Python Course 1: Introduction to Python

Patrick Munroe

GERAD

2025-10-15

Introduction

- **Objectives**:
  - Learn Python basics
  - Get an introduction to popular optimization and data science libraries
- **Prerequisites**: Familiarity with basic programming concepts (variables, data types, control structures, and functions) in any programming language
- **Pedagogical approach**: Learn fundamental Python concepts by solving an optimization problem

Plan

- Introduction
- Python fundamentals
- Data science and optimization libraries:
  - Manipulating data with `pandas`
  - Visualizing data with `Matplotlib`
  - Formulating and solving optimization models with `Pyomo`
- Additional topics (if time permits)

Creating a virtual environment

- Use a virtual environment to keep your project's dependencies isolated.
- To **create** the environment, in a terminal, navigate to your project folder and run:

```
python -m venv venv
```

- To **activate** the environment:
  - `venv\Scripts\activate` (Windows)
  - `source venv/bin/activate` (macOS/Linux)
- When activated, `(venv)` appears at the prompt.
- To **exit** the environment, type `deactivate`.

Getting started with Python

- Strings:

```
"Hello, world!"     # Double quotes
'Hello, world!'     # Single quotes
"That's fine!"      # Double quotes with a single quote inside
```

- Printing:

```
print("Hello, world!")
```

- Arithmetic:

```
2 + 3       # Addition: 5
3 ** 2      # Exponentiation: 9
```

- Comments:

```
# This is a comment
```

Transportation problem

A motivating example for learning Python

Goal: Distribute goods from suppliers to customers at minimal cost.

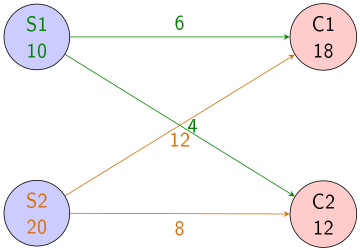
Given:

- Supplier capacities
- Customer demands
- Shipping costs

Constraints:

- Supply limits per supplier
- Demand requirements per customer

Objective: Minimize total transportation cost



Transportation problem

Data representation

Typical data for the transportation problem:

- **Suppliers:** List with supply amounts
- **Customers:** List with demand amounts
- **Cost Matrix:** Table of shipping costs per unit from each supplier to each customer

Example:

Suppliers

- S1: 10 units
- S2: 20 units

Customers

- C1: 18 units
- C2: 12 units

Cost Matrix

	C1	C2
S1	5	7
S2	4	6

Variables and data types

Concepts

- A **variable** is a name (identifier) that refers to an **object** (the value) stored in memory.
- Each object has a **data type**, which determines what kind of data it represents (e.g. integer, string).
- The **name** (the variable) and the **content** (the object/value) are separate:
  - **Variable:** The label or identifier (e.g. `x`)
  - **Object/Value:** The data assigned to the variable (e.g. `42` or `"hello"`)
  - **Data type:** The kind of the object/value (e.g. `int`, `str`)
- *In Python, every value is represented as an object.*

Variables and data types

Dynamic typing in Python

- **No type declaration** is needed when defining a variable.
- For example, you simply assign a value without: `x = 10` (no need to specify that it is an integer)
- **Variables can be reassigned** to values of different types:
  - `x = 10` # x is an integer (int)
  - `x = "hello"` # now x is a string (str)
- This is different from languages like Java or C++, where the type must be specified.

Variables and data types

Example

The input data from our transportation problem can be stored in variables:

```
# Suppliers
s1 = 10
s2 = 20

# Customers
c1 = 18
c2 = 12

# Cost matrix
s1_c1 = 5
s1_c2 = 7
s2_c1 = 4
s2_c2 = 6
```

Printing strings with variables

Examples

- You can print the value of a variable by passing it as an argument to the `print` function:

```
print(s1)                # Output: 10
```

- You can print multiple variables by passing them as separate arguments:

```
print(c1, c2)            # Output: 18 12
print(s1, c2, s1_c2)       # Output: 10 12 7
```

- You can include variable values inside a string using an `f`-string (formatted string literal):

```
print(f"s1: {s1}, s2: {s2}")   # Output: s1: 10, s2: 20
```

The type function

- Use the type function to determine the type of any object.
- It can be called directly on a value:

```
print(type(23.54))       # Output: <class 'float'>
print(type("Customer")) # Output: <class 'str'>
```

- Or on a variable:

```
s1 = 10
print(type(s1))          # Output: <class 'int'>
```

- The output is always in the form <class 'type'>.

Python basic data types

Examples:

- **int**: Integer numbers
  `x = 42`
- **float**: Decimal numbers
  `y = 3.14`
- **str**: Text strings
  `name = "test"`
- **bool**: Boolean values
  `flag = True`
- **list**: Ordered collection
  `nums = [1, 2, 3]`
- **tuple**: Immutable collection
  `coords = (10, 20)`
- **dict**: Key-value pairs
  `info = {"S1": 35, "S2": 25}`
- **set**: Unordered unique items
  `unique = {2, 4, 6}`

Remarks

- There are many other built-in data types.
- You can also define your own data types (using object-oriented programming).

Lists

Concept and syntax

- A **list** is a built-in Python data structure that stores an ordered collection of items.
- Lists can contain elements of any type (integers, strings, other lists, etc.).
- You can access elements in a list by their position, starting from 0.

General syntax:

```
my_list = [element1, element2, element3, ...]
```

Remark

Python lists can mix types, e.g. `mixed = [1, "hello", 3.14, [2, 3]]`

Lists

Access elements

- Access elements by index: `my_list[0]` (first element)
- Lists are 0-indexed (the first item is at position 0)
- Access the last element with `my_list[-1]`
- Access a range of elements (a **slice**) with `my_list[start:stop]` (from start up to but not including stop)

Example:

```
mixed = [1, "hello", 3.14, [2, 3]]

print(mixed[0])     # Output: 1       (first element)
print(mixed[-1])    # Output: [2, 3] (last element)
print(mixed[1:3])   # Output: ['hello', 3.14]
                    # (slice from index 1 up to 3)
```

Lists

Useful functions on lists

- `len(x)`: Returns the number of elements in the list `x`.
- `sorted(x)`: Returns a new sorted list with the same elements as `x` (the original list is unchanged).
- `sum(x)`: Returns the sum of all elements in the list (works with numbers).
- `min(x)`: Returns the smallest element in the list.
- `max(x)`: Returns the largest element in the list.

```
supply = [10, 20]
demand = [18, 12]

print(len(supply))          # Output: 2
print(sorted(demand))       # Output: [12, 18]
print(demand)               # Output: [18, 12]
print(sum(supply))          # Output: 30
print(min(demand))          # Output: 12
print(max(demand))          # Output: 18
```

Lists

Sequence of numbers with `range`

- The `range` function creates a sequence of numbers:
- `range(n)` generates a sequence of numbers from `0` to `n-1`.
- The sequence generated by `range` is not a list, but can be converted to one using `list()`:

```
print(range(10)) # Output: range(0, 10) NOT A LIST

print(list(range(10))) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **In general**, `range(start, stop, step)` generates a sequence:
  - starting from `start`;
  - incrementing by `step` (or decrementing if `step` < 0);
  - stopping before `stop`

```
print(list(range(4, 10, 2))) # [4, 6, 8]
print(list(range(11, 0, -3))) # [11, 8, 5, 2]
```

Methods

What is a method?

- A **method** is a function that is associated with a specific object (like a list or a string).
- You call a method using the "dot notation":

```
object.method(arguments)
```

- The method can use or change the contents of the object it belongs to.

Examples with strings:

```python
my_string = "hello world"
print(my_string.upper())        # Output: "HELLO WORLD"
print(my_string.replace("o", "0"))  # Output: "hell0 w0rld"
print(my_string.split())        # Output: ['hello', 'world']
```

Methods

Common list methods

- Lists in Python have several useful methods for modifying their contents:
  - `append(x)`: Adds element `x` to the end of the list.
  - `insert(i, x)`: Inserts element `x` at position `i`.
  - `pop([i])`: Removes and returns the element at position `i` (last element if `i` is omitted).
  - `remove(x)`: Removes the first occurrence of element `x`.
  - `extend(iterable)`: Adds all elements from another iterable to the end of the list.
  - `sort()`: Sorts the list in place.
  - `reverse()`: Reverses the elements of the list in place.
- Elements of a list can also be changed or accessed using the `[]` (index) operator.

Note

These methods modify the list directly (in place).

Methods

Examples of common list methods

```
lst = [10, 20, 30]

lst.append(40)          # [10, 20, 30, 40]
lst.insert(1, 15)       # [10, 15, 20, 30, 40]
lst.pop()               # [10, 15, 20, 30]
lst.remove(15)          # [10, 20, 30]
lst.extend([50, 60])    # [10, 20, 30, 50, 60]
lst.sort()              # [10, 20, 30, 50, 60]
lst.reverse()           # [60, 50, 30, 20, 10]
lst[0] = 99             # [99, 50, 30, 20, 10]
```

Mutability

Concept

- Objects in Python can be *mutable* (can be changed after creation) or *immutable* (cannot be changed after creation).
  - **Mutable objects:** `list`, `dict`, `set`
  - **Immutable objects:** `int`, `float`, `str`, `tuple`
- Mutable objects can have their contents changed after they are created, while immutable objects cannot be changed—any operation that seems to modify them actually creates a new object.

Mutability

Examples

```python
# Mutable example (list)
mixed = [1, "hello", 3.14, [2, 3]]
mixed[0] = 42
mixed.pop()
print(mixed)   # Output: [42, 'hello', 3.14]

# Immutable example (str)
s = "hello"
# s[0] = "H" # This would raise an error
s = "Hello" # A new string object is assigned to s

# Immutable example (int)
x = 10
x = x + 5 # A new int object is assigned to x
```

Dictionaries

Concept and syntax

- A **dictionary** is a built-in Python data structure that stores data as key-value pairs.
- Each **key** is unique and is used to access its corresponding **value**.
- Dictionaries are useful for representing mappings, such as supplier to supply amounts, or customer to demand amount.

General syntax:

```
my_dict = {key1: value1, key2: value2, ...}
```

- Access a value by its key: `my_dict[key1]`
- Keys can be of any **immutable** type (e.g., strings, numbers, tuples).

Dictionaries

Example: suppliers and customers

```
supply_by_sup = {"S1": 10, "S2": 20}
demand_by_cust = {"C1": 18, "C2": 12}

print(supply_by_sup["S2"])  # Output: 20
print(demand_by_cust["C1"])  # Output: 18
print(len(supply_by_sup))   # Output: 2
```

- In `{"S1": 10, "S2": 20}`, `"S1"` and `"S2"` are the keys, whereas `10` and `20` are the values.
- The `len` of a dictionary in Python corresponds to the number of key-value pairs it contains.

Dictionaries

Example: cost matrix

```
cost_by_sup_cust = {
        ("S1", "C1"): 5,
        ("S1", "C2"): 7,
        ("S2", "C1"): 4,
        ("S2", "C2"): 6
}

print(cost_by_sup_cust[("S1", "C2")])   # 7
print(cost_by_sup_cust[("S2", "C1")])   # 4
print(len(cost_by_sup_cust))            # 4
```

- A **tuple** (such as a pair `("S1", "C2")`) can be used as a dictionary key because it is **immutable**.
- A **list** cannot be used as a key, because lists are **mutable**.

Dictionaries

Common dictionary methods

- Dictionaries in Python have several useful methods for accessing and working with their contents:
  - `keys()`: Returns a view of all the keys in the dictionary.
  - `values()`: Returns a view of all the values in the dictionary.
  - `items()`: Returns a view of all key-value pairs as tuples.
  - `get(key, default)`: Returns the value for `key` if it exists; otherwise returns `default` (or `None` if `default` is not provided).
  - `pop(key)`: Removes the item with the specified key and returns its value.

Dictionaries

Examples of common dictionary methods

```
supply_by_sup.keys()     # dict_keys(['S1', 'S2'])
demand_by_cust.values()  # dict_values([18, 12])
cost_by_sup_cust.items()
# dict_items([(('S1', 'C1'), 5), (('S1', 'C2'), 7),
#             (('S2', 'C1'), 4), (('S2', 'C2'), 6)])

supply_by_sup.get("S3", 0)
# 0 (since "S3" is not in suppliers_dict)
demand_by_cust.pop("C2")
# 12 (and removes "C2" from the dictionary)
```

Dictionary views vs. lists

- Methods like `mydict.keys()`, `mydict.values()`, and `mydict.items()` return **dictionary view** objects.
- A dictionary view is **not exactly a list**, but you can loop over it like a list.
- If you need a real list (for example, to index or modify it), use `list(dict_view)`.

```
list(supply_by_sup.keys())     # ['S1', 'S2', 'S3']
list(demand_by_cust.values()) # [18, 12]
list(cost_by_sup_cust.items())
# [(('S1', 'C1'), 5), (('S1', 'C2'), 7)]
```

Greedy heuristic for the transportation problem

Goal: Assign flow from suppliers to customers in order of increasing unit cost, without violating supply or demand.

Pseudo code:

1. Initialize empty flow dictionary.
2. Sort all (*supplier*, *customer*) pairs by their unit cost (ascending).
3. **For each** (*supplier*, *customer*) pair in sorted order:
    a. *supply* $\leftarrow$ *supplier*'s current available units.
    b. *demand* $\leftarrow$ *customer*'s current required units.
    c. Set *quantity* to the smaller of *supply* and *demand*.
    d. Assign *quantity* from *supplier* to *customer* in the flow dictionary.
    e. Subtract *quantity* from *supplier*'s supply and *customer*'s demand.

Greedy heuristic for the transportation problem

Goal: Assign flow from suppliers to customers in order of increasing unit cost, without violating supply or demand.

Pseudo code:

1. Initialize empty flow dictionary.
2. Sort all (*supplier*, *customer*) pairs by their unit cost (ascending).
3. **For each** (*supplier*, *customer*) pair in sorted order:
    a. *supply* ← *supplier*'s current available units.
    b. *demand* ← *customer*'s current required units.
    c. Set *quantity* to the smaller of *supply* and *demand*.
    d. Assign *quantity* from *supplier* to *customer* in the flow dictionary.
    e. Subtract *quantity* from *supplier*'s supply and *customer*'s demand.

Greedy heuristic for the transportation problem

Initialize empty flow dictionary

- Create a dictionary to store the final solution: the quantity shipped from each supplier to each customer.
- The keys will be (*supplier*, *customer*) pairs, and the values are quantities shipped.

```
flow_by_sup_cust = {}
```

Greedy heuristic for the transportation problem

Goal: Assign flow from suppliers to customers in order of increasing unit cost, without violating supply or demand.

Pseudo code:

1. Initialize empty flow dictionary.
2. Sort all (*supplier*, *customer*) pairs by their unit cost (ascending).
3. **For each** (*supplier*, *customer*) pair in sorted order:
    a. *supply* ← *supplier*'s current available units.
    b. *demand* ← *customer*'s current required units.
    c. Set *quantity* to the smaller of *supply* and *demand*.
    d. Assign *quantity* from *supplier* to *customer* in the flow dictionary.
    e. Subtract *quantity* from *supplier*'s supply and *customer*'s demand.

Greedy heuristic for the transportation problem

Sort all (*supplier*, *customer*) pairs by their unit cost

- We want to sort the supplier-customer pairs by cost:

Original Dictionary		Sorted by Value
`('S1', 'C1'): 5`		`('S2', 'C1'): 4`
`('S1', 'C2'): 7`	$\longrightarrow$	`('S1', 'C1'): 5`
`('S2', 'C1'): 4`		`('S2', 'C2'): 6`
`('S2', 'C2'): 6`		`('S1', 'C2'): 7`

- This can be done with the help of the `sorted` function.

The `sorted()` function

General syntax

- `sorted()` is a built-in function that returns a new sorted list from an iterable.

```
sorted(iterable, key=None, reverse=False)
```

- **Parameters:** (Parameters marked as `optional` have a default value and can be omitted.)
  - `iterable`: The sequence (e.g., list, tuple, dict, etc.) to sort.
  - `key` (optional): A function that tells Python what to sort by.
  - `reverse` (optional): If `True`, sorts in descending order.
- `sorted()` always returns a new list and does not modify the original data.

The sorted() function

Example: Sorting shipments

- Applying `sorted` directly to `cost_by_sup_cust` returns a list of the dictionary's keys, sorted in ascending (lexicographic) order:

```
sorted(cost_by_sup_cust)
# [('s1', 'c1'), ('s1', 'c2'), ('s2', 'c1'), ('s2', 'c2')]
```

- To include both the keys and values in the result, sort the dictionary's items:

```
sorted(cost_by_sup_cust.items())
# [(('s1', 'c1'), 5), (('s1', 'c2'), 7),
#  (('s2', 'c1'), 4), (('s2', 'c2'), 6)]
```

Defining a function

- General syntax:

```
def function_name(arg1, arg2, ...):
    # Code block
    return value
```

- Example:

```
def get_cost(cost_tuple):
    cost = cost_tuple[1]
    return cost
```

- The function `get_cost` takes one argument: `cost_tuple`.
- The variable `cost` is assigned the 2nd element of the tuple (the cost).
- The function returns the `cost`.
- For example: `get_cost((('S1', 'C1'), 4))` returns `4`.

The sorted() function

Example: Sorting shipments by cost

- To sort by cost, we pass our function `get_cost` to the `key` parameter:

```
def get_cost(cost_tuple):
    cost = cost_tuple[1]
    return cost


...

sorted_costs = sorted(cost_by_sup_cust.items(), key=get_cost)
# [(('s2', 'c1'), 4), (('s1', 'c1'), 5),
#  (('s2', 'c2'), 6), (('s1', 'c2'), 7)]
```

Lambda Expressions

- A **lambda expression** is an **anonymous function**—a function without a name.
- **General syntax:** `lambda x, y, ...: f(x, y, ...)` (for example: `lambda x, y: x + y`)
- When a function is simple and only used once, a lambda expression makes the code shorter and clearer.

```
# Instead of defining a separate function:
sorted_costs = sorted(cost_by_sup_cust.items(),
                      key=lambda x: x[1])
```

Greedy heuristic for the transportation problem

Goal: Assign flow from suppliers to customers in order of increasing unit cost, without violating supply or demand.

Pseudo code:

1. Initialize empty flow dictionary.
2. Sort all (*supplier*, *customer*) pairs by their unit cost (ascending).
3. **For each** (*supplier*, *customer*) pair in sorted order:
   a. *supply* ← *supplier*'s current available units.
   b. *demand* ← *customer*'s current required units.
   c. Set *quantity* to the smaller of *supply* and *demand*.
   d. Assign *quantity* from *supplier* to *customer* in the flow dictionary.
   e. Subtract *quantity* from *supplier*'s supply and *customer*'s demand.

Greedy heuristic for the transportation problem

Iterating over the sorted shipments

- We want to iterate over each pair of `sorted_costs`.

Iteration 1	Iteration 2	Iteration 3	Iteration 4
('S2', 'C1'): 4	('S2', 'C1'): 4	('S2', 'C1'): 4	('S2', 'C1'): 4
('S1', 'C1'): 5	('S1', 'C1'): 5	('S1', 'C1'): 5	('S1', 'C1'): 5
('S2', 'C2'): 6	('S2', 'C2'): 6	('S2', 'C2'): 6	('S2', 'C2'): 6
('S1', 'C2'): 7	('S1', 'C2'): 7	('S1', 'C2'): 7	('S1', 'C2'): 7

- This can be done with the help of a `for` loop.

The `for` loop

General syntax

- A `for` loop lets you repeat actions for each item in a sequence (like a list, tuple, or dictionary).

```
for variable in sequence:
    # code to execute for each item
```

- The loop ends automatically after all items are processed.

Remark

The code inside the `for` loop must be indented. Python uses indentation to determine which lines belong inside the loop.

The for loop

Unpacking values

```
for s_c_cost in sorted_costs:
    print(s_c_cost)  # 1st output: (("S1", "C1"), 4)

for s_c, cost in sorted_costs:
    print(s_c)  # 1st output: ("S1", "C1")
    print(cost)  # 1st output: 4

for (s, c), cost in sorted_costs:
    print(s)  # 1st output: S2
    print(c)  # 1st output: C3
    print(cost)  # 1st output: 3
```

- The loop visits each item in `sorted_costs` one at a time.
- You can unpack multiple values directly in the loop header.
- You can use the unpacked values (e.g, `s`, `c` and `cost`) inside the loop body.

More about loops

- The `while` **loop**:

```
while condition:
    # code block to repeat
```

- Control flow statements for managing loop execution (in both `for` and `while` loops):
  - `break`: Exits the loop immediately.
  - `continue`: Skips to the next iteration of the loop.
  - `else`: Executes after the loop only if it was not exited by `break`.

```
for s in supply_by_sup.values():
    if s == 10:
        print("A supply of 10 was found!")
        break
else:
    print("A supply of 10 was not found.")
```

Greedy heuristic for the transportation problem

Goal: Assign flow from suppliers to customers in order of increasing unit cost, without violating supply or demand.

Pseudo code:

1. Initialize empty flow dictionary.
2. Sort all (*supplier*, *customer*) pairs by their unit cost (ascending).
3. **For each** (*supplier*, *customer*) pair in sorted order:
    a. *supply* ← *supplier*'s current available units.
    b. *demand* ← *customer*'s current required units.
    c. Set *quantity* to the smaller of *supply* and *demand*.
    d. Assign *quantity* from *supplier* to *customer* in the flow dictionary.
    e. Subtract *quantity* from *supplier*'s supply and *customer*'s demand.

Greedy heuristic for the transportation problem

Assigning *supply* and *demand*

- The current supply of supplier `s` is given by the value for key `s` in the dictionary `supply_by_sup`:

```
supply = supply_by_sup[s]
```

- The current demand of customer `c` is given by the value for key `c` in the dictionary `demand_by_cust`:

```
demand = demand_by_cust[c]
```

Greedy heuristic for the transportation problem

Goal: Assign flow from suppliers to customers in order of increasing unit cost, without violating supply or demand.

Pseudo code:

1. Initialize empty flow dictionary.
2. Sort all (*supplier*, *customer*) pairs by their unit cost (ascending).
3. **For each** (*supplier*, *customer*) pair in sorted order:
   a. *supply* ← *supplier*'s current available units.
   b. *demand* ← *customer*'s current required units.
   c. Set *quantity* to the smaller of *supply* and *demand*.
   d. Assign *quantity* from *supplier* to *customer* in the flow dictionary.
   e. Subtract *quantity* from *supplier*'s supply and *customer*'s demand.

Greedy heuristic for the transportation problem

Deciding how much to ship at each step

- At each step, we want to ship as many units from supplier `s` to customer `c` as possible, without exceeding either the supplier's supply or the customer's demand.
- Setting *quantity* to the smaller value ensures that neither limit is violated.
- In other words, $quantity = \min(supply, demand)$.
- This is equivalent to the following decision process:
  - **If** $supply > demand$:
    - $quantity \leftarrow demand$
  - **Else if** $supply < demand$:
    - $quantity \leftarrow supply$
  - **Else**:
    - $quantity \leftarrow supply$ *(or demand, they are equal)*

Conditional statements

The if, elif, and else statements

```python
if condition1:
    # if condition1 is True
elif condition2:
    # if condition2 is True (and condition1 is False)
elif condition3:
    # if condition3 is True (and previous are False)
    # ...
else:
    # if none of the above conditions are True
```

- `if` checks the first condition.
- You can have zero, one, or many `elif` blocks to check additional conditions in order.
- `else` is optional, and runs if none of the above conditions are `True`.

Conditional statements

Example: deciding how much to ship at each step

- **Decision process:**
  - **If** *supply* $>$ *demand*:
    - $quantity \leftarrow demand$
  - **Else if** *supply* $<$ *demand*:
    - $quantity \leftarrow supply$
  - **Else**:
    - $quantity \leftarrow supply$ *(or demand, they are equal)*
- This decision process can be translated in Python as follows:

```python
if supply > demand:
    quantity = demand
elif supply < demand:
    quantity = supply
else:
    quantity = supply  # or demand
```

Greedy heuristic for the transportation problem

Goal: Assign flow from suppliers to customers in order of increasing unit cost, without violating supply or demand.

Pseudo code:

1. Initialize empty flow dictionary.
2. Sort all (*supplier*, *customer*) pairs by their unit cost (ascending).
3. **For each** (*supplier*, *customer*) pair in sorted order:
   a. *supply* $\leftarrow$ *supplier*'s current available units.
   b. *demand* $\leftarrow$ *customer*'s current required units.
   c. Set *quantity* to the smaller of *supply* and *demand*.
   d. Assign *quantity* from *supplier* to *customer* in the flow dictionary.
   e. Subtract *quantity* from *supplier*'s supply and *customer*'s demand.

Storing the shipment plan

- Record the assigned quantity of shipment in `flow_by_sup_cust` to keep track of how much is shipped from each supplier to each customer:

```
flow_by_sup_cust = {} # Defined previously

# ... some logic ...

for (s, c), cost in sorted_costs:

    # ... determine quantity to ship ...

    flow_by_sup_cust[(s, c)] = quantity
```

Greedy heuristic for the transportation problem

Goal: Assign flow from suppliers to customers in order of increasing unit cost, without violating supply or demand.

Pseudo code:

1. Initialize empty flow dictionary.
2. Sort all (*supplier*, *customer*) pairs by their unit cost (ascending).
3. **For each** (*supplier*, *customer*) pair in sorted order:
   a. *supply* $\leftarrow$ *supplier*'s current available units.
   b. *demand* $\leftarrow$ *customer*'s current required units.
   c. Set *quantity* to the smaller of *supply* and *demand*.
   d. Assign *quantity* from *supplier* to *customer* in the flow dictionary.
   e. Subtract *quantity* from *supplier*'s supply and *customer*'s demand.

Greedy heuristic for the transportation problem

Updating supply and demand

- Subtract `quantity` from the supply of supplier `s`:

```
supply_by_sup[s] -= quantity
```

- Subtract `quantity` from the demand of customer `c`:

```
demand_by_cust[c] -= quantity
```

- After each shipment, these updates keep track of what each supplier still has and what each customer still needs.

Greedy heuristic for the transportation problem

Printing the solution

- Print the shipment plan (how much is shipped from each supplier to each customer):

```
print(f"Shipment plan: {flow_by_sup_cust}")
# Shipment plan: {('S2', 'C1'): 18, ('S1', 'C1'): 0,
#                ('S2', 'C2'): 2, ('S1', 'C2'): 10}
```

- Print the remaining supply for each supplier:

```
print(f"supply_by_sup: {supply_by_sup}")
# Remaining supply: {'S1': 0, 'S2': 0}
```

- Print the remaining demand for each customer:

```
print(f"demand_by_cust: {demand_by_cust}")
# Remaining demand: {'C1': 0, 'C2': 0}
```

Greedy heuristic for the transportation problem

Creating a function

- To reuse our greedy heuristic, we can put it inside a function:

```python
def greedy_solve(supply_by_sup, demand_by_cust,
                 cost_by_sup_cust):
    flow_by_sup_cust = {}

    # ... greedy heuristic ...

    return flow_by_sup_cust
```

- We can then call the function with different input data:

```python
cost_by_sup_cust2 = {("S1", "C1"): 8, ("S1", "C2"): 2,
                     ("S2", "C1"): 5, ("S2", "C2"): 9}
supply_by_sup2 = {"S1": 13, "S2": 17}
demand_by_cust2 = {"C1": 9, "C2": 21}

solution2 = greedy_solve(supply_by_sup2, demand_by_cust2,
                         cost_by_sup_cust2)
```

Greedy heuristic for the transportation problem

Returning remaining supply and demand

- In addition to the shipment plan, we can return the remaining supply and the remaining demand.
- Python lets you return more than one value from a function (as a tuple).

```
def greedy_solve(supply_by_sup, demand_by_cust,
                 cost_by_sup_cust):

    # ... greedy heuristic ...

    return flow_by_sup_cust, supply_by_sup, demand_by_cust
```

- You can assign each return value to a different variable:

```
sol, rem_sup, rem_dem = greedy_solve(supply_by_sup,
                                     demand_by_cust,
                                     cost_by_sup_cust)
```

Greedy heuristic for the transportation problem

Adding an optional `verbose` parameter

- To control whether the solution is printed, we can add an **optional** `verbose` parameter (default: `False`).
- Optional arguments should always appear after positional arguments in the function definition.

```
def greedy_solve(supply_by_sup, demand_by_cust,
                 cost_by_sup_cust, verbose=False):
    # ... greedy heuristic ...
    if verbose:
        print(f"Shipment plan: {flow_by_sup_cust}")
        print(f"Remaining supply: {supply_by_sup}")
        print(f"Remaining demand: {demand_by_cust}")

    return flow_by_sup_cust
...

greedy_solve(sup, dem, costs)                # No output
greedy_solve(sup, dem, costs, verbose=True) # Prints solution
```

Modifying arguments

Example

- What are the values of `sup` and `dem` after calling `greedy_solve` in the following example?

```
cost = {("S1", "C1"): 5, ("S1", "C2"): 7,
        ("S2", "C1"): 4, ("S2", "C2"): 6}
sup = {"S1": 10, "S2": 20}
dem = {"C1": 18, "C2": 12}

greedy_solve(sup, dem, costs) # Are the dicts modified?
print(f"sup={sup}") # Output: ?
print(f"dem={dem}") # Output: ?
```

Modifying arguments

Example

- What are the values of `sup` and `dem` after calling `greedy_solve` in the following example?

```
cost = {("S1", "C1"): 5, ("S1", "C2"): 7,
        ("S2", "C1"): 4, ("S2", "C2"): 6}
sup = {"S1": 10, "S2": 20}
dem = {"C1": 18, "C2": 12}

greedy_solve(sup, dem, costs) # Are the dicts modified?
print(f"sup={sup}") # Output: ?
print(f"dem={dem}") # Output: ?
```

- Output of `print(f"sup={sup}")` and `print(f"dem={dem}")`:

```
sup={'S1': 0, 'S2': 0}
dem={'C1': 0, 'C2': 0}
```

- The dictionaries were modified! **Why?**

Mutable vs immutable arguments in functions

- The type of object you pass to a function—**mutable** or **immutable**—affects how it can be changed.
- **Immutable objects** (e.g., `int`, `float`, `str`, `tuple`):
  - The function **cannot change the original object**.
  - If "modified", a **new object** is created and the original remains unchanged.
- **Mutable objects** (e.g., `list`, `dict`, `set`):
  - The function **can change the original object**.
  - Changes made inside the function are **visible outside the function**.

Mutable vs immutable arguments in functions

Example: Direct modification

```
def greedy_solve(supply_by_sup, demand_by_cust,
                 cost_by_sup_cust):
    # ... some logic ...
    for (s, d), c in sorted(cost_by_sup_cust.items()):
        # ... determine quantity ...
        supply_by_sup[s] -= quantity # MODIFIES sup
        demand_by_cust[d] -= quantity # MODIFIES dem
    # ...

    return flow_by_sup_cust, supply_by_sup, \
        demand_by_cust

greedy_solve(sup, dem, costss)
```

- The lines marked above directly modify the dictionaries `sup` and `dem` passed as arguments.

Mutable vs immutable arguments in functions

Example: Making copies

```
def greedy_solve(supply_by_sup, demand_by_cust,
                 cost_by_sup_cust):
    rem_sup = supply_by_sup.copy()
    rem_dem = demand_by_cust.copy()
    # ... some logic ...
    for (s, d), c in sorted(cost_by_sup_cust.items()):
        supply = rem_sup[s]
        demand = rem_dem[d]
        # ... determine quantity ...
        rem_sup[s] -= quantity
        rem_dem[d] -= quantity
    # ...

    return flow_by_sup_cust, rem_sup, rem_dem
```

- To avoid modifying the original arguments, make a copy of the dictionaries inside the function before modifying them.
- Now, the original dictionaries remain unchanged after the function call.

Modules

- Putting the greedy heuristic in a function lets us use it multiple times in the same script with different data.
- What if we want to reuse `greedy_solve` in other programs, without copying the code each time?
- We can make any function or variable reusable by putting it in a **module** (a separate .py file), and then **import** it when needed.

Creating a module

```
# File: mymodule.py

def my_function(arg1, arg2):
    # ... implementation ...
```

Importing from a module

```
import mymodule                        # import the whole module
from mymodule import my_function # import only my_function
```

Modules

Example: Moving greedy_solve to greedy.py

- Place the greedy_solve function in its own file, greedy.py, to make it reusable.

```
# File: greedy.py

def greedy_solve(supply_by_sup, demand_by_cust,
                 cost_by_sup_cust):
    # ...implementation...
```

- Now, import and use it in any script:

```
# File: myscript.py

from greedy import greedy_solve

# Use the function as needed
greedy_solve(sup, dem, costs)
```

Pandas

What is pandas?

Pandas is a popular Python library for working with structured data:

- Efficiently loads, manipulates, and analyzes tabular data (like spreadsheets, CSV files, or SQL tables).
- The main data structure is the **DataFrame**, which looks like a table. Pandas also provides the **Series**, for one-dimensional data.
- Great for data cleaning, exploration, and preparation.

Installation:

```
pip install pandas
```

Import pandas:

```python
import pandas as pd
```

Example: Loading data for the transportation problem

```
costs_df = pd.read_csv("data/costs.csv")
```

- `pd.read_csv` loads a CSV file into a **DataFrame**.

columns

rows	supplier	customer	cost
0	S1	C1	5
1	S1	C2	7
2	S2	C1	4
3	S2	C2	6

- The row labels are called the **index** (accessible with `.index`).
- The column labels are called **columns** (accessible with `.columns`).

Pandas

Inspecting DataFrames

```
costs_df.head()     # Show the first 5 rows
costs_df.head(10)   # Show the first 10 rows
costs_df.tail()     # Show the last 5 rows

costs_df['cost']    # Select a single column as a Series
costs_df[['cost']]  # Select a single column as a DataFrame
costs_df[['cost', 'supplier']]  # Select multiple columns
                                # as a DataFrame
```

- `head(n)` shows the first `n` rows (`n` is optional, default is 5).
- `tail(n)` shows the last `n` rows (`n` is optional, default is 5).
- `df['supplier']` (single square brackets) returns a Series, not a DataFrame.
- `df[['col']]` (double square brackets) returns a DataFrame, even for one column.
- Double brackets are required to select multiple columns: `df[['col1','col2']]`.

Pandas

Accessing data with `loc`

- The `loc` method allows you to select rows and columns by label.

```
# Select a single row by index label
costs_df.loc[0]

# Select specific rows and columns by label
costs_df.loc[0, 'cost']

# Select multiple rows and columns by label
costs_df.loc[[0, 3], ['supplier', 'cost']]
```

- `loc[index, columns]` selects data by label, not position.
- Useful for precise data selection, filtering, and subsetting.

Pandas

Basic statistics with DataFrames

- `describe()` gives count, mean, std deviation, min, max, and percentiles:

```
costs_df.describe() # Summary statistics for the entire
                    # DataFrame

costs_df['cost'].describe() # Summary statistics for the
                            # cost column
```

- You can also use `mean()`, `std()`, `min()`, `max()`, and `median()` directly.

```
costs_df['cost'].mean() # Mean
costs_df['cost'].std() # Standard deviation
costs_df['cost'].min() # Minimum value
costs_df['cost'].max() # Maximum value
print(costs_df['cost'].median()) # Median
```

Pandas

Filtering rows with boolean conditions

```
# Show all rows where cost is less than 12
costs_df[costs_df['cost'] < 12]

# Combine conditions (e.g., cost is less than 12
# and supplier is 'S2')
costs_df[(costs_df['cost'] < 12)
         & (costs_df['supplier'] == 'S2')]

# Use | for "or" (e.g., cost is 7 or 19)
costs_df[(costs_df['cost'] == 7)
         | (costs_df['cost'] == 19)]
```

- Use boolean expressions inside [] to select rows.
- Combine conditions with & (and), | (or), and surround each condition with parentheses.
- The result is a filtered DataFrame.

Pandas

Converting DataFrames to dictionaries

```
# Convert supply and demand DataFrames to dict
supply_df.set_index('supplier').to_dict()['supply']
demand_df.set_index('customer').to_dict()['demand']

# Convert costs DataFrame to a dictionary
costs_df.set_index(
    ['supplier', 'customer']).to_dict()['cost']
```

- The `set_index` method sets one or more columns as the dictionary keys.
- The `to_dict()` method converts the DataFrame (or Series) to a Python dictionary.

Matplotlib

What is Matplotlib?

Matplotlib is a popular Python library for creating visualizations such as graphs and charts.

- Lets you turn data into plots: line charts, bar charts, heatmaps, and more.
- Integrates well with pandas and plain Python lists.
- The main module for plotting is `matplotlib.pyplot`, commonly imported as `plt`.

Installation:

```
pip install matplotlib
```

Import Matplotlib:

```python
import matplotlib.pyplot as plt
```

Matplotlib

The `plot` function

`plt.plot(x, y, ...)`

- Creates a **line plot**.
- `x`: Sequence of positions or labels.
- `y`: Sequence of numeric values.
- Common options: `marker`, `linestyle`, `color`.
- `plt.show()` renders and displays the current figure.

Example:

```
plt.plot(supply.keys(), supply.values(), marker="o",
         color="orange")
plt.show()
```

Matplotlib

Adding labels, titles, and legends

- `plt.xlabel("...")` sets the label for the x-axis.
- `plt.ylabel("...")` sets the label for the y-axis.
- `plt.title("...")` sets the plot title.
- `plt.legend()` shows a legend for labeled artists (e.g., lines or bars with `label=`).

Example:

```
plt.plot(supply.keys(), supply.values(), marker="o",
         color="orange", label="Supply")
plt.xlabel("Supplier")
plt.ylabel("Units")
plt.title("Supply")
plt.legend()
plt.show()
```

Matplotlib

The bar function

`plt.bar(x, height, ...)`

- Creates a **bar chart**.
- `x`: Sequence of labels or positions.
- `height`: Sequence of numeric values.
- Additional options: `color`, `label`, etc.

Example:

```
plt.bar(demand.keys(), demand.values(), color="blue",
        label="Demand")
plt.xlabel("Customer")
plt.ylabel("Units")
plt.title("Demand")
plt.legend()
plt.show()
```

Matplotlib

Saving a plot to a file

- Use `plt.savefig("filename.png")` to save the current plot to an image file.
- You can specify the file format by changing the extension (e.g., `.png`, `.jpg`, `.pdf`, `.svg`).
- `plt.savefig` must be called **before** `plt.show()`.

Example: Saving a line plot

```
plt.plot(supply.keys(), supply.values(), marker="o",
         color="orange", label="Supply")
plt.xlabel("Supplier")
plt.ylabel("Units")
plt.title("Supply")
plt.legend()
plt.savefig("supply_plot.png")
plt.show()
```

Pyomo

What is Pyomo?

- **Pyomo** is a Python-based, open-source optimization modeling language.
- It allows you to define mathematical models for linear, integer, and nonlinear optimization problems.
- Pyomo provides a high-level, readable, and flexible interface for expressing variables, constraints, and objectives.
- Pyomo models can be solved by a variety of external solvers (e.g., GLPK, CBC, Gurobi, CPLEX).

Installation:

```
pip install pyomo
```

Import Pyomo:

```
from pyomo.environ import *
```

Pyomo

Transportation problem: sets, parameters and variables

- Sets:
  - I: set of suppliers
  - J: set of customers
- Parameters:
  - s_i: supply available at supplier $i \in I$
  - d_j: demand required at customer $j \in J$
  - c_{ij}: cost per unit shipped from supplier i to customer j
- Decision variables:
  - x_{ij}: quantity shipped from supplier i to customer j

Pyomo

Transportation problem: objective function and constraints

- Objective function:

$$\text{Minimize } Z = \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij}$$

- Constraints:
  - Supply capacity:

$$\sum_{j \in J} x_{ij} \leq s_i \qquad \forall i \in I$$

  - Demand satisfaction:

$$\sum_{i \in I} x_{ij} \geq d_j \qquad \forall j \in J$$

  - Non-negativity:

$$x_{ij} \geq 0 \qquad \forall i \in I,\ \forall j \in J$$

Pyomo

Basic building blocks in Pyomo

- **ConcreteModel**: The main object that holds sets, parameters, variables, constraints, and objectives.
- **Set**: Represents a finite collection of elements (e.g., suppliers, customers).
- **Var**: Decision variables whose values are determined by the optimization solver.
- **Constraint**: Mathematical conditions that must be satisfied.
- **Objective**: The function to be minimized or maximized.

Pyomo

Defining a Pyomo model

- A `ConcreteModel` in Pyomo is a container for sets, parameters, variables, constraints, and objectives.
- `ConcreteModel()` creates a model where all components are fully specified when the script runs.

```
model = ConcreteModel()
```

- Sets represent index collections for variables and constraints.
- The `initialize` argument fills the set from Python objects.

```
model.S = Set(initialize=supply.keys())   # Suppliers
model.C = Set(initialize=demand.keys())   # Customers
```

- `S` and `C` are user-defined names for sets.

Pyomo

Defining variables

- Variables represent the decisions to be made (e.g., how much to ship from each supplier to each customer).
- In the code:

```
model.x = Var(model.S, model.C, domain=NonNegativeReals)
```

- `model.x[i,j]` is the amount shipped from supplier i to customer j.
- `NonNegativeReals` ensures variables are ≥ 0.

Pyomo

Defining the objective function

- Minimize the total transportation cost:

```
model.obj = Objective(
    expr=sum(costs[(i, j)] * model.x[i, j]
             for i in model.S for j in model.C),
    sense=minimize
)
```

- `costs[(i, j)] * model.x[i, j]` is computed for every supplier *i* and customer *j*.
- `for i in model.S for j in model.C` loops over all supplier-customer pairs.
- The expression inside `sum(...)` is called a Python **generator expression**—it produces terms one at a time, making this both concise and memory efficient.

Pyomo

Defining the constraints

- Each supplier's total shipment does not exceed its available supply.

```
model.sup_const = Constraint(
    model.S,
    rule=lambda m, i: sum(m.x[i, j]
                          for j in m.C) <= supply[i]
)
```

- Each customer's demand is satisfied.

```
model.dem_const = Constraint(
    model.C,
    rule=lambda m, j: sum(m.x[i, j]
                          for i in m.S) >= demand[j]
)
```

- Pyomo uses external solvers (like GLPK, CBC, Gurobi, CPLEX) to find optimal solutions.
- You can choose a solver by name:

```
solver = SolverFactory("cbc")
```

- Replace `"cbc"` with the name of your solver (e.g., `"gurobi"`, `"cplex"`).

- Once the solver is configured, you can solve the model:

```
results = solver.solve(model, tee=True)
```

- `solve()` runs the optimization and returns the results.
- The `tee` argument controls whether to display solver output in the console.

Pyomo

Extracting the results

- After solving, you can access the optimal variable values and objective value:

```
flows_opt = {}
for i in model.S:
    for j in model.C:
        flows_opt[(i, j)] = value(model.x[i, j])

opt_cost = value(model.obj)
```

- value() extracts the numerical value from a Pyomo variable or expression.