Software Engineering Project
Spring Semester 2024

Spatio-Temporal Datacubes: the OGC/ISO Coverage Standards

Instructor:	Prof. Peter Baumann
Teaching Assistant:	Eremina Elizaveta
	Sirotkina Veronika
	Getahun Raey Addisu

Project members:

Sprint1	Diana Maria Harambas
	Mark Kanafeev
Sprint2	Zarina Abulkassova
	Daria Soloveva
Sprint3	Sulehri Sabeeh Ur Rehman
	Wang Yat Sin

Version：1.0

Date : 15 May 2024

Content

I) Introduction

This project relies heavily on data, specifically on coverage data models. Accessing the datacubes is done via OGC WCS and WCPS standards. The coverages that we are going to use can be found here: https://standards.rasdaman.com/demo_wcs.html . Usually in this case, we have space/time datacubes, meaning that we have information about both dates and spatial coordinates, thus the representations could contain more axes.

Certain WCS (Web Coverage Service) and WCPS (Web Coverage Processing Service) requests require trimming the dataset, i.e. looking at subsets of the spatial information, and specific dates. In the case of slicing, one is interested in a lower-dimensional section of the datacube. We will be covering both of them in the project.

Data is retrieved from the Rasdaman server. Via http requests to the endpoint https://ows.rasdaman.org/rasdaman/ows , one can access a coverage description and check the axes that appear by pressing *Describe Coverage*. Accessing the *Get Coverage* page, one can choose their preferred method (slicing/trimming of the data) and enter allowed values in order to obtain a proper request URL. Our goal was to be able to use the data obtained in such manner and even perform queries on it.

For the purpose of storing and classifying the data, we will make use of the Datacube class and also of the Coverage class and its subclasses, which will help us navigate through different types of examples. Using classes like QueryTree we are also able to perform several actions on the given data and submit queries to the server.

II) Installation

In order to start working with WDC following steps should be done:

- Install Python 3.6 How to Install python.
- Install Requests 2.3 How to install Requests.
- Install IPython latest How to install IPython.
- Install other libraries that covered in the requirements.txt (./requirements.txt)

Required dependencies

Package	**Version**
Python	3.6
Requests	2.3
owslib	Latest
numpy	Latest
scipy	Latest
matplotlib	Latest
ipykernel	Latest
Pillow	Latest

III) Package Overview

The WDC Extension Library is a Python package designed to extend the functionality of the WDC (Web Coverage Service Datacube) library. It aims to facilitate seamless interaction with a WCPS (Web Coverage Processing Service) server, enabling users to efficiently query, retrieve, and manipulate geospatial coverage data.

Key Concepts:

Coverage Data: Our library operates on the principles of coverage data, which represent raster data and datacubes, including regular and irregular grids, point clouds, meshes, raster data, and datacubes. These data types are fundamental in geospatial analysis and processing.

WCPS Interface: Our library provides an interface to interact with a WCPS server. WCPS (Web Coverage Processing Service) is a query language specifically designed for retrieving and processing geospatial data. It allows users to perform operations such as trimming, slicing, and applying mathematical and logical expressions to data, generating new results without the need to handle full datasets locally.

User Benefits:

Simplified Workflow: The revamped architecture streamlines the user workflow, abstracting complexities and reducing cognitive load. Users can focus on analysis tasks rather than wrestling with technical intricacies.

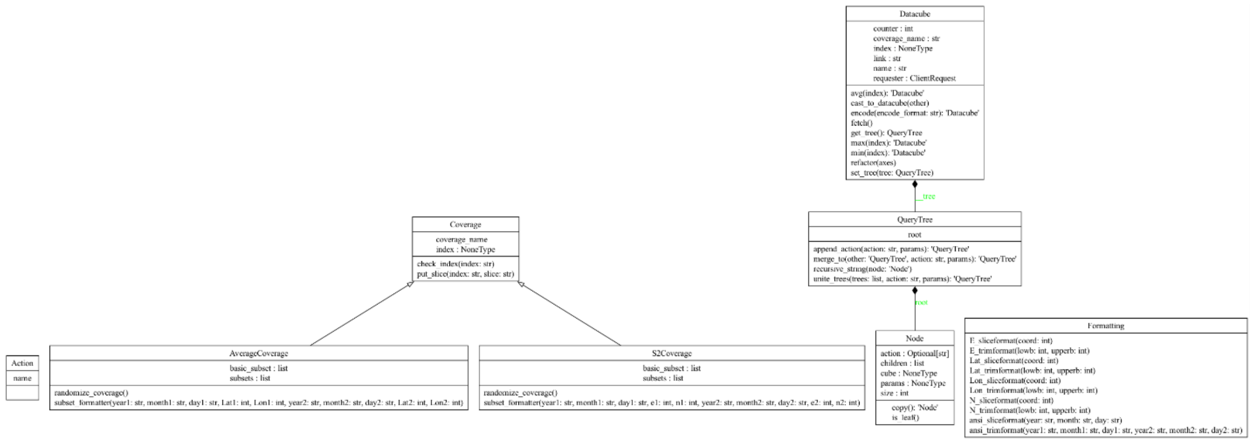
Improved Test Coverage: A comprehensive test suite is integrated into the library, validating functionality across different scenarios. This ensures robustness and reliability, instilling confidence in the library's performance and stability.

Classes

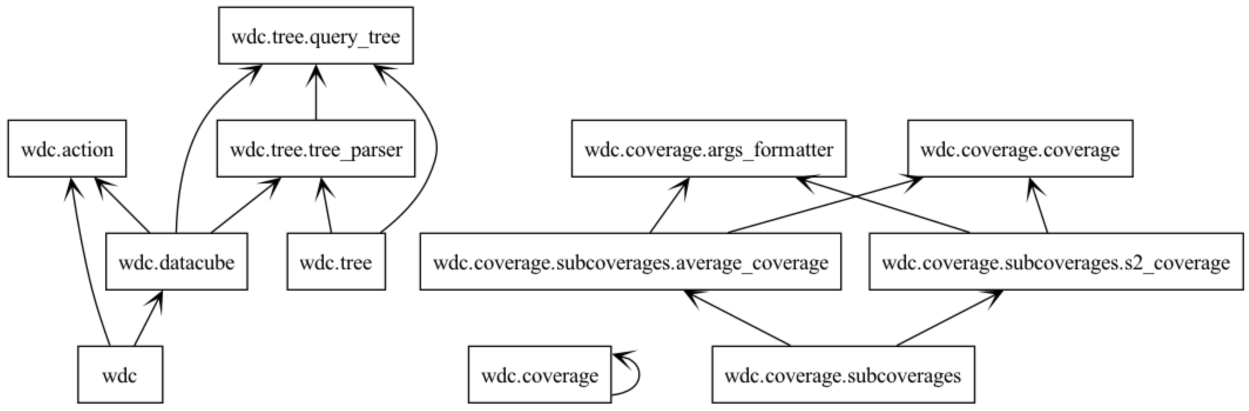
Datacube	class serves as the core component of the library, providing functionality for interacting with geospatial datacubes and performing various operations on them. It encapsulates methods for generating WCPS queries, sending them to the server, and processing the results.
Action	implemented as an enumeration, defines different types of operations that can be applied to datacubes. Each action corresponds to a specific WCPS operation or manipulation on the data
QueryTree	class serves as a representation of operations under datacubes, organized in a tree-like structure. It utilizes the Node class to construct the tree, where each leaf node denotes a valid datacube
Node	class represents individual nodes within the query tree. Each node can either contain a reference to a datacube or represent an operation/action
Formatting	class provides static methods for formatting arguments used in client requests. It offers functionality for formatting subsets, ANSI date slices, E coordinates, N coordinates, latitude, and longitude coordinates.
Coverage	class encapsulates information about coverages, providing functionality for managing coverage names and index subsets.
S2Coverage	class extends the Coverage class and specializes in managing Sentinel-2 (S2) coverages
AverageCoverage	class inherits from the Coverage class and specializes in managing average coverages

IV) Diagram

Class Diagram


Datacube
counter : int
coverage_name : str
index : NoneType
link : str
name : str
requester : ClientRequest
avg(index): 'Datacube'
cast_to_datacube(other)
encode(encode_format: str): 'Datacube'
fetch()
get_tree(): QueryTree
max(index): 'Datacube'
min(index): 'Datacube'
refactor(axes)
set_tree(tree: QueryTree)
__tree
QueryTree
root
append_action(action: str, params): 'QueryTree'
merge_to(other: 'QueryTree', action: str, params): 'QueryTree'
recursive_string(node: 'Node')
unite_trees(trees: list, action: str, params): 'QueryTree'
root
Coverage
coverage_name
index : NoneType
check_index(index: str)
put_slice(index: str, slice: str)
Action
name
AverageCoverage
basic_subset : list
subsets : list
randomize_coverage()
subset_formatter(year1: str, month1: str, day1: str, Lat1: int, Lon1: int, year2: str, month2: str, day2: str, Lat2: int, Lon2: int)
S2Coverage
basic_subset : list
subsets : list
randomize_coverage()
subset_formatter(year1: str, month1: str, day1: str, e1: int, n1: int, year2: str, month2: str, day2: str, e2: int, n2: int)
Node
action : Optional[str]
children : list
cube : NoneType
params : NoneType
size : int
copy(): 'Node'
is_leaf()
Formatting
E_sliceformat(coord: int)
E_trimformat(lowb: int, upperb: int)
Lat_sliceformat(coord: int)
Lat_trimformat(lowb: int, upperb: int)
Lon_sliceformat(coord: int)
Lon_trimformat(lowb: int, upperb: int)
N_sliceformat(coord: int)
N_trimformat(lowb: int, upperb: int)
ansi_sliceformat(year: str, month: str, day: str)
ansi_trimformat(year1: str, month1: str, day1: str, year2: str, month2: str, day2: str)

Activity Diagram


wdc.tree.query_tree
wdc.action
wdc.tree.tree_parser
wdc.coverage.args_formatter
wdc.coverage.coverage
wdc.datacube
wdc.tree
wdc.coverage.subcoverages.average_coverage
wdc.coverage.subcoverages.s2_coverage
wdc
wdc.coverage
wdc.coverage.subcoverages

V) Specification of the Classes

About Datacube

This class (datacube.py) serves as the core component of the library, providing functionality for interacting with geospatial datacubes and performing various operations on them. It encapsulates methods for generating WCPS queries, sending them to the server, and processing the results

ATTRIBUTES:

- **link (string):** URL endpoint of the WCPS server.
- **index (list of Subset):** List of subsets defining the datacube indices.
- **__tree:** QueryTree object representing the tree-like data structure storing operations on the datacube.
- **coverage_name (string):** Name of the datacube coverage.
- **requester:** ClientRequest object for making requests to the server.
- **name (string):** Name of the datacube instance.
- **counter (int):** Class attribute to enumerate new datacubes.

METHODS:

- __init__(link='https://ows.rasdaman.org/rasdaman/ows', index: List[Subset] = None, coverage_name="S2_L2A_32631_TCI_60m"):
  - **Parameters:**
    - link (str): The URL of the WCPS server. Default is 'https://ows.rasdaman.org/rasdaman/ows'.
    - index (List[Subset]): Optional list of Subsets representing the index of the Datacube. Default is None.
    - coverage_name (str): The name of the coverage. Default is "S2_L2A_32631_TCI_60m".
  - **Return Type:** None
  - **Description:** Constructor method initializes a new Datacube object with default or provided parameters.
- get_tree() -> QueryTree:
  - **Return Type:** QueryTree
  - **Description:** Getter method returns the QueryTree object representing the datacube's operation tree.

- set_tree(tree: QueryTree):
    - **Parameters:**
        - tree (QueryTree): The QueryTree object representing the datacube's operation tree.
    - **Return Type:** None
    - **Description:** Setter method sets the QueryTree object representing the datacube's operation tree.
- __add__(self, other) -> 'Datacube':
    - **Parameters:**
        - other: Another Datacube object to perform addition operation.
    - **Return Type:** Datacube
    - **Description:** Method for addition operation on datacubes.
- __sub__(self, other) -> 'Datacube':
    - **Parameters:**
        - other: Another Datacube object to perform subtraction operation.
    - **Return Type:** Datacube
    - **Description:** Method for subtraction operation on datacubes.
- __mul__(self, other) -> 'Datacube':
    - **Parameters:**
        - other: Another Datacube object to perform multiplication operation.
    - **Return Type:** Datacube
    - **Description:** Method for multiplication operation on datacubes.
- __truediv__(self, other) -> 'Datacube':
    - **Parameters:**
        - other: Another Datacube object to perform division operation.
    - **Return Type:** Datacube
    - **Description:** Method for division operation on datacubes.
- avg(self, index: str) -> 'Datacube':
    - **Parameters:**
        - index (str): Optional index for aggregation query.
    - **Return Type:** Datacube
    - **Description:** Method for calculating the average of the datacube with optional index.
- min(self, index: List[Subset] = []) -> 'Datacube':
    - **Parameters:**
        - index (List[Subset]): Optional list of Subsets for aggregation query.
    - **Return Type:** Datacube
    - **Description:** Method for calculating the minimum value of the datacube with optional index.
- max(self, index: List[Subset] = []) -> 'Datacube':
    - **Parameters:**
        - index (List[Subset]): Optional list of Subsets for aggregation query.
    - **Return Type:** Datacube
    - **Description:** Method for calculating the maximum value of the datacube with optional index.

- encode(self, encode_format: str) -> 'Datacube':
    - **Parameters:**
        - encode_format (str): The desired format for encoding.
    - **Return Type:** Datacube
    - **Description:** Method for encoding the datacube to the desired format.
- __getitem__(self, index: List[Subset]) -> 'Datacube':
    - **Parameters:**
        - index (List[Subset]): List of Subsets representing the subindex operation.
    - **Return Type:** Datacube
    - **Description:** Method for subindexing operation on datacube.
- refactor(cls, axes):
    - **Parameters:**
        - axes (list): List of pairs (str, Datacube) representing the name and datacube for each axis.
    - **Return Type:** Datacube
    - **Description:** Method for creating a new datacube with specified indices.
- fetch(self):
    - **Return Type:** Data
    - **Description:** Method for fetching and loading data from the server according to the current request.

About Subset

This class (subset.py) represents a subset operation that can be applied to a datacube. It essentially encapsulates a query string that specifies how to extract a subset of data from the datacube.

Attributes:

- operation: A string representing the operation to be applied to the subset.
- values: A variable number of values representing the parameters of the subset operation.
- query: A string representing the formatted query for the subset.

Methods:

- __init__(self, operation: str, *values): Initializes a Subset object with the specified operation and values. It formats the values into a query string based on the operation and stores it in the query attribute.
- operation: Returns the operation of the subset.
- values: Returns the values of the subset.
- query: Returns the formatted query string representing the subset.
- __str__(self): Returns a string representation of the Subset object, which is the formatted query string.

About Action

This class (action.py) implemented as an enumeration, defines different types of operations that can be applied to datacubes. Each action corresponds to a specific WCPS operation or manipulation on the data

Action Attributes:

- ADD (str): Represents the addition operation between datacubes.
- SUB (str): Represents the subtraction operation between datacubes.
- MULT (str): Represents the multiplication operation between datacubes.
- DIV (str): Represents the division operation between datacubes.
- ENCODE (str): Represents the encoding operation for the datacube.
- AVG (str): Represents the average aggregation operation on the datacube.
- MIN (str): Represents the minimum aggregation operation on the datacube.
- MAX (str): Represents the maximum aggregation operation on the datacube.
- REFACTOR (str): Represents the action to create a new datacube with specified indices.
- SUBINDEX (str): Represents the subindexing operation on the datacube.

Action Methods:

- __str__():
    - **Return Type:** str
    - **Description:** Converts the Action enum instance to its corresponding string value.

About QueryTree

This class (query_tree.py) serves as a representation of operations under datacubes, organized in a tree-like structure. It utilizes the Node class to construct the tree, where each leaf node denotes a valid datacube

ATTRIBUTES:

- root: Node
    - **Data Type:** Node
    - **Description:** The root node of the QueryTree representing the operations under datacubes.

METHODS:

- __init__(cube=None) -> None:
  - **Parameters:**
    - cube: Datacube object. Default is None.
  - **Return Type:** None
  - **Description:** Constructor method initializes a new QueryTree object with an optional cube as the root node.
- merge_to(self, other: 'QueryTree', action: str, params=None) -> 'QueryTree':
  - **Parameters:**
    - other (QueryTree): Another QueryTree to merge with.
    - action (str): Char symbol of actions like +, -, *, /, etc.
    - params (dict): Dictionary of parameters for the new action. Default is None.
  - **Return Type:** QueryTree
  - **Description:** Merge current tree to another tree by creating one new root that refers to the previous trees.
- append_action(self, action: str, params=None) -> 'QueryTree':
  - **Parameters:**
    - action (str): Unary action like scale, encode, etc.
    - params (dict): Dictionary of parameters for the new action. Default is None.
  - **Return Type:** QueryTree
  - **Description:** Append a new action on top of the tree.
- unite_trees(cls, trees: list, action: str, params=None) -> 'QueryTree':
  - **Parameters:**
    - trees (list): List of QueryTree objects.
    - action (str): Unary action like scale, encode, etc.
    - params (dict): Dictionary of parameters for the new action. Default is None.
  - **Return Type:** QueryTree
  - **Description:** Append a new action on top of several trees.
- recursive_string(cls, node: 'Node'):
  - **Parameters:**
    - node (Node): The starting node for constructing the string.
  - **Return Type:** str
  - **Description:** Make a string representation of the QueryTree with the help of depth-first search (DFS) traversal.
- __str__(self):
  - **Return Type:** str
  - **Description:** Return the string representation of the QueryTree

About tree_parser

The tree_parser.py file contains functions for processing a query tree and generating a corresponding query string following the syntax of Rasdaman, a raster data management system.

Methods:

- iterate_tree(node: 'Node', datacubes: set, encodes: set, indexes: set, execution_lines: list, num=0): This function recursively traverses a query tree and generates execution lines based on the operations represented by the tree nodes. It accumulates datacubes, encodings, and indexes encountered during traversal. If an unsupported action is encountered, it raises an AttributeError.
- make_process_query_from_tree(tree: 'QueryTree') -> str: This function creates a query string from a given query tree. It utilizes the iterate_tree function to traverse the tree and generate execution lines. Then, it constructs the query string by organizing the datacubes, indexes, and execution lines according to the Rasdaman syntax. If multiple encodings are encountered or the encoding action is not the last operation in the tree, it raises an AttributeError.

About Node

This class (query_tree.py) represents individual nodes within the query tree. Each node can either contain a reference to a datacube or represent an operation/action

ATTRIBUTES:

- action:
  - **Data Type:** str
  - **Description:** Represents the action associated with the node in the QueryTree.
- cube:
  - **Data Type:** Datacube object or None
  - **Description:** Represents the Datacube associated with the node. It's None if the node is not a leaf.
- children:
  - **Data Type:** list
  - **Description:** List of child nodes of the current node.
- params:
  - **Data Type:** dict or None
  - **Description:** Dictionary of parameters associated with the action of the node.
- size:
  - **Data Type:** int

    - **Description:** Represents the size of the node, including itself and all its children.

METHODS:

- __init__(cube=None, action: str = None, params=None) -> None:
    - **Parameters:**
        - cube (optional): Datacube object. Default is None.
        - action (optional): str representing the action associated with the node. Default is None.
        - params (optional): dict representing parameters associated with the action. Default is None.
    - **Return Type:** None
    - **Description:** Constructor method initializes a new Node object with the provided cube, action, and parameters.
- copy(self) -> 'Node':
    - **Return Type:** Node
    - **Description:** Makes a non-leaf copy of this node and its children.
- is_leaf(self):
    - **Return Type:** bool
    - **Description:** Checks if the node is a leaf or not. Returns True if the node is a leaf (i.e., has no action associated with it), otherwise False.

About Formatting

This class (args_formatter.py) provides static methods for formatting arguments used in client requests. It offers functionality for formatting subsets, ANSI date slices, E coordinates, N coordinates, latitude, and longitude coordinates. It also validates the format of ANSI date, latitude and longitude coordinates.

METHODS:

- subsets_format(subsets: List[Subset]) -> str:
    - **Parameters:**
        - subsets (List[Subset]): List of Subset objects.
    - **Return Type:** str
    - **Description:** Static method to format subsets into a string for client requests.
- ansi_sliceformat(year:str, month:str, day:str) -> str:
    - **Parameters:**
        - year (str): 4-digit number as string.
        - month (str): 2-digit number as string.
        - day (str): 2-digit number as string.
    - **Return Type:** str
    - **Description:** Method to create a specific ANSI date slice format.

- ansi_trimformat(year1:str, month1:str, day1:str, year2:str, month2:str, day2:str) -> str:
  - **Parameters:**
    - year1 (str): 4-digit number as string for the start year.
    - month1 (str): 2-digit number as string for the start month.
    - day1 (str): 2-digit number as string for the start day.
    - year2 (str): 4-digit number as string for the end year.
    - month2 (str): 2-digit number as string for the end month.
    - day2 (str): 2-digit number as string for the end day.
  - **Return Type:** str
  - **Description:** Method to create a specific ANSI date trim format.
- E_sliceformat(coord:int) -> str:
  - **Parameters:**
    - coord (int): Coordinate of E.
  - **Return Type:** str
  - **Description:** Method to create E coordinate slice.
- E_trimformat(lowb:int, upperb:int) -> str:
  - **Parameters:**
    - lowb (int): Lower bound of E subset.
    - upperb (int): Upper bound of E subset.
  - **Return Type:** str
  - **Description:** Method to create E coordinates trim.
- N_sliceformat(coord:int) -> str:
  - **Parameters:**
    - coord (int): Coordinate of N.
  - **Return Type:** str
  - **Description:** Method to create N coordinate slice.
- N_trimformat(lowb:int, upperb:int) -> str:
  - **Parameters:**
    - lowb (int): Lower bound of N subset.
    - upperb (int): Upper bound of N subset.
  - **Return Type:** str
  - **Description:** Method to create N coordinates trim.
- Lat_sliceformat(coord:int) -> str:
  - **Parameters:**
    - coord (int): Coordinate of latitude.
  - **Return Type:** str
  - **Description:** Method to create Lat coordinate slice.
- Lat_trimformat(lowb:int, upperb:int) -> str:
  - **Parameters:**
    - lowb (int): Lower bound of Lat subset.
    - upperb (int): Upper bound of Lat subset.
  - **Return Type:** str
  - **Description:** Method to create Lat coordinates trim.
- Lon_sliceformat(coord:int) -> str:
  - **Parameters:**
    - coord (int): Coordinate of longitude.

    - **Return Type:** str
    - **Description:** Method to create Lon coordinate slice.
- Lon_trimformat(lowb:int, upperb:int) -> str:
    - **Parameters:**
        - lowb (int): Lower bound of Lon subset.
        - upperb (int): Upper bound of Lon subset.
    - **Return Type:** str
    - **Description:** Method to create Lon coordinates trim.

About Coverage

This class (coverage.py) encapsulates information about coverages, providing functionality for managing coverage names and index subsets.

ATTRIBUTES:

- coverage_name: Name of the coverage.
- index: Index of the coverage.

METHODS:

- __init__(coverage_name, index = None): Constructor method initializes a new Coverage object with the provided coverage name and optional index.
    - **Parameters:**
        - coverage_name: Name of the coverage.
        - index (optional): Index of the coverage. Defaults to None.
- check_index(index: str) -> bool: Method to check if an index exists in the coverage.
    - **Parameters:**
        - index (str): Index to check.
    - **Return Type:** bool
    - **Description:** Returns True if the index exists in the coverage, otherwise False.
- put_slice(index: str, slice: str) -> bool: Method to put a slice into the coverage.
    - **Parameters:**
        - index (str): Index where the slice will be put.
        - slice (str): Slice to put into the coverage.
    - **Return Type:** bool
    - **Description:** Puts the given slice into the coverage at the specified index. Returns True if successful, otherwise False.

About S2Coverage

This class (s2_coverage.py) extends the Coverage class and specializes in managing Sentinel-2 (S2) coverages

ATTRIBUTES:

- basic_subset: Initial values to use for coverages, defined as a list.

METHODS:

- __init__(coverage_name, subsets = basic_subset, index = None): Constructor method initializes a new S2Coverage object with the provided coverage name, subsets, and optional index.
    - **Parameters:**
        - coverage_name: Name of the coverage.
        - subsets (optional): Subsets for the coverage. Defaults to basic_subset.
        - index (optional): Index of the coverage. Defaults to None.
- subset_formatter(year1:str, month1:str, day1:str, e1:int, n1:int, year2:str = None, month2:str = None, day2:str = None, e2:int = None, n2:int = None) -> list: Static method for formatting subsets for coverages from S2 class.
    - **Parameters:**
        - year1, month1, day1: For initial date.
        - e1: For initial E coordinate.
        - n1: For initial N coordinate.
        - year2, month2, day2 (optional): For upper bound date, if trimming is wanted.
        - e2 (optional): For upper bound E coordinate, if trimming is wanted.
        - n2 (optional): For upper bound E coordinate, if trimming is wanted.
    - **Return Type:** list
    - **Description:** Generates formatted subsets containing the ansi, E, N axis, ready to be sent to a request.
- randomize_coverage(): Static method for selecting a random ID of an S2 coverage and random subsets of descriptive coverage values.
    - **Return Type:** tuple (name, subsets)
    - **Description:** Returns a random coverage ID from the list of possible IDs and a list of formatted subsets of the ansi, E, N axis.

About AverageCoverage

This class (average_coverage.py) inherits from the Coverage class and specializes in managing average coverages

ATTRIBUTES:

- basic_subset: Initial values to use for coverages, defined as a list.

METHODS:

- __init__(coverage_name, subsets = basic_subset, index = None): Constructor method initializes a new AverageCoverage object with the provided coverage name, subsets, and optional index.
  - **Parameters:**
    - coverage_name: Name of the coverage.
    - subsets (optional): Subsets for the coverage. Defaults to basic_subset.
    - index (optional): Index of the coverage. Defaults to None.
- subset_formatter(year1:str, month1:str, day1:str, Lat1:int, Lon1:int, year2:str = None, month2:str = None, day2:str = None, Lat2:int = None, Lon2:int = None) -> list: Static method for formatting subsets for coverages from Average class.
  - **Parameters:**
    - year1, month1, day1: For initial date.
    - Lat1: For initial latitude coordinate.
    - Lon1: For initial longitude coordinate.
    - year2, month2, day2 (optional): For upper bound date, if trimming is wanted.
    - Lat2 (optional): For upper bound latitude coordinate, if trimming is wanted.
    - Lon2 (optional): For upper bound longitude coordinate, if trimming is wanted.
  - **Return Type:** list
  - **Description:** Generates formatted subsets containing the ansi, Lat, Lon axis, ready to be sent to a request.
- randomize_coverage(): Static method for selecting a random ID of an Average coverage and random subsets of descriptive coverage values.
  - **Return Type:** tuple (name, subsets)
  - **Description:** Returns a random coverage ID from the list of possible IDs and a list of formatted subsets of the ansi, Lat, Lon axis.

About Connection Requester (ClientRequest)

This class (requester.py) acilitates communication with a remote server offering a Web Coverage Service (WCS). It provides methods to interact with the server, such as retrieving service capabilities, describing coverages, fetching subset coverages, and evaluating WCPS (Web Coverage Processing Service) queries.

ATTRIBUTES:

- service_endpoint: Base URL for the service.

METHODS:

- __init__(base_wcs_url = service_endpoint + "?service=WCS&version=2.0.1"): Constructor method initializes a new ClientRequest object with the provided base WCS URL. If not specified, it uses the default value.
  - **Parameters:**
    - base_wcs_url (optional): Base URL for the Web Coverage Service (WCS) endpoint. Defaults to service_endpoint + "?service=WCS&version=2.0.1".
- get_capabilities(): Method to retrieve the XML description of service capabilities and overview of coverages.
  - **Parameters:** None
  - **Returns:** Response object containing the XML description.
  - **Description:** Sends a GET request to the service endpoint with the GetCapabilities request and returns the response.
- describe_coverage(cov_id): Method to retrieve the XML-encoded description of a specific coverage.
  - **Parameters:**
    - cov_id (str): Coverage ID (e.g., "S2_L2A_32631_TCI_60m").
  - **Returns:** Response object containing the XML-encoded coverage description.
  - **Description:** Sends a GET request to the service endpoint with the DescribeCoverage request for the specified coverage ID and returns the response.
- get_subset_coverage(cov_id, subsets, encode_format=None): Method that returns an encoded subset coverage with already defined subsets.
  - **Parameters:**
    - cov_id (str): Coverage ID (e.g., "S2_L2A_32631_TCI_60m").
    - subsets (list): List of descriptive subsets of the coverage.
    - encode_format (optional): Encoding format for the coverage. Defaults to None.
  - **Returns:** Response object containing the encoded subset coverage.
  - **Description:** Constructs the request URL with the specified coverage ID and subsets, sends a GET request to the service endpoint, and returns the response.
- evaluate_query(query): Method to send a WCPS query for evaluation.
  - **Parameters:**
    - query: WCPS query to be evaluated.
  - **Returns:** Content of the response.
  - **Description:** Sends a POST request to the service endpoint with the WCPS query data and returns the response content.

VI) Testing

About Testing

We've employed the Pytest framework to conduct unit tests on the functionalities provided by the 'wdc' library.

To execute these tests using Pytest, ensure that Pytest is installed in your Python environment. You can achieve this by running 'pip install pytest' in your terminal.

Pytest follows a naming convention to automatically discover test files and functions. Testing files must be surffixed with 'test_' and all test functions within these files should begin with 'test_' as well.

To run tests within a specific directory, navigate to that directory in your terminal and simply type 'pytest'. This command will execute all unit tests within the directory.

Remember to import classes from the 'wdc' library into your test files to access the functionalities being tested.

Tests

Test name	Source
args_formatter_test	./tests/args_formatter_test.py
comparison_test	./tests/comparison_test.py
datacube_test	./tests/datacube_test.py
query_tree_test	./tests/query_tree_test.py
random_coverage_test	./tests/random_coverage_test.py
requester_test	./tests/requester_test.py
tree_parser_test	./tests/tree_parser_test.py

args_formatter_test

TestAnsi:

- test_ansi_slice_output: Verifies that the ansi_sliceformat function correctly formats a date slice into the expected ANSI format.
- test_ansi_trim_output: Ensures that the ansi_trimformat function correctly formats a date trim into the expected ANSI format.

TestSubsetE:

- test_E_slice_output: Validates that the E_sliceformat function correctly formats an E slice into the expected format.
- test_E_trim_output: Checks that the E_trimformat function correctly formats an E trim into the expected format.

TestSubsetN:

- test_N_slice_output: Checks if the N_sliceformat function correctly formats an N slice into the expected format.
- test_N_trim_output: Verifies that the N_trimformat function correctly formats an N trim into the expected format.

TestLat:

- test_Lat_slice_output: Ensures that the Lat_sliceformat function correctly formats a latitude slice into the expected format.
- test_Lat_trim_output: Validates that the Lat_trimformat function correctly formats a latitude trim into the expected format.

TestLon:

- test_Lon_slice_output: Checks if the Lon_sliceformat function correctly formats a longitude slice into the expected format.
- test_Lon_trim_output: Verifies that the Lon_trimformat function correctly formats a longitude trim into the expected format.

comparison_test

test_temporal_change_analysis:

- Compares the result of a specific query executed against the Rasdaman server (response.content) with the result of an equivalent query executed using the Datacube class (b.fetch()).
- Verifies whether the output of the query, which involves mathematical operations and encoding, matches between the two methods.

test_refactor:

- Compares the output of a query executed against the Rasdaman server (response.content) with the output of an equivalent query executed using the Datacube class (f.fetch()).
- Validates whether the result of the query, which involves restructuring and encoding, is consistent between the two methods.

test_subindex:

- Compares the response of a query executed against the Rasdaman server (response.content) with the result of an equivalent query executed using the Datacube class (c.avg().fetch()).
- Verifies whether the outcome of the aggregation operation (avg) on the datacube matches between the two approaches.

datacube_test

test_add_output:

- Creates two Datacube instances, performs addition (a + b), and verifies whether the resulting tree structure matches the expected output.
- Checks if the addition operation on Datacube instances generates the correct tree representation.

test_sub_output:

- Similar to the previous test, but for subtraction (a - b).
- Validates whether the subtraction operation on Datacube instances produces the expected tree structure.

test_mult_output:

- Similar to the previous tests, but for multiplication (a * b).
- Ensures that the multiplication operation on Datacube instances yields the correct tree representation.

test_div_output:

- Similar to the previous tests, but for division (a / b).
- Verifies whether the division operation on Datacube instances generates the expected tree structure.

test_equation_output:

- Constructs a more complex equation ((a + b) * (a + b)) and checks if the resulting tree structure matches the expected output.
- Validates whether complex equations involving multiple operations on Datacube instances generate the correct tree representation.

test_equation_performance:

- Measures the execution time of repeated addition (a = a + b) operations on Datacube instances.
- Asserts that the execution time does not exceed a predefined time bound, ensuring reasonable performance.

test_fetch:

- Fetches data using the fetch() method and compares the SHA-256 hash of the fetched data with a predefined hash.
- Validates whether fetching data from the server using the Datacube class produces the expected result.

query_tree_test

test_reference_validness:

- Creates two Datacube instances (a and b) and obtains their respective tree representations.
- Merges these trees together using the merge_to method with the operator '+' to create a new tree c.
- Stores the string representation of c in a temporary variable tmp.
- Performs random operations (merges with '/' and '*') on a and b trees to simulate changes.
- Checks whether the string representation of c remains unchanged after these random operations.
- Asserts that the string representation of c before the random operations is equal to the string representation of c after the random operations.

random_coverage_test

Testing S2Coverage.randomize_coverage():

- Prints a header indicating the output is for the first call of randomize_coverage for S2Coverage.
- Calls the randomize_coverage method of S2Coverage.
- Prints the returned name and subset.
- Prints a header indicating the output is for the second call of randomize_coverage for S2Coverage.
- Calls randomize_coverage again and prints the returned name and subset.

Testing AverageCoverage.randomize_coverage():

- Prints a header indicating the output is for the first call of randomize_coverage for AverageCoverage.
- Calls the randomize_coverage method of AverageCoverage.
- Prints the returned name and subset.
- Prints a header indicating the output is for the second call of randomize_coverage for AverageCoverage.
- Calls randomize_coverage again and prints the returned name and subset.

requester_test

TestGetCapabilities:

- Method: test_get_capabilities_output
- Purpose: Tests the get_capabilities method of the ClientRequest class.
- Assertion: Checks if the HTTP status code of the response is 200, indicating a successful request.

TestDescribeCoverage:

- Method: test_describe_coverage_output
- Purpose: Tests the describe_coverage method of the ClientRequest class.
- Assertion: Checks if the HTTP status code of the response is 200, indicating a successful request.

TestGetSubsetCoverage:

- Method: test_get_subset_coverage_output
- Purpose: Tests the get_subset_coverage method of the ClientRequest class.
- Assertion: Checks if the HTTP status code of the response is 200, indicating a successful request.

tree_parser_test

test_small_query:

- Purpose: Tests the parsing of a small query tree into a process query string.
- Assertion: Compares the generated process query string against an expected query string.

test_query_integers:

- Purpose: Tests the parsing of a query tree with integer values into a process query string.
- Assertion: Compares the generated process query string against an expected query string.

test_avg:

- Purpose: Tests the parsing of an average query operation into a process query string.
- Assertion: Compares the generated process query string against an expected query string.

test_min:

- Purpose: Tests the parsing of a minimum query operation into a process query string.
- Assertion: Compares the generated process query string against an expected query string.

test_max:

- Purpose: Tests the parsing of a maximum query operation into a process query string.
- Assertion: Compares the generated process query string against an expected query string.

test_refactor:

- Purpose: Tests the parsing of a refactored query tree into a process query string.
- Assertion: Compares the generated process query string against an expected query string.

test_subindex:

- Purpose: Tests the parsing of a query tree with sub-indices into a process query string.
- Assertion: Compares the generated process query string against an expected query string.

VII) Document Change Log:

Version	**Authors**	**Date**	**Descriptions**
Sprint1	Diana Maria Harambas Mark Kanafeev	18 April 2024	Original authors of the design, codes, test and documentation
Sprint2	Zarina Abulkassova Daria Soloveva	7 May 2024	Added - document file - args_formatter_test.py
Sprint3	Sulehri, Sabeeh Ur Rehman Wang Yat Sin	15 May 2024	Amended args_formatter.py with format validation of date, latitude and longitude values. Added test cases in args_formatter_test.py for format validation of date, latitude and longitude, Amended the in-line documentations in sources to enhance the readability Added Document Change Log in the documentation and readme.md