

# Python String<->float/int Conversion Optimization for IBM Z

Intern (and Author): Jonathan Alexander  
Mentor: Andreas Krebbel  
June-August 2022

## Contents

About.....	2
Overview .....	2
Main Goals .....	2
Test Cases.....	2
Designs .....	3
Design 1.....	3
Main advantages.....	3
Main drawbacks .....	3
Design 2.....	3
Main advantages.....	3
Main drawbacks .....	3
Designs 3 and 4 .....	3
Main advantages.....	3
Main drawbacks .....	4
Other Considerations .....	4
Appendices.....	5
Appendix A – Important terms .....	5
Appendix B - Useful Links.....	6
Appendix C - Setting Up a Z-O/S Server .....	7
Appendix D – Test Cases’ Code .....	8

## About

This report was written to be a “get going fast” document.

## Overview

Converting string to float/int and float/int to string requires expensive operations such as  $\div 10$  and  $\times 10$ . Adding an intermediate step of converting to decimal (in both directions) should accelerate the process.

[Libmpdec](#) is the open-source library currently used by python to support Decimal Floating Point (DFP) operations. This is done by using the `_Decimal32/64/128` types available in the library. The library supports all the usual operators, as well as many special cases. While it claims to be a fast software implementation it has no enhancements specifically for the IBM zSystems. The project’s aim is to add those hardware specific enhancements and by that improve the Python String<->float/int conversion performance.

According to test cases ran on a LinuxOne server (runs on a Z mainframe system), it is possible to accelerate the basic arithmetic operations of DFPs by 83% (+ and -), 65% ( $\times$ ), and 94% ( $\div$ ). See under Test Cases for exact results; the code is available in Appendix D – Test Cases’ Code.

## Main Goals

Maximize Z-O/S’s processing potential for floating point arithmetic by using platform specific enhancements.

Optimizing the Python’s string to float/int and float/int to string conversion in the IBM Z environment.

## Test Cases

All four test cases are built identically (See the abstract below the table, and the complete code in Appendix D – Test Cases’ Code), and the results are as follows:

#	Operation	Value 1	Value 2	Time (libmpdec)	Time (libdfp)	Decrease (%)	New Time (%)
1	Addition	1.23456	5.6789	0.038360	0.006355	83.433%	16.567%
2	Subtraction	1.23456	5.6789	0.038947	0.006363	83.662%	16.338%
3	Multiplication	1.23456	5.6789	0.026600	0.009155	65.583%	34.417%
4	Division	98765.4	4.3201	0.141570	0.008468	94.019%	5.981%

Table 1

The code’s abstract:

1. Main method initializes four variables, two for libmpdec and two when using libdfp.
2.  $r$  is assigned to carry the result and is first assigned the value of  $a$  (*Value 1* in the table above).
3. Eight *for* loops, each with 1,000,000 iterations, containing the respective arithmetic operation (addition, subtraction, multiplication, or division) for each calculation method. In all cases, loop is:
  - a.  $r = r[op]b$ , where  $[op]$  is one of the arithmetic operations ( $+-\div\times$ ) and  $b$  is *Value 2* in the table above.
4. Each *for* loop is timed individually.
5. The figures were then taken to excel and compared using two formulas, with the results presented in the last two columns. The table is below, see Table 1.
6. The formulas:
7. Decrease (%):  $\frac{m-d}{m \times 100\%}$  where  $m$  is libmpdec’s duration and  $d$  is libdfp’s duration.
8. New Time (%):  $\frac{d}{m \times 100\%}$  where  $m$  is libmpdec’s duration and  $d$  is libdfp’s duration.

## Designs

### Design 1

Use libmpdec and utilize the existing API, only changing the underlying code. This is the design option we most explored up to this point and is likely to be the least complex and yet deliver the best results overall.

This design will require contacting libmpdec's current maintainer(s) and see if they are open to including hardware specific implementations.

#### Main advantages

- Using an existing API could mean one less thing to plan for.
- Software that already uses the library on IBM Z will experience a healthy boost to performance.
- Some software may not need to be updated to accommodate the change as it will be integrated under an existing umbrella.
- Will be easier to make it backwards compatible.

#### Main drawbacks

- An open-source library means constant maintenance may be required
- The current maintainer(s) may or may not be interested in the addition.
- The current maintainer(s) do not have access to an IBM-Z system, making future maintenance potentially more complex.

### Design 2

Map the python operations to libdfp - a library to support DFP hardware operations. It is more complex and is likely to also require the use of libmpdec.

This is a good option if the first design falls through due to current maintainer(s) of libmpdec refusing to add the additional code.

#### Main advantages

- Doesn't require the agreement of additional parties.
- Can use the existing libmpdec API and change the underlying code.

#### Main drawbacks

- More complex.
- Will require constant maintenance to make backwards compatible.

### Designs 3 and 4

Implement a new library from scratch using compiler extensions or a whole new library entirely for IBM-Z, using C or even assembly.

The nature of doing either will by default allow for better speeds and optimization, since it will be designed with those goals in mind and with ZSystems in mind, but it does require more work during the entire process – no code is fool proof.

#### Main advantages

- Designing something new gives better control of outcome.
- May allow better optimization.
- Designed specifically for IBM Z means it is far less likely to have problem

## Main drawbacks

- It may be more work than it is worth.
- Far more complex.
- May overcomplicate what could be very simple.
- Could introduce new bugs.
- May not work as intended.
- May be challenging (or impossible) to make backwards compatible.
- Programmers will need to specifically choose the new library and work with its API.

## Other Considerations

- This is only the first part of a larger project. The second (or eventual) step will involve hardware specific String<->float/int conversion optimization for Python in IBM Z.
- It is important to have some level of API integration with libmpdec, to make the code backwards compatible.
- Making a new package **may** allow further optimizations in other languages and/or packages. There is no guarantee for that happening and shouldn't be a main goal (at least not yet).
- Creating a new library is far more time consuming than using an existing API, but it could (potentially) allow a far better optimization than what otherwise could have been.
- The fastest option is naturally the first design, but it may not be possible due to 3<sup>rd</sup> parties.
- Some compilers can accelerate execution of code through things like loop unrolling and improving locality.
- IBM Z uses big Endian. It may have an impact during the final step when optimizing the conversion itself.

## Appendices

### Appendix A – Important terms

**Mainframe** – high performance computers with large amounts of memory and processors, capable of processing billions of simple calculations and transactions in real time. Critical for commercial databases, transaction servers, and applications requiring high resiliency, security, and agility <sup>[1]</sup>. For [more information](#).

**IBM zSystems** – a family name that's used by IBM for all of its z/Architecture mainframe computers <sup>[2]</sup>. The “Z” stands for “Zero Downtimes” <sup>[2]</sup>. Usually runs on Linux. For [more information](#).

**Binary Floating Point (BFP)** – floating point numbers with a base 2 encoding of the mantissa. In C and C++, float stands for 32-bit BFP, double for 64-bit BFP, and long double for a 128-bit BFP data type. In Python the type *float* usually means 64-bit BFP (C's *double*), although the mapping may differ.

**Decimal Floating Point (DFP)** – floating point numbers with a base 10 encoding of the mantissa. With the C/C++ extension there are data types `_Decimal32`, `_Decimal64`, and `_Decimal128` for the respective bit widths.

### [Cache-Oblivious Algorithm](#)

**Endianness:** The ordering of multiple bytes which are intended to be interpreted together as a single number.

- **Little-Endian:** The least-significant byte is stored/sent first.
  - Ordering: 0xDD, 0xCC, 0xBB, 0xAA
  - Most new tech uses little endian.
- **Big-Endian:** The most-significant byte is stored/sent first.
  - Ordering: 0xAA, 0xBB, 0xCC, 0xDD
  - IBM Z uses this ordering.

#### Demonstration of Endianness

Storing the integer 0xAABBCCDD in the two orderings will make a mirror:

Address	Big	Little
$n$	AA	DD
$n + 1$	BB	CC
$n + 2$	CC	BB
$n + 3$	DD	AA

## Appendix B - Useful Links

Webpage description	URL
Libmpdec on GitHub.	<a href="#">Libmpdec</a>
Register for use of a LinuxOne server (runs on a Z-system)	<a href="#">Linux1 registration page</a>
LinuxOne login page.	<a href="#">Linux1 login page</a>
IBM's ssh requirements. <b>Important:</b> The ssh command shown on this page uses a "-" instead of "-". The correct command is: <pre>ssh -i /path/to/key/keyname.pem linux1@serveripaddress</pre> the /path.../keyname.pem should be replaced by the automatically generated pem file, download it and place it in the same folder from which you are working, it'll save you some hassle. The ip address is generated automatically in the instance management page. Use "linux1@...." as your login, linux1 is <b>always</b> the user ID. Everything else in the link is explained well.	<a href="#">how to setup and connect to IBM-Z</a> <i>The file you are reading now and this link have nearly nothing in common.</i>
(pdf) All you need to know about IBM-Z, and far more.	<a href="#">Introduction to the New Mainframe</a>
(pdf) Even more to know about IBM-Z.	<a href="#">IBM-Z Principles of Operation</a>
(pdf) IBM-Z's assembly instructions and more.	<a href="#">Z-OS Reference Summary</a>
About the Mainframe	<a href="#">About the Mainframe</a>
IBM zSystems fundamentals: An introductory Q&A	<a href="#">What is IBM Z</a>
IBM Mainframe Assembler – potentially of value.	<a href="#">General Articles on Programming in Assembler</a>
Python: ctypes tutorial	<a href="#">Python/ctypes</a>
Python: how extending works	<a href="#">Python/extending</a>
Python: extending C and C++	<a href="#">Python/extending/extending</a>

Table 2

## Appendix C - Setting Up a Z-O/S Server

Setting up a LinuxOne server proved to be a far more complex task than I imagined.

Using the Linux commands below, and in the same order, could make it easier for you to setup your own instance.

```
sudo apt install make g++
sudo apt install gdb
wget https://www.bytereef.org/software/mpdecimal/releases/mpdecimal-2.5.1.tar.gz
tar xf mpdecimal-2.5.1.tar.gz
cd mpdecimal-2.5.1/
./configure
CFLAGS="-O0 -g3" ./configure
make -j10
export LD_LIBRARY_PATH=mpdecimal-2.5.1/libmpdec
cd ..
```

if you are using mpdecimal, I found that I usually had to use the following commands (in that order) to compile and run the program:

```
cd mpdecimal-2.5.1/
make clean
CFLAGS="-O0 -g3" ./configure
make -j10
export LD_LIBRARY_PATH=mpdecimal-2.5.1/libmpdec
cd ..
```

once I did that, I could compile the program and run it, using:

```
gcc [filename].c -L mpdecimal-2.5.1/libmpdec -l mpdec -O0 -g3 -I mpdecimal-2.5.1/libmpdec
-o [out name] --save-temps
```

## Appendix D – Test Cases' Code

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpdecimal.h>

_Decimal64 __attribute__((noinline, noclone))
dfp_add(_Decimal64 a, _Decimal64 b) {
    return a + b;
}

_Decimal64 __attribute__((noinline, noclone))
dfp_sub(_Decimal64 a, _Decimal64 b) {
    return a - b;
}

_Decimal64 __attribute__((noinline, noclone))
dfp_div(_Decimal64 a, _Decimal64 b) {
    return a / b;
}

_Decimal64 __attribute__((noinline, noclone))
dfp_mult(_Decimal64 a, _Decimal64 b) {
    return a * b;
}

int main(int argc, char **argv) {
    mpd_context_t ctx;
    mpd_t *a, *b;
    mpd_t *result;
    char *rstring;
    char status_str[MPD_MAX_FLAG_STRING];
    clock_t start_clock, end_clock;
    mpd_init(&ctx, 38);
    ctx.traps = 0;
    result = mpd_new(&ctx);
    a = mpd_new(&ctx);
    b = mpd_new(&ctx);

    mpd_set_string(a, "98765.4", &ctx);
    mpd_set_string(b, "4.3201", &ctx);
    start_clock = clock();
    for (int i = 0; i < 1000000; i++)
        mpd_add(result, a, b, &ctx);
    end_clock = clock();
    printf("Test Cases\nValue 1: 1.23456\nValue 2: 5.6789\n");
    printf("Test Case 1: Addition\n");
    fprintf(stderr, "Libmpdec time: %f\n",
        (double) (end_clock - start_clock) / (double) CLOCKS_PER_SEC);
    rstring = mpd_to_sci(result, 1);
    mpd_snprint_flags(status_str, MPD_MAX_FLAG_STRING, ctx.status);
    printf("\nResult: %s %s\n", rstring, status_str);
}

```



```

_Decimal64 aa = 98765.4DD;
_Decimal64 bb = 4.3201DD;
_Decimal64 rresult = aa;
start_clock = clock();
for (int i = 0; i < 1000000; i++)
    rresult = dfp_add(rresult, bb);
end_clock = clock();
fprintf(stderr, "Optimized time: %f\n",
    (double) (end_clock - start_clock) / (double) CLOCKS_PER_SEC);

printf("\nResult: %lx\n", (long) rresult);

printf("Test Case 3: Subtraction\n");
start_clock = clock();
for (int i = 0; i < 1000000; i++)
    mpd_sub(result, a, b, &ctx);
end_clock = clock();
fprintf(stderr, "Libmpdec time: %f\n",
    (double) (end_clock - start_clock) / (double) CLOCKS_PER_SEC);
rstring = mpd_to_sci(result, 1);
mpd_snprint_flags(status_str, MPD_MAX_FLAG_STRING, ctx.status);
printf("\nResult: %s %s\n", rstring, status_str);
rresult = aa;
start_clock = clock();
for (int i = 0; i < 1000000; i++)
    rresult = dfp_sub(rresult, bb);
end_clock = clock();
fprintf(stderr, "Optimized time: %f\n",
    (double) (end_clock - start_clock) / (double) CLOCKS_PER_SEC);

printf("\nResult: %lx\n", (long) rresult);

printf("Test Case 3: Multiplication\n");
start_clock = clock();
for (int i = 0; i < 1000000; i++)
    mpd_mul(result, a, b, &ctx);
end_clock = clock();
fprintf(stderr, "Libmpdec time: %f\n",
    (double) (end_clock - start_clock) / (double) CLOCKS_PER_SEC);
rstring = mpd_to_sci(result, 1);
mpd_snprint_flags(status_str, MPD_MAX_FLAG_STRING, ctx.status);
printf("\nResult: %s %s\n", rstring, status_str);
rresult = aa;
start_clock = clock();
for (int i = 0; i < 1000000; i++)
    rresult = dfp_mult(rresult, bb);
end_clock = clock();
fprintf(stderr, "Optimized time: %f\n",
    (double) (end_clock - start_clock) / (double) CLOCKS_PER_SEC);

printf("\nResult: %lx\n", (long) rresult);

```

```

mpd_set_string(a, "98765.4", &ctx);
mpd_set_string(b, "4.3201", &ctx);
aa = 98765.4DD;
bb = 4.3201DD;

printf("Test Case 4: Division\n");
start_clock = clock();
for (int i = 0; i < 1000000; i++)
    mpd_div(result, a, b, &ctx);
end_clock = clock();
fprintf(stderr, "Libmpdec time: %f\n",
    (double) (end_clock - start_clock) / (double) CLOCKS_PER_SEC);
rstring = mpd_to_sci(result, 1);
mpd_snprint_flags(status_str, MPD_MAX_FLAG_STRING, ctx.status);
printf("\nResult: %s %s\n", rstring, status_str);
rresult = aa;
start_clock = clock();
for (int i = 0; i < 1000000; i++)
    rresult = dfp_div(aa, bb);
end_clock = clock();
fprintf(stderr, "Optimized time: %f\n",
    (double) (end_clock - start_clock) / (double) CLOCKS_PER_SEC);

printf("\nResult: %lx\n", (long) rresult);
mpd_del(a);
mpd_del(b);
mpd_del(result);
mpd_free(rstring);
}

```