# Installation

It is very simple to start working with WDC. All you need to do is:

- Install Python  How to Install python.
- Install Requests  How to install Requests.
- Make sure both Python and Requests are in the same folder, as the file you are working with.

**Required dependencies**

| Package | Version |
|---|---|
| Python | 3.6 |
| Requests | 2.3 |

**Optional Dependencies**

If you wish to work in ‘jupyter notebooks’, you also need to install ‘IPython' for displaying images.
How to install IPython

**Warnings**

If you wish to suppress warnings, you can also import warnings.

```
import warnings
warnings.filterwarnings("ignore")
```

# Package Overview

WDC is a library, designed to facilitate easy interaction with coverage data sets often used in geospatial data (essentially any type of scientific data that varies over space and/or time, like satellite imagery, digital elevation models, and other geospatial data) processing and analysis. In particular, it provides an abstraction between a user and web coverage processing service

queries(WCPS) in order to let users get easy access to geo-spatial coverage data sets and to analyze and manipulate the retrieved data.

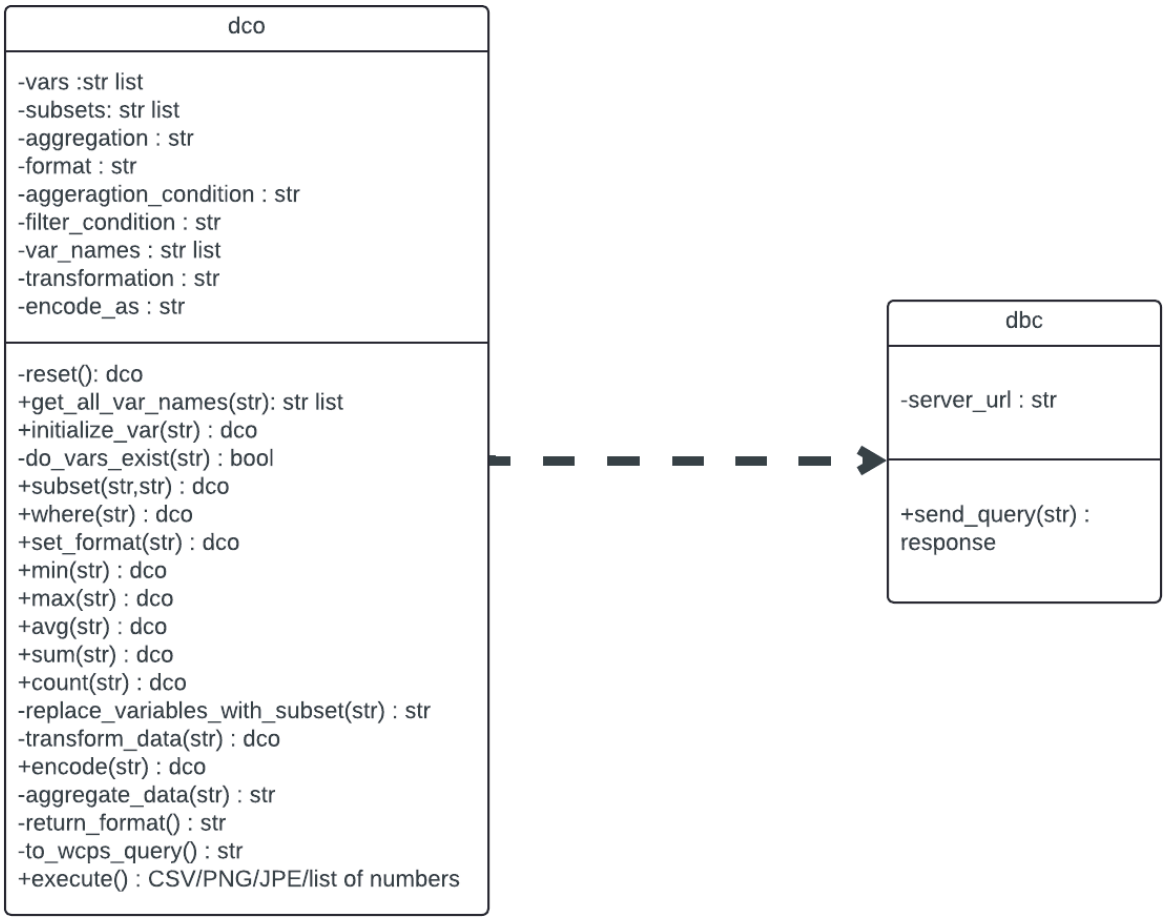
**What is coverage ?**
Coverage is a worldwide accepted concept for representing raster data and datacubes. Generally, they represent regular and irregular grids, point clouds and general meshes, raster data and datacubes.
Learn more about coverages

**What is WCPS?**
WCPS is a high-level query language that allows users to retrieve and process geospatial-data without having to handle the full datasets locally. This can include operations like trimming, slicing, and applying mathematical and logical expressions to the data to generate new results.
Learn WCPS basics
Learn WCPS advanced

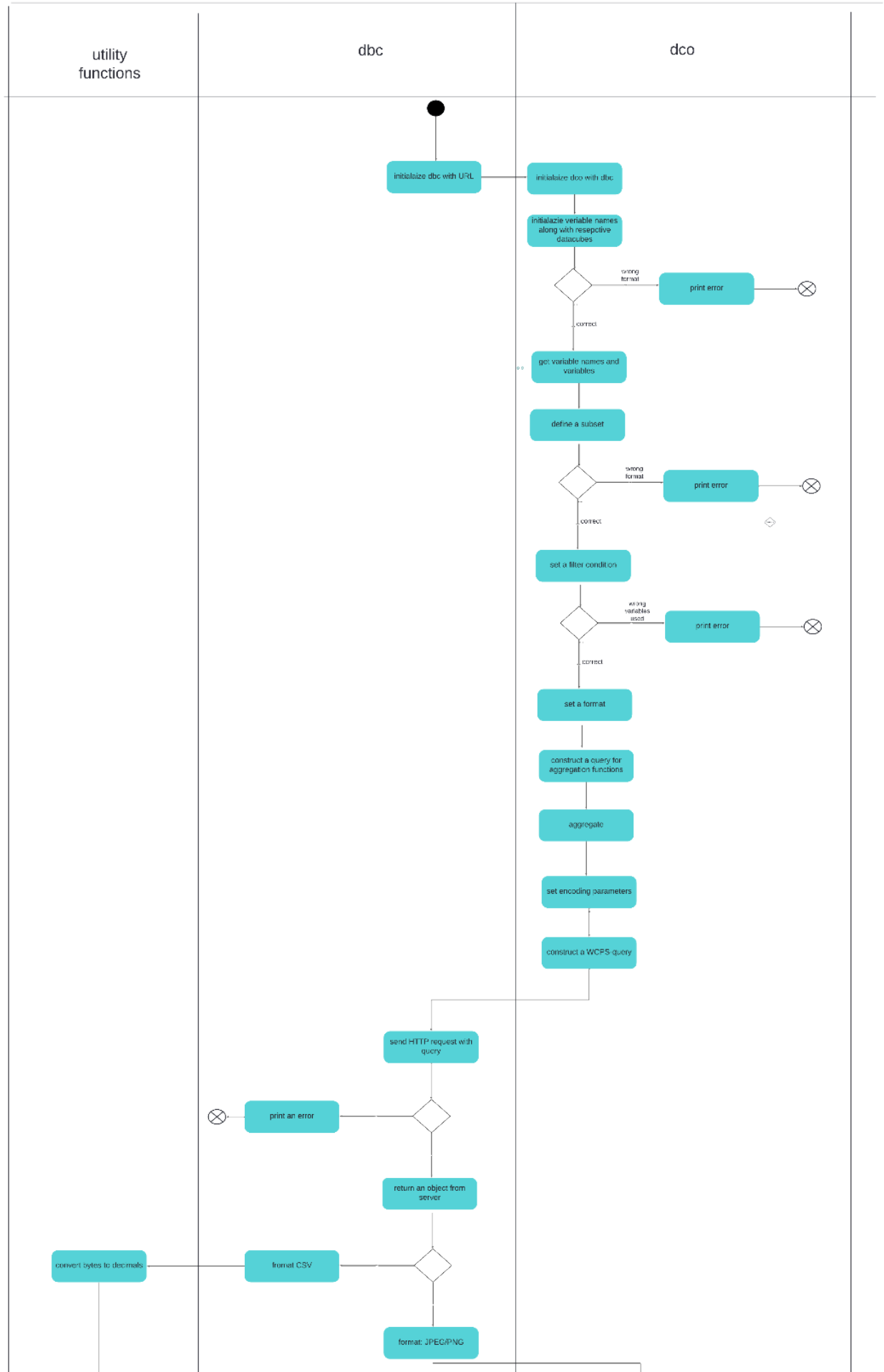
**Classes in WDC**

| | |
|---|---|
| dbc (Database Connection object) | provides a method for sending WCPS queries to the specified server endpoints and handling the responses |
| dco (Datacube object) | designed to construct and manage queries to datacube |

## Class Diagram




Activity Diagram


utility functions
dbc
dco
initialaize dbc with URL
initialaize dco with dbc
initialazie veriable names along with resepctive datacubes
wrong format
print error
correct
get variable names and variables
define a subset
wrong format
print error
correct
set a filter condition
wrong variables used
print error
correct
set a format
construct a query for aggregation functions
aggregate
set encoding parameters
construct a WCPS query
send HTTP request with query
print an error
return an object from server
convert bytes to decimals
fromat CSV
format: JPEG/PNG

# About DBC

This class provides a method for sending WCPS queries to the specified server endpoints and handling the responses. It also includes error handling protocol and provides useful feedback in case something goes wrong with the network communication or server response.

**ATTRIBUTES:**
Server_url (string)

**METHODS:**
send_query

**RETURNS:**
Object from the server

First, it initializes a new ‘dbc’ instance with ‘url’, which will serve as an endpoint URL of the WCPS server.

Using the predefined  library function ‘requests.post’, ‘dbc’ will send a HTTP request.

‘dbc’ returns The response object from the server, which includes the HTTP status code, response headers, and the raw response content.

# About DCO

'dco' is designed to construct and manage queries to datacube via a WCPS.

The class handles various aspects of data querying, including initialization of query parameters, manipulation of subsets, specification of output formats, and application of aggregation function.

‘dco’ instances must be initialized with ‘dbc’, in other words, ‘dco’ requires ‘dbc’

## DCO Attributes:

| **Name** | **Data type** |
|---|---|
| vars | list |
| subsets | list |
| aggregation | string |
| format | string |
| aggregation_condition | string |
| filter_condition | string |
| var_names | list |
| transformation | string |
| encode_as | string |

## DCO Methods:

| Name | Parameter | Return Type |
|---|---|---|
| reset | | dco |
| get_all_var_names | str | list |
| initialize_var | str | dco |
| do_vars_exist | str | bool |
| subset | str,str | dco |
| where | str | dco |
| set_format | str | dco |
| min | str | dco |
| max | str | dco |
| avg | str | dco |
| sum | str | dco |
| count | str | dco |
| replace_variables_with_subset | str | str |
| ransform_data | str | dco |
| encode | str | dco |
| aggregate_data | str | str |
| return_format | | str |
| to_wcps_query | | str |
| execute | | CSV/PNG/JPEG file/ list of numbers |

## Reset()

This method resets all the attributes of the ‘dco’ instance to their default values, except for the ‘dbc’ connection, which remains unchanged.

## get_all_var_names()

Parameters:

string (str): A string potentially containing multiple variables each prefixed by '$'.

Returns:

list: A list of extracted variable names. Returns an empty list if no variables are found.

This method takes string as an input and extracts variables prefixed with ’$’ and stores them in var_names.

The loop iterates from the beginning of the string to its end.
It first finds the first ‘$’ in the input string and stores it as start_index.

Then, it finds a delimiter after the first occurrence of ‘$’ and stores it as end_index.

The variable name is a substring between start and end index(including these indexes).

The loop will now continue to iterate from the last_index, until it finds all the variables.

get_all_var_names() will return the list of all these variables after execution.

## initialize_var()

Parameters:

s (str): A string that combines the variable name and its associated datacube, formatted as '$variable_name in (coverage_name)'.

Returns:

self: Returns the instance itself if the variable is successfully added.

This method is crucial for ensuring that variables are properly initialized and ready for use in forming WCPS queries.

This method will raise an error, if the passed parameter is not a string.
It will also raise an error, if the string is not in a specific format.
The expected format is '$variable_name in (coverage_name)'.

After it receives a properly formatted string, it will be appended to vars.
The code will also extract the variable name with get_all_var_names() method and append it to var_names list.

## do_vars_exist()

Parameters:

string (str): The string from which variable names are extracted and checked. Variables in the string should be prefixed by '$'.

Returns:

bool: True if all extracted variables exist in the instance's variable list.

Checks whether all variable names extracted from the input string exist in the predefined list of variable names of the current instance.

By first replacing newline (\n), carriage return (\r), and tab (\t) characters in the input string with spaces the method ensures that variable names are not disrupted by formatting characters which might interfere with name extraction.

Then, with get_all_var_names(), the method extracts all the variable names and checks it against the predefined list of variable names.

The method will return either TRUE or raise an error.

## subset()

Parameters:

subset (str): The subset condition to apply, typically specifying a range or filter for the data retrieval.

var_name (str): The name of the variable to which the subset will be applied.

Returns:

self: Returns the instance itself after updating the subset for the specified variable, allowing for method chaining.

Adds a subset specification for a specific variable in the datacube object.

The method first makes sure, by raising errors, that both input parameters are strings and verifies that input var_name indeed exists in the predefined var_name list.

Then, it finds the index of the var_name in the var_name list and updates ‘subset’ with the input subset string at the same index in its respective list.

subset() will return an updated ‘dco’.

## where()

Parameters:

filter_condition (str): A string representing the condition to be applied to filter the data. This condition should be formatted according to the expected WCPS syntax.

Returns:

self: Returns the instance itself, allowing for method chaining and further configuration.

This method allows specifying conditions that filter the data according to certain criteria, similar to a SQL WHERE clause.

The method makes sure, by raising an error otherwise, that the input parameter is a string, and with do_vars_exist() verifies that the variable from the input string has been pre defined.

Lastly it updates dco's filter_condition with the input and returns the new dco.

## set_format()

Parameters:

output_format (str): The format to set for output data.

Returns:

self: Returns the instance itself after setting the output format, allowing for method chaining.

The method verifies that the parameter passed is indeed a string, and either 'PNG', 'CSV', 'JPEG', otherwise it will raise an error.

The method will set the format attribute of the 'dco' with the passed parameter and return the updated 'dco'.

## min()

Parameters:

condition (str, optional)

Returns:

self: Returns the instance itself, allowing for method chaining.

Configures the datacube to compute minimum value of the specified data.
An optional condition can specify a criteria.

If a condition is passed, the method makes sure that it's indeed a string and that variable in the string has been predefined.

Then, it updates the aggregate_condition with the condition passed to the method. And sets aggregation as 'MIN'.

## max()

Parameters:

condition (str, optional)

Returns:

self: Returns the instance itself, allowing for method chaining.

Configures the datacube to compute maximum value of the specified data.
An optional condition can specify a criteria.

If a condition is passed, the method makes sure that it's indeed a string and that variable in the string has been predefined.

Then, it updates the aggregate_condition with the condition passed to the method. And sets aggregation as 'MAX'.

## Avg()

Parameters:

condition (str, optional)

Returns:

self: Returns the instance itself, allowing for method chaining.

Configures the datacube to compute the average value of the specified data.
An optional condition can specify a criteria.

If a condition is passed, the method makes sure that it's indeed a string and that variable in the string has been predefined.

Then, it updates the aggregate_condition with the condition passed to the method. And sets aggregation as ‘AVG’.

## sum()

Parameters:

condition (str, optional)

Returns:

self: Returns the instance itself, allowing for method chaining.

Configures the datacube to compute the sum value across specified data.
An optional condition can specify a criteria.

If a condition is passed, the method makes sure that it’s indeed a string and that variable in the string has been predefined.

Then, it updates the aggregate_condition with the condition passed to the method. And sets aggregation as ‘SUM’.

## count()

Parameters:

condition (str, optional)

Returns:

self: Returns the instance itself, allowing for method chaining.

Configures the datacube to compute the number of data points that meet specified condition.

An optional condition can specify a criteria.

If a condition is passed, the method makes sure that it’s indeed a string and that variable in the string has been predefined.

Then, it updates the aggregate_condition with the condition passed to the method. And sets aggregation as ‘COUNT’.

## replace_variables_with_subsets()

Parameters:

str_to_transform (str, optional): A string containing variable names that need to be replaced with their subsets.

Returns:

str: A new string with variables replaced by their subsets, or a concatenated string of all variables and subsets.

Replaces variables in a given string with their corresponding subset, if defined. If no string is provided, it constructs a string representation of all variables and their subsets.

The method first checks if a string was passed to the function.
If it was, it stores the input as ‘expression’

```
for var, subset in zip(self.var_names, self.Subsets):
```

This loop goes through each variable and its subset in parallel.
if a subset is defined , the method replaces occurrences of the variable name in the expression with the variable name followed by the subset in square brackets:

```
expression = expression.replace(var, f'{var}[{subset}]')
```

If there is no input string, the expression is initialized as an empty string.
For no input case as well, the method iterates through the same loop and appends formatted strings with variables and its respective subset, if applicable.
If not, only variable names will be added to expression.

```
        expression += f'''{var}[{subset}] '''
    else
        expression += f'''{var}'''
```

## transform_data()

Parameters:

operation (str): A string representing the transformation operation to be applied.

Returns:

self: Returns the instance itself, allowing for method chaining.

The method verifies that input is a string, and does not include undefined variables.

Then, it sets the transformation value as the input.

Returns updated ‘dco’

## encode()

Parameters:

operation (str): A string representing the encoding function, such as "encode".

Returns:

self: Returns the instance itself, allowing for method chaining.

First it verifies that the parameter is a string and does not include undefined variables.

Then it sets the encode_as with the input string.

## aggregate_data()

Parameters:

Not Applicable

Returns:

str: A string representing the aggregation part of the WCPS query.

Constructs a part of the WCPS query specifically for performing aggregate functions.

```
helper_query =
self.replace_variables_with_subsets(self.aggregation_condition)
```

This line calls another method of the datacube class, replace_variables_with_subsets().

The result, stored in helper_query, is a string ready to be used in a WCPS query.

Then, the method determines which aggregation function to use based on the self.aggregation value (which can be only ‘MIN’, ‘AVG’, ‘MAX’, ‘SUM’, or ‘COUNT’)

Depending on the value of self.aggregation, the method constructs a WCPS query segment using an f-string to embed helper_query within the appropriate aggregation function cal

```
f'''min({helper_query})'''
f'''max({helper_query})'''
.
.
.
```

## return_format()

Parameters:

Not Applicable

Returns:

str: A string indicating the desired output format for the WCPS query.

Determines the format for the output based on the configured settings of the datacube.

## to_wcps_query()

### Parameters:

Not Applicable

### Returns:

str: A string that represents the complete WCPS query based on the current state of the ‘dco’ instance

This method allows constructing a dynamically generated WCPS query based on various configurable settings in the datacube object, such as variable names, filter conditions, aggregation settings, transformations, and output formats.

First, the query string is initiated with the for keyword, as all WCPS queries begin with ‘for’.

The method iterates over all variables stored in the instance's vars list. Each variable is appended to the query, each on a new line:

```
for var in
query += (var + '\n')
```

If there is a filter condition, the method will append it to the query with proper formatting,

```
query += f'''where {self.filter_condition}\n'''
```

Then it adds the ‘return’ keyword on the new line.

If an aggregation method is specified the corresponding query part is added by calling aggregate_data() in the query string.
If an aggregation was specified, the method will immediately return the query.

If aggregation has not been specified, the method goes on to check if there's a specific encoding or transformation specified and constructs that part of the query using replace_variables_with_subsets().

```
helper_query = self.replace_variables_with_subsets(self.encode_as)
helper_query = self.replace_variables_with_subsets(self.transformation)
```

If neither encoding or transformation has been specified, replace_variables_with_subsets() is called without passing, which  will append formatted strings with variables and its respective subset, if applicable. If not, only variable names will be added to expression.

Finally,  if no specific output format is set, helper_query is directly added to the query string. If it has been defined, it will also be appended to the query string in a proper format that WCPS query expects.

### Execute()

Parameters:

Not Applicable

Returns:

Varies: The processed data as per the requested format

This method calls the to_wcps_query() to construct a query string and then using ‘dbc’, it sends a request to the server.

Then, The method uses a series of conditional checks to determine how to process the received response based on the format attribute of the datacube:

If the format is JPEG' or  ‘PNGthe method resets the ‘dco’ with reset() and returns the response content.

If the format is ‘CSV’ or unidentified, the byte_to_list() function is used to parse the binary string from a CSV file into a list before resetting and then it returns the response content.

## byte_to_list()

Parameters:

byte_str (bytes): The byte string to be converted.

Returns:

list of float: A list of floats derived from the byte string.

This utility function decodes the byte string to a regular string using ‘UTF-8’ encoding.

```
str_list = decoded_str.split(',')
```

After decoding, the function splits the string into a list of substrings using the comma (,) as the delimiter. This is based on the assumption that the numeric values are comma-separated in the input byte string.

```
num_list = [float(num) for num in str_list]
```

This line uses a list comprehension to iterate over each substring in str_list and convert each substring to a float. This step transforms the list of string representations of numbers into a list of actual float values.

Finally, the function returns this list of floats.

# Comparison

## The benefits of using the WDC library instead of writing WCPS queries

1. **The order of methods almost doesn't matter.**

Initialize_var() method must come first after you create a ‘dco’ instance, but the order of the following methods can be determined by a user.

When we write a WCPS query, we must follow a specific structure and sometimes it might be confusing, especially if you are not familiar with the WCPS query's structure. Using WDC is simpler and you don't have to memorize much.

## 2. **No need to specify a subset all the time.**

Let's take a look at the following WCPS query:

```
for $c in ( AvgLandTemp )
return
encode(
    switch
            case $c[ansi("2014-07"), Lat(35:75), Long(-20:40)] =
99999
                return {red: 255; green: 255; blue: 255}
            case 18 > $c[ansi("2014-07"), Lat(35:75), Long(-20:40)]
                return {red: 0; green: 0; blue: 255}
            case 23 > $c[ansi("2014-07"), Lat(35:75), Long(-20:40)]
                return {red: 255; green: 255; blue: 0}
            case 30 > $c[ansi("2014-07"), Lat(35:75), Long(-20:40)]
                return {red: 255; green: 140; blue: 0}
            default return {red: 255; green: 0; blue: 0}
, "image/png")
```

We see that here the subset is specified 4 times. If you use WDC, a similar solution will look like this:

```
my_dco.initialize_var('$c in ( AvgLandTemp )')
my_dco.encode('''
    switch
            case $c = 99999
                return {red: 255; green: 255; blue: 255}
            case 18 > $c
                return {red: 0; green: 0; blue: 255}
            case 23 > $c
                return {red: 255; green: 255; blue: 0}
            case 30 > $c
                return {red: 255; green: 140; blue: 0}
            default return {red: 255; green: 0; blue: 0}
''')
```

```
my_dco.subset(var_name = '$c', subset = 'ansi("2014-07"), Lat(35:75),
Long(-20:40)')
my_dco.set_format('PNG')
Image(my_dco.execute())
```

Here the subset is specified only once and the program replaces '$c' with $c[ansi("2014-07"), Lat(35:75), Long(-20:40)]'. Which can save some time.

### 3. **Byte strings are converted to the list of numbers automatically.**

In general, when you try to use a simple requests.post(server_url, data = {'query': wcps_query}, verify = False) and then access requests.content, you get a byte string, but not the list of numbers. By using WDC, you get a list of numbers automatically in case the output is not expected to be an image.

# Testing

We used Unit tests using the Pytest framework to test the ‘wdc’ library.

We have finished testing for around 70% of the program. In the “test_wdc.py’ file, we have documented a total of 57 tests that all pass.

In the file you can find tests for both error handling and functionality of the program.

Error handling scenarios tested: type errors, value errors, general exeptions.

Functional areas tested: Initialization and Configuration, Data Querying and Manipulation, Aggregation and Filtering, End-to-End Query Composition

**How to use Pytest ?** Pytest

First, make sure Pytest is installed in your python environment. You can do this on the terminal by executing ‘pip install pytest’.

Pytest finds tests by naming convention; this means that testing files must be prefixed by ‘test_’, and all the functions inside the file must also begin with ‘test_’.

To run a test inside of a directory,  go to that directory, simply type ‘pytest’ and it will run all the unit tests in that directory.

Do not forget to import classes from ‘wdc’ in the test file.

# Learn More

For more information and examples, please look at the training guide for the WDC library “guide_to_the_library”.

This guide will walk you through how to set up and start using WDC for working with coverage data in your projects

# Possible Improvements

- Add a Switch Method:
  Make a method that handles different conditions and decisions more efficiently.
- Add Coverage Constructor:
  Develop a method that makes it easier to set up and start using datasets that cover specific areas or topics.
- Develop a Clipping Function:
  Create a function that cuts data to fit within certain limits.
- Add a method for displaying diagrams.