` fields in the old Python/C, 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`.

3.2 `PyModule_AddObject`

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

3.3 `Py_tp_dealloc`

`Py_tp_dealloc` becomes `HPy_tp_destroy`. We changed the name a little bit because only “lightweight” destructors are supported. Use `tp_finalize` if you really need to do things with the context or with the handle of the object.

3.4 `Py_tp_methods`, `Py_tp_members` and `Py_tp_getset`

`Py_tp_methods`, `Py_tp_members` and `Py_tp_getset` are no longer needed. Methods, members and getsets are specified “flatly” together with the other slots, using the standard mechanism of `HPyDef_{METH, MEMBER, GETSET}` and `HPyType_Spec.defines`.

3.5 `PyList_New/PyList_SET_ITEM`

`PyList_New(5)/PyList_SET_ITEM()` becomes:

```
HPyListBuilder builder = HPyListBuilder_New(ctx, 5);
HPyListBuilder_Set(ctx, builder, 0, h_item0);
...
HPyListBuilder_Append(ctx, builder, h_item5);
...
HPy h_list = HPyListBuilder_Build(ctx, builder);
```

For lists of (say) integers:

```
HPyListBuilder_i builder = HPyListBuilder_i_New(ctx, 5);
HPyListBuilder_i_Set(ctx, builder, 0, 42);
...
HPy h_list = HPyListBuilder_i_Build(ctx, builder);
```

And similar for building tuples or bytes

3.6 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.

3.7 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.

DEBUG MODE

HPy includes a debug mode which includes a lot of useful run-time checks to ensure that C extensions use the API correctly. The major points of the debug mode are:

1. no special compilation flags are required: it is enough to compile the extension with the Universal ABI.
2. The debug mode can be activated at *import time*, and it can be activated per-extension.
3. You pay the overhead of the 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 (that wraps the normal context) without affecting the performance of the regular context at all.

The debugging context can already check for:

- Leaked handles.
- Handles used after they are closed.
- Reading from a memory which is no longer guaranteed to be still valid, for example, the buffer returned by `HPyUnicode_AsUTF8AndSize` after the corresponding HPy handle was closed.
- Writing to a memory which should be read-only, for example, the buffer returned by `HPyUnicode_AsUTF8AndSize`.

4.1 Activating Debug Mode

The debug mode works only for extensions built with HPy universal ABI.

To enable the debug mode, use environment variable `HPY_DEBUG`. If `HPY_DEBUG=1`, then all HPy modules are loaded with debug context. Alternatively `HPY_DEBUG` can be set to a comma separated list of names of the modules that should be loaded in the debug mode.

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

```
mod = hpy.universal.load(module_name, so_filename, debug=True)
```

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

4.2 Using Debug Mode

The HPy debug module exposes `LeakDetector` class for detection of leaked handles. `LeakDetector` can be used to check that some code does not leave behind unclosed 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 exposes pytest fixture `hpy_debug` that, for the time being, enables the `LeakDetector`, but 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
```

ATTENTION: the usage of `LeakDetector` or `hpy_debug` by itself does not enable the HPy debug mode! If the debug mode is not enabled for any extension, then those features do nothing useful (but also nothing harmful).

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()
```

4.3 Example

Following HPy function leaks a handle:

```
HPyDef_METH(test_leak_stacktrace, "test_leak_stacktrace",
            test_leak_stacktrace_impl, 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=1
import hpy.debug
import snippets

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+gd7ecec6.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+gd7ecec6.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 (`LD_FLAGS="-rdynamic"`)
- build your code without optimizations and with debug symbols (`C_FLAGS="-O0 -g"`)
- use `addr2line` command line utility, e.g.: `addr2line -e /path/to/snippets.hpy.so -C -f +0x122c`

API REFERENCE

WARNING: Generated API reference documentation is work in progress. Some parts of the API are not included in this documentation yet.

5.1 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: contrarily than 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 decref. 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 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.

5.2 HPyGlobal

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

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 stores directly 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.

5.3 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`.

5.4 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.

5.4.1 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.

5.4.2 Argument Parsing API

int **HPyArg_Parse** (HPyContext *ctx, HPyTracker *ht, HPy *args, HPy_ssize_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 no formatting options to this function require 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, an exception 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, HPy *args, HPy_ssize_t nargs, HPy kw, **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.
- **kw** – A handle to the dictionary of keyword arguments.
- **fmt** – The format string to use to parse the arguments.
- **keywords** – An *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 precede 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, an exception 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, kw, "ll", &a, &b))
    return HPy_NULL;
...
```

Using *HPyArg_ParseKeywords* with an *HPyTracker*:

```
HPy a, b;
HPyTracker ht;
if (!HPyArg_ParseKeywords(ctx, &ht, args, nargs, kw, "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.

5.5 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.

5.5.1 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.
- 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'.

5.5.2 API

5.6 Runtime Helpers

Runtime helper functions.

These are not part of the HPy context or ABI, but are available for HPy extensions to incorporate at compile time.

5.6.1 Runtime Helpers API

`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))
```

(continues on next page)

(continued from previous page)

```
    return HPy_NULL;  
    ...
```


MISC NOTES

6.1 bytes/str building API

We need to design an HPy API to build `bytes` and `str` objects. Before making any proposal, it is useful to understand:

1. What is the current API to build strings.
2. What are the constraints for alternative implementations and the problems of the current C API.
3. What are the patterns used to build string objects in the existing extensions.

Some terminology:

- “string” means both `bytes` and `str` objects
- “unicode” or “unicode string” indicates `str`

Note: In this document we are translating `PyUnicode_*` functions into `HPyStr_*`. See [issue #213](#) for more discussion about the naming convention.

Note: The goal of the document is only to describe the current CPython API and its real-world usage. For a discussion about how to design the equivalent HPy API, see [issue #214](#)

6.1.1 Current CPython API

Bytes

There are essentially two ways to build `bytes`:

1. Copy the content from an existing C buffer:

```
PyObject* PyBytes_FromString(const char *v);
PyObject* PyBytes_FromStringAndSize(const char *v, Py_ssize_t len);
PyObject* PyBytes_FromFormat(const char *format, ...);
```

2. Create an uninitialized buffer and fill it manually:

```
PyObject s = PyBytes_FromStringAndSize(NULL, size);
char *buf = PyBytes_AS_STRING(s);
strcpy(buf, "hello");
```

(1) is easy for alternative implementations and we can probably provide an HPy equivalent without changing much, so we will concentrate on (2): let's call it "raw-buffer API".

Unicode

Similarly to bytes, there are several ways to build a `str`:

```
PyObject* PyUnicode_FromString(const char *u);
PyObject* PyUnicode_FromStringAndSize(const char *u, Py_ssize_t size);
PyObject* PyUnicode_FromKindAndData(int kind, const void *buffer, Py_ssize_t size);
PyObject* PyUnicode_FromFormat(const char *format, ...);
PyObject* PyUnicode_New(Py_ssize_t size, Py_UCS4 maxchar);
```

Note: `PyUnicode_FromString{,AndSize}` take an UTF-8 string in input

The following functions are used to initialize an uninitialized object, but I could not find any usage of them outside CPython itself, so I think they can be safely ignored for now:

```
Py_ssize_t PyUnicode_Fill(PyObject *unicode, Py_ssize_t start, Py_ssize_t length, Py_
↳ UCS4 fill_char);
Py_ssize_t PyUnicode_CopyCharacters(PyObject *to, Py_ssize_t to_start, PyObject *from,
↳ Py_ssize_t from_start, Py_ssize_t how_many);
```

There are also a bunch of API functions which have been deprecated (see [PEP 623](#) and [PEP 624](#)) so we will not take them into account. The deprecated functions include but are not limited to:

```
PyUnicode_FromUnicode
PyUnicode_FromStringAndSize(NULL, ...) // use PyUnicode_New instead
PyUnicode_AS_UNICODE
PyUnicode_AS_DATA
PyUnicode_READY
```

Moreover, CPython 3.3+ adopted a flexible string representation ([PEP 393](#)) which means that the underlying buffer of `str` objects can be an array of 1-byte, 2-bytes or 4-bytes characters (the so called "kind").

`str` objects offer a raw-buffer API, but you need to call the appropriate function depending on the kind, returning buffers of different types:

```
typedef uint32_t Py_UCS4;
typedef uint16_t Py_UCS2;
typedef uint8_t Py_UCS1;
Py_UCS1* PyUnicode_1BYTE_DATA(PyObject *o);
Py_UCS2* PyUnicode_2BYTE_DATA(PyObject *o);
Py_UCS4* PyUnicode_4BYTE_DATA(PyObject *o);
```

Uninitialized unicode objects are created by calling `PyUnicode_New(size, maxchar)`, where `maxchar` is the maximum allowed value of a character inside the string, and determines the kind. So, in cases in which `maxchar` is known in advance, we can predict at compile time what will be the kind of the string and write code accordingly. E.g.:

```
// ASCII only --> kind == PyUnicode_1BYTE_KIND
PyObject *s = PyUnicode_New(size, 127);
Py_UCS1 *buf = PyUnicode_1BYTE_DATA(s);
strcpy(buf, "hello");
```

Note: CPython distinguishes between `PyUnicode_New(size, 127)` and `PyUnicode_New(size, 255)`: in both cases the kind is `PyUnicode_1BYTE_KIND`, but the former also sets a flag to indicate that the string is ASCII-only.

There are cases in which you don't know the kind in advance because you are working on generic data. To solve the problem in addition to the raw-buffer API, CPython also offers an “Opaque API” to write a char inside an unicode:

```
int PyUnicode_WriteChar(PyObject *unicode, Py_ssize_t index, Py_UCS4 character)
void PyUnicode_WRITE(int kind, void *data, Py_ssize_t index, Py_UCS4 value)
```

Note that the character to write is always `Py_UCS4`, so `_WriteChar/_WRITE` have logic to do something different depending on the kind.

Note: `_WRITE` is a macro, and its implementation contains a `switch(kind)`: I think it is designed with the explicit goal of allowing the compiler to hoist the `switch` outside a loop in which we repeatedly call `_WRITE`. However, it is worth noting that I could not find any code using it outside CPython itself, so it's probably something which we don't need to care of for HPy.

6.1.2 Raw-buffer vs Opaque API

There are two ways to initialize a non-initialized string object:

- **Raw-buffer API:** get a C pointer to the memory and fill it directly: `PyBytes_AsString`, `PyUnicode_1BYTE_DATA`, etc.
- **Opaque API:** call special functions API to fill the content, without accessing the buffer directly: e.g., `PyUnicode_WriteChar`.

From the point of view of the implementation, a completely opaque API gives the most flexibility in terms of how to implement a builder and/or a string. A good example is PyPy's `str` type, which uses UTF-8 as the internal representation. A completely opaque `HPyStrBuilder` could allow PyPy to fill directly its internal UTF-8 buffer (at least in simple cases). On the other hand, a raw-buffer API would force PyPy to store the UCS{1,2,4} bytes in a temporary buffer and convert them to UTF-8 during the `build()` phase.

On the other hand, from the point of view of the C programmer it is easier to have direct access the memory. This allows to:

- use `memcpy()` to copy data into the buffer
- pass the buffer directly to other C functions which write into it (e.g., `read()`)
- use standard C patterns such as `*p++ = ...` or similar.

6.1.3 Problems and constraints

`bytes` and `str` are objects are immutable: the biggest problem of the current API boils down to the fact that the API allows to construct objects which are not fully initialized and to mutate them during a not-well-specified “initialization phase”.

Problems for alternative implementations:

1. it assumes that the underlying buffer **can** be mutated. This might not be always the case, e.g. if you want to use a Java string or an RPython string as the data buffer. This might also lead to unnecessary copies.
2. It makes harder to optimize the code: e.g. a JIT cannot safely assume that a string is actually immutable.

3. It interacts badly with a moving GC, because we need to ensure that `buf` doesn't move.

Introducing a builder solves most of the problems, because it introduces a clear separation between the mutable and immutable phases.

6.1.4 Real world usage

In this section we analyze the usage of some string building API in real world code, as found in the [Top 4000 PyPI packages](#).

PyUnicode_New

This is the recommended “modern” way to create `str` objects but it's not widely used outside CPython. A simple `grep` found only 17 matches in the 4000 packages, although some are in very important packages such as `ffi`, `markupsafe` (1, 2, 3) and `simplejson` (1, 2).

In all the examples linked above, `maxchar` is hard-coded and known at compile time.

There are only four usages of `PyUnicode_New` in which `maxchar` is actually unknown until runtime, and it is curious to note that the first three are in runtime libraries used by code generators:

1. `mypyc`
2. `Cython`
3. `siplib`
4. `PyICU`: this is the only non-runtime library usage of it, and it's used to implement a routine to create a `str` object from an UTF-16 buffer.

For HPy, we should at least consider the opportunity to design special APIs for the cases in which `maxchar` is known in advance, e.g. `HPyStrBuilder_ASCII`, `HPyStrBuilder_UCS1`, etc., and evaluate whether this would be beneficial for alternative implementations.

Create empty strings

A special case is `PyUnicode_New(0, 0)`, which constructs an empty `str` object. CPython special-cases it to always return a prebuilt object.

This pattern is used a lot inside CPython but only once in 3rd-party extensions, in the `regex` library (1, 2).

Other ways to build empty strings are `PyUnicode_FromString("")` which is used 27 times and `PyUnicode_FromStringAndSize("", 0)` which is used only once.

For HPy, maybe we should just have a `ctx->h_EmptyStr` and `ctx->h_EmptyBytes`?

PyUnicode_From*, PyUnicode_Decompose*

Functions of the `PyUnicode_From*` and `PyUnicode_Decompose*` families should be easy to adapt to HPy, so we won't discuss them in detail. However, here is the of matches found by `grep` for each function, to get an idea of how much each is used:

`PyUnicode_From*` family:

Documented:

```

964 PyUnicode_FromString
259 PyUnicode_FromFormat
125 PyUnicode_FromStringAndSize
58 PyUnicode_FromWideChar
48 PyUnicode_FromEncodedObject
17 PyUnicode_FromKindAndData
9 PyUnicode_FromFormatV

```

Undocumented:

```

7 PyUnicode_FromOrdinal

```

Deprecated:

```

66 PyUnicode_FromObject
45 PyUnicode_FromUnicode

```

PyUnicode_Decode* family:

```

143 PyUnicode_DecodeFSDefault
114 PyUnicode_DecodeUTF8
99 PyUnicode_Decode
64 PyUnicode_DecodeLatin1
51 PyUnicode_DecodeASCII
12 PyUnicode_DecodeFSDefaultAndSize
10 PyUnicode_DecodeUTF16
8 PyUnicode_DecodeLocale
6 PyUnicode_DecodeRawUnicodeEscape
3 PyUnicode_DecodeUTF8Stateful
2 PyUnicode_DecodeUTF32
2 PyUnicode_DecodeUnicodeEscape

```

Raw-buffer access

Most of the real world packages use the raw-buffer API to initialize `str` objects, and very often in a way which can't be easily replaced by a fully opaque API.

Example 1, markupsafe: the `DO_ESCAPE` macro takes a parameter called `outp` which is obtained by calling `PyUnicode*BYTE_DATA(1BYTE, (2BYTE, (4BYTE))`. `DO_ESCAPE` contains code like this, which would be hard to port to a fully-opaque API:

```

memcpy(outp, inp-ncopy, sizeof(*outp)*ncopy); \
outp += ncopy; ncopy = 0; \
*outp++ = '&'; \
*outp++ = '#'; \
*outp++ = '3'; \
*outp++ = '4'; \
*outp++ = ';'; \
break; \

```

Another interesting example is `pybase64`. After removing the unnecessary stuff, the logic boils down to this:

```

out_len = (size_t)((buffer.len + 2) / 3) * 4;
out_object = PyUnicode_New((Py_ssize_t)out_len, 127);
dst = (char*)PyUnicode_1BYTE_DATA(out_object);
...
base64_encode(buffer.buf, buffer.len, dst, &out_len, libbase64_simd_flag);

```

Note that `base64_encode` is an external C function which writes stuff into a `char *` buffer, so in this case it is **required** to use the raw-buffer API, unless you want to allocate a temporary buffer and copy chars one-by-one later.

There are other examples similar to these, but I think there is already enough evidence that HPy **must** offer a raw-buffer API in addition to a fully-opaque one.

Typed vs untyped raw-buffer writing

To initialize a `str` object using the raw-buffer interface, you need to get a pointer to the buffer. The vast majority of code uses `PyUnicode_{1,2,4}BYTE_DATA` to get a buffer of type `Py_UCS{1,2,4}*` and write directly to it:

```
PyObject *s = PyUnicode_New(size, 127);
Py_UCS1 *buf = PyUnicode_1BYTE_DATA(s);
buf[0] = 'H';
buf[1] = 'e';
buf[2] = 'l';
...
```

The other way to get a pointer to the raw-buffer is to call `PyUnicode_DATA()`, which returns a `void *`: the only reasonable way to write something in this buffer is to `memcpy()` the data from another `str` buffer of the same kind. This technique is used for example by [CPython's textio.c](#).

Outside CPython, the only usage of this technique is inside cython's helper function `__Pyx_PyUnicode_Join`.

This probably means that we don't need to offer untyped raw-buffer writing for HPy. If we really need to support the `memcpy` use case, we can probably just offer a special function in the builder API.

PyUnicode_WRITE, PyUnicode_WriteChar

Outside CPython, `PyUnicode_WRITE()` is used only inside Cython's helper functions (1, 2). Considering that Cython will need special support for HPy anyway, this means that we don't need an equivalent of `PyUnicode_WRITE` for HPy.

Similarly, `PyUnicode_WriteChar()` is used only once, inside [JPy](#).

PyUnicode_Join

All the API functions listed above require the user to know in advance the size of the string: `PyUnicode_Join()` is the only native API call which allows to build a string whose size is not known in advance.

Examples of usage are found in [simplejson](#) (1, 2), [pycairo](#), [regex](#) (1, 2, 3, 4, 5, 6) and others, for a total of 25 grep matches.

Note: Contrarily to its unicode equivalent, `PyBytes_Join()` does not exist. There is `__PyBytes_Join()` which is private and undocumented, but some extensions rely on it anyway: [Cython](#), [regex](#), [dulwich](#).

In theory, alternative implementations should be able to provide a more efficient way to achieve the goal. E.g. for pure Python code PyPy offers `__pypy__.builders.StringBuilder` which is faster than both `StringIO` and `''.join`, so maybe it might make sense to offer a way to use it from C.

CHANGELOG

7.1 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

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

C

CPython ABI, [4](#)

H

HPy Hybrid ABI, [4](#)

HPy Universal ABI, [4](#)

HPy_LeavePythonExecution (*C function*), [24](#)

HPy_ReenterPythonExecution (*C function*), [24](#)

HPyArg_Parse (*C function*), [27](#)

HPyArg_ParseKeywords (*C function*), [27](#)

HPyField_Load (*C function*), [23](#)

HPyField_Store (*C function*), [23](#)

HPyGlobal_Load (*C function*), [24](#)

HPyGlobal_Store (*C function*), [24](#)

HPyHelpers_AddType (*C function*), [30](#)

P

Python Enhancement Proposals

PEP 3149, [4](#)