HPy

Release 0.0.1

HPy Collective

Jun 10, 2021

CONTENTS

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, 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

CHAPTER

ONE

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 supports 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, but it will be something like `hpy-1`, where `1` is the version of the API.

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`.

Moreover, each alternative Python implementation could decide to implement its own non-universal ABI if it makes sense for them. For example, a hypotetical 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.

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 debugging 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. The plan is to write a 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 2021, the following milestones have been reached:

- 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*).
- one can implement custom types.
- 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 straighforward, so for now the priority is to work on the "hard" features to prove that the HPy approach works, and we will port new functions as needed
- handles are intended to be short-lived, but sometimes one needs a long-lived reference to a Python object. In HPy, we call this long-lived reference an *HPyField*, but we still need to implement it. We also need *HPy_Store* and *HPy_Load* to save and load these fields. This will allow alternative implementations to use a moving GC.
- add C-level module state. Often an extension needs module state that is accessible from C (e.g. if a module implements a new type *ArrayType*, many of the module methods written in C may need access to *ArrayType*) but HPy does not have convenient support for this yet.
- improve the debug mode so that it returns a C traceback showing where a handle was leaked or a closed handle was used.

- there is no integration with Cython. The medium-term plan is to extend Cython to automatically generate HPy-compatible C code.

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 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.

1.10 Projects involved

HPy was born during EuroPython 2019, were 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 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
- 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

CHAPTER

TWO

PORTING GUIDE

2.1 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).

2.2 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.

2.3 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`.

2.4 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

2.5 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.

2.6 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.

CHAPTER

THREE

HPY API

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

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/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 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()`.

3.1.1 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` an `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 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:

```
void 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()`:

```
void 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.

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/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-interpreter. 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.

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/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"

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

There are a couple of points which are worth noting:

- We use the macro `HPy_DEF_METH_O` to declare we are going to define a HPy function called `myabs`, which uses the `METH_O` calling convention. As in Python/C, `METH_O` means that the function receives a single argument.
- The actual C function which implements `myabs` is called `myabs_impl`.
- It receives two arguments of type `HPy`, which are handles which are guaranteed to be valid: they are automatically closed by the caller, so there is no need to call `HPy_Close` on them.
- It 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.

The `HPy_DEF_METH_O` 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`.

Now, we can define our module:

```
static HPyMethodDef SimpleMethods[] = {
    {"myabs", myabs, HPy_METH_O, "Compute the absolute value of the given argument"},
    {NULL, NULL, 0, NULL}
};

static HPyModuleDef moduledef = {
```

(continues on next page)

(continued from previous page)

```
    HPyModuleDef_HEAD_INIT,
    .m_name = "simple",
    .m_doc = "HPy Example",
    .m_size = -1,
    .m_methods = SimpleMethods
};
```

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, and that we have to indicate the calling convention again. This is a deliberate choice, to minimize the changes needed to port existing extensions, and to make it easier to support 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`.

3.3.1 Building the module

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

```
from setuptools import setup, Extension

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

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
```

3.3.2 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:

```
HPy_DEF_METH_VARARGS(add_ints)
static HPy add_ints_impl(HPyContext *ctx, HPy self, HPy *args, HPy_ssize_t nargs)
{
    long a, b;
    if (!HPyArg_Parse(ctx, 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 HPyMethodDef SimpleMethods[] = {
    {"myabs", myabs, HPy_METH_O, "Compute the absolute value of the given argument"},
    {"add_ints", add_ints, HPy_METH_VARARGS, "Add two integers"},
    {NULL, NULL, 0, NULL}
};
```

CHAPTER

FOUR

API REFERENCE

4.1 Argument Parsing

Implementation of HPyArg_Parse and HPyArg_ParseKeywords.

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.

4.1.1 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.

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.

4.1.2 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 strint, *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 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 strint, *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.

CHAPTER

FIVE

MISC NOTES

5.1 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

5.1.1 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 represenation (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.

5.1.2 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.

5.1.3 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-specificed "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.

5.1.4 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 cffi, `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 lest 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 contructs 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_Decode*

Functions of the `PyUnicode_From*` and `PyUnicode_Decode*` 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:

Another interesting example is **`pybase64 <https://github.com/hpyproject/top4000-pypi-packages/blob/0cd919943a007f95f4bf8510e667cfff5bd059fc/top4000/1925-pybase64-1.1.4/pybase64/_pybase64.c#L320-349`_**. After removing the unnecessary stuff, the logic boils down to this:

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 JPype.

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 implementaions 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.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX