
Python/C++ Homogeneous Containers

Release 0.1.0

Paul Ross

Oct 24, 2021

CONTENTS:

1	Introduction	1
1.1	TODO	1
1.1.1	Alternatives	1
1.2	A Problematic Example	1
1.3	Why This Project	2
1.4	Hand Written Functions	3
1.4.1	Converting a Python tuple or list to a C++ <code>std::vector<T></code>	4
1.4.2	Partial Specialisation to Convert a Python list to a C++ <code>std::vector<T></code>	4
1.5	Generated Functions	5
1.5.1	Usage	6
1.5.2	Converting a C++ <code>std::vector<T></code> to a Python tuple or list	6
2	Using this Library in Your C++ Code	7
2.1	The Basics	7
2.1.1	Code Generation	7
2.1.2	Build Configuration	7
2.1.3	Source Inclusion	7
2.2	Examples	8
2.2.1	Using C++ to Double the Values in a Python List of <code>float</code>	8
2.2.2	Reversing a tuple of bytes in C++	9
2.2.3	Incrementing dict values in C++	10
3	Examples by Container	11
3.1	Python Tuples	11
3.1.1	Converting a Python Tuple to a C++ <code>std::vector</code>	11
3.1.2	Converting a C++ <code>std::vector</code> to a Python Tuple	11
3.2	Python Lists	12
3.2.1	Converting a Python List to a C++ <code>std::vector</code>	12
3.2.2	Converting a C++ <code>std::vector</code> to a Python List	12
3.3	Python Sets	12
3.3.1	Converting a Python Set to a C++ <code>std::unordered_set</code>	12
3.3.2	Converting a C++ <code>std::unordered_set</code> to a Python Set	12
3.4	Python Dicts	13
3.4.1	Converting a Python dict to a C++ <code>std::unordered_map</code>	13
3.4.2	Converting a C++ <code>std::unordered_map</code> to a Python dict	13
3.5	Matrix Example	14
3.5.1	Converting a Python Tuple[Tuple[float]] to a C++ <code>std::vector<std::vector<double>></code>	14
3.5.2	Converting a C++ <code>std::vector<std::vector<double>></code> to a Python Tuple[Tuple[float]]	14

4	Design	17
4.1	<code>python_convert.h</code>	17
4.1.1	Conversion of Objects	17
4.1.2	Creating and Accessing Python Containers	18
4.2	Conversion Templates	18
4.3	Python Lists and Tuples	18
4.3.1	Conversion From a <code>std::vector<T></code> to a Python List or Tuple	18
4.3.2	Conversion From a Python List or Tuple to a <code>std::vector<T></code>	20
5	Performance	23
5.1	Summary	23
5.1.1	Fundamental Types	23
5.1.2	Strings of Different Lengths	24
5.2	Python Tuple to a C++ <code>std::vector</code>	24
5.3	C++ <code>std::vector</code> to a Python Tuple	25
5.4	Python Tuple of bytes to a C++ <code>std::vector<std::string>></code>	25
5.5	C++ <code>std::vector<std::string>></code> to a Python Tuple of bytes	26
5.6	Python Dict of [float, float] to a C++ <code>std::unordered_map<double, double></code>	26
5.7	Python Dict of [bytes, bytes] to a C++ <code>std::unordered_map<std::string, std::string></code>	27
6	Indices and tables	29

INTRODUCTION

Python is well known for its ability to handle *heterogeneous* data in containers such as lists. But what if you need to interact with C++ containers such as `std::vector<T>` that require *homogeneous* data types?

This project is about converting Python containers (list, dict, set, tuple) containing homogeneous types to and from their C++ equivalent.

1.1 TODO

1.1.1 Alternatives

Buffer protocol

`multiprocessing.shared_memory`

`numpy` is a common example.

1.2 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>`.

To convert that Python list to a C++ `std::vector<double>` 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...
}
```

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. Perhaps by adding code such as:

```
PyObject *example(PyObject *op) {
    // Do something in C++ with the vector...
    // Convert the vector back to a Python list.
    // Function to be defined...
    return read_from_vector(vec);
}
```

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

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

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

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

```
PyObject *read_from_vector(const std::vector<double> &vec) {
    PyObject *ret = PyList_New(vec.size());
    for (size_t i = 0; i < vec.size(); ++i) {
        PyList_SET_ITEM(ret, i, PyFloat_FromDouble(vec[i]));
    }
    return ret;
}
```

This is fine, except notice the absence of error handling above, and all errors are runtime errors. And runtime errors are much harder to deal with than compile time errors.

TODO:

Until you want to support other object types, say `int`, `bytes`. It gets worse when you want to support other containers such as `(tuple, set, dict)`. Then you have to write individual conversion functions for all the combinations of object types *and* containers.

The problem with this approach is that functions have to be written for every combination of interesting types (`float`, `int`, `bytes`) **and** every container of interest (`tuple`, `list`, `set`, `dict`). This is tedious and error prone.

Also, notice the absence of error handling above, and all errors are runtime errors.

1.3 Why This Project

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

If we want to support a fairly basic set of types:

Table 1: Supported Object types.

Python Type	C++ Type
True, False	bool
int	long
float	double
bytes	std::string

And a basic set of containers:

Table 2: Supported Containers.

Python Type	C++ Type
<code>tuple</code>	<code>std::vector</code>
<code>list</code>	<code>std::vector</code>
<code>set</code>	<code>std::unordered_set</code>
<code>frozenset</code>	<code>std::unordered_set</code>
<code>dict</code>	<code>std::unordered_map</code>

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

For example for unary containers (`tuple`, `list`, `set`, `frozenset`) there are four containers with four types. Each container/type combination requires two functions to give two way conversion from Python to C++ and back. Thus $4 \text{ (containers)} * 4 \text{ (types)} * 2 \text{ (way conversion)} = 32$ functions required.

For `dict` there are four types but the key and the value can be either so 16 possible variations (any 2 out of 4, as given by `itertools.product('ABCD', repeat=2)`). With two way conversion this means 32 conversion functions.

Thus the combination of these types and containers would require 64 conversion functions to write, document and debug.

The aim of this project is to reduce this amount of code to a manageable level, namely 6 hand written functions to cover all these cases.

It uses a mix of C++ templates and code generators to achieve this. The result is a few small functions for objects and a pair of hand written templates for each container. As lists and tuples are very similar the code is reduced even further.

The only hand written code that needs to be maintained is for the two-way conversions for any type are:

- Two C++ templates for Python `tuple` and `list` two way conversions for all types.
- Two C++ templates for Python `set` and `frozenset` two way conversions for all types.
- Two C++ templates for Python `dict` two way conversions for all types combinations.

This reduces the number of maintained functions from 64 down to 6. Then a Python script is used to generate the C++ code that declares and defines all 64 specialised functions. This code generation script means it requires minimal work to add further types or containers.

1.4 Hand Written Functions

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

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

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

This generic function that works with unary Python containers (tuple and list) has a signature that looks like this:

```
template<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
generic_py_unary_to_cpp_std_vector(PyObject *op, std::vector<T> &vec);
```

This template has these parameters:

Table 3: `generic_py_unary_to_cpp_std_vector()` template parameters.

Template Parameter	Notes
<code>PyObject_Check</code>	A pointer to a function that checks that any <code>PyObject *</code> in the Python container is the correct type, for example that it is a bytes object.
<code>PyObject_Convert</code>	A pointer to a function that converts any <code>PyObject *</code> in the Python container to the C++ type, for example from bytes -> <code>std::string</code> .
<code>PyUnaryContainer_Check</code>	A pointer to a function that checks that the <code>PyObject *</code> argument is the correct container type, for example a tuple.
<code>PyUnaryContainer_Size</code>	A pointer to a function that returns the size of the Python container.
<code>PyUnaryContainer_Get</code>	A pointer to a function that gets a <code>PyObject *</code> from the Python container at a given index.

The function has the following parameters.

Table 4: `generic_py_unary_to_cpp_std_vector()` parameters.

Type	Name	Notes
<code>PyObject *</code>	<code>op</code>	The Python container to read from.
<code>std::vector<T></code>	<code>vec</code>	The C++ to write to.

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

This template can be partially specialised for converting Python *lists* of any type to C++ `std::vector<T>`. This is hand written code but is just has a single function call. Note the use of the function pointers to `py_list_check`, `py_list_len` and `py_list_get`.

```
template<typename T, int (*PyObject_Check)(PyObject *), T (*PyObject_Convert)(PyObject_
↪ *)>
int generic_py_list_to_cpp_std_vector(PyObject *op, std::vector<T> &vec) {
    return generic_py_unary_to_cpp_std_vector<T, PyObject_Check, PyObject_Convert,
        &py_list_check, &py_list_len, &py_list_get>(op, vec);
}
```


1.5 Generated Functions

These are created by a script that takes the cartesian product of object types and container types and creates functions for each container/object. For example, to convert a Python list of float to a C++ `std::vector<double>` the following are created:

A base declaration in *auto_py_convert_internal.h*:

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

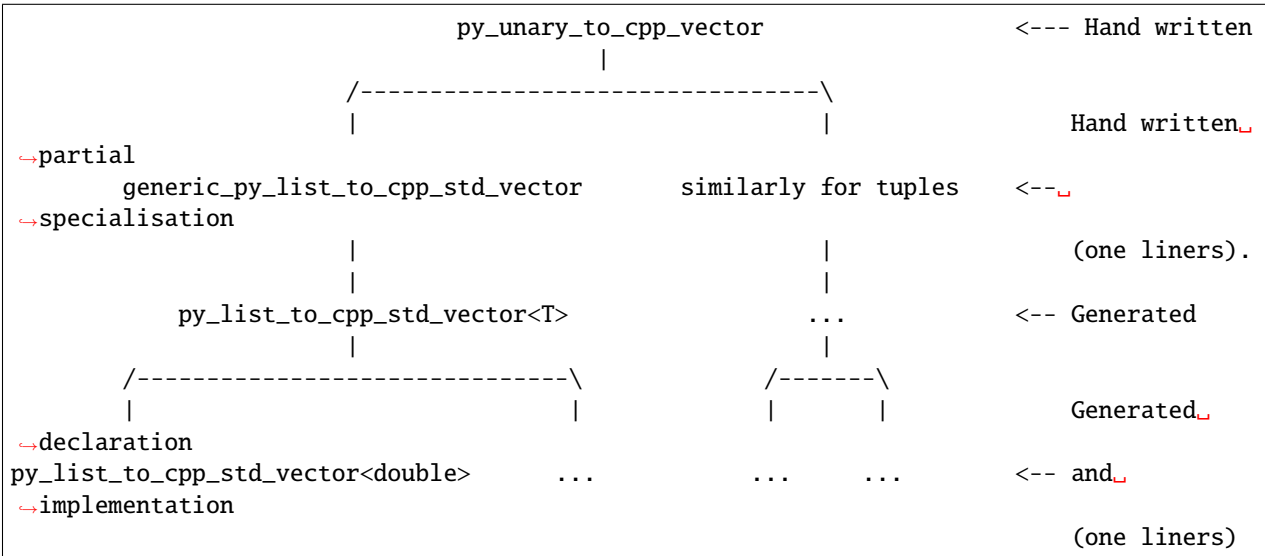
And a concrete declaration for each C++ target type T in *auto_py_convert_internal.h*:

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

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

```
template <>
int
py_list_to_cpp_std_vector<double>(PyObject *op, std::vector<double> &container) {
    return generic_py_list_to_cpp_std_vector<double>, &py_float_check, &py_float_to_cpp_
    double>(
        op, container
    );
}
```

This is the function hierarchy for the code that converts Python lists and tuples to C++ `std::vector` for all object types:



1.5.1 Usage

Using the concrete function is as simple as this:

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

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

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

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

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

The generic function signature looks like this:

```
template<typename T,
        PyObject *(*ConvertCppToPy)(const T &),
        PyObject *(*PyUnaryContainer_New)(size_t),
        int(*PyUnaryContainer_Set)(PyObject *, size_t, PyObject *)>
PyObject *
generic_cpp_std_vector_to_py_unary(const std::vector<T> &vec);
```

USING THIS LIBRARY IN YOUR C++ CODE

2.1 The Basics

2.1.1 Code Generation

If necessary run the code generator:

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

Which should give you something like:

```
Target directory "src/cpy"
Writing declarations to "src/cpy/auto_py_convert_internal.h"
Wrote 910 lines of code with 66 declarations.
Writing definitions to "src/cpy/auto_py_convert_internal.cpp"
Wrote 653 lines of code with 64 definitions.

Process finished with exit code 0
```

2.1.2 Build Configuration

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

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

2.1.3 Source Inclusion

Your pre-processor needs access to the header files with the compiler flag:

```
-I src/cpy
```

Then in your Python extension include the line:

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

And this gives you access to the whole API.

2.2 Examples

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

To build this extension:

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

And to use it:

```
import cPyCppContainer
```

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

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

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

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

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

```
using namespace Python_Cpp_Containers;
```

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

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

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

```
/** Create a new list of floats with doubled values. */
static PyObject *
list_x2(PyObject *Py_UNUSED(module), PyObject *arg) {
    std::vector<double> vec;
    // py_list_to_cpp_std_vector() 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_vector(arg, vec)) {
        // Double the values in pure C++ code.
        vector_double_x2(vec);
        // cpp_std_vector_to_py_list() returns NULL on failure
        // and a Python exception will be set.
        return cpp_std_vector_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_vector()` implementation does not exist for double there will be a compile time error.
- Giving `py_list_to_cpp_std_vector()` anything other than a list of floats will create a Python runtime error.
- If `cpp_std_vector_to_py_list()` fails for any reason there will be a Python runtime error.

Using the Extension

Once the extension is built you can use it thus:

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

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

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

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

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

2.2.2 Reversing a tuple of bytes in C++

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

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

Here is the extension code that call this:

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

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

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

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

2.2.3 Incrementing dict values in C++

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

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

Once the extension is built this can be used thus:

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

EXAMPLES BY CONTAINER

3.1 Python Tuples

3.1.1 Converting a Python Tuple to a C++ `std::vector`

Here is some demonstration code that takes a Python tuple of floats then converts that to C++ vector of doubles with a single function call:

```
#include "python_convert.h"

// Demonstration code.
void test_example_py_tuple_to_vector_double(PyObject *op) {
    // Create the vector of the appropriate type.
    std::vector<double> cpp_vector;
    // Copy the tuple to a vector
    int err = Python_Cpp_Containers::py_tuple_to_cpp_std_vector(op, cpp_vector);
    if (err != 0) {
        // Handle error
        // ...
    } else {
        // Use C++ vector.
        // ...
    }
}
```

`Python_Cpp_Containers::py_tuple_to_cpp_std_vector` has implementations for vectors of `bool`, `long`, `double` and `std::string`.

3.1.2 Converting a C++ `std::vector` to a Python Tuple

Here is some demonstration code that creates a C++ vector of doubles then converts that to a Python tuple with a single function call:

```
#include "python_convert.h"

PyObject *test_example_vector_to_py_tuple_double() {
    // An imaginary function that creates a C++ std::vector<double>
    std::vector<double> cpp_vector = get_cpp_vector_doubles();
    // Convert to a Python tuple that contains floats
    PyObject *op = Python_Cpp_Containers::cpp_std_vector_to_py_tuple(cpp_vector);
}
```

(continues on next page)

(continued from previous page)

```
    if (! op) {  
        // Handle error condition.  
        // ...  
    }  
    return op;  
}
```

Python_Cpp_Containers::cpp_std_vector_to_py_tuple has implementations for vectors of bool, long, double and std::string.

3.2 Python Lists

3.2.1 Converting a Python List to a C++ std::vector

This is done with Python_Cpp_Containers::cpp_std_vector_to_py_list which is very similar to the code for tuples. Python_Cpp_Containers::cpp_std_vector_to_py_list has implementations for vectors of bool, long, double and std::string.

3.2.2 Converting a C++ std::vector to a Python List

This is done with Python_Cpp_Containers::py_list_to_cpp_std_vector which is very similar to the code for tuples. Python_Cpp_Containers::py_list_to_cpp_std_vector has implementations for vectors of bool, long, double and std::string.

3.3 Python Sets

3.3.1 Converting a Python Set to a C++ std::unordered_set

This is done with Python_Cpp_Containers::cpp_std_unordered_set_to_py_set which is very similar to the code for tuples and lists. Python_Cpp_Containers::cpp_std_unordered_set_to_py_set has implementations for the C++ types of bool, long, double and std::string.

3.3.2 Converting a C++ std::unordered_set to a Python Set

This is done with Python_Cpp_Containers::py_list_to_cpp_std_unordered_set which is very similar to the code for tuples and lists. Python_Cpp_Containers::py_list_to_cpp_std_unordered_set has implementations for C++ types of bool, long, double and std::string.

3.4 Python Dicts

3.4.1 Converting a Python dict to a C++ `std::unordered_map`

This is done with `Python_Cpp_Containers::py_dict_to_cpp_std_unordered_map`. This has implementations for all the combinations of C++ types of `bool`, `long`, `double` and `std::string` as keys and values so there are 16 combinations.

Here is an example of converting a Python dict of `[int, bytes]` to a C++ `std::unordered_map<long, std::string>`:

```
#include "python_convert.h"

void test_example_py_dict_to_cpp_std_unordered_map(PyObject *op) {
    std::unordered_map<long, std::string> cpp_map;
    int err = Python_Cpp_Containers::py_dict_to_cpp_std_unordered_map(op, cpp_map);
    if (err != 0) {
        // Handle error.
        // ...
    } else {
        // Do something with cpp_map
        // ...
    }
}
```

3.4.2 Converting a C++ `std::unordered_map` to a Python dict

This is done with `Python_Cpp_Containers::cpp_std_unordered_map_to_py_dict`. This has implementations for all the combinations of C++ types of `bool`, `long`, `double` and `std::string` as keys and values so there are 16 combinations.

Here is an example of converting a C++ `std::unordered_map<long, std::string>` to a Python dict of `[int, bytes]`:

```
#include "python_convert.h"

PyObject *test_example_cpp_std_unordered_map_to_py_dict() {
    // An imaginary function that creates a C++ std::unordered_map<long, std::string>
    std::unordered_map<long, std::string> cpp_map = get_cpp_map();
    // Convert to a Python dict.
    PyObject *op = Python_Cpp_Containers::cpp_std_unordered_map_to_py_dict(cpp_map);
    if (!op) {
        // Handle error.
        // ...
    }
    return op;
}
```

3.5 Matrix Example

Supposing there is a C++ library that provides matrix support for a `std::vector<std::vector<double>>` type and you want it to work on a Python tuple of tuples of floats.

Firstly creating the C++ matrix from Python.

3.5.1 Converting a Python Tuple[Tuple[float]] to a C++ `std::vector<std::vector<double>>`

```
#include "python_convert.h"

// Demonstration code.
void py_matrix_to_cpp_matrix(PyObject *op) {
    // Create the matrix of the appropriate type.
    std::vector<std::vector<double>> cpp_matrix;
    for (Py_ssize_t i = 0; i < Python_Cpp_Containers::py_tuple_len(op), ++i) {
        std::vector<double> cpp_vector;
        PyObject *row = Python_Cpp_Containers::py_tuple_get(op, i);
        int err = Python_Cpp_Containers::py_tuple_to_cpp_std_vector(row, cpp_vector);
        if (err != 0) {
            // Handle error
            // ...
            return;
        } else {
            cpp_matrix.push_back(cpp_vector);
        }
    }
    // Use the matrix
    some_function_that_uses_a_matrix(cpp_matrix);
}
```

Note: Some error checking omitted.

3.5.2 Converting a C++ `std::vector<std::vector<double>>` to a Python Tuple[Tuple[float]]

And the reverse, given a C++ matrix this converts that to a Python tuple of tuples with a single function call:

```
#include "python_convert.h"

PyObject *
cpp_matrix_to_py_matrix() {
    // An imaginary function that creates a C++ std::vector<double>
    std::vector<std::vector<double>> cpp_matrix = get_cpp_matrix();
    PyObject *op = Python_Cpp_Containers::py_tuple_new(cpp_matrix.size());
    for (size_t i = 0; i < cpp_matrix.size(); ++i) {
        PyObject *row = Python_Cpp_Containers::cpp_std_vector_to_py_tuple(cpp_matrix[i]);
        if (! row) {
```

(continues on next page)

(continued from previous page)

```
        // Handle error condition.
        // ...
        return NULL;
    }
    int err = Python_Cpp_Containers::py_tuple_set(op, i, row)
    if (err != 0) {
        // Handle error
        // ...
        return;
    }
}
return op;
}
```

Note: Some error checking omitted.

4.1 python_convert.h

This is a hand written file that contains:

- Functions to convert objects to and fro between Python and C++. See “Conversion of Objects” below.
- Function to create and access Python containers such as tuples, lists etc. See “Creating and Accessing Python Containers” below.
- Templates that convert containers to and fro between Python and C++. These are described in detail below.

4.1.1 Conversion of Objects

These are wrappers around existing Python C API functions. There are the following functions for each supported object type:

- Check that a Python `PyObject *` is of the correct object type.
- Convert a Python `PyObject *` to a C++ fundamental type.
- Create a new Python `PyObject *` from a C++ fundamental type.

Their implementations are in `python_convert.cpp`

For example for Python floats to and from a C++ `double` there will be these function implementations:

```
PyObject *cpp_double_to_py_float(const double &d) {
    return PyFloat_FromDouble(d);
}

double py_float_to_cpp_double(PyObject *op) {
    assert(py_float_check(op));
    return PyFloat_AsDouble(op);
}

int py_float_check(PyObject *op) {
    return PyFloat_Check(op);
}
```

4.1.2 Creating and Accessing Python Containers

These are wrappers around existing Python C API functions. There are the following functions for each supported object type:

- Check that a Python PyObject * is of the correct container type.
- Create a new Python PyObject * container.
- Set a new PyObject * in the container.
- Get a new PyObject * from the container.

Their implementations are in `python_convert.cpp`

For example for a Python tuple there will be these function implementations:

```
// Tuple check, create, set, get functions.
int py_tuple_check(PyObject *op) {
    return PyTuple_Check(op);
}

PyObject *py_tuple_new(size_t len) {
    return PyTuple_New(len);
}

Py_ssize_t py_tuple_len(PyObject *op) {
    return PyTuple_Size(op);
}

int py_tuple_set(PyObject *tuple_p, size_t pos, PyObject *op) {
    PyTuple_SET_ITEM(tuple_p, pos, op);
    return 0;
}

PyObject *py_tuple_get(PyObject *tuple_p, size_t pos) {
    return PyTuple_GET_ITEM(tuple_p, pos);
}
```

4.2 Conversion Templates

4.3 Python Lists and Tuples

4.3.1 Conversion From a `std::vector<T>` to a Python List or Tuple

```
template<typename T,
        PyObject *(*Convert)(const T &),
        PyObject *(*PyUnary_New)(size_t),
        int(*PyUnary_Set)(PyObject *, size_t, PyObject *)>
PyObject *
generic_cpp_std_vector_to_py_unary(const std::vector<T> &vec);
```

Table 1: Convert a `std::vector` to a Python Tuple or List.

Type	Description
<code>typename T</code>	The C++ type of the object.
<code>PyObject *(*Convert)(const T &)</code>	A pointer to a function that takes a type T and returns a new Python object.
<code>PyObject *(*PyUnary_New)(size_t)</code>	A pointer to a function that returns a new Python container of the given length.
<code>int(*PyUnary_Set)(PyObject *, size_t, PyObject *)></code>	Sets a Python object in the Python container at the given position.

This template is then partially specified for both tuples and lists of type T:

```
template<typename T, PyObject *(*Convert)(const T &)>
PyObject *
generic_cpp_std_vector_to_py_tuple(const std::vector<T> &vec) {
    return generic_cpp_std_vector_to_py_unary<T,
                                          Convert,
                                          &py_tuple_new,
                                          &py_tuple_set>(vec);
}

template<typename T, PyObject *(*Convert)(const T &)>
PyObject *
generic_cpp_std_vector_to_py_list(const std::vector<T> &vec) {
    return generic_cpp_std_vector_to_py_unary<T,
                                          Convert,
                                          &py_list_new,
                                          &py_list_set>(vec);
}
```

Then these are specialised by auto-generated in `auto_py_convert_internal.h` code for the types `bool`, `long`, `double` and `std::string`. Their declarations are:

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

// Instantiations
template <>
PyObject *
cpp_std_vector_to_py_tuple<bool>(const std::vector<bool> &container);

template <>
PyObject *
cpp_std_vector_to_py_tuple<long>(const std::vector<long> &container);

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

template <>
```

(continues on next page)

(continued from previous page)

```
PyObject *
cpp_std_vector_to_py_tuple<std::string>(const std::vector<std::string> &container);
```

Their declarations are auto-generated in `auto_py_convert_internal.cpp`:

```
template <>
PyObject *
cpp_std_vector_to_py_tuple<bool>(const std::vector<bool> &container) {
    return generic_cpp_std_vector_to_py_tuple<bool, &cpp_bool_to_py_bool>(container);
}

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

template <>
PyObject *
cpp_std_vector_to_py_tuple<double>(const std::vector<double> &container) {
    return generic_cpp_std_vector_to_py_tuple<double, &cpp_double_to_py_float>
    ↪(container);
}

template <>
PyObject *
cpp_std_vector_to_py_tuple<std::string>(const std::vector<std::string> &container) {
    return generic_cpp_std_vector_to_py_tuple<std::string, &cpp_string_to_py_bytes>
    ↪(container);
}
```

4.3.2 Conversion From a Python List or Tuple to a `std::vector<T>`

```
template<typename T,
        int (*Check)(PyObject *),
        T (*Convert)(PyObject *),
        int (*PyUnary_Check)(PyObject *),
        Py_ssize_t (*PyUnary_Size)(PyObject *),
        PyObject *(*PyUnary_Get)(PyObject *, size_t)>
int generic_py_unary_to_cpp_std_vector(PyObject *op, std::vector<T> &vec);
```


Table 2: Convert a `std::vector` to a Python Tuple or List.

Type	Description
<code>typename T</code>	The C++ type of the object.
<code>int (*Check)(PyObject *)</code>	A pointer to a function returns true if Python object can be converted to type T.
<code>int(*PyUnary_Check)(PyObject *)</code>	A pointer to a function that returns true if the given Python container of the correct type (list or tuple respectively).
<code>Py_ssize_t(*PyUnary_Size)(PyObject *)</code>	A pointer to a function that returns the size of the Python container.
<code>PyObject *(*PyUnary_Get)(PyObject *, size_t)</code>	Gets a Python object in the Python container at the given position.

This template is then partially specified for both tuples and lists of type T:

```
template<typename T, int (*Check)(PyObject *), T (*Convert)(PyObject *)>
int generic_py_tuple_to_cpp_std_vector(PyObject *op, std::vector<T> &vec) {
    return generic_py_unary_to_cpp_std_vector<T,
                                          Check,
                                          Convert,
                                          &py_tuple_check,
                                          &py_tuple_len,
                                          &py_tuple_get>(op, vec);
}

template<typename T, int (*Check)(PyObject *), T (*Convert)(PyObject *)>
int generic_py_list_to_cpp_std_vector(PyObject *op, std::vector<T> &vec) {
    return generic_py_unary_to_cpp_std_vector<T,
                                          Check,
                                          Convert,
                                          &py_list_check,
                                          &py_list_len,
                                          &py_list_get>(op, vec);
}
```

Then these are specialised by auto-generated in `auto_py_convert_internal.h` code for the types `bool`, `long`, `double` and `std::string`. Their declarations for tuple are (similarly for lists):

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

// Instantiations
template <>
int
py_tuple_to_cpp_std_vector<bool>(PyObject *tuple, std::vector<bool> &container);

template <>
int
py_tuple_to_cpp_std_vector<long>(PyObject *tuple, std::vector<long> &container);
```

(continues on next page)

(continued from previous page)

```

template <>
int
py_tuple_to_cpp_std_vector<double>(PyObject *tuple, std::vector<double> &container);

template <>
int
py_tuple_to_cpp_std_vector<std::string>(PyObject *tuple, std::vector<std::string> &
↳ container);

```

Their definitions for tuple are auto-generated in `auto_py_convert_internal.cpp` (similarly for lists):

```

template <>
int
py_tuple_to_cpp_std_vector<bool>(PyObject *op, std::vector<bool> &container) {
    return generic_py_tuple_to_cpp_std_vector<bool, &py_bool_check, &py_bool_to_cpp_bool>
↳ (op, container);
}

template <>
int
py_tuple_to_cpp_std_vector<long>(PyObject *op, std::vector<long> &container) {
    return generic_py_tuple_to_cpp_std_vector<long, &py_long_check, &py_long_to_cpp_long>
↳ (op, container);
}

template <>
int
py_tuple_to_cpp_std_vector<double>(PyObject *op, std::vector<double> &container) {
    return generic_py_tuple_to_cpp_std_vector<double, &py_float_check, &py_float_to_cpp_
↳ double>(op, container);
}

template <>
int
py_tuple_to_cpp_std_vector<std::string>(PyObject *op, std::vector<std::string> &
↳ container) {
    return generic_py_tuple_to_cpp_std_vector<std::string, &py_bytes_check, &py_bytes_to_
↳ cpp_string>(op, container);
}

```

PERFORMANCE

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

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

Model Name:	MacBook Pro
Model Identifier:	MacBookPro15,2
Processor Name:	Intel Core i7
Processor Speed:	2.7 GHz
Number of Processors:	1
Total Number of Cores:	4
L2 Cache (per Core):	256 KB
L3 Cache:	8 MB
Hyper-Threading Technology:	Enabled
Memory:	16 GB
System Version:	macOS 10.14.6 (18G9323)
Kernel Version:	Darwin 18.7.0

5.1 Summary

- Sequences of fundamental types are converted at around 100m objects/sec.
- Sequences of strings are converted at a memory rate of around 4000 Mb/sec.
- Dicts are about 8-10x slower. Why this should be so different from the expected 2x is a mystery at the moment.

5.1.1 Fundamental Types

Converting and copying of `int/long` and `float/double` takes about 0.01 μ s per object (100m objects per second) for large containers. This corresponds to around 800 Mb/s. `boolean/bool` is around 2x to 5x faster.

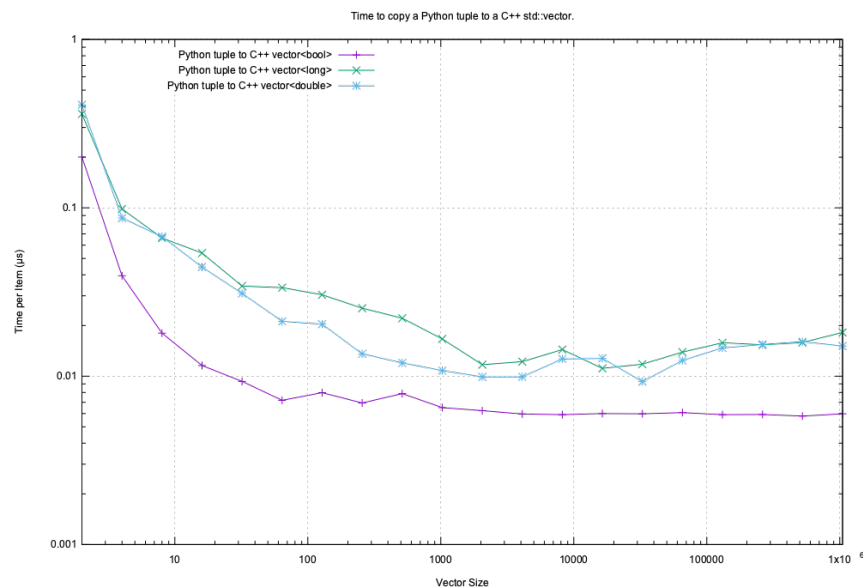
5.1.2 Strings of Different Lengths

With `bytes/std::string` converting and conversion takes about the following. The performance appears appears linear (with some latency for small strings):

String size	~Time per object (μ s)	~Rate, million per second	~Rate x Size Mb/s
8	0.02	50	400
64	0.03	30	2000
512	0.1	10	5000
4096	1.0	1	4000

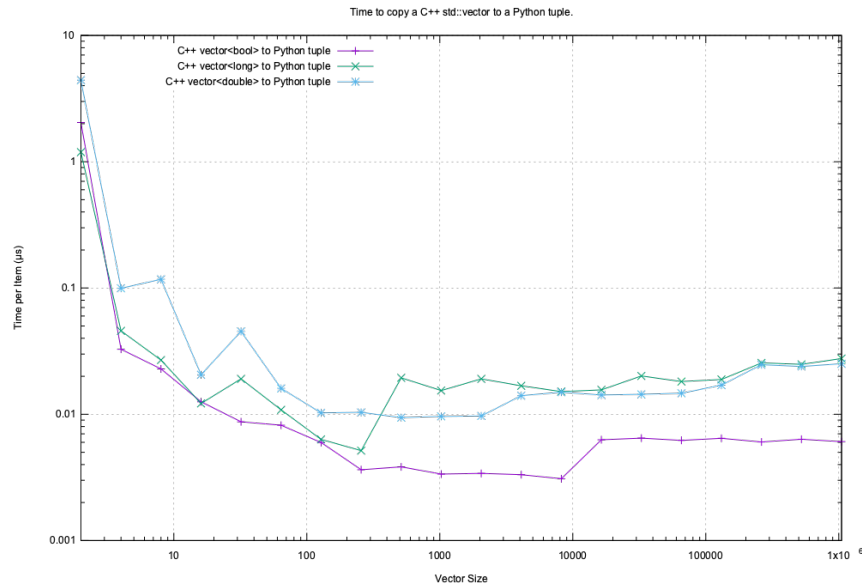
5.2 Python Tuple to a C++ `std::vector`

Here is an example of converting a Python tuple to a C++ `std::vector<T>` for up to 1m bool, long and double types. Time is per-object in μ s. So 1m float/long conversion takes about 10 to 20 ms.



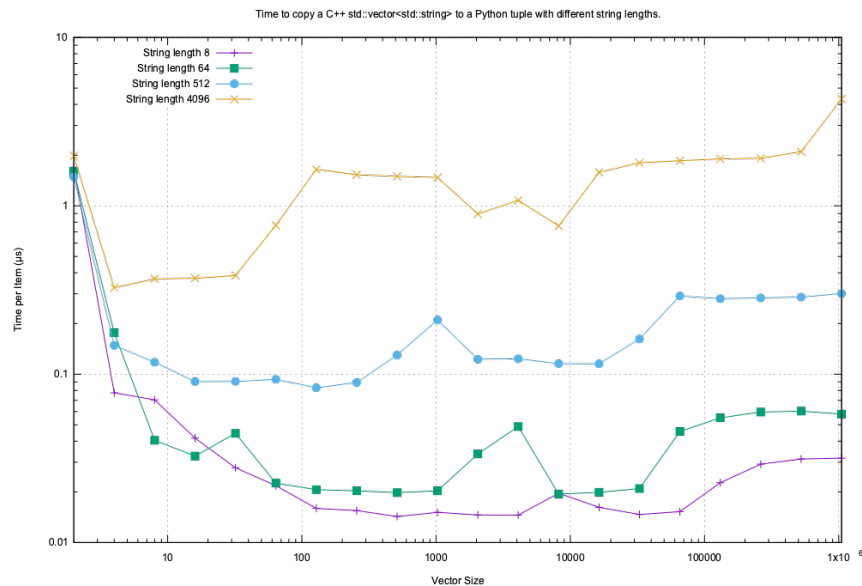
5.3 C++ `std::vector` to a Python Tuple

This is the reverse of the above, the time to convert a C++ `std::vector<T>` to a Python tuple for up to 1m bool, long and double types.



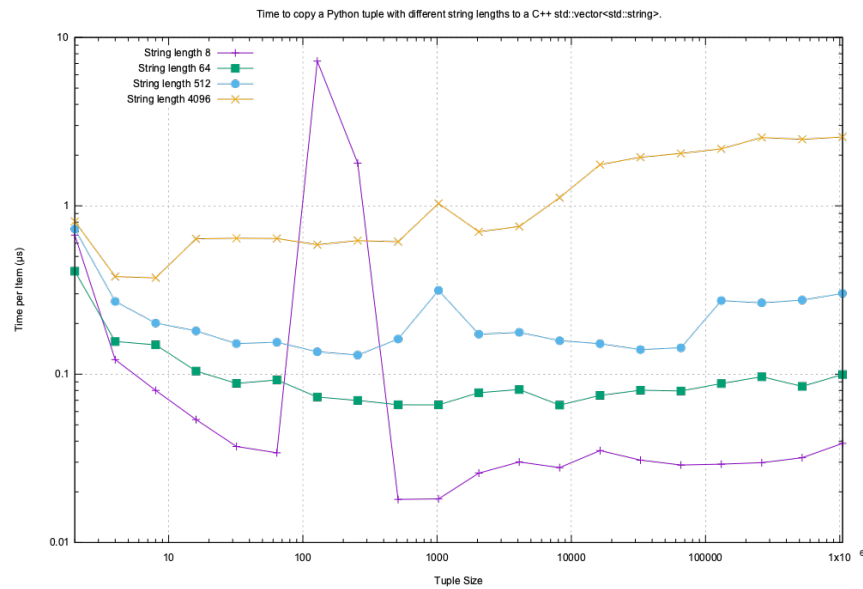
5.4 Python Tuple of bytes to a C++ `std::vector<std::string>`

This shows the conversion cost of various length strings.



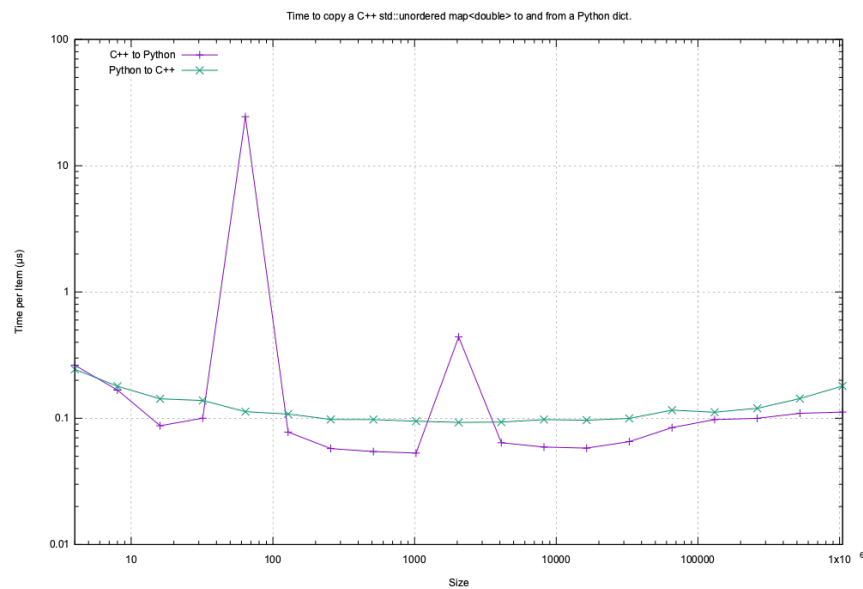
5.5 C++ `std::vector<std::string>` to a Python Tuple of bytes

This is the reverse of the above.



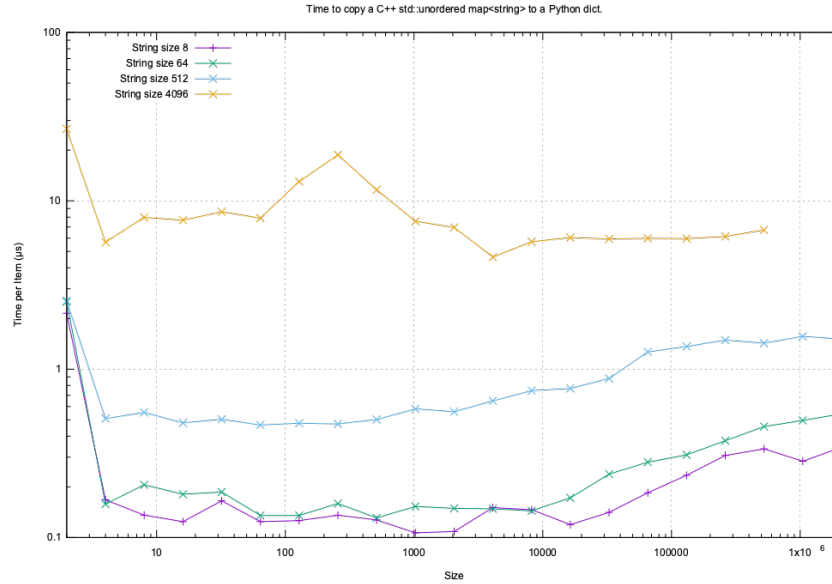
5.6 Python Dict of [float, float] to a C++ `std::unordered_map<double, double>`

This shows the conversion rate of a dict of floats to and from Python. At 0.1 µs per item (10m objects/s) this rate is about one-tenth of the rate of converting a sequence.



5.7 Python Dict of [bytes, bytes] to a C++ `std::unordered_map<std::string, std::string>`

Similarly for dicts of bytes. This corresponds, roughly, to a data rate of around 500 Mb/s.



INDICES AND TABLES

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