Python and C++ Containers

Release 0.4.0

Paul Ross

Oct 18, 2024

CONTENTS:

1 Introduction **1**

1.1 A Problematic Example . . . 2

1.2 Why This Project . . . 3

1.3 Using This Library . . . 4

1.3.1 Python to C++ . . . 4

1.3.2 C++ to Python . . . 5

1.4 The Hand Written Functions . . . 6

1.4.1 Converting a C++ `std::vector<T>` or `std::list<T>` to a Python `tuple` or `list` . . . 6

1.4.2 C++ to Python Implementation . . . 7

1.4.3 Partial Specialisation to Convert a C++ `std::vector<T>` or `std::list<T>` to a Python `list`` . . . 8

1.4.4 Converting a Python `tuple` or `list` to a C++ `std::vector<T>` or `std::list<T>` . . . 9

1.4.5 Python to C++ Implementation . . . 10

1.4.6 Partial Specialisation to Convert a Python `list` to a C++ `std::vector<T>` or `std::list<T>` 11

1.5 Generated Functions . . . 13

1.5.1 C++ to Python . . . 13

1.5.2 Python to C++ . . . 14

2 Using this C++ Library **15**

2.1 The Basics . . . 15

2.1.1 Get the Project from `git` . . . 15

2.1.2 Code Generation . . . 15

2.1.3 Build Configuration . . . 15

2.1.4 Source Inclusion . . . 16

2.1.5 Errors . . . 16

2.2 Examples . . . 17

2.2.1 Using C++ to Double the Values in a Python List of `float` . . . 17

2.2.2 Reversing a `tuple` of `bytes` in C++ . . . 18

2.2.3 Incrementing `dict` values in C++ . . . 19

3 Building and Testing **21**

3.1 Building and Testing C++ Code . . . 21

3.1.1 Code Generation . . . 21

3.1.2 Building C++ Code . . . 21

3.1.3 Testing C++ Code . . . 22

3.2 Building and Testing Python Code . . . 23

3.2.1 Building Python Code . . . 23

3.2.2 Testing Python Code . . . 23

3.3 Building the Documentation . . . 25

3.3.1 `gnuplot` Plots . . . 25

3.3.2 Sphinx . . . 26
3.3.3 Doxygen . . . 26
3.4 Building and Testing Everything for Multiple Python Versions . . . 26

4 C++ API **29**
4.1 Include File and Namespace . . . 29
4.2 Python Containers to C++ . . . 29
4.2.1 Error Indication . . . 29
4.2.2 Python `tuple` to `std::vector` or `std::list` . . . 29
4.2.3 Python `list` to `std::vector` or `std::list` . . . 30
4.2.4 Python `set` to `std::unordered_set` . . . 31
4.2.5 Python `frozenset` to `std::unordered_set` . . . 32
4.2.6 Python `dict` to `std::unordered_map` or `std::map` . . . 32
4.3 C++ Containers to Python . . . 33
4.3.1 Error Indication . . . 33
4.3.2 C++ `std::vector` or `std::list` to Python `tuple` . . . 33
4.3.3 C++ `std::vector` or `std::list` to Python `list` . . . 34
4.3.4 C++ `std::unordered_set` to Python `set` . . . 35
4.3.5 C++ `std::unordered_set` to Python `frozenset` . . . 35
4.3.6 C++ `std::unordered_map` or `std::map` to a Python `dict` . . . 36

5 Design **37**
5.1 Object Conversion Source Files . . . 37
5.2 Container Conversion Source Files . . . 38
5.3 `python_convert.h` . . . 39
5.4 Python `list` and `tuple` . . . 39
5.4.1 Conversion From C++ to Python . . . 39
5.4.2 Conversion From Python to C++ . . . 43
5.5 Python `set` and `frozenset` . . . 46
5.5.1 Conversion From C++ to Python . . . 46
5.5.2 Conversion From Python to C++ . . . 47
5.6 Python `dict` . . . 49
5.6.1 Conversion From C++ to Python . . . 49
5.6.2 Conversion From Python to C++ . . . 51
5.7 Code Generation . . . 52

6 User Defined Types **55**
6.1 User Defined Types in a C Extension . . . 55
6.1.1 A Python Class . . . 55
6.1.2 The C++ Class . . . 57
6.1.3 Checking the Python Type . . . 57
6.1.4 Conversion Code . . . 58
6.1.5 Template Specialisation . . . 59
6.2 Using the C++ Conversion Functions . . . 61
6.2.1 From C++ to Python . . . 61
6.2.2 From Python to C++ . . . 61
6.2.3 Example of Round-trip Conversion . . . 61
6.2.4 Supporting `dict[int, cUserDefined.Custom]` . . . 62
6.3 User Defined Types From Pure Python Types . . . 65
6.4 Interoperation with `numpy` ND Arrays . . . 65

7 Performance **67**
7.1 Summary . . . 67
7.2 C++ Performance Tests . . . 67
7.2.1 Test Procedure . . . 67

7.2.2 Fundamental Types 70
7.2.3 Python List to and from a C++ `std::vector<T>` 76
7.2.4 Python Tuple to and from a C++ `std::vector<T>` 86
7.2.5 Python Set to and from a C++ `std::unordered_set<T>` 86
7.2.6 Python Dict to and from a C++ `std::unordered_map<K, V>` 94
7.2.7 Summary 94
7.3 Round-trip Python to C++ and back to Python 95
7.3.1 Python Objects 95
7.3.2 Python Containers Code 101
7.3.3 Python Lists 102
7.3.4 Python Sets 106
7.3.5 Python Dictionaries 109
7.3.6 Summary 116
7.4 Memory Use 116
7.4.1 Single Bytes and Strings 116
7.4.2 Containers 119

8 History **127**
8.1 Version 0.4.0 127
8.2 Version 0.3.2 127
8.3 Version 0.3.1 127
8.4 Version 0.3.0 127
8.5 Version 0.2.0 127
8.6 Version 0.1.0 128

9 TODO **129**

10 Indices and tables **131**

CHAPTER

ONE

INTRODUCTION

Python is well known for it's ability to handle *heterogeneous* data in containers such as lists like:

```
>>> l = [1, 2.0, "some string", ]
```

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 such as `list`, `tuple`, `dict`, `set`, `frozenset` containing homogeneous types such as `bool`, `int`, `float`, `complex`, `bytes`, `str` or user defined types to and from their C++ equivalent.

Here is a general example of the use of this library where Python data needs to be passed to and from a C++ library and those results need to be presented in Python. Like this, visually:

```
      Python        |   This Library (C++/Python)   |  Some C++ Library
------------------- . ----------------------------- . ------------------
        |           .                               .
 Get Python data    .                               .
        |           .                               .
        \---------------------->\                   .
                    .           |                   .
                    .  Convert Python data to C++   .
                    .           |                   .
                    .           \---------------------------->\
                    .                               .         |
                    .                               .  Process C++ data
                    .                               .         |
                    .           /<----------------------------/
                    .           |                   .
                    .  Convert C++ data to Python   .
                    .           |                   .
        /<----------------------/                    .
        |           .                               .
Process Python data .                               .
        |           .                               .
```

Here is a, problematic, example of how to do this:

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 C++ 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>` with Pythons C-API:

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

There is no error handling shown here, and 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; Python to C++ and C++ to Python. It gets worse when you want to support other containers such as `tuple`, `list`, `set`, `frozenset`, `dict`. You end up with hundreds of functions, all individually named, to handle all the combinations. Then you have to write individual conversion functions, and their tests, for all the combinations of object types *and* containers.

This is tedious and error prone and hard to extend in the general case.

1.2 Why This Project

This project simplifies the problem of converting data from Python to C++ and vice versa *in general*.

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

The types and containers this library supports are:

Table 1: **Supported Object types.**

C++ Type	Python Type	Notes
`bool`	`True`, `False`	
`long`	`int`	
`double`	`float`	
`std::complex<double>`	`complex`	
`std::vector<char>`	`bytes`	`bytearray` is not supported as we need hashable types for `set` and `dict` containers.
`std::string`	`str`	Specifically a `PyUnicode_1BYTE_KIND`[1]. Python documentation
`std::u16string`	`str`	Specifically a `PyUnicode_2BYTE_KIND`. Python documentation
`std::u32string`	`str`	Specifically a `PyUnicode_4BYTE_KIND`. Python documentation

Used in these containers:

Table 2: **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`

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 352 conversion functions to be written, tested and documented[2] .

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 352 cases.

[1] We are currently targeting C++14 so we use `std::string` which is defined as `std::basic_string<char>`. C++20 allows a stricter, and more desirable, definition `std::basic_string<char8_t>` that we could use here. See C++ reference for std::string

[2] 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 eight types (`bool`, `int`, `float`, `complex`, `bytes`, `str[1]`, `str[2]`, `str[4]`). Each container/type combination requires two functions to give two way conversion from Python to C++ and back. Thus 6 (container pairings) * 8 (types) * 2 (way conversion) = 96 required functions. For `dict` there are two container pairings (`dict` <-> `std::map`, `dict` <-> `std::unordered_map`) with the eight types either of which can be the key or the value so 64 (8**2) possible variations. Thus 2 (container pairings) * 64 (type pairs) * 2 (way conversion) = 256 required functions. Thus is a total of 96 + 256 = 352 functions.

1.3 Using This Library

1.3.1 Python to C++

Using the library is as simple as this, suppose you have data in Python that needs to be passed to a C++ library:

```
      Python         |   This Library (C++/Python)   |  Some C++ Library
-------------------- . ----------------------------- . ------------------
        |            .                               .
Python data source   .                               .
        |            .                               .
        \----------------------->\                   .
                     .           |                   .
                     .  Convert Python data to C++   .
                     .           |                   .
                     .           \------------------------------->\
                     .                               .            |
                     .                               .    Process C++ data
```

The C++ code using this library looks like this:

C++ Code

```
#include "python_convert.h"

// Create a Python list of floats: [21.0, 42.0, 3.0]
PyObject *op = Py_BuildValue("[ddd]", 21.0, 42.0, 3.0);

// Create the C++ vector that we want to convert this data to...
std::vector<double> cpp_vector;

// The template specialisation will automatically invoke the appropriate
// function call.
// It will be a compile time error if the container/type function
// is not supported.
// 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 = Python_Cpp_Containers::py_list_to_cpp_std_list_like(op, cpp_vector);
// Handle error checking if err is non-zero...
```

Note: If you were to change the C++ container to a `std::list<double>` the function call `py_list_to_cpp_std_list_like()` would be the same. Of course `py_list_to_cpp_std_list_like()` would then dispatch to code handling a `std::list<double>`.

Another example, suppose the Python data source is a `typing.Dict[int, str]` and this needs to be converted to a C++ `std::map<long, std::string>>` then a function using the conversion code using this library is as simple as this:

```
#include "python_convert.h"

void convert_py_data_to_cpp(PyObject *arg) {
    std::unordered_map<long, std::string> map;
    if (Python_Cpp_Containers::py_dict_to_cpp_std_map_like(arg, map)) {
        // Handle error...
    } else {
        // Use the map...
    }
}
```

1.3.2 C++ to Python

Suppose that you have data from a C++ library and this data needs to be represented in Python:

```
      Python        |   This Library (C++/Python)   |  Some C++ Library
------------------- . ----------------------------- . ------------------
                    .                               .  C++ data source
                    .                               .        |
                    .           /<---------------------------/
                    .           |                   .
                    .  Convert C++ data to Python   .
                    .           |                   .
        /<----------------------/                    .
        |           .                               .
    Python data     .                               .
        |           .                               .
```

The C++ code using this library looks like this:

```
#include "python_convert.h"

std::vector<double> cpp_vector;
// Populate the C++ vector...
cpp_vector.push_back(21.0);
cpp_vector.push_back(42.0);
cpp_vector.push_back(3.0);

// Now convert to Python.
// This will be a compile time error if the C++ type is not supported.
PyObject *op  = Python_Cpp_Containers::cpp_std_list_like_to_py_list(cpp_vector);
// op is a Python list of floats: [21.0, 42.0, 3.0]
// op will be null on failure and a Python exception will have been set.
```

Note: If you were to change the C++ container to a `std::list<double>` the function call `cpp_std_list_like_to_py_list()` would be the same. Of course `cpp_std_list_like_to_py_list()` would then dispatch to code handling a `std::list<double>`.

Another example, suppose the C++ data source is a `std::map<long, std::string>>` and we need this a Python dict `typing.Dict[int, str]` then the conversion code in this library is as simple as this:

```
#include "python_convert.h"

PyObject *convert_cpp_data_to_py() {
    std::map<long, std::string> map;
    // Populate map from the C++ data source
    // ...
    // Now convert to a Python dict:
    return Python_Cpp_Containers::cpp_std_map_like_to_py_dict(map);
}
```

1.4 The Hand Written Functions

At the heart off this library here are only six non-trivial hand written functions along with a much larger of generated functions that successively specialise these handwritten functions. They are defined as templates in `src/cpy/python_object_convert.h`.

- 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 pairs.

These six handwritten templates are short, fairly simple and comprehensible. Then, for simplicity, a Python script is used to create the final, instantiated, 352 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.

1.4.1 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 code, see "C++ to Python Implementation" below.
    // ...
}
```

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`. The function signature is `PyObject *ConvertCppToPy(const T&)`. This returns NULL on failure.
`PyUnaryContainer_New`	A pointer to a function that creates a new Python container, for example a `list`, of a particular length. The function signature is `PyObject *PyUnaryContainer_New(Py_ssize_t)`. This returns NULL on failure.
`PyUnaryContainer_Set`	A pointer to a function that sets a `PyObject *` in the Python container at a given index. The function signature is `int PyUnaryContainer_Set(PyObject *container, size_t pos, PyObject *value))`. This returns 0 on success.

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.

1.4.2 C++ to Python Implementation

The implementation is fairly straightforward in `src/cpy/python_object_convert.h` (lightly edited):

```
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) {
    assert(!PyErr_Occurred());
    PyObject *ret = PyUnaryContainer_New(list_like.size());
    if (ret) {
        size_t i = 0;
        for (const auto &val: list_like) {
            PyObject *op = (*ConvertCppToPy)(val);
            if (!op) {
                // Failure, do not need to decref the contents as that will
```

(continues on next page)

(continued from previous page)

```
                // be done when decref'ing the container.
                // e.g. tupledealloc():
                // https://github.com/python/cpython/blob/main/Objects/tupleobject.c
                PyErr_Format(PyExc_ValueError, "C++ value of can not be converted.");
                goto except;
            }
            // PyUnaryContainer_Set usually wraps a void function, always succeeds
            // returning non-zero.
            if (PyUnaryContainer_Set(ret, i++, op)) { // Stolen reference.
                PyErr_Format(PyExc_RuntimeError, "Can not set unary value.");
                goto except;
            }
        }
    } else {
        PyErr_Format(
            PyExc_ValueError,
            "Can not create Python container of size %ld",
            list_like.size()
        );
        goto except;
    }
    assert(!PyErr_Occurred());
    assert(ret);
    goto finally;
except:
    Py_XDECREF(ret);
    assert(PyErr_Occurred());
    ret = NULL;
finally:
    return ret;
}
```

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

As an example this is specialised for a C++ `std::vector` and a 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 a Python tuple:

```
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_tuple_new, &py_tuple_set
    >(container);
}
```

1.4.4 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 *),
        PyObject *(*PyUnaryContainer_Get)(PyObject *, size_t)>
int very_generic_py_unary_to_cpp_std_list_like(
    PyObject *op, ListLike<T> &list_like
) {
    // Handwritten code, see "Python to C++ Implementation" below.
    // ...
}
```

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. The function signature is `int PyObject_Check(PyObject *)`. This returns non-zero if the Python object is as expected.
`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>`. The function signature is `T PyObject_Convert(PyObject *)`.
`PyUnaryContainer_Check`	A pointer to a function that checks that the `PyObject *` argument is the correct container type, for example a `tuple`. The function signature is `int PyUnaryContainer_Check(PyObject *)`. This returns non-zero if the Python container is not as expected.
`PyUnaryContainer_Size`	A pointer to a function that returns the size of the Python container. The function signature is `Py_ssize_t PyUnaryContainer_Size(PyObject *op)`. This returns the size of the the Python container.
`PyUnaryContainer_Get`	A pointer to a function that gets a `PyObject *` from the Python container at a given index. The function signature is `PyObject *PyUnaryContainer_Get(PyObject *, size_t)`.

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).

1.4.5 Python to C++ Implementation

The implementation is fairly straightforward in `src/cpy/python_object_convert.h` (lightly edited):

```
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) {
    assert(!PyErr_Occurred());
```

(continues on next page)

(continued from previous page)

```
    int ret = 0;
    list_like.clear();
    Py_INCREF(op); // Increment borrowed reference
    if (!PyUnaryContainer_Check(op)) {
        PyErr_Format(
            PyExc_ValueError,
            "Can not convert Python container of type %s",
            op->ob_type->tp_name
        );
        ret = -1;
        goto except;
    }
    for (Py_ssize_t i = 0; i < PyUnaryContainer_Size(op); ++i) {
        PyObject *value = PyUnaryContainer_Get(op, i);
        if (!value) {
            ret = -2;
            goto except;
        }
        if (!(*PyObject_Check)(value)) {
            list_like.clear();
            PyErr_Format(
                    PyExc_ValueError,
                    "Python value of type %s can not be converted",
                    value->ob_type->tp_name
            );
            ret = -3;
            goto except;
        }
        list_like.push_back((*PyObject_Convert)(value));
        // Check !PyErr_Occurred() which could never happen as we check first.
    }
    assert(!PyErr_Occurred());
    goto finally;
except:
    assert(PyErr_Occurred());
    list_like.clear();
finally:
    Py_DECREF(op); // Decrement borrowed reference
    return ret;
}
```

1.4.6 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,
```

(continues on next page)

(continued from previous page)

```
    int (*PyObject_Check)(PyObject *),
    T (*PyObject_Convert)(PyObject *)
>
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 the Python `tuple`:

```
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 it is any other Python object then ignore it as py_tuple_len()
    // may give undefined behaviour.
    // Leave it to very_generic_py_unary_to_cpp_std_list_like() to 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);
}
```

1.5 Generated Functions

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

1.5.1 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)
```

1.5.2 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)
```

CHAPTER TWO

USING THIS C++ LIBRARY

2.1 The Basics

2.1.1 Get the Project from `git`

```
$ git pull https://github.com/paulross/PyCppContainers.git
```

2.1.2 Code Generation

If necessary run the code generator:

```
cd src/py
python3 code_gen.py
```

Which should give you something like:

```
Version: 0.4.0
Target directory "src/cpy"
Writing declarations to "src/cpy/auto_py_convert_internal.h"
Wrote 4125 lines of code with 356 declarations.
Writing definitions to  "src/cpy/auto_py_convert_internal.cpp"
Wrote 3971 lines of code with 352 definitions.

Process finished with exit code 0
```

2.1.3 Build Configuration

You need to compile the following C++ files by adding them to your makefile or `CMakeLists.txt`: This project has a `CMakeLists.txt` as an example.

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

2.1.4 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 *C++ API* in the namespace `Python_Cpp_Containers`.

2.1.5 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 "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:

```
>>> 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::vector<char>> vec;
    if (!py_tuple_to_cpp_std_vector(arg, vec)) {
        return cpp_std_vector_to_py_tuple(reverse_vector(vec));
    }
    return NULL;
}
```

Once again the declaration `std::vector<std::vector<char>> vec;` ensures that the correct instantiations of the 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::vector<char>, 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*.

CHAPTER

THREE

BUILDING AND TESTING

This chapter describes how to build and test this library. It assumes that you are in your chosen directory and have done:

```
$ git pull https://github.com/paulross/PyCppContainers.git
```

3.1 Building and Testing C++ Code

3.1.1 Code Generation

If necessary run the code generator:

```
cd src/py
python3 code_gen.py
```

Which should give you something like:

```
Version: 0.4.0
Target directory "src/cpy"
Writing declarations to "src/cpy/auto_py_convert_internal.h"
Wrote 4125 lines of code with 356 declarations.
Writing definitions to  "src/cpy/auto_py_convert_internal.cpp"
Wrote 3971 lines of code with 352 definitions.

Process finished with exit code 0
```

3.1.2 Building C++ Code

Here are some examples of cleaning and building this project, both debug and release builds:

```
echo "---> C++ clean debug"
cmake --build cmake-build-debug --target clean -- -j 6
echo "---> C++ build debug"
cmake --build cmake-build-debug --target PyCppContainers -- -j 6
echo "---> C++ clean release"
cmake --build cmake-build-release --target clean -- -j 6
echo "---> C++ build release"
cmake --build cmake-build-release --target PyCppContainers -- -j 6
```

3.1.3 Testing C++ Code

```
echo "---> C++ debug tests"
cmake-build-debug/PyCppContainers
echo "---> C++ release tests"
cmake-build-release/PyCppContainers
```

The debug tests only test the basic functionality but they do so with some expensive integrity tests. Release tests are much more extensive testing the performance of the library.

For example running the debug tests:

```
time cmake-build-debug/PyCppContainers
Hello, World!
Python version: 3.12.1
test_functional_all START
RSS(Mb): was:     17.555 now:     17.613 diff:     +0.059 Peak was:     17.555 now:    ␣
↪17.613 diff:     +0.059 test_vector_to_py_tuple<bool>
RSS(Mb): was:     17.617 now:     17.660 diff:     +0.043 Peak was:     17.617 now:    ␣
↪17.660 diff:     +0.043 test_vector_to_py_tuple<long>
RSS(Mb): was:     17.660 now:     17.664 diff:     +0.004 Peak was:     17.660 now:    ␣
↪17.664 diff:     +0.004 test_vector_to_py_tuple<double>
8<---- Snip ---->8
RSS(Mb): was:     16.762 now:     16.805 diff:     +0.043 Peak was:     16.762 now:    ␣
↪16.805 diff:     +0.043 test_cpp_std_map_to_py_dict_string
RSS(Mb): was:     16.805 now:     16.828 diff:     +0.023 Peak was:     16.805 now:    ␣
↪16.828 diff:     +0.023 test_py_dict_to_cpp_std_map_like_string
RSS(Mb): was:     16.805 now:     16.828 diff:     +0.023 Peak was:     16.805 now:    ␣
↪16.828 diff:     +0.023 test_py_dict_to_cpp_std_map_string
==== RSS(Mb): was:      7.203 now:     16.828 diff:     +9.625 Peak was:      7.203 now:␣
↪     16.828 diff:     +9.625 ==== test_functional.cpp
test_functional_all FINISH
```

This is followed by the test results:

```
Number of tests: 3333
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 1048576       1     0.001676959             N/A             N/A             N/
↪A          1         596.3 test_functional_tuple_setitem():[1048576]
TEST:     0 1048576       1     0.000595792             N/A             N/A             N/
↪A          1        1678.4 test_functional_list_setitem():[1048576]
TEST:     0 1048576       1     0.000947000             N/A             N/A             N/
↪A          1        1056.0 test_functional_set_add():[1048576]     8<---- Snip ---->8
TEST:     0   32768       1     0.117341292             N/A             N/A             N/
↪A          1           8.5 test_py_tuple_str32_to_vector std::string[2048]>():[32768]
TEST:     0   65536       1     0.249899958             N/A             N/A             N/
↪A          1           4.0 test_py_tuple_str32_to_vector std::string[2048]>():[65536]
TEST:     0    4096       1     4.806965084             N/A             N/A             N/
↪A          1           0.2 test_unordered_set_bytes_to_py_set std::string[1048576]>
↪():[4096]
```

(continues on next page)

(continued from previous page)

```
TAIL: Passed=3333/3333 Failed=0/3333
All tests pass.

====RSS(Mb): was:     10.012 now:    180.496 diff:   +170.484 Peak was:     10.012 now: ␣
↪ 3250.059 diff:  +3240.047 main.cpp
Total execution time:     2370.715 (s)
Count of unique strings created: 5895690
Bye, bye! Returning 0
     2371.63 real      2324.66 user        32.06 sys
```

This takes, typically, 40 minutes. A return code of 0 is success. If there are any failing tests then the return code will be the number of failing tests.

The release tests are similar but they include all the performance tests which take a long while. Run time is around six hours.

3.2 Building and Testing Python Code

3.2.1 Building Python Code

To build all the Python code create a virtual environment then:

```
$ pip install -r requirements-dev.txt
$ python setup.py develop
```

This takes a minute or so.

3.2.2 Testing Python Code

The Python tests check these things:

- Functional testing for Python C extensions.
- Performance testing for Python C extensions that exercise the C++ library. Usually round tripping Python structures to C++ and back again.
- Memory usage testing for Python C extensions that use this C++ library.

As a basic, from your virtual environment:

```
$ pytest tests/
================================ test session starts ==============================
platform darwin -- Python 3.12.1, pytest-8.3.3, pluggy-1.5.0
rootdir: PyCppContainers, configfile: pytest.ini
collected 128 items

tests/unit/test_cPyCppContainers.py ......x................................. [ 35%]
.................................                                            [ 57%]
tests/unit/test_cUserDefined.py .........                                    [ 64%]
tests/unit/test_perf_cPyCppContainers.py ssssssssssssssssssssssssssssssssss [ 91%]
tests/unit/test_with_pymemtrace.py sssssssssss                               [100%]
```

(continues on next page)

(continued from previous page)

```

====================== 81 passed, 46 skipped, 1 xfailed in 2.74s ==================
```

By default this only does the functional tests and skips the others such as performance and memory tests. To run these tests you need to add the arguments `--runslow` and `-pymemtrace` respectively, see below. This takes about 40 minutes.

Testing Performance

To include all the performance tests, this takes about 25 minutes:

```
$ pytest tests/ --runslow
```

Example:

```
$ time pytest tests --runslow
============================ test session starts ==================================
platform darwin -- Python 3.12.1, pytest-8.3.3, pluggy-1.5.0
rootdir: PythonCppHomogeneousContainers
configfile: pytest.ini
collected 128 items

tests/unit/test_cPyCppContainers.py ......x................................. [ 35%]
.................................                                            [ 57%]
tests/unit/test_cUserDefined.py .........                                    [ 64%]
tests/unit/test_perf_cPyCppContainers.py ................................... [ 91%]
tests/unit/test_with_pymemtrace.py sssssssssss                               [100%]

=========== 116 passed, 11 skipped, 1 xfailed in 1595.44s (0:26:35) ===============
pytest tests --runslow  1122.24s user 376.19s system 93% cpu 26:35.99 total
```

Testing Memory Usage

To include all the memory tests, this takes about 20 minutes:

```
$ time pytest tests --pymemtrace
============================ test session starts ==================================
platform darwin -- Python 3.12.1, pytest-8.3.3, pluggy-1.5.0
rootdir: PythonCppHomogeneousContainers
configfile: pytest.ini
collected 128 items

tests/unit/test_cPyCppContainers.py ......x................................. [ 35%]
.................................                                            [ 57%]
tests/unit/test_cUserDefined.py .........                                    [ 64%]
tests/unit/test_perf_cPyCppContainers.py sssssssssssssssssssssssssssssssssss [ 91%]
tests/unit/test_with_pymemtrace.py ...........                               [100%]

============ 92 passed, 35 skipped, 1 xfailed in 1237.96s (0:20:37) ==============
Opening log file 20241010_102101_11927.log
Opening log file 20241010_102106_11927.log
```

(continues on next page)

(continued from previous page)

```
Opening log file 20241010_102115_11927.log
Opening log file 20241010_102136_11927.log
Opening log file 20241010_102202_11927.log
Opening log file 20241010_102214_11927.log
Opening log file 20241010_103049_11927.log
Opening log file 20241010_103453_11927.log
Opening log file 20241010_103925_11927.log
Opening log file 20241010_104003_11927.log
Opening log file 20241010_104051_11927.log
pytest tests --pymemtrace  982.16s user 248.42s system 99% cpu 20:38.30 total
```

Running with both `--runslow` and `--pymentrace` takes about about 40 minutes.

3.3 Building the Documentation

This describes how create the documentation with `gnuplot`, `Sphinx` or `doxygen`.

3.3.1 `gnuplot` Plots

Recreating Plot Data

If required the performance data can recreated. Firstly the C++ performance, from the project directory:

```
$ # Pipe the results of the C++ tests to a specific file.
$ cmake-build-release/PyCppContainers > perf_notes/cpp_test_results.txt
$ cd perf_notes
$ # Run this script that will take the C++ output and split it into .dat files
$ # in perf_notes/dat.
$ python write_dat_files_for_cpp_test_results.py
```

Now for the Python tests, from the project directory, this takes about about 40 minutes:

```
$ # Pipe the results of the Python tests to a specific file.
$ pytest tests --runslow --pymemtrace -vs > perf_notes/python_test_results.txt
$ cd perf_notes
$ # Run this script that will take the Python output and split it into .dat files
$ # in perf_notes/dat.
$ python write_dat_files_for_python_test_results.py
```

```
$ # Copy the .dat files to the documentation ready for gnuplot, from project directory.
$ cp perf_notes/dat/*.dat docs/sphinx/source/plots/dat
```

Note: Memory plots

The .dat files for memory plots are not (yet) automated and have to be done by hand by copying the `pymemtrace` log files.

Recreating Plot Images

To recreate the gnuplot plot images that are used by the documentation from the project directory:

```
$ cd docs/sphinx/source/plots
$ gnuplot -p *.plt
```

3.3.2 Sphinx

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

```
$ cd docs/sphinx
$ make clean
$ make html latexpdf
$ cp build/latex/PyCppContainers.pdf ..
$ open build/html/index.html
$ open ../PyCppContainers.pdf
```

3.3.3 Doxygen

To build the HTML Doxygen documentation from the project directory:

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

The Doxygen PDF:

```
$ cd docs/doxygen/latex
$ make pdf
$ cp refman.pdf ../../PyCppContainers_Doxygen.pdf
```

3.4 Building and Testing Everything for Multiple Python Versions

The script `build_all.sh` will execute:

- C++ clean and build debug and release versions.
- Run C++ debug build and the associated tests (this omits C++ performance tests).
- Run C++ release build and the all the tests including C++ performance tests.
- **For each Python version (currently 3.8, 3.9, 3.10, 3.11, 3.12, 3.13) it:**
    - Creates a new virtual environment.
    - Runs `pip install -r requirements-dev.txt`.
    - Runs `python setup.py develop`.
    - Runs `pytest tests/ -x` to catch any functional errors.
    - Runs `pytest tests/ -vs --runslow --pymemtrace` to run all tests.
    - Runs `python setup.py bdist_wheel` to create the wheels.

    - Runs `python setup.py sdist` to create the source distribution.

If any of these fail the script will halt with a failure indication.

The output is verbose typically 30,000 lines.

The typical time breakdown is:

- C++ debug and release builds: 5 minutes.
- C++ debug tests (3,000+): 40 minutes.
- C++ release tests (around 25,000): about 40 minutes.
- Python: create environment and run all tests, including slow and memory tests (127): around 45 minutes per Python version.

For all Python versions (6 currently) this takes about eight hours.

This does not build the documentation.

CHAPTER

FOUR

C++ API

This chapter describes the C++ API to this library.

4.1 Include File and Namespace

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

All these APIs are in the namespace `Python_Cpp_Containers`.

4.2 Python Containers to C++

4.2.1 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 and the given container cleared.

4.2.2 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...
    } else {
        // Use vec...
    }
}
```

4.2.3 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...
    } else {
        // Use vec...
    }
}
```

4.2.4 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...
    } else {
        // Use set...
    }
}
```

4.2.5 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...
    } else {
        // Use frozenset...
    }
}
```

4.2.6 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...
}
```

4.3 C++ Containers to Python

4.3.1 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 the 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.

4.3.2 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);
}
```

4.3.3 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);
}
```

4.3.4 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);
}
```

4.3.5 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);
}
```

4.3.6 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]` from a `std::unordered_map<long, double>`:

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

Create a dict of Python `[int, str]` from a `std::map<long, std::string>`:

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

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.

Essentially there are functions to make these conversions between Python and C++:

- Objects that can be members of containers. Examples are float/double, strings and user defined types. See *User Defined Types* for more information on the latter.
- Containers. Examples are Python lists to and from C++ `std::list<T>`.

5.1 Object Conversion Source Files

The are in `python_object_convert.h` and `python_object_convert.cpp`.

There 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:

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

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

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

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

```
#include "Python.h"

int py_long_check(PyObject *op) {
    return PyLong_Check(op);
}

long py_long_to_cpp_long(PyObject *op) {
    assert(py_long_check(op));
    return PyLong_AsLong(op);
```

(continues on next page)

(continued from previous page)

```
}

PyObject *cpp_long_to_py_long(const long &l) {
    return PyLong_FromLong(l);
}
```

The implementations for more complex types, such as string conversion, are a little more complicated but not greatly so.

5.2 Container Conversion Source Files

These are in `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 (such as `list`, `tuple`, `set`). There are a number of functions to each container, for example a `list`:

- Check that each Python container 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. For example:

```
// List wrappers around PyList_Check, PyList_New, PyList_SET_ITEM, PyList_GET_ITEM
int py_list_check(PyObject *op) {
    return PyList_Check(op);
}
PyObject *py_list_new(size_t len) {
    return PyList_New(len);
}
Py_ssize_t py_list_len(PyObject *op) {
    return PyList_Size(op);
}
int py_list_set(PyObject *list_p, size_t pos, PyObject *op) {
    PyList_SET_ITEM(list_p, pos, op);
    return 0;
}
PyObject *py_list_get(PyObject *list_p, size_t pos) {
    return PyList_GET_ITEM(list_p, pos);
}
```

5.3 `python_convert.h`

This is the top level file that gives access to the whole library. Is 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`, that declares the templates then includes `auto_py_convert_internal.h` the auto generated file of template specialisations.

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...
}
```

The template types are:

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 *(*ConvertCppToPy)(const T &)`	A pointer to a function that takes a type T and returns a new Python `PyObject*`.
`PyObject *(*PyUnaryContainer_New)(size_t)`	A pointer to a function that returns a new Python container of the given length.
`int(*PyUnaryContainer_Set)(PyObject *, size_t, PyObject *)>`	Sets a Python object in the Python container at the given position.

And the parameters are:

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

Parameter	Description
`list_like`	The C++ container.

This returns a new PyObject or NULL on failure.

The hand written implementation 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) {
    assert(!PyErr_Occurred());
    PyObject *ret = PyUnaryContainer_New(list_like.size());
    if (ret) {
        size_t i = 0;
        for (const auto &val: list_like) {
            PyObject *op = (*ConvertCppToPy)(val);
            if (!op) {
                // Failure, do not need to decref the contents as that will be done
                // when decref'ing the container. e.g. tupledealloc():
                // https://github.com/python/cpython/blob/main/Objects/tupleobject.c#L268
                PyErr_Format(PyExc_ValueError, "C++ value of can not be converted.");
                goto except;
            }
#ifndef NDEBUG
            // Refcount may well be >> 1 for interned objects.
            Py_ssize_t op_ob_refcnt = op->ob_refcnt;
#endif
            // PyUnaryContainer_Set usually wraps a void function,
            // always succeeds returning non-zero.
            if (PyUnaryContainer_Set(ret, i++, op)) { // Stolen reference.
                PyErr_Format(PyExc_RuntimeError, "Can not set unary value.");
                goto except;
            }
#ifndef NDEBUG
            assert(op->ob_refcnt == op_ob_refcnt
                   && "Reference count incremented instead of stolen.");
#endif
        }
    } else {
        PyErr_Format(
            PyExc_ValueError,
            "Can not create Python container of size %ld",
            list_like.size()
        );
        goto except;
    }
```

(continues on next page)

(continued from previous page)

```
    assert(!PyErr_Occurred());
    assert(ret);
    goto finally;
except:
    Py_XDECREF(ret);
    assert(PyErr_Occurred());
    ret = NULL;
finally:
    return ret;
}
```

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>`, `std::string` and so on.

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++

The conversion from a Python `list` or `tuple` to a C++ `std::vector<T>` or `std::list<T>` 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 3: Template to convert a Python `tuple` or `list` to a `std::vector` or `std::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 4: 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`.

The implementation looks like this:

```
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) {
    assert(!PyErr_Occurred());
    int ret = 0;
    list_like.clear();
    Py_INCREF(op); // Increment borrowed reference
    if (!PyUnaryContainer_Check(op)) {
        PyErr_Format(
            PyExc_ValueError,
            "Can not convert Python container of type %s",
            op->ob_type->tp_name
        );
        ret = -1;
        goto except;
    }
    for (Py_ssize_t i = 0; i < PyUnaryContainer_Size(op); ++i) {
        PyObject *value = PyUnaryContainer_Get(op, i);
        if (!value) {
            ret = -2;
            goto except;
        }
        if (!(*PyObject_Check)(value)) {
            list_like.clear();
            PyErr_Format(
                    PyExc_ValueError,
                    "Python value of type %s can not be converted",
                    value->ob_type->tp_name
            );
            ret = -3;
            goto except;
        }
        list_like.push_back((*PyObject_Convert)(value));
        // Check !PyErr_Occurred() which could never happen as we check first.
    }
    assert(!PyErr_Occurred());
    goto finally;
except:
    assert(PyErr_Occurred());
    list_like.clear();
finally:
    Py_DECREF(op); // Decrement borrowed reference
    return ret;
}
```

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 reallocation.

```
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 implementation. Omitted for simplicity.
}
```

The template types are:

Table 5: Convert a std::unordered_set to a Python set or frozenset.

Type	Description
typename T	The C++ type of each object in the container.
PyObject *(*ConvertCppToPy)(const T &)	A pointer to a function that takes a type T and returns a new Python PyObject*.
PyObject *(*PyUnaryContainer_New)(size_t)	A pointer to a function that returns a new Python container.

And the parameters are:

Table 6: Function to convert a std::vector or std::list to a Python tuple or list.

Parameter	Description
set	The C++ container.

This returns a new PyObject or NULL on failure.

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 *
```

(continues on next page)

(continued from previous page)

```
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. Omitted for simplicity.
}
```

Template parameters are:

Table 7: Template to convert a Python set or frozenset to a std::unordered)set.

Type	Description
typename T	The C++ type of the object.
int(*PyUnaryContainer_Check)(PyObject *)	A pointer to a function returns true if the Python container is of the relevant type (set or frozenset in this case).
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.

Parameters are:

Table 8: Function to convert a Python set or frozenset to a std::unordered)set.

Parameter	Description
op	The Python container.
set	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.

Here are the specialisations for set and frozenset:

```
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
```

(continues on next page)

(continued from previous page)

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

// Instantiations
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 handwritten 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. Omitted for simplicity.
}
```

The template types are:

Table 9: Convert a `std::map` or `std::unordered_map` to a Python `dict`.

Type	Description
`Map`	The C++ container, for example a `std::map` or a `std::unordered_map`.
`typename K`	The C++ type of the keys in the container.
`typename V`	The C++ type of the values in the container.
`PyObject *(*Convert_K)(const T &)`	A pointer to a function that takes a type `K` and returns a new Python `PyObject *`.
`PyObject *(*Convert_V)(const T &)`	A pointer to a function that takes a type `V` and returns a new Python `PyObject *`.

And the parameters are:

Table 10: Function to convert a `std::map` or `std::unordered_map` to a Python `dict`.

Parameter	Description
`map`	The C++ container.

This returns a new PyObject or NULL on failure.

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 *
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,
```

(continues on next page)

(continued from previous page)

```
        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. Omitted for simplicity.
}
```

Template parameters are:

Table 11: Template to convert a Python `dict` to a `std::map` or a `std::unordered_map`.

Type	Description
`Map`	The C++ container, for example a `std::map` or a `std::unordered_map`.
`typename K`	The C++ type of the keys.
`typename V`	The C++ type of the values.
`int (*Check_K)(PyObject *)`	A pointer to a function returns true if Python object can be converted to a C++ object of type `K`.
`int (*Check_V)(PyObject *)`	A pointer to a function returns true if Python object can be converted to a C++ object of type `V`.
`T (*Convert_K)(PyObject *)`	A pointer to a function to convert a Python object to a C++ object of type `K`.
`T (*Convert_V)(PyObject *)`	A pointer to a function to convert a Python object to a C++ object of type `V`.

Parameters are:

Table 12: Function to convert a `std::vector` or `std::list` to a Python `tuple` or `list`.

Parameter	Description
`dict`	The Python container.
`map`	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 `K` or `V`.

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`:

```
// 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`:

```
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

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
Version: 0.4.0
Target directory "src/cpy"
Writing declarations to "src/cpy/auto_py_convert_internal.h"
Wrote 4125 lines of code with 356 declarations.
Writing definitions to  "src/cpy/auto_py_convert_internal.cpp"
Wrote 3971 lines of code with 352 definitions.

Process finished with exit code 0
```

CHAPTER

SIX

USER DEFINED TYPES

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

This is probably best done by example. In this case we take an existing Python object defined in a CPython extension and develop its equivalent in C++. Of course, the opposite, having an existing C++ class and needing to develop a Python equivelent in a CPython extension, might be the use case. The principles are the same.

This example will demonstrate supporting the conversion of a `list` of user defined Python to and from a `std::vector` C++ equivalents.

There are several steps:

- Have the definitions of both the CPython and C++ equivalent objects. See the examples *A Python Class* and *The C++ Class*.
- Define the function to check the Python object type. An example is *Checking the Python Type*.
- Define the two conversion functions from CPython to C++ and the reverse. See the examples *From Python to C++* and *From Python to C++*.

These steps only has to be done once regardless of how many containers are to be supported.

Finally for each container conversion declare the two way template specialisation and definitions. These will be simple one-liner calls to this project's generic functions. See *From Python to C++* and *From C++ to Python*.

All this code is in the project directory in `src/ext/cUserDefined.h` and `src/ext/cUserDefined.cpp`.

6.1 User Defined Types in a C Extension

6.1.1 A Python Class

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`.

Otherwise it is identical to the example in the Python documentation.

In this example the `CustomObject` class is created in this project in `src/ext/cUserDefined.cpp`. It looks like this:

```
typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
```

(continues on next page)

(continued from previous page)

```
    int number;
} CustomObject;
```

This also has a method `name()` that combines the first and last names such as this:

```
static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored)) {
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}
```

Note: For clarity this is equivalent to this Python code:

```
import dataclasses

@dataclasses.dataclass
class CustomObject:
    first: str
    last: str
    number: int

def name(self) -> str:
    return f'{self.first} {self.last}'
```

The `setup.py` file would include this Extension definition:

```
Extension(
    'cUserDefined',
    sources=[
        'src/ext/cUserDefined.cpp',
        # Include this libraries source files.
        'src/cpy/auto_py_convert_internal.cpp',
        'src/cpy/python_container_convert.cpp',
        'src/cpy/python_object_convert.cpp',
    ],
    include_dirs=[
        'src',
    ],
    extra_compile_args=extra_compile_args,
),
```

Once the Python `cUserDefined` extension is built it can be used in Python like this:

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

So much for the CPython example, now for the equivalent code in C++.

6.1.2 The C++ Class

Here is the user defined pure C++ class that contains a first name, second name and a number which mirrors the CPython code above. It is declared in the File `cUserDefined.h`:

```
#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;
};
```

6.1.3 Checking the Python Type

We need to know that any `PyObject` is really a well formed `CustomObject`. Here is the code to verify the Python type and its contents in `cUserDefined.cpp`. It returns 1 on success, 0 otherwise:

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

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;
}
```

Now add some conversion code from the CPython `CustomObject` to the C++ `CppCustomObject`:

6.1.4 Conversion Code

Now, in the Python C extension add the verification and conversion code between the Python CustomObject and the C++ CppCustomObject.

This code is in cUserDefined.cpp and include the necessary files, this ensures that we have access to the C++ CppCustomObject class definition and this library's conversion machinery:

From Python to C++

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

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

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_unicode8_to_cpp_string(p->first),
            Python_Cpp_Containers::py_unicode8_to_cpp_string(p->last),
            p->number
    );
}
```

From C++ to Python

The code to convert from a C++ CppCustomObject to a new Python CustomObject (error checking omitted for clarity):

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

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_unicode8(obj.first());
        op->last = Python_Cpp_Containers::cpp_string_to_py_unicode8(obj.last());
        op->number = obj.number();
    }
    return (PyObject *) op;
}
```

6.1.5 Template Specialisation

Now in the file, cUserDefined.h, include this project's header file and then in this project's namespace declare the specialisations to call this library's generic functions to convert to and from containers. Specifically a std::vector of these objects. These are basically one-liners:

From Python to C++

```
#include "python_convert.h"

// Specialised declaration in cUserDefined.h

namespace Python_Cpp_Containers {

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

} // namespace Python_Cpp_Containers
```

In the file cUserDefined.cpp implement the specialisation, this is just a one-liner calling the generic conversion code in this library with the types and functions we have created.

```
#include "cUserDefined.h"

// Specialised definition in cUserDefined.cpp

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
```

From C++ to Python

And for the reverse:

```
#include "python_convert.h"

// Specialised declaration in cUserDefined.h

namespace Python_Cpp_Containers {

    // C++ to Python
    template<>
    PyObject *
    cpp_std_list_like_to_py_list<CppCustomObject>(
        const std::vector<CppCustomObject> &container
    );

} // namespace Python_Cpp_Containers
```

In the file `cUserDefined.cpp` implement the specialisation, this is just a one-liner calling the generic conversion code in this library.

```
#include "cUserDefined.h"

// Specialised declaration in cUserDefined.cpp

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
```

Note: If you wanted to support Python lists to and from C++ `std::list<CppCustomObject>`

Then create new specialisations of the templates with `std::list<CppCustomObject>` Instead of `std::vector<CppCustomObject>`.

Note: If you also wanted to support Python tuples to and from C++ `std::vector<T>` then specialise the templates with `generic_py_tuple_to_cpp_std_list_like` and `generic_cpp_std_list_like_to_py_tuple`.

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

6.2 Using the C++ Conversion Functions

6.2.1 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);
```

6.2.2 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
    // ...
}
```

6.2.3 Example of Round-trip Conversion

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

(continues on next page)

(continued from previous page)

```
    }
    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']
```

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

Now it takes very little additional work to support conversion between a Python `dict[int, cUserDefined.Custom]` to and from a C++ `std::map<long, CppCustomObject>` or, indeed, any other container.

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

```
namespace Python_Cpp_Containers {

    // Specialised declarations

    // Python to C++
    template <>
    int
    py_dict_to_cpp_std_map_like<std::map, long, CppCustomObject>(
```

(continues on next page)

(continued from previous page)

```
        PyObject* op, std::map<long, CppCustomObject> &map
    );

    // C++ to Python
    template<>
    PyObject *
    cpp_std_map_like_to_py_dict<std::map, long, CppCustomObject>(
        const 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 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
```

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
    ) {
```

(continues on next page)

(continued from previous page)

```
        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
```

Example Code

Here is an example of using both of them in the `cUserDefined` extension 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;
    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 now 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>}
```

Check that the names have been reversed:

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

6.3 User Defined Types From Pure Python Types

Todo: User Defined Types From Pure Python Types: Add in version 0.5.0

6.4 Interoperation with `numpy` ND Arrays

Todo: Interoperation with `numpy` ND Arrays: Add the existing example code in version 0.5.0.

CHAPTER

SEVEN

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 Apple M1 (2020) CPU running Mac OS X 13.5.1 (Ventura) with 16 GB RAM. These test results are built with Python 3.12.

There is a lot of formal engineering data in this chapter but it starts with a summary:

7.1 Summary

- Fundamental types (`bool`, `int`, `float`, `complex`) can be converted at around 100m objects/sec. Converting these from Python to C++ is often around 4x faster than the reverse.
- Sequences of bytes or 8 bit character strings are converted at a memory rate of around 30 GB/sec.
- Sequences 8 bit character strings are 3x slower than bytes achieving a memory rate of around 10 GB/sec.
- 16 and 32 bit character strings are much slower than 8 bit strings (100x), around 100 MB/s.
- 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.
- Memory usage matches the object data sizes.
- There are no memory leaks in this library.

7.2 C++ Performance Tests

7.2.1 Test Procedure

The main entry point to the `PyCppContainers` project is in `src/main.cpp` and runs the functional, performance and memory tests.

The performance 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 6.5 hours.

The tests can be run by building and running the C++ binary from the project root:

```
cmake --build cmake-build-release --target clean -- -j 6
cmake --build cmake-build-release --target PyCppContainers -- -j 6
cmake-build-release/PyCppContainers
```

Note: The debug build includes more exhaustive internal tests (using `assert()`) but excludes the performance tests as they take a *very* long time for a debug build.

The output is large and looks like this:

```
$ cmake-build-release/PyCppContainers
---> C++ release tests
Hello, World!
Python version: 3.12.1
test_functional_all START
RSS(Mb): was:     16.523 now:     16.531 diff:     +0.008 Peak was:     16.523 now:     16.531 diff:     +0.008 test_
↪vector_to_py_tuple<bool>
RSS(Mb): was:     16.531 now:     16.535 diff:     +0.004 Peak was:     16.531 now:     16.535 diff:     +0.004 test_
↪vector_to_py_tuple<long>
RSS(Mb): was:     16.535 now:     16.539 diff:     +0.004 Peak was:     16.535 now:     16.539 diff:     +0.004 test_
↪vector_to_py_tuple<double>

8<---- Snip ---->8

TEST:    0    4096       1     0.002047584            N/A            N/A            N/A          1         488.4 test_py_
↪tuple_str32_to_vector std::string[2048]>():[4096]
TEST:    0    8192       1     0.004002917            N/A            N/A            N/A          1         249.8 test_py_
↪tuple_str32_to_vector std::string[2048]>():[8192]
TEST:    0   16384       1     0.008183250            N/A            N/A            N/A          1         122.2 test_py_
↪tuple_str32_to_vector std::string[2048]>():[16384]
TEST:    0   32768       1     0.039068668            N/A            N/A            N/A          1          25.6 test_py_
↪tuple_str32_to_vector std::string[2048]>():[32768]
TEST:    0   65536       1     0.044092626            N/A            N/A            N/A          1          22.7 test_py_
↪tuple_str32_to_vector std::string[2048]>():[65536]
TEST:    0    4096       1     4.745317500            N/A            N/A            N/A          1           0.2 test_
↪unordered_set_bytes_to_py_set std::string[1048576]>():[4096]
TAIL: Passed=24192/24192 Failed=0/24192
All tests pass.

====RSS(Mb): was:      9.262 now:    844.883 diff:   +835.621 Peak was:      9.262 now:   3593.207 diff:  +3583.945 main.
↪cpp
Total execution time:    23880.011 (s)
Count of unique strings created: 131724750
Bye, bye! Returning 0
```

The complete output can be captured to `perf_notes/cpp_test_results.txt` with this command:

```
$ time cmake-build-release/PyCppContainers > perf_notes/cpp_test_results.txt
```

Then there is a Python script `perf_notes/write_dat_files_for_cpp_test_results.py` that will extract all the performance data into `perf_notes/dat` suitable for gnuplot. Copy those `*.dat` files into `docs/sphinx/source/plots/dat` then `cd` into `docs/sphinx/source/plots` and run `gnuplot -p *.plt` to update all the performance plots referenced in the documentation.

Note: See *Round-trip Python to C++ and back to Python* for the Python plots which can be built by gnuplot at the same time.

7.2.2 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.

Numeric Types

Table 1: Fundamental Type Conversion Time. Times in nanoseconds.

Type C++/Py	C++ to Py	Py to C++	Ratio	Notes
`bool`, `bool`	1.56	1.46	1.07x	The mean is around 660 million/s
`long`, `int`	18.4	4.16	4.42x	The mean is around 88 million/s.
`double`, `float`	14.2	5.56	2.55x	The mean is around 100 million/s.
`complex<double>`, `complex`	21.2	6.42	3.30x	The mean is around 72 million/s.

Converting from C++ to Python is always slower than from Python to C++. Presumably this reflects to cost of 'boxing' a Python object is higher that the cost of extracting ('unboxing') the object

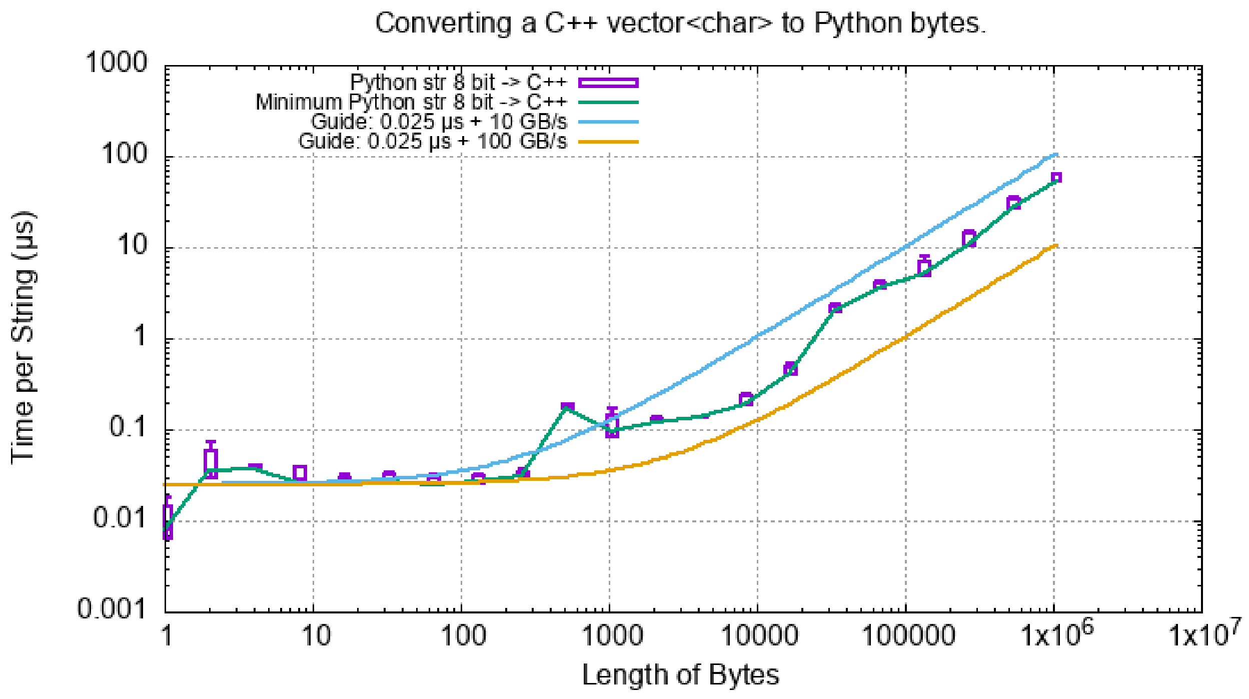
The actual tests in `src/cpy/tests/test_performance.cpp` are:

Table 2: Fundamental Type Conversion Time Test Code.

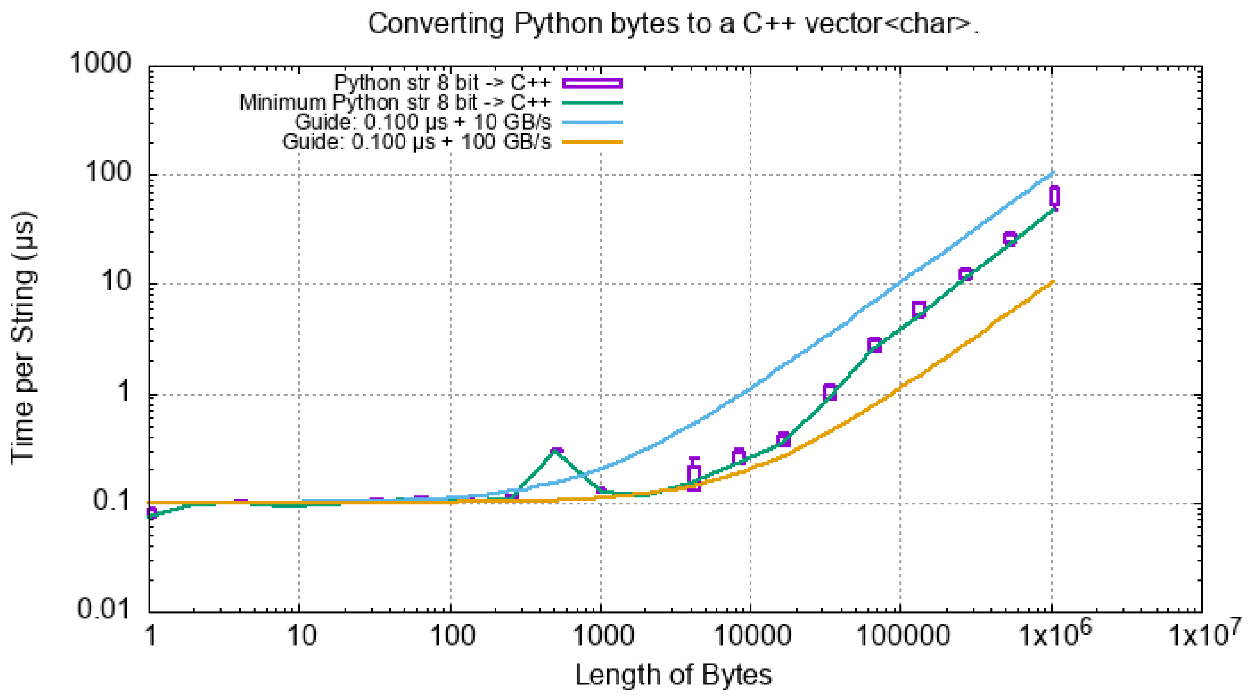
Type C++/Py	C++ to Py Test	Py to C++ Test
`bool`, `bool`	`test_bool_to_py_bool_multiple()` calls `cpp_bool_to_py_bool()`.	`test_py_bool_to_bool_multiple()` calls `py_bool_to_cpp_bool()`.
`long`, `int`	`test_long_to_py_int_multiple()` calls `cpp_long_to_py_long()`.	`test_py_int_to_cpp_long_multiple()` calls `py_long_to_cpp_long()`.
`double`, `float`	`test_double_to_py_float_multiple()` calls `cpp_double_to_py_float()`.	`test_py_float_to_cpp_double_multiple()` calls `cpp_double_to_py_float()`.
`complex<double>`, `complex`	`test_complex_to_py_complex_multiple()` calls `cpp_complex_to_py_complex()`.	`test_py_complex_to_cpp_complex_multiple()` calls `py_complex_to_cpp_complex()`.

`bytes`

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



This shows a linear rate asymptotic to around 30 GB/s.



This is symmetric with the performance of Python to C++.

The tests are in `src/cpy/tests/test_performance.cpp`:

Table 3: Bytes Conversion Time Test Code.

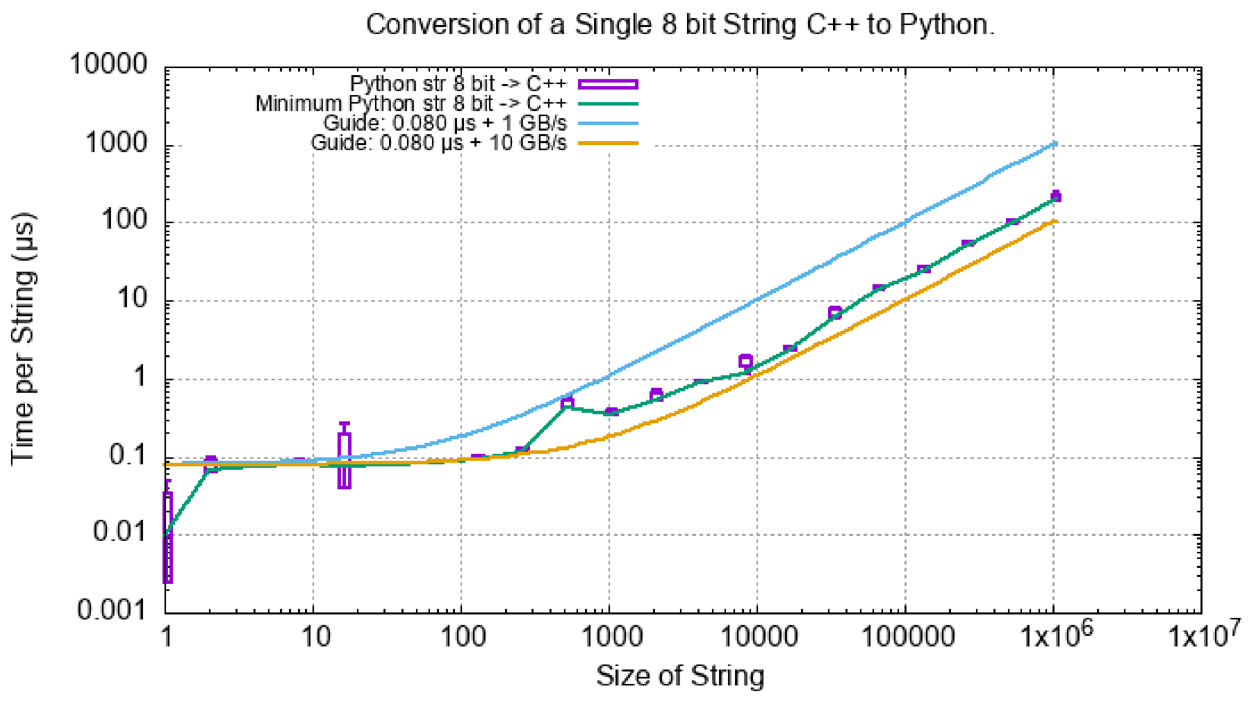
Conversion	Test Function	Calls
C++ to Python	`test_cpp_vector_char_to_py_bytes_multiple()`	`cpp_vector_char_to_py_bytes()`.
Python to C++	`test_py_bytes_to_cpp_vector_char_multiple()`	`py_bytes_to_cpp_vector_char()`.

Strings

For a single C++ `std::string`, `std::u32string` and `std::u32string` to and from Python `str` of different lengths and different word sizes.

Table 4: String Conversion Time Test Code, C++ to Python.

Type C++	Test Function	Calls
`std::string`	`test_cpp_string_to_py_str_multiple()`	`cpp_string_to_py_unicode8()`.
`std::u16string`	`test_cpp_u16string_to_py_str16_multiple()`	`cpp_u16string_to_py_unicode16()`.
`std::u32string`	`test_cpp_u32string_to_py_str32_multiple()`	`cpp_u32string_to_py_unicode32()`.



Firstly the 8 bit Unicode converts consistently at a rate of around 10 GB/s. This compares with the conversion of `std::vector<char>` to `bytes` objects at 30 GB/s (above). The threefold increase can be possibly explained by having more internal checks on unicode objects.

Intermezzo: Creating Python 16/32 bit Unicode strings

Initially this code was 100x slower than for 8 bit string and this section describes why. An explanation might be the way the Python Unicode C-API works. There are several ways of creating Unicode strings which are UCS1, UCS2 or UCS4 in CPython. The function PyUnicode_FromKindAndData() is the recommended way. However if a PyUnicode_2BYTE_KIND or a PyUnicode_4BYTE_KIND this function inspects the multibyte data and if there are no code points above 0xFF then a PyUnicode_1BYTE_KIND is created which is not what we want.

Instead we use PyUnicode_New with a suitable `maxchar` to ensure that we get the correct word size. Then we copy each character into the Unicode string in a loop.

Here is an example from this library using 16 bit unicode characters:

```
PyObject *cpp_u16string_to_py_unicode16(const std::u16string &s) {
    assert(! PyErr_Occurred());
    PyObject *ret = PyUnicode_New(s.size(), 65535);
    assert(py_unicode16_check(ret));
    for (std::u16string::size_type i = 0; i < s.size(); ++i) {
        int result = PyUnicode_WriteChar(ret, i, s[i]);
        if (result) {
            PyErr_Format(
                PyExc_SystemError,
                "PyUnicode_WriteChar() failed to write at [%ld] returning %d.",
                i, result
            );
            Py_DECREF(ret);
            return NULL;
        }
    }
    assert(py_unicode16_check(ret));
    assert(! PyErr_Occurred());
    return ret;
}
```

This code make the conversion of `std::u16string` and `std::u32string` to Python `str` around 100 times slower than for 8 bit strings. This loop, the type conversions and the PyUnicode_WriteChar internal checks is probably what is causing the slowdown.

To overcome this the memory was copied directly:

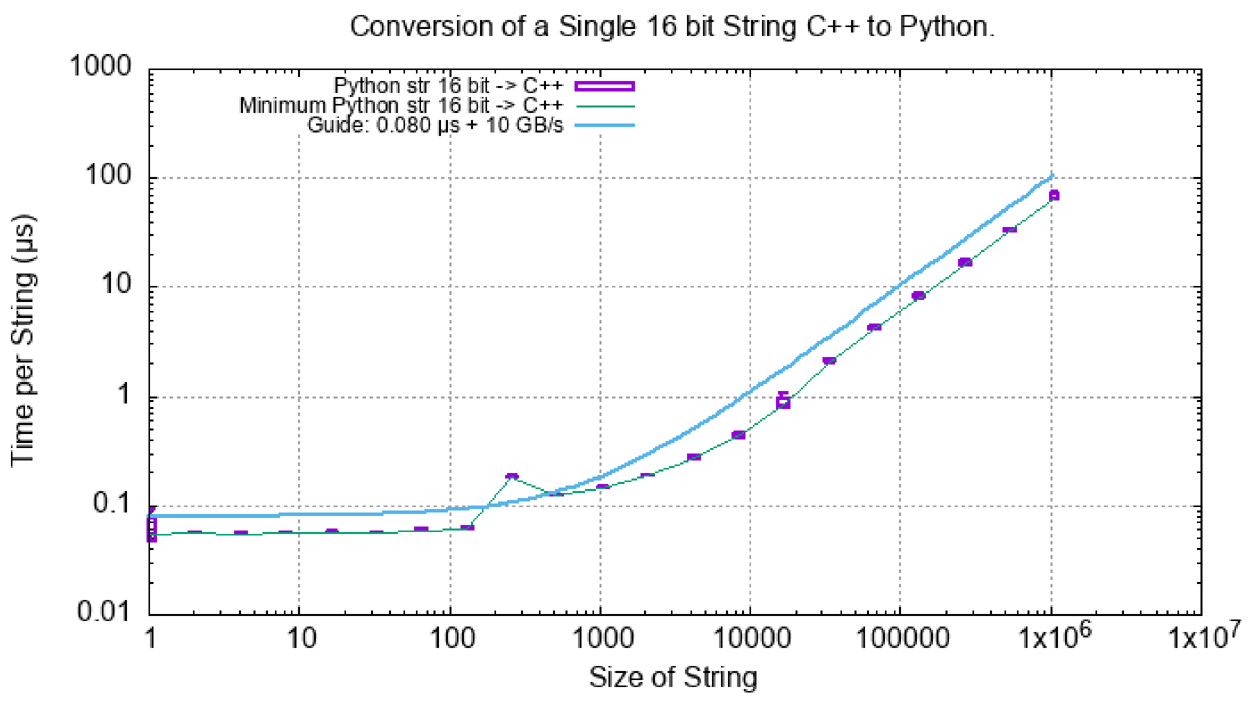
```
PyObject *cpp_u16string_to_py_unicode16(const std::u16string &s) {
    assert(! PyErr_Occurred());
    PyObject *ret = PyUnicode_New(s.size(), 65535);
    assert(py_unicode16_check(ret));
    void *dest = PyUnicode_DATA(ret);
    const void *src = s.c_str();
    rsize_t size = s.size() * sizeof(std::u16string::value_type);
    if (memcpy(dest, src, size) != dest) {
        // memcpy failure
        Py_DECREF(ret);
        return NULL;
    }
    assert(! PyErr_Occurred());
    return ret;
}
```

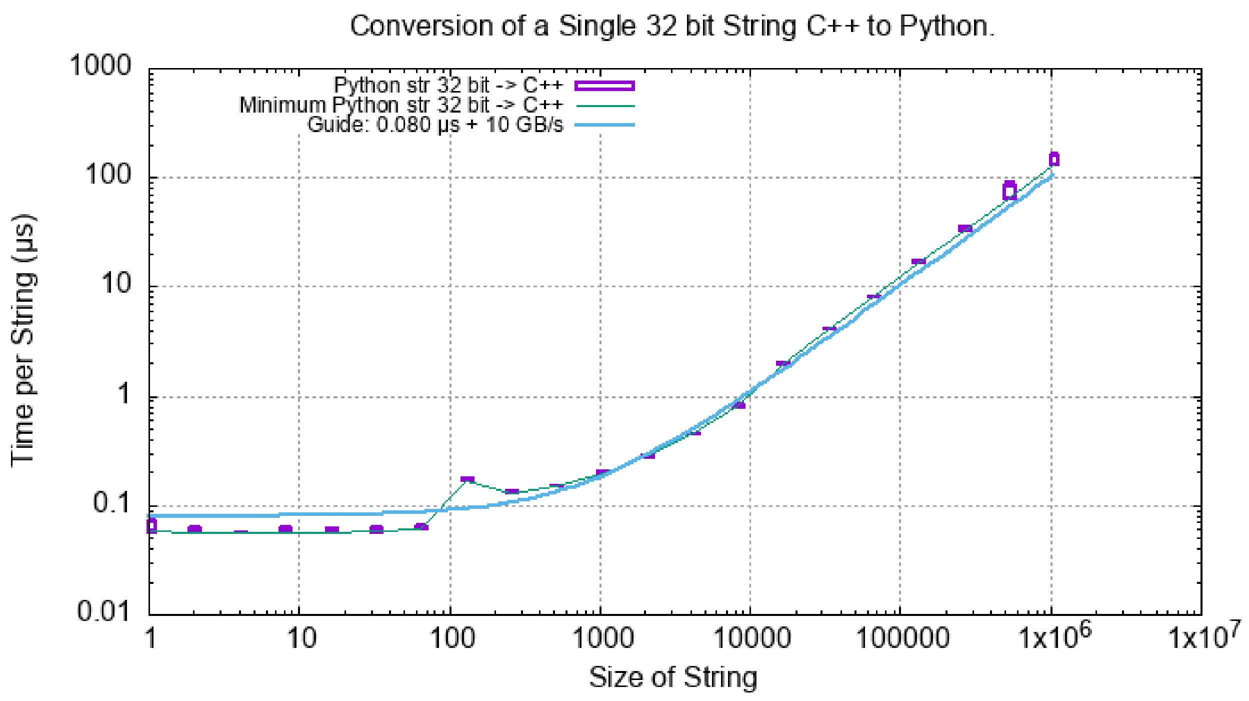
This gives 16/32 bit conversion the same performance as 8 bit conversion.

See the notes on `cpp_u16string_to_py_unicode16()` and `cpp_u32string_to_py_unicode32()` in `src/cpy/python_object_convert.cpp` for more information.

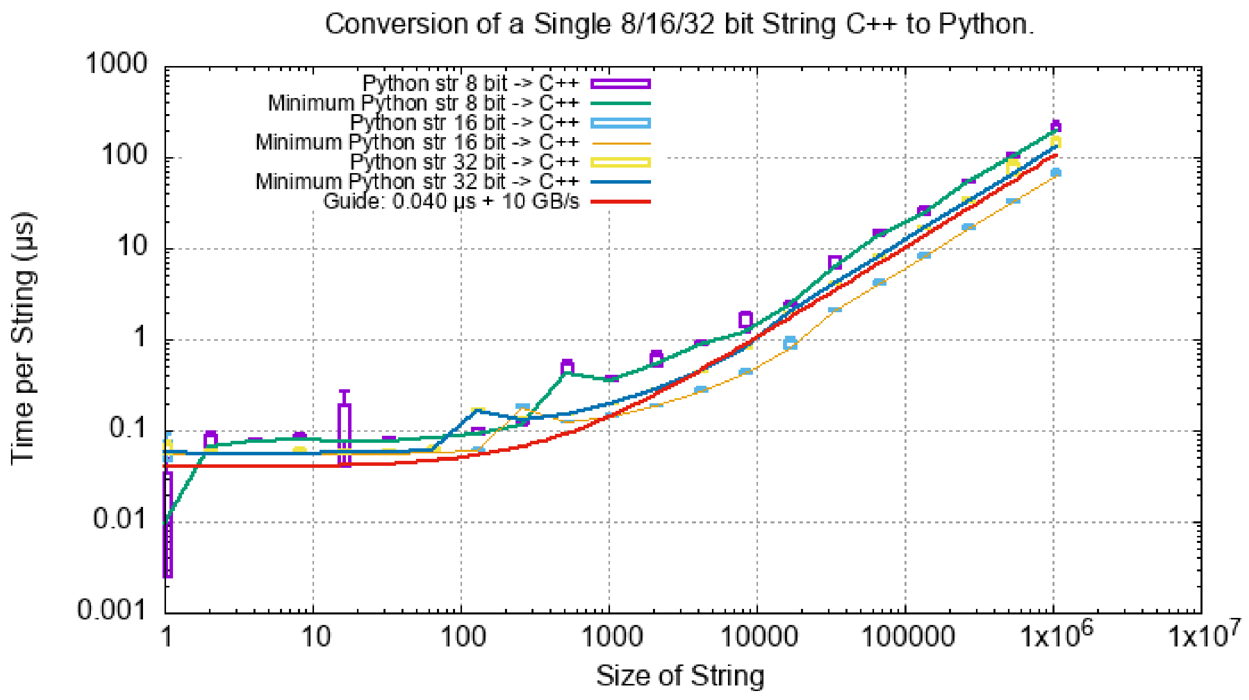
Back to 8/16/32 bit C++ to Python Strings

So now the 16/32 bit word strings are the same speed as 8 bit strings:





Or all three 8/16/32 bit plotted together:

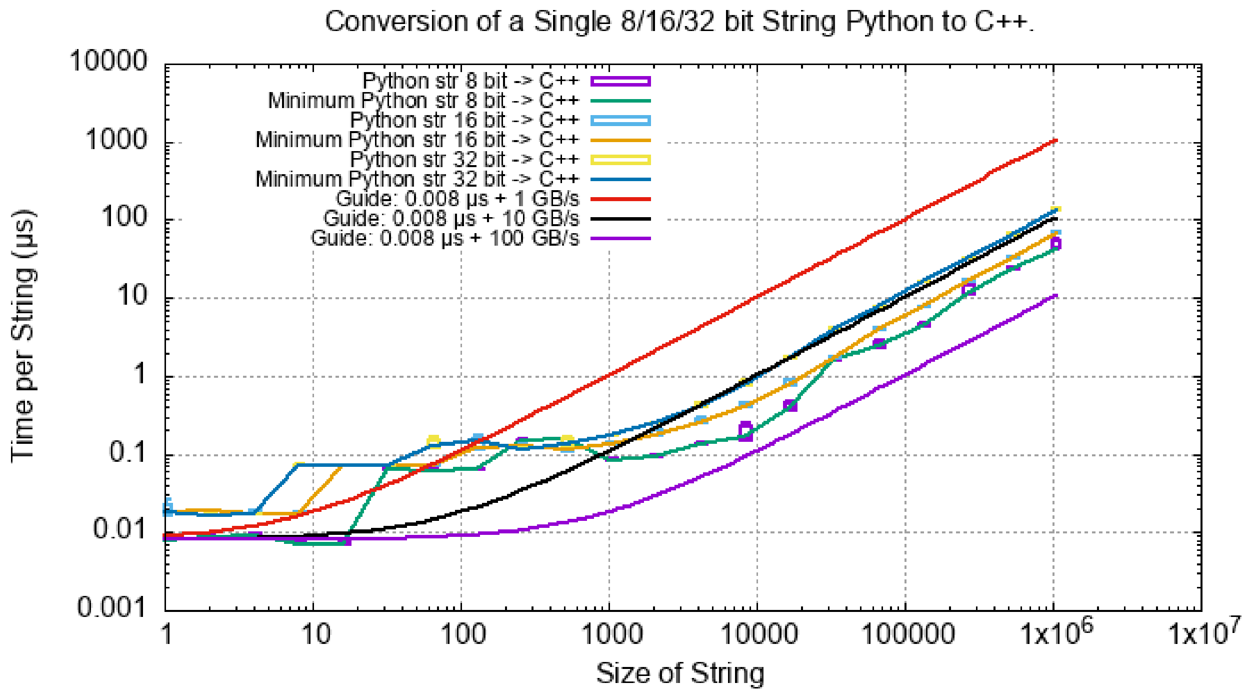


Python to C++:

Table 5: String Conversion Time Test Code, Python to C++.

Type C++	Test Function	Calls
`std::string`	`test_py_str_to_cpp_string_multiple()`	`py_unicode8_to_cpp_string()`.
`std::u16string`	`test_py_str16_to_cpp_u16string_multiple()`	`py_unicode16_to_cpp_u16string()`.
`std::u32string`	`test_py_str32_to_cpp_u32string_multiple()`	`py_unicode32_to_cpp_u32string()`.

And the plot of Python `str` to C++ `std::string`, `std::u16string` and `std::u32string`:



This is much more consistent, typically asymptotic to 10 GB/s. The conversion code does involve `memcpy()` (presumably). Here is an example from this library using 16 bit unicode characters:

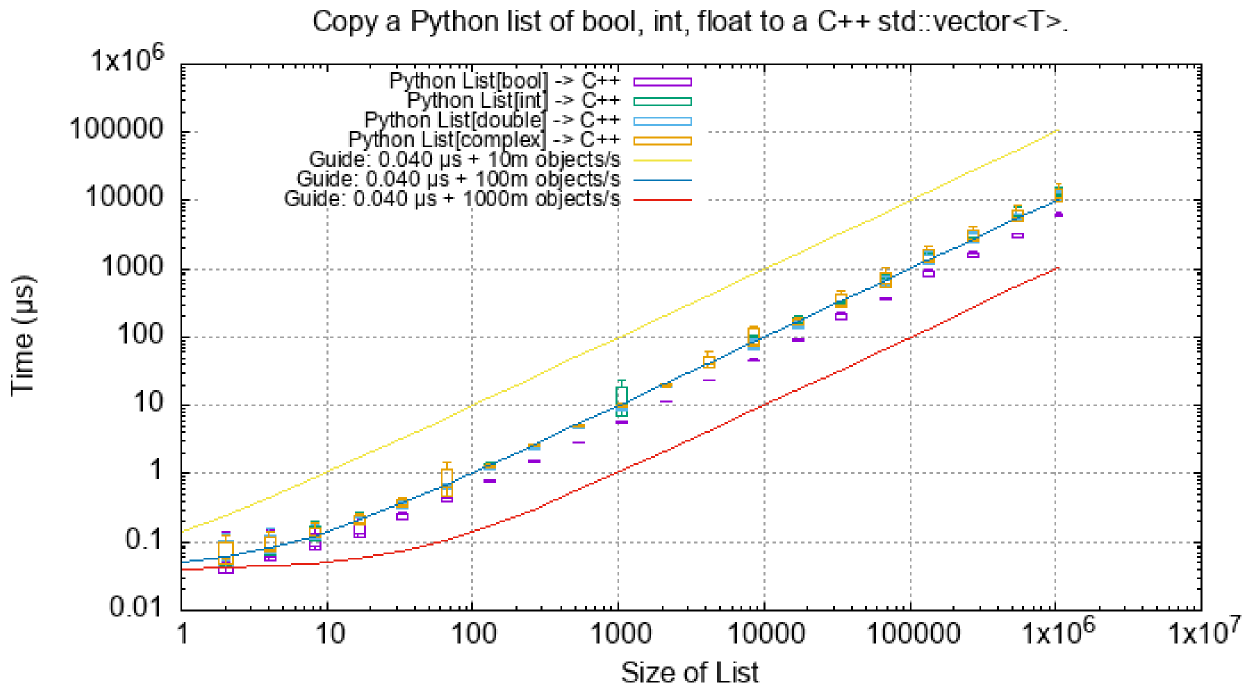
```
std::u16string py_unicode16_to_cpp_u16string(PyObject *op) {
    assert(! PyErr_Occurred());
    assert(op);
    assert(py_unicode16_check(op));
    std::u16string ret(
        (const char16_t *)PyUnicode_2BYTE_DATA(op), PyUnicode_GET_LENGTH(op)
    );
    return ret;
}
```

The conversion time of 10 GB/s is about thrice the time for `bytes` to an from a `std::vector<char>`. Presumably this is because of the complexities of the Unicode implementation.

7.2.3 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.

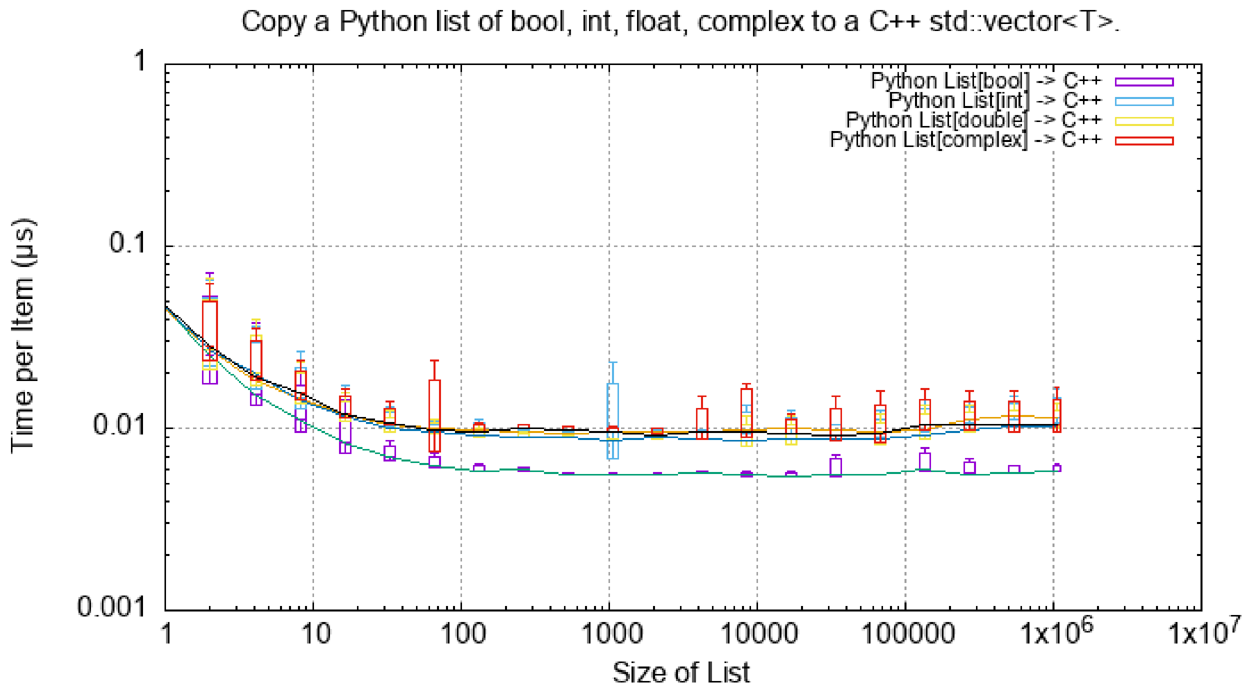
For example here is the total time to convert a list of `bool`, `int`, `float` and `complex` Python values to C++ for various list lengths:



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 same data plotted as a *rate of conversion* of a list of `bool`, `int`, `float` and `complex` Python values to C++ for various list lengths:



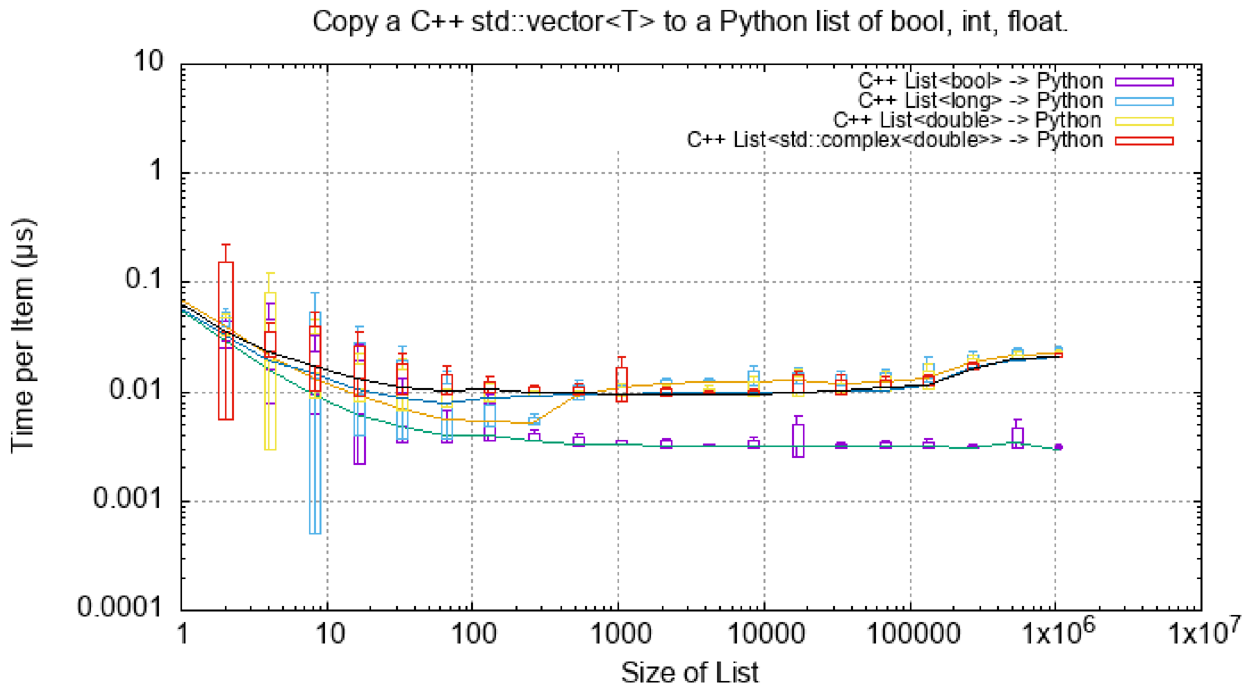
These rate plots are used for the rest of this section.

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 from C++ to Python.
- `bool` objects take around 0.007 µs per object.

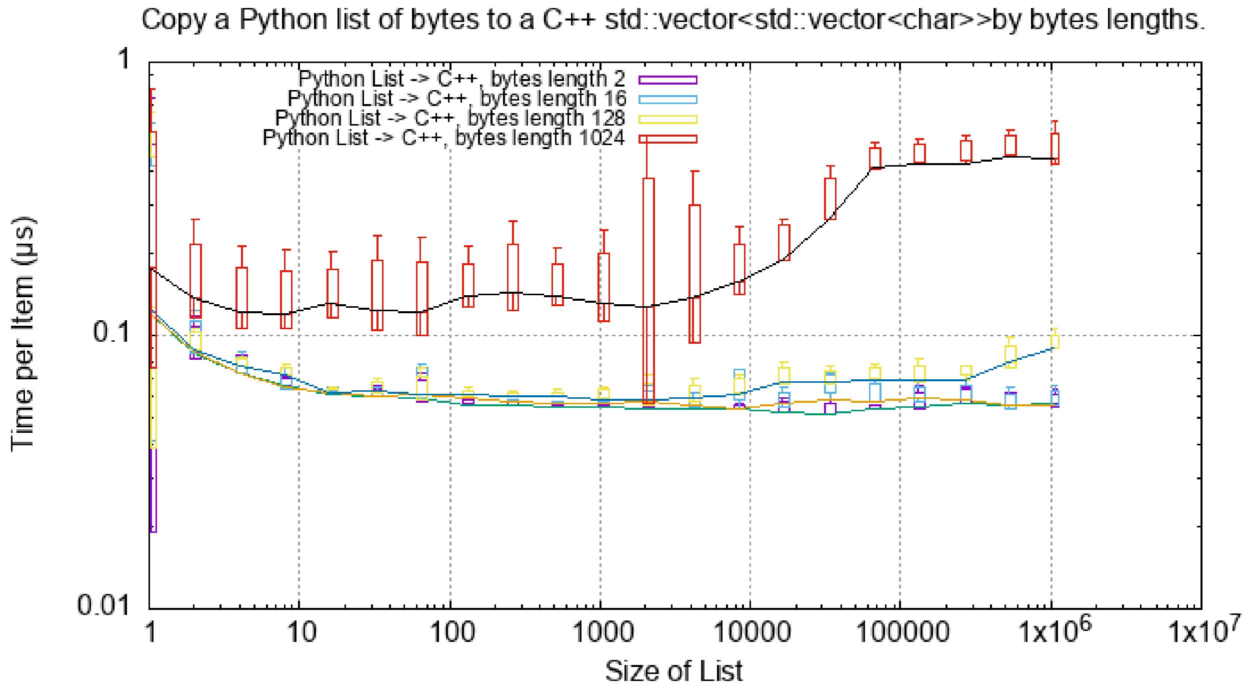
And the reverse converting a list of `bool`, `int`, `float` and `complex` from C++ to Python:



This is broadly symmetric with the Python to C++ performance except that `bool` values are twice as quick (typically 0.003 µs per object) compared with Python to C++.

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++.

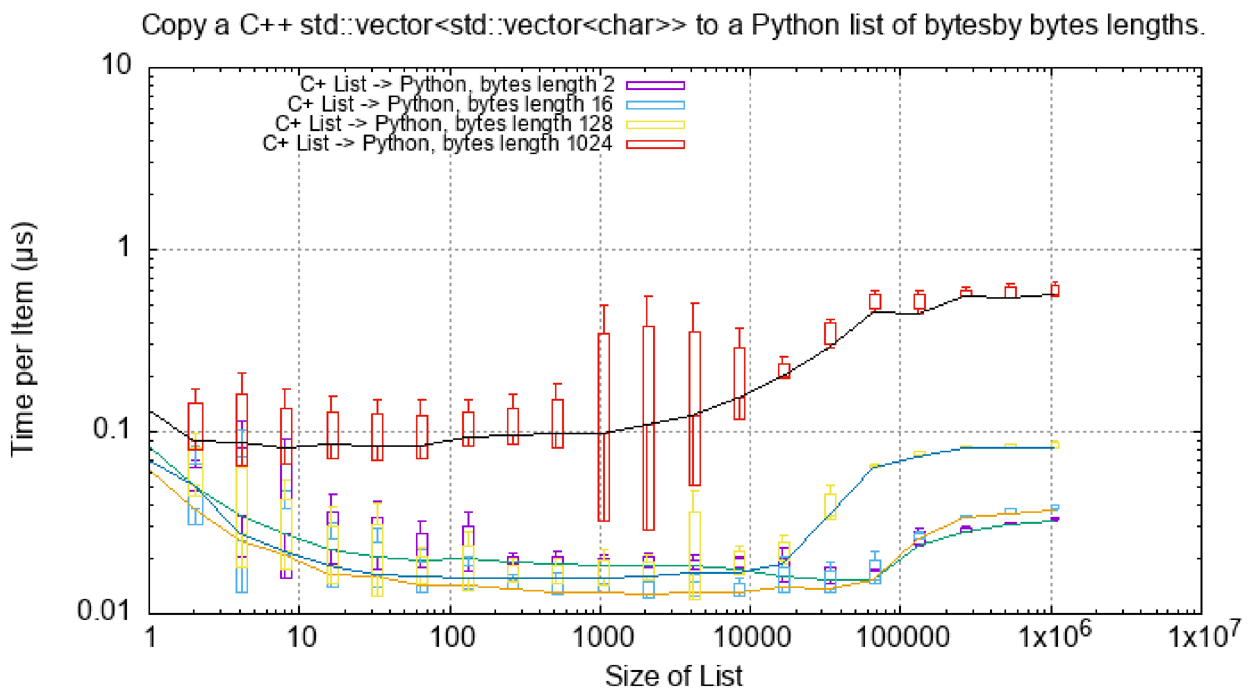


This graph shows a characteristic rise in rate for larger list lengths of larger objects. This is most likely because of memory contention issues with the larger, up to 1GB, containers. This characteristic is observed on most of the following plots, particularly with containers of `bytes` and `str`.

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`:

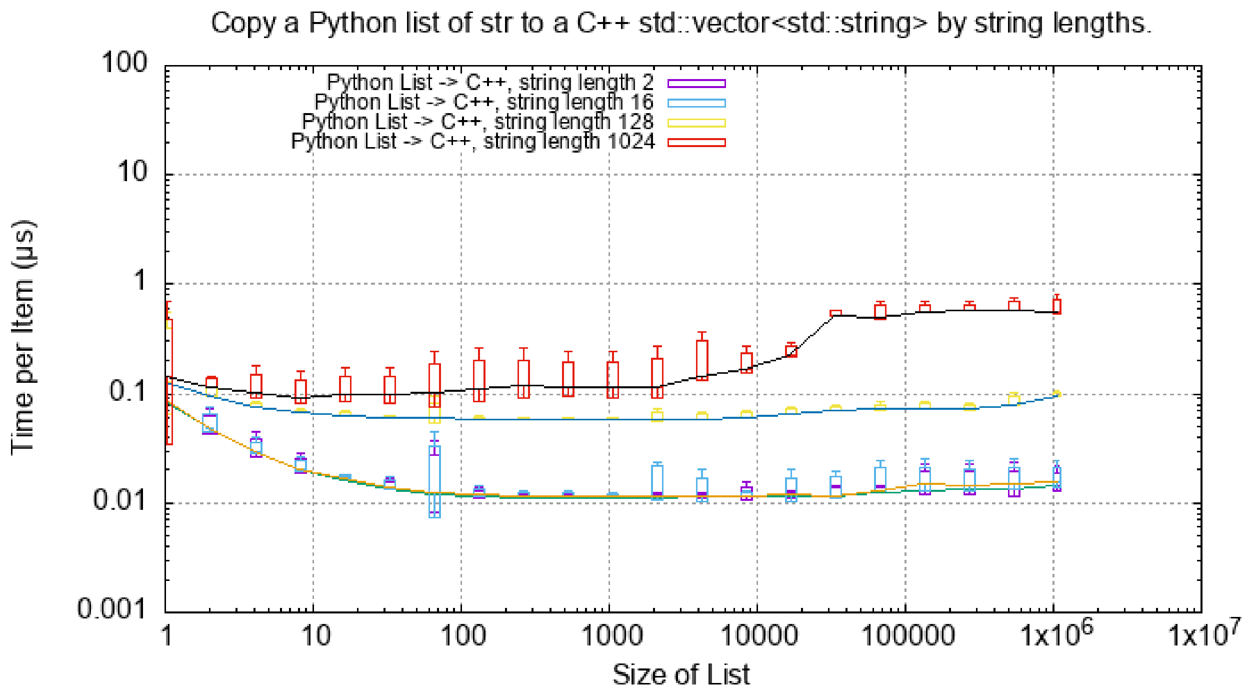


Object	~Time per object (µs)	Rate Mb/s	Notes
bytes[2]	0.015 to 0.03	67 to 133	
bytes[16]	0.015 to 0.04	400 to 133	
bytes[128]	0.02 to 0.09	1,400 to 6,400	
bytes[1024]	0.1 to 0.6	1,600 to 10,000	

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` [8 bit]

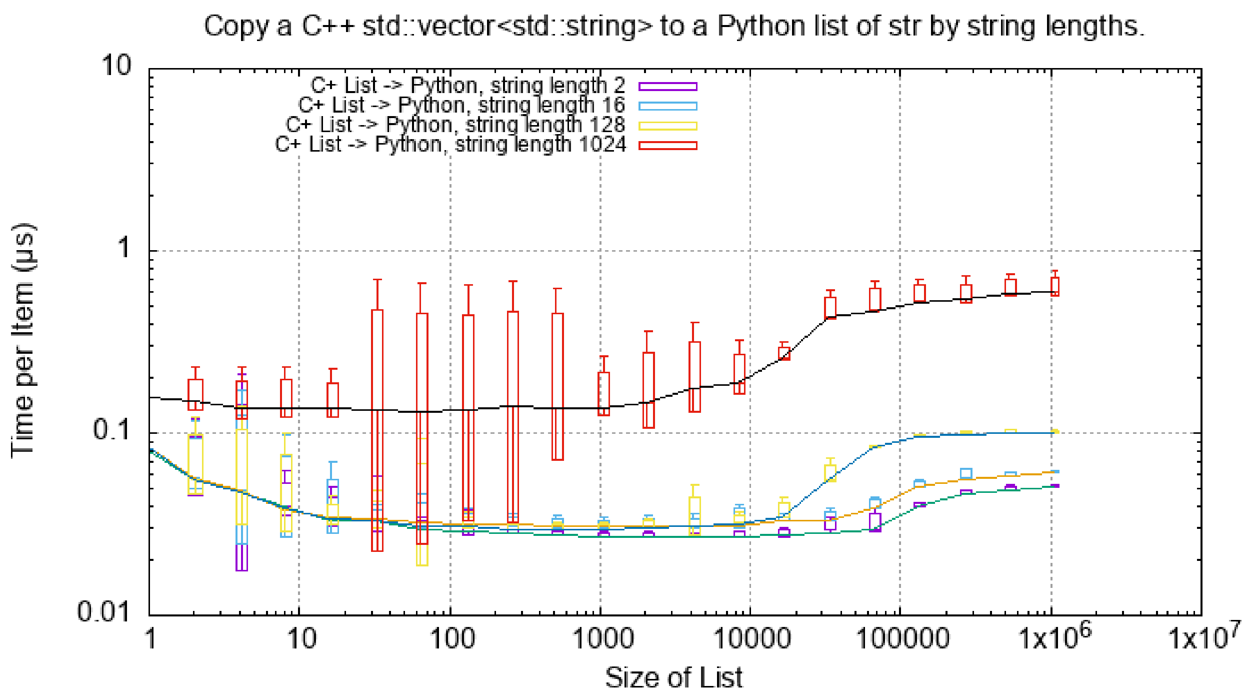
Similarly for converting a a Python list of `str` to and from a C++ `std::vector<std::string>`. First Python -> C++:



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]	0.08	1,600	
str[1024]	0.1 to 0.8	1,300 to 10,000	

And C++ -> Python:

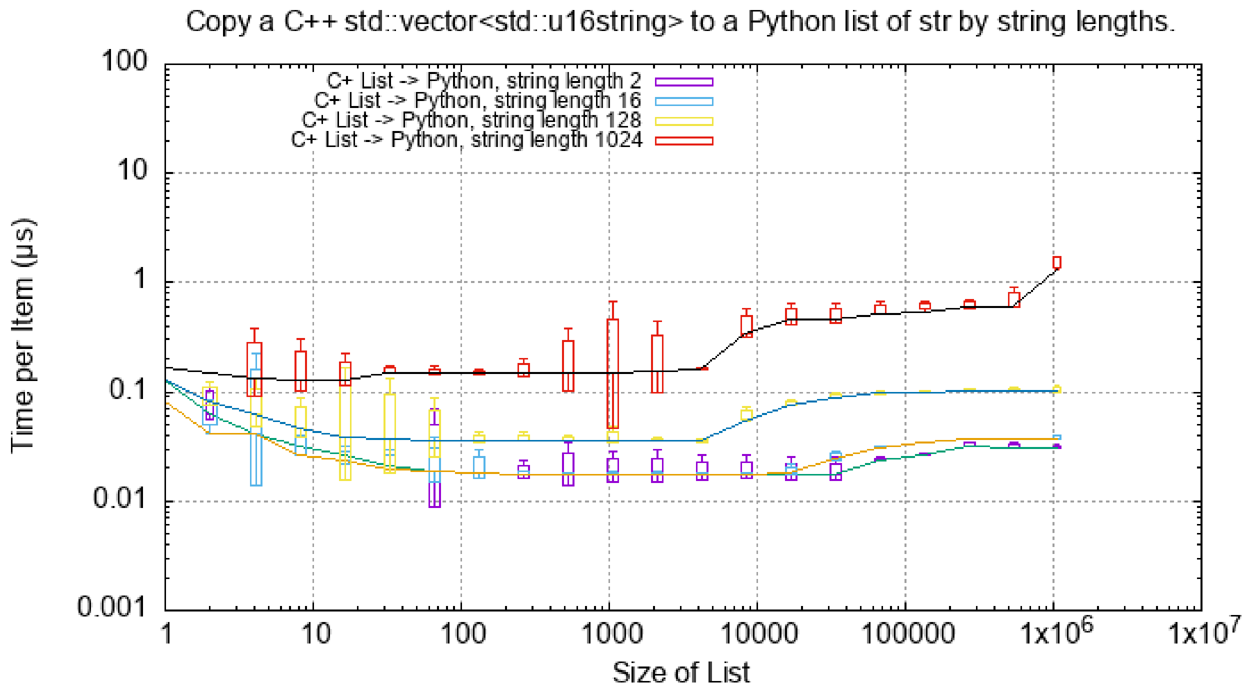


Object	~Time per object (µs)	Rate Mb/s	Notes
str[2]	0.03	70	
str[16]	0.03	500	
str[128]	0.03 to 0.1	1,300 to 4,000	
str[1024]	0.15 to 0.8	1,300 to 6,800	

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>>`

Lists of `str` [16 bit]

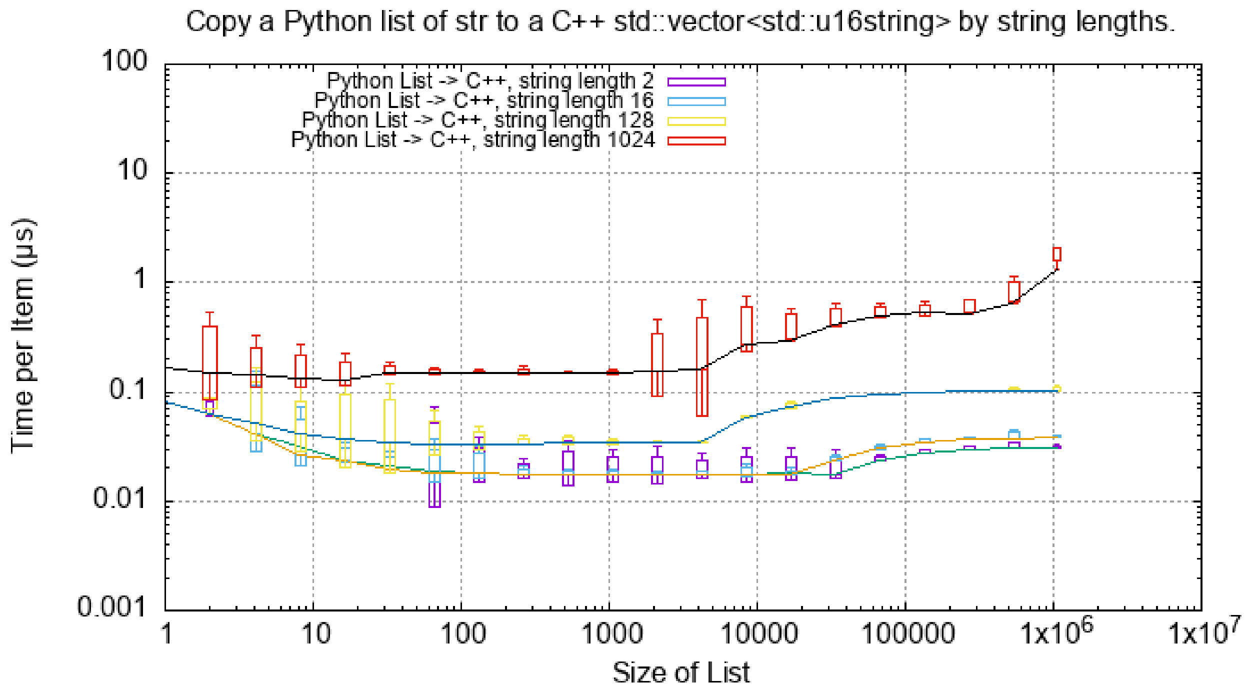
C++ to Python:



Object	~Time per object (µs)	Rate Mb/s	Notes
str[2]	0.03	70	
str[16]	0.1	160	
str[128]	0.9	140	
str[1024]	7	145	

This is about 100x slower than that for 8 bit strings which is aligned with the performance of *Strings*.

Python to C++:



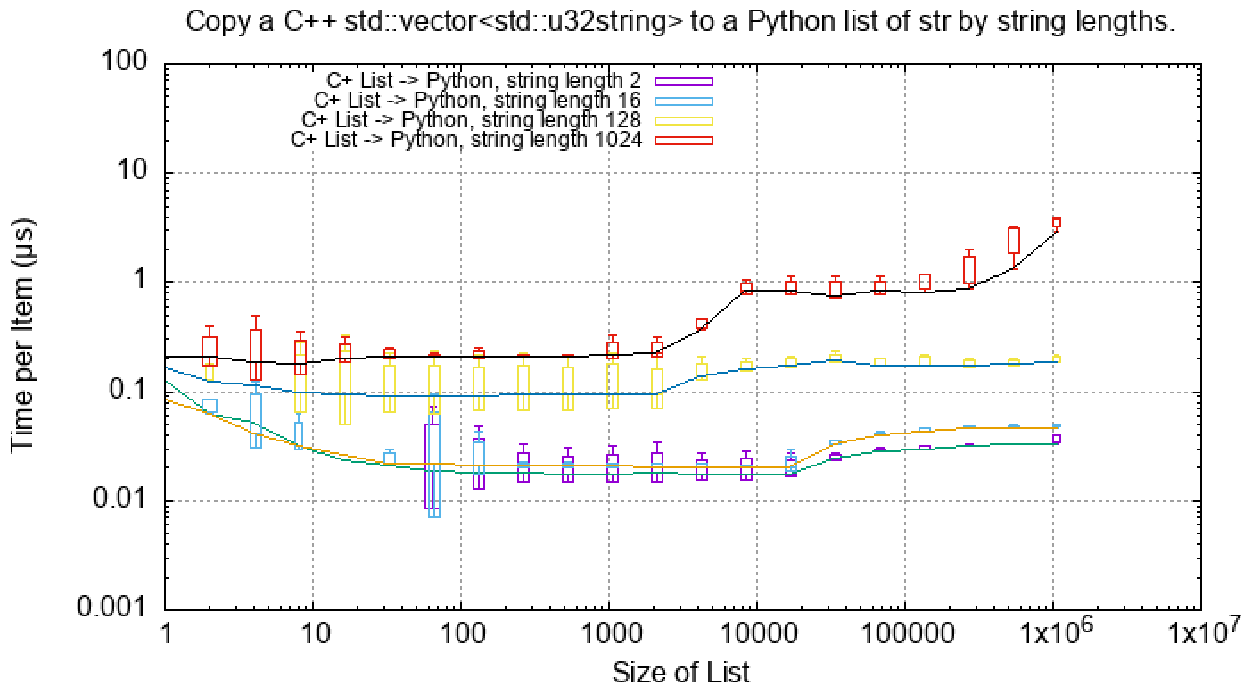
Object	~Time per object (µs)	Rate Mb/s	Notes
str[2]	0.03	70	
str[16]	0.1	160	
str[128]	0.9	140	
str[1024]	7	145	

This is symmetric performance with the C++ to Python conversion code.

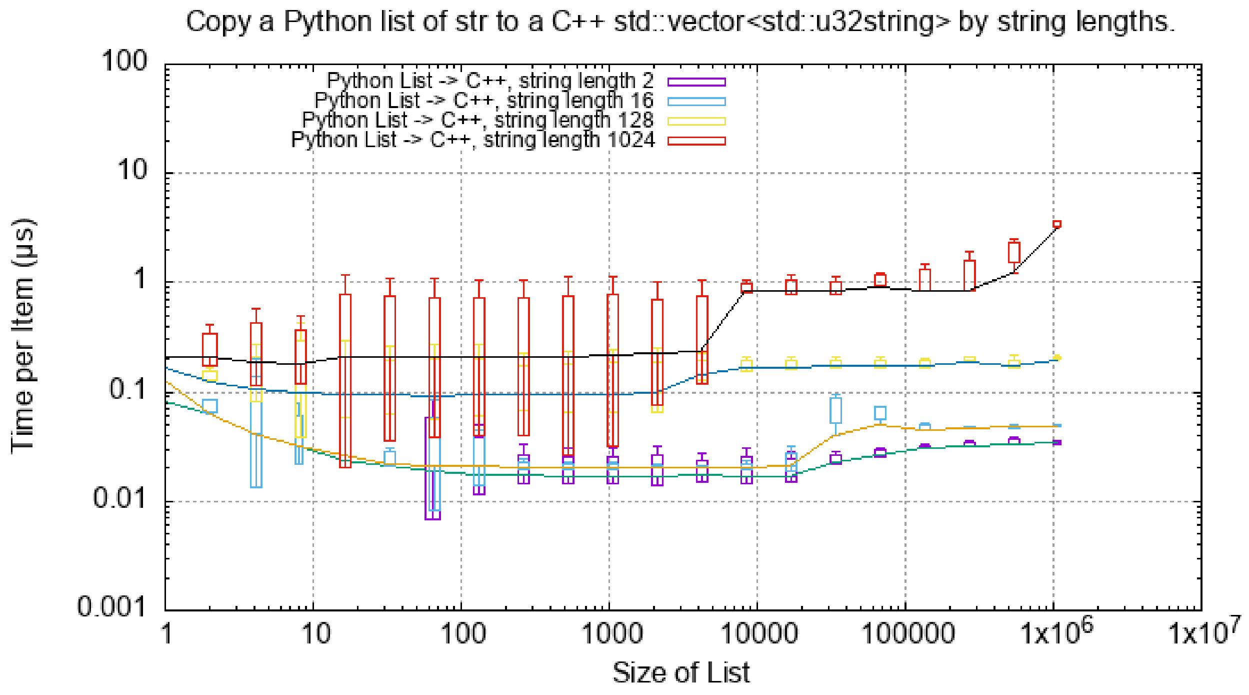
Lists of `str` [32 bit]

The performance is very close to that of 16 bit strings:

C++ to Python:


Copy a C++ std::vector<std::u32string> to a Python list of str by string lengths.
C+ List -> Python, string length 2
C+ List -> Python, string length 16
C+ List -> Python, string length 128
C+ List -> Python, string length 1024
Time per Item (µs)
Size of List
100
10
1
0.1
0.01
0.001
1
10
100
1000
10000
100000
1x10^6
1x10^7

Python to C++:


Copy a Python list of str to a C++ std::vector<std::u32string> by string lengths.
Python List -> C++, string length 2
Python List -> C++, string length 16
Python List -> C++, string length 128
Python List -> C++, string length 1024
Time per Item (µs)
Size of List
100
10
1
0.1
0.01
0.001
1
10
100
1000
10000
100000
1x10^6
1x10^7

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

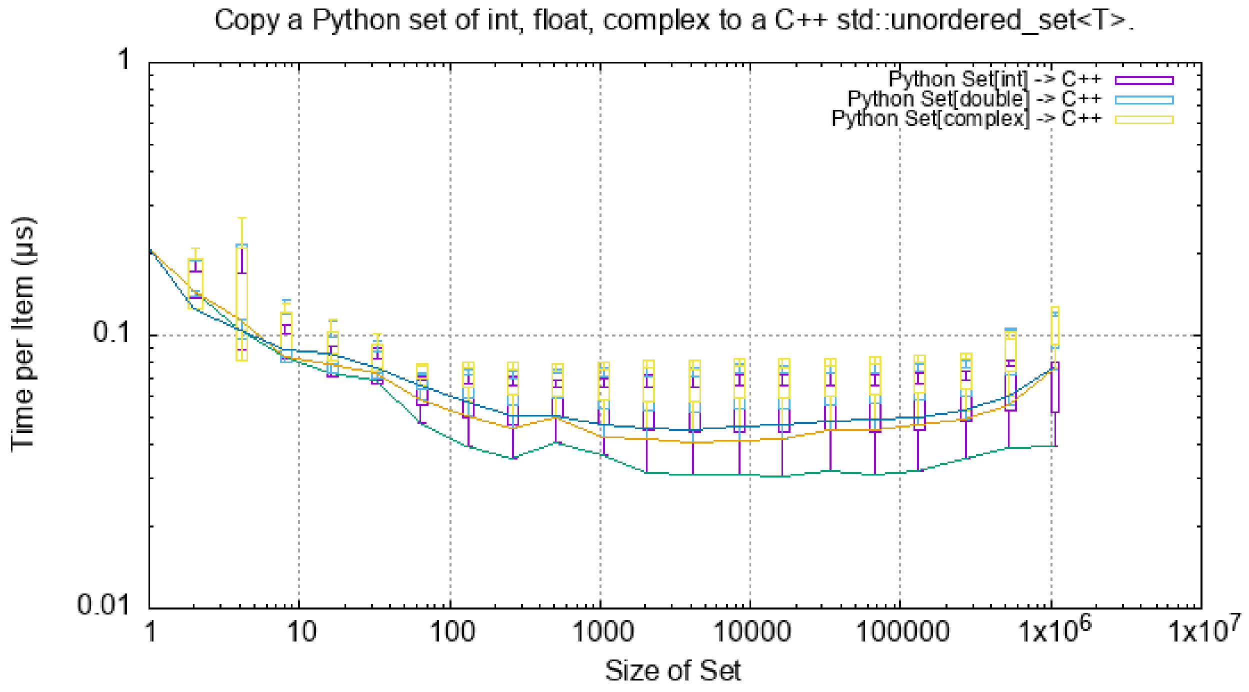
This is near identical to the performance of a list for:

- 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.

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

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

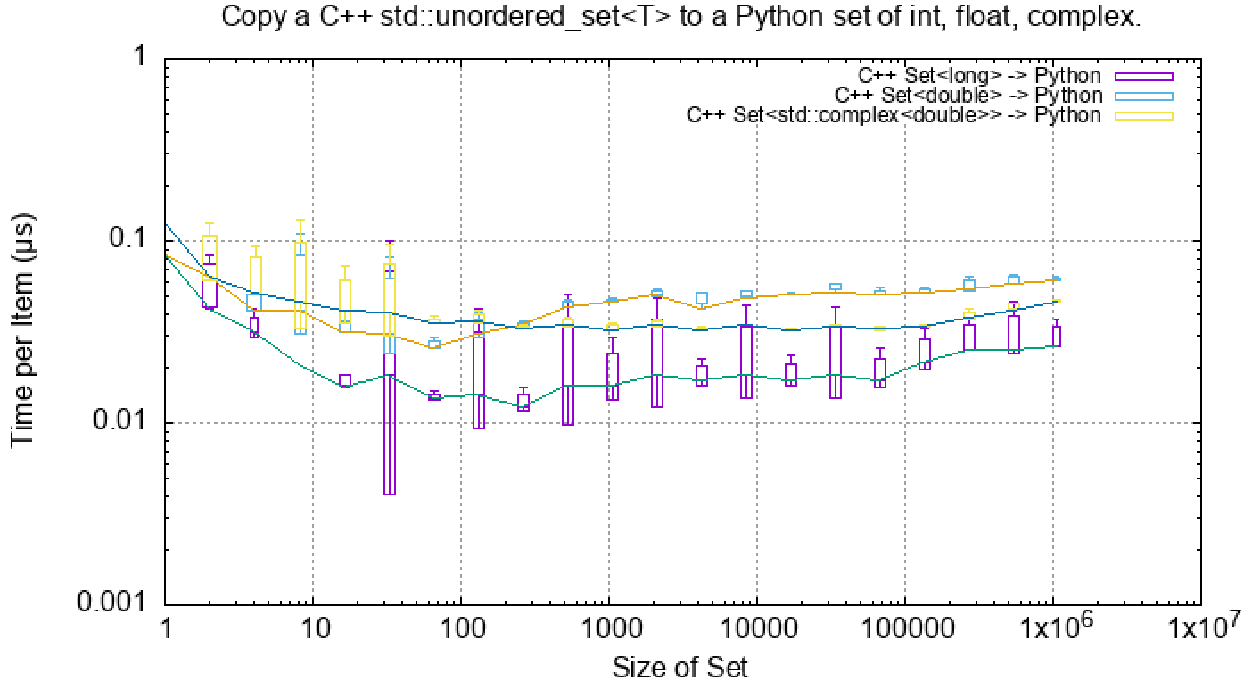
Here is the rate graph for converting a Python `set` to C++ `std::unordered_set<T>` for Python `int`, `float` and `complex` objects:



Object	set (µs)	list (µs)	Ratio	Notes
int	0.03	0.01	3x	
double	0.05	0.01	5x	
complex	0.05	0.01	5x	

The cost of insertion is O(N) for both list and set but due to the hashing heeded for the set it is about 3x to 5x slower.

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

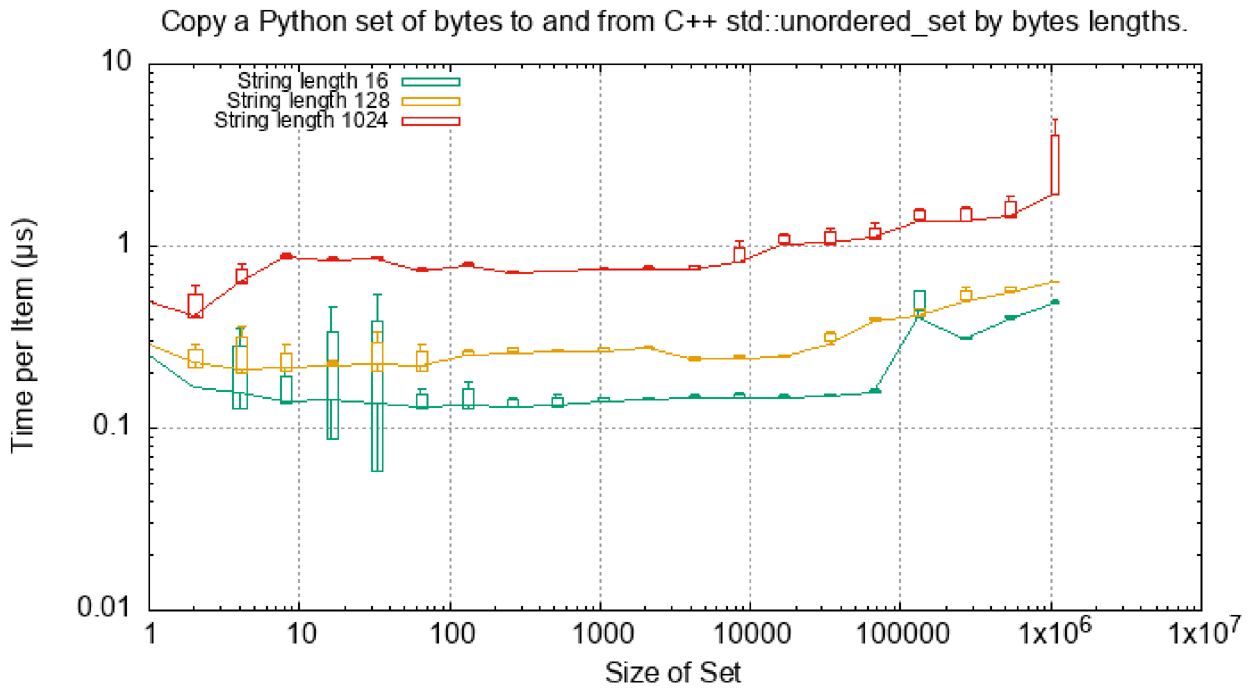


The conversion and insertion of C++ to Python is significantly faster that from Python to C++. Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
int	0.02	0.01	2x	
double	0.06	0.01	6x	
complex	0.04	0.01	4x	

Set of `bytes`

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

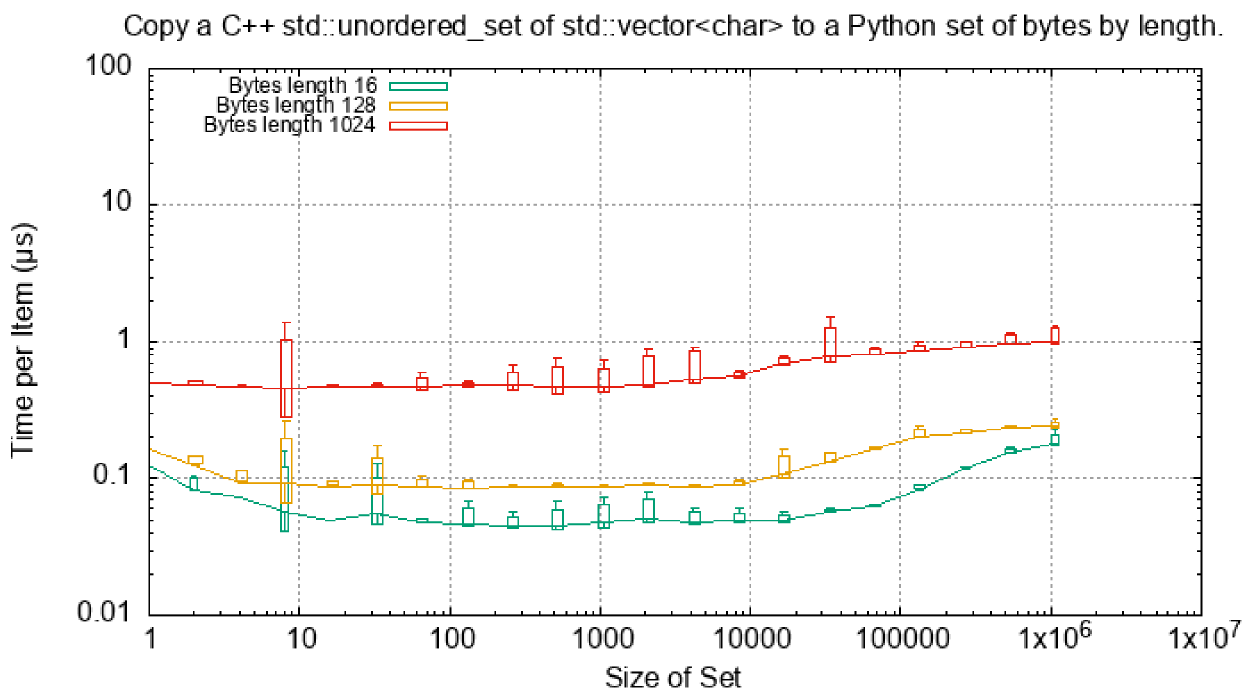


Object	~Time per object (µs)	Rate Mb/s	Notes
bytes[16]	0.2	80	
bytes[128]	0.3	400	
bytes[1024]	1.0	1,000	

Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
bytes[16]	0.2	0.06	3x	
bytes[128]	0.3	0.06	5x	
bytes[1024]	1.0	0.15 to 0.4	x2.5 to 6x	

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



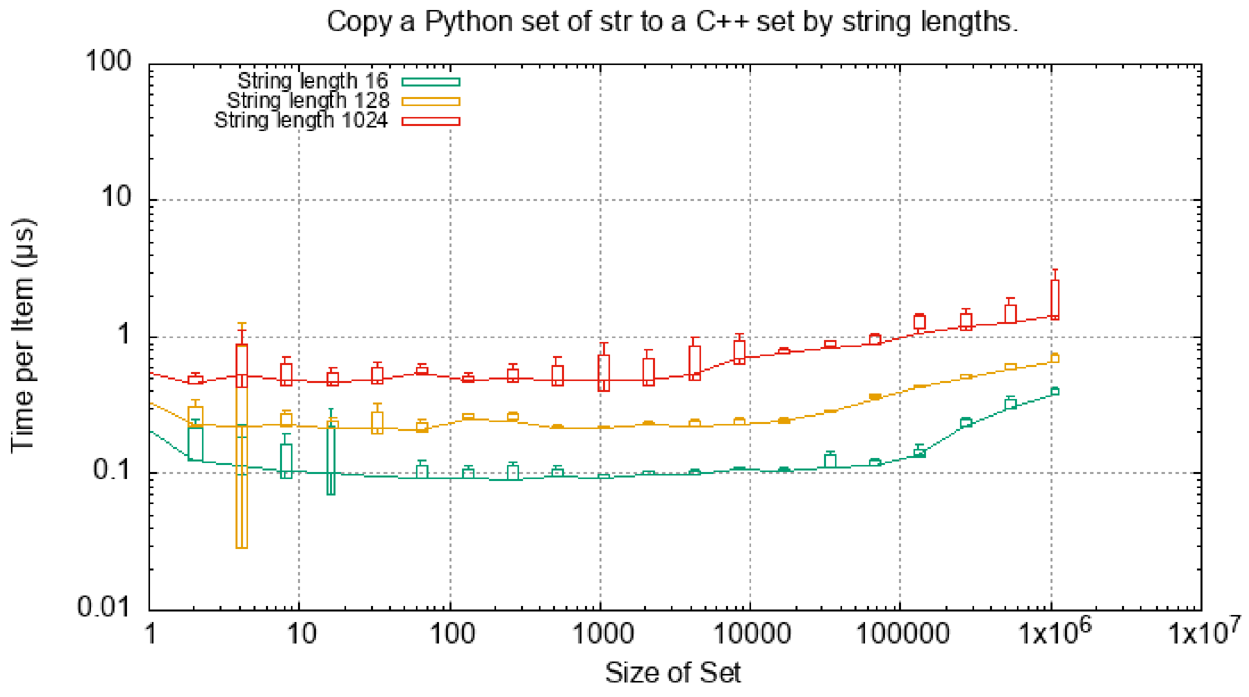
Object	~Time per object (µs)	Rate Mb/s	Notes
bytes[16]	0.05	320	
bytes[128]	0.1	1,280	
bytes[1024]	0.8	1,300	

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	1x to 3x	
bytes[128]	0.1	0.02 to 0.09	1x to 5x	
bytes[1024]	0.8	0.1 to 0.6	1.25x to x8	

Set of `str` (8 bit)

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

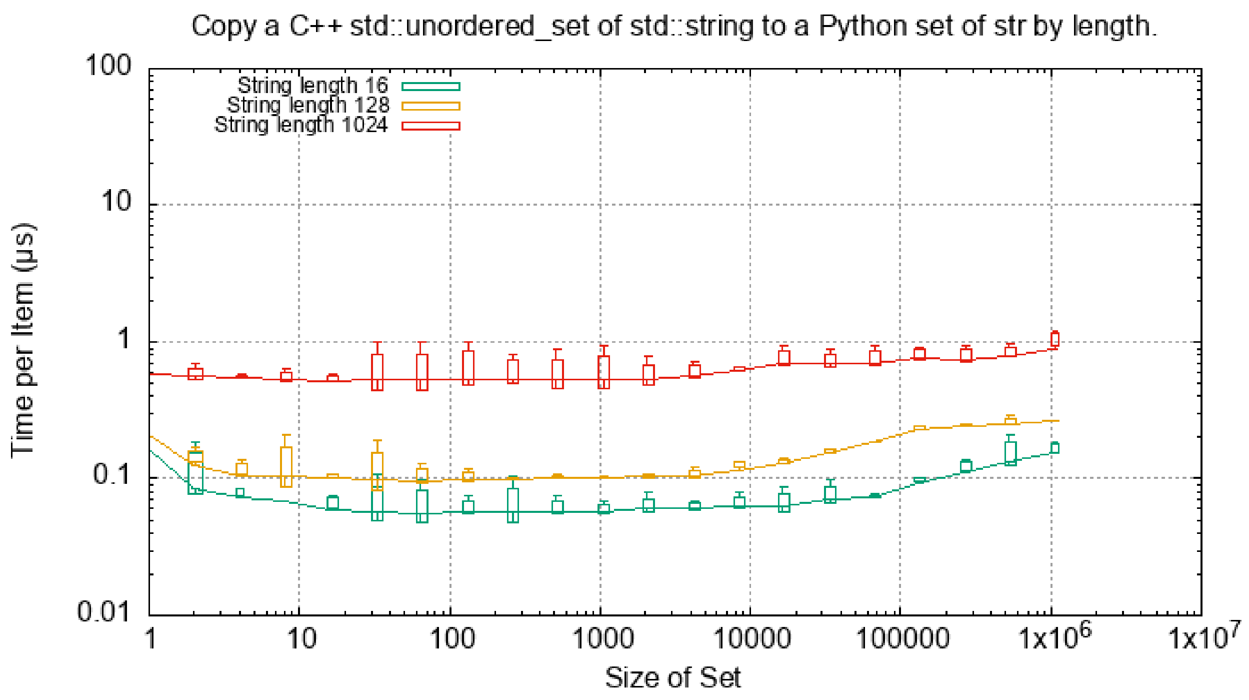


Object	~Time per object (µs)	Rate Mb/s	Notes
string[16]	0.1	160	
string[128]	0.2	640	
string[1024]	0.7 to 1.0	1,000 to 1,500	

Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
string[16]	0.1	0.01	10x	
string[128]	0.2	0.08	2.5x	
string[1024]	0.7 to 1.0	0.1 to 0.8	~8x	

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
string[16]	0.08	200	
string[128]	0.1	1,300	
string[1024	0.8	1,300	

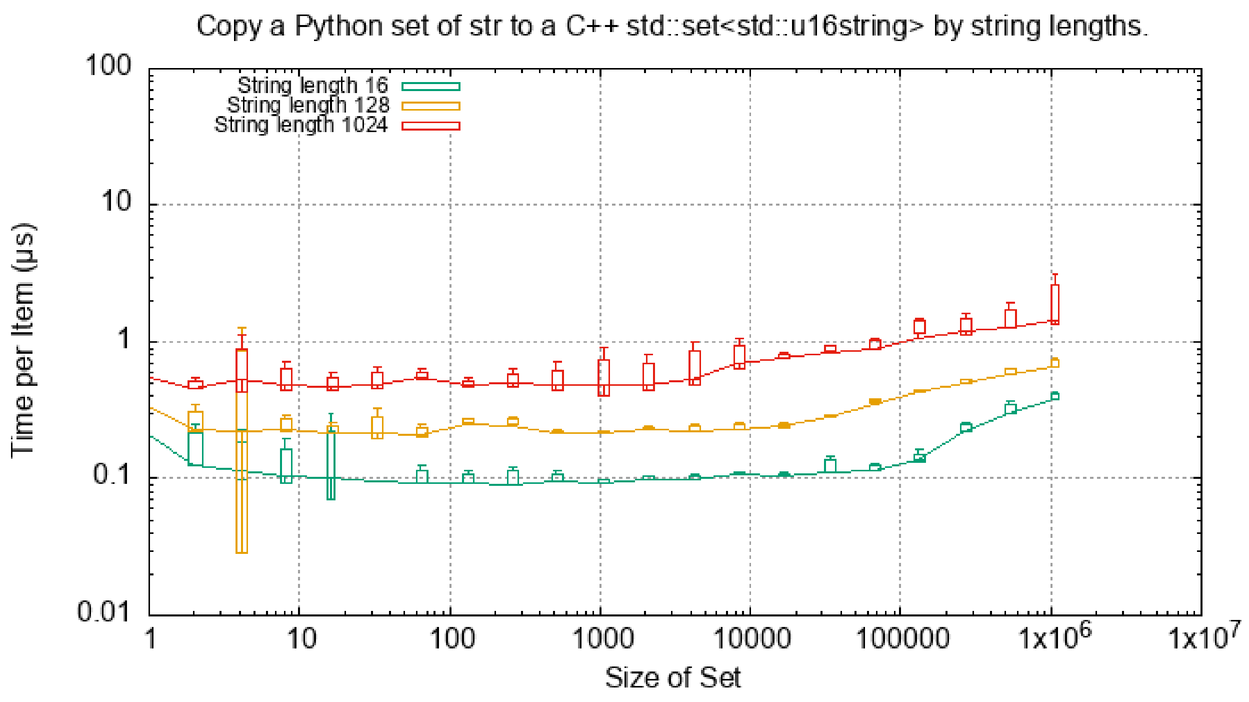
Here is the time per object compared with a list:

Object	set (µs)	list (µs)	Ratio	Notes
string[16]	0.08	0.03	3x	
string[128]	0.1	0.03 to 0.1	1x to 3x	
string[1024]	0.8	0.15 to 0.8	1x to 5x	

Set of `str` (16 bit)

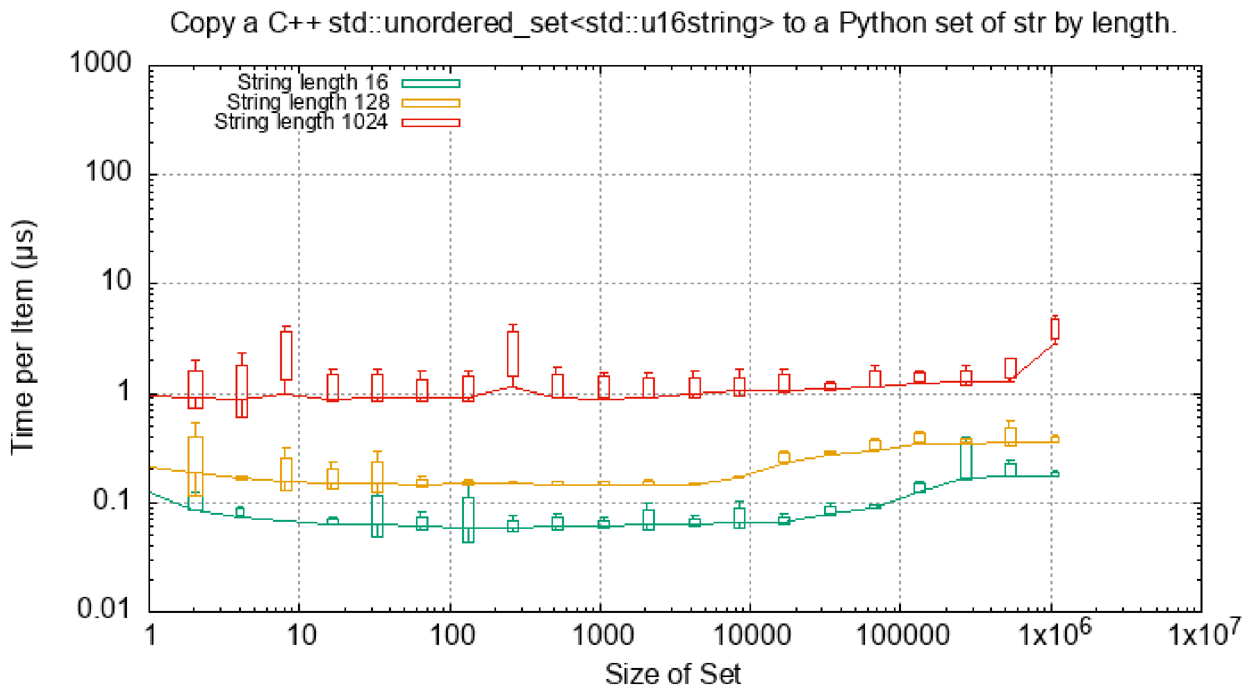
Here is the rate graph for converting a Python `set` of `str` to C++ `std::unordered_set<std::u16string>`:

Python to C++:



This is pretty much comparable with the 8 bit string conversion from Python to C++.

And the reverse, from C++ to Python:



Because of the issues identified in *Strings* string16 conversion of `std::u16string` and `std::u32string` to Python `str` is around 100 times slower than for 8 bit strings.

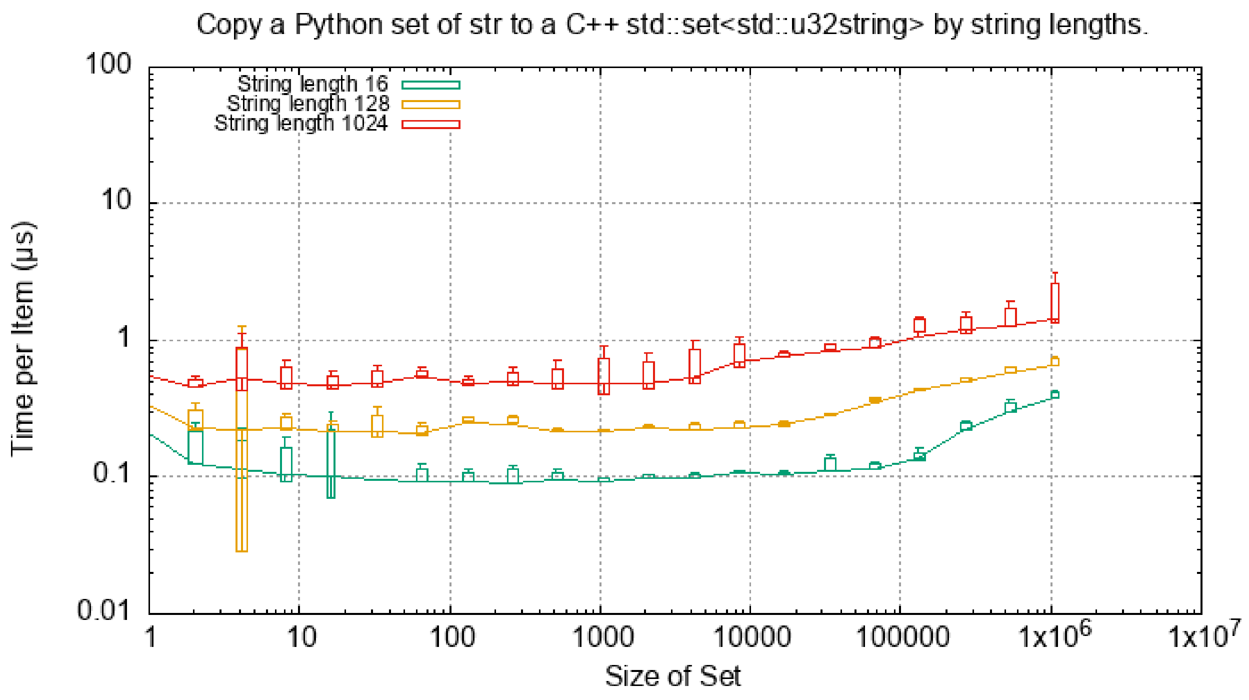
Here is the comparison with 8 bit strings:

Object	8 bit (µs)	16 bit (µs)	Ratio	Notes
string[16]	0.08	0.15	2x	
string[128]	0.1	1	10x	
string[1024]	0.8	8	10x	

Set of `str` (32 bit)

Here is the rate graph for converting a Python `set` of `str` to C++ `std::unordered_set<std::u32string>`:

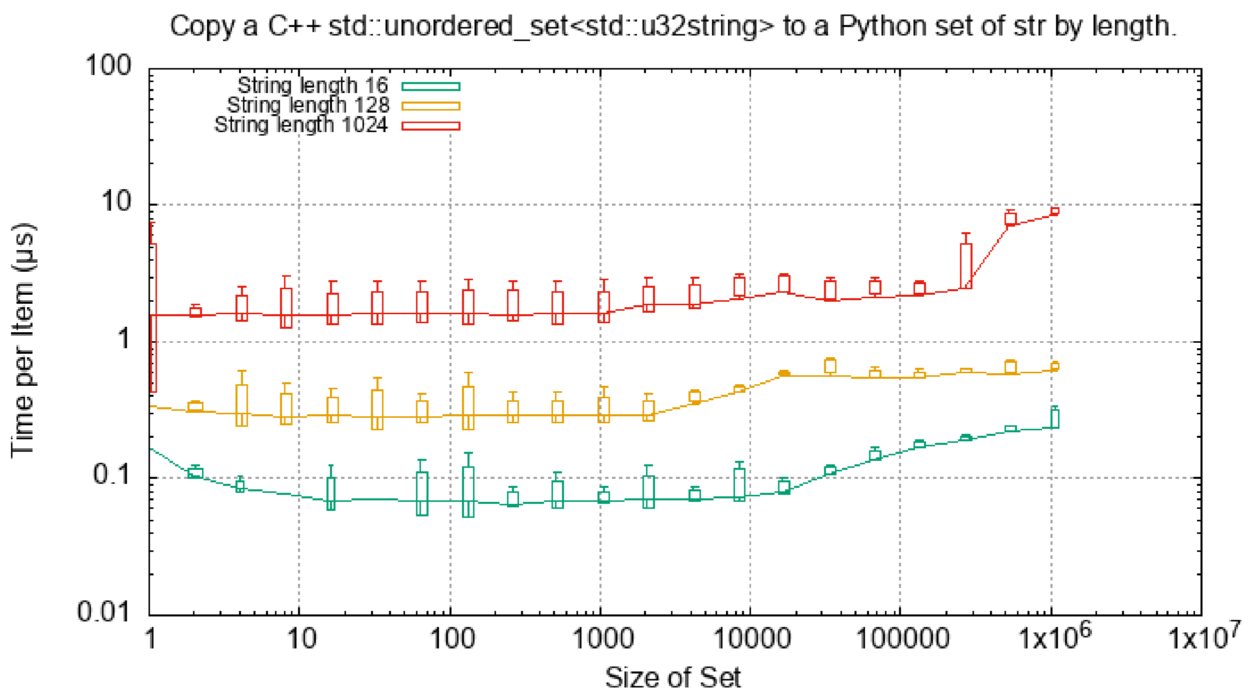
Python to C++:



This is pretty much comparable with the 8 and 16 bit string conversion from Python to C++.

And the reverse, from C++ to Python:

C++ to Python:



This is essentially the same as for 16 bit strings.

7.2.6 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	

7.2.7 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 C++ to Python strings with word sizes of 16 and 32 bits is typically 10x to 100x than for 8 bits because of the Python C-API. Conversion from Python to C++ is pretty much identical for all string word sizes of 8/16/32 bits.

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.

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

7.3.1 Python Objects

This shows the performance of converting a Python single object to its C++ equivalent and back again.

Python Bytes

The Python test creates a Python bytes object of specified length(s), converts it to a C++ `std::vector<char>` and then back to a Python bytes object all using this C++ library code.

The test code looks like this:

```python
@pytest.mark.slow
def test_new_bytes():
    results = []
    proc = psutil.Process()
    rss = proc.memory_info().rss
    for size in SIZE_DOUBLING:
        original = b' ' * size
        timer = TimedResults()
        for _r in range(REPEAT):
            time_start = time.perf_counter()
            # See below for this function.
            cPyCppContainers.new_bytes(original)
            time_exec = time.perf_counter() - time_start
            timer.add(time_exec)
        results.append((size, timer))
    # pprint.pprint(results)
    print()
    print('test_new_bytes()')
    rss_new = proc.memory_info().rss
    print(f'RSS was {rss:,d} now {rss_new:,d} diff: {rss_new - rss:+,d}')
    print(f'{"Size":<8s} {results[0][1].str_header():s} {"Min/Size e9":>12s}')
    for s, t in results:
        print(f'{s:<8d} {t} {1e9 * t.min() / s:12.1f}')
```

And the C/C++ code that provides the function `cPyCppContainers.new_bytes()`:

```
/**
 * Take a Python \c bytes object, convert it to a \c std::vector<char> then convert that␣
↪back to a Python \c bytes
 * object.
 */
static PyObject *
new_bytes(PyObject *Py_UNUSED(module), PyObject *arg) {
    assert(! PyErr_Occurred());
    PyObject *ret = NULL;
    if (py_bytes_check(arg)) {
        std::vector<char> vec = py_bytes_to_cpp_vector_char(arg);
        ret = cpp_vector_char_to_py_bytes(vec);
    } else {
        PyErr_Format(
            PyExc_ValueError,
            "new_bytes() argument is not bytes but type %s", arg->ob_type->tp_name
        );
        ret = NULL;
    }
    assert(ret || PyErr_Occurred());
    return ret;
}
```

And the output looks like this:

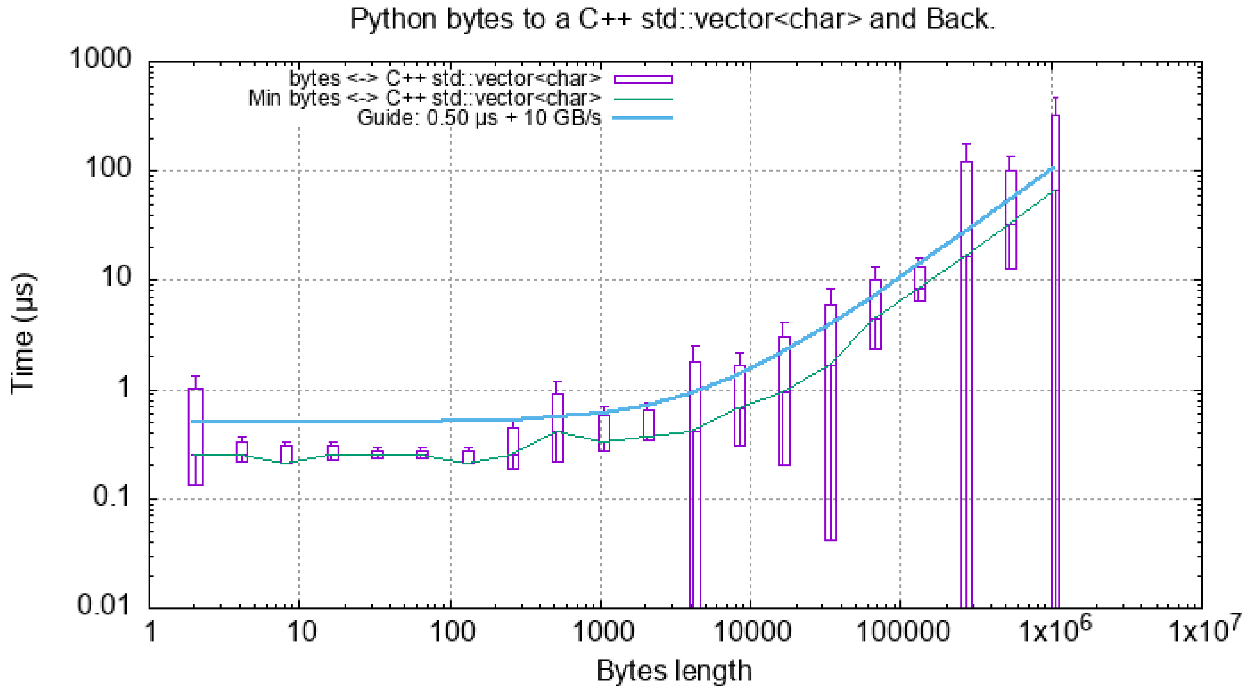
```
test_new_bytes()
RSS was 55,828,480 now 58,531,840 diff: +2,703,360
Size       Count           Min          Mean        Median      Std.Dev.           Max     Max/Min ↵
↪Min/Size e9
2              5   0.000000235   0.000002002   0.000000322   0.000003841   0.000008873      37.757 ↵
↪    117.5
4              5   0.000000253   0.000000381   0.000000277   0.000000168   0.000000614       2.427 ↵
↪     63.3
8              5   0.000000253   0.000000276   0.000000258   0.000000044   0.000000355       1.403 ↵
↪     31.6
16             5   0.000000251   0.000000269   0.000000255   0.000000028   0.000000317       1.263 ↵
↪     15.7
32             5   0.000000254   0.000000267   0.000000256   0.000000024   0.000000310       1.220 ↵
↪      7.9
64             5   0.000000255   0.000000314   0.000000258   0.000000128   0.000000544       2.133 ↵
↪      4.0
128            5   0.000000253   0.000000359   0.000000270   0.000000199   0.000000714       2.822 ↵
↪      2.0
256            5   0.000000257   0.000000385   0.000000271   0.000000255   0.000000839       3.265 ↵
↪      1.0
512            5   0.000000260   0.000000421   0.000000264   0.000000344   0.000001035       3.981 ↵
↪      0.5
1024           5   0.000000462   0.000000788   0.000000492   0.000000563   0.000001774       3.840 ↵
↪      0.5
2048           5   0.000000365   0.000000803   0.000000455   0.000000783   0.000002197       6.019 ↵
↪      0.2
4096           5   0.000000407   0.000000593   0.000000569   0.000000225   0.000000957       2.351 ↵
↪      0.1
8192           5   0.000000528   0.000000708   0.000000620   0.000000267   0.000001180       2.235 ↵
↪      0.1
16384          5   0.000000729   0.000002129   0.000000794   0.000002982   0.000007462      10.236 ↵
↪      0.0
32768          5   0.000001100   0.000004238   0.000001217   0.000006875   0.000016537      15.034 ↵
↪      0.0
65536          5   0.000002512   0.000003666   0.000002617   0.000002208   0.000007598       3.025 ↵
↪      0.0
131072         5   0.000004827   0.000006307   0.000005193   0.000002674   0.000011073       2.294 ↵
↪      0.0
```

(continues on next page)

(continued from previous page)

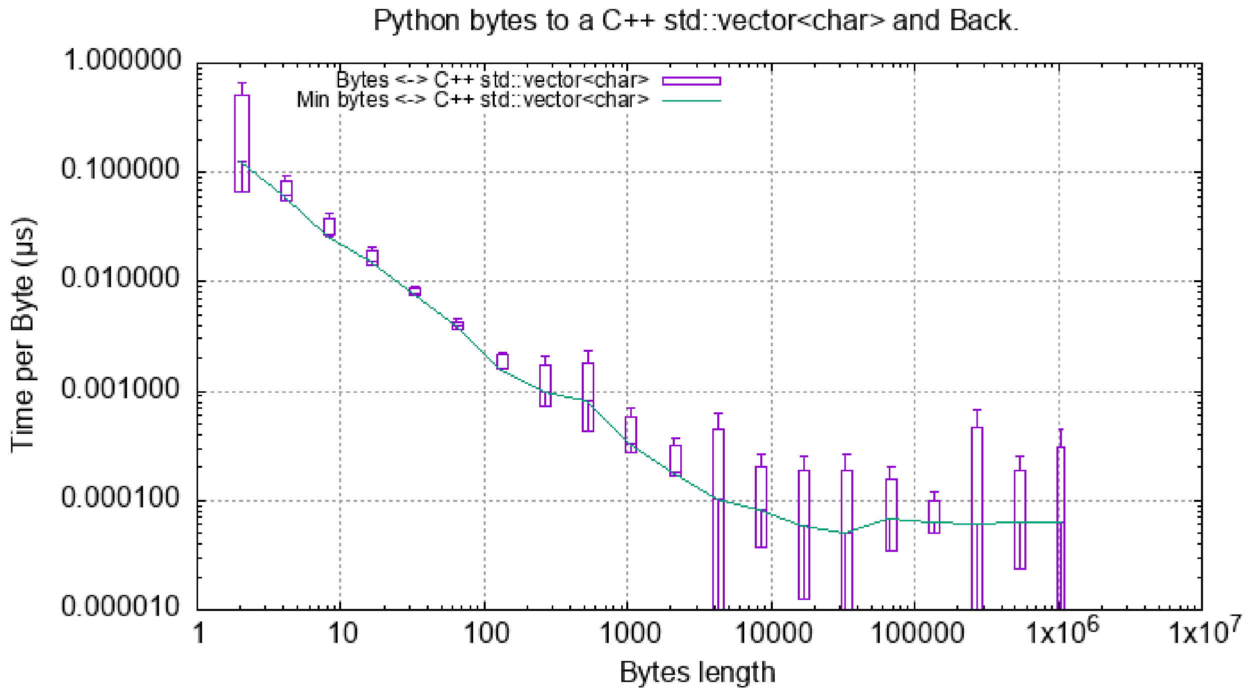
```
262144                  5       0.000009739       0.000045346       0.000009934       0.000077901       0.000184683         18.963  ␣
↪       0.0
524288                  5       0.000017699       0.000066024       0.000017893       0.000107562       0.000258436         14.602  ␣
↪       0.0
1048576                 5       0.000034801       0.000130087       0.000035806       0.000207265       0.000500793         14.390  ␣
↪       0.0
```

When plotted in time the performance looks like this:



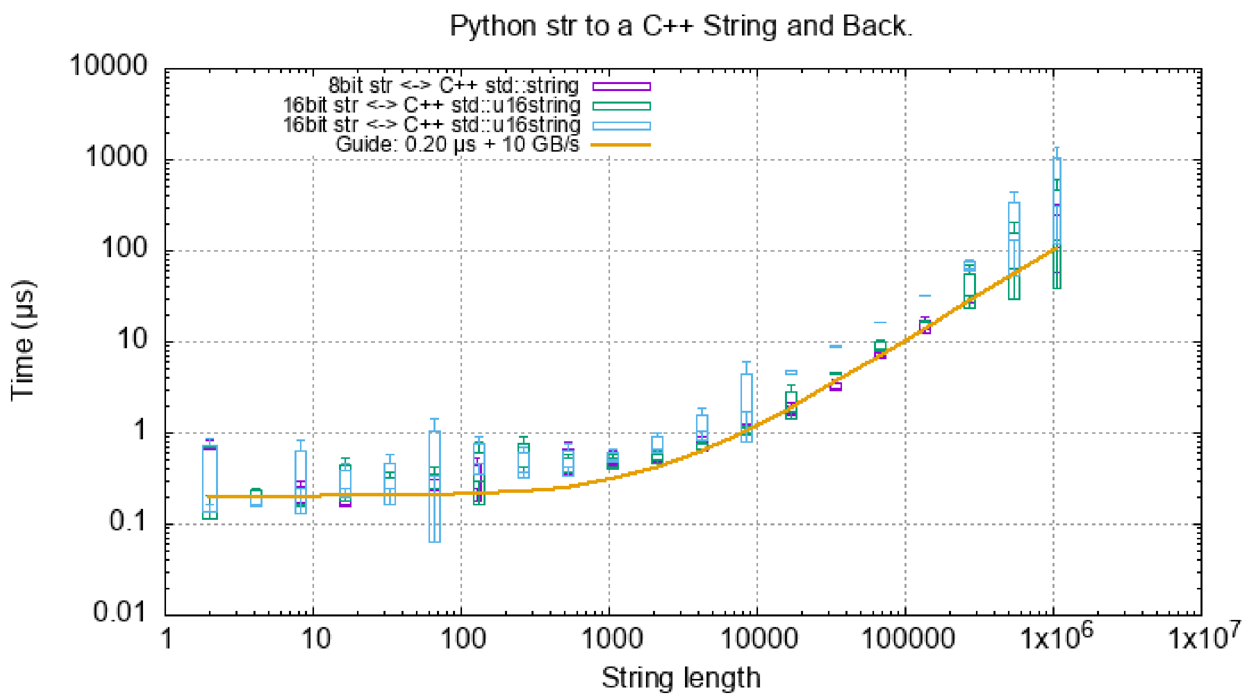
This is asymptotic to around 5 GB/s round trip conversion time.

The rate plot, that is the time value divided by the length of the bytes is:

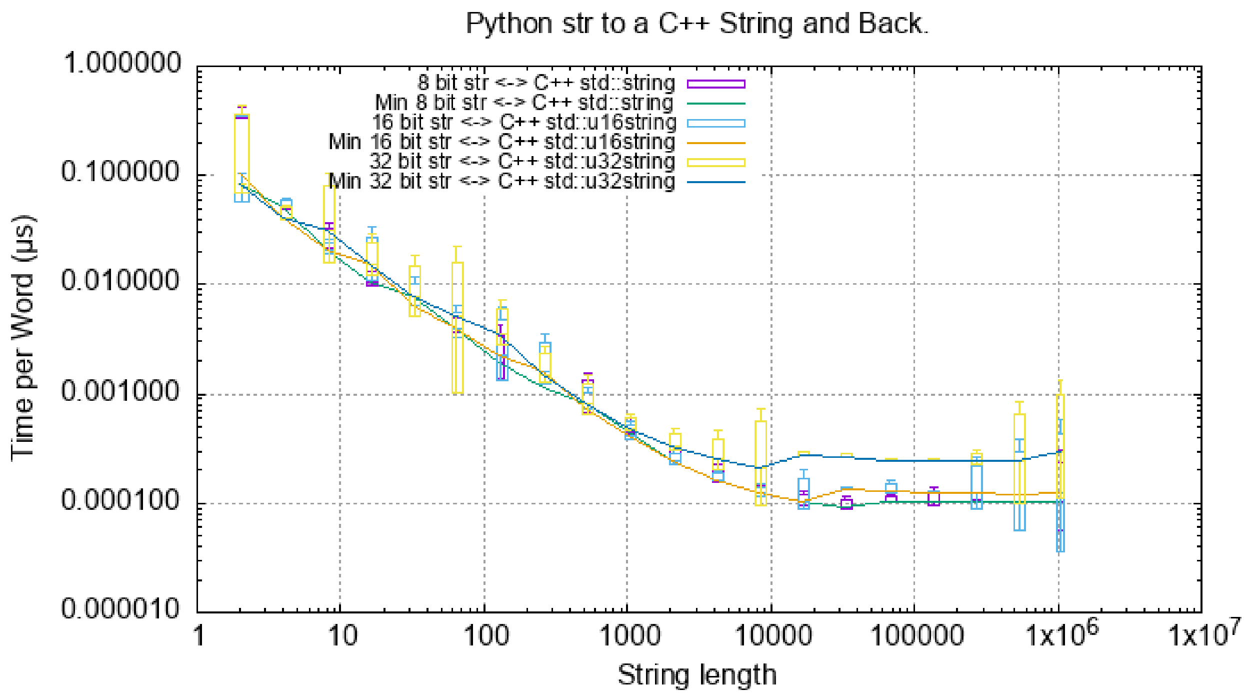


Python Strings

For strings of 8/16/32 bit ord sizes the roundtrip time plot looks like this:



Again we are getting around 10 GB/s roundtrip conversion. The rate plot is rather more revealing.



This shows the 16 bit word size takes about twice the 8 bit word size and the 32 bit word size takes about four time the 8 bit word size which is exactly as expected.

7.3.2 Python Containers Code

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 "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) {
    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);
}

static PyObject *
```

(continues on next page)

(continued from previous page)

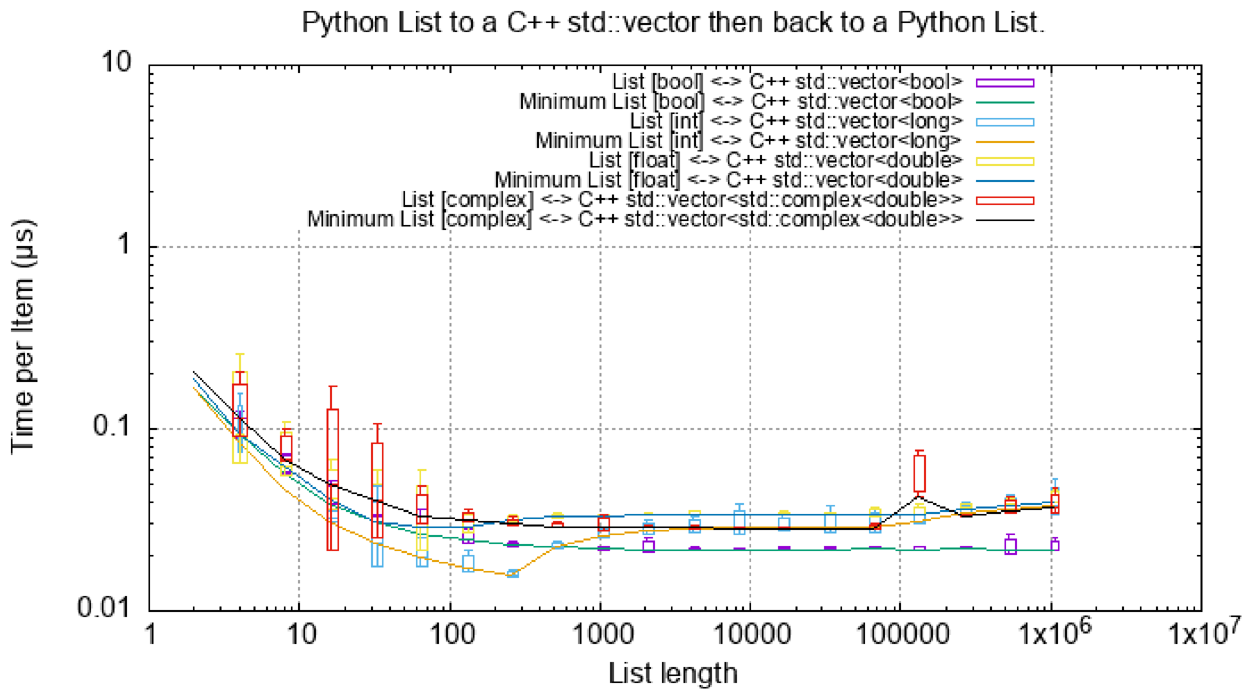
```
new_list_str16(PyObject *Py_UNUSED(module), PyObject *arg) {
    return new_list<std::u16string>(arg);
}

static PyObject *
new_list_str32(PyObject *Py_UNUSED(module), PyObject *arg) {
    return new_list<std::u32string>(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.

7.3.3 Python Lists

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

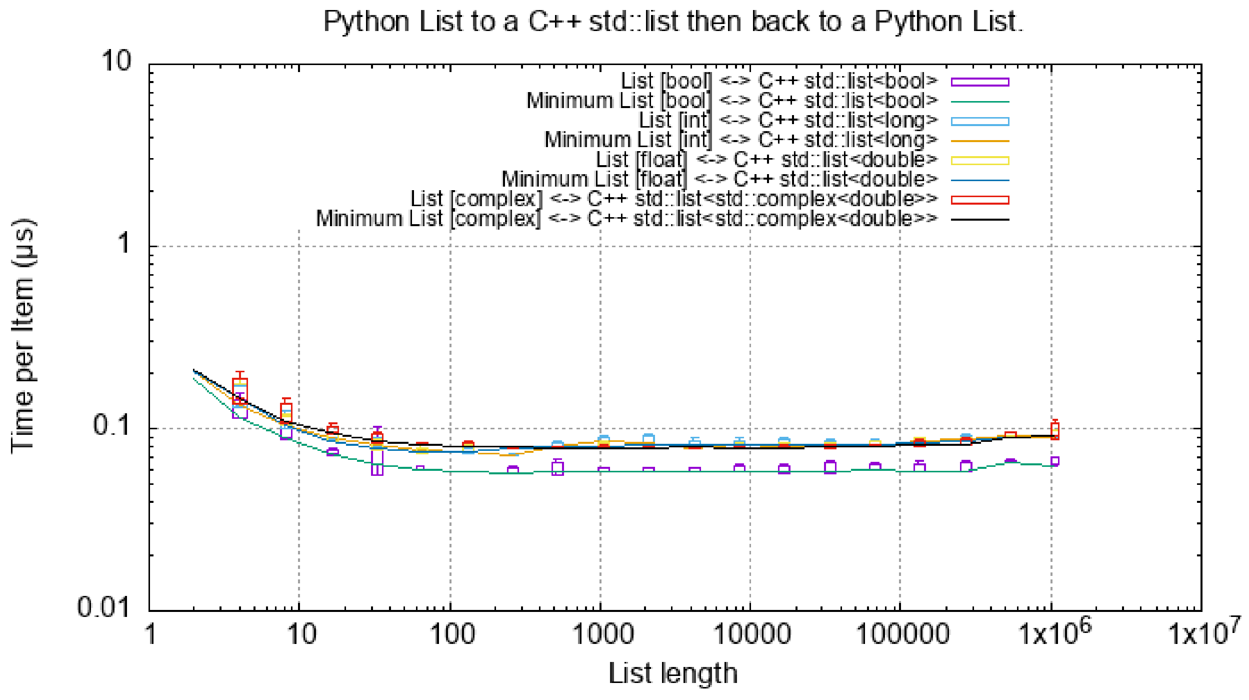
Here is the *round trip* performance of a Python list of `bool`, `int`, `float` and `complex` numbers via a C++ `std::vector`:



These are typically *round trip* converted at:

- 0.02 µs per object for booleans, say 50m objects a second.
- 0.04 µs per object for `int`, `float` and `complex`, say 25m objects a second.

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

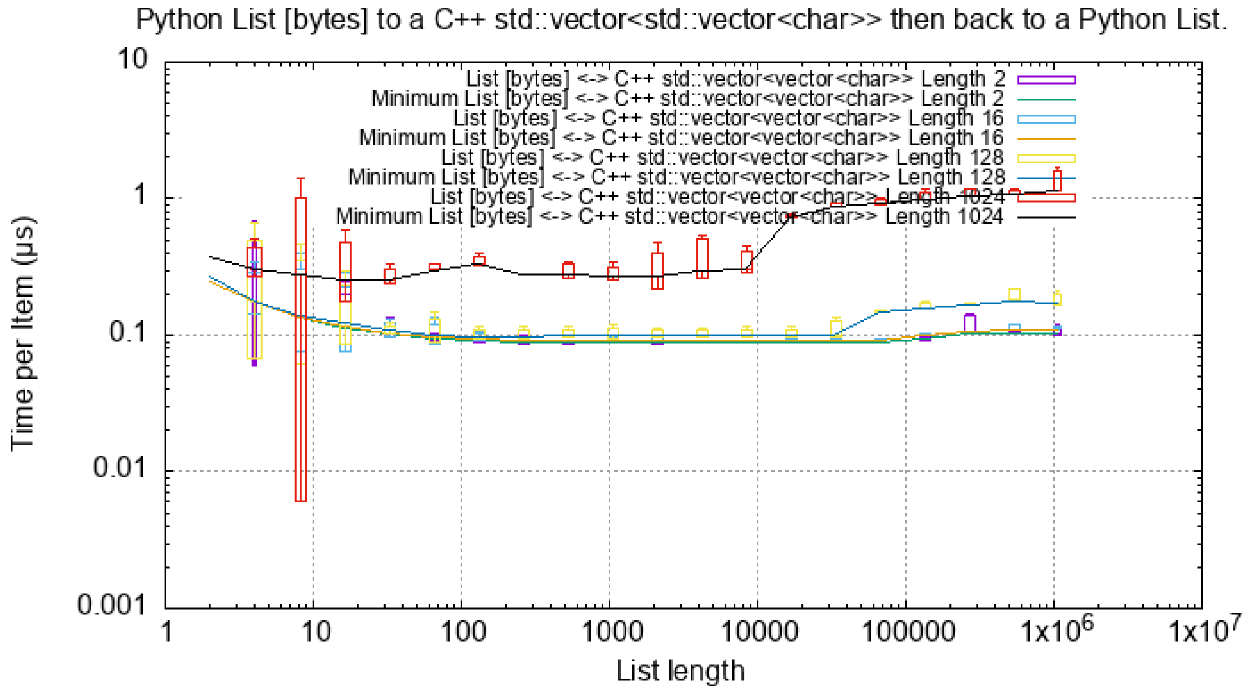


These are typically *round trip* converted at:

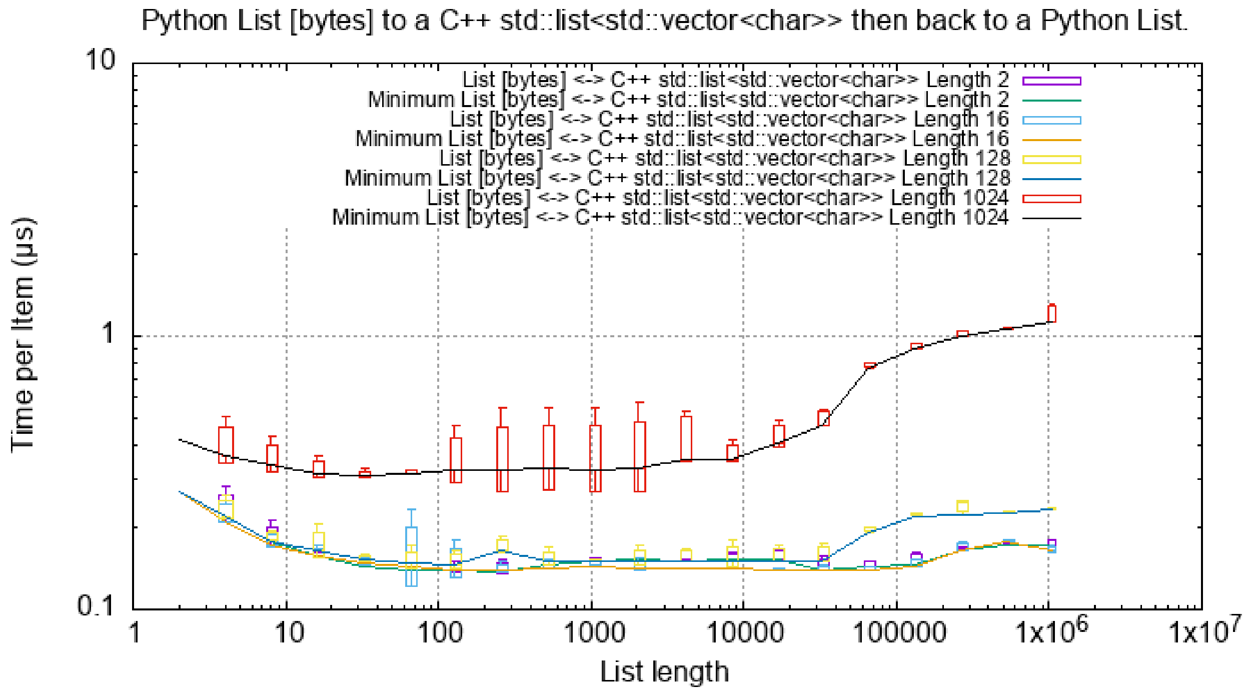
- 0.08 µs per object for booleans, say 120m objects a second. This is about 8x the cost of using a `std::vector`.
- 0.9 µs per object for other objects, say 10m objects a second. This is about 2x the cost of using a `std::vector`.

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`:

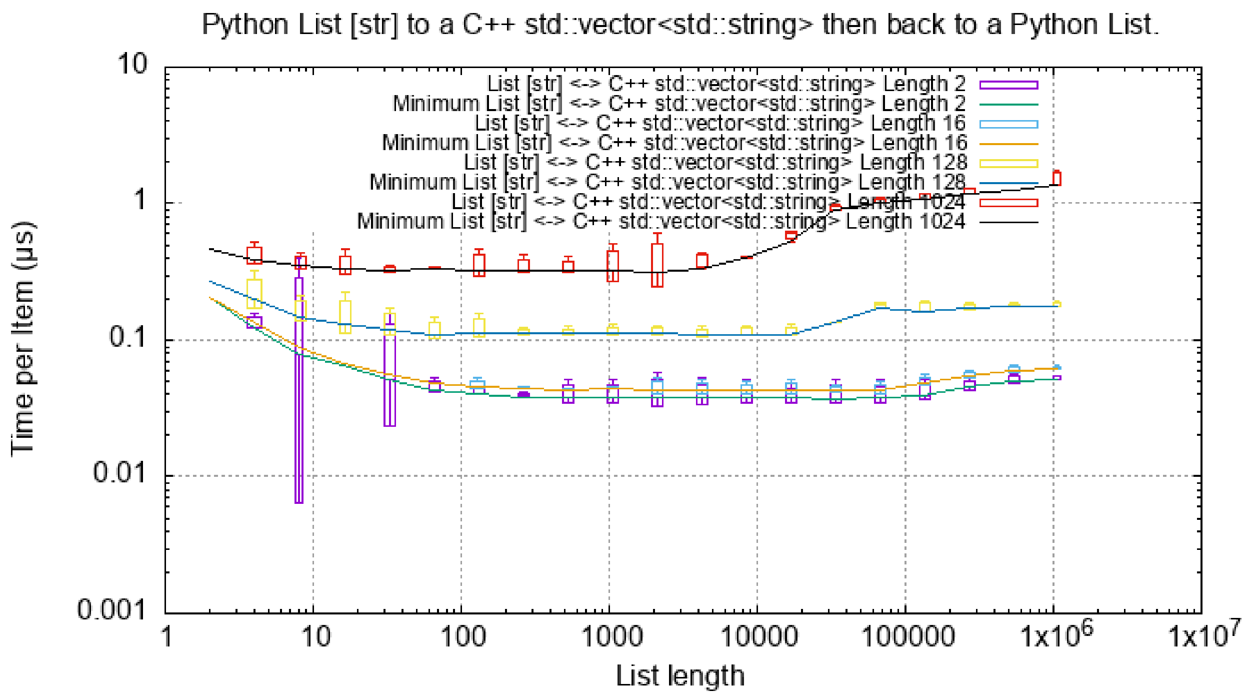


Given the size of each object this *round trip* time for lists can be summarised as:

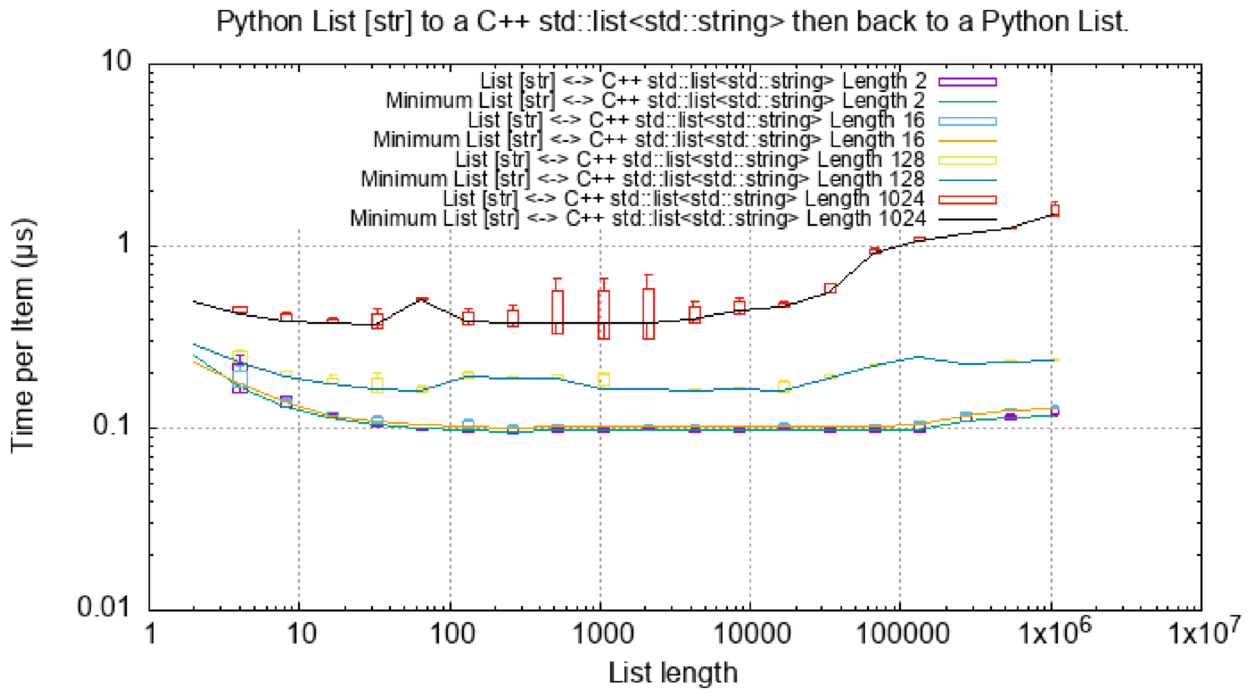
Object	Time per object (µs)	Rate (million/s)	Rate (Mb/s)	Notes
bytes[2]	0.1	10	20	
bytes[16]	0.1	10	160	
bytes[128]	0.1	10	1280	
bytes[1024]	0.4 to 2.0	0.5 to 2.5	500 to 2500	

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:

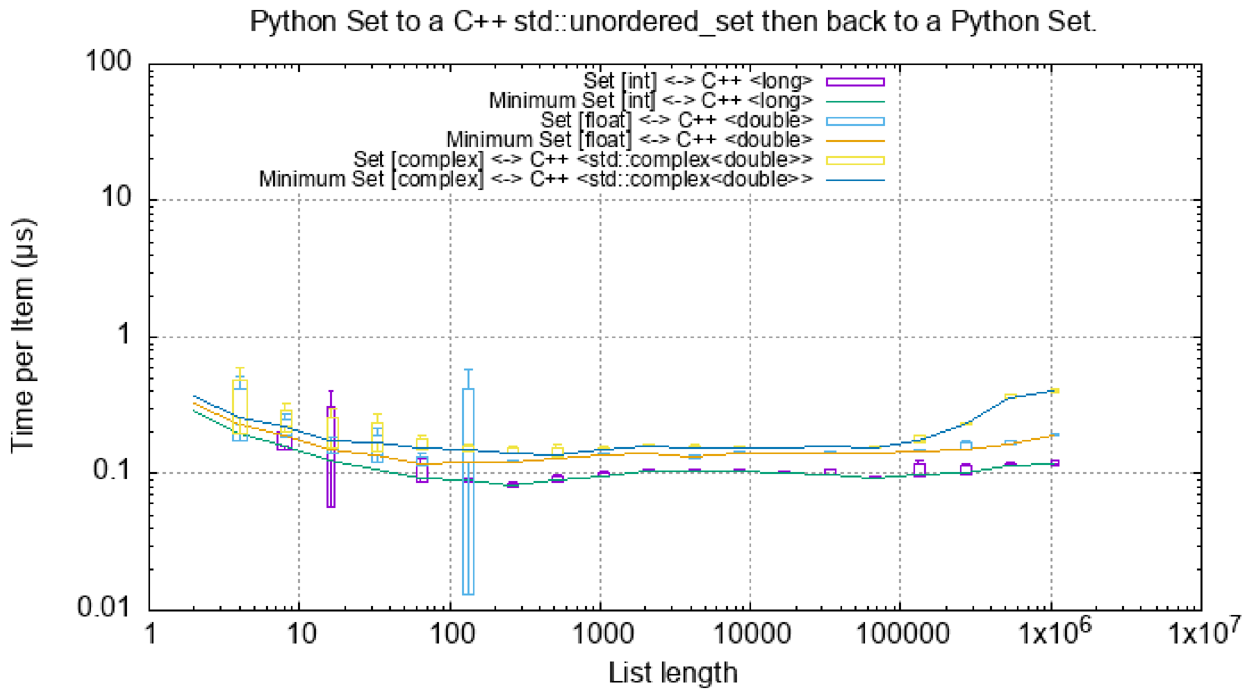
Object	Time per object (µs)	Rate (million/s)	Rate (Mb/s)	Notes
str[2]	0.05 to 0.1	10 to 20	20 to 40	
str[16]	0.05 to 0.1	10 to 20	160 to 320	
str[128]	0.2 to 0.4	2.5 to 5	320 to 640	
str[1024]	0.4 to 1.5	0.7 to 2.5	700 to 2500	

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

7.3.4 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):

- 0.1 µs per object for `int`, say 10m 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:

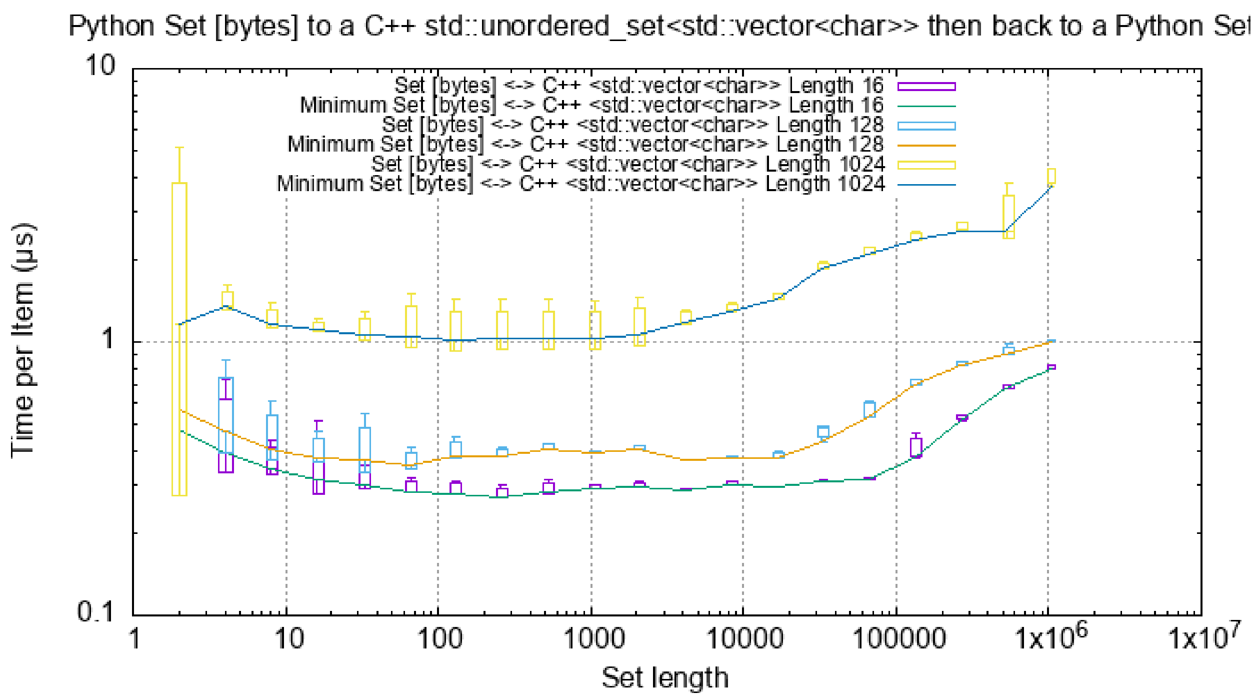
- 2.5x longer for an `int`
- 5x longer for a `float`.
- 8x longer for a `complex` number.

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	6x	
bytes[128]	0.6 to 1.5	0.1	6x to 15x	
bytes[1024]	1.0 to 5.0	0.4 to 2	2.5x	

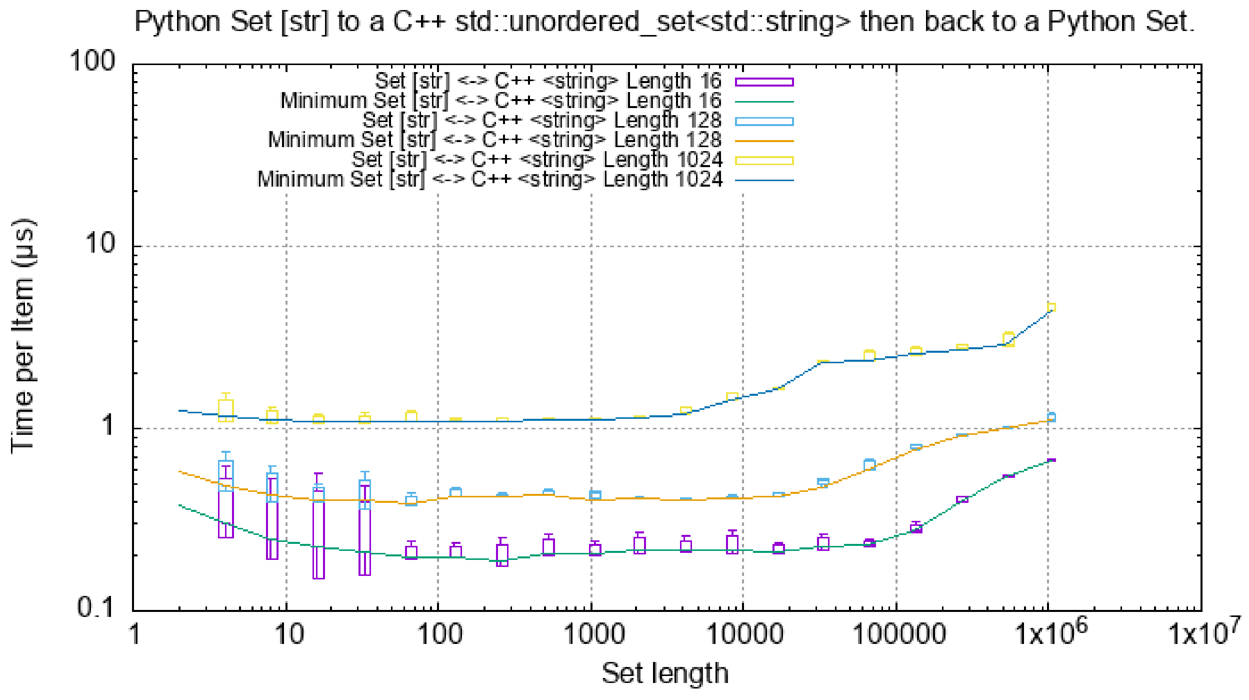
Again, the cost of hashing and insertion explains the difference.

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`

And a Python set of `str` (8 bit) 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	3x to 6x	
str[128]	0.8	0.2 to 0.4	2x to 44	
str[1024]	1.0 to 5.0	0.4 to 1.5	1x to 10x	

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

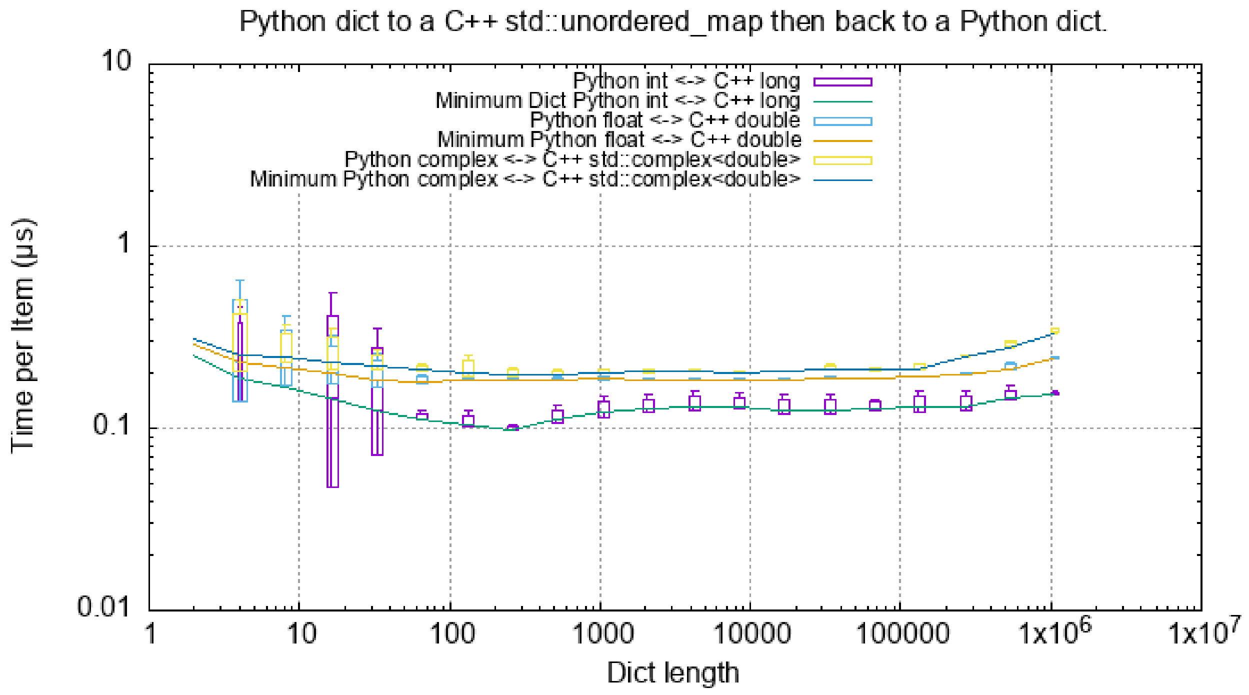
Given the size of each object this *round trip* time for sets can be summarised as:

Object	Time per object (µs)	Rate (million/s)	Rate (Mb/s)	Notes
bytes[16]	0.3	3	48	
bytes[128]	0.8	1.25	160	
bytes[1024]	1.0 to 5.0	0.2 to 1	200 to 1000	

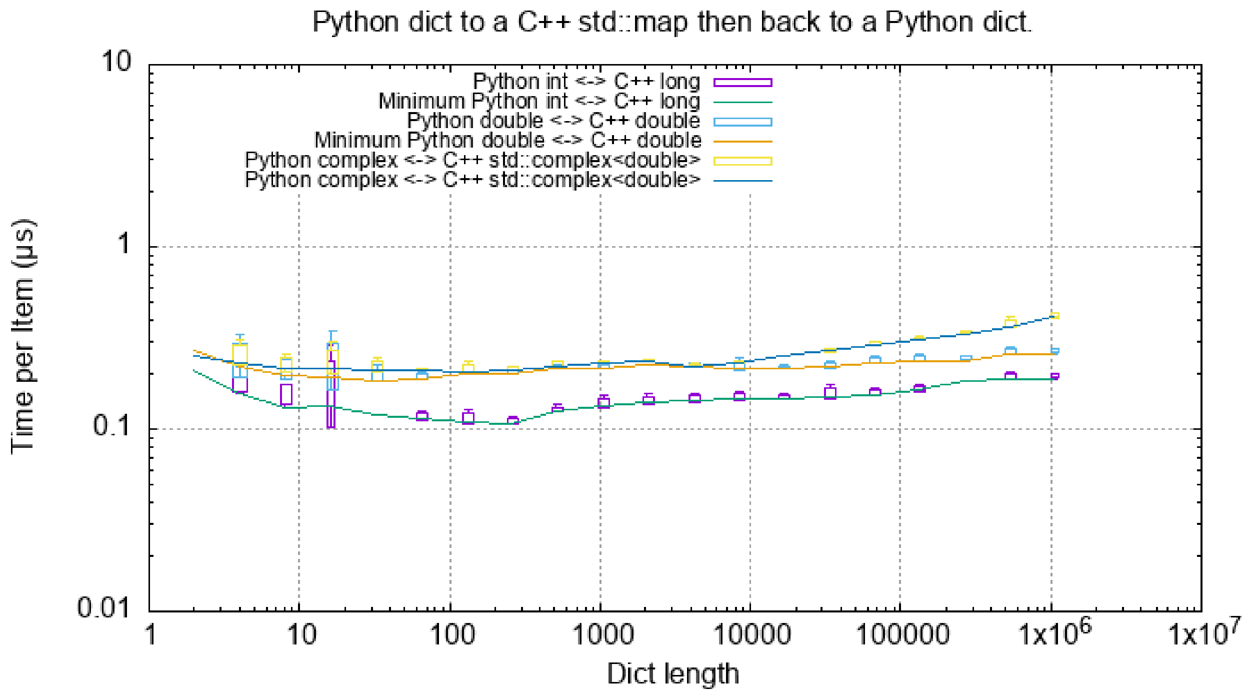
7.3.5 Python Dictionaries

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` for integers, floats and complex numbers. The keys and values are the same type. This plots the *round trip* cost *per key/value pair* against dict size.



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



The results for `std::unordered_map` and `std::map` are nearly identical.

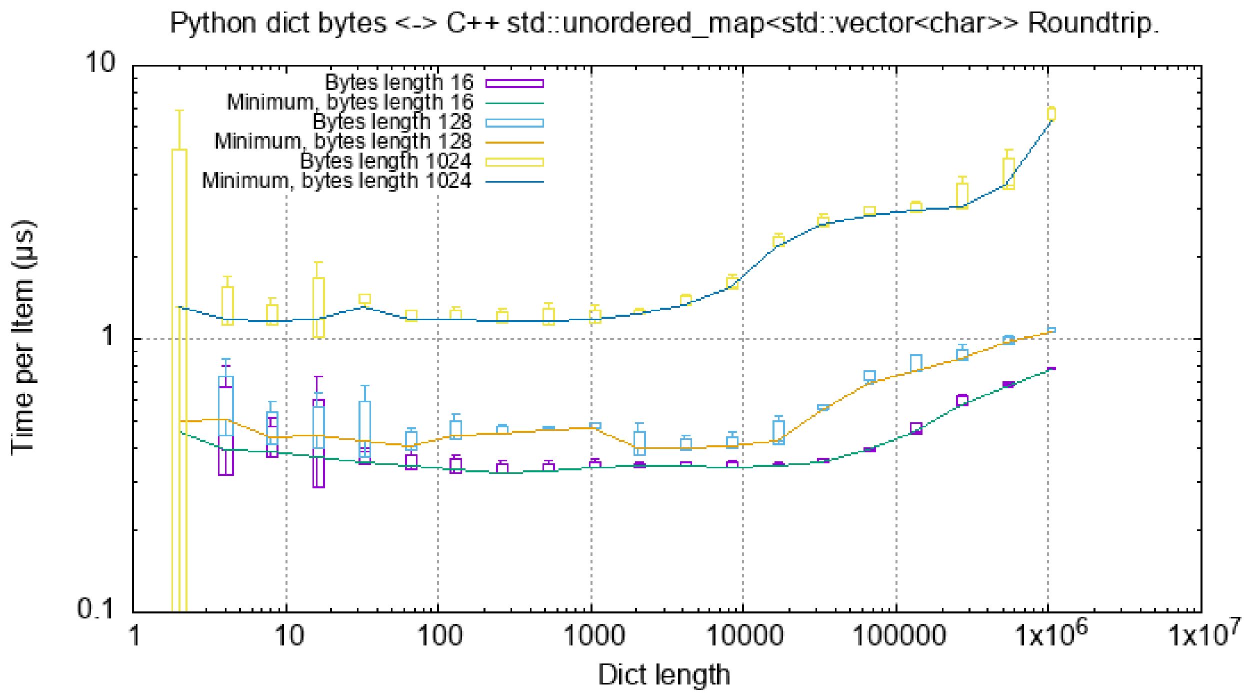
These are typically *round trip* converted at:

- 0.2 µs per object for an int or float, say 5m objects a second.
- 0.25 µs per object for a complex number, say 4m objects a second.

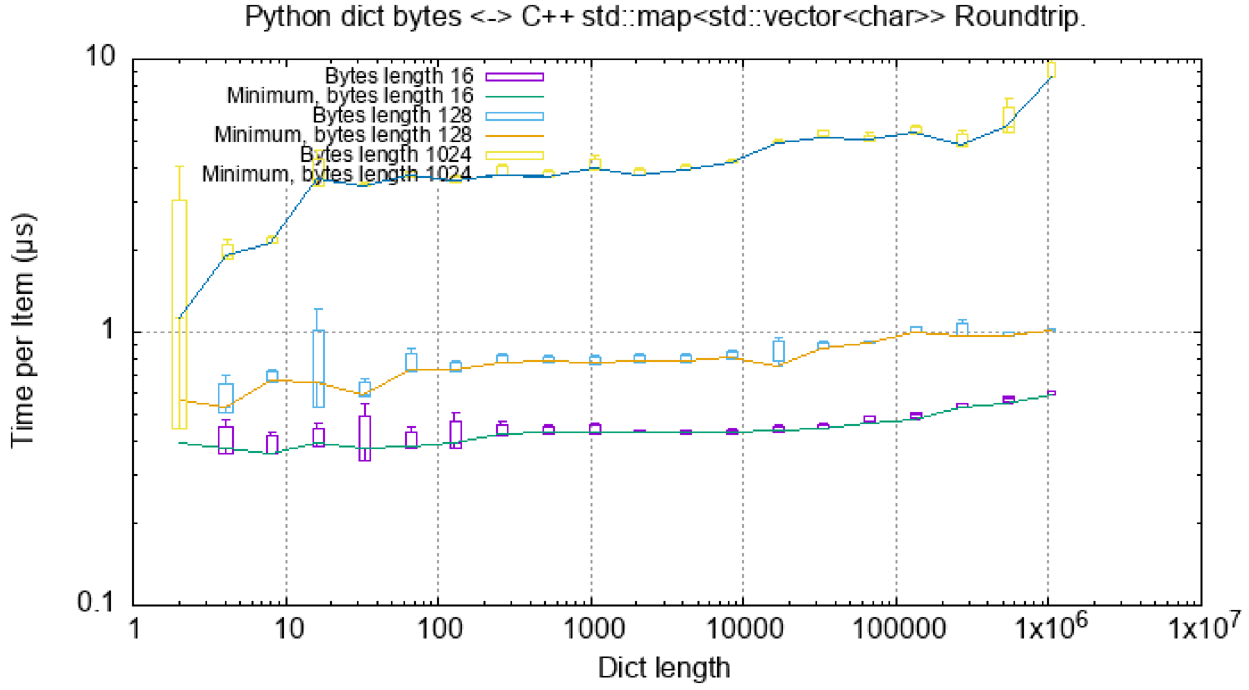
This is identical to the values for the set but includes the conversion time for both key and value. The hashing, insertion and potential re-hashing dominate the performance.

Python Dict of `bytes`

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`:



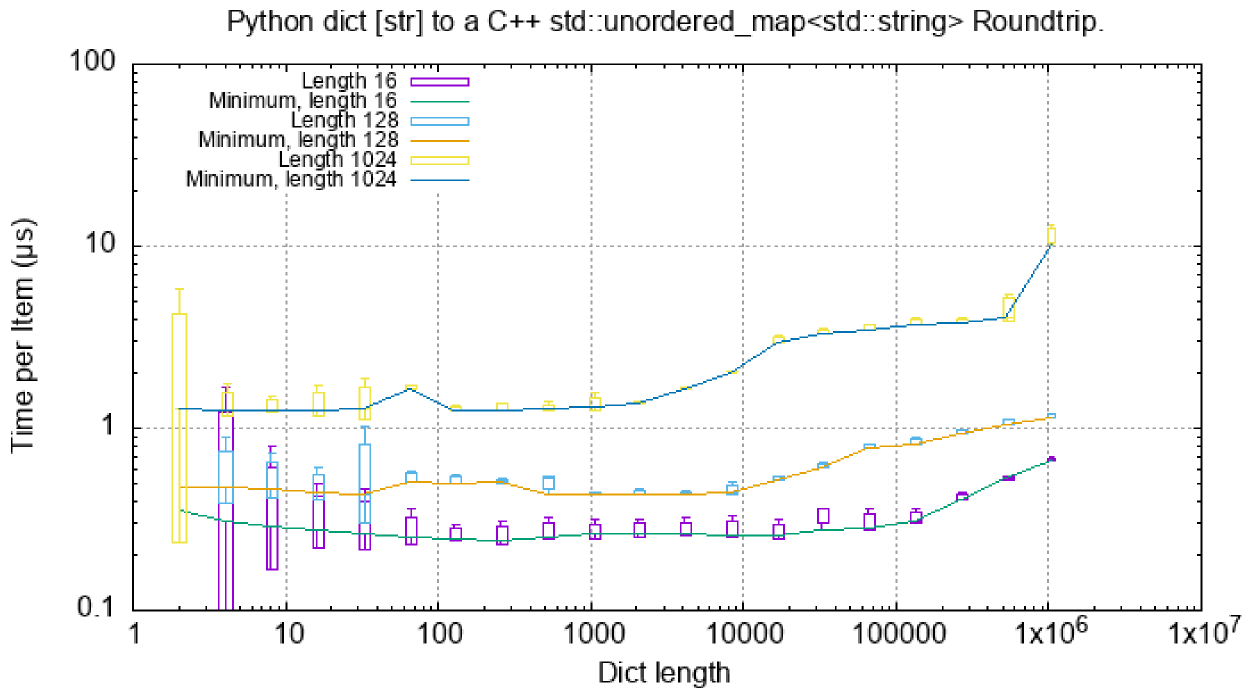
The `std::map` takes a shade longer than the `std::unordered_map`.

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

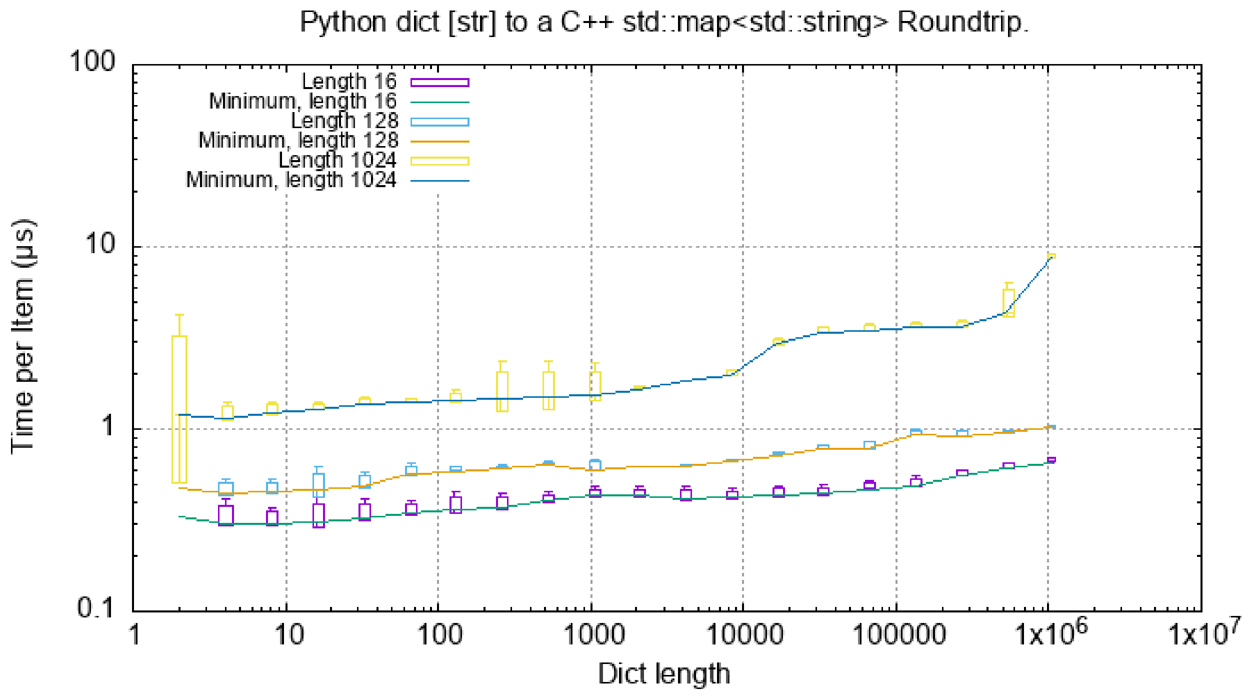
Object	Time per object (µs)	Rate (million/s)	Rate (Mb/s)	Notes
bytes[16]	0.5	2	64	
bytes[128]	0.6 to 1	1 to 1.5	256 to 425	
bytes[1024]	1 to 6	0.15 to 1.0	300 to 2000	

Python Dict of `str`

Here is the *round trip* time for a Python dict [str, str] (8 bit strings) 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`:



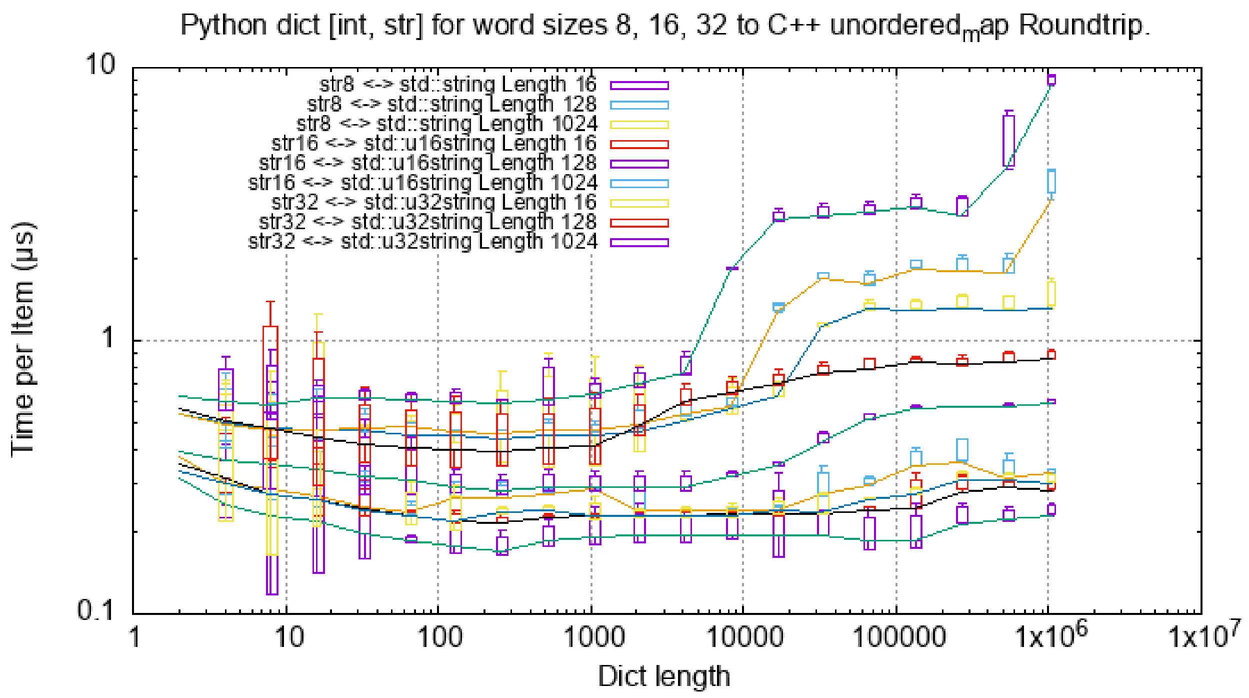
The `std::map` is near identical to the `std::unordered_map`.

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

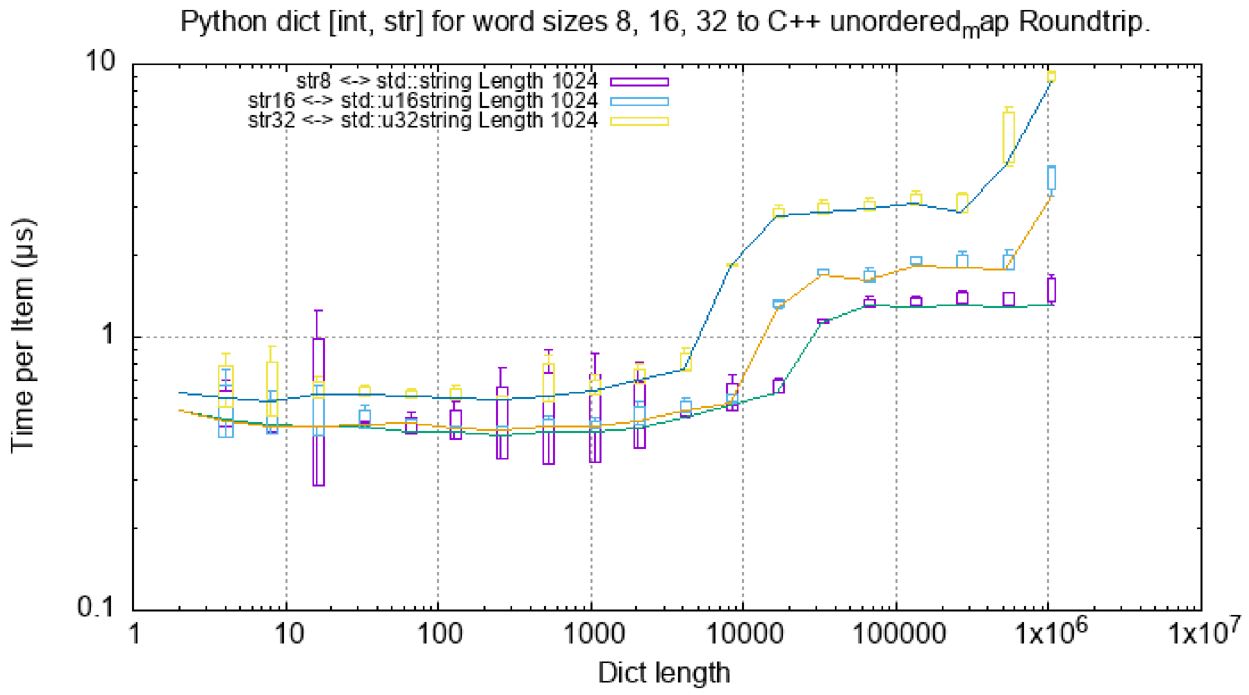
Object	Time per object (µs)	Rate (million/s)	Rate (Mb/s)	Notes
str[16]	0.3	3	96	
str[128]	0.6 to 1	1 to 1.7	256 to 440	
str[1024]	1 to 10	0.1 to 1	200 to 2000	

Unicode Strings of Different Codepoint Sizes

Here is a plot of round tripping a dict of `[int, str]` for unicode sizes of 8 bit, 16 bit and 32 bit to a C++ `std::unordered_map` and back:

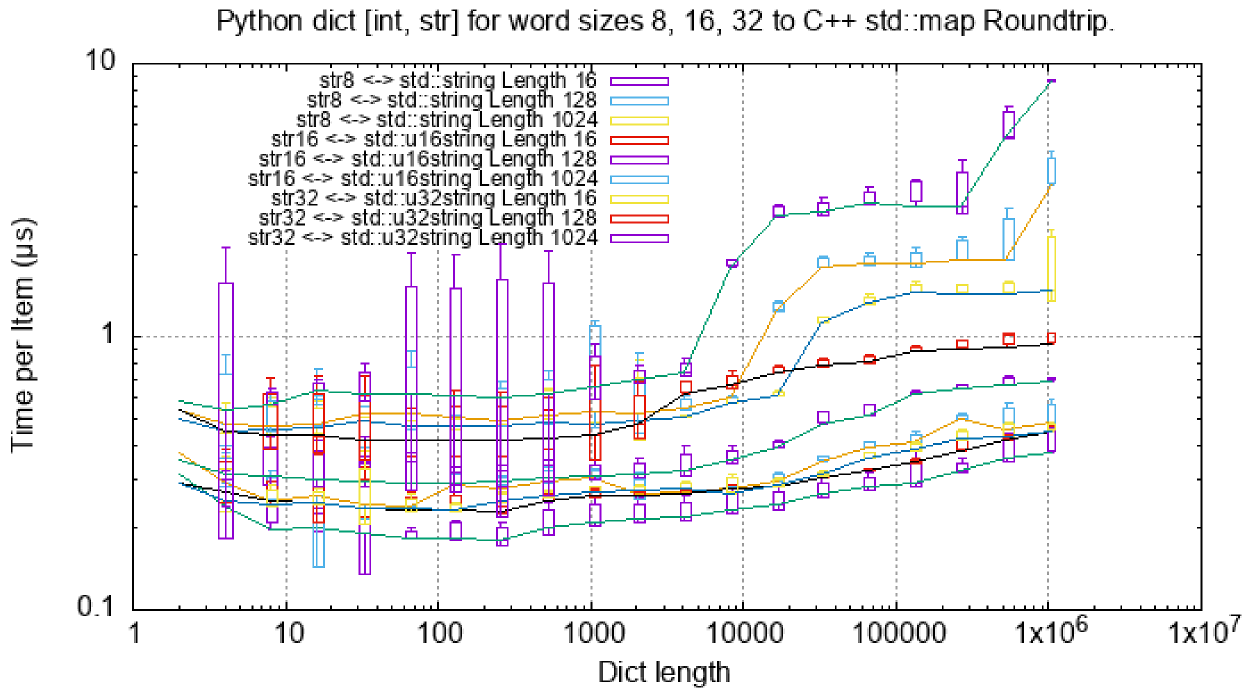


This is probably best shown in simplified form for 1024 length strings only:

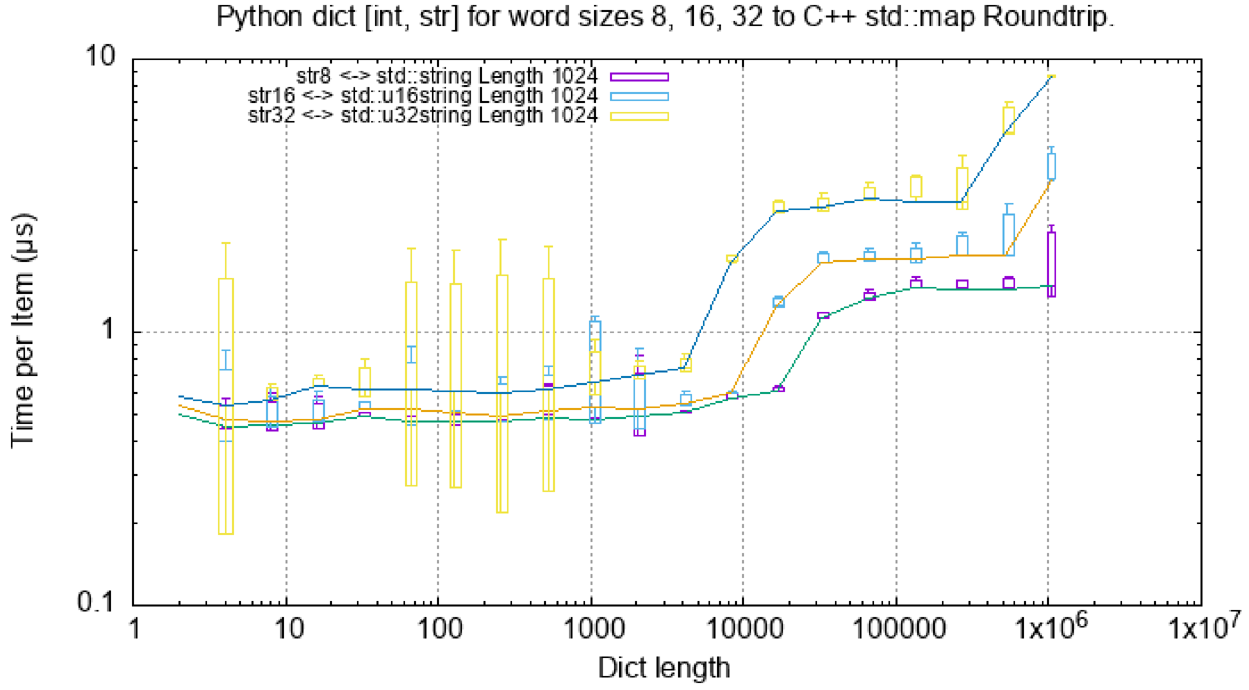


As expected, the 16 bit strings take around twice as long to convert as the 8 bit strings and the 32 bit strings take around four times as long to convert as the 8 bit strings.

And similar plots for converting to a `std::map`:



And, simplified for 1024 length strings.



The `std::map` is near identical to the `std::unordered_map`.

7.3.6 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.

7.4 Memory Use

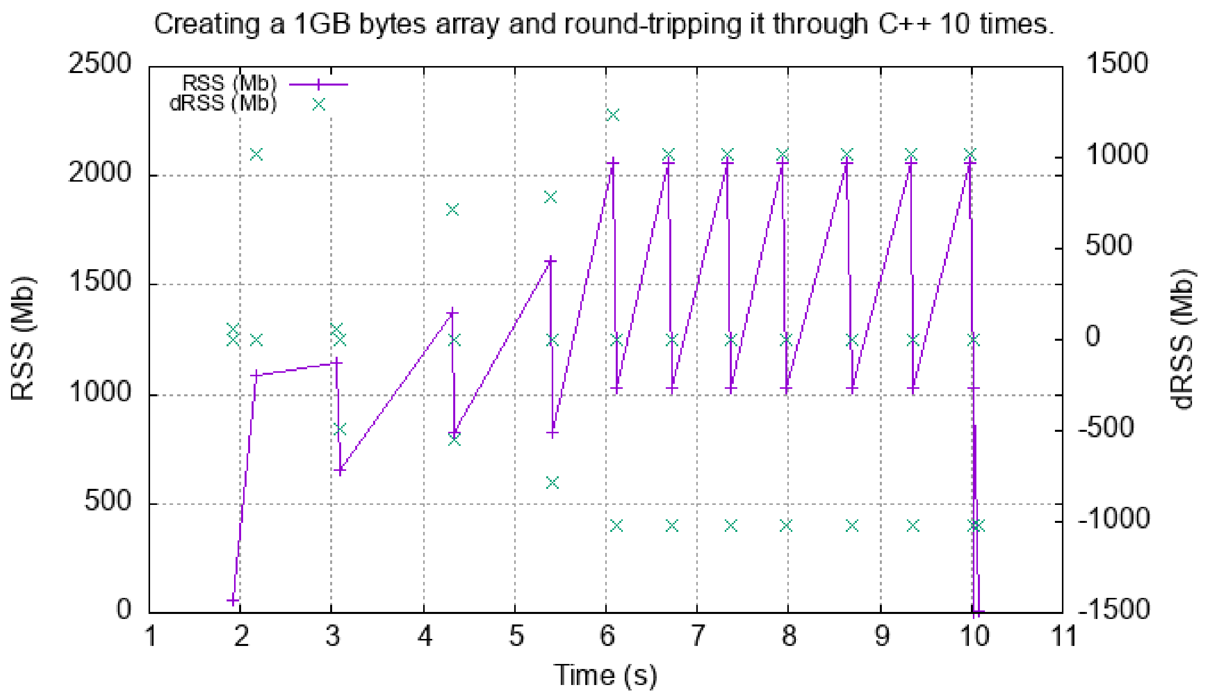
7.4.1 Single Bytes and Strings

Here is a typical test that takes a single 1GB bytes object, passes it into `cPyCppContainers.new_bytes()` which creates a new C++ `std::vector<char>` then converts that back into a new Python bytes object. This is repeated 10 times under the watchful eye of `cPyMemTrace.Profile()`. The test is from `tests/unit/test_with_pymemtrace.py`:

```
import cPyCppContainers
import cPyMemTrace

def test_new_bytes():
    with cPyMemTrace.Profile(4096 * 16):
        original = b' ' * (2 << 30)
        results = []
        for _r in range(10):
            cPyCppContainers.new_bytes(original)
        // This helps cPyMemTrace complete.
        gc.collect()
```

Bytes

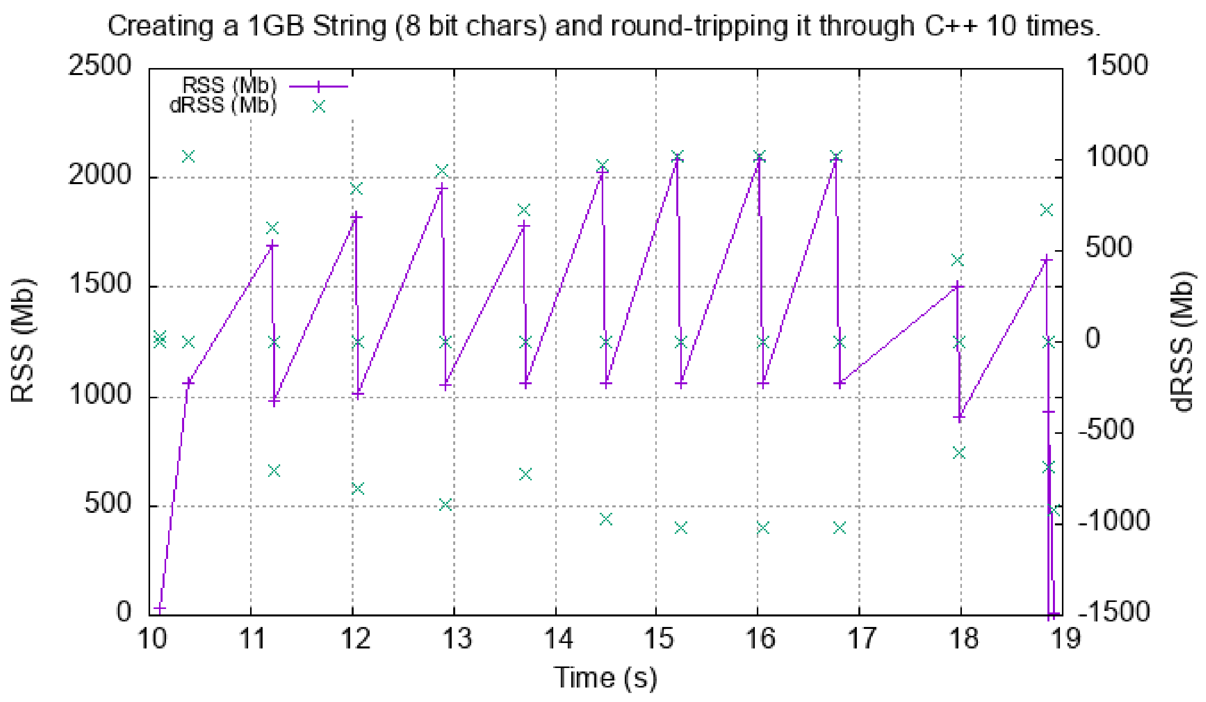
Creating a 1GB bytes object and round-tripping it between a C++ `std::vector<char>` and back to a new Python bytes object.



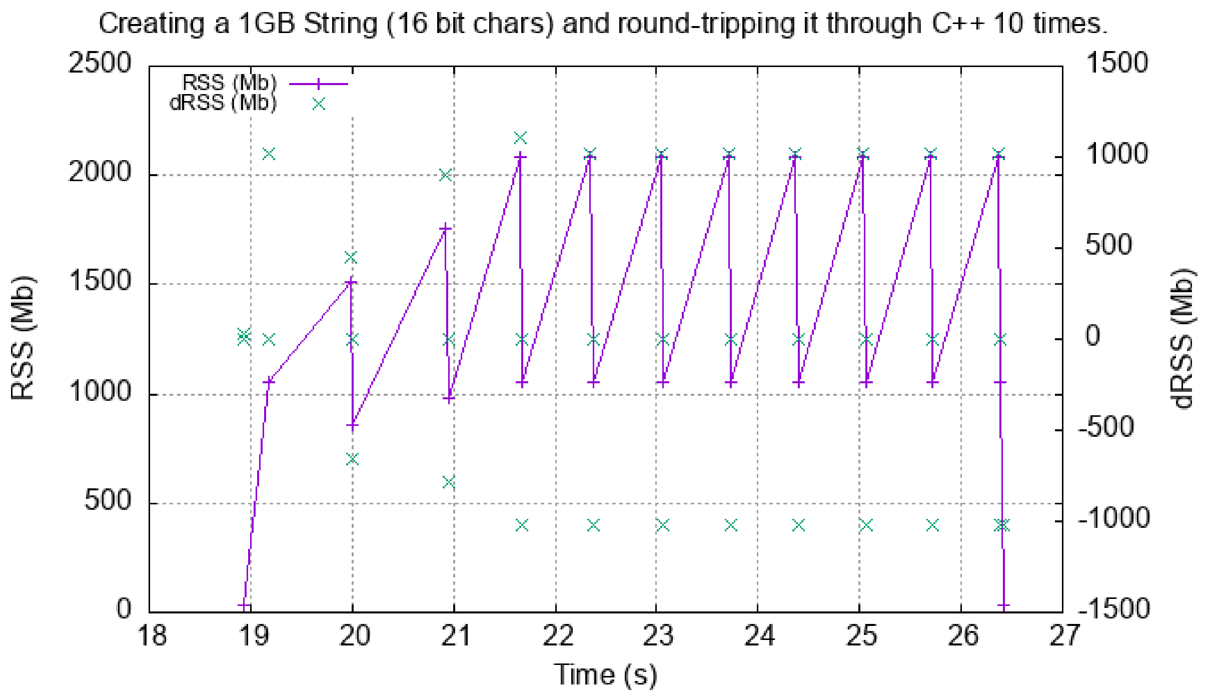
This demonstrates that there is no memory leak in converting these objects.

Strings

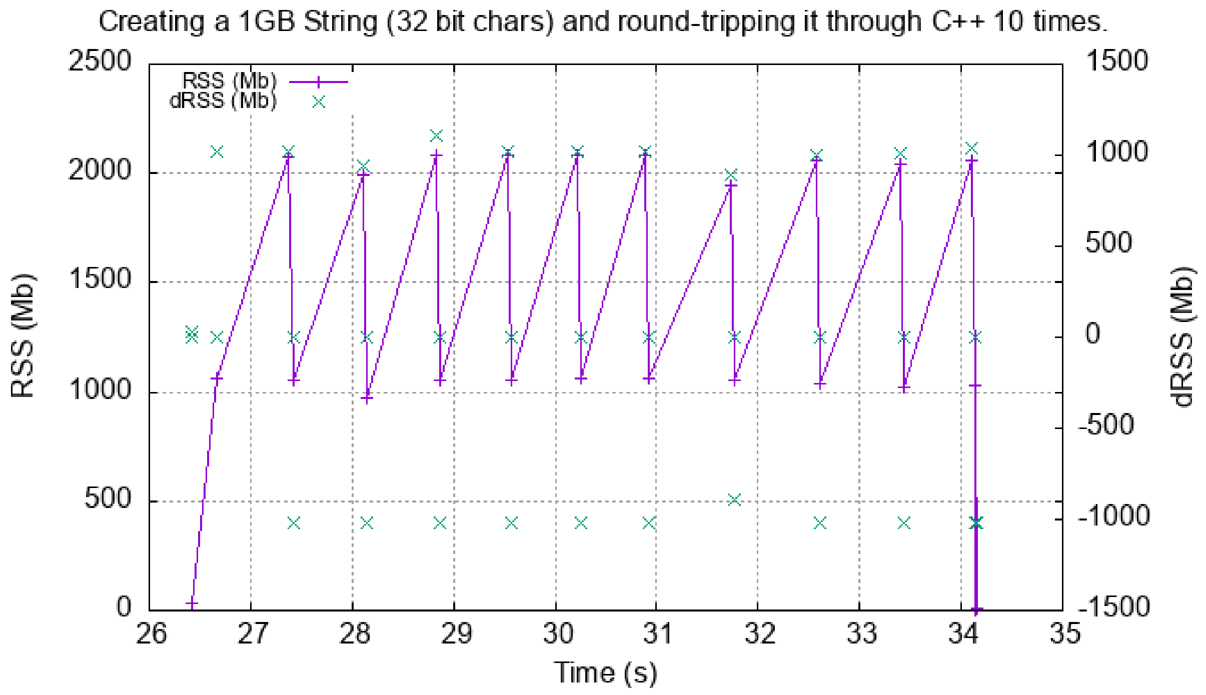
Strings with 8 bit characters, this 1GB of 8 bit characters:



Strings with 16 bit characters, this 0.5GB of 16 bit characters:



Strings with 32 bit characters, this 0.25GB of 32 bit characters:



7.4.2 Containers

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.

These tests were made using Python 3.12.

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

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

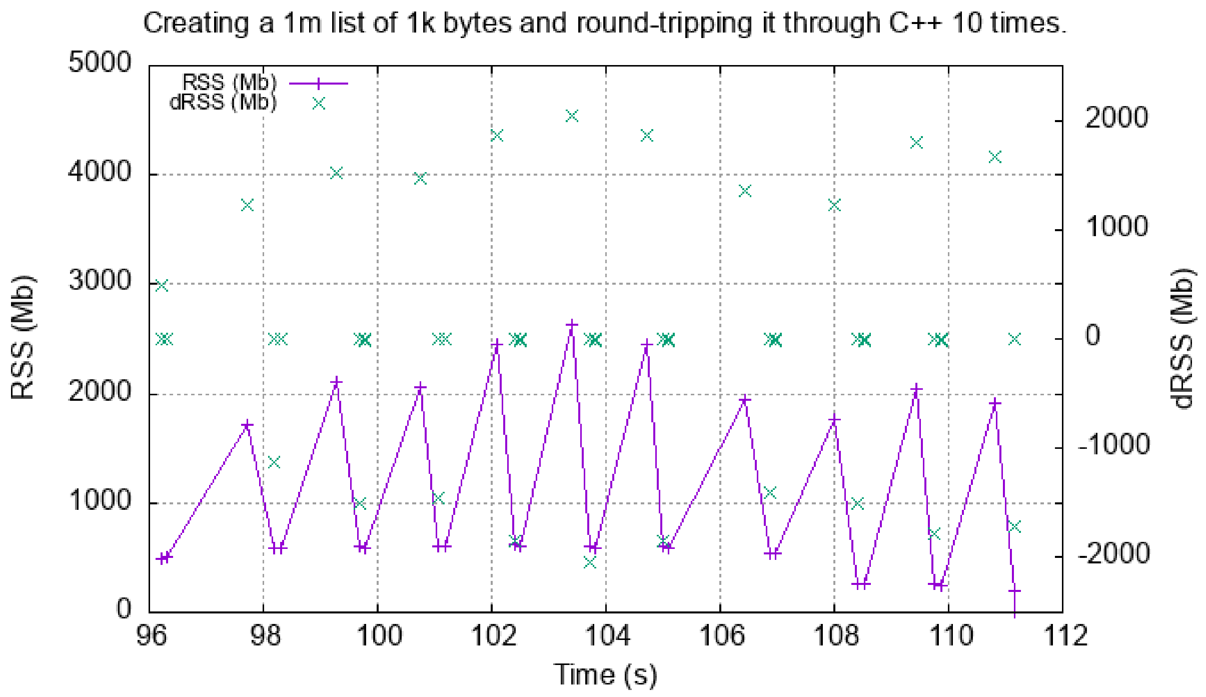
```python
from pymemtrace import cPyMemTrace

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
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
```

Python List of bytes

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



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

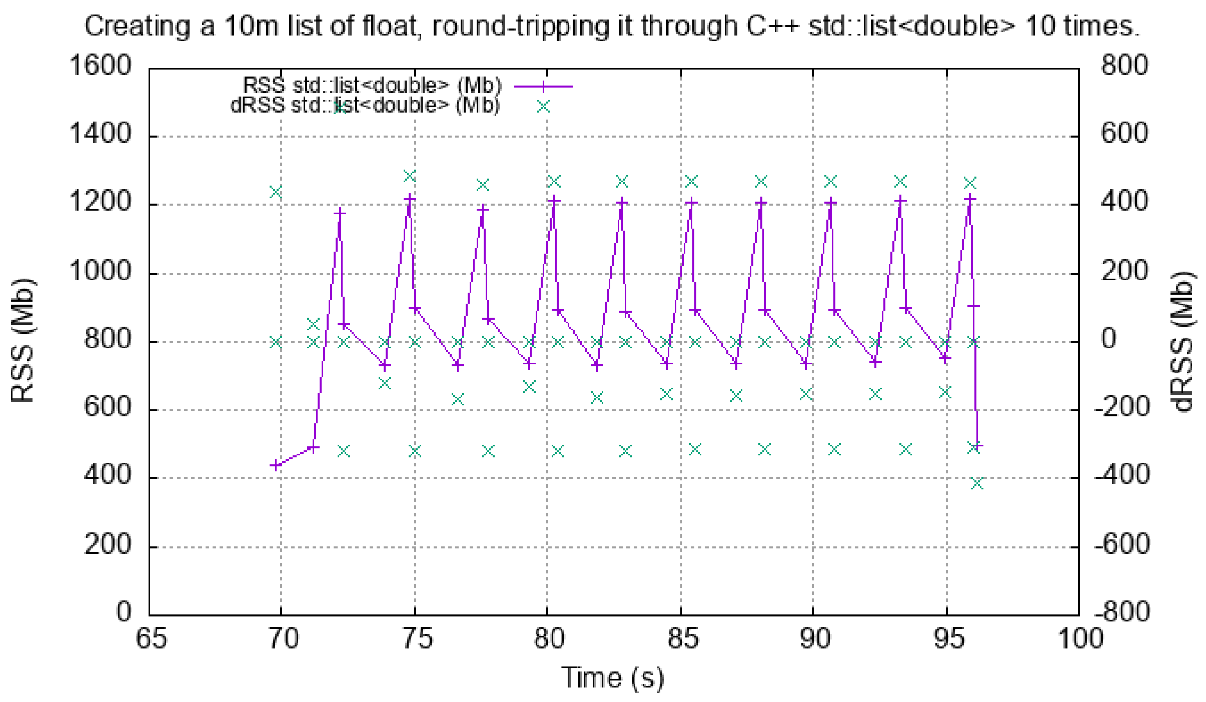
- Basic Python, say 30Mb
- The original Python list of bytes, 1024Mb.
- The C++ `std::vector<std::string>`, 1024Mb.
- The new Python list of bytes, 1024Mb.

This would be a total of 3102Mb which is, broadly speaking the maximum RSS that we are seeing.

Note: Earlier Python versions with different memory managers displayed significantly lower maximum RSS of around 2200 MB.

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.

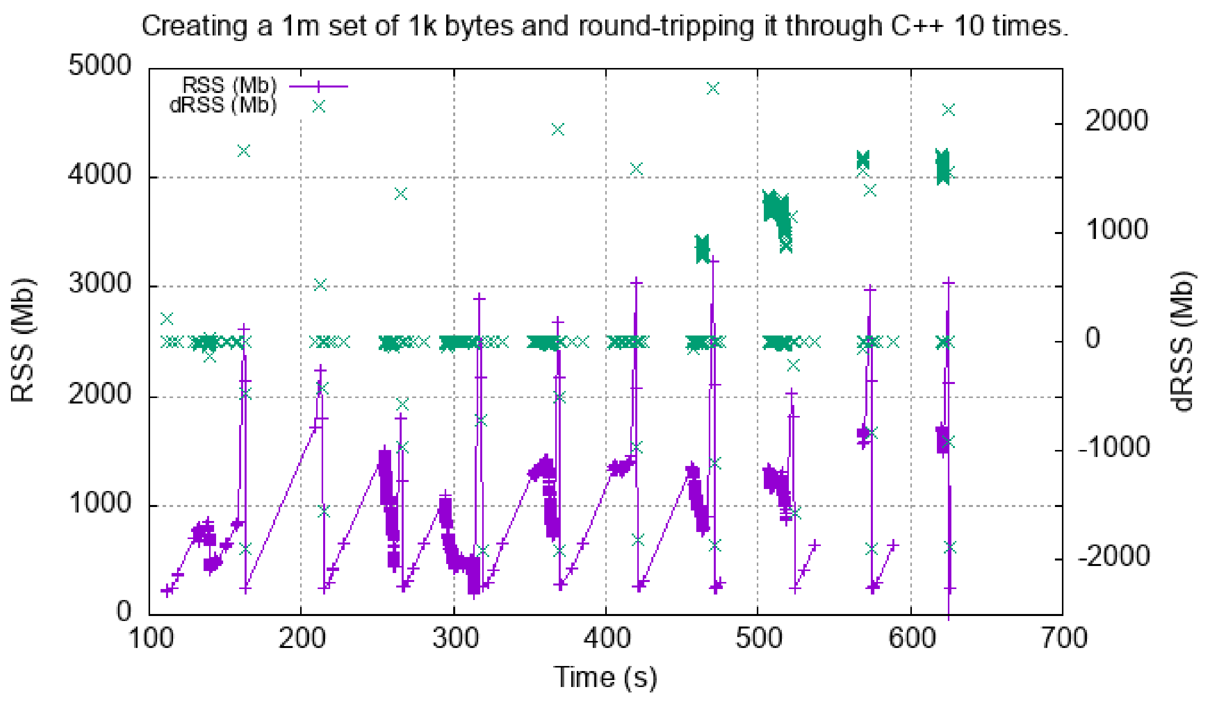
Python Set of bytes

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.

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:

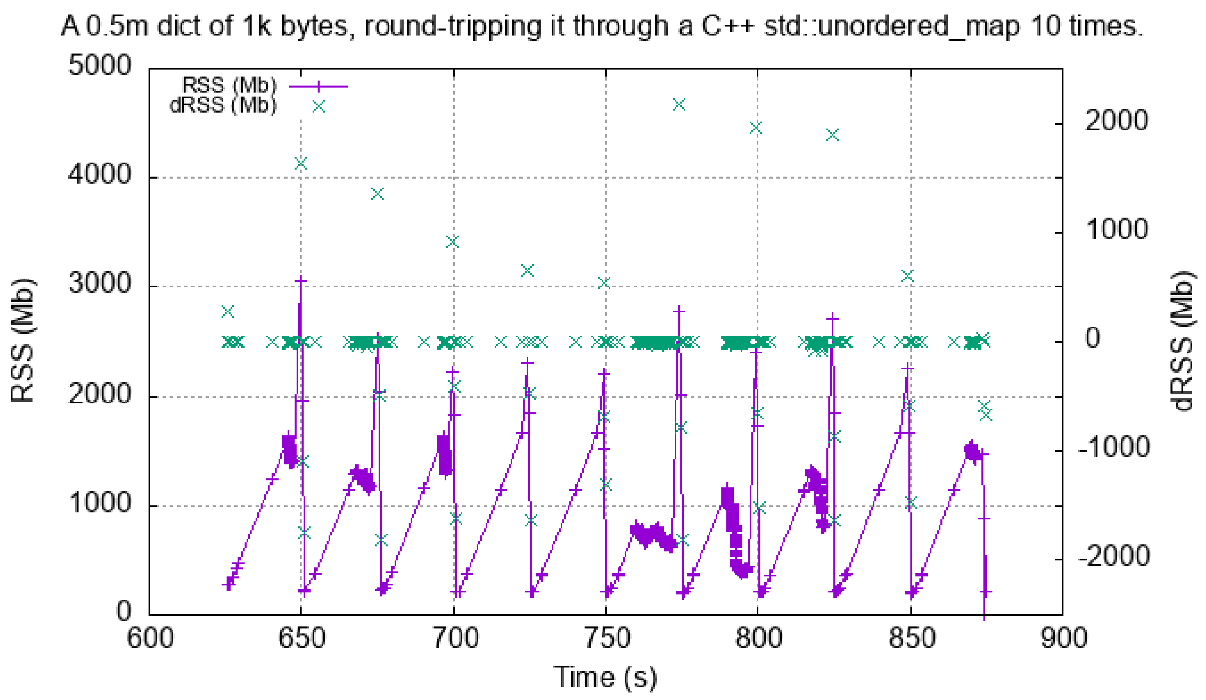
```
with cPyMemTrace.Profile(4096 * 16):
    total_bytes = 2**20 * 2**10
    byte_length = 1024
    dict_length = total_bytes // byte_length // 2
    random_bytes = [random.randint(0, 255) for _i in range(byte_length)]
    for _r in range(10):
        original = {}
        for i in range(dict_length):
            k = bytes(random_bytes)
```

(continues on next page)

(continued from previous page)

```
            original[k] = b' ' * byte_length
            # 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_dict_bytes_bytes(original)
```

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



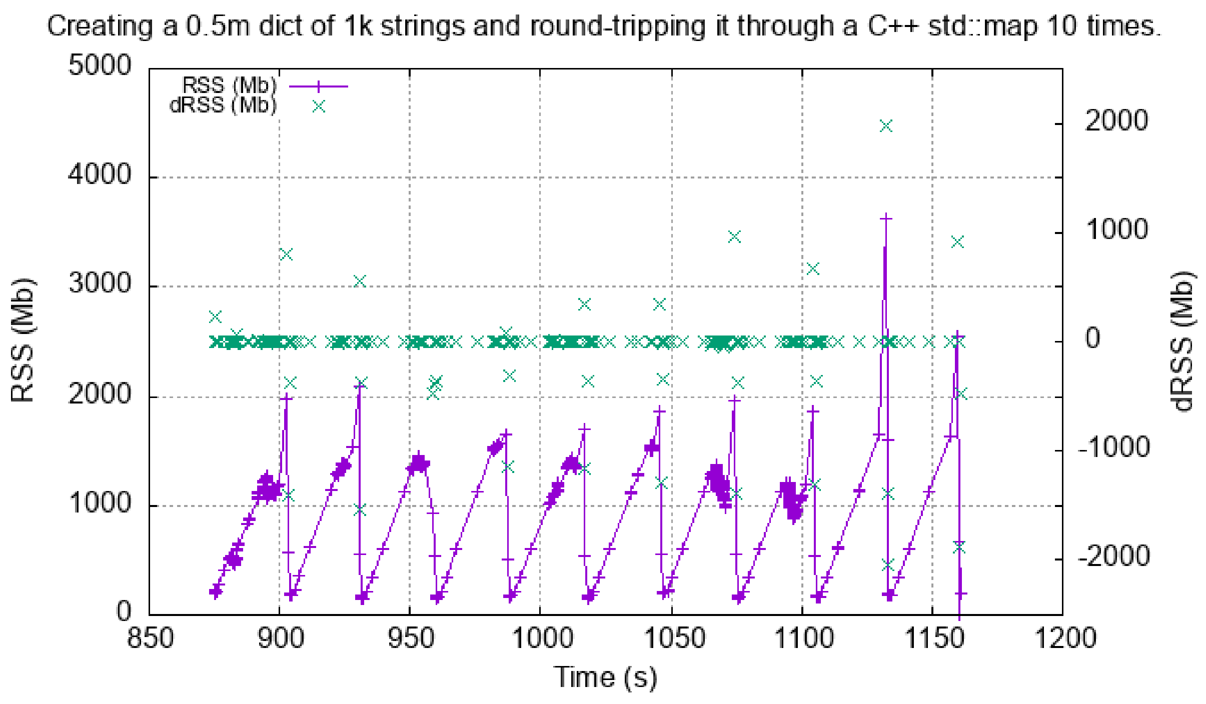
In the dictionary case constructing the original dict takes around 1500Mb. So on entry to `new_dict_bytes_bytes` the RSS is typically 1700Mb. Constructing the `std::unordered_map<std::vector<char>, std::vector<char>>` and a new Python dict takes an extra 2500Mb taking the total memory to around 4200MB. 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.

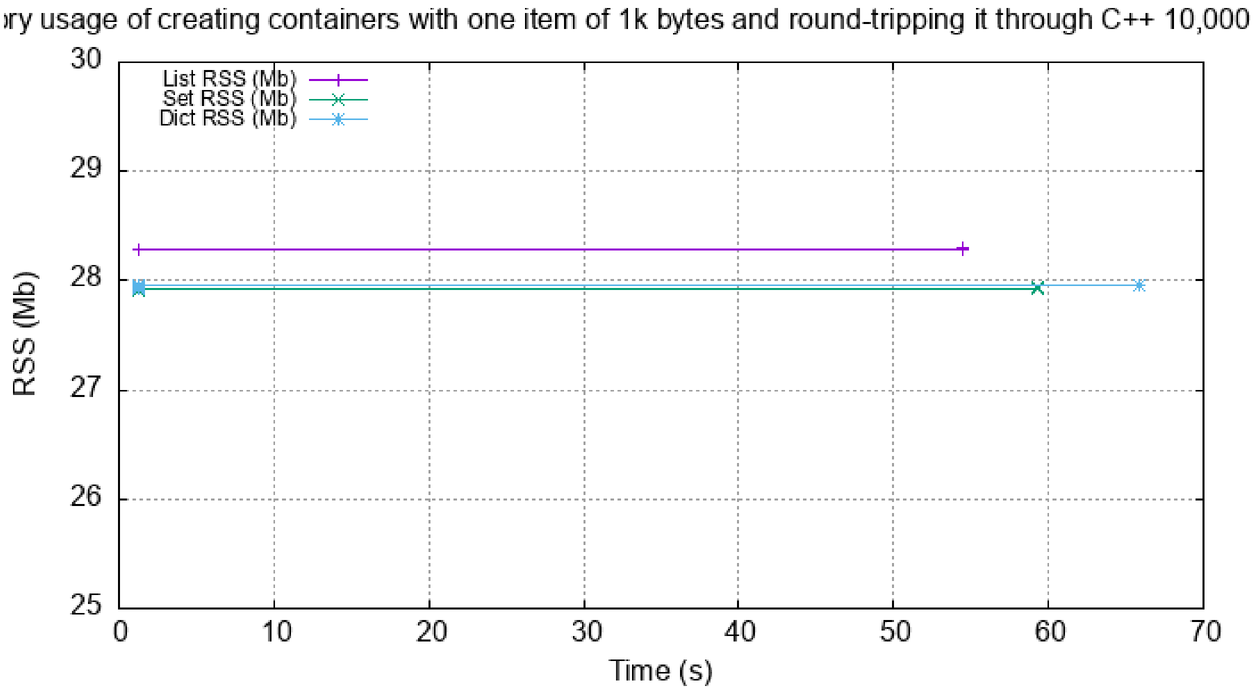
Containers of Just One Object

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.

CHAPTER

EIGHT

HISTORY

8.1 Version 0.4.0

- Support std::u16string and std:u32string.
- Renaming to make more consistent use of "PyCppContainers".
- Fairly major documentation and test changes.

8.2 Version 0.3.2

- Support Python versions 3.8 to 3.13.

8.3 Version 0.3.1

- First release.

8.4 Version 0.3.0

- Migrate to CLion

8.5 Version 0.2.0

- Unpublished.
- Date: Mon Mar 15 10:04:48 2021 +0000

8.6 Version 0.1.0

- Unpublished.
- Date: Thu Nov 22 10:54:39 2018 +0000

CHAPTER

NINE

TODO

Todo: User Defined Types From Pure Python Types: Add in version 0.5.0

original entry

Todo: Interoperation with `numpy` ND Arrays: Add the existing example code in version 0.5.0.

original entry

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search