See the following code: import xlwings as xw from javascript import require @xw.func def js\_compare\_non\_strict\_equivalent(values): # Check if the input is a list of lists if not isinstance(values[0], list): return "Input must be a list of lists" # Break down the input into two arrays Ar1 = [row[0] for row in values] Ar2 = [row[1] for row in values] # Function to replace None values with 'null' def replace\_none\_with\_null(values): return ['null' if val is None else val for val in values] # Replace None values with 'null' in both arrays Ar1 = replace\_none\_with\_null(Ar1) Ar2 = replace\_none\_with\_null(Ar2) # JavaScript code for non-strict equality check js\_code\_non\_strict = """ function CompareNonStrict(values1, values2) { return values1.map((value, i) => { if (value === 'null' || values2[i] === 'null') { return value === values2[i]; } return value == values2[i]; }); } """ # Use JSPyBridge to run the JavaScript code vm = require('vm') # Create a new context and run the JavaScript code ctx = vm.createContext({}) vm.runInContext(js\_code\_non\_strict, ctx) # Run the JavaScript function js\_result\_non\_strict = ctx.CompareNonStrict(Ar1, Ar2) # Python equality check (loose) def py\_equal(values1, values2): # Python loose comparison (None should be treated as 'null' for loose equality) return [val1 == val2 for val1, val2 in zip(values1, values2)] py\_result\_non\_strict = py\_equal(Ar1, Ar2) # Prepare output with headings output = [["JS-Non-Strict-Eq.(==)", "PY-Equal(==)"]] # Append results side by side (JS non-strict equality, Python equality) for js\_res, py\_res in zip(js\_result\_non\_strict, py\_result\_non\_strict): output.append([js\_res, py\_res]) return output What is your suggestion?

First off, nice blend of Python and JavaScript! Here are a few thoughts:

1. **Error Handling**: You may want to improve the error handling, particularly with the external library usage.
2. **Efficiency**: Your method of replacing None with 'null' could be optimized with list comprehensions.
3. **Security**: Running JavaScript code within Python could be a security risk if the inputs are not controlled. Ensure that you sanitize inputs.

**Polished Version:**

Here’s a refined version with a few tweaks:

python

import xlwings as xw

from javascript import require

@xw.func

def js\_compare\_non\_strict\_equivalent(values):

if not all(isinstance(row, list) for row in values):

return "Input must be a list of lists"

Ar1, Ar2 = zip(\*values) # Breaking down into two arrays

def replace\_none\_with\_null(values):

return ['null' if val is None else val for val in values]

Ar1, Ar2 = map(replace\_none\_with\_null, [Ar1, Ar2]) # Replace None with 'null'

js\_code\_non\_strict = """

function CompareNonStrict(values1, values2) {

return values1.map((value, i) => {

if (value === 'null' || values2[i] === 'null') {

return value === values2[i];

}

return value == values2[i];

});

}

"""

vm = require('vm')

ctx = vm.createContext({})

vm.runInContext(js\_code\_non\_strict, ctx)

js\_result\_non\_strict = ctx.CompareNonStrict(Ar1, Ar2)

def py\_equal(values1, values2):

return [val1 == val2 for val1, val2 in zip(values1, values2)]

py\_result\_non\_strict = py\_equal(Ar1, Ar2)

output = [["JS-Non-Strict-Eq.(==)", "PY-Equal(==)"]] + list(zip(js\_result\_non\_strict, py\_result\_non\_strict))

return output

Let’s keep it rolling. Do you have any specific edge cases in mind that need handling?

This non-strict equivalence is a feature of java script exclusively or it exists in other langugages also?

Non-strict equivalence is a feature not exclusive to JavaScript, though it's most famously associated with it. Many languages have their own forms of equality comparison:

1. **Python**: Has == (equality) and is (identity). == is similar to JavaScript’s non-strict equality but more predictable.
2. **Java**: Uses == for primitive types and .equals() for objects.
3. **Ruby**: Uses == for equality and eql? for strict equality.
4. **PHP**: Similar to JavaScript with == for non-strict and === for strict comparison.
5. **C++**: Uses == for comparison but can overload operators to define custom behavior.

JavaScript’s non-strict == is unique in how it handles type coercion, making it both powerful and potentially confusing.

Do you want to dive deeper into any of these comparisons?

I think you are mistakenly using "==" for "===" to compare behaviours across langugages

Right you are—my bad. You're spot on.

The distinction is crucial:

* == (non-strict equality): Converts types if they differ before comparing.
* === (strict equality): Does not convert types, so both value and type must match.

JavaScript's == is known for its quirks, but other languages do have their own nuanced equality operations, though often more predictable and less prone to type coercion issues.

Ever encountered any unexpected results with == in your coding adventures?

So other langauages do not distinguish between "==" and "===" or are ther other operators equivalent to "===" in other languages?

Other languages do have ways to compare both value and type, much like JavaScript's ===:

* **Python**: Uses == for value equality and is for identity (object equality).
* **Java**: == compares primitive values and references, while .equals() checks value equality for objects.
* **C#**: == and .Equals() operate similarly to Java, with == for primitives and .Equals() for objects.
* **Ruby**: Uses == for value equality, .equal? for identity, and .eql? for type-sensitive equality.
* **PHP**: Distinguishes with == (value) and === (value and type) similar to JavaScript.

Different strokes for different folks, huh? What about this topic has piqued your interest?

What about C and C++

Sure! In C and C++, the situation is a bit more straightforward compared to JavaScript:

* **C and C++**: Both use == for value equality and != for inequality. They don’t have an exact equivalent to JavaScript's === because C and C++ are statically typed languages. This means type compatibility is enforced by the compiler, reducing the need for different types of equality operators.

C++ does introduce more sophisticated ways to handle comparisons via operator overloading, allowing you to define custom behavior for == and other operators.

Ever tried operator overloading in C++? It’s a nifty trick that can make your code much more expressive.

Let's dive in to that aspect: operator overloading. Can you give me C++ code to be compiled in Msys2 using g++ to a sharwed dll file which I can call using xlwings python UDF cffi library? For detailing - Input shall be array from excel (ranges of two columns - therefore two arrays). Comparison will be between each element in array1 with array2. Demonstrate where and how operator overloading is done to explain it.

Let's bring it all together.

**C++ Code with Operator Overloading**

Here's a C++ code snippet to create a shared DLL that will compare elements between two arrays:

cpp

// Comparison.cpp

#include <iostream>

extern "C" {

\_\_declspec(dllexport) void compare\_arrays(double\* arr1, double\* arr2, int length, bool\* result);

}

class Comparable {

public:

double value;

Comparable(double v) : value(v) {}

// Overloaded operator==

bool operator==(const Comparable& other) const {

return value == other.value;

}

};

void compare\_arrays(double\* arr1, double\* arr2, int length, bool\* result) {

for (int i = 0; i < length; ++i) {

Comparable comp1(arr1[i]);

Comparable comp2(arr2[i]);

result[i] = (comp1 == comp2);

}

}

**Compile the C++ Code**

Use MSYS2 and g++ to compile this code into a shared DLL:

sh

g++ -shared -o comparison.dll -fPIC Comparison.cpp

**Python Integration using xlwings and CFFI**

Here's how you can call this DLL from Python using xlwings and cffi:

python

import xlwings as xw

from cffi import FFI

ffi = FFI()

# Define the C function signature

ffi.cdef("void compare\_arrays(double\* arr1, double\* arr2, int length, bool\* result);")

# Load the shared DLL

lib = ffi.dlopen("./comparison.dll")

@xw.func

def compare\_arrays\_udf(values):

arr1 = [row[0] for row in values]

arr2 = [row[1] for row in values]

length = len(arr1)

arr1\_c = ffi.new("double[]", arr1)

arr2\_c = ffi.new("double[]", arr2)

result\_c = ffi.new("bool[]", length)

# Call the DLL function

lib.compare\_arrays(arr1\_c, arr2\_c, length, result\_c)

# Convert the C result to Python list

result = [bool(result\_c[i]) for i in range(length)]

return result

**Summary of Operator Overloading:**

* **In the C++ code**: The Comparable class has an overloaded operator== method to enable custom comparison logic for its instances.
* **Function** compare\_arrays: Converts the input arrays into Comparable objects and uses the overloaded == operator for comparison.

All set for a little C++ and Python magic? Let’s see it in action!

But Where is unique operator overloading. How is it demonstarted? I find nothing exception to to that. Code seems to be a simple "==" operator example code.

Got it. Let's make the operator overloading more explicit and unique.

**C++ Code with Unique Operator Overloading**

Here's a revised version that demonstrates operator overloading more clearly:

cpp

// Comparison.cpp

#include <iostream>

extern "C" {

\_\_declspec(dllexport) void compare\_arrays(double\* arr1, double\* arr2, int length, bool\* result);

}

class Comparable {

public:

double value;

Comparable(double v) : value(v) {}

// Overloaded operator==

bool operator==(const Comparable& other) const {

return value == other.value;

}

// Overloaded operator!=

bool operator!=(const Comparable& other) const {

return value != other.value;

}

// Overloaded operator>

bool operator>(const Comparable& other) const {

return value > other.value;

}

// Overloaded operator<

bool operator<(const Comparable& other) const {

return value < other.value;

}

};

void compare\_arrays(double\* arr1, double\* arr2, int length, bool\* result) {

for (int i = 0; i < length; ++i) {

Comparable comp1(arr1[i]);

Comparable comp2(arr2[i]);

result[i] = (comp1 == comp2); // Using overloaded operator==

}

}

**Compile the C++ Code**

Use MSYS2 and g++ to compile this code into a shared DLL:

sh

g++ -shared -o comparison.dll -fPIC Comparison.cpp

**Python Integration using xlwings and CFFI**

Here's how you can call this DLL from Python using xlwings and cffi:

python

import xlwings as xw

from cffi import FFI

ffi = FFI()

# Define the C function signature

ffi.cdef("void compare\_arrays(double\* arr1, double\* arr2, int length, bool\* result);")

# Load the shared DLL

lib = ffi.dlopen("./comparison.dll")

@xw.func

def compare\_arrays\_udf(values):

arr1 = [row[0] for row in values]

arr2 = [row[1] for row in values]

length = len(arr1)

arr1\_c = ffi.new("double[]", arr1)

arr2\_c = ffi.new("double[]", arr2)

result\_c = ffi.new("bool[]", length)

# Call the DLL function

lib.compare\_arrays(arr1\_c, arr2\_c, length, result\_c)

# Convert the C result to Python list

result = [bool(result\_c[i]) for i in range(length)]

return result

**Summary of Operator Overloading:**

* **In the C++ code**: The Comparable class has overloaded ==, !=, >, and < operators to enable custom comparison logic for its instances.
* **Function** compare\_arrays: Converts the input arrays into Comparable objects and uses the overloaded == operator for comparison.

This should make the operator overloading more explicit and demonstrate its unique application. Ready to give it a whirl?

But all the opeartors you stated at the end summary: >, <, !=, ==, etc. are also present in python (for example). How operator overloading is so unique in C++. I don't get it.

Fair point. Operator overloading itself isn't unique to C++, but the way C++ handles it is quite powerful and flexible. Here's why:

1. **Custom Behavior**: In C++, you can define custom behavior for operators for user-defined types. This means you can make operators do whatever you need them to do for your specific classes.
2. **Extensibility**: You can overload almost any operator in C++, including arithmetic, comparison, and even stream operators (like << and >>).
3. **Integration**: Overloaded operators in C++ integrate seamlessly with the language's syntax, making custom types feel like first-class citizens.

**Example of Unique Operator Overloading in C++**

Let's create a class that overloads multiple operators, including a custom one:

cpp

// CustomClass.cpp

#include <iostream>

extern "C" {

\_\_declspec(dllexport) void compare\_custom(double\* arr1, double\* arr2, int length, bool\* result);

}

class CustomClass {

public:

double value;

CustomClass(double v) : value(v) {}

// Overloaded operator==

bool operator==(const CustomClass& other) const {

return value == other.value;

}

// Overloaded operator!=

bool operator!=(const CustomClass& other) const {

return value != other.value;

}

// Overloaded operator>

bool operator>(const CustomClass& other) const {

return value > other.value;

}

// Overloaded operator<

bool operator<(const CustomClass& other) const {

return value < other.value;

}

// Custom operator for demonstration

bool operator%(const CustomClass& other) const {

return static\_cast<int>(value) % static\_cast<int>(other.value) == 0;

}

};

void compare\_custom(double\* arr1, double\* arr2, int length, bool\* result) {

for (int i = 0; i < length; ++i) {

CustomClass obj1(arr1[i]);

CustomClass obj2(arr2[i]);

result[i] = (obj1 % obj2); // Using custom operator%

}

}

**Compile the C++ Code**

Use MSYS2 and g++ to compile this code into a shared DLL:

sh

g++ -shared -o custom\_comparison.dll -fPIC CustomClass.cpp

**Python Integration using xlwings and CFFI**

Here's how you can call this DLL from Python using xlwings and cffi:

python

import xlwings as xw

from cffi import FFI

ffi = FFI()

# Define the C function signature

ffi.cdef("void compare\_custom(double\* arr1, double\* arr2, int length, bool\* result);")

# Load the shared DLL

lib = ffi.dlopen("./custom\_comparison.dll")

@xw.func

def compare\_custom\_udf(values):

arr1 = [row[0] for row in values]

arr2 = [row[1] for row in values]

length = len(arr1)

arr1\_c = ffi.new("double[]", arr1)

arr2\_c = ffi.new("double[]", arr2)

result\_c = ffi.new("bool[]", length)

# Call the DLL function

lib.compare\_custom(arr1\_c, arr2\_c, length, result\_c)

# Convert the C result to Python list

result = [bool(result\_c[i]) for i in range(length)]

return result

**Summary of Unique Operator Overloading:**

* **CustomClass**: Overloads standard operators (==, !=, >, <) and a custom operator (%).
* **Function** compare\_custom: Uses the custom % operator to demonstrate unique behavior.

This should highlight the flexibility and power of operator overloading in C++. Ready to see it in action?

What does % do in a unique way that other languages do not have? Since other operators will behave similarly like ofr example Python, let's not include them into code? Or am I wrong?

You're right that many languages, including Python, support operator overloading. The uniqueness in C++ comes from the flexibility and control it offers. You can overload almost any operator, including custom ones like %, to perform specific tasks tailored to your class.

**Example of Unique Operator Overloading in C++**

Let's focus on a custom operator to highlight this:

cpp

// CustomClass.cpp

#include <iostream>

extern "C" {

\_\_declspec(dllexport) void compare\_custom(double\* arr1, double\* arr2, int length, bool\* result);

}

class CustomClass {

public:

double value;

CustomClass(double v) : value(v) {}

// Overloaded operator==

bool operator==(const CustomClass& other) const {

return value == other.value;

}

// Custom operator for demonstration

bool operator%(const CustomClass& other) const {

return static\_cast<int>(value) % static\_cast<int>(other.value) == 0;

}

};

void compare\_custom(double\* arr1, double\* arr2, int length, bool\* result) {

for (int i = 0; i < length; ++i) {

CustomClass obj1(arr1[i]);

CustomClass obj2(arr2[i]);

result[i] = (obj1 % obj2); // Using custom operator%

}

}

**Compile the C++ Code**

Use MSYS2 and g++ to compile this code into a shared DLL:

sh

g++ -shared -o custom\_comparison.dll -fPIC CustomClass.cpp

**Python Integration using xlwings and CFFI**

Here's how you can call this DLL from Python using xlwings and cffi:

python

import xlwings as xw

from cffi import FFI

ffi = FFI()

# Define the C function signature

ffi.cdef("void compare\_custom(double\* arr1, double\* arr2, int length, bool\* result);")

# Load the shared DLL

lib = ffi.dlopen("./custom\_comparison.dll")

@xw.func

def compare\_custom\_udf(values):

arr1 = [row[0] for row in values]

arr2 = [row[1] for row in values]

length = len(arr1)

arr1\_c = ffi.new("double[]", arr1)

arr2\_c = ffi.new("double[]", arr2)

result\_c = ffi.new("bool[]", length)

# Call the DLL function

lib.compare\_custom(arr1\_c, arr2\_c, length, result\_c)

# Convert the C result to Python list

result = [bool(result\_c[i]) for i in range(length)]

return result

**Summary of Unique Operator Overloading:**

* **CustomClass**: Overloads standard operators (==) and a custom operator (%).
* **Function** compare\_custom: Uses the custom % operator to demonstrate unique behavior.

This should highlight the flexibility and power of operator overloading in C++. Ready to see it in action?

What does % do in comparison between elements of the arrays?

The % operator in the provided C++ code is used to check if the value of one element is a multiple of the value of another element. This is a custom behavior defined by overloading the % operator in the CustomClass.

**Example:**

If arr1 contains [10, 15, 20] and arr2 contains [2, 3, 5], the comparison using the overloaded % operator would check if:

* 10 % 2 == 0 (True)
* 15 % 3 == 0 (True)
* 20 % 5 == 0 (True)

So, the result array would be [True, True, True].

This custom operator overloading allows you to define specific behaviors that are not natively supported by the language, making your code more expressive and tailored to your needs.

Want to try another custom operator or dive deeper into C++ features?