Assignment 2: Automating the boring stuff

Contents

The goal of this assignment is to introduce you to the capability of using the operating system (OS) as a way of managing your directories and files. You will get good practice in managing path variables and in using unit tests and temporary directories as indispensable support to your program-writing.

Updates

- Removed suggestion that you should check if strings *can* be paths in Task 2.2. Almost any string can be a path, and that's up to the OS to handle, and note that `is_gas_csv` only checks the paths themselves, ignoring whether there is a file at the path.
- Noted that the oil and gas directory is spelled `oil_and_gass`

Practical information

Total points: 15 (+ 2 bonus points)

This assignment contains three tasks, each being broken into several subtasks. The order of the tasks is adapted such that you develop your code incrementally and test your functions as you go.

Before you start, please note the following:

Skip to main content

- Your solution to this mandatory assignment needs to be placed in the directory `assignment2` in your GitHub repository.
- We have written a code skeleton for you containing the functions you need to write and the corresponding tests. Make sure to keep the function names, arguments, and return types as specified as this makes grading easier. *However, you are free to write whatever helper functions you want.*
- Download the code skeleton here: `assignment2.zip` and extract it in your homework repo. It should create the `assignment2` directory in your repo.
- Add and commit the initial version to git, so you can see what code you've changed during the assignment.
- Your code needs to be well commented and documented. **All** functions need to have doc-strings explaining what the function does, how it should be used, an explanation of the parameters and return value(s) (including types).
- Writing tests to run with `pytest` is not just a great way to check your code, but it also shows how you expect your code to work. We have provided a collection of tests, which you are asked to complete to test the functions you will use. Feel free to add your own *additional* tests if you want, but they should not replace any of the tests you are explicitly asked to write. To make testing easier for you during code development, we have created *test markers* for each specific task that let you select only specific tests, and avoid seeing failures for code that is not yet developed. To see all the markers, type `python3 -m pytest --markers` in the terminal. If you, for instance, want to run the test in task 1.2 only, you may type `python3 -m pytest -v -m task12 tests` in your terminal, when you are in `assignment2`. If you want to run all tests in a specific script, type `python3 -m pytest -v tests/test_[name of script].py`.
- Make sure to read all the sections preceding the tasks as they contain a lot of information necessary to understand the tasks and our expectations to your implementation.

Below is a list of the relevant libraries and modules for this assignment. Click on the names to get to their documentation:

- pathlib
- pytest
- shutil
- tempfile
- os

These libraries are used in the code provided to you:

Skip to main content

- numpy
- matplotlib

Note: Pathlib, shutil, tempfile and os are already included in the Python package, but the remaining ones must be locally installed in your environment.

There are a lot of similar functions in both os and pathlib libraries of Python. We recommend and encourage you to use pathlib as it has more advantages and a simpler syntax. **We have provided a short guide to pathlib in the appendix of this assignment**. It is by no means extensive, and you should always check the proper documentation for the functions you use.

Tip: If a task asks for something that has not yet been covered in class, you can make a comment saying e.g. `# TODO: Raise TypeError` and return at a later point.

Motivation

In many work or life situations you might find yourself working on a big repository or a directory, creating or editing content within it. Doing things by hand can quickly become tedious if there are a lot of changes to be made, or if the directory has a complex structure.

“Yes,” you might think, “but letting a program do it can lead to unexpected errors”. That is true, but to avoid that we need unit testing. It enables us to detect a lot of errors before the real execution. Additionally, with the machinery of temporary directories, you can test the final results in completely isolated directories, and specialize the behavior of your program to your needs.

Getting started

In this assignment, you will be working on gas pollution statistics. After unpacking the provided `.zip` file, your working directory should look like this:

```
assignment2/
    - analytic_tools
    - pollution_data
    - tests
    - analyze_pollution_data.py
    - pyproject.toml
    - README.md
```

Skip to main content

This is quite a lot of files! Do not worry, everything will be explained below, and you will only modify a few of them.

First, take a short glimpse on the contents of the `pollution_data` directory by clicking through it in your file browser.

This directory contains data on amounts of air pollution in Norway, measured in " 1000 ton CO_2 equivalents" as function of year for the period $1990 - 2022$, from different sources: air traffic, agriculture, industry, oil and gas production and road_traffic. The air pollution data is divided into different types of greenhouse gasses. The data retrieved and adapted from SSB, and you can click here to see the complete source dataset.

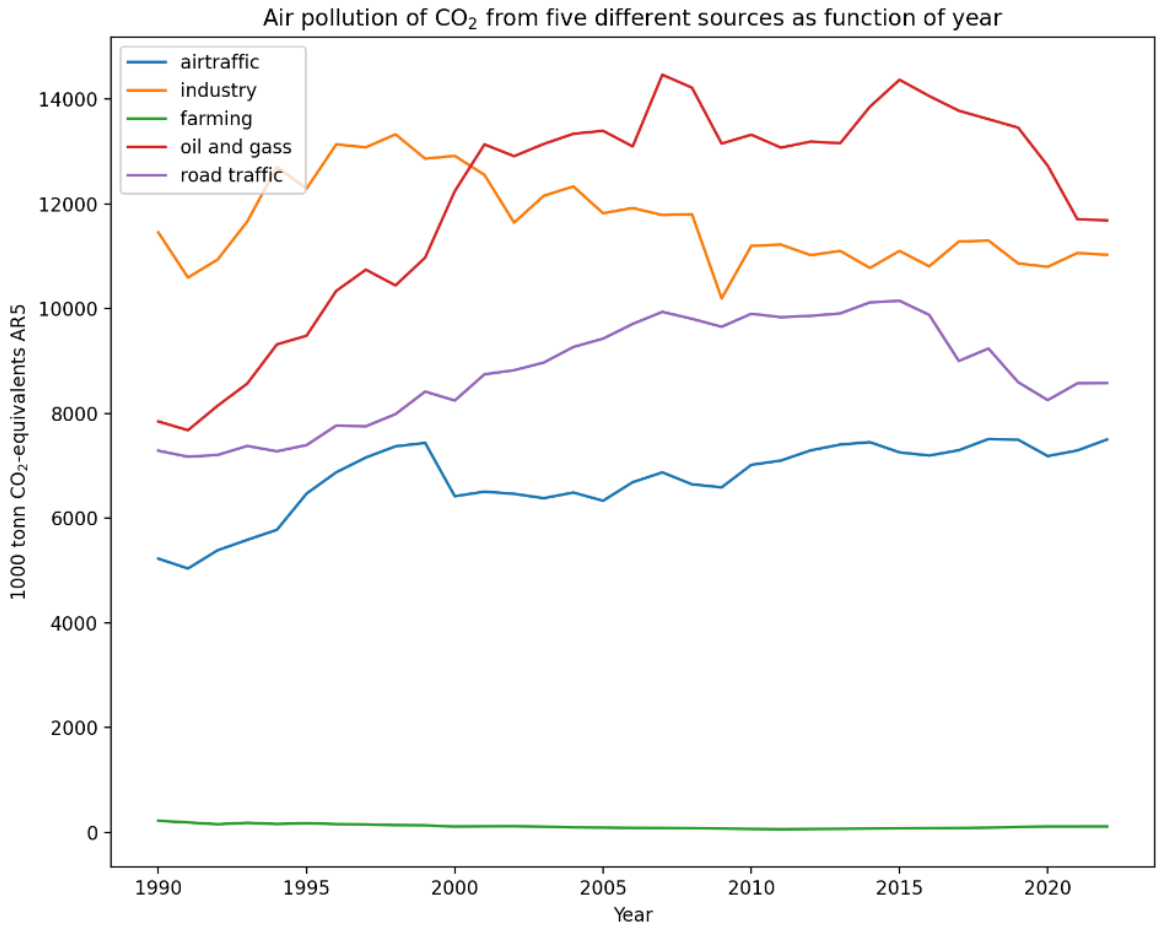
The possible greenhouse gasses that could be documented are $CO_2, CH_4, N_2O, SF_6, H_2$. These are some of the gasses monitored by *World meteorological organization*, but not all of them might be present in your `pollution_data` directory.

Unfortunately, as you saw, the `pollution_data` is quite messy because people who have worked on it earlier, have not been considerate, and left all of their intermediate files among the real data. However, you are made aware that the true, original data files follow this pattern:

- The original files all have a `.csv` suffix
- Their names follow `[gas_formula].csv` pattern, where `[gas_formula]` is replaced by one of the elements in `['CO2', 'CH4', 'N2O', 'SF6', 'H2']` list without the brackets. For instance, `CO2.csv` is an original file.
- The files are stored in subdirectories distinguished by source. If, for instance, two different sources pollute with CO_2, they will both contain a file named `CO2.csv`.

Your task will be to identify all the original data files, make a copy and store them in a separate directory. In this directory, the files will be grouped by gas type and not by source type. Then, you will use `plot_pollution_data(by_gas_dir, fig_dir)` function in `analytic_tools/plotting.py` to produce a plot for each gas, where the emitted amount of each gas is plotted against years for each of the corresponding sources. The resulting plot for CO_2, for instance, will look like:

Skip to main content



The resulting structure of your `assignment2` directory will then be:

```
assignment2/
    - analytic_tools
    - pollution_data
        - by_src
    - pollution_data_restructured
        - by_gas
        - figures
    - tests
    - analyze_pollution_data.py
    - pyproject.toml
    - README.md
```

Here, `pollution_data_restructured` is derived from `pollution_data` and contains a subdirectory `by_gas` and `figures`. The latter is where the resulting `.png` plots must be stored.

To help you understand the tasks better, we provide you with an example of a `pollution_data` directory, which is a simplified version of the one that you will get. Some of the examples in the tasks will refer to this directory as 'example configuration'. This how the example configuration looks:

Skip to main content

Figure 1: Example configuration

```
pollution_data
    - by_src
        - src_agriculture
            - N2O_455.npy
            - H2.csv
            - H2_mkL.csv
        - src_airtraffic
            - CO2.csv
            - CO2_GHk.csv
            - CH4_327.npy
        - src_oil_and_gass
            - CH4.csv
            - SF6_Hgt.csv
            - H2O.csv
            - CO2.csv
```

note: this is an *example* and a subset. Your exact list of sources and gasses will vary, and your code should not be sensitive to what specific values are in the source names `src_whatever`.

Having received so many files, you deserve a good explanation of them. Here is a summary:

- `analytic_tools` is a package containing two modules `plotting.py` and `utilities.py`. The former contains functions that do the plotting of the data. You will use `plot_pollution_data(by_gas_dir, fig_dir)`, in `analyze_pollution_data.py` script. All functions inside `plotting.py` are provided to you and you must not modify them, but you should understand what they do by reading their documentation. The latter, `utilities.py`, is a module that your will work on, containing functions that perform specific operations on the files inside `pollution_data` directory.
- `tests` is a directory containing `test_utilities.py`, `test_analyze_pollution_data.py`, `test_handin.py` and `conftest.py`. The former is a test script that you will develop to test all the functions in the `analytic_tools/utilities.py` module and run with pytest. `test_analyze_pollution_data.py` is a script provided to you to test the functions within `analyze_pollution_data.py`. `test_handin.py` is another script provided to you to check your assignment directory before submission. `conftest.py` is supplementary script used by pytest. You do not need to edit this script, and you do not have to understand it. More details are given in the *Testing* section below.
- `analyze_pollution_data.py` is the script you will develop, that will perform the restructuring and the plotting of the data. This script will use the `analytic_tools/utilities.py` and `analytic_tools/plotting.py` modules to produce

Skip to main content

the final result, i.e. the plots of gas pollution as function of year and store them in a separate directory.

- `pyproject.toml` is a file that enables you to use `analytic_tools` as a package, you should neither understand it, nor modify it.
- `README.md` is a description of the `analytic_tools` package and how to install it and make it editable (since you will be editing the `analytic_tools/utilities.py` module).

> **Set up task:** (0 points) Install `analytic_tools` as a package in your local environment by following the instructions in `README.md` file.

Familiarize yourself with the contents of `analyze_pollution_data.py`, `analytic_tools/utilities.py` and `tests/test_utilities.py` before proceeding from here. These are the only files you will be asked to modify. You are allowed to create additional, supplementary files/directories/functions if you find it necessary during the assignment, but make sure to describe them well.

Testing

In the `tests/test_utilities.py` script, you have been provided several function declarations. Some of them take `example_config` as argument. Technically, `example_config` is a pytest fixture, defined in `conftest.py`. **Fixture is not a focus in this assignment**, so do not be discouraged by the technicalities of this script, and do not edit it.

You should focus on the following:

- If a function has `example_config` as argument, then the code in the function body is executed in a temporary directory that is pre-configured to contain the example configuration of the `pollution_data` directory presented in *Figure 1*. You can treat `example_config` as a path variable, pointing to a virtual, temporary directory. This mimics the following set-up:

```
example_config/
    - pollution_data
    - test_utilities.py
```

- If a function does not take any argument, then no temporary directory is created.

All of the functions within `analytic_tools/utilities.py` are made available to this script by importing the analytic_tools package. Thus, even though `example_config` does not contain

Skip to main content

`analytic_tools/utilities.py` by importing them properly into the `tests/test_utilities.py` script. Thus, you can treat `analytic_tools` as any other package, like `numpy` for instance.

Remember to **not edit the names of the pre-existing test functions, nor their parameters**. However, you can always create your own additional test functions if you want, and choose the setting you want to test them in.

Optional info, if you are curious about pytest fixtures: see fixture documentation.

Error handling

As an important part of this assignment, you must make sure to do proper error handling in all of the functions that you will create. Since most of the function arguments will be paths, these are the (least) checks you should do:

- Is the path pointing to a real object, it the function assumes that it is real?
- Is the object of the correct type? For example, if you expect a file, the object should be a file and not a directory.
- If the object is a file, does it have the correct suffix? (`.csv` is an example of a suffix)

`pathlib.Path` objects have a lot of properties and methods that let you check these things directly, make sure to take a good look at the documentation. Some very useful methods to look up are `Path.exists(), Path.is_dir(), Path.is_file()`.

Remember to test the error handling as well in the functions of `tests/test_utilities.py`. A convenient tool to do this with is pytest.raises.

It is standard practice for anything that is meant to accept a Path object to also be able to accept a plain string that contains the same information. Given that you use pathlib, you may do a conversion at the beginning of the function body:

```
def do_something_with_paths(path: str | Path):
    '''This function does something with a path

    Parameters:
        path (str or pathlib.Path) : A path to something

    Returns:
        None
    '''
```

Skip to main content

```
    # Function does something
```

If the `path` object happens to be of some type that is not convertible to a path, then pathlib.Path will raise a *TypeError*. Good practice is to do custom error handling right afterwards, such that you have good control of your types and values:

```
def do_something_with_csvfiles(path: str | Path):
    '''This function does something a .csv file pointed to by path

    Parameters:
        path (str or pathlib.Path) : A path to some csv file

    Returns:
        None
    '''

    path = Path(path)   # Will raise type-error if path is an int for example

    # Check if the path exists and points to a file
    if not path.is_file():
        raise #A descriptive error type here (optionally with a message)

    # Check the suffix
    elif path.suffix != ".csv":
        raise #A descriptive error type here (optionally with a message)

    # Function does something
```

Note that there is no need to encapsulate the rest of the function body in an `else`-block. If an error is raised, the execution halts at the corresponding line, and nothing is executed below it.

This is just an example of one way of doing error handling. What you should check for and what types of errors you should raise will depend on the situation and your needs. It is important that you use *concrete exceptions* as much as possible, since they are more informative. The key takeaway here is that you should always check that your function receives what you expect before proceeding to do any work.

We will be running tests on your submission to check the error handling. For this purpose, we define below which error we expect you to raise in different situations, but it will also be explicitly stated in the tasks:

> - Expected a directory, but received a file: *NotADirectoryError*
> - Expected an existing directory, but received a non-existing one: *NotADirectoryError*

Skip to main content

- Expected parameter of type A, but received of type B: *TypeError*
- Expected an argument that has the right type but an inappropriate value: *ValueError*. For example, if a function expects a `.csv` file, but receives a `.txt` files, this exception should be raised.
- Expected an existing file, but received a non-existing one: *FileNotFoundError*

If you are converting a path-parameter using `Path()` at the beginning of a function, you do not need to make extra checks for whether the path parameter is a `str` or `pathlib.Path` beforehand, since calling `Path(path)` will raise appropriate **TypeError** if it's not already a str or Path.

Tip: If you are not familiar working with paths and pathlib, we have created a short guide that you can check out before proceeding with the tasks. The guide is attached in the appendix, named ***A note on paths and the magic of pathlib*** at the end of this assignment. Taking your time to become familiar with the methods and properties of pathlib.Path objects may be well worth it, since many of the tasks in this assignment will ask you to do something with paths. Treating them as strings might require you to write more code and handle some edge-cases by hand.
Having a clear picture of the directory you are working in is an important first step when working with files and data. However, manually clicking through a directory tree can be both boring and confusing, especially if there are a lot of subdirectories and different files stored within it, as it is in your case. In this task, you will develop functions that are meant to help you understand the structure of `pollution_data`.

Task 1.1

(1 point) Inside `analytic_tools/utilities.py`, complete the function named `get_diagnostics`, that takes a path to a directory and counts up the following content within its directory tree:

- Total number of files (all types)
- Total number of subdirectories
- Total number of .csv files
- Total number of .txt files
- Total number of .npy files
- Total number of .md files
- Total number of other files

Skip to main content

and returns a dictionary of these values. Each number must be the total for whole directory tree, i.e. you must look into all subdirectories, sub-subdirectories etc.

(0.5 points) Implement error handling in `get_diagnostics`. The function should raise:

- *TypeError* if it expected a path, but received an object that is not path-like (for example an int or a bool)
- *NotADirectoryError* if it expected a path to a directory, but received a path to something else
- *NotADirectoryError* if it expected a path to an existing directory, but received a path to a non-existing one

Tip: If `dir` is a pathlib.Path object, there exists a convenient method to traverse this directory from down to top (reversely): `dir.rglob('*')`. You are strongly encouraged to take a look at the documentation of this method, as it might save you for a lot of work here!

Another tip: If you want to search for specific file type within the whole directory tree, then `dir.glob('**/*.[suffix]')` is a useful method. `[suffix]` may be `csv` for instance. Look it up!

Task 1.2

(0.5 points) Inside the `tests/test_utilities.py` script, complete the test-function `test_get_diagnostics` and test `get_diagnostics`. This test-function takes `example_config` as argument, meaning that your are working in a temporary directory containing `pollution_data` as in *Figure 1*. Use *Figure 1* to manually count up the true values of all objects in the dictionary that `get_diagnostics` returns, and use this in the assertion of the result. You must check that all elements in the dictionary are counted up correctly.

(0.5 points) Inside the `tests/test_utilities.py` script, complete the test-function `test_get_diagnostics_exceptions` and test the error handling of `get_diagnostics`. All of the `if-raise` statements inside the function's body should be triggered and asserted. Can you think of more edge-cases that would raise an error you haven't accounted for? If so, revisit the function definition and add more to the error handling. `test_get_diagnostics_exceptions` takes two arguments: `exception` and `dir`, which are used in the `pytest.mark.parametrize` framework (see note below). You may add your tests to the `[]` brackets.

Skip to main content

Feel free to add new sub-directories and files to `example_config` inside this test function's body, to test more configurations and edge-cases (but *do not remove* any). Look up the pathlib guide we have provided as an appendix for a convenient way to create new files and directories. NB: adding these does not affect `example_config` fixture in general, just the local temporary directory your function executes in.

It is part of your task to decide how much testing is sufficient.

Test your implementation by typing:

```
python3 -m pytest -v -m task12 tests
```

A note on assertions and parametrization of tests with pytest

Using assertions is convenient when testing implementation. For example, one important check is that the return type is what it is expected to be. This can be done as:

```
def test_function_that_returns_an_int():
    """Docstring here"""
    solution_int = 5
    result = function_that_returns_an_int(some_parameter)
    assert isinstance(result, int), f"Return type was {type(result)}, but expected
    assert result == solution_int, f"function_that_returns_an_int did not return co

# some more code here
```

If `result` is actually of type int, then the first assertion will be `True`, and the code below it will be executed. If the return type is something else, then the assertion will be `False` and the assertion will raise `AssertionError`, displaying the message written after the comma to the terminal. As in the usual case of raising exceptions, nothing after the assertion will be executed. The same happens for every following assertion. Pay attention to the order of the assert statements. If the return type is not `int`, then it does not make sense to compare `result` and `solution_int`, and you may be misled by the type of error this will raise. Make sure to create a logical structure with your assertions.

However, in many cases you may want to make several assertions, regardless if one of them fails. Instead of writing several functions, pytest provides a convenient way to *parametrize* your test functions, such that they test for several cases subsequently, but independently. Assume you have the following function:

Skip to main content

```
    Parameters:
        - a (int): One coefficient
        - b (int): Another coefficient
    Returns:
        - (int): The product a times b
    """
    return a * b
```

If you want to test for several ints independently, you may parametrize like this:

```
import pytest

@pytest.mark.parametrize(
    "a, b, solution",
    [
        (2, 2, 4),
        (1, 1, 1),
        (-2, -5, 10)
    ]
)
def test_multiply_numbers(a, b, solution):
    """Test multiply_numbers

    Parameters:
        etc ...
    """
    result = multiply_numbers(a, b)
    assert result == solution, f"Expected {solution}, but got {result}"
```

This will make `test_multiply_numbers` run *three* times subsequently, for each of the values of `a, b, solution`. If one fails, the others will not be affected by it. You can read more about pytest parametrization here. The **important** syntax to take away is:

- The parametrization must be a decorator above the corresponding test-function.
- The names of variables between "" in `@pytest.mark.parametrize` must be taken as arguments in the corresponding test-function in order for it to use them.
- The order of values in the parentheses () of `@pytest.mark.parametrize` must correspond to the order of variables declared in between the "".

> **Task 1.3**
>
> **(0.5 points)** Inside `analytic_tools/utilities.py`, complete the function named `display_diagnostics`, that takes a path to a directory and a dictionary of its contents, which is of the same type as the return value of `get_diagnostics` described above. The function should display the following, to the terminal:

Skip to main content

- Each key in the dictionary and its corresponding value

(0.5 points) Implement error handling in `display_diagnostics`. The function should raise:

- *TypeError* if it expected a path, but received an object that is not path-like (for example an int or a bool)
- *NotADirectoryError* if it expected a path to a directory, but received path to something else
- *NotADirectoryError* if it expected a path to an existing directory, but received a path to a non-existing one
- *TypeError* if it expected a dictionary as its second parameter, but received an another type.

The output might look similar to this (but you can choose your own style):

```
Diagnostics for foo/dir/subdir/pollution_data:
-----------------------------------------------
Number of files: x
Number of folders: xx
Number of .csv files : xxx
.
.
.
-----------------------------------------------
```

Task 1.4 (Bonus task 1)

(0.5 bonus points) Inside `analytic_tools/utilities.py`, complete the function named `display_directory_tree`, that takes a path to a directory and a `max_files` variable as input, and displays a nice directory tree to the terminal.

`display_directory_tree` should:

- explicitly show the root directory name
- separate each level inside the directory tree, such that it is clear what the level's depth is relative to the root directory
- display all the sub-directories present at each level
- display `max_files` arbitrary files present at each level
- indicate with `...` mark on the next entry if there exists more files that `max_files`

Skip to main content

> **(0.5 bonus points)** Implement error handling in `display_directory_tree`. The function should raise:
>
> - *TypeError* if it expected a path, but received an object that is not path-like (for example an int or a bool)
> - *NotADirectoryError* if it expected a path to a directory, but received path to something else
> - *NotADirectoryError* if it expected a path to an existing directory, but received a path to a non-existing one
> - *TypeError* if it expected an integer as the `max_files` parameter, but received an another type
> - *ValueError* if it received an integer less that one for the `max_files` parameter.

Here is an example of one way to display the tree, assuming input path points to a directory named `root_dir` and `max_files = 2` here:

```
root_dir/
    - sub_dir_1
        - file_1.txt
        - file_2.csv
        - ...
    - sub_dir_2
        - sub_sub_dir_1
            - file_1.txt
            - file_2.txt
        - file_1.csv
        - file_2.md
        - ...
    - sub_dir_3
        - sub_sub_dir_1
        - sub_sub_dir_2
        - sub_sub_dir_3
```

Here, `sub_dir_1` and `sub_dir_2` contains more files that `max_files`, but `sub_sub_dir_1` does not. Note, the `max_files` variable should only limit the output of files, not sub-directories. Feel free to experiment with different formatting of the output.

Now, take your time to familiarize yourself with the contents of `pollution_data` with your tools:

Inside `analyze_pollution_data.py`, there is a function named `analyze_pollution_data`, that takes a path to the working directory containing the `pollution_data` directory, which in

Skip to main content

should just make a call to `get_diagnostics` and `display_diagnostics` (and optionally `display_directory_tree`). Take your time to familiarize yourself with the contents of `pollution_data` from the information written to the terminal.

Since `analyze_pollution_data.py` is a script, you should write the following at the end of it to run `analyze_pollution_data`:

```
if __name__ == "__main__":
    work_dir =  # your working directory path
    analyze_pollution_data(work_dir)
```

Well done so far! Now as you have a good picture of what you have to deal with, it is time to venture into the restructuring of the data.

Inside `analyze_pollution_data.py`, you are going to traverse the `pollution_data` directory and look for original `.csv` files. Remember, the original data files follow this pattern, as described in the introduction:

- The original files all have a `.csv` suffix
- Their names follow `[gas_formula].csv` pattern, where `[gas_formula]` is replaced by one of the elements in `['CO2', 'CH4', 'N2O', 'SF6', 'H2']` list. For instance, `CO2.csv` is an original file.

Whenever one such file is found, you are going to copy it to a subdirectory specific for the corresponding gas under `assignment2/pollution_data_restructured/by_gas`, which you must create yourself. In this location, the file must be stored under a new name that is indicative of the source the data originated from. Whenever one such file is found, you are going to copy it to a subdirectory specific for the corresponding gas under `assignment2/pollution_data_restructured/by_gas`, which you must create yourself. In this location, the file must be stored under a new name that is indicative of the source the data originated from.

Assume the example configuration in *Figure 1*. The result in `pollution_data_restructured` after the transformation will be:

Figure 2: Example configuration in Figure 1 after restructuring of data

```
pollution_data_restructured
    - by_gas
        - gas_H2
            - src_agriculture_H2.csv
        - gas_CO2
```

Skip to main content

```
        - gas_CH4
            - src_oil_and_gass_CH4.csv
```

note: this is an *example* and a subset. Your exact list of sources and gasses will vary.

The `pollution_data_restructured` will be on the same level as `pollution_data`, i.e. under `assignment2`. Compare this result to the contents of example configuration. There, the original files inside `pollution_data/by_src` contained statistics for gasses `[H2, CO2, CH4]`, which gave rise to three corresponding sub-directories in the new directory. Also, note how the new name of the files is a merge of the original parent directory (the source) and the filename of the original gas `.csv` file (`[gas_formula].csv`).

In this task, you will gradually develop functions that perform specific operations needed to achieve this structure. You will be working in `analytic_tools/utilities.py` and `tests/test_utilities.py` only.

Given a file, check whether it is an original data file

Firstly, create a function that checks whether a given file is an original gas data file, i.e. that it follows the pattern described above.

> **Task 2.1**
>
> **(1 point)** Inside `analytic_tools/utilities.py`, complete the function named `is_gas_csv` that takes a path to a `.csv` file as an input, and returns a boolean of whether the name of the file satisfies the `[gas_formula].csv` pattern, where `[gas_formula]` is replaced by one of the elements in `['CO2', 'CH4', 'N2O', 'SF6', 'H2']` list. `is_gas_csv()` **should not need to check if the path exists or not**.
>
> **(0.5 points)** Implement error-handling in `is_gas_csv`. The function should raise:
>
> - *TypeError* if it expected a path, but received an object that is not path-like (for example an int or a bool)
> - *ValueError* if it expected a path to a `.csv` file, but received a path to something else
>
> `is_gas_csv` should *not* require that the files exists.

Skip to main content

Hint: Take a look at pathlib documentation, there exists many properties and methods on `Path()` objects that let you achieve this result in only a few lines.

> **Task 2.2**
>
> **(0.5 points)** Inside `tests/test_utilities.py`, complete the function named `test_is_gas_csv` and test `is_gas_csv`. Since this test-function does not have an argument, you are not in a temporary directory. You might create your own paths to pass as arguments to `is_gas_csv`, or point to some real locations of your choice inside your working directory. You should pretend that you do not know the implementation of this function and pass several paths with different syntax, combinations of lowercase/uppercase and even non-alphanumerical characters. Then you should do sanity checks on the return values (are they of correct type?) and assert their validity. **Note: there does not need to be an existing file at the path to test this function**.
>
> **(0.5 points)** Inside `tests/test_utilities.py`, complete the function named `test_is_gas_csv_exceptions`, and test the error handling of `is_gas_csv`. `test_is_gas_csv_exceptions` takes two arguments: `exception` and `path`, which are used in the `pytest.mark.parametrize` framework. Add more tests to the parametrization clause for sufficient testing.

Test your implementation by typing:

```
pytest -v -m task22 tests
```

Given an original file, assign a correct new destination

When we find a original gas data file, we must assign to it a correct destination in the new `pollution_data_restructured/by_gas` directory.

> **Task 2.3**
>
> **(1 point)** Inside `analytic_tools/utilities.py`, complete the function named `get_dest_dir_from_csv_file`, that takes two arguments:
>
> - `dest_parent` - a path to the parent directory of the destination directory, i.e. path

Skip to main content

- `file_path` - a path to the original gas .csv file we want to copy to the new destination.

The function must return the complete, absolute path of type *pathlib.Path* pointing to the `dest_parent/gas_[gas_formula]` directory where the file will be stored. If you are not working with pathlib, you may do the conversion at the return line like this: `return Path(path)`

`get_dest_dir_from_csv_file` should:

- Derive the correct name for the destination directory based on the gas in question, following `gas_[gas_formula]` pattern as shown in *Figure 2*.
- Check if such a directory already exists. If it exists, it should return the absolute path to this directory. And if not, it should create one under the parent directory pointed to by `dest_parent`, and then return the absolute path afterwards.

(0.5 points) Implement error handling in `get_dest_dir_from_csv_file`. The function should raise:

- *TypeError* if it expected a path, but received an object that is not path-like (for example an int or a bool)
- *ValueError* if it expected `file_path` to be a path to a file, but receive a path to something else
- *ValueError* if it expected `file_path` to point to an original gas .csv file, but did not receive that
- *NotADirectoryError* if it expected `dest_parent` to point to an existing directory, but received a path to a non-existing one or a path to something else

Task 2.4

(0.5 points) Inside `tests/test_utilities.py`, complete the function `test_get_dest_dir_from_csv_file` and test `get_dest_dir_from_csv_file`. This test-function takes `example_config` as argument, meaning that your are working in a temporary directory containing `pollution_data` as in *Figure 1*. You can use this to check if you get the three `gas_[gas_formula]` directories that you expect as shown in *Figure 2*. You must check that the function returns the correct path for each directory and that it *creates* the corresponding directory inside the temporary directory tree.

(0.5 points) Inside `tests/test_utilities.py`, complete the function named

Skip to main content

`get_dest_dir_from_csv_file`. `test_get_dest_dir_from_csv_file_exceptions` takes three arguments: `exception`, `dest_parent` and `file_path`, which are used in the `pytest.mark.parametrize` framework. Add more tests to the parametrization clause for sufficient testing.

Test your implementation by typing:

```
pytest -v -m task24 tests
```

Given a valid file, derive its new filename in `gas_[gas_formula]` directory

Now that you have a way to derive the new destination directory name, you need a function that can derive the new file name, following the pattern `src_[source type]_[gas_formula].csv` as shown in *Figure 2*. One way to do this, is to merge the name of the valid `.csv` file and its parent directory in `pollution_data/by_src`.

Task 2.5

(1 point) Inside `analytic_tools/utilities.py`, complete the function named `merge_parent_and_basename` that takes a path as an input and merges the basename of the path, with the first parent name in the path structure. The separating character, like `/` (mac) or `\` (windows), must be replaced by a `_` character. The function must return the resulting merged basename, as a *string*.

`merge_parent_and_basename` should *not* require that the path is a file, nor that it exists.

(0.5 points) Implement the error handling in `merge_parent_and_basename`. The function should raise:

- *TypeError* if it expected a path, but received an object that is not path-like (for example an int or a bool)
- *ValueError* if it expected a path with a filename *and* a parent-name, but did not receive that.

Hint: `os.sep` is the separator on all platforms, which might come to use here.

Skip to main content

Examples:

```
input: '/User/.../assignment2/pollution_data/by_src/src_agriculture/CO2.csv'
output: 'src_agriculture_CO2.csv'

input: 'some_dir/some_sub_dir'
output: 'some_dir_some_sub_dir'

input: 'some_dir/some_file.txt'
output: 'some_dir_some_file.txt'

input: 'some_file.txt'
output: Raise ValueError (missing parent directory)
```

> **Task 2.6**
>
> **(0.5 points)** Inside `tests/test_utilities.py`, complete the function named `test_merge_parent_and_basename` and test `merge_parent_and_basename`. Since this test-function does not have an argument, you are not in a temporary directory. You might create your own paths to pass as arguments to `merge_parent_and_basename`, or point to some real locations of your choice inside your working directory.
>
> **(0.5 points)** Inside `tests/test_utilities.py`, complete the function named `test_merge_parent_and_basename_exceptions`, and test the error handling of `merge_parent_and_basename`. `test_merge_parent_and_basename_exceptions` takes two arguments: `exception` and `path`, which are used in the `pytest.mark.parametrize` framework. Add more tests to the parametrization clause for sufficient testing.

Test your implementation by typing:

```
pytest -v -m task26 tests
```

Your `analytic_tools/utilities.py` module should now have at least these functions:

- `get_diagnostics`
- `display_diagnostics`
- `is_gas_csv`
- `get_dest_dir_from_csv_file`
- `merge_parent_and_basename`

And your `tests/test_utilities.py` script should have proper tests for all these functions

Skip to main content

terminal output).
If you have come this far you might give yourself a big hug, since you have done a lot of important preparations. Now you have a toolbox, designed and tested to help you automate your task, and you are just a few lines of code away from being able to let the magic of automation happen!

In this part, you will be working in `analyze_pollution_data.py` script, which will make extensive use of your `analytic_tools/utilities.py` module and functions inside `analytic_tools/plotting.py` module, so make sure you import both of them correctly.

Also, you will be making modifications your `assignment2` directory, creating new directories. As it often goes with programming, the things do not necessarily work out well for the first time, and you might want to restart a few times. Make sure to commit your progress at significant checkpoints.

Given the messy directory, copy the original data in a new, restructured directory

The first thing you want to do is to orchestrate the restructuring of `pollution_data`, such that you achieve the results similar to *Figure 2*.

You will now be working inside `analyze_pollution_data.py`. We have created a test script for you `tests/test_analyze_pollution_data` that you can run to test your implementation of the tasks in this section. The script calls the functions in a temporary directory containing the same version of pollution_data as you get, and checks if the expected changes have been made. It only checks if the names and structures correspond to the given requirements, so it does not reveal if all of the gasses have been restructured correctly, neither if the figures make sense. It is a part of your task to consider the validity of the outcome.

> **Task 3.1**
>
> **(1.5 points)** Inside `analyze_pollution_data.py`, complete the function named `restructure_pollution_data`, that takes two arguments:
>
> - `pollution_dir` - the absolute path to pollution_data directory
> - `dest_dir` - the absolute path to new directory where gas-specific subdirectories will be created. This will be `pollution_data_restructured/by_gas` in your case
>
> `restructure_pollution_data` should:

Skip to main content

1. Iterate through the contents of `pollution_dir`
2. Find original files (`[gas_formula].csv` files).
3. Create/assign new directory to store them under `dest_dir` using `get_dest_dir_from_csv_file`
4. Assign a new name using `merge_parent_and_basename` and copy the file to the new destination. If the file happens already to exist there, it should be overwritten.

This function does not have a return value, but your working directory will be modified after its execution (the `pollution_data_restructured/by_gas` subdirectories and their contents will be created).

(0.5 points) Implement error handling in `restructure_pollution_data`. The function should raise:

- *TypeError* if it expected a path, but received an object that is not path-like (for example an int or a bool)
- *NotADirectoryError* if it expected paths to *existing* directories for `pollution_dir` and `dest_dir` parameters, but received non-existing ones or paths that do not point to directories.

You can test your implementation by running:

```
pytest -v -m task31 tests
```

Given the sorted pollution data, plot the statistics

Now, you are ready to complete `analyze_pollution_data` function inside `analyze_pollution_data.py`.

Task 3.2

(1.5 points) Inside `analyze_pollution_data.py`, complete the function named `analyze_pollution_data`. This function should:

Skip to main content

1. Make a call to `display_diagnostics` (and `display_directory_tree` if it is implemented) with `pollution_data` directory as argument, which will display the output to the terminal
2. Create `pollution_data_restructures/by_gas` directory
3. Make a call to `restructure_pollution_data`
4. Create the `pollution_data_restructures/figures` directory
5. Make a call to `plot_pollution_data` from `analytic_tools/plotting.py`, passing the `pollution_data_restructured/by_gas` directory as argument, to create and save the figures for all of the `gas_[gas_formula]` subdirectories.

(0.5 points) Implement the error handling of `analyze_pollution_data`. The function should raise:

- *TypeError* if it expected a path, but received an object that is not path-like (for example an int or a bool)
- *NotADirectoryError* if it expected a path to an *existing* directory as `work_dir`, but received a non-existing one or a path that does not point to a directory.

You can test your implementation by running:

```
pytest -v -m task32 tests
```

If the test passes, you should call this function at the bottom of the `analyze_pollution_data.py` and run the whole script afterwards. If everything goes well, you should now have created and populated the `pollution_data_restructured/figures` directory with statistics of the gasses present. Inspect the figures.

You should ask yourself these questions:

- Do they seem reasonable, based on your prior knowledge of greenhouse gas pollution and sources?
- Is the trend increasing or decreasing for the various gasses? Does that correspond to your expectations?
- Does something seem surprising about the data?

Optional task (0 points) It is probable that the first runs of `analyze_pollution_data.py` will not yield the correct result. Having created new

Skip to main content

> `analytic_tools/utilities.py`, create a function named `delete_directories` that takes a list of paths to directories, and removes them in a proper way. To not risk deleting something important, the function should display all of the directories it is going to delete, and prompt the user for confirmation before execution. *None*: there are no points granted for this task, but it will be convenient to implement this if you run `analyze_pollution_data.py` script several times.

You are free to choose the implementation of this function. You can either make a call to it at the end of `analyze_pollution_data` function, or at the end of the `if __name__ == '__main__'` clause:

```
if __name__ == "__main__":
    work_dir =  # your working directory path
    analyze_pollution_data(work_dir)
    utilities.delete_directories([work_dir / "pollution_data_restructured"]) # opti
```

You can of course also delete everything manually, by clicking in the file structure, but now as you are all into automation, why not getting it going?

Sometimes, you might not want to keep intermediate files in your working directory, and only save the final result. Why not try doing the restructuring of the data in a temporary directory and only save the figures in your working directory?

> **Task 3.3 (Bonus task 2)**
>
> **(0.5 bonus points)** Inside `analyze_pollution_data.py`, complete the function named `analyze_pollution_data_tmp` that has exactly the same logic as `analyze_pollution_data`, except that the restructuring is done in a temporary directory and only the figures are saved in your `assignment2` directory. The function should create a real directory named `figures` in your `assignment2` directory, containing the `.png` files produced for the different gasses, and that must be the only change done to `assignment2`.
>
> **(0.5 bonus points)** Implement the error handling of `analyze_pollution_data_tmp`. The function should raise:
>
> - *TypeError* if it expected a path, but received an object that is not path-like (for example an int or a bool)

Skip to main content

> - *NotADirectoryError* if it expected path to an *existing* directory as `work_dir`, but received a non-existing one or a path that does not point to a directory.

You can test your implementation by running:

```
pytest -v -m task33 tests
```

If you want this function to execute when running `analyze_pollution_data.py`, you might change the end of the script to:

```
if __name__ == "__main__":
    work_dir =  # your working directory path
    analyze_pollution_data_tmp(work_dir)
    utilities.delete_directories([work_dir / "figures"]) # optional
```

Before handing in the assignment, make sure that your `assignment2` directory contains the following:

- analytic_tools
- pollution_data
- pollution_data_restructured
  - by_gas
  - figures
- tests
- analyze_pollution_data
- pyproject.toml
- README.md

It is explicitly shown that figures must be a sub-directory of `pollution_data_restructured`, i.e. even if you did *Bonus Task 2*, you must submit as if you have executed `analyze_pollution_data` and not `analyze_pollution_data_tmp` in `analyze_pollution_data.py` script.

To check this automatically, run `tests/test_handin.py` script with pytest before submitting. Note: the script does not check if "by_gas" and "figures" have the correct content (which is the solution of the assignment), only if they exist and are non-empty.

If you are not familiar working with paths and pathlib, then this section will provide you with a short intro. It is meant to set your on the right track of thinking, for further elaboration you

Skip to main content

You can think of paths as the bridge from your program to the environment it exists in. Accessing files, directories, other programs is all done by referring to them via their paths. A path is either a complete or a relative description of where a certain object is located at your computer.

A complete description is called **an absolute path**. Assume you are writing a script called "cool_script.py", which is located in a directory you have called `my_scripts`, which is in your local `Documents` directory on your computer. An absolute path to your script will be:

```
Windows: C:\Documents\my_scripts\cool_script.py
Mac: /Users/karinordmann/Documents/my_scripts/cool_script.py
```

A relative description is always referred to some place in your computer, and is called a **relative path**. A relative path to your script will be the following, relative to two different locations:

```
Relative to `my_scripts` folder:
Windows: cool_script.py
Mac: cool_script.py

Relative to `Documents` folder:
Windows: my_scripts\cool_script.py
Mac: my_scripts/cool_script.py
```

If you are in your `Dashboard` folder, and you want to refer to the script from this location, you may write in relative terms:

```
Windows: ..\Documents\cool_script.py
Mac: ../Documents/cool_script.py
```

Where `..` stands for `the parent folder to where I am currently at`, assuming that `Documents` and `Dashboard` are both on the same level in your directory tree, i.e. under `/User` (mac) or `C:` (windows).

Even though relative paths may seem more neat, it is recommended to work with absolute paths as much as possible. A relative path is always resolved *relative to the current working directory*, and will point to a different location depending on where the program is being run. The working directory can also change during the life of a single program, for example if it executes in a temporary directory, which will change the meaning of the relative paths.

Skip to main content

Since absolute paths may be long chunks of plain text, it is strongly encouraged to use path-objects to encapsulate them. It makes your code more flexible and improves readability. For instance, if your are working on `cool_script.py` and want to save the actual directory containing the script on your computer, which in this case is `Documents`, as a variable, you may write this using pathlib:

```
documents_dir = Path(__file__).parent.absolute()
```

`Path(__file__)` refers to the location of your script on the computer, `.parent` is a property that refers to the parent directory and `.absolute()` is a method that converts the whole thing to an absolute path. You could, alternatively, also name the variable "script_dir" or "here", whatever that works best for you to indicate that this is the actual directory holding the script. Note, this must be done before entering any temporary directory, since afterwards, `Path(__file__).parent.absolute()` will be resolved relative to the temporary location.

For pathlib.Path objects, the `/` operator is overloaded to provide an intuitive way of joining paths. Assume that `Documents` directory from above also has a subdirectory named `data` which contains a `coordinates.txt` file. You want to point to this file in `cool_script.py`. Since `documents_dir` is now a pathlib.Path object, you can simply write:

```
file = documents_dir / "data" / "coordinates.txt"
```

Which will yield the absolute path to this file.

If the `data` directory did not exist on beforehand, and you would like to create it during execution of `cool_script.py`, you can write:

```
(documents_dir / "data").mkdir(exist_ok=True)
```

Here, `exist_ok=True` makes sure to overwrite this directory if it already exists. Remove it if you do not want that to happen.

Now, assume you have acquired some data during the execution of `cool_script.py` and you want to save it in a file named `"coordinates.txt"`. pathlib enables you to create a file like this:

```
(documents_dir / "data" / "coordinates.txt").touch()
```

Skip to main content

This will create an empty file. If you want to create a specific directory tree all at once you can do:

```
(documents_dir / "data" / "weather").mkdir(parents=True, exist_ok=True)
```

The `parents=True` argument makes sure to create and overwrite both “weather” and “data” directories if they do not already exist.

Take a moment to appreciate how neatly the syntax of pathlib conveys the different OS-operations you want to perform. Imagine how much more lines of code would be needed if we would write everything as strings and join them manually.

Previous
Assignment 1: init with git

Next
Assignment 3: Python for the gram