Python and C++ Containers

Release 0.3.1

Paul Ross

Feb 15, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Python is well known for it's ability to handle *heterogeneous* data in containers such as lists. But what if you need to interact with C++ containers such as `std::vector<T>` that require *homogeneous* data types?

This project is about converting Python containers (`list`, `tuple`, `dict`, `set`, `frozenset`) containing homogeneous types (`bool`, `int`, `float`, `complex`, `bytes`, `str`) or user defined types to and from their C++ equivalent.

1.1 A Problematic Example

Suppose that you have a Python list of floats and need to pass it to a C++ library that expects a `std::vector<double>`. If the result of that call modifies the C++ vector, or creates a new one, you need to return a Python list of floats from the result.

Your code might look like this:

```
PyObject *example(PyObject *op) {
    std::vector<double> vec;
    // Populate the vector, function to be defined...
    write_to_vector(op, vec);
    // Do something in C++ with the vector
    // ...
    // Convert the vector back to a Python list.
    // Function to be defined...
    return read_from_vector(vec);
}
```

What should the implementation of `write_to_vector()` and `read_from_vector()` look like?

The answer seems fairly simple; firstly `write_to_vector` converting a Python list to a C++ `std::vector<double>`:

```
void write_to_vector(PyObject *op, std::vector<double> &vec) {
    vec.clear();
    for (Py_ssize_t i = 0; i < PyList_Size(op); ++i) {
        vec.push_back(PyFloat_AsDouble(PyList_GET_ITEM(op, i)));
    }
}
```

And the inverse, `read_from_vector` creating a new Python list from a C++ `std::vector<double>`:

```
PyObject *read_from_vector(const std::vector<double> &vec) {
    PyObject *ret = PyList_New(vec.size());
    for (size_t i = 0; i < vec.size(); ++i) {
```

(continues on next page)

(continued from previous page)

```
        PyList_SET_ITEM(ret, i, PyFloat_FromDouble(vec[i]));
    }
    return ret;
}
```

There is no error handling here, all errors would be runtime errors.

However if you need to support other object types, say lists of `int`, `str`, `bytes` then each one needs a pair of hand written functions. It gets worse when you want to support other containers such as (`tuple`, `list`, `set`, `frozenset`, `dict`). Then you have to write individual conversion functions for all the combinations of object types *and* containers. This is tedious and error prone.

1.2 Why This Project

This project makes extensive use of C++ templates, partial template specialisation and code generation to reduce dramatically the amount of hand maintained code. It also converts many runtime errors to compile time errors.

This project supports two way conversion of this set of containers:

Table 1: Supported Containers.

C++ Container	Python Equivalent
`std::vector`	Either a `tuple` or `list`
`std::list`	Either a `tuple` or `list`
`std::unordered_set`	Either a `set` or `frozenset`
`std::unordered_map`	`dict`
`std::map`	`dict`

Which contain any of this set of types:

Table 2: Supported Object types.

C++ Type	Python Equivalent
`bool`	`True`, `False`
`long`	`int`
`double`	`float`
`std::complex<double>`	`complex`
`std::vector<char>`	`bytes`
`std::string`	`str`

The number of possible conversion functions is worse than the cartesian product of the types and containers as in the case of a dict the types can appear as either a key or a value.

Supporting all these conversions would normally require 216 conversion functions to be written, tested and documented[1] .

This project simplifies this by using a mix of C++ templates and code generators to reduce this number to just **six** hand written templates for all 216 cases.

[1] There are six unary container pairings (`tuple` <-> `std::list`, `tuple` <-> `std::vector`, `list` <-> `std::list`, `list` <-> `std::vector`, `set` <-> `std::unordered_set`, `frozenset` <-> `std::unordered_set`) with six types (`bool`, `int`, `float`, `complex`, `bytes`, `str`). Each container/type combination requires two functions to give two way conversion from Python to C++ and back. Thus 6 (container pairings) * 6 (types) * 2 (way conversion) = 72 required functions. For `dict` there are two container pairings (`dict` <-> `std::map`, `dict` <-> `std::unordered_map`) with the six types either of which can be the key or the value so 36 possible variations. Thus 2 (container pairings) * 36 (type pairs) * 2 (way conversion) = 144 required functions. Thus is a total of 72 + 144 = 216 functions.

- Two C++ templates for Python `tuple` / `list` to and from `std::list` or `std::vector` for all types.
- Two C++ templates for Python `set` / `frozenset` to and from `std::unordered_set` for all types.
- Two C++ templates for Python `dict` to and from `std::map` or `std::unordered_map` for all type combinations.

These six handwritten templates are fairly simple and comprehensible. Then, for simplicity, a Python script is used to create the final, instantiated, 216 functions.

1.3 Hand Written Functions

There are only six non-trivial hand written functions along with a much larger of generated functions that successively specialise these functions.

As an example, here how the function is developed that converts a Python list of `float` to and from a C++ `std::vector<double>` or `std::list<double>`.

First C++ to Python.

1.3.1 Converting a C++ `std::vector<T>` or `std::list<T>` to a Python `tuple` or `list`

The generic function signature looks like this:

```
template<
    template<typename ...> class ListLike,
    typename T,
    PyObject *(*ConvertCppToPy)(const T &),
    PyObject *(*PyUnaryContainer_New)(size_t),
    int(*PyUnaryContainer_Set)(PyObject *, size_t, PyObject *)
>
PyObject *
very_generic_cpp_std_list_like_to_py_unary(const ListLike<T> &list_like) {
    // Handwritten
    // ...
}
```

Table 3: `very_generic_cpp_std_list_like_to_py_unary()` template parameters.

Template Parameter	Notes
`ListLike`	The C++ container type, either a `std::vector<T>` or `std::list<T>`.
`T`	The C++ type of the objects in the target C++ container.
`ConvertCppToPy`	A pointer to a function that converts any C++ `T` to a `PyObject *`, for example from `double` -> `float`.
`PyUnaryContainer_New`	A pointer to a function that creates a new Python container, for example a `list`, of a particular length.
`PyUnaryContainer_Set`	A pointer to a function that sets a `PyObject *` in the Python container at a given index.

And the function has the following parameters.

Table 4: `very_generic_cpp_std_list_like_to_py_unary()` parameters.

Type	Name	Notes
`ListLike<T> &`	`list_like`	The C++ list like container to read from to.

The return value is non-NULL on success or NULL if there is a runtime error. These errors could be:

- `PyObject *` container can not be created.
- A member of the Python container can not be created from the C++ type `T`.
- The `PyObject *` can not be inserted into the Python container.

1.3.2 Partial Specialisation to Convert a C++ `std::vector<T>` or `std::list<T>` to a Python `list``

As an example this is specialised for Python `list` with a handwritten oneliner:

```
template<
    typename T,
    PyObject *(*ConvertCppToPy)(const T &)
>
PyObject *
generic_cpp_std_list_like_to_py_list(const std::vector<T> &container) {
    return very_generic_cpp_std_list_like_to_py_unary<
        std::vector,
        T,
        ConvertCppToPy,
        &py_list_new,
        &py_list_set
    >(container);
}
```

Note: The use of the function pointers to `py_list_new`, and `py_list_set` that are defined in this project namespace. These are thin wrappers around existing functions or macros in `"Python.h"`.

There is a similar partial specialisation for `tuple`.

1.3.3 Converting a Python `tuple` or `list` to a C++ `std::vector<T>` or `std::list<T>`

The reverse is converting Python to C++. This generic function that converts unary Python indexed containers (`tuple` and `list`) to a C++ `std::vector<T>` or `std::list<T>` for any type has this signature:

```
template<
        template<typename ...> class ListLike,
        typename T,
        int (*PyObject_Check)(PyObject *),
        T (*PyObject_Convert)(PyObject *),
        int(*PyUnaryContainer_Check)(PyObject *),
        Py_ssize_t(*PyUnaryContainer_Size)(PyObject *),
```

(continues on next page)

(continued from previous page)

```
        PyObject *(*PyUnaryContainer_Get)(PyObject *, size_t)>
int very_generic_py_unary_to_cpp_std_list_like(
    PyObject *op, ListLike<T> &list_like
) {
    // Hand written code.
    // ...
}
```

This template has these parameters:

Table 5: `very_generic_py_unary_to_cpp_std_list_like()` template parameters.

Template Parameter	Notes
`ListLike`	The C++ container type, either a `std::vector<T>` or `std::list<T>`.
`T`	The C++ type of the objects in the target C++ container.
`PyObject_Check`	A pointer to a function that checks that any `PyObject *` in the Python container is the correct type, for example that it is a `bytes` object.
`PyObject_Convert`	A pointer to a function that converts any `PyObject *` in the Python container to the C++ type, for example from `bytes` -> `std::vector<char>`.
`PyUnaryContainer_Check`	A pointer to a function that checks that the `PyObject *` argument is the correct container type, for example a `tuple`.
`PyUnaryContainer_Size`	A pointer to a function that returns the size of the Python container.
`PyUnaryContainer_Get`	A pointer to a function that gets a `PyObject *` from the Python container at a given index.

And the function has the following parameters.

Table 6: `generic_py_unary_to_cpp_std_list_like()` parameters.

Type	Name	Notes
`PyObject *`	`op`	The Python container to read from.
`ListLike<T> &`	`list_like`	The C++ list like container to write to.

The return value is zero on success or non-zero if there is a runtime error. These errors could be:

- `PyObject *op` is not a container of the required type.
- A member of the Python container can not be converted to the C++ type `T` (`PyObject_Check` fails).

1.3.4 Partial Specialisation to Convert a Python `list` to a C++ `std::vector<T>` or `std::list<T>`

This template can be partially specialised for converting Python *lists* of any type to C++ `std::vector<T>` or `std::list<T>`. This is hand written code but it is trivial by wrapping a single function call.

In the particular case of a `std::vector` we can use `.reserve()` as an optimisations to avoid excessive re-allocations.

```
template<
    typename T,
    int (*PyObject_Check)(PyObject *),
    T (*PyObject_Convert)(PyObject *)
>
```

(continues on next page)

(continued from previous page)

```
int generic_py_list_to_cpp_std_list_like(
    PyObject *op, std::vector<T> &container
) {
    // Reserve the vector, but only if it is a list.
    // If it is any other Python object then ignore it as py_list_len()
    // may give undefined behaviour.
    // Leave it to very_generic_py_unary_to_cpp_std_list_like() to error
    if (py_list_check(op)) {
        container.reserve(py_list_len(op));
    }
    return very_generic_py_unary_to_cpp_std_list_like<
        std::vector,
        T,
        PyObject_Check,
        PyObject_Convert,
        &py_list_check,
        &py_list_len,
        &py_list_get
    >(op, container);
}
```

Note: The use of the function pointers to `py_list_check`, `py_list_len` and `py_list_get` that are defined in this project namespace. These are thin wrappers around existing functions or macros in `"Python.h"`.

There is a similar partial specialisation for `tuple`.

1.4 Generated Functions

The particular function specialisations are created by a script that takes the cartesian product of object types and container types and creates functions for each container/object.

1.4.1 C++ to Python

For example, to convert a C++ `std::vector<double>` to a Python `list` of `float` the following are created:

A base declaration in *auto_py_convert_internal.h*:

```
template<typename T>
PyObject *
cpp_std_list_like_to_py_list(const std::vector<T> &container);
```

And a concrete declaration for each C++ target type `T` in *auto_py_convert_internal.h*:

```
template <>
PyObject *
cpp_std_list_like_to_py_list<double>(const std::vector<double> &container);
```

And the concrete definition is in *auto_py_convert_internal.cpp*, this simply calls the generic function:

```
template <>
PyObject *
cpp_std_list_like_to_py_list<double>(const std::vector<double> &container) {
    return generic_cpp_std_list_like_to_py_list<
        double, &cpp_double_to_py_float
    >(container);
}
```

Here is the function hierarchy for converting lists to C++ `std::vector<T>` or `std::list<T>`: This is the function hierarchy for the code that converts C++ `std::vector<T>` or `std::list<T>` to Python `list` and `tuple` for all supported object types.

```
                   very_generic_cpp_std_list_like_to_py_unary <-- Hand written
                                         |
                        /--------------------------\
                        |                          |            Hand written partial
        generic_cpp_std_list_like_to_py_list    tuples...    <-- specialisation for
                        |                          |            std::vector
                        |                          |            and std::list
                        |                          |            (generally trivial).
                        |                          |
          cpp_std_list_like_to_py_list<T>         ...        <-- Generated
                        |                          |
        /-------------------------------\      /-------\
        |                               |      |       |        Generated declaration
cpp_std_list_like_to_py_list<double>   ...    ...     ...    <-- and implementation
                                                                (one liners)
```

1.4.2 Python to C++

For example, to convert a Python `list` of `float` to a C++ `std::vector<double>` the following are generated:

A base declaration in *auto_py_convert_internal.h*:

```
template<typename T>
int
py_list_to_cpp_std_list_like(PyObject *op, std::list<T> &container);
```

And a concrete declaration for each C++ target type T in *auto_py_convert_internal.h*:

```
template <>
int
py_list_to_cpp_std_list_like<double>(PyObject *op, std::list<double> &container);
```

And the concrete definition is in *auto_py_convert_internal.cpp*:

```
template <>
int
py_list_to_cpp_std_list_like<double>(PyObject *op, std::vector<double> &container) {
    return generic_py_list_to_cpp_std_list_like<
        double, &py_float_check, &py_float_to_cpp_double
    >(op, container);
}
```

This is the function hierarchy for the code that converts Python `list` and `tuple` to C++ `std::vector<T>` or `std::list<T>` for all supported object types.

```
                  very_generic_py_unary_to_cpp_std_list_like <-- Hand written
                                     |
                        /--------------------------\
                        |                          |          Hand written partial
          generic_py_list_to_cpp_std_list_like    tuples...    <-- specialisation for
                        |                          |          std::vector
                        |                          |          and std::list
                        |                          |          (generally trivial).
                        |                          |
            py_list_to_cpp_std_list_like<T>       ...         <-- Generated
                        |                          |
          /--------------------------------\     /-------\
          |                                |     |       |          Generated declaration
py_list_to_cpp_std_list_like<double>      ...   ...     ...   <-- and implementation
                                                              (one liners)
```

1.5 Usage

Using the concrete function is as simple as this:

```
using namespace Python_Cpp_Containers;
// Create a PyObject* representing a list of Python floats.
PyObject *op = PyList_New(3);
PyList_SetItem(op, 0, PyFloat_FromDouble(21.0));
PyList_SetItem(op, 1, PyFloat_FromDouble(42.0));
PyList_SetItem(op, 2, PyFloat_FromDouble(3.0));

// Create the output vector...
std::vector<double> cpp_vector;

// Template specialisation will automatically invoke the appropriate
// function call.
// It will be a compile time error if the container/type function
// is not available.
// At run time this will return zero on success, non-zero on failure,
// for example if op is not a Python tuple or members of op can not be
// converted to C++ doubles.
int err = py_list_to_cpp_std_list_like(op, cpp_vector);
// Handle error checking...

// Now convert back.
// Again this will be a compile time error if the C++ type is not supported.
PyObject *new_op  = cpp_std_list_like_to_py_list(cpp_vector);
// new_op is a Python list of floats.
// new_op will be null on failure and a Python exception will have been set.
```

CHAPTER

TWO

USING THIS C++ LIBRARY

2.1 The Basics

2.1.1 Code Generation

If necessary run the code generator:

```
cd src/py
python code_gen.py
```

Which should give you something like:

```
venv/bin/python src/py/code_gen.py
Target directory "src/cpy"
Writing declarations to "src/cpy/auto_py_convert_internal.h"
Wrote 2654 lines of code with 220 declarations.
Writing definitions to  "src/cpy/auto_py_convert_internal.cpp"
Wrote 2384 lines of code with 216 definitions.

Process finished with exit code 0
```

2.1.2 Build Configuration

You need to compile the following C++ files by adding them to your makefile or CMakeLists.txt:

```
src/cpy/auto_py_convert_internal.cpp
src/cpy/python_container_convert.cpp
src/cpy/python_object_convert.cpp
```

2.1.3 Source Inclusion

Your pre-processor needs access to the header files with the compiler flag `-I src/cpy`.

Then in your C++ code include:

```
#include "python_convert.h"
```

Which gives you access to the whole API in the namespace `Python_Cpp_Containers`.

2.1.4 Errors

If using this library in C++ there will be a linker error if you specify a template type that is not supported. For example here is some code that tries to copy a Python list of unsigned integers. The two conversion functions are not defined for `unsigned int`.

```
static PyObject *
new_list_unsigned_int(PyObject *Py_UNUSED(module), PyObject *arg) {
    std::vector<unsigned int> vec;
    if (!py_list_to_cpp_std_list_like(arg, vec)) {
        return cpp_std_list_like_to_py_list(vec);
    }
    return NULL;
}
```

A C++ tool chain will complain with a linker error such as:

```
Undefined symbols for architecture x86_64:
  "_object* Python_Cpp_Containers::cpp_std_list_like_to_py_list<unsigned int>(std::__
↪1::vector<unsigned int, std::__1::allocator<unsigned int> > const&)", referenced from:
      new_list_unsigned_int(_object*, _object*) in cPyCppContainers.cpp.o
  "int Python_Cpp_Containers::py_list_to_cpp_std_list_like<unsigned int>(_object*, std::_
↪_1::vector<unsigned int, std::__1::allocator<unsigned int> >&)", referenced from:
      new_list_unsigned_int(_object*, _object*) in cPyCppContainers.cpp.o
ld: symbol(s) not found for architecture x86_64
```

If you are building a Python extension this will, most likely, build but importing the extension will fail immediately with something like:

```
>>> import cPyCppContainers
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: dlopen(cPyCppContainers.cpython-39-darwin.so, 2): Symbol not found: __
↪ZN21Python_Cpp_Containers25cpp_std_list_like_to_py_listIjEEP7_objectRKNSt3__16vectorIT_
↪NS3_9allocatorIS5_EEEE
  Referenced from: cPyCppContainers.cpython-39-darwin.so
  Expected in: flat namespace
 in cPyCppContainers.cpython-39-darwin.so
```

2.2 Examples

There are some examples of using this library in *src/ext/cPyCppContainers.cpp*. This extension is built by *setup.py* and tested with *tests/unit/test_cPyCppContainers.py*.

To build this extension:

```
$ python setup.py develop
```

And to use it:

```
import cPyCppContainer
```

2.2.1 Using C++ to Double the Values in a Python List of `float`

Here is one of those examples in detail; doubling the values of a Python list of floats.

At the beginning of the extension C/C++ code we have:

```
#include "cpy/python_convert.h"
```

For convenience we use the namespace that the conversion code is within:

```
using namespace Python_Cpp_Containers;
```

Here is the C++ function that we want to call that multiplies the values of a `std::vector<double>` in-place by 2.0:

```
/** Double the values of a vector in-place. */
static void
vector_double_x2(std::vector<double> &vec) {
    for (size_t i = 0; i < vec.size(); ++i) {
        vec[i] *= 2.0;
    }
}
```

And here is the code that takes a Python list of floats, then calls the C++ function and finally converts the C++ `std::vector<double>` back to a new Python list of floats:

```
/** Create a new list of floats with doubled values. */
static PyObject *
list_x2(PyObject *Py_UNUSED(module), PyObject *arg) {
    std::vector<double> vec;
    // py_list_to_cpp_std_list_like() will return non-zero if the Python
    // argument can not be converted to a std::vector<double>
    // and a Python exception will be set.
    if (!py_list_to_cpp_std_list_like(arg, vec)) {
        // Double the values in pure C++ code.
        vector_double_x2(vec);
        // cpp_std_list_like_to_py_list() returns NULL on failure
        // and a Python exception will be set.
        return cpp_std_list_like_to_py_list(vec);
    }
    return NULL;
}
```

The vital piece of code is the declaration `std::vector<double> vec;` and that means:

- If a `py_list_to_cpp_std_list_like()` implementation does not exist for `double` there will be a compile time error.
- Giving `py_list_to_cpp_std_list_like()` anything other than a list of floats will create a Python runtime error.
- If `cpp_std_list_like_to_py_list()` fails for any reason there will be a Python runtime error.

Using the Extension

Once the extension is built you can use it thus:

```
>>> import cPyCppContainers
>>> cPyCppContainers.list_x2([1.0, 2.0, 4.0])
[2.0, 4.0, 8.0]
```

You can verify that the returned list is a new one rather than modifying the input in-place: .. code-block:: python

```
>>> a = [1.0, 2.0, 4.0]
>>> b = cPyCppContainers.list_x2(a)
>>> hex(id(a))
'0x1017150c0'
>>> hex(id(b))
'0x101810dc0'
```

If the values are not floats or the container is not a list a `ValueError` is raised:

```
>>> cPyCppContainers.list_x2([1, 2, 4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Python value of type int can not be converted
>>> cPyCppContainers.list_x2((1.0, 2.0, 4.0))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Can not convert Python container of type tuple
```

2.2.2 Reversing a `tuple` of `bytes` in C++

Here is another example, suppose that we have a function to to reverse a `tuple` of `bytes` in C++:

```
/** Returns a new vector reversed. */
template<typename T>
static std::vector<T>
reverse_vector(const std::vector<T> &input){
    std::vector<T> output;
    for (size_t i = input.size(); i-- > 0;) {
        output.push_back(input[i]);
    }
    return output;
}
```

Here is the extension code that call this:

```
/** Reverse a tuple of bytes in C++. */
static PyObject *
tuple_reverse(PyObject *Py_UNUSED(module), PyObject *arg) {
    std::vector<std::string> vec;
    if (!py_tuple_to_cpp_std_vector(arg, vec)) {
        return cpp_std_vector_to_py_tuple(reverse_vector(vec));
    }
```

(continues on next page)

(continued from previous page)

```
    return NULL;
}
```

Once again the declaration `std::vector<std::string> vec;` ensures that the correct instantiations of conversion functions are called.

When the extension is built it can be used like this:

```
>>> import cPyCppContainers
>>> cPyCppContainers.tuple_reverse((b'ABC', b'XYZ'))
(b'XYZ', b'ABC')
```

2.2.3 Incrementing `dict` values in C++

Here is an example of taking a Python `dict` of `[bytes, int]` and creating a new `dict` with the values increased by one. The C++ code in the extension is this:

```
/** Creates a new dict[bytes, int] with the values incremented by 1 in C++ */
static PyObject *
dict_inc(PyObject *Py_UNUSED(module), PyObject *arg) {
    std::unordered_map<std::string, long> dict;
    /* Copy the Python structure to the C++ one. */
    if (!py_dict_to_cpp_std_unordered_map(arg, dict)) {
        /* Increment. */
        for(auto &key_value: dict) {
            key_value.second += 1;
        }
        /* Copy the C++ structure to a new Python dict. */
        return cpp_std_unordered_map_to_py_dict(dict);
    }
    return NULL;
}
```

Once the extension is built this can be used thus:

```
>>> import cPyCppContainers
>>> cPyCppContainers.dict_inc({b'A' : 65, b'Z' : 90})
{b'Z': 91, b'A': 66}
```

There are several other examples in *src/ext/cPyCppContainers.cpp* with tests in *tests/unit/test_cPyCppContainers.py*.

2.3 Testing

2.3.1 Testing With C++

Debug Build

Building the C++ code and running with `main()` will execute all functional tests when built as a debug build. This takes a couple of minutes or so.

```
test_functional_all START
...
Number of tests: 430
REGEX_HEAD: "HEAD:\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\
↪S+)\s+(\S+)"
REGEX_TEST: "TEST:\s+(\d+)\s+(\d+)\s+(\d+)\s+([0-9+-.]+)\s+([0-9+-.]+)\s+([0-9+-.]+)\
↪s+([0-9+-.]+)\s+(\d+)\s+([0-9+-.]+)\s+(\S+)"
REGEX_TAIL: "TAIL:\s+(.+)"
HEAD: Fail   Scale  Repeat         Mean(s)     Std.Dev.(s)         Min.(s)         Max.
↪(s)     Count      Rate(/s) Name
TEST:    0    1024       1     0.000026474             N/A             N/A             N/
↪A         1       37772.2 test_vector_to_py_tuple<<bool>>():[1024]
...
TEST:    0   65536       1     0.084335436             N/A             N/A             N/
↪A         1          11.9 test_vector_vector_char_to_py_tuple<std::string[2048]>
↪():[65536]
TAIL: Passed=430/430 Failed=0/430

====RSS(Mb): was:      5.633 now:    116.824 diff:   +111.191 Peak was:      5.633 now: ␣
↪  340.168 diff:   +334.535 main.cpp
Total execution time: 142 (s)
Bye, bye!
```

Release Build

With a release build this will run the performance tests as well. This can require 10Gb of memory and can take 20 minutes or so.

```
test_functional_all START
...
test_memory_all FINISH
Number of tests: 2226
REGEX_HEAD: "HEAD:\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\
↪S+)\s+(\S+)"
REGEX_TEST: "TEST:\s+(\d+)\s+(\d+)\s+(\d+)\s+([0-9+-.]+)\s+([0-9+-.]+)\s+([0-9+-.]+)\
↪s+([0-9+-.]+)\s+(\d+)\s+([0-9+-.]+)\s+(\S+)"
REGEX_TAIL: "TAIL:\s+(.+)"
HEAD: Fail   Scale  Repeat         Mean(s)     Std.Dev.(s)         Min.(s)         Max.
↪(s)     Count      Rate(/s) Name
TEST:    0    1024       1     0.000003310             N/A             N/A             N/
↪A         1      302069.2 test_vector_to_py_tuple<<bool>>():[1024]
...
TEST:    0   65536       1     0.029584157             N/A             N/A             N/
↪A         1          33.8 test_vector_vector_char_to_py_tuple<std::string[2048]>
↪():[65536]
TAIL: Passed=9890/9890 Failed=0/9890

====RSS(Mb): was:      5.430 now:   2047.426 diff:  +2041.996 Peak was:      5.430 now: ␣
↪ 7725.137 diff:  +7719.707 main.cpp
Total execution time: 1e+03 (s)
Bye, bye!
```

2.3.2 Testing With Python

Unit Tests

Running the basic unit tests on the `cPyCppContainers` extension that exercises all the code:

```
$ pytest tests/
```

This takes two or three seconds.

Extra Tests

There are a couple of options that can be added:

- `--runslow` will run slow tests including performance test. Use the `-s` option to obtain the performance output.
- `--pymemtrace` will run memory tracing tests. This requires pymemtrace to be installed.

For the full set of tests use:

```
$ pytest tests/ -vs --runslow --pymemtrace
```

This can take around 30 minutes to complete.

2.4 Documentation

To create the documentation with `Sphinx` or `doxygen`.

2.4.1 Sphinx

To build the HTML and PDF documentation from the project directory:

```
$ cd docs/sphinx
$ make html latexpdf
$ open build/html/index.html
$ open build/latex/PythonCppContainers.pdf
```

2.4.2 Doxygen

To build the HTML Doxygen documentation from the project directory:

```
$ cd docs
$ doxygen PythonCppContainers.dox
$ open doxygen/html/index.html
```

CHAPTER

THREE

C++ API

3.1 Include File and Namespace

```
#include "python_convert.h"
```

All these APIs are in the namespace `Python_Cpp_Containers`.

3.2 Python Containers to C++

3.2.1 Error Indication

All of the conversion functions from Python to C++ return an integer which is zero on success, non-zero otherwise. Reasons for failure can be:

- The `PyObject *` is not the expected Python container, for example passing a Python tuple when a list is expected.
- A member of the Python container can not be converted to C++ type <T>.

In the error case a `PyErr_...` will be set.

3.2.2 Python `tuple` to `std::vector` or `std::list`

API

```
template<typename T>
int
py_tuple_to_cpp_std_list_like(PyObject *op, std::vector<T> &container);

template<typename T>
int
py_tuple_to_cpp_std_list_like(PyObject *op, std::list<T> &container);
```

Arguments

Argument op	Argument container	Return value
A Python tuple containing values convertable to type <T>.	The std::vector or std::list to write to.	0 on success, non-zero on failure in which case the container will be empty. The causes of failure can be; op is not a tuple or a member of the op can not be converted to type <T>.

Example

Process a tuple of Python float:

```
void tuple_float_to_cpp(PyObject *arg) {
    std::vector<double> vec;
    if (! py_tuple_to_cpp_std_list_like(arg, vec)) {
        // Handle error...
    }
    // Use vec...
}
```

3.2.3 Python list to std::vector or std::list

API

```
template<typename T>
int
py_list_to_cpp_std_list_like(PyObject *op, std::vector<T> &container);

template<typename T>
int
py_list_to_cpp_std_list_like(PyObject *op, std::list<T> &container);
```

Arguments

Argument op	Argument container	Return value
A Python list containing values convertable to type <T>.	The std::vector or std::list to write to.	0 on success, non-zero on failure in which case the container will be empty. The causes of failure can be; op is not a list or a member of the op can not be converted to type <T>.

Example

Process a list of Python `float`:

```
void list_float_to_cpp(PyObject *arg) {
    std::list<double> list;
    if (! py_list_to_cpp_std_list_like(arg, list)) {
        // Handle error...
    }
    // Use vec...
}
```

3.2.4 Python set to `std::unordered_set`

API

```
template<typename T>
int
py_set_to_cpp_std_unordered_set(PyObject *op, std::unordered_set<T> &container);
```

Arguments

Argument op	Argument `container`	Return value
A Python `set` containing values convertable to type <T>.	The `std::unordered_set` to write to.	0 on success, non-zero on failure.

Example

Process a set of Python `float`:

```
void set_float_to_cpp(PyObject *arg) {
    std::unordered_set<double> set;
    if (! py_set_to_cpp_std_unordered_set(arg, set)) {
        // Handle error...
    }
    // Use set...
}
```

3.2.5 Python `frozenset` to `std::unordered_set`

API

```
template<typename T>
int
py_frozenset_to_cpp_std_unordered_set(PyObject *op, std::unordered_set<T> &container);
```

Arguments

Argument op	Argument container	Return value
A Python `frozenset` containing values convertable to type <T>.	The `std::unordered_set` to write to.	0 on success, non-zero on failure.

Example

Process a frozenset of Python `float`:

```
void frozenset_float_to_cpp(PyObject *arg) {
    std::unordered_set<double> frozenset;
    if (! py_frozenset_to_cpp_std_unordered_set(arg, frozenset)) {
        // Handle error...
    }
    // Use frozenset...
}
```

3.2.6 Python `dict` to `std::unordered_map` or `std::map`

API

```
template<typename K, typename V>
int
py_dict_to_cpp_std_map_like(PyObject *op, std::unordered_map<K, V> &container);

template<typename K, typename V>
int
py_dict_to_cpp_std_map_like(PyObject *op, std::map<K, V> &container);
```

Arguments

Argument op	Argument container	Return value
A Python `dict` containing keys convertable to type <K> and values convertable to type <V>.	The `std::unordered_map` or `std::map` to write to.	0 on success, non-zero on failure.

Example

Process a dict of Python `[int, float]`:

```
void dict_int_float_to_cpp(PyObject *arg) {
    std::unordered_map<long, double> map;
    if (! py_dict_to_cpp_std_map_like(arg, map)) {
        // Handle error...
    }
    // Use map...
}
```

3.3 C++ Containers to Python

3.3.1 Error Indication

All of the conversion functions from C++ to Python return an `PyObject *`. If this is non-NULL it is a *new reference* and it is te responsibility of the caller to dispose off it.

On failure these functions will return NULL Reasons for failure can be:

- The new Python container can not be created with the CPython API, perhaps for memory reasons.
- A C++ object can not be converted to a Python object. I can not imagine how this would be the case.
- The converted C++ object can not be can not be inserted into the Python container. I can not imagine how this would be the case.

In the failure case a `PyErr_...` will be set.

3.3.2 C++ `std::vector` or `std::list` to Python `tuple`

API

To convert to a Python `tuple`:

```
template<typename T>
PyObject *
cpp_std_list_like_to_py_tuple(const std::vector<T> &container);

template<typename T>
PyObject *
cpp_std_list_like_to_py_tuple(const std::list<T> &container);
```

Arguments

Argument `container`	Return value
A `std::vector` or `std::list` of type <T> convertable to an appropriate Python type.	The new Python container, `NULL` on failure in which case a `PyErr` will be set.

Example

Create a tuple of Python `float`:

```
PyObject *vector_double_to_tuple() {
    std::vector<double> vec;
    // Populate vec
    // ...
    return cpp_std_list_like_to_py_tuple(vec);
}
```

3.3.3 C++ std::vector or std::list to Python list

API

To convert to a Python list:

```
template<typename T>
PyObject *
cpp_std_list_like_to_py_list(const std::vector<T> &container);

template<typename T>
PyObject *
cpp_std_list_like_to_py_list(const std::list<T> &container);
```

Arguments

Argument container	Return value
A std::vector or std::list of type <T> convertable to an appropriate Python type.	The new Python container, NULL on failure in which case a PyErr will be set.

Example

Create a list of Python float:

```
PyObject *vector_double_to_list() {
    std::vector<double> vec;
    // Populate vec
    // ...
    return cpp_std_list_like_to_py_list(vec);
}
```

3.3.4 C++ std::unordered_set to Python set

API

```
template<typename T>
PyObject *
cpp_std_unordered_set_to_py_set(const std::unordered_set<T> &container);
```

Arguments

Argument container	Return value
A std::unordered_set of type <T> convertable to an appropriate Python type.	The new Python container, NULL on failure in which case a PyErr will be set.

Example

Create a set of Python float:

```
PyObject *vector_double_to_list() {
    std::unordered_set<double> set;
    // Populate set
    // ...
    return cpp_std_unordered_set_to_py_set(set);
}
```

3.3.5 C++ std::unordered_set to Python frozenset

API

```
template<typename T>
PyObject *
cpp_std_unordered_set_to_py_frozenset(const std::unordered_set<T> &container);
```

Arguments

Argument container	Return value
A std::unordered_set of type <T> convertable to an appropriate Python type.	The new Python container, NULL on failure in which case a PyErr will be set.

Example

Create a frozenset of Python float:

```
PyObject *vector_double_to_list() {
    std::unordered_set<double> set;
    // Populate set
    // ...
    return cpp_std_unordered_set_to_py_frozenset(set);
}
```

3.3.6 C++ `std::unordered_map` or `std::map` to a Python `dict`

API

```
template<typename K, typename V>
PyObject *
cpp_std_map_like_to_py_dict(const std::unordered_map<K, V> &container);

template<typename K, typename V>
PyObject *
cpp_std_map_like_to_py_dict(const std::map<K, V> &container);
```

Arguments

Argument `container`	Return value
A `std::unordered_map` or `std::map` of type `<K, V>` convertable to appropriate Python types.	The new Python container, `NULL` on failure in which case a `PyErr` will be set.

Example

Create a dict of Python `[int, float]`:

```
PyObject *map_double_to_list() {
    std::unordered_map<long, double> map;
    // Populate map
    // ...
    return cpp_std_map_like_to_py_dict(map);
}
```

CHAPTER

FOUR

USER DEFINED TYPES

Contents

This shows how to support conversion of containers of user defined types between Python and C++.

There are several steps:

- Have the definitions of both the C++ and CPython equivalent objects for two way conversion. This only has to be done once regardless of how many containers are to be supported.
- Define the functions to check a Python object and the two way conversion functions between CPython and C++ objects. This only has to be done once regardless of how many containers are to be supported.
- For each container conversion declare the template specialisation and definition. These will be one-liner calls to this project's generic functions.

This is probably best done by example.

4.1 User Defined Types in a C Extension

This example will demonstrate supporting the conversion of `std::vector` s of a user defined C++ class with a `list` of Python equivalents.

4.1.1 The C++ Class Declared in the File `cUserDefined.h`

Here is an example of a user defined C++ class that contains a first name, second name and a number.:

```
#include <string>

class CppCustomObject {
public:
    CppCustomObject(
        const std::string &first,
        const std::string &last,
        long number) : m_first(first), m_last(last), m_number(number) {}
    // Accessors
    const std::string &first() const { return m_first; }
    const std::string &last() const { return m_last; }
    long number() const { return m_number; }
    std::string name() { return m_first + " " + m_last; }
    // Other methods here...
private:
    std::string m_first;
    std::string m_last;
    long m_number;
};
```

4.1.2 The Python Equivalent in the File `cUserDefined.cpp`

This is based on the example in the Python documentation That is varied slightly for this example:

- The module name is `cUserDefined` (rather than `custom` in the original example).
- The code for the C extension is in `cUserDefined.cpp`.

In this example a `CustomObject` class is created in `cUserDefined.cpp`:

```
typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;
```

This also has a method `name()` that combines the first and last names. Once the module is built it can be used in Python like this:

```
>>> import cUserDefined
>>> custom_object = cUserDefined.Custom('François', 'Truffaut', 21468)
>>> custom_object.name()
'François Truffaut'
```

4.1.3 Adding Conversion Code in `cUserDefined.cpp`

In the Python C extension add the verification and conversion code between the Python `CustomObject` and the C++ `CppCustomObject`.

First the include files:

```
#include "cUserDefined.h"
#include "cpy/python_object_convert.h"
```

Checking the Python Type

The the code to verify the Python type and its contents.

```
int py_custom_object_check(PyObject *op) {
    if (Py_TYPE(op) != &CustomType) {
        return 0;
    }
    CustomObject *p = (CustomObject *) op;
    if (!Python_Cpp_Containers::py_unicode_check(p->first)) {
        return 0;
    }
    if (!Python_Cpp_Containers::py_unicode_check(p->last)) {
        return 0;
    }
    return 1;
}
```

From C++ to Python

The code to convert from a C++ `CppCustomObject` to a Python `CustomObject`:

```
PyObject *
cpp_custom_object_to_py_custom_object(const CppCustomObject &obj) {
    CustomObject *op = (CustomObject *) Custom_new(&CustomType, NULL, NULL);
    if (op) {
        op->first = Python_Cpp_Containers::cpp_string_to_py_unicode(obj.first());
        op->last = Python_Cpp_Containers::cpp_string_to_py_unicode(obj.last());
        op->number = obj.number();
    }
    return (PyObject *) op;
}
```

From Python to C++

The code to convert from a Python `CustomObject` to a C++ `CppCustomObject`:

```
CppCustomObject py_custom_object_to_cpp_custom_object(PyObject *op) {
    // Check type, could throw here.
    assert(py_custom_object_check(op));
    CustomObject *p = (CustomObject *) op;
    return CppCustomObject(
            Python_Cpp_Containers::py_unicode_to_cpp_string(p->first),
            Python_Cpp_Containers::py_unicode_to_cpp_string(p->last),
            p->number
    );
}
```

4.1.4 Template Specialisation Declarations in `cUserDefined.h`

In the file, `cUserDefined.h`, include this project's header file and then in this project's namespace declare the specialisations to convert to and from a `std::vector` of these objects:

```
#include "cpy/python_convert.h"
```

From C++ to Python

```
namespace Python_Cpp_Containers {
    // Specialised declaration
    // C++ to Python
    template<>
    PyObject *
    cpp_std_list_like_to_py_list<CppCustomObject>(
        const std::vector<CppCustomObject> &container
    );
} // namespace Python_Cpp_Containers
```

From Python to C++

```
namespace Python_Cpp_Containers {
    // Specialised declaration
    // Python to C++
    template<>
    int
    py_list_to_cpp_std_list_like<CppCustomObject>(
        PyObject *op, std::vector<CppCustomObject> &container
    );
} // namespace Python_Cpp_Containers
```

4.1.5 Template Specialisation Definitions in cUserDefined.cpp

In the file cUserDefined.cpp implement the specialisations, these are just one-liners calling the generic conversion code.

From C++ to Python

```
namespace Python_Cpp_Containers {
    // Specialised implementations
    template<>
    PyObject *
    cpp_std_list_like_to_py_list<CppCustomObject>(
        const std::vector<CppCustomObject> &container
    ) {
        return generic_cpp_std_list_like_to_py_list<
                CppCustomObject, &cpp_custom_object_to_py_custom_object
        >(container);
    }
} // namespace Python_Cpp_Containers
```

From Python to C++

```
namespace Python_Cpp_Containers {
    template<>
    int
    py_list_to_cpp_std_list_like<CppCustomObject>(
        PyObject *op, std::vector<CppCustomObject> &container
    ) {
        return generic_py_list_to_cpp_std_list_like<
                CppCustomObject,
                &py_custom_object_check,
                &py_custom_object_to_cpp_custom_object
        >(op, container);
    }
} // namespace Python_Cpp_Containers
```

Now you have all the code needed to convert sequences of these objects between C++ and Python.

4.1.6 Using the C++ Conversion Functions

From C++ to Python

Here is an example of converting a C++ `std::vector<CppCustomObject>` to a Python `list` of `CustomObject`:

```
std::vector<CppCustomObject> vec_cpp_custom_object;
// Populate the C++ vector
// ...
// Convert to a new Python list of Python CustomObject. This will return NULL on failure.
return Python_Cpp_Containers::cpp_std_list_like_to_py_list(vec_cpp_custom_object);
```

From Python to C++

Here is an example of converting a Python `list` of `CustomObject` to a C++ `std::vector<CppCustomObject>`:

```
// op is a PyObject* which is a list of Python CustomObject
// Convert to C++
std::vector<CppCustomObject> vec_cpp_custom_object;
// Populate this C++ vector from the Python list
if (! Python_Cpp_Containers::py_list_to_cpp_std_list_like(op, vec_cpp_custom_object)) {
    // Converted successfully, use the vec_cpp_custom_object
    // ...
} else {
    // Handle error condition
    // ...
}
```

4.1.7 Example of Round-trip Conversion

Here is a complete example that takes a list of Python `CustomObject` and creates a list of C++ `CppCustomObject` with the first name and last name reversed in C++. Then it converts that C++ `std::vector` of `CppCustomObject` back to a new list of of Python `CustomObject`.

In `cUserDefined.cpp`:

```
static PyObject *
reverse_list_names(PyObject *Py_UNUSED(module), PyObject *arg) {
    std::vector<CppCustomObject> input;
    // Convert to a C++ vector
    if (! Python_Cpp_Containers::py_list_to_cpp_std_list_like(arg, input)) {
        // Create a new C++ vector with names reversed.
        std::vector<CppCustomObject> output;
        for (const auto &object: input) {
            // Note: reversing names.
            output.emplace_back(
                CppCustomObject(object.last(), object.first(), object.number())
            );
        }
        // Convert to a new Python list.
        return Python_Cpp_Containers::cpp_std_list_like_to_py_list(output);
    }
    return NULL;
}
```

Add this function to the module, in `cUserDefined.cpp`:

```
// Module functions
static PyMethodDef cUserDefinedMethods[] = {
        {"reverse_list_names", reverse_list_names, METH_O,
            "Take a list of cUserDefined.Custom objects"
            " and return a new list with the name reversed."},
        {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

Build the `cUserDefined` module and try it out:

```
>>> import cUserDefined
>>> list_of_names = [cUserDefined.Custom('First', 'Last', 21), cUserDefined.Custom(
↪'François', 'Truffaut', 21468)]
>>> list_of_names
[<cUserDefined.Custom object at 0x103d43450>, <cUserDefined.Custom object at 0x103f520f0>
↪]
>>> [v.name() for v in list_of_names]
['First Last', 'François Truffaut']
```

Now reverse the names in C++, the objects returned are new objects (compare with above):

```
>>> result = cUserDefined.reverse_list_names(list_of_names)
>>> result
[<cUserDefined.Custom object at 0x103d43720>, <cUserDefined.Custom object at 0x103f52e40>
↪]
```

And the names are reversed:

```
>>> [v.name() for v in result]
['Last First', 'Truffaut François']
```

4.1.8 Supporting `dict[int, cUserDefined.Custom]`

It takes very little additional work to support conversion between a C++ `std::map<long, CppCustomObject>` to a Python `dict[int, cUserDefined.Custom]`.

First add two specialised declarations in `cUserDefined.h`:

```
namespace Python_Cpp_Containers {
    // Specialised declarations
    // C++ to Python
    template<>
    PyObject *
    cpp_std_map_like_to_py_dict<std::map, long, CppCustomObject>(
        const std::map<long, CppCustomObject> &map
    );

    // Python to C++
    template <>
    int
    py_dict_to_cpp_std_map_like<std::map, long, CppCustomObject>(
        PyObject* op, std::map<long, CppCustomObject> &map
    );
} // namespace Python_Cpp_Containers
```

And their definitions in `cUserDefined.cpp`. Again these are just one-liners to this project's generic functions (expanded for clarity).

From C++ to Python

```
namespace Python_Cpp_Containers {
    // Specialised definitions
    // C++ to Python
    template<>
    PyObject *
    cpp_std_map_like_to_py_dict<std::map, long, CppCustomObject>(
        const std::map<long, CppCustomObject> &map
    ) {
        return generic_cpp_std_map_like_to_py_dict<
            std::map,
            long,
            CppCustomObject,
            &cpp_long_to_py_long,
            &cpp_custom_object_to_py_custom_object
        >(map);
    }
} // namespace Python_Cpp_Containers
```

From Python to C++

```
namespace Python_Cpp_Containers {
    // Python to C++
    template <>
    int
    py_dict_to_cpp_std_map_like<std::map, long, CppCustomObject>(
        PyObject* op, std::map<long, CppCustomObject> &map
    ) {
        return generic_py_dict_to_cpp_std_map_like<
            std::map,
            long,
            CppCustomObject,
            &py_long_check,
            &py_custom_object_check,
            &py_long_to_cpp_long,
            &py_custom_object_to_cpp_custom_object
        >(op, map);
    }
} // namespace Python_Cpp_Containers
```

Example Code

Here is an example of using both of them in a similar way to above by creating a new dict with the names reversed in C++.

In `cUserDefined.cpp`:

```
static PyObject *
reverse_dict_names(PyObject *Py_UNUSED(module), PyObject *arg) {
    std::map<long, CppCustomObject> input;
```

(continues on next page)

(continued from previous page)

```
    if (! Python_Cpp_Containers::py_dict_to_cpp_std_map_like(arg, input)) {
        std::map<long, CppCustomObject> output;
        for (const auto &iter: input) {
            output.emplace(
                std::make_pair(
                    iter.first,
                    CppCustomObject(
                        iter.second.last(), iter.second.first(), iter.second.number()
                    )
                )
            );
        }
        return Python_Cpp_Containers::cpp_std_map_like_to_py_dict(output);
    }
    return NULL;
}
```

Add this in to the module methods so they look like this:

```
// Module functions
static PyMethodDef cUserDefinedMethods[] = {
        {"reverse_list_names", reverse_list_names, METH_O,
            "Take a list of cUserDefined.Custom objects"
            " and return a new list with the name reversed."},
        {"reverse_dict_names", reverse_dict_names, METH_O,
            "Take a dict of [int, cUserDefined.Custom] objects"
            " and return a new dict with the name reversed."},
        {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

Rebuild the module and try it:

```
>>> import cUserDefined
>>> d = {
    0 : cUserDefined.Custom('First', 'Last', 17953),
    1 : cUserDefined.Custom('François', 'Truffaut', 21468),
}
>>> d
{0: <cUserDefined.Custom object at 0x10e0ec6f0>, 1: <cUserDefined.Custom object at␣
↪0x10e0ec450>}
```

Create a new dict with the names reversed in C++ code. The IDs show that we have new objects:

```
>>> e = cUserDefined.reverse_dict_names(d)
>>> e
{0: <cUserDefined.Custom object at 0x10e2fb4e0>, 1: <cUserDefined.Custom object at␣
↪0x10e2fb1b0>}
```

Chcek that the names have been reversed:

```
>>> {k: v.name() for k, v in e.items()}
{0: 'Last First', 1: 'Truffaut François'}
```

4.2 User Defined Types From Pure Python Types

Todo: Add in version 0.4.0

4.3 Interoperation with `numpy` ND Arrays.

Todo: Add the existing example code in version 0.4.0.

CHAPTER

FIVE

DESIGN

This library uses C++ templates but not in a particularly complex way. There are six essential C++ templates and a Python script is used to to auto-generate the partial template specialisations and their instantiations.

As described in the previous chapter new types can be added pretty easily, alternatively the code generator can be manipulated to do this.

5.1 Source Files `python_object_convert.h` and `python_object_convert.cpp`

These are hand written files that contains implementations of functions to convert Python types to and from their C++ equivalent. There are three functions to each type:

- Convert a C++ value to a new Python object.
- Convert a Python object to a C++ value.
- Check that a Python object is of the expected type.

For example here are the three functions for Python `int` and C++ `long`:

```
PyObject *cpp_long_to_py_long(const long &l);
long py_long_to_cpp_long(PyObject *op);
int py_long_check(PyObject *op);
```

The implementations of these are just one line wrappers around functions or macros in the Python C API.

5.2 `python_container_convert.h` and `python_container_convert.cpp`

This is a hand written file that contains implementations of functions to create and access Python unary containers (`list`, `tuple`, `set`). There are a number off functions to each container, for example a list:

- Check that a Python object is of the expected type.
- Create a new Python container.
- Find the length of a Python container.
- Set a value in a Python container.
- Get a value from a Python container.

For example here are the three functions for Python lists:

```
int py_list_check(PyObject *op);
PyObject *py_list_new(size_t len);
Py_ssize_t py_list_len(PyObject *list_p);
int py_list_set(PyObject *list_p, size_t pos, PyObject *op);
PyObject *py_list_get(PyObject *list_p, size_t pos);
```

The implementations of these are just one line wrappers around functions or macros in the Python C API.

5.3 `python_convert.h`

This is a hand written file that contains templates that convert containers to and fro between Python and C++. It includes `python_object_convert.h` and `python_container_convert.h`, declares the templates then includes `auto_py_convert_internal.h`.

5.4 Python `list` and `tuple`

There are several levels of specialisation here as we want to support conversion from Python `list` and `tuple` to and from `std::vector` and `std::list`.

These functions are described in detail and, for brevity, the functions that handle sets and dicts that follow the same pattern are describe in less detail.

5.4.1 Conversion From C++ to Python

This provides conversion From a `std::vector<T>` or a `std::list<T>` to a Python List or Tuple. Firstly there is a highly generic handwritten function in `python_convert.h`:

```
template<
        template<typename ...> class ListLike,
        typename T,
        PyObject *(*ConvertCppToPy)(const T &),
        PyObject *(*PyUnaryContainer_New)(size_t),
        int(*PyUnaryContainer_Set)(PyObject *, size_t, PyObject *)>
PyObject *
very_generic_cpp_std_list_like_to_py_unary(const ListLike<T> &list_like) {
    // Handwritten code...
}
```

Table 1: Convert a `std::vector` or `std::list` to a Python `tuple` or `list`.

Type	Description
`ListLike`	The C++ container, for example a `std::vector` or a `std::list`.
`typename T`	The C++ type of each object in the container.
`PyObject *(*Convert)(const T &)`	A pointer to a function that takes a type T and returns a new Python `PyObject*`.
`PyObject *(*PyUnary_New)(size_t)`	A pointer to a function that returns a new Python container of the given length.
`int(*PyUnary_Set)(PyObject *, size_t, PyObject *)>`	Sets a Python object in the Python container at the given position.

This template is then partially specified four ways for both Python `tuple` and `list` from both C++ `std::vector<T>` and `std::list<T>`. This is handwritten code in `python_convert.h` but they are, effectively, just one-liners:

```
// C++ std::vector<T> to a Python tuple
template<typename T, PyObject *(*ConvertCppToPy)(const T &)>
PyObject *
generic_cpp_std_list_like_to_py_tuple(const std::vector<T> &container) {
    return very_generic_cpp_std_list_like_to_py_unary<
        std::vector, T, ConvertCppToPy, &py_tuple_new, &py_tuple_set
    >(container);
}

// C++ std::list<T> to a Python tuple
template<typename T, PyObject *(*ConvertCppToPy)(const T &)>
PyObject *
generic_cpp_std_list_like_to_py_tuple(const std::list<T> &container) {
    return very_generic_cpp_std_list_like_to_py_unary<
        std::list, T, ConvertCppToPy, &py_tuple_new, &py_tuple_set
    >(container);
}

// C++ std::vector<T> to a Python list
template<typename T, PyObject *(*ConvertCppToPy)(const T &)>
PyObject *
generic_cpp_std_list_like_to_py_list(const std::vector<T> &container) {
    return very_generic_cpp_std_list_like_to_py_unary<
        std::vector, T, ConvertCppToPy, &py_list_new, &py_list_set
    >(container);
}

// C++ std::list<T> to a Python list
template<typename T, PyObject *(*ConvertCppToPy)(const T &)>
PyObject *
generic_cpp_std_list_like_to_py_list(const std::list<T> &container) {
    return very_generic_cpp_std_list_like_to_py_unary<
        std::list, T, ConvertCppToPy, &py_list_new, &py_list_set
    >(container);
}
```

Then these are specialised by auto-generated code in `auto_py_convert_internal.h` for the specific types `bool`, `long`, `double`, `std::vector<char>` and `std::string`.

For brevity only the declarations and definitions are shown for the type `long`. For example to create a Python `tuple` from a C++ `std::vector` the base declaration for any type `T` is:

```
// Base declaration
template<typename T>
PyObject *
cpp_std_list_like_to_py_tuple(const std::vector<T> &container);
```

And the declaration for type `long` in `auto_py_convert_internal.h` is:

```
// Instantiations
template <>
PyObject *
cpp_std_list_like_to_py_tuple<long>(const std::vector<long> &container);
```

The definitions are auto-generated in `auto_py_convert_internal.cpp`, for example for C++ type `long`. These are just one-liners:

```
template <>
PyObject *
cpp_std_list_like_to_py_tuple<long>(const std::vector<long> &container) {
    return generic_cpp_std_list_like_to_py_tuple<long, &cpp_long_to_py_long>(container);
}
```

That is for `std::vector`, for `std::list` the declarations and definitions are very similar. Firstly in `auto_py_convert_internal.h`, again just showing for `long`:

```
// Base declaration
template<typename T>
PyObject *
cpp_std_list_like_to_py_tuple(const std::list<T> &container);

// Instantiations
template <>
PyObject *
cpp_std_list_like_to_py_tuple<long>(const std::list<long> &container);

// And so on...
```

And the declarations auto-generated in `auto_py_convert_internal.cpp`:

```
template <>
PyObject *
cpp_std_list_like_to_py_tuple<long>(const std::list<long> &container) {
    return generic_cpp_std_list_like_to_py_tuple<long, &cpp_long_to_py_long>(container);
}

// And so on...
```

5.4.2 Conversion From Python to C++

This covers conversion From a Python `list` or `tuple` to a C++ `std::vector<T>` or `std::list<T>`. It follows a similar pattern as described above.

Firstly there is a highly generic handwritten function in `python_convert.h`:

```
template<
        template<typename ...> class ListLike,
        typename T,
        int (*PyObject_Check)(PyObject *),
        T (*PyObject_Convert)(PyObject *),
        int(*PyUnaryContainer_Check)(PyObject *),
        Py_ssize_t(*PyUnaryContainer_Size)(PyObject *),
        PyObject *(*PyUnaryContainer_Get)(PyObject *, size_t)>
int very_generic_py_unary_to_cpp_std_list_like(PyObject *op, ListLike<T> &list_like) {
    // Handwritten code
}
```

Template parameters are:

Table 2: Template to convert a `std::vector` or `std::list` to a Python `tuple` or `list`.

Type	Description
`ListLike`	The C++ container, for example a `std::vector` or a `std::list`.
`typename T`	The C++ type of the object.
`int (*PyObject_Check)(PyObject *)`	A pointer to a function returns true if Python object can be converted to a C++ object of type `T`.
`T (*PyObject_Convert)(PyObject *)`	A pointer to a function to convert a Python object to a C++ object of type `T`.
`int(*PyUnaryContainer_Check)(PyObject *)`	A pointer to a function returns true if the Python container is of the relevant type (`list` or `tuple` in this case).
`Py_ssize_t(*PyUnaryContainer_Size)(PyObject *)`	A pointer to a function that returns the size of the Python container.
`PyObject *(*PyUnaryContainer_Get)(PyObject *, size_t)`	Gets a Python object in the Python container at the given position.

Parameters are:

Table 3: Function to convert a `std::vector` or `std::list` to a Python `tuple` or `list`.

Parameter	Description
`op`	The Python container.
`list_like`	The C++ container. This will be empty on failure.

This returns zero on success, non-zero on failure. Failure reasons can be:

- The Python object is not the expected container type.
- A Python object in the container is `NULL`.
- A Python object in the container can not be converted to a C++ type `T`.

This template is then partially specified with handwritten code. Here is the handwritten code in `python_convert.h` for Python `tuple` to a C++ `std::vector` or `std::list`. They are basically one-liners, the interesting variation is for the `std::vector` where we exploit `.reserve()` to reduce reallocations.

```
template<typename T, int (*PyObject_Check)(PyObject *), T (*PyObject_Convert)(PyObject␣
↪*)>
int generic_py_tuple_to_cpp_std_list_like(PyObject *op, std::vector<T> &container) {
    // Reserve the vector, but only if it is a tuple. If not then ignore it as
    // very_generic_py_unary_to_cpp_std_list_like() will error
    if (py_tuple_check(op)) {
        container.reserve(py_tuple_len(op));
    }
    return very_generic_py_unary_to_cpp_std_list_like<
        std::vector, T, PyObject_Check, PyObject_Convert,
        &py_tuple_check, &py_tuple_len, &py_tuple_get
    >(op, container);
}

template<typename T, int (*PyObject_Check)(PyObject *), T (*PyObject_Convert)(PyObject␣
↪*)>
int generic_py_tuple_to_cpp_std_list_like(PyObject *op, std::list<T> &container) {
    return very_generic_py_unary_to_cpp_std_list_like<
        std::list, T, PyObject_Check, PyObject_Convert,
        &py_tuple_check, &py_tuple_len, &py_tuple_get
    >(op, container);
}
```

The declarations for Python `tuple` to a C++ `std::vector` are auto-generated in `auto_py_convert_internal.h`. Here shown just for `long`:

```
// Base declaration
template<typename T>
int
py_tuple_to_cpp_std_list_like(PyObject *op, std::vector<T> &container);

// Instantiations
template <>
int
py_tuple_to_cpp_std_list_like<long>(PyObject *op, std::vector<long> &container);
```

The definitions are auto-generated in `auto_py_convert_internal.cpp`, here shown just for `long`:

```
template <>
int
py_tuple_to_cpp_std_list_like<long>(PyObject *op, std::list<long> &container) {
    return generic_py_tuple_to_cpp_std_list_like<
        long, &py_long_check, &py_long_to_cpp_long
    >(op, container);
}
```

5.5 Python set and frozenset

Here is the handwritten code in `python_convert.h` supports the conversion too and from a Python `set` or `frozenset` to and from a a C++ `std::unordered_set`.

5.5.1 Conversion From C++ to Python

```
template<
        typename T,
        PyObject *(*ConvertCppToPy)(const T &),
        PyObject *(*PyContainer_New)(PyObject *)
>
PyObject *
generic_cpp_std_unordered_set_to_py_set_or_frozenset(const std::unordered_set<T> &set) {
    // Handwritten
}
```

Here is the handwritten specialisations in `python_convert.h` supports the conversion too and from a Python `set` and `frozenset`. They are basically one-liners.

```
template<typename T, PyObject *(*ConvertCppToPy)(const T &)>
PyObject *
generic_cpp_std_unordered_set_to_py_set(const std::unordered_set<T> &set) {
    return generic_cpp_std_unordered_set_to_py_set_or_frozenset<
        T, ConvertCppToPy, &PySet_New
    >(set);
}

template<typename T, PyObject *(*ConvertCppToPy)(const T &)>
PyObject *
generic_cpp_std_unordered_set_to_py_frozenset(const std::unordered_set<T> &set) {
    return generic_cpp_std_unordered_set_to_py_set_or_frozenset<
        T, ConvertCppToPy, &PyFrozenSet_New
    >(set);
}
```

Then declarations are auto-generated in `auto_py_convert_internal.h`, here shown just for a Python `set` containing `long`:

```
// Base declaration
template<typename T>
PyObject *
cpp_std_unordered_set_to_py_set(const std::unordered_set<T> &container);

// Instantiations
template <>
PyObject *
cpp_std_unordered_set_to_py_set<long>(const std::unordered_set<long> &container);

// And so on..
```

The definitions are auto-generated in `auto_py_convert_internal.cpp`, here shown just for a Python `set` containing `long`:

```
template <>
PyObject *
cpp_std_unordered_set_to_py_set<long>(const std::unordered_set<long> &container) {
    return generic_cpp_std_unordered_set_to_py_set<long, &cpp_long_to_py_long>
↪(container);
}

// And so on..
```

5.5.2 Conversion From Python to C++

```
template<
        typename T,
        int (*PyContainer_Check)(PyObject *),
        int (*PyObject_Check)(PyObject *),
        T (*PyObject_Convert)(PyObject *)
>
int generic_py_set_or_frozenset_to_cpp_std_unordered_set(
    PyObject *op, std::unordered_set<T> &set
) {
    // Handwritten
}
```

```
template<typename T, int (*PyObject_Check)(PyObject *), T (*PyObject_Convert)(PyObject␣
↪*)>
int generic_py_set_to_cpp_std_unordered_set(PyObject *op, std::unordered_set<T> &set) {
    return generic_py_set_or_frozenset_to_cpp_std_unordered_set<
        T, &py_set_check, PyObject_Check, PyObject_Convert
    >(op, set);
}

template<typename T, int (*PyObject_Check)(PyObject *), T (*PyObject_Convert)(PyObject␣
↪*)>
int generic_py_frozenset_to_cpp_std_unordered_set(PyObject *op, std::unordered_set<T> &
↪set) {
    return generic_py_set_or_frozenset_to_cpp_std_unordered_set<
        T, &py_frozenset_check, PyObject_Check, PyObject_Convert
    >(op, set);
}
```

The declarations are auto-generated in `auto_py_convert_internal.h`, here shown just for a Python `set` containing `long`:

```
// Base declaration
template<typename T>
int
py_set_to_cpp_std_unordered_set(
    PyObject *op, std::unordered_set<T> &container
);

// Instantiations
```

(continues on next page)

(continued from previous page)

```
template <>
int
py_set_to_cpp_std_unordered_set<long>(
    PyObject *op, std::unordered_set<long> &container
);

// And so on..
```

The definitions are auto-generated in `auto_py_convert_internal.cpp`, here shown just for a Python `set` containing `long`:

```
template <>
int
py_set_to_cpp_std_unordered_set<long>(
    PyObject *op, std::unordered_set<long> &container
) {
    return generic_py_set_to_cpp_std_unordered_set<
        long, &py_long_check, &py_long_to_cpp_long
    >(op, container);
}

// And so on..
```

5.6 Python `dict`

This supports the two-way conversion from a Python `dict` to and from a C++ `std::unordered_map` or a `std::map`.

5.6.1 Conversion From C++ to Python

A hanbdwritten function in `python_convert.h` provides the basis for specialisation:

```
template<
        template<typename ...> class Map,
        typename K,
        typename V,
        PyObject *(*Convert_K)(const K &),
        PyObject *(*Convert_V)(const V &)
>
PyObject *
generic_cpp_std_map_like_to_py_dict(const Map<K, V> &map) {
    // Handwritten function.
}
```

The specialised declarations are auto-generated in `auto_py_convert_internal.h`, here shown just for a Python `dict` from a `std::unordered_map` or a `std::map` containing `long, long`:

```
// Base declaration
template<template<typename ...> class Map, typename K, typename V>
PyObject *
```

(continues on next page)

(continued from previous page)

```
cpp_std_map_like_to_py_dict(const Map<K, V> &map);

// Instantiations
template <>
PyObject *
cpp_std_map_like_to_py_dict<std::unordered_map, long, long>(
    const std::unordered_map<long, long> &map
);

template <>
PyObject *
cpp_std_map_like_to_py_dict<std::map, long, long>(
    const std::map<long, long> &map
);
```

The definitions are auto-generated in `auto_py_convert_internal.cpp`, here shown just for a Python `dict` from a `std::unordered_map` containing `long, long`:

```
template <>
PyObject *
cpp_std_map_like_to_py_dict<std::unordered_map, long, long>(
    const std::unordered_map<long, long> &map
) {
    return generic_cpp_std_map_like_to_py_dict<
        std::unordered_map,
        long, long,
        &cpp_long_to_py_long, &cpp_long_to_py_long
    >(map);
}
```

5.6.2 Conversion From Python to C++

The reverse, converting from Python to C++, is accomplished by a single handwritten template in `python_convert.h`:

```
template<
        template<typename ...> class Map,
        typename K,
        typename V,
        int (*Check_K)(PyObject *),
        int (*Check_V)(PyObject *),
        K (*Convert_K)(PyObject *),
        V (*Convert_V)(PyObject *)
>
int generic_py_dict_to_cpp_std_map_like(PyObject *dict, Map<K, V> &map) {
    // Handwritten function.
}
```

The declarations are auto-generated in `auto_py_convert_internal.h`, here shown just for a Python `dict` from a `std::unordered_map` or `std::map` containing `long, long`:

```cpp
// Base declaration
template<template<typename ...> class Map, typename K, typename V>
int
py_dict_to_cpp_std_map_like(PyObject *op, Map<K, V> &map);

// Instantiations
template <>
int
py_dict_to_cpp_std_map_like<std::unordered_map, long, long>(
    PyObject* op, std::unordered_map<long, long> &map
);

template <>
int
py_dict_to_cpp_std_map_like<std::map, long, long>(
    PyObject* op, std::map<long, long> &map
);
```

The definitions are auto-generated in `auto_py_convert_internal.cpp`, here shown just for a Python `dict` from a `std::unordered_map` containing `long, long`:

```cpp
template <>
int
py_dict_to_cpp_std_map_like<std::unordered_map, long, long>(
    PyObject* op, std::unordered_map<long, long> &map
) {
    return generic_py_dict_to_cpp_std_map_like<
        std::unordered_map,
        long, long,
        &py_long_check, &py_long_check,
        &py_long_to_cpp_long, &py_long_to_cpp_long
    >(op, map);
}
```

5.7 Code Generation

Todo: Add in version 0.4.0

CHAPTER

SIX

PERFORMANCE

Here are some benchmarks for converting Python containers to and from their C++ equivalents.

The C++ code was compiled with `-O3` and run on the following hardware:

```
Model Name:                   MacBook Pro
Model Identifier:             MacBookPro15,2
Processor Name:               Intel Core i7
Processor Speed:              2.7 GHz
Number of Processors:         1
Total Number of Cores:        4
L2 Cache (per Core):          256 KB
L3 Cache:                     8 MB
Hyper-Threading Technology: Enabled
Memory:                       16 GB

System Version:               macOS 10.14.6
```

6.1 C++ Performance Tests

These tests are in `src/cpy/tests/test_performance.h` and `src/cpy/tests/test_performance.cpp`. There are a number of macros `TEST_PERFORMANCE_*` there that control which tests are run. Running all tests takes about 900 seconds.

6.1.1 Conversion of Fundamental Types

These C++ functions test the cost of converting ints, floats and bytes objects between Python and C++. These test are executed if the macro `TEST_PERFORMANCE_FUNDAMENTAL_TYPES` is defined.

Operation	C++ to Python (µs)	Python to C++ (µs)	Notes
C++ `bool` <-> Python `bool`	0.0027	0.0016	The mean is around 400m/s
C++ `long` <-> Python `int`	0.0146	0.0046	The mean is around 50m/s. Converting C++ to Python is around x3 times the reverse.
C++ `double` <-> Python `float`	0.0086	0.0027	The mean is around 200m/s. Converting C++ to Python is around x3 times the reverse.
C++ `std::complex<double>` <-> Python `complex`	0.0122	0.0049	The mean is around 125m/s. Converting C++ to Python is around x2.5 times the reverse.

For a single C++ `std::vector<char>` to and from Python `bytes` of different lengths:

Length	C++ to Python (µs)	Python to C++ (µs)	Notes
2	0.0173	0.0047	
16	0.0169	0.0040	
128	0.0201	0.0641	
1024	0.0807	0.0671	Corresponds to about 14 Gb/s
8192	0.1317	0.1197	Corresponds to about 64 Gb/s
65536	1.567	1.551	Corresponds to about 41 Gb/s

Bytes conversion time from C++ to Python or the reverse takes asymptotically and roughly: t (µs) = 0.017 * length / 50,000

For a single C++ `std::string` to and from Python `str` of different lengths:

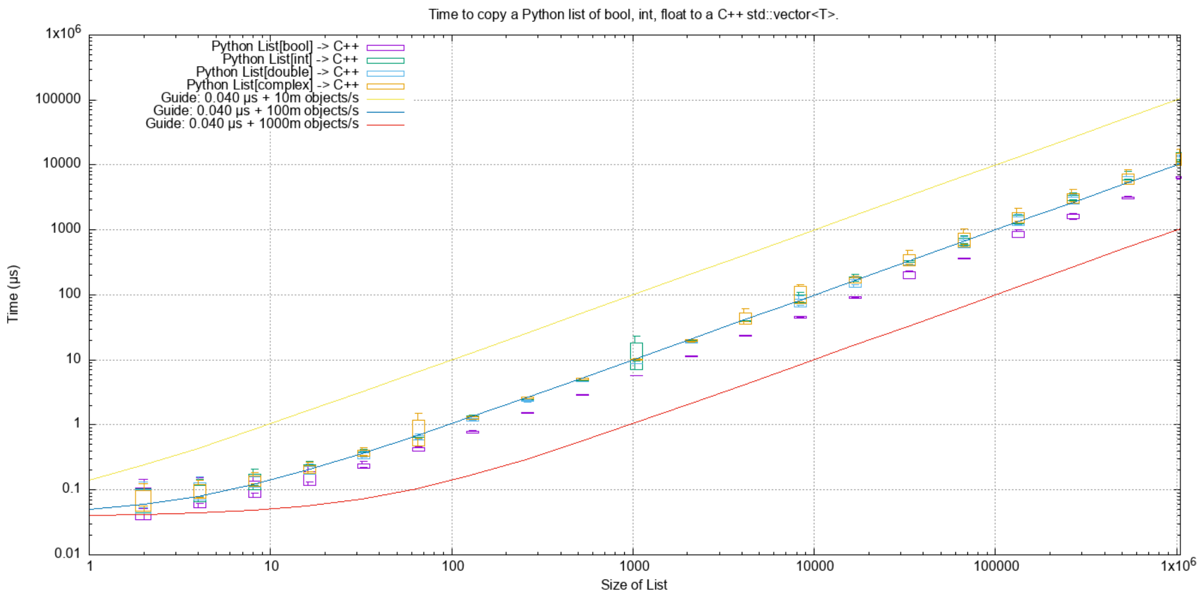
Length	C++ to Python (µs)	Python to C++ (µs)	Notes
2	0.0309	0.0052	
16	0.0337	0.0045	
128	0.0301	0.0634	
1024	0.126	0.0667	Corresponds to about 8 to 15 Gb/s, Python to C++ being about twice as fast.
8192	0.435	0.122	Corresponds to about 20 to 65 Gb/s, Python to C++ being about thrice as fast.
65536	3.46	1.53	Corresponds to about 20 to 40 Gb/s, Python to C++ being about twice as fast.

String conversion time from C++ to Python or the reverse takes asymptotically and roughly: t (µs) = 0.015 * length / 24,000. This is about twice the time for `bytes` and `std::vector<char>`.

6.1.2 Python List to and from a C++ `std::vector<T>`

This as an extensive example of the methodology used for performance tests. Each container test is repeated 5 times and the min/mean/max/std. dev. is recorded. The min value is regarded as the most consistent one as other results may be affected by arbitrary context switching. The tests are run on containers of lengths up to 1m items.

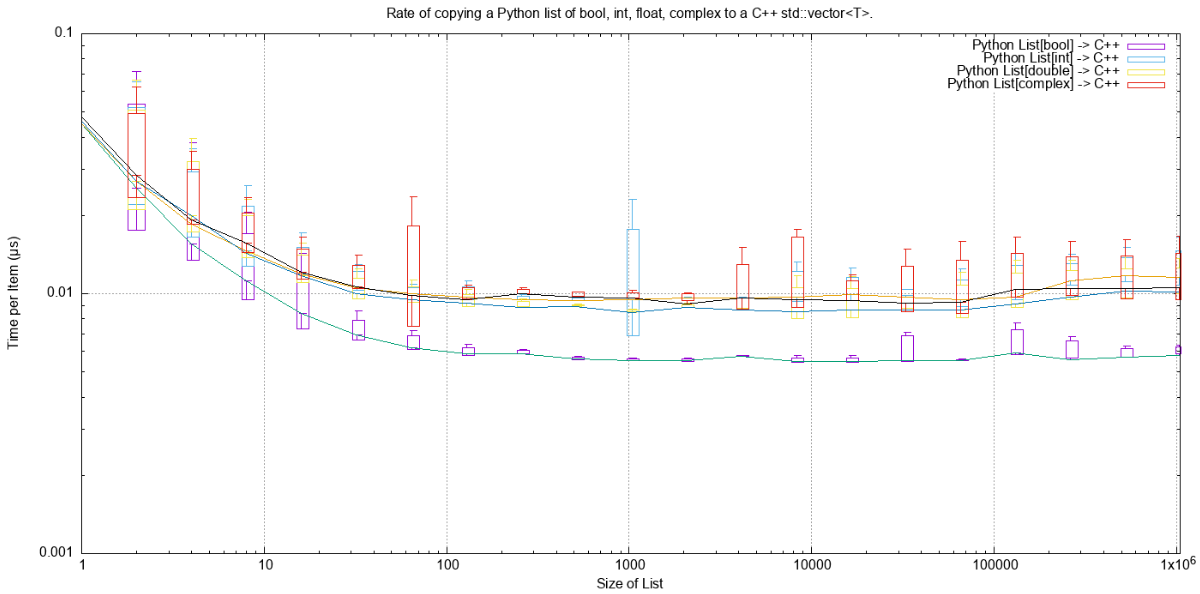
For example here is the total time to convert a list of `bool`, `int`, `float` and `complex` Python values to C++ for various list lengths:



This time plot is not that informative apart from showing linear behaviour. More useful are *rate* plots that show the total time for the test divided by the container length. These rate plots have the following design features:

- For consistency a rate scale of µs/item is used.
- The extreme whiskers show the minimum and maximum test values.
- The box shows the mean time ±the standard deviation, this is asymmetric as it is plotted on a log scale.
- The box will often extend beyond a minimum value where the minimum is close to the mean and the maximum large.
- The line shows the minimum time per object in µs.

Here is the rate of converting a list of `bool`, `int`, `float` and `complex` Python values to C++ for various list lengths:



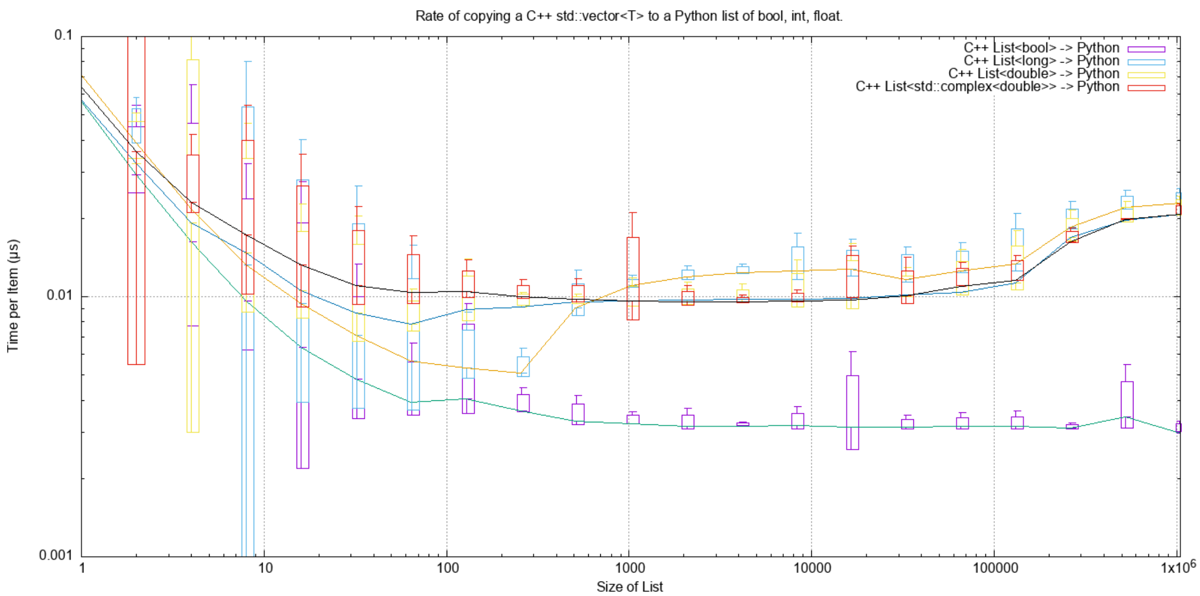
These rate plots are used for the rest of this section.

Lists of `bool`, `int`, `float` and `complex`

The rate plot is shown above, it shows that:

- `int`, `float` and `complex` take 0.01 µs per object to convert.
- `bool` objects take around 0.006 µs per object, roughly twice as fast.

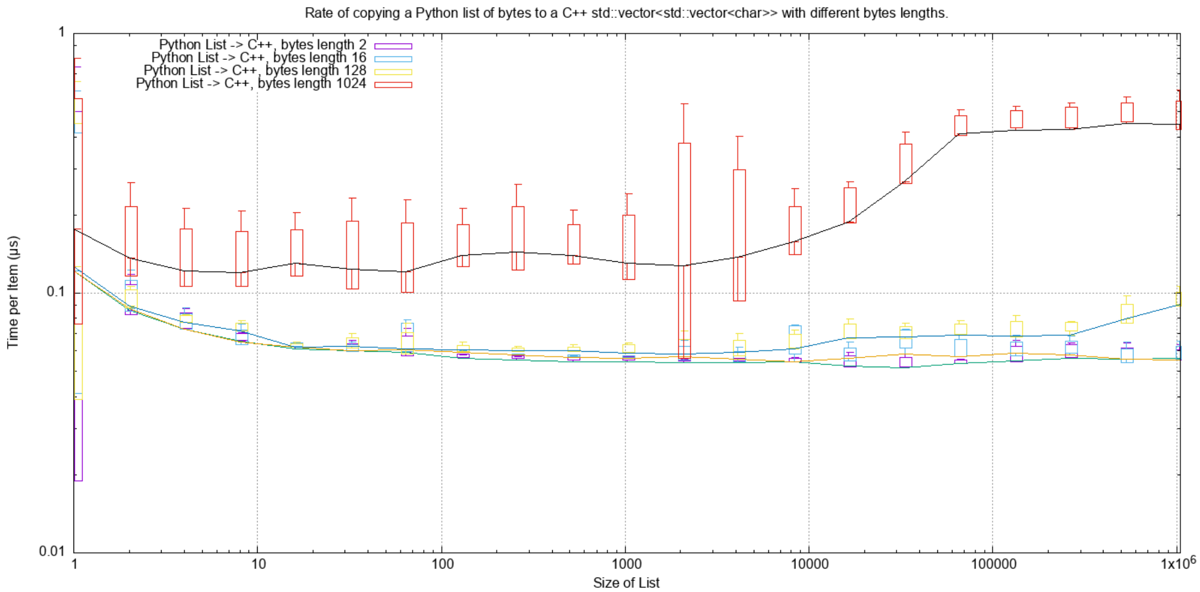
And the reverse converting a list of `bool`, `int`, `float` and `complex` from C++ to Python:



This is broadly symmetric with the Python to C++ performance except that `bool` values are twice as quick.

Lists of bytes

Another area of interest is the conversion of a list of bytes or str between Python and C++. In these tests a list of of bytes or str objects of lengths 2, 16, 128 and 1024 are used to convert from Python to C++.

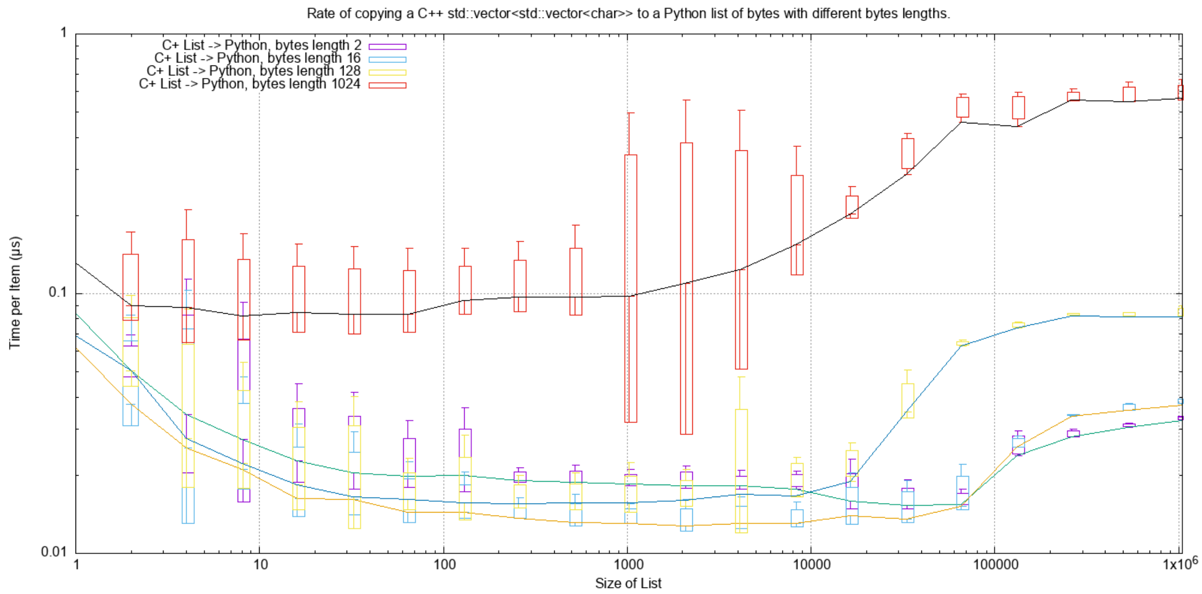


This graph shows a characteristic rise in rate for larger list lengths of larger objects. This is most likely because of memory contention issues with the larger, up to 1GB, containers. This characteristic is observed on most of the following plots, particularly with containers of bytes and str.

In summary:

Object	~Time per object (µs)	Rate Mb/s	Notes
bytes[2]	0.06	30	
bytes[16]	0.06	270	
bytes[128]	0.06	2,000	
bytes[1024]	0.15 to 0.4	2,500 to 6,800	

This is the inverse, converting a C++ std::vector<std::vector<char>> to a Python list of bytes:

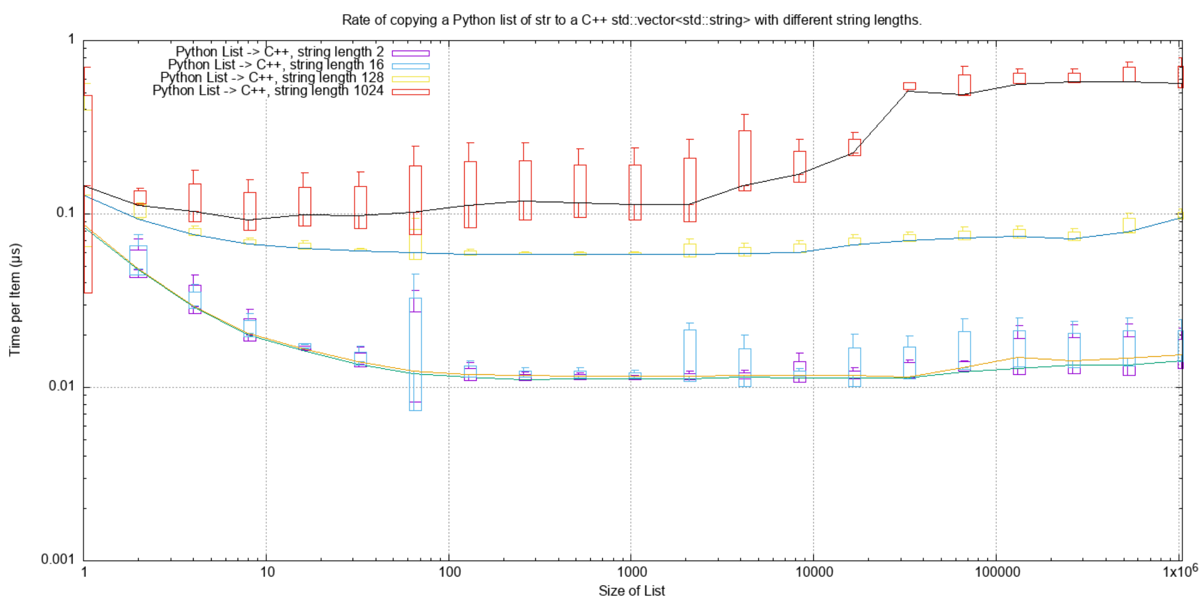


Object	~Time per object (µs)	Rate Mb/s	Notes
bytes[2]	0.015 to 0.03	67 to 133	
bytes[16]	0.015 to 0.04	400 to 133	
bytes[128]	0.02 to 0.09	1,400 to 6,400	
bytes[1024]	0.1 to 0.6	1,600 to 10,000	

This shows that converting C++ to Python is about twice as fast as the other way around. This is in line with the performance of conversion of fundamental types described above.

Lists of `str`

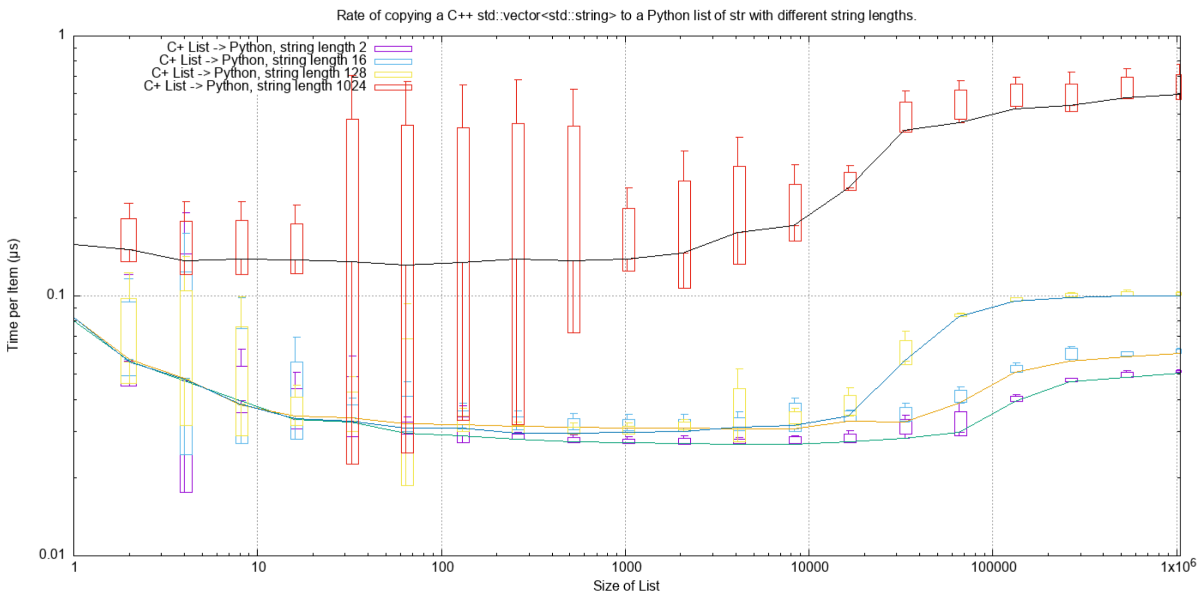
Similarly for converting a a Python list of `str` to and from a C++ `std::vector<std::string>`. First Python -> C++:



Notably with small strings (2 and 16 long) are about eight times faster that for bytes. For larger strings this perfformance is very similar to Python `bytes` to a C++ `std::vector<std::vector<char>>`:

Object	~Time per object (µs)	Rate Mb/s	Notes
str[2]	0.01	200	
str[16]	0.01	1600	
str[128]	0.07	1,800	
str[1024]	0.1 to 0.6	1,600 to 10,000	

And C++ -> Python:



Object	~Time per object (µs)	Rate Mb/s	Notes
str[2]	0.03	70	
str[16]	0.03	500	
str[128]	0.03 to 0.1	1,300 to 4,000	
str[1024]	0.15 to 0.6	1,700 to 6,800	

Slightly slower than the twice the time for converting `bytes` especially for small strings this is abut twice the time for converting `bytes` but otherwise very similar to Python `bytes` to a C++ `std::vector<std::vector<char>>`:

6.1.3 Python Tuple to and from a C++ `std::vector<T>`

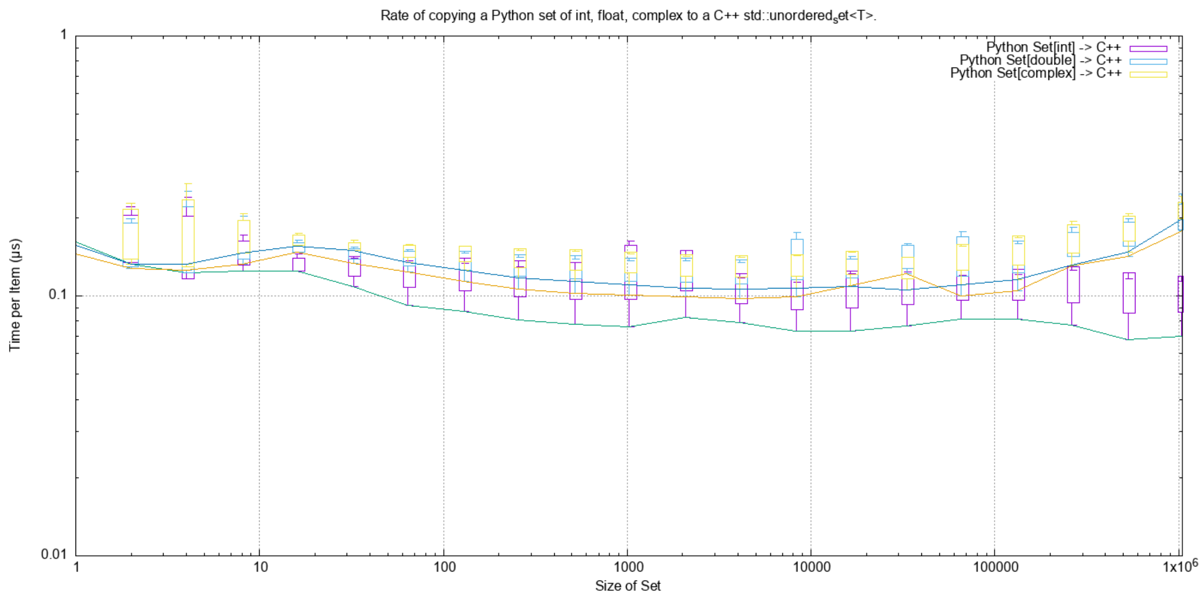
This is near identical to the performance of a list for:

- The conversion of `bool`, `int`, `float` and `complex` for Python to C++ and C++ to Python.
- The conversion of `bytes` for Python to C++ and C++ to Python.
- The conversion of `str` for Python to C++ and C++ to Python.

6.1.4 Python Set to and from a C++ `std::unordered_set<T>`

Set of `int`, `float` and `complex`

Here is the rate graph for converting a Python `set` to C++ `std::unordered_set<T>` for Python `int`, `float` and `complex` objects:

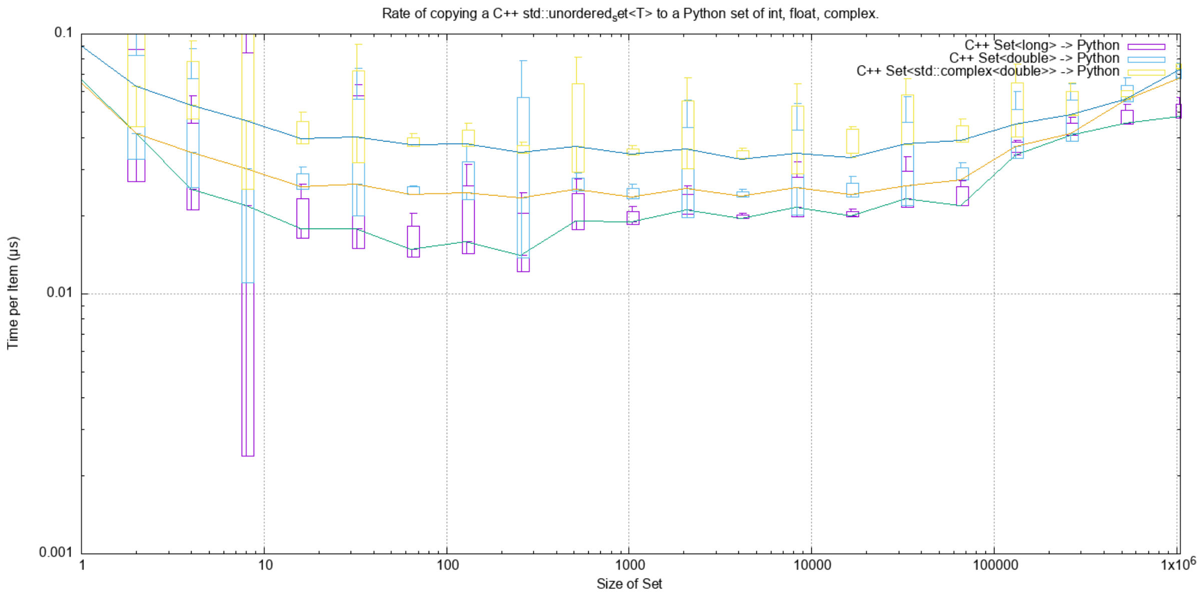


Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
int	0.09	0.01	x9	
double	0.1	0.01	x10	
complex	0.1	0.01	x10	

The cost of insertion is O(N) for both list and set but due to the hashing heeded for the set it is about 10x slower.

And the reverse, converting a C++ `std::unordered_set<T>` to a Python `set` to for Python `int`, `float` and `complex` objects:

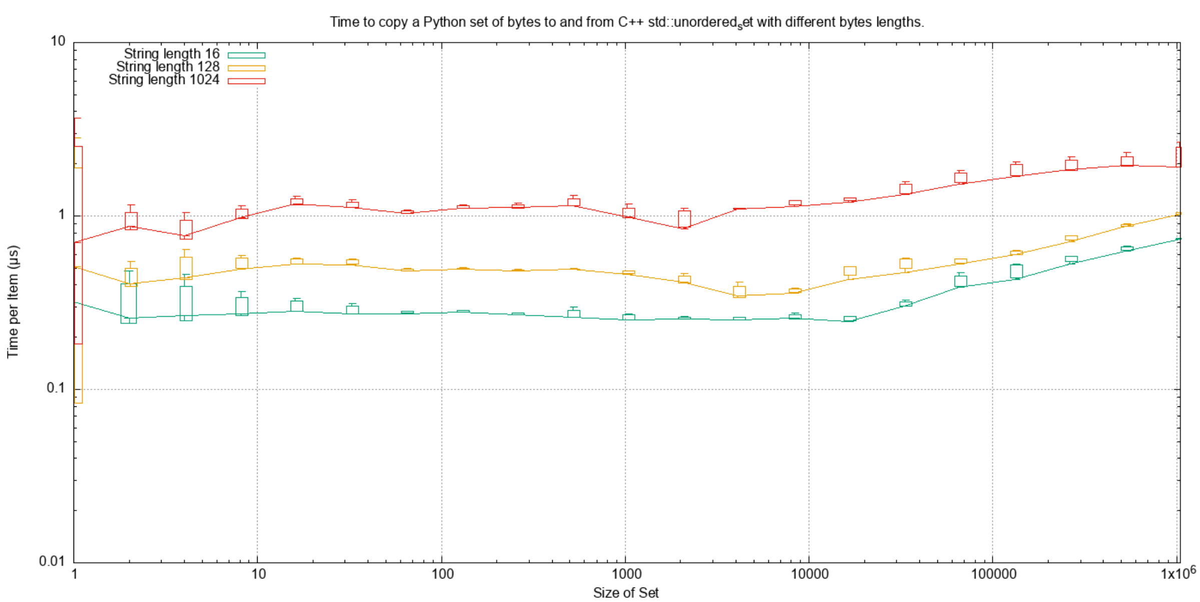


The conversion and insertion of C++ to Python is significantly faster that from Python to C++. Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
int	0.02	0.01	x2	
double	0.025	0.01	x2.5	
complex	0.04	0.01	x4	

Set of `bytes`

Here is the rate graph for converting a Python `set` of `bytes` to C++ `std::unordered_set<std::vector<char>>`:

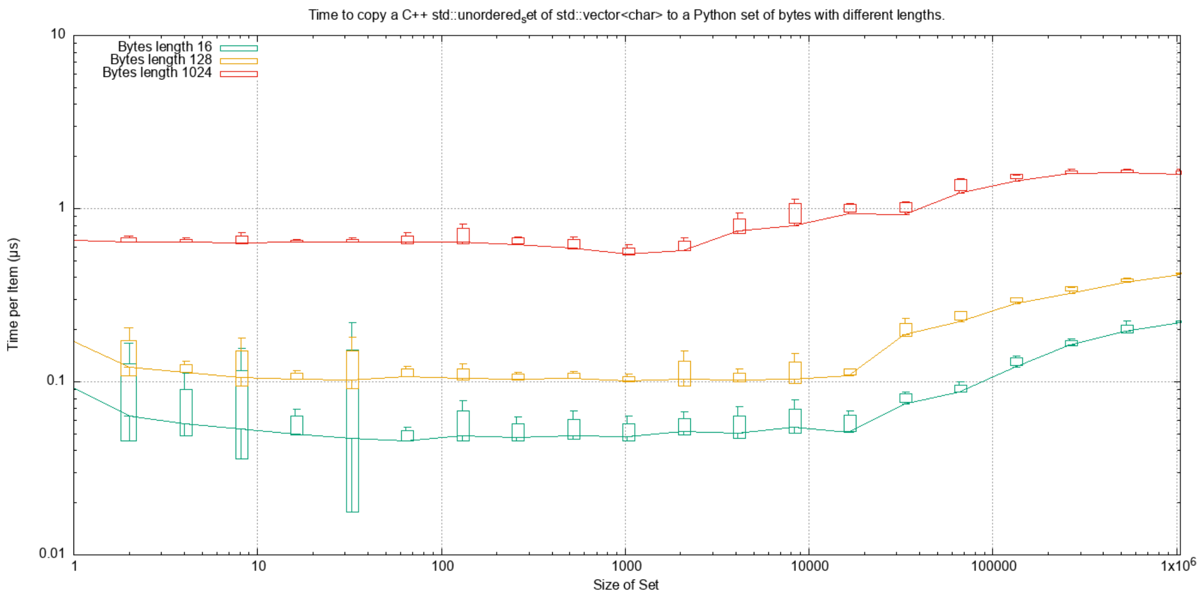


Object	~Time per object (µs)	Rate Mb/s	Notes
bytes[16]	0.4	40	
bytes[128]	0.5	250	
bytes[1024]	1.0	1,000	

Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
bytes[16]	0.4	0.06	x7	
bytes[128]	0.5	0.06	x8	
bytes[1024]	1.0	0.15 to 0.4	x2.5 to x7	

And the reverse, converting a C++ `std::unordered_set<std::vector<char>>` to a Python `set` of `bytes`:



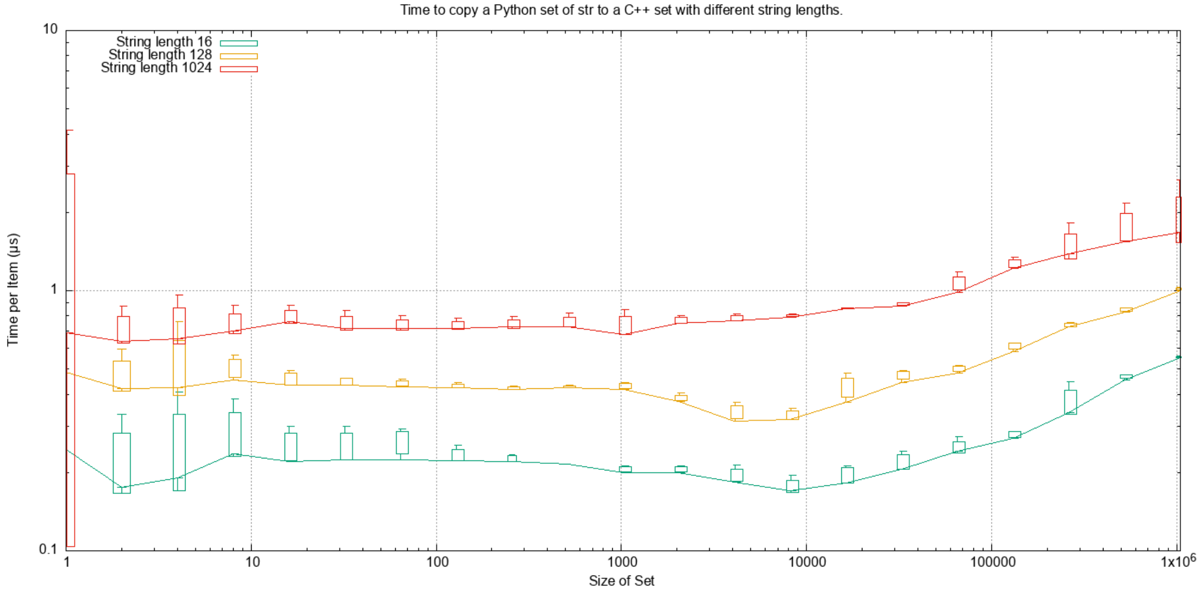
Object	~Time per object (µs)	Rate Mb/s	Notes
bytes[16]	0.05	320	
bytes[128]	0.1	1,280	
bytes[1024]	0.6	1,600	

Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
bytes[16]	0.05	0.015 to 0.04	x3 to x1.25	
bytes[128]	0.1	0.02 to 0.09	x1 to x5	
bytes[1024]	0.6	0.1 to 0.6	x1 to x6	

Set of `str`

Here is the rate graph for converting a Python `set` of `str` to C++ `std::unordered_set<std::string>`:

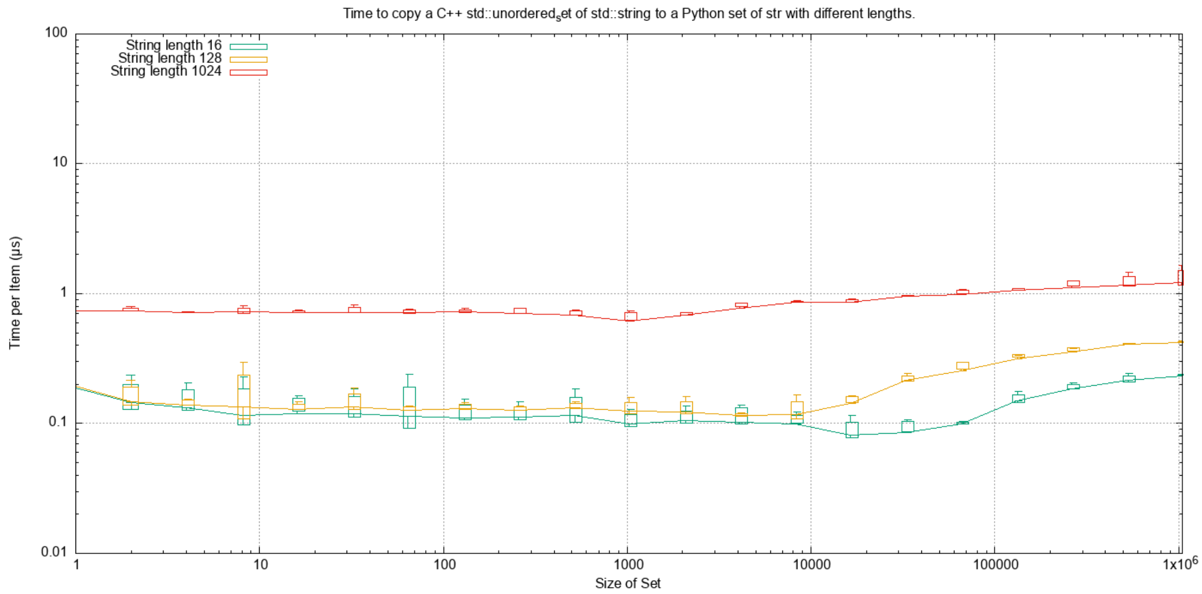


Object	~Time per object (µs)	Rate Mb/s	Notes
bytes[16]	0.2	80	
bytes[128]	0.4	3000	
bytes[1024]	0.5 to 2.0	500 to 2,000	

Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
bytes[16]	0.2	0.01	x20	
bytes[128]	0.4	0.07	x6	
bytes[1024]	0.5 to 2.0	0.1 to 0.6	~x5	

And the reverse, converting a C++ `std::unordered_set<std::string>` to a Python `set` of `str`:



Object	~Time per object (µs)	Rate Mb/s	Notes
bytes[16]	0.08	200	
bytes[128]	0.15	850	
bytes[1024]	0.8	1,300	

Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
bytes[16]	0.08	0.03	x3	
bytes[128]	0.15	0.03	x5	
bytes[1024]	0.8	0.15	x5	

6.1.5 Python Dict to and from a C++ `std::unordered_map<K, V>`

Since dictionaries operate in much the same way as sets the performance is rather similar. For brevity the full results of dictionaries are not reproduced here, instead here is a summary of the performance of a dictionary compared to a set.

Object	Python to C++	C++ to Python	Notes
`int`, `float`, `complex`	Same as a set	Twice that of a set	
`bytes`	Slightly slower than a set	Twice that of a set	
`str`	Same as a set	Twice that of a set	

6.1.6 Summary

Converting Individual Objects

- `bool`, `int`, `float`, `complex` from C++ to Python is around two to three times faster than from Python to C++.
- Converting `bytes` from C++ to Python is the same as from Python to C++. This is memory bound at around 50 Gb/s.
- With `str` then Python to C++ is about twice as fast as C++ to Python. With the former performance is twice as fast as `bytes`, for the latter it is broadly similar to `bytes` conversion.

Converting Containers of Objects

- The performance of Python `lists` and `tuple` is the same.
- For Python `list` containers converting C++ to Python may be 2x faster in some cases compared to Python to C++.
- For Python `list` containing `bytes` and `str` objects are converted at a rate of 2 to 5 Gib/s, with some latency.
- Python `set` <-> C++ `std::unordered_set` and Python `dict` <-> C++ `std::unordered_map` conversion is typically x3 to x10 times slower than for lists and tuples.

6.2 Round-trip Python to C++ and back to Python

This uses some methods in the `cPyCppContainers` module that takes a Python container, converts it to a new C++ container and then converts that to a new Python container. Timing is done in the Python interpreter.

This template converts a Python list to C++ and back:

```
#include "cpy/python_convert.h"

using namespace Python_Cpp_Containers;

template<typename T>
static PyObject *
new_list(PyObject *arg) {
    std::vector<T> vec;
    if (!py_list_to_cpp_std_vector(arg, vec)) {
        return cpp_std_vector_to_py_list(vec);
    }
    return NULL;
}
```

Then the extension has the following instantiations for `bool`, `int`, `float`, `complex`, `bytes` and `str`:

```
static PyObject *
new_list_bool(PyObject *Py_UNUSED(module), PyObject *arg) {
    return new_list<bool>(arg);
}

static PyObject *
new_list_float(PyObject *Py_UNUSED(module), PyObject *arg) {
```

(continues on next page)

(continued from previous page)

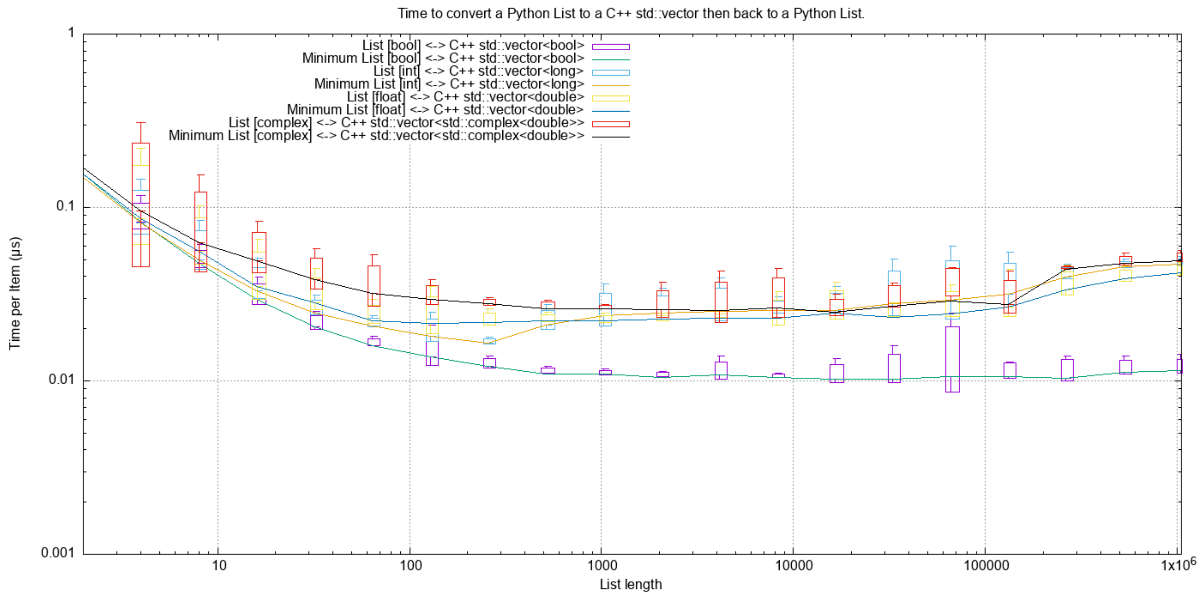
```
    return new_list<double>(arg);
}

static PyObject *
new_list_int(PyObject *Py_UNUSED(module), PyObject *arg) {
    return new_list<long>(arg);
}

static PyObject *
new_list_complex(PyObject *Py_UNUSED(module), PyObject *arg) {
    return new_list<std::complex<double>>(arg);
}

static PyObject *
new_list_bytes(PyObject *Py_UNUSED(module), PyObject *arg) {
    return new_list<std::vector<char>>(arg);
}

static PyObject *
new_list_str(PyObject *Py_UNUSED(module), PyObject *arg) {
    return new_list<std::string>(arg);
}
```

Similar code exists for Python sets and dicts of specific types. Since the tuple conversion C++ code is essentially identical to the list conversion code no performance tests are done on tuples. It might be that the Python C API for tuples is significantly different than for list but this is considered unlikely.

6.2.1 Python Lists

Python List of `bool`, `int`, `float` and `complex`

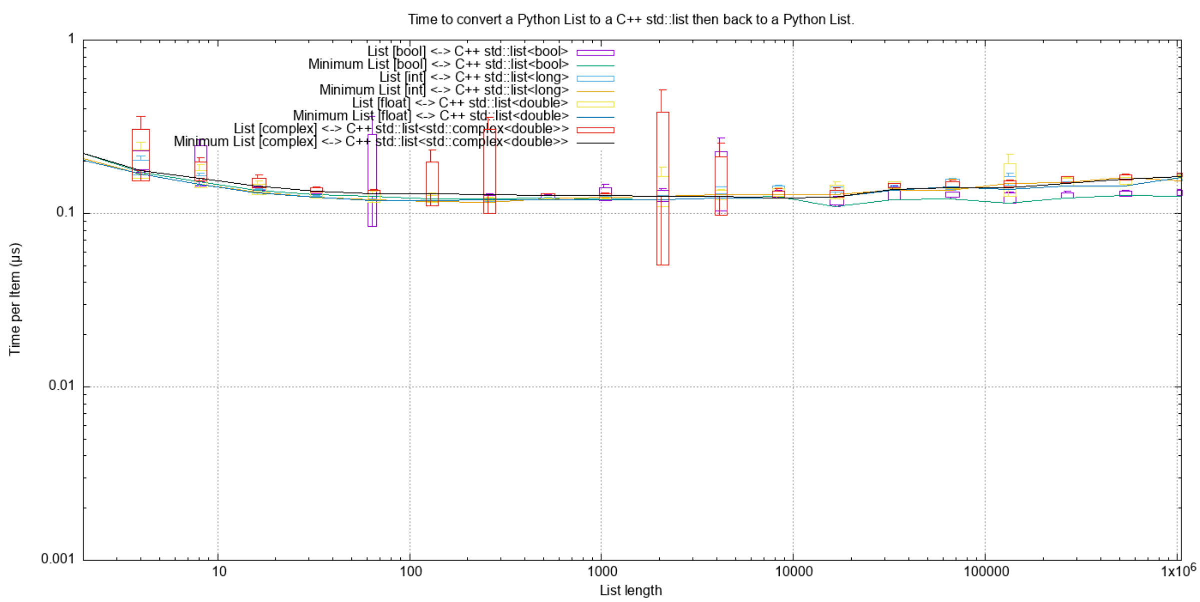
Here is the *round trip* performance of a Python list of `bool`, `int`, `float` and `complex` numbers via a C++ `std::vector`:



These are typically *round trip* converted at:

- 0.01 µs per object for booleans, say 100m objects a second.
- 0.025 µs per object for `int`, `float` and `complex`, say 40m objects a second.

And the *round trip* performance of a Python list of `bool`, `int`, `float` and `complex` numbers via a C++ `std::list`:

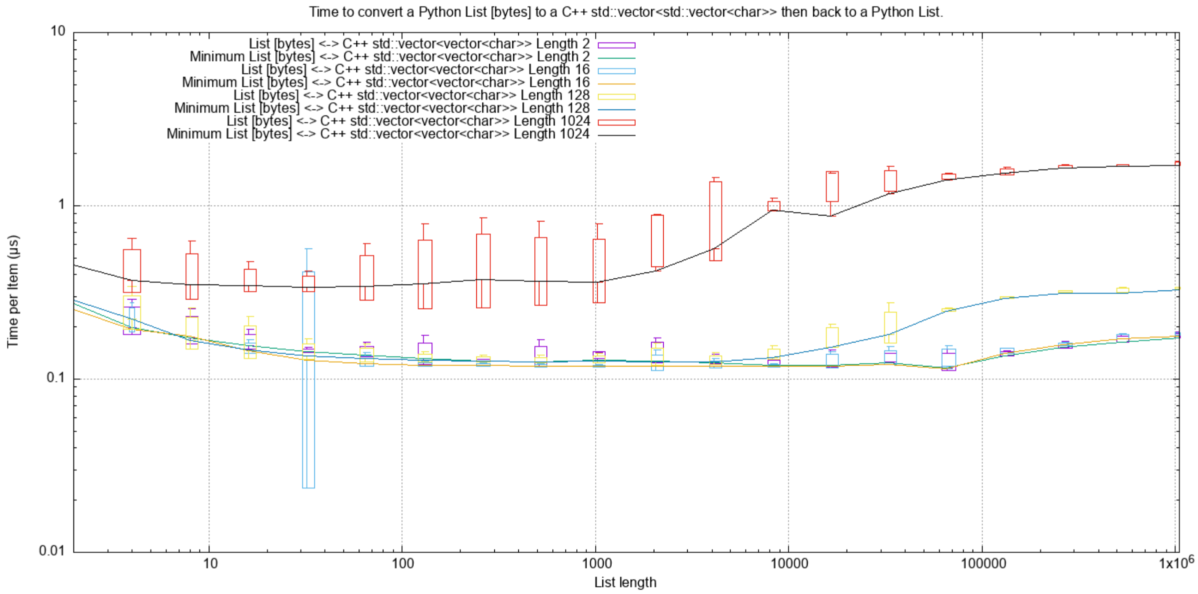


These are typically *round trip* converted at:

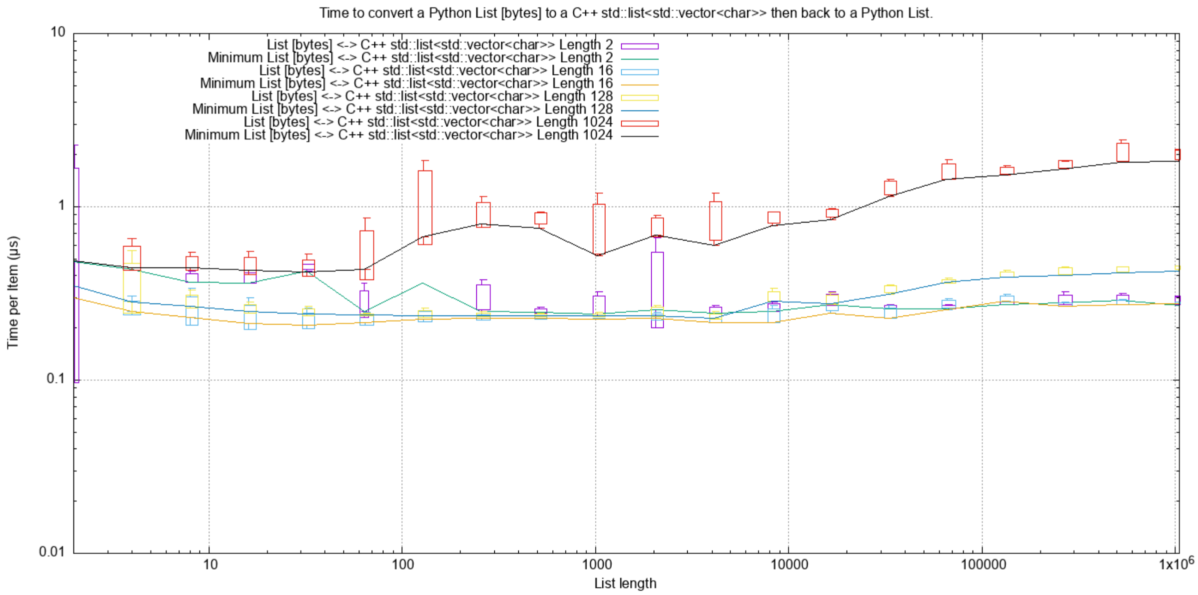
- 0.1 µs per object for booleans, say 100m objects a second. This is about 10x the cost of using a `std::vector`.

Python List of `bytes`

And a Python list of `bytes` for different lengths; 2, 16, 128 and 1024 bytes long via a C++ `std::vector`:



And a Python list of `bytes` for different lengths; 2, 16, 128 and 1024 bytes long via a C++ `std::list`:

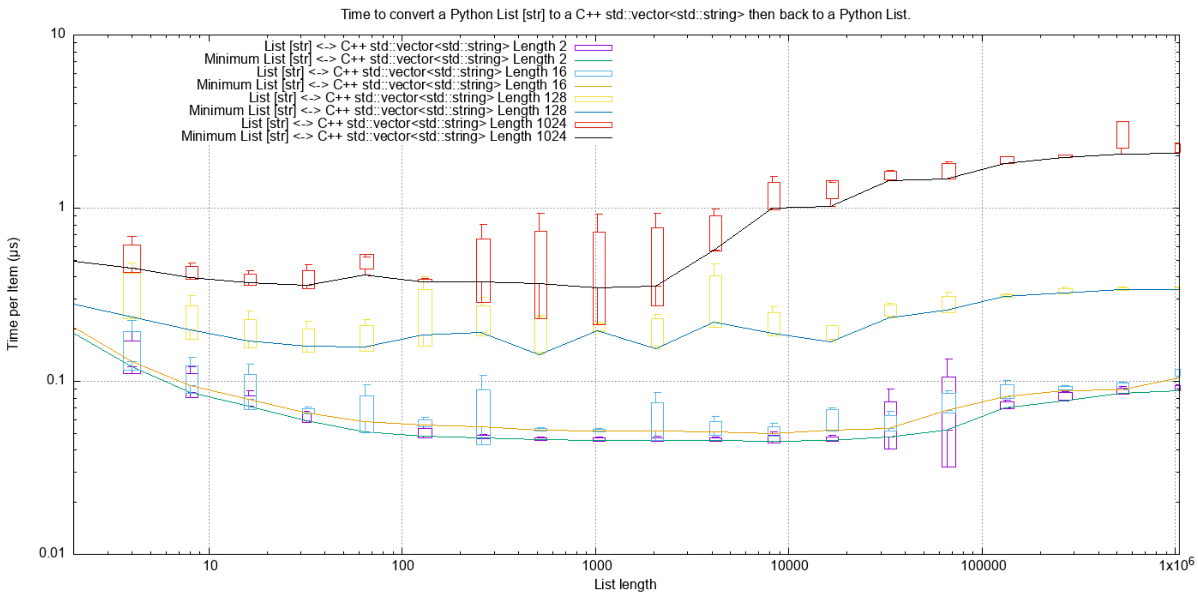


Given the size of each object this *round trip* time for lists can be summarised as:

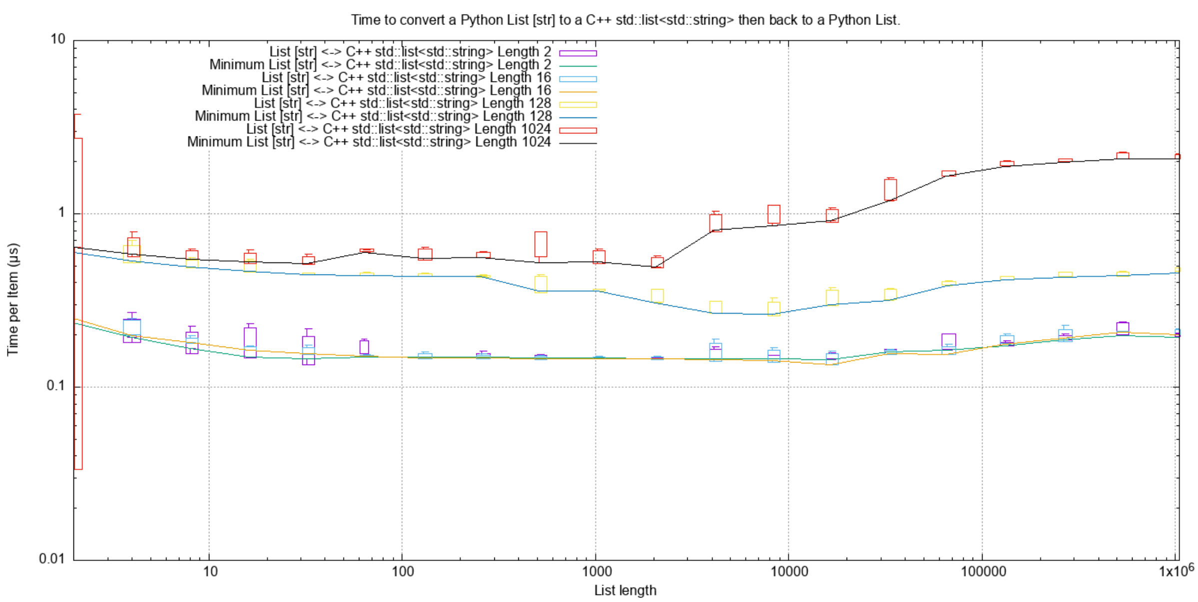
Object	Time per object (µs)	Rate (million/s)	Rate (Mb/s)	Notes
bytes[2]	0.1	10	20	
bytes[16]	0.1	10	160	
bytes[128]	0.1	10	1280	
bytes[1024]	0.4 to 2.0	0.5 to 2.5	500 to 2500	

Python List of `str`

And a Python list of `str` for different lengths; 2, 16, 128 and 1024 via a C++ `std::vector`:



And via a C++ `std::list`:



Given the size of each object this *round trip* time for lists can be summarised as:

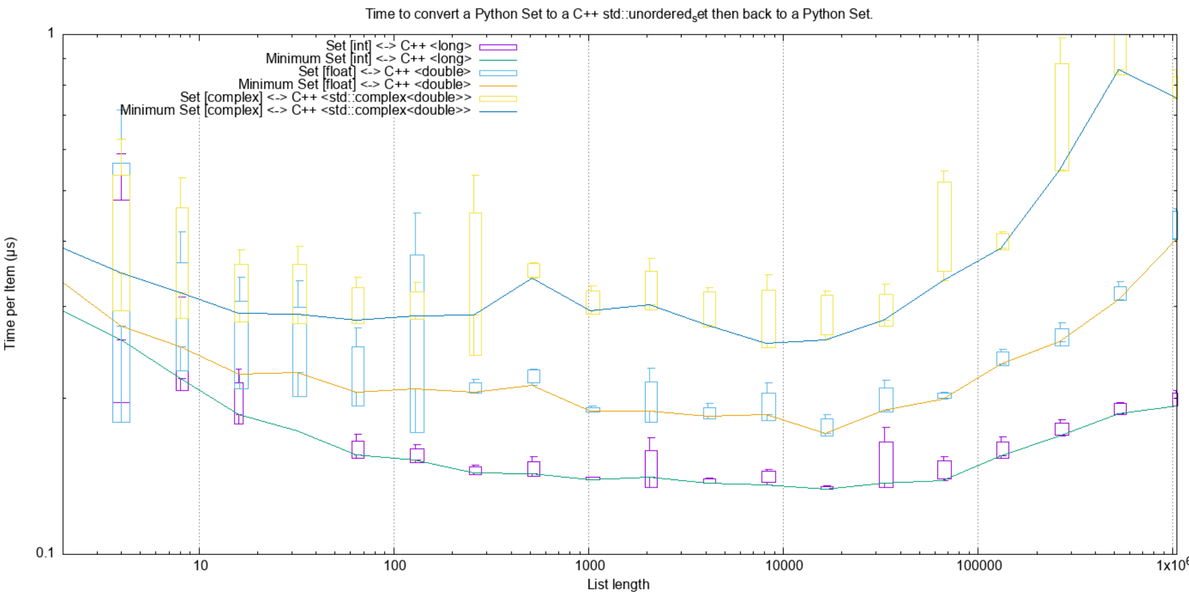
Object	Time per object (µs)	Rate (million/s)	Rate (Mb/s)	Notes
str[2]	0.05 to 0.1	10 to 20	20 to 40	
str[16]	0.05 to 0.1	10 to 20	160 to 320	
str[128]	0.2 to 0.4	2.5 to 5	320 to 640	
str[1024]	0.4 to 1.5	0.7 to 2.5	700 to 2500	

Lists of `str` has, essentially, the same performance as a list of `bytes`.

6.2.2 Python Sets

Python Set of `int`, `float` and `complex`

Here is the *round trip* performance of a Python set of `int`, `float` and `complex` numbers:



These are typically *round trip* converted at (for sets < 100,000 long):

- 0.15 µs per object for `int`, say 6m objects a second.
- 0.2 µs per object for `float`, say 5m objects a second.
- 0.3 µs per object for `complex`, say 3m objects a second.

The *round trip* time for a list takes 0.025 µs for `int`, `float` and `complex` so a set takes:

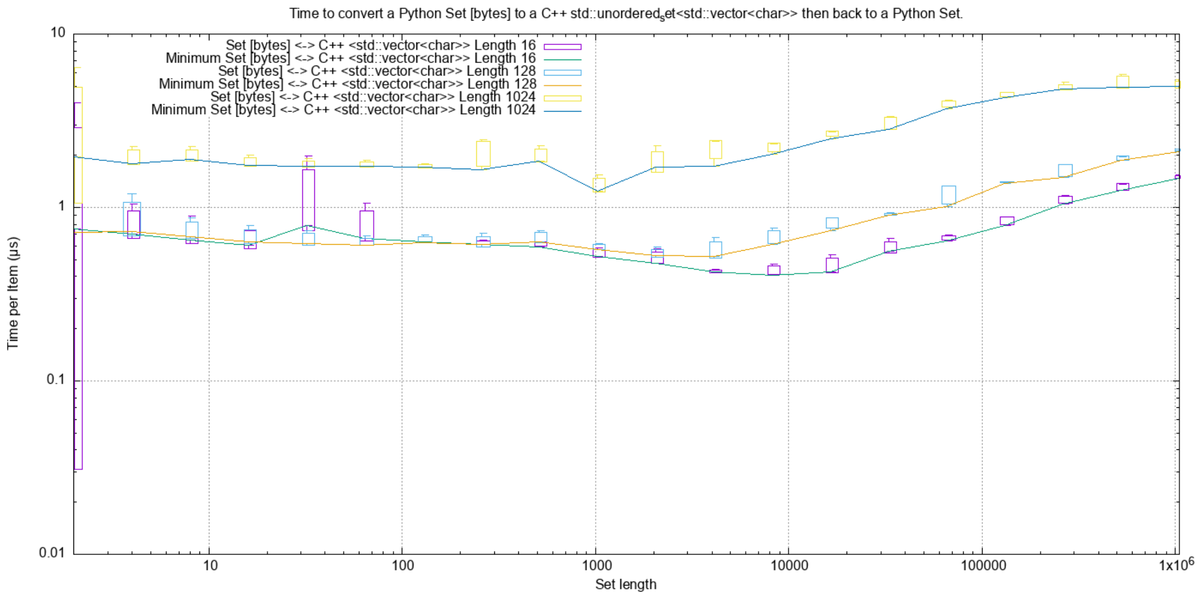
- 6x longer for an `int`
- 8x longer for a `float`.
- 12x longer for a `complex` number.

An explanation would be that the cost of hashing and insertion (and possible re-hashing the container) dominates the performance compared to the cost of object conversion.

The rise in rate towards larger sets also suggests that re-hashing becomes dominant with larger sets.

Python Set of `bytes`

And a Python set of bytes for different lengths; 16, 128 and 1024 bytes long:



Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
bytes[16]	~0.6	0.1	x6	
bytes[128]	0.6 to 1.5	0.1	x6 to x15	
bytes[1024]	1.0 to 5.0	0.4 to 2	x2.5	

Again, the cost of hashing and insertion explains the difference.

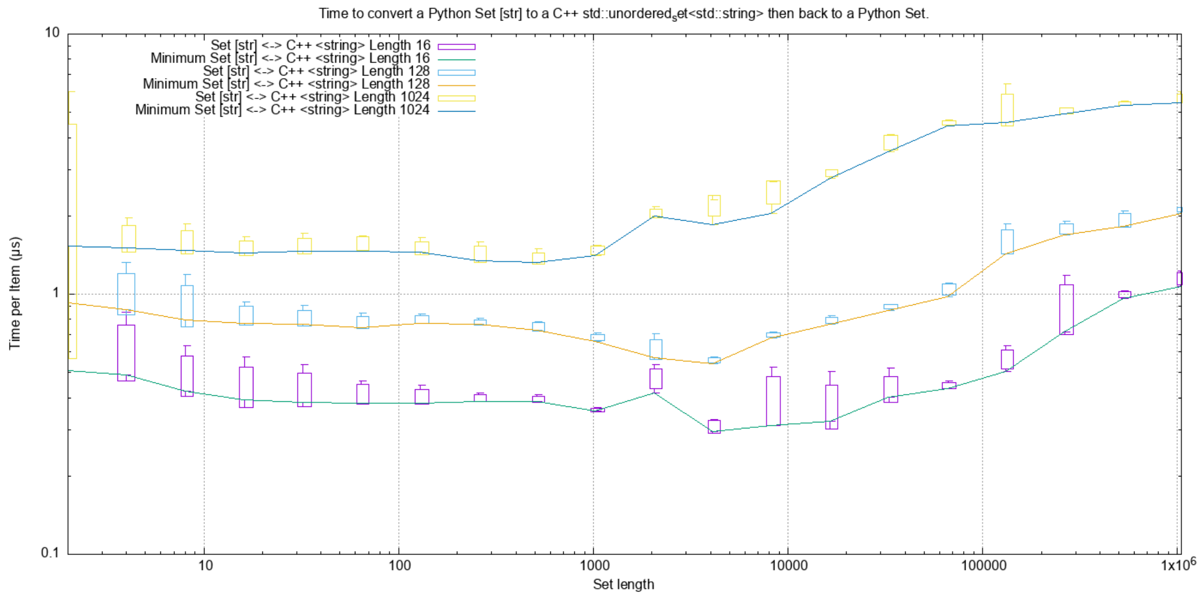
Given the size of each object this *round trip* time for sets can be summarised as:

Object	Time per object (µs)	Rate (million/s)	Rate (Mb/s)	Notes
bytes[16]	~0.6	1.7	27	
bytes[128]	0.6 to 1.5	0.7 to 1.7	90 to 220	
bytes[1024]	1.0 to 5.0	0.2 to 1	200 to 1000	

Python Set of `str`

TODO:

And a Python set of `str` for different lengths; 16, 128 and 1024 bytes long:



This is near identical with bytes with small strings having a slight edge.

Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
str[16]	0.3	0.05 to 0.1	x3 to x6	
str[128]	0.8	0.2 to 0.4	x2 to x4	
str[1024]	1.0 to 5.0	0.4 to 1.5	x1 to x10	

Again, the cost of hashing and insertion explains the difference.

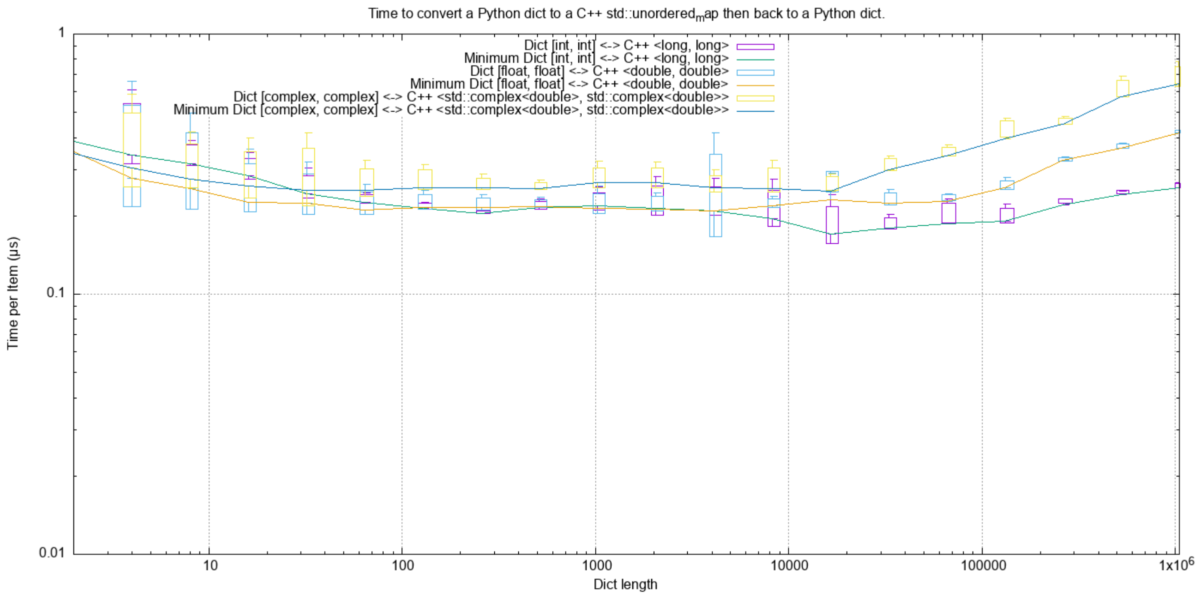
Given the size of each object this *round trip* time for sets can be summarised as:

Object	Time per object (µs)	Rate (million/s)	Rate (Mb/s)	Notes
bytes[16]	~0.6	1.7	27	
bytes[128]	0.6 to 1.5	0.7 to 1.7	90 to 220	
bytes[1024]	1.0 to 5.0	0.2 to 1	200 to 1000	

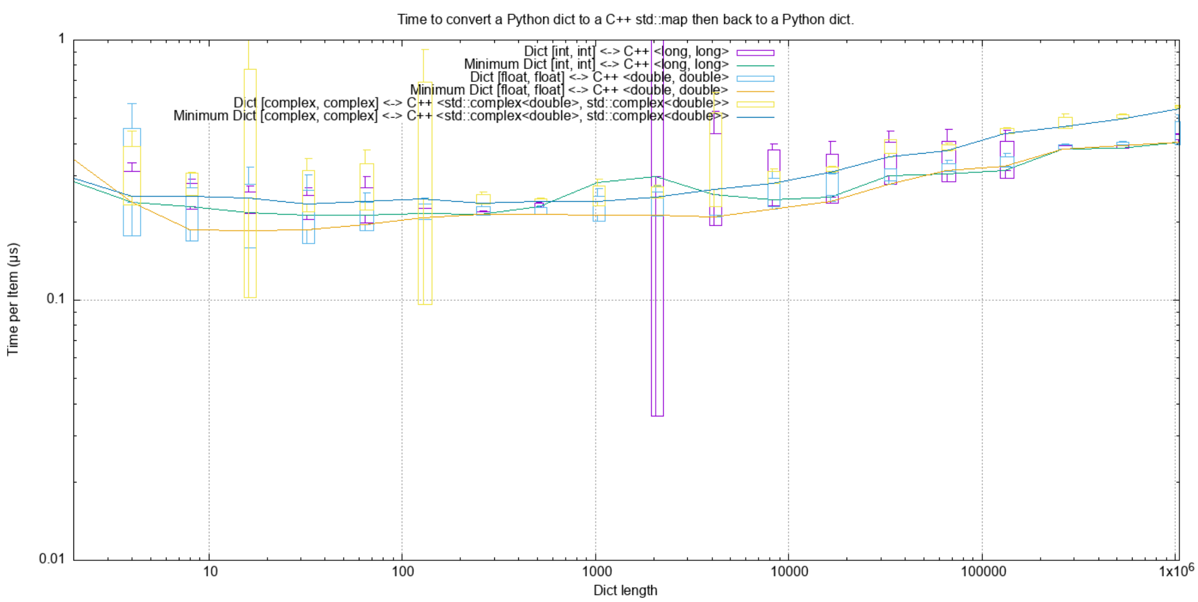
6.2.3 Python Dictionaries

Python Dict of `int`, `float` and `complex`

Here is the round trip time for a Python dict to and from a C++ `std::unordered_map<long, long>`. This plots the *round trip* cost *per key/value pair* against dict size.



And for conversion via a C++ `std::map`:



These are typically *round trip* converted at:

TODO:

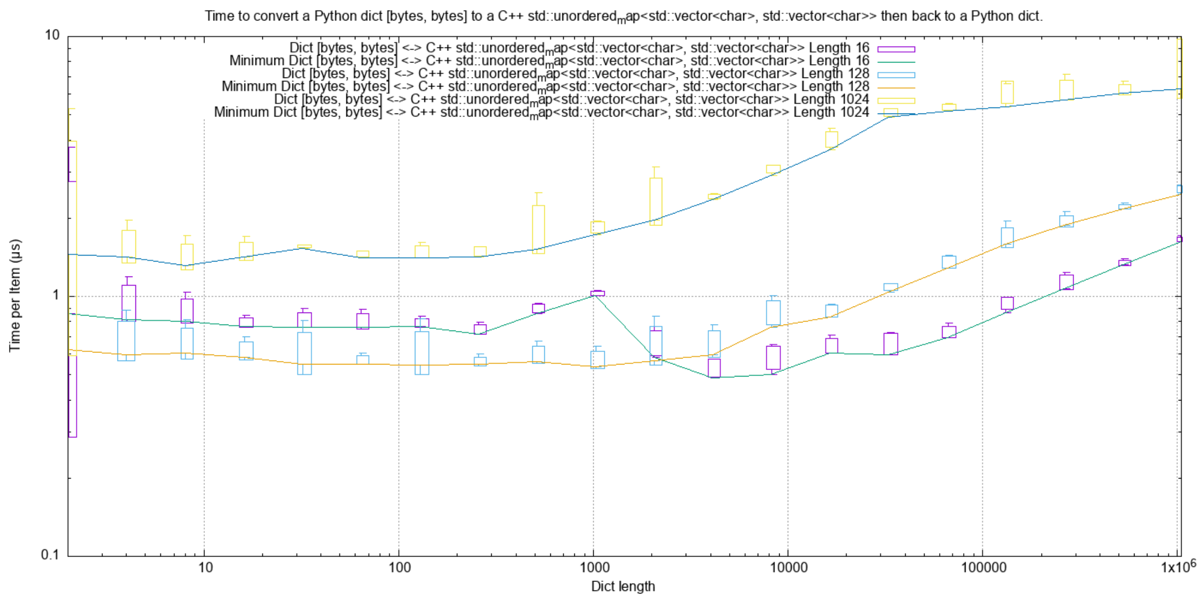
- 0.2 µs per object for an int or float, say fm objects a second.
- 0.25 µs per object for a complex number, say 4m objects a second.

This is identical to the values for the set but includes the conversion time for both key and value. The hashing, insertion and potential re-hashing dominate the performance.

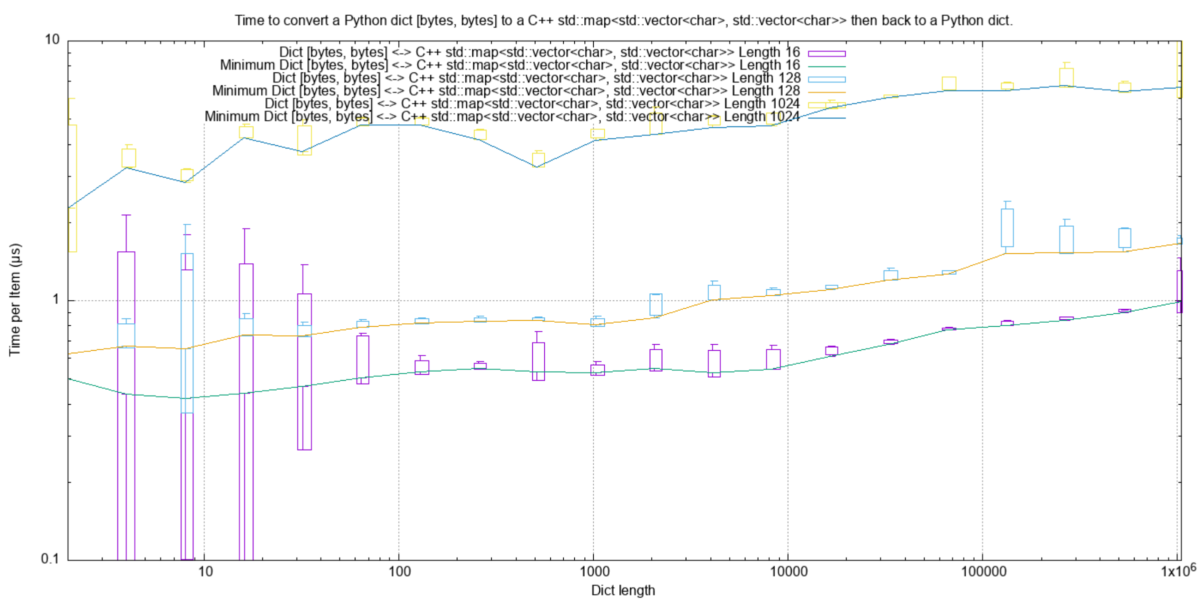
Python Dict of `bytes`

TODO:

Here is the *round trip* time for a Python dict [bytes, bytes] to and from a C++ `std::unordered_map<std::vector<char>, std::vector<char>>` for different lengths; 16, 128 and 1024 bytes long. The key and the value are the same length.



And via a C++ `std::map`:

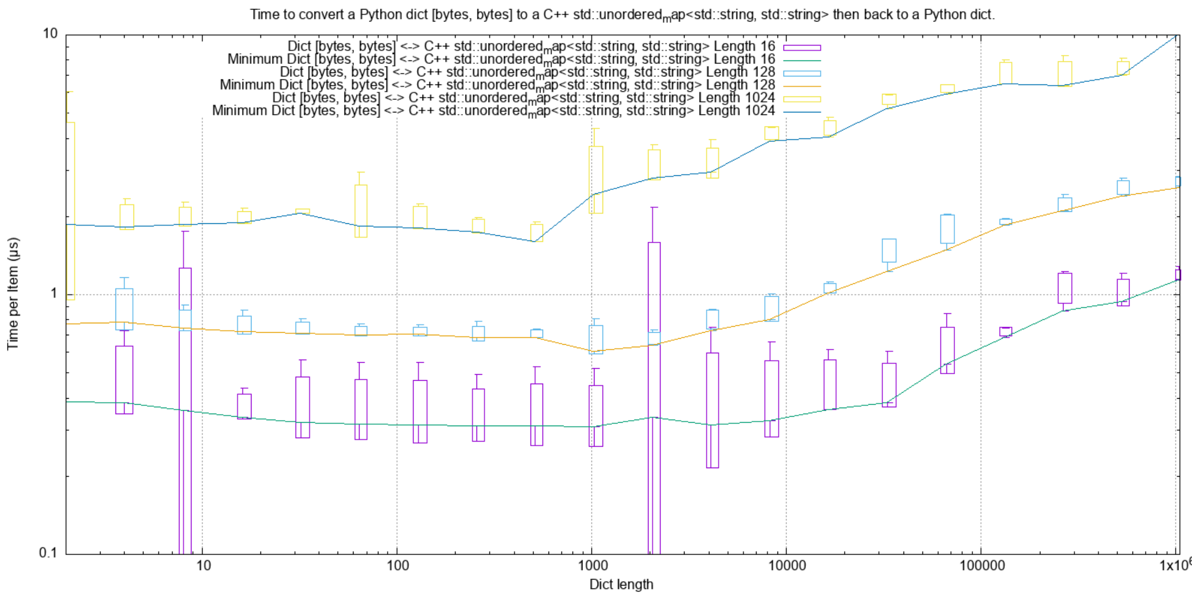


This *round trip* time for both keys and values for dicts can be summarised as:

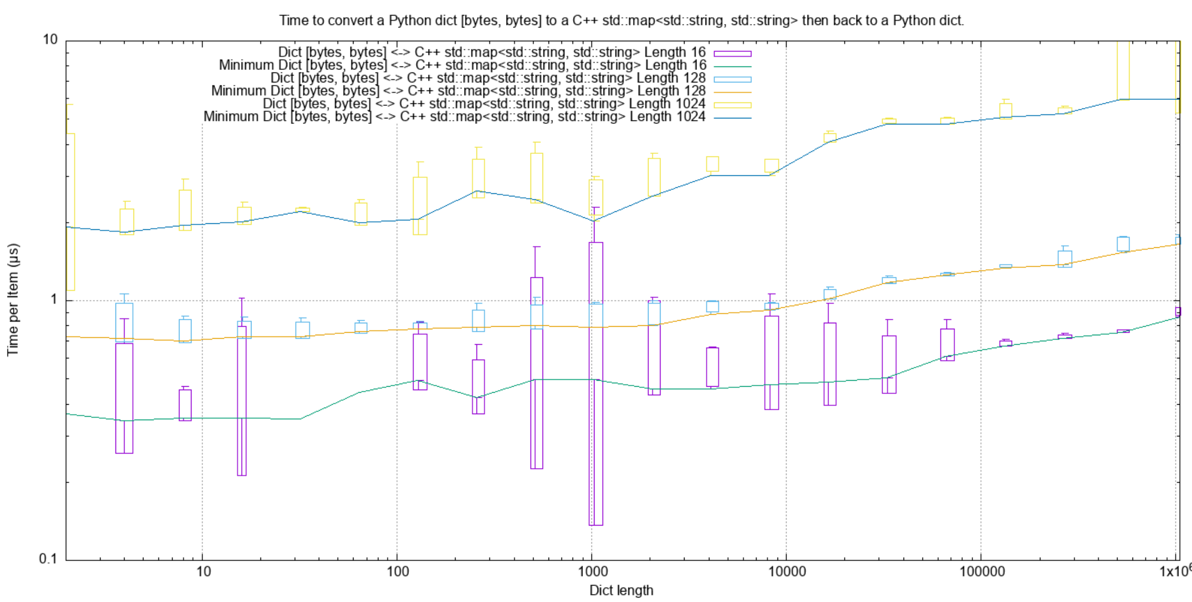
Object	Time per object (µs)	Rate (million/s)	Rate (Mb/s)	Notes
bytes[16]	0.5	2	32	
bytes[128]	0.6 to 2	0.5 to 1.5	64 to 256	
bytes[1024]	2 to 6	0.15 to 0.5	150 to 512	

Python Dict of `str`

Here is the *round trip* time for a Python dict [str, str] to and from a C++ `std::unordered_map<std::string, std::string>` for different lengths; 16, 128 and 1024 bytes long. The key and the value are the same length.



And via a C++ `std::map`:



This *round trip* time for both keys and values for dicts can be summarised as:

Object	Time per object (µs)	Rate (million/s)	Rate (Mb/s)	Notes
str[16]	0.4 to 1	1 to 2.5	16 to 48	
str[128]	0.6 to 2	0.5 to 1.7	64 to 220	
str[1024]	2 to 8	0.125 to 0.5	125 to 500	

6.2.4 Summary

The fairly simple summary is that the round trip performance, as measured by the Python interpreter, agrees very closely with the total cost Python -> C++ and C++ -> Python. In some cases the performance is twice that figure but no more.

6.3 Memory Use

To examine the typical memory use a round-trip was made between Python to C++ and back to Python with a container (`list`, `set` or `dict`) of `bytes`. The container was 1m long and each member was 1k bytes, so a total of 1Gb to convert to C++ and back to a new Python container.

The creation/destruction was repeated 10 times and the memory profiled using pymemtrace.

The code to do this for a `list` is something like:

```
from pymemtrace import cPyMemTrace

import cPyCppContainers

with cPyMemTrace.Profile():
    for _r in range(10):
        original = [b' ' * 1024 for _i in range(1024 * 1024)]
        new_list = cPyCppContainers.new_list_bytes(original)
```

pymemtrace produces a log file of memory usage such as (not the actual data that created the plot below):

```
      Event         dEvent  Clock         What     File                     #line Function  ␣
↪                                 RSS          dRSS
NEXT: 0             +0      1.267233      CALL     test_with_pymemtrace.py#   15 _test_new_
↪list_bytes                  29384704     29384704
PREV: 83            +83     1.267558      CALL     test_with_pymemtrace.py#   26 <listcomp>␣
↪                           29384704            0
NEXT: 84            +84     1.268744      RETURN   test_with_pymemtrace.py#   26 <listcomp>␣
↪                           29544448       159744
PREV: 87            +3      1.268755      C_CALL   test_with_pymemtrace.py#   28 new_list_
↪bytes                        29544448            0
NEXT: 88            +4      2.523796      C_RETURN test_with_pymemtrace.py#   28 new_list_
↪bytes                      1175990272   1146445824
NEXT: 89            +1      2.647460      C_CALL   test_with_pymemtrace.py#   29 perf_
↪counter                          34713600  -1141276672
PREV: 93            +4      2.647496      CALL     test_with_pymemtrace.py#   26 <listcomp>␣
↪                           34713600            0
NEXT: 94            +5      2.648859      RETURN   test_with_pymemtrace.py#   26 <listcomp>␣
↪                           34844672       131072
NEXT: 95            +1      2.648920      C_CALL   test_with_pymemtrace.py#   27 perf_
↪counter                          34775040       -69632
```

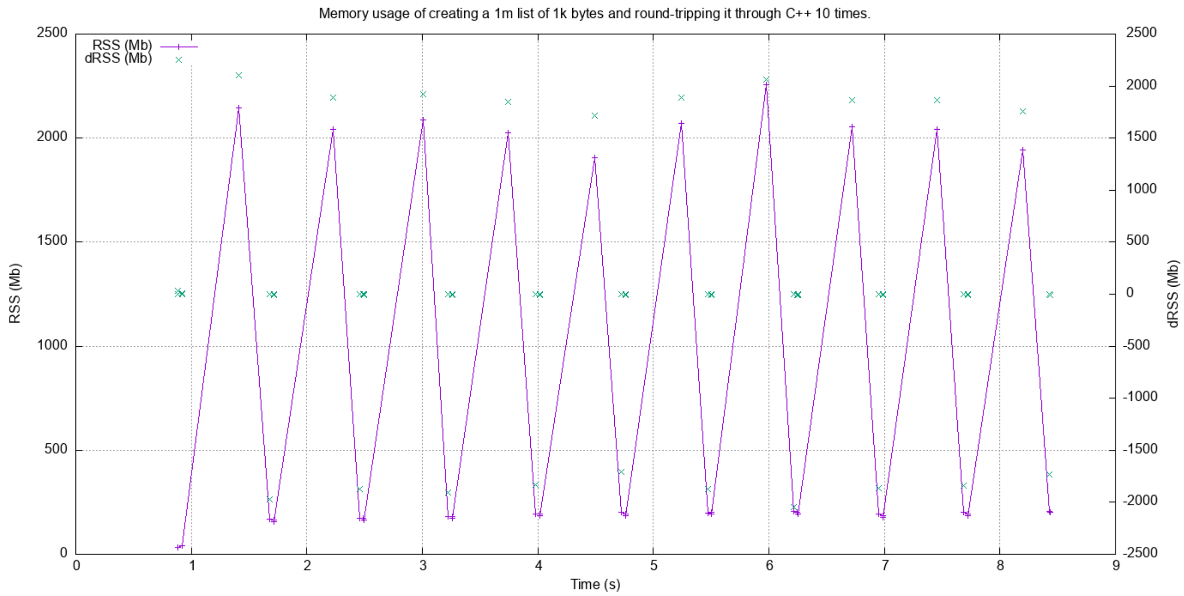
(continues on next page)

(continued from previous page)

```
PREV: 97           +2      2.648929     C_CALL   test_with_pymemtrace.py#  28 new_list_
↪bytes                        34775040          0
NEXT: 98           +3      3.906950     C_RETURN test_with_pymemtrace.py#  28 new_list_
↪bytes                      1176018944   1141243904
NEXT: 99           +1      4.041886     C_CALL   test_with_pymemtrace.py#  29 perf_
↪counter                          34713600  -1141305344
```

6.3.1 Python List of bytes

The following is a plot of RSS and change of RSS over time:



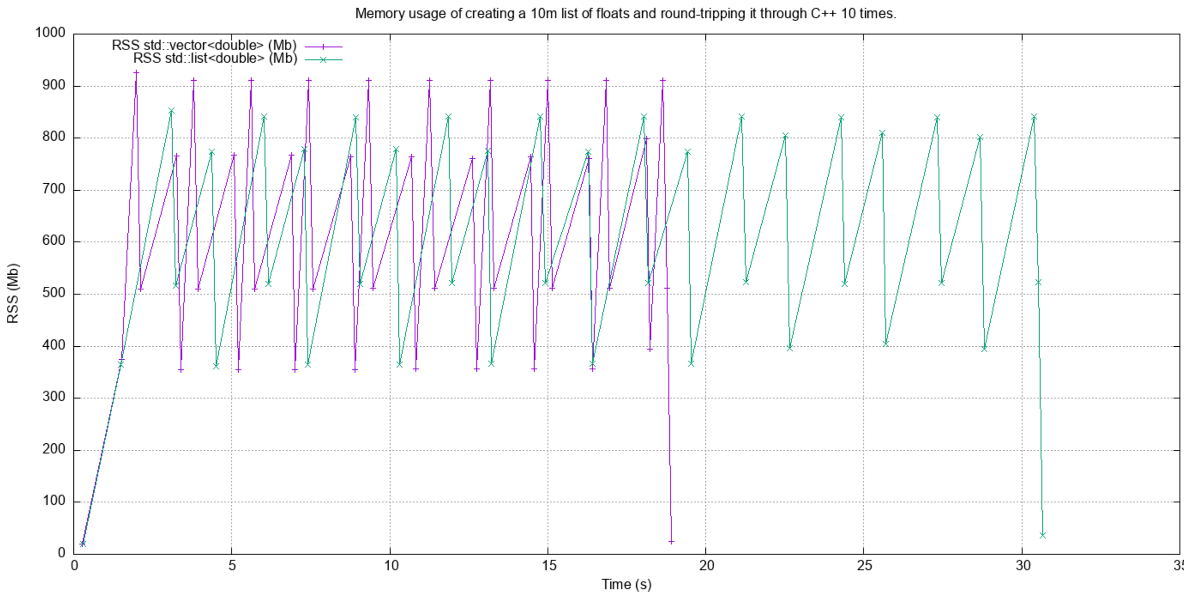
This result is rather surprising. The maximum RSS should reflect that at some point the following are held in memory:

- Basic Python, say 30Mb
- The original Python list of bytes, 1024Mb.
- The C++ `std::vector<std::string>`, 1024Mb.
- The new Python list of bytes, 1024Mb.

This would be a total of 3102Mb. However we are seeing a maximum RSS of only around 2200Mb.

6.3.2 Python List of floats

For comparison here is the time/memory plot of round-tripping a list of Python `float` as a C++ `std::vector` or `std::list`:



The memory usage is not significantly different but using a `std::list` takes about twice as long.

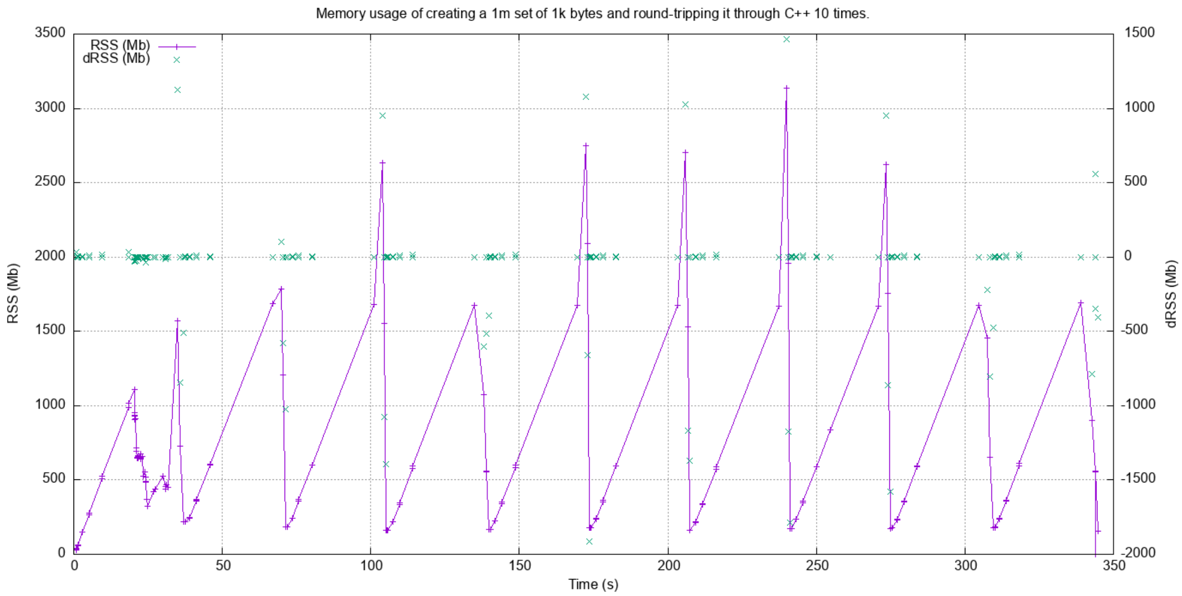
6.3.3 Python Set of bytes

A similar test was made of a gigabyte sized Python set of bytes. Each key and value were 1024 bytes long and the set was 1m long. The Python set was round-tripped to a C++ `std::unordered_set<std::string>` and back to a new Python set.

The code looks like this:

```
with cPyMemTrace.Profile(4096 * 16):
    total_bytes = 2**20 * 2**10
    byte_length = 1024
    set_length = total_bytes // byte_length // 2
    random_bytes = [random.randint(0, 255) for _i in range(byte_length)]
    for _r in range(10):
        original = set()
        for i in range(set_length):
            k = bytes(random_bytes)
            original.add(k)
            # Shuffle is quite expensive. Try something simpler:
            # chose a random value and increment it with roll over.
            index = random.randint(0, byte_length - 1)
            random_bytes[index] = (random_bytes[index] + 1) % 256
        cPyCppContainers.new_set_bytes(original)
```

The following is a plot of RSS and change of RSS over time:



In the set case constructing the original set takes around 1500Mb. So on entry to `new_set_bytes` the RSS is typically 1700Mb. Constructing the `std::unordered_set<std::string>` and a new Python set takes an extra 1000Mb taking the total memory to around 2500MB. On exit from `new_set_bytes` the RSS decreases back down to 200Mb.

In theory the maximum RSS use should be:

- Basic Python, say 30Mb
- The original Python set, 1024Mb.
- The C++ `std::unordered_set<std::string>`, 1024Mb.
- The new Python dict, 1024Mb.

This would be a total of 3102Mb.

6.3.4 Python Dictionary of `bytes` or `str`

A similar test was made of a gigabyte sized Python dict of bytes. Each key and value were 1024 bytes long and the dictionary was 0.5m long. The Python dict was round-tripped to a C++ `std::unordered_map<std::vector<char>, std::vector<char>>` and back to a new Python dict.

The code looks like this:

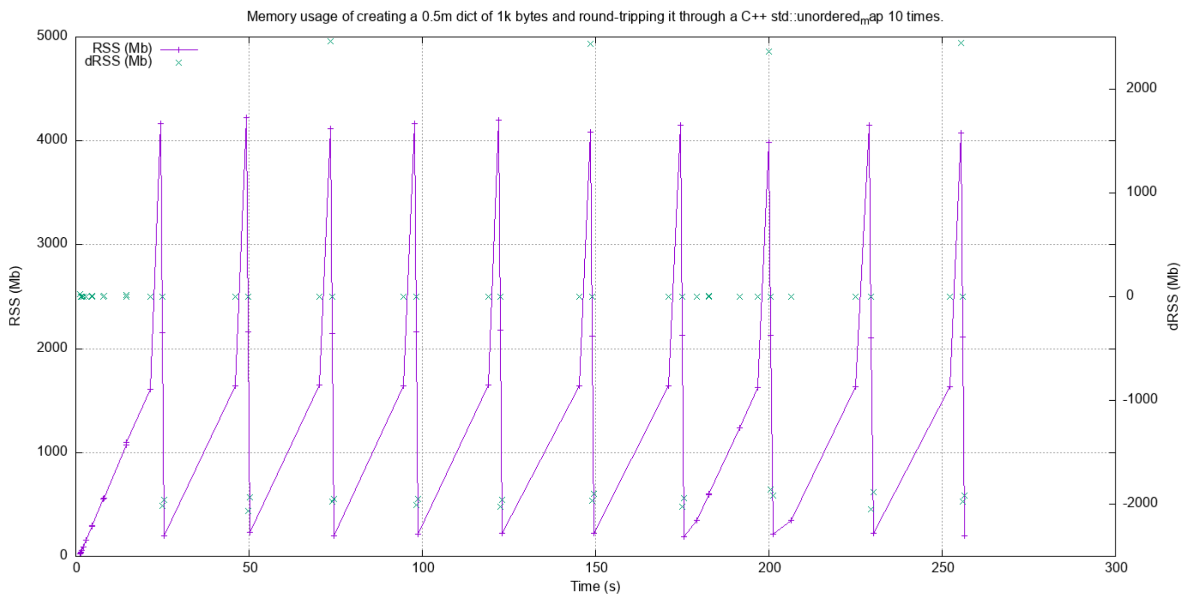
```
with cPyMemTrace.Profile(4096 * 16):
    total_bytes = 2**20 * 2**10
    byte_length = 1024
    dict_length = total_bytes // byte_length // 2
    random_bytes = [random.randint(0, 255) for _i in range(byte_length)]
    for _r in range(10):
        original = {}
        for i in range(dict_length):
            k = bytes(random_bytes)
            original[k] = b' ' * byte_length
            # Shuffle is quite expensive. Try something simpler:
            # chose a random value and increment it with roll over.
```

(continues on next page)

(continued from previous page)

```
            index = random.randint(0, byte_length - 1)
            random_bytes[index] = (random_bytes[index] + 1) % 256
        cPyCppContainers.new_dict_bytes_bytes(original)
```

The following is a plot of RSS and change of RSS over time:



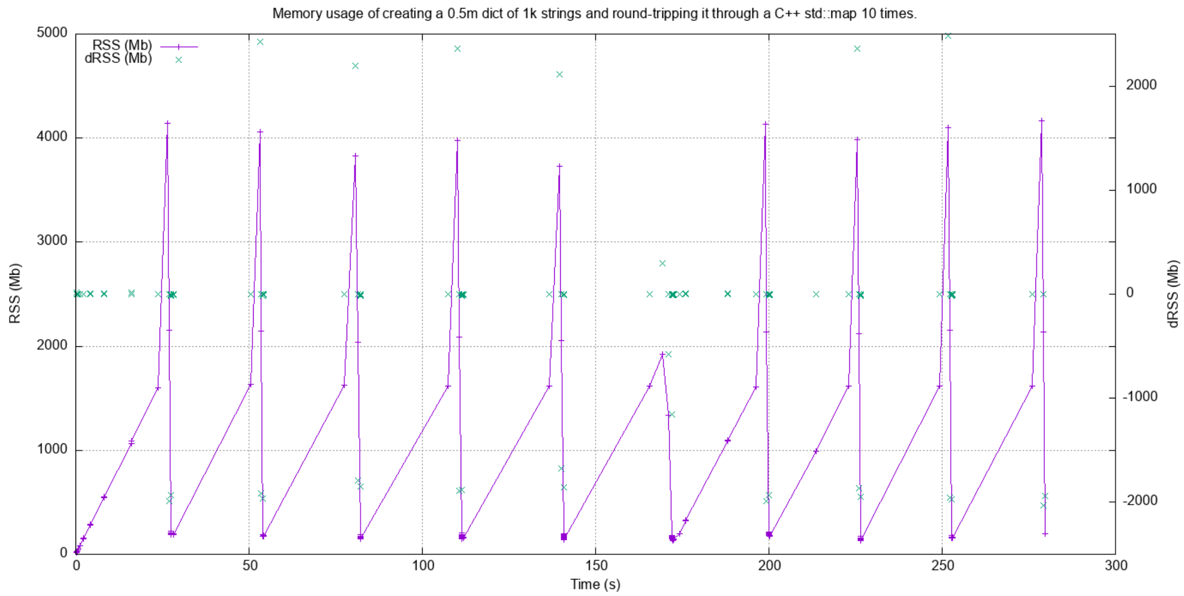
In the dictionary case constructing the original dict takes around 1500Mb. So on entry to `new_dict_bytes_bytes` the RSS is typically 1700Mb. Constructing the `std::unordered_map<std::vector<char>, std::vector<char>>` and a new Python dict takes an extra 2500Mb taking the total memory to around 4200MB. On exit from `new_dict_bytes_bytes` the RSS decreases in two stages, destroying the `std::unordered_map<std::string, std::string>` frees 2000Mb then freeing the original gives back another 2000Mb. This brings the total RSS back down to 200Mb.

In theory the maximum RSS use should be:

- Basic Python, say 30Mb
- The original Python dict, 1024Mb.
- The C++ `std::unordered_map<std::vector<char>, std::vector<char>>`, 1024Mb.
- The new Python dict, 1024Mb.

This would be a total of 3102Mb. The fact that we are seeing around 4200Mb, 35% more, is probably due to over-allocation either any or all of the Python dict or bytes allocators or the C++ `std::unordered_map<T>` or `std::vector<char>` allocators.

Similar results are obtained for a Python dict was round-tripped to a C++ `std::map<std::string, std::string>` and back to a new Python dict.



This is broadly similar to the results for `std::unordered_map<std::vector<char>, std::vector<char>>`.

All these graphs show that there are no memory leaks.

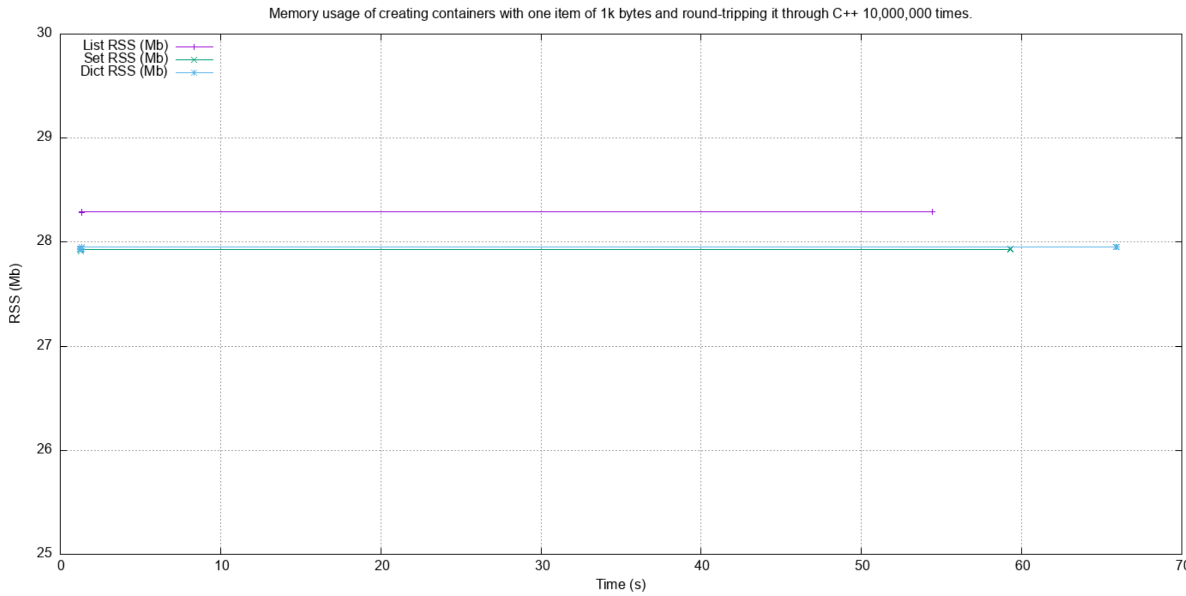
6.3.5 Containers of Just One Object

This test was to create a list, set or dict with one entry of 1024 bytes and then convert it 10,000,000 times to a C++ container and then back to Python. The memory was monitiored with pymemtrace set up to spot and changes in RSS of >=4096 bytes.

For example here is the code for a list:

```
original = [b' ' * 1024]
with cPyMemTrace.Profile():
    for _r in range(10_000_000):
        cPyCppContainers.new_list_bytes(original)
    # Tends to force an event in pymemtrace.
    gc.collect()
```

The following is a plot of RSS and change of RSS over time for list, set, dict:



This graph shows that there are no memory leaks on container construction.

6.4 Summary

- Fundamental types (`bool`, `int`, `float`, `complex`) can be converted at around 100m objects/sec.
- Sequences of bytes or strings are converted at a memory rate of around 4,000 Mb/sec.
- Dicts and sets are about 3-10x slower than lists and tuples. This can be explained by, whilst both list and dict operations are O(1), the list insert is much faster as an insert into a dict/set involves hashing.
- In some cases the performance of converting Python to C++ or the reverse is faster but the difference is <= 2x.
- There are no memory leaks.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search