A diagram of a process

Description automatically generated

The above flowchart outlines a systematic approach used in optimizing the problem.

1. Data Acquisition: The process begins by collection of relevant datasets containing raw information about the problem at hand, such as purchase data, production metrics, or shipping details i.e purchase\_vars, production\_vars, crude\_shipping\_vars.

2. Identifying Variables: Next, we pinpoint the key factors, known as decision variables, that we can control within the problem's boundaries to achieve the best possible outcome. These variables are typically informed by the data we've gathered.

3. Objective and Constraints: We then evaluate a mathematical expression, called the objective function, which defines what we're trying to maximize or minimize. Alongside this, we consider any constraints that limit our options, like budgetary constraints or production capacity.

4. Optimality Check: We assess whether our current solution is the best one possible, given the defined objective and constraints. If it's not optimal, we move to the next step.

5. Adjusting Variables: Here, we tweak our decision variables based on various optimization techniques. This might involve methods like adjusting production levels, changing shipment routes, or altering resource allocations.

6. Iteration: The process of evaluating, adjusting, and reassessing continues in a loop until we reach an optimal solution.

7. Achieving Optimal Solution: Finally, we stop when we've achieved the best possible outcome according to our objective function, within the constraints we've set.

This methodical approach ensures that businesses make informed decisions to maximize efficiency, minimize costs, and optimize their operations for better overall performance.

Model 2

A screen shot of a computer

Description automatically generated

Importing Libraries

1. import cplex: This imports the CPLEX optimization library into the Python script, allowing access to its functionality for solving linear programming (LP), mixed-integer programming (MIP), and quadratic programming (QP) problems.

2. import sys: This imports the “sys” module, which provides access to some variables used or maintained by the Python interpreter and to functions that interact strongly with the interpreter.

3. import itertools: This imports the “itertools” module, which provides various functions that work on iterators to produce complex iterators.

4. from cplex.exceptions import CplexError: This imports the “CplexError” exception class from the “cplex.exceptions” module. This exception class is used to handle errors specific to the CPLEX library.

5. from cplex import SparsePair: This imports the “SparsePair” class from the “cplex” module. A “SparsePair” object represents a sparse vector, which can be used in various places within the CPLEX library, such as when specifying constraints or objectives.

6. from cplex.exceptions import CplexError, CplexSolverError: This imports both “CplexError” and “CplexSolverError” exception classes from the “cplex.exceptions” module. “CplexError” is used to handle general errors raised by the CPLEX library, while “CplexSolverError” specifically handles errors related to solving optimization problems using CPLEX.

A grey background with black text

Description automatically generated

We are creating a dictionary named crude\_oil\_costs\_per\_barrel which contains information about costs of different types of crude oil per barrel in key value format within the dictionary.

A computer code with numbers and letters

Description automatically generated with medium confidence

This code snippet defines a nested dictionary named **tanker\_rates**, which contains information about the rates of different types of tankers for transporting crude oil. We have used key value pair for all the classes.

A computer code with numbers and symbols

Description automatically generated

The above code we are making nested dictionary named “tankers\_capacity\_in\_barrels” consisting of capacity of barrels for each tanker.

A group of numbers on a white background

Description automatically generated

The above snippet is about dictionary port\_charges which consists of all the port charges.

A group of blue dots

Description automatically generated

This snippet explains the cost per hour for all the classes.

A close-up of a text

Description automatically generated

We made nested dictionary named shipping\_times\_in\_days which contains key value of all the ports and travel times.

A screenshot of a computer

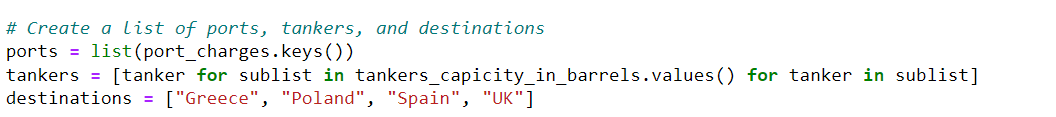
Description automatically generated

These dictionaries facilitate easy lookup of the port and tanker type associated with a specific type of crude oil. They provide a convenient way to organize and access this mapping information within the code.

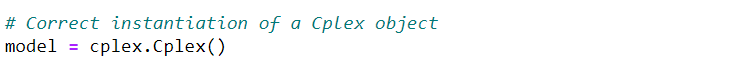
A close-up of a number

Description automatically generated

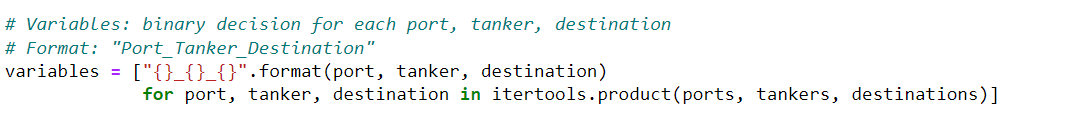
This code snippet creates a new dictionary ‘tanker\_type\_to\_crude’ by inverting the ‘crude\_to\_tanker\_type’ dictionary and defines a constant ‘QUANTITY\_PER\_TYPE\_PER\_DESTINATION’ representing the quantity of barrels per type per destination.



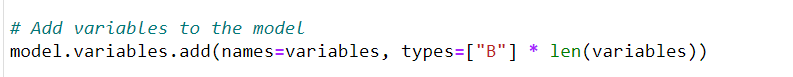
Using above code we are creating list, these lists are created to provide easy access to the names of ports, tankers, and destinations, which may be useful for various calculations or operations within the code. Firstly, it retrieves the names of ports by extracting the keys from the dictionary “port\_charges” using the “keys()” method and converting them into a list. Secondly, it compiles a list of tankers by iterating over the values of the dictionary “tankers\_capicity\_in\_barrels” and appending each tanker to the list. Lastly, it directly creates a list called “destinations”, comprising the names of four predetermined destinations: Greece, Poland, Spain, and the UK. These lists are intended to facilitate easy access to relevant information for subsequent calculations or operations within the code.



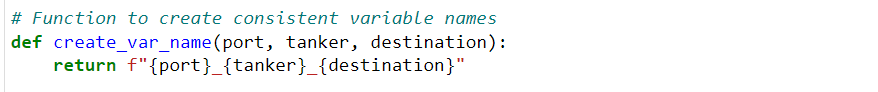
This line of code correctly instantiates a Cplex object named “model”. The “cplex.Cplex()” function call creates a new instance of a Cplex object, which can then be used to define and solve optimization problems using the CPLEX optimization library in Python. However, typically you would specify additional parameters or options when creating the Cplex object to customize its behavior according to your specific optimization problem.



This code snippet utilizes list comprehension and the itertools module to construct a list of variable names representing binary decision variables for every conceivable combination of port, tanker, and destination. It iterates over each combination of elements from the lists of ports, tankers, and destinations, formatting a string representation for each combination. These formatted strings, representing variable names in the format "Port\_Tanker\_Destination", are then appended to the list of variables. This approach provides a concise and efficient way to generate the necessary variables for formulating optimization problems involving the transportation of crude oil.



This code snippet enhances the previously instantiated Cplex model by introducing variables into it. It employs the “variables.add()” method to achieve this. Specifically, the “names” parameter specifies the variable names derived earlier, while the “types” parameter assigns binary type ("B") to each variable, indicating they are binary decision variables. The multiplication “\* len(variables)” ensures that each variable is matched with its corresponding type. Essentially, this line of code ensures the incorporation of essential binary decision variables into the model, a pivotal step in formulating and solving optimization problems using the CPLEX optimization library.

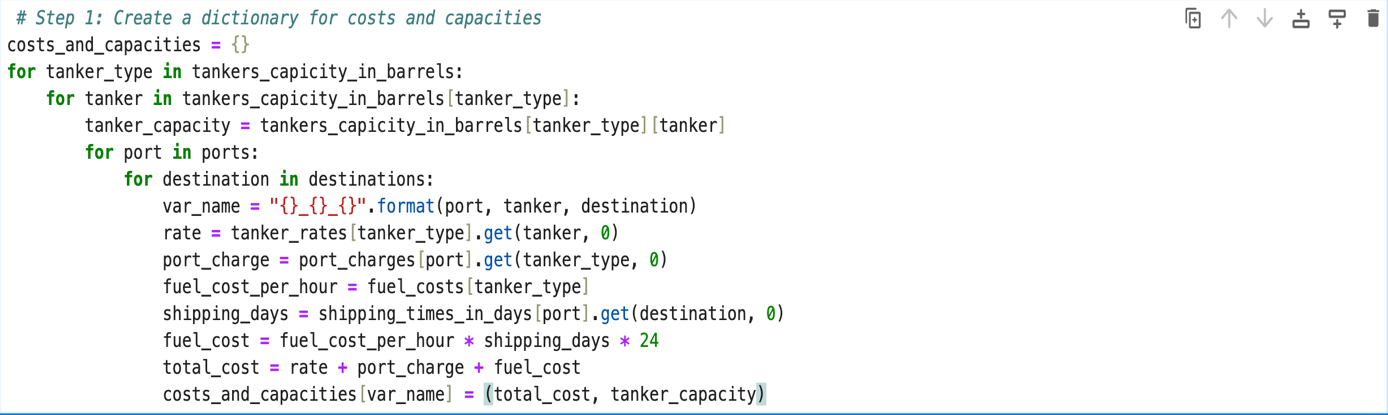


This function, named “create\_var\_name”, is designed to produce uniform variable names using input parameters representing a combination of port, tanker, and destination. The function accepts three parameters: “port”, “tanker”, and “destination”, which correspond to the components necessary for forming a variable name. Inside the function, an f-string is utilized to concatenate these parameters with underscores (\_) as separators, resulting in a consistent variable name format. This function offers a concise and reusable approach to generating variable names, ensuring consistency across the naming convention, particularly beneficial when handling extensive sets of variables within optimization problems.

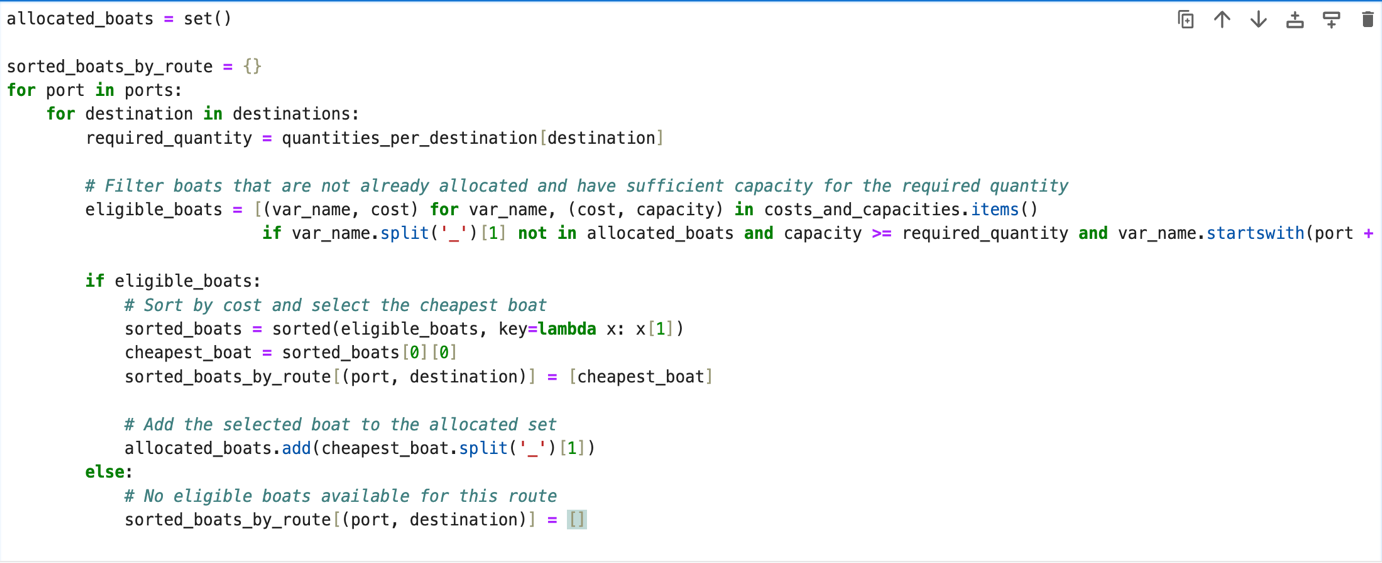
A computer code with text

Description automatically generated with medium confidence

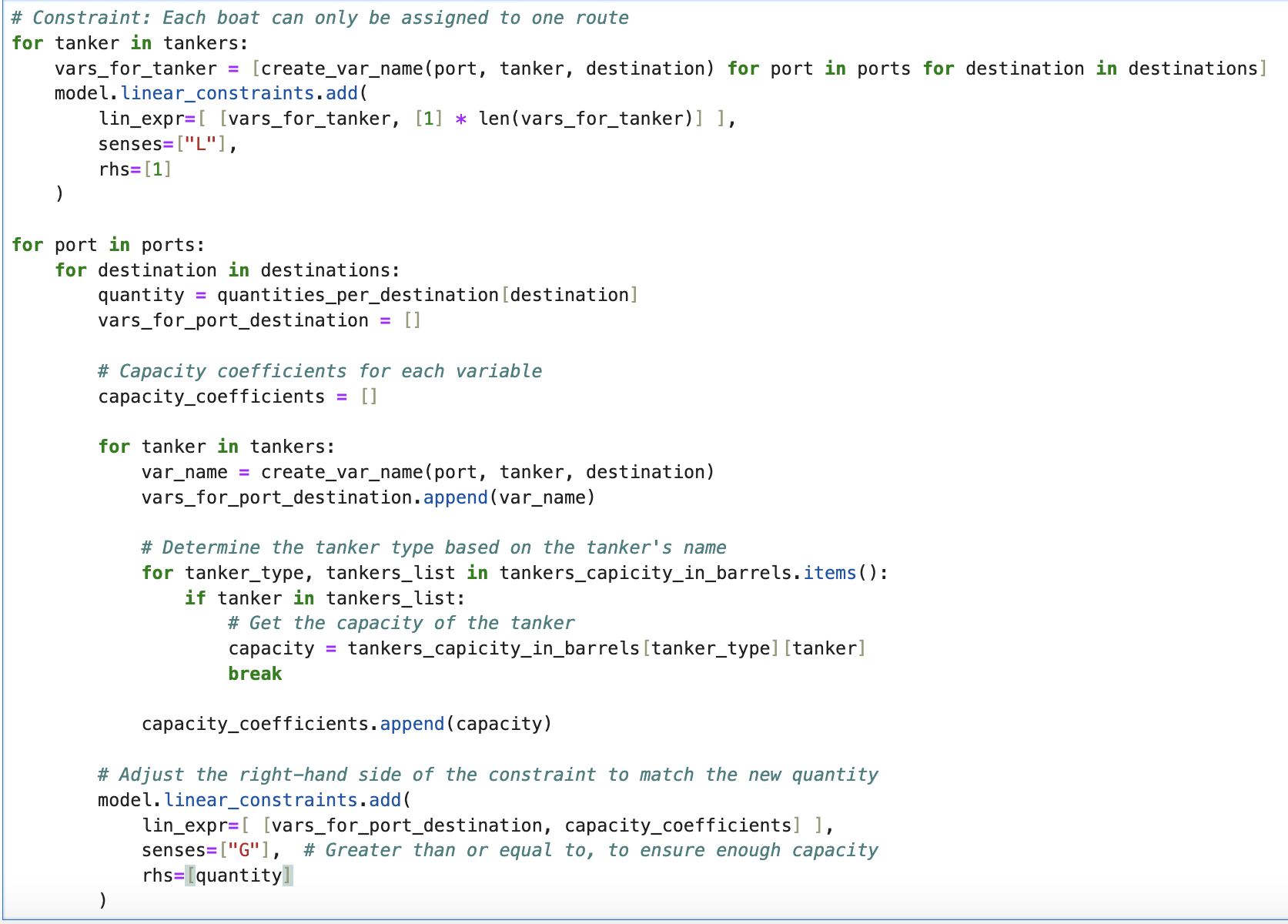
This code constructs the objective function for an optimization model by iterating over combinations of port, tanker, and destination. It initializes an empty list named “objective” to store tuples containing variable names and their corresponding total costs. It iterates over tanker types and, for each type, retrieves associated crude types and ports. Then, it calculates the total cost for each combination of port, tanker, and destination, considering factors like tanker rates, port charges, fuel costs, and crude oil costs. These total costs are appended to the “objective” list. Finally, the code sets the objective function sense to minimize and defines the linear part of the objective function using the list of tuples stored in “objective”. Overall, this adjustment ensures that the optimization model aims to minimize the total cost associated with transporting crude oil to different destinations using various tankers and ports.



This code computes the costs and capacities associated with transporting fuel using different types of tankers from various ports to diverse destinations. It begins by initializing an empty dictionary called “costs\_and\_capacities”, which will store the calculated total costs and tanker capacities for each unique combination of port, tanker, and destination. Through nested loops, it iterates over each tanker type and then each specific tanker within that type, along with all available ports and destinations. Within these iterations, it dynamically generates a variable name representing the combination of port, tanker, and destination. The code then calculates the total cost of transportation considering factors such as the tanker's rate, port charges, fuel costs, and shipping times. Finally, it stores the computed total cost and tanker capacity in the “costs\_and\_capacities” dictionary using the generated variable name as the key. This systematic approach enables efficient computation and storage of essential data for fuel transportation logistics.



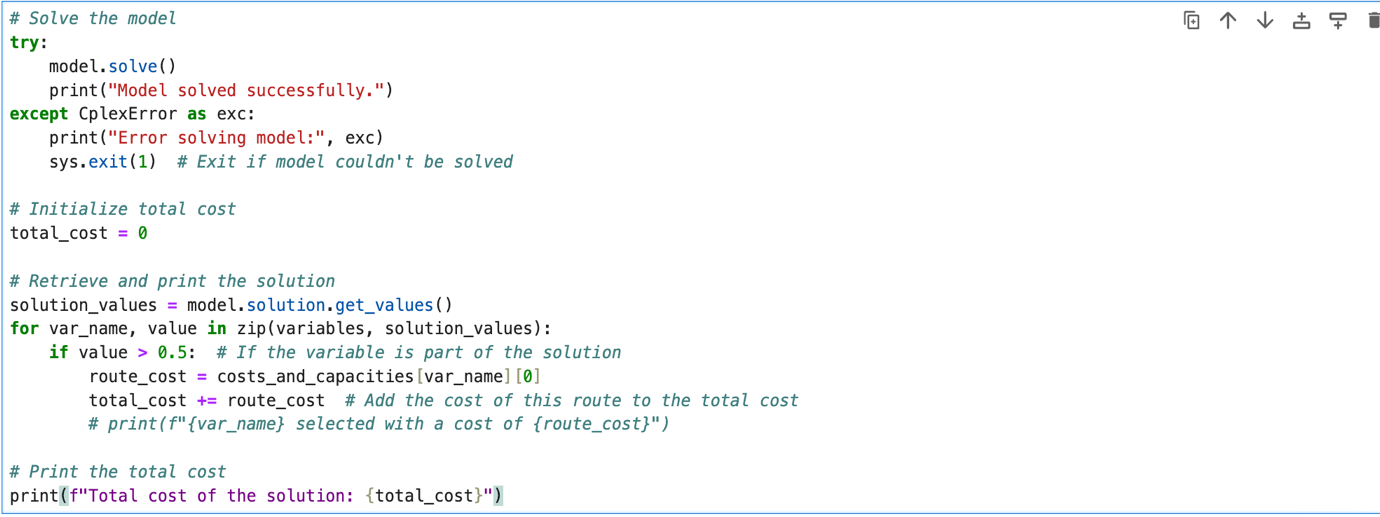
This section of the code is responsible for the allocation of boats to transportation routes based on considerations such as cost and capacity. It begins by initializing two data structures: a set named “allocated\_boats”, which tracks boats that have already been assigned to routes, and a dictionary called “sorted\_boats\_by\_route”, which will contain sorted lists of boats for each route. Through nested loops iterating over the available ports and destinations, the code determines the quantity of fuel required for each destination. It then filters out boats that are eligible for allocation, considering criteria such as whether the boat has already been assigned (not in allocated\_boats), if it has the necessary capacity (capacity >= required\_quantity), and if it is available at the current port. Eligible boats are sorted by cost, and the cheapest boat is selected for each route. This information is stored in the “sorted\_boats\_by\_route” dictionary. Additionally, the selected boat is added to the “allocated\_boats” set to mark it as allocated for future reference. If no eligible boats are found for a particular route, an empty list is assigned to indicate the unavailability of boats. This allocation process optimizes fuel transportation logistics by efficiently assigning boats to routes based on cost, capacity, and availability constraints.



This code segment operates within a broader optimization model and is tasked with setting up constraints related to tanker capacity. It iterates over each tanker and generates variable names for each combination of ports and destinations using the “create\_var\_name” function. These variable names represent the amount of fuel transported by each tanker from a particular port to a specific destination. Subsequently, the code adds linear constraints to the optimization model using the “linear\_constraints.add” method.

For each tanker, it adds a constraint ensuring that only one route is selected for transportation, indicating that each tanker can only be assigned to one route. This constraint is enforced by setting the sum of variables associated with each tanker to be less than or equal to 1.

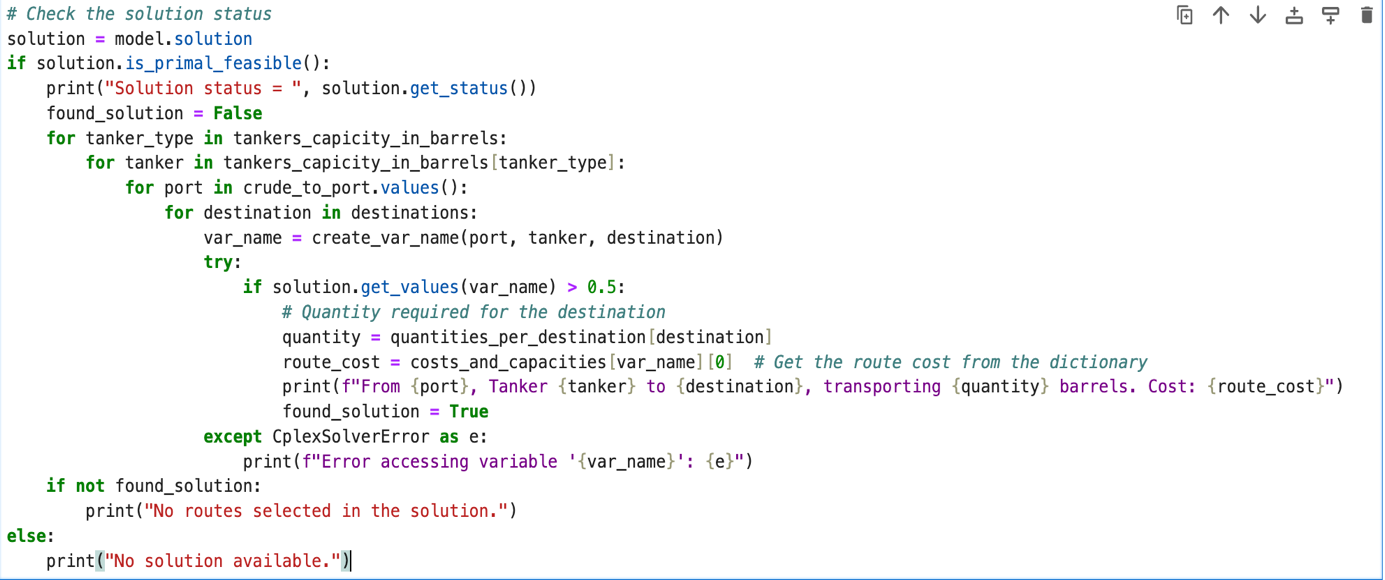
Next, for each combination of ports and destinations, the code calculates the required quantity of fuel based on the destination's demand. It then constructs constraints to ensure that the total capacity of tankers assigned to transport fuel from a particular port to a specific destination is sufficient to meet the demand. The capacities of the tankers are used as coefficients in these constraints to reflect the available capacity for each route. These constraints are added to the model with the sense set to "greater than or equal to," ensuring that the total capacity of assigned tankers is at least equal to the required quantity of fuel for each destination.



Following the optimization of the model, the code attempts to solve it using the CPLEX solver by calling “model.solve()”. Upon successful completion of the solving process, it prints a message confirming the successful resolution of the model. In case of any errors during the solving process, it catches the “CplexError”, prints an error message indicating the issue, and exits the program with a status code of 1.

After solving the model, the code initializes the variable “total\_cost” to track the cumulative cost of the solution. It then retrieves the solution values using “model.solution.get\_values()”. For each variable in the solution, if its value is greater than 0.5, indicating it is part of the solution, the code calculates the cost associated with the corresponding route and adds it to the “total\_cost”. Finally, it prints out the total cost of the solution.

This segment of the code effectively handles the solving of the optimization model, retrieves the solution, calculates the total cost, and prints it out for analysis and further action.



This section of the code evaluates the feasibility of the obtained solution and provides information about the selected routes if a feasible solution is found. It begins by checking if the solution is feasible using “solution.is\_primal\_feasible()”. If the solution is feasible, it prints the status of the solution obtained from “solution.get\_status()”. Then, it iterates over various combinations of tankers, ports, and destinations to identify the selected routes.

For each combination, it constructs the variable name representing the route using “create\_var\_name” and attempts to retrieve the value of the variable from the solution. If the value is greater than 0.5, indicating that the route is part of the solution, it retrieves the required quantity of fuel for the destination and the cost of the route from the “quantities\_per\_destination” and “costs\_and\_capacities” dictionaries, respectively. It then prints information about the selected route, including the departure port, tanker type, destination, quantity of fuel transported, and the associated cost.

In case of any errors encountered during the retrieval of variable values, it catches the “CplexSolverError” and prints an error message. If no solution is found or no routes are selected in the solution, appropriate messages are printed to indicate the absence of a valid solution or the absence of selected routes.