6CS012 - Artificial Intelligence and Machine Learning. A Tutorial and Assessment on Python and NumPy for Deep Learning.

Prepared By: Siman Giri {Module Leader - 6CS012}

Feb 17, 2025

——————— Welcome - to - 6CS012. ———————-

1 About a Document

This document, **A Tutorial and Assessment on Python and NumPy for Deep Learning,** serves as a comprehensive guide for students to strengthen their foundational programming skills essential for deep learning. Completion and submission of all tasks outlined in this document is a mandatory requirement for the course.
It combines an in-depth tutorial on Python and NumPy with structured assessments to reinforce key concepts. The tutorial covers fundamental Python programming constructs, data manipulation techniques, and NumPy operations critical for efficient numerical computing in deep learning applications. The assessment section includes hands-on exercises and problem-solving tasks to evaluate the reader's understanding and application of these concepts. Whether you are a beginner or looking to refine your skills, this document provides a structured approach to mastering Python and NumPy for deep learning.

2 Instructions

This is a Pre-requisite homework assignment to be completed on your own before your second workshop and is compulsory to submit.
{**Cautions!!!:**Failure to Submit this assignment might affect your future grades and ability to receive highest grades}.
Please answer the questions below using python in the Jupyter Notebook and follow the guidelines below:

- This worksheet must be completed individually.
- All the solutions must be written in Jupyter Notebook.
- For exercise on Python:
  - You are allowed to use basic packages like `time, collection etc.` but do not use the packages to solve the problem directly.
- For exercise on Numpy:
  - You are allowed to use packages and module with in Numpy Library.

3 Getting Started with Python.

This is NOT a problem, but you are highly recommended to run the following code with some of the input changed in order to understand the meaning of the operations.

- **Cautions!!!:**
  - This Guide doesnot contain sample output, as we expect you to re-write the code and observe the output.
  - If found: any error or bugs, please report to your instructor and Module leader.
    {Will hugely appreciate your effort.}

3.1 About a Python:

Python is a high-level, general-purpose programming language created by Guido van Rossum, first released in 1991. Its design focuses on making code easy to read and understand, using clear formatting and meaningful whitespace. Python's structure and support for object-oriented programming help programmers write organized, logical code adaptable for both small-scale and large-scale projects. Known for its flexibility and powerful libraries, Python has become a popular language for machine learning research and is the main language used in frameworks like `Numpy, Pandas, Matplotlib, sickit learn` and many more.

Python Version Check

```
import sys
# Check Python version
if sys.version_info.major == 3 and sys.version_info.minor >= 6:
    print("Hello, World!")
    print(f"Python version: {sys.version}")
else:
    print("Please use Python 3.6 or higher.")
```

3.2 Data Types in Python.

Mutable Data Types: Mutable objects can be changed after they are created. This means you can modify their contents, such as adding or removing items, or changing their values without creating a new object. For example - `list`

Immutable Data Types: Immutable objects cannot be changed once they are created. Any operation that seems to modify an immutable object will actually create a new object instead of changing the original.For example - `tuple`

Difference between Mutable and Immutable Data Types.

```
# Mutable Data Type Example (List)
my_list = [1, 2, 3]
print("Original List:", my_list)
# Modifying the list (changing contents)
my_list.append(4)
print("Modified List:", my_list)
# Immutable Data Type Example (Tuple)
my_tuple = (1, 2, 3)
print("Original Tuple:", my_tuple)
```

```
# Trying to modify the tuple (will raise an error)
# my_tuple.append(4) # Uncommenting this line will raise an AttributeError
```

3.2.1 Some Common Data Types in Python:

Numeric:

These data types are used to represent numerical values.

- `int`: Represents integer values (e.g., 10, -3).
- `float`: Represents floating-point (decimal) values (e.g., 10.5, -2.7).
- `complex`: Represents complex numbers (e.g., 3 + 4j).

Data Types - Numeric

```
age = 23 #int
pi = 3.14 #float
temperature = -5.5 #float
print("data type of variable age = ", type(age))
print("data type of variable pi = ", type(pi))
print("data type of variable temperature = ", type(temperature))
```

Sequence:

These data types are ordered collections of items. You can access elements by their position (index).

- `str`: string (str) represents sequence of characters enclosed by double quotes or single quotes. (e.g., "Hello, World!").It is an immutable sequence.

Data Types - Sequence - String

```
name = "Alice"
greeting = 'Hello'
address = "123 Main St"
print("data type of the variable name = ",type(name))
print("data type of the variable greeting = ",type(greeting))
print("data type of the variable address = ",type(address))
# slice only one element
print("The first letter of the name is:", name[0])
print("The last letter of the name is:", name[-1])
# slice a range of elements
print("The second letter to the fourth of the name is:", name[1:4])
print("The first two letters of the name are:", name[:2])
print("Substring starting from the third letter is:", name[2:])
```

- `list`: Represents lists, which can contain mixed data types and are mutable (e.g., `["apple", "banana"]`).

Data Types - Sequence - List

```
list1 = [1, 2, 3, 4]
mixed_list = [12, "Hello", True]
# List is mutable
mixed_list[0] = False
mixed_list
```

- `tuple`: Represents tuples, which are ordered and immutable collections (e.g., `(1, 2, 3)`).

Data Types - Sequence - Tuple

```
colors = ('red', 'green', 'yellow', 'blue')
print("First element:", colors[0])
print("Last two elements:", colors[2:])
print("Middle two elements:", colors[1:3])
colors[0] = 'purple'
colors # will generate an error as tuple is immutable.
```

Mapping:

This category includes data types that store key-value pairs, allowing for efficient retrieval based on keys.

- `dict`: Represents dictionaries, which can store various data types as values associated with unique keys (e.g., `"name": "Alice", "age": 25`).

Data Types - Sequence - Dict

```
person = {'name':'John','age':30,'city':'Pittsburgh'}
print(f"Hello my name is {person['name']}. I am {person['age']} years old and I live at {
    person['city']}.")
print("All keys:", list(person.keys()))
print("All values:", list(person.values()))
```

Set:

Sets are unordered collections of unique elements. They are useful for membership testing and eliminating duplicate entries.

- `set`: A mutable collection of unique items (e.g., `1, 2, 3`).
- `frozenset`: An immutable version of a set (e.g., `frozenset([1, 2, 3])`).

Data Types - Set

```
unique_numbers = {1,2,3,3,3,3,4,5}
print(unique_numbers)
```

Boolean:

This category contains types that represent truth values.

- `bool`: Represents boolean values (True or False).

Data Types - Boolean

```
is_student = True
has_license = False
print("data type of the variable is_student = ",type(is_student))
print("data type of the variable has_license = ",type(has_license))
```

Special:

This category is used for unique data types that do not fit into the other categories.

- `NoneType`: Represents the absence of a value or a null value (e.g., `None`).

Data Types - Boolean

```
result = None
```

Data - Types - Summary:

Data Type	Category	Sample Code
`list`	Sequence	`fruits = ["apple", "banana", "cherry"]`
`dict`	Mapping	`person = {"name": "Bob", "age": 25}`
`set`	Set	`unique_values = {1, 2, 3}`
`bytearray`	Sequence	`data = bytearray(b"hello")`

Table 1: Mutable Data Types in Python

Data Type	Category	Sample Code
`int`	Numeric	`x = 10`
`float`	Numeric	`y = 10.5`
`complex`	Numeric	`z = 3 + 4j`
`str`	Sequence	`name = "Alice"`
`tuple`	Sequence	`coordinates = (10, 20)`
`frozenset`	Set	`frozen_values = frozenset([1, 2, 3])`
`bool`	Boolean	`is_active = True`
`NoneType`	Special	`value = None`

Table 2: Immutable Data Types in Python

3.3 Logical Statements and Loops.

Python code can be decomposed into packages, modules, statements, and expressions, as follows:

1. Expressions create and process objects:
    - Expressions are part of statements that return a value, such as variables, operators, or function calls.
2. Statements contain expressions:
    - Statements are sections of code that perform an action. The main groups of Python statements are: assignment statements, print statements, conditional statements (if, break, continue, try), and looping statements (for, while).
3. Module Contain statements:
    - Modules are Python files that contain Python statements, and are also called scripts.
4. Packages are composed of modules:
    - Packages are Python programs that collect related modules together within a single directory hierarchy.

3.4 Logical Statements:

Logical statements are used to perform conditional operations. The primary logical statements in Python are:

- `if`: Executes a block of code if the condition is true.
```
if condition:
    # Code to execute if condition is true
```
- `elif`: Short for "else if," allows for multiple conditions to be checked sequentially.
```
if condition1:
    # Code for condition1
elif condition2:
    # Code for condition2
```
- `else`: Executes a block of code if none of the preceding conditions are true.
```
if condition:
    # Code if condition is true
else:
    # Code if condition is false
```

Sample Code - if-else statement

```
num = 10
if num > 0:
    print("Positive")
elif num == 0:
    print("Zero")
else:
    print("Non-positive")
```

- Comparison operators: Used to compare values.
  - `==`: Equal to
  - `!=`: Not equal to
  - `<`: Less than
  - `>`: Greater than
  - `<=`: Less than or equal to
  - `>=`: Greater than or equal to

Sample Code - Comparision Operator

```
x = 5
y = 10
if x > 0 and y < 20:
    print("Both conditions are true")
if x > 0 or y > 20:
    print("At least one condition is true")
if not x == 0:
    print("x is not equal to 0")
```

- Logical operators: Used to combine multiple conditions.
  - `and`: True if both conditions are true.
  - `or`: True if at least one condition is true.
  - `not`: Inverts the truth value of the condition.

Sample Code - Logical Operator

```
# Define two boolean variables
a = True
b = False
# Using the 'and' operator
if a and b:
    print("Both a and b are True")
else:
    print("Either a or b is False") # This will be printed
# Using the 'or' operator
if a or b:
    print("At least one of a or b is True") # This will be printed
else:
    print("Both a and b are False")
# Using the 'not' operator
if not a:
    print("a is False") # This will not be printed
else:
    print("a is True") # This will be printed
```

3.4.1 Loops:

Loops are used to execute a block of code multiple times. The primary loop types in Python are:

- `for` loop: Iterates over a sequence (like a list, tuple, or string).

```
for item in iterable:
    # Code to execute for each item
```

- `while` loop: Repeats as long as a condition is true.

```
while condition:
    # Code to execute while condition is true
```

- `break`: Exits the loop immediately.

```
for item in iterable:
    if some_condition:
        break  # Exit loop
```

- `continue`: Skips the current iteration and continues with the next iteration of the loop.

```
for item in iterable:
    if some_condition:
        continue  # Skip to the next iteration
```

Sample Code - Various Loops

```python
# for loop:
fruits = ["apple", "banana","cherry"]
for fruit in fruits:
    print(fruit)
# While loop:
count = 0
while count < 5:
    print("Count is:", count)
    count += 1
# Break
fruits = ["apple", "banana", "orange"]
print("loop 1")
for fruit in fruits:
    print(fruit)
    if fruit == "apple":
        break
# Continiue
print("loop 2")
for fruit in fruits:
    if fruit == "apple":
        continue
    else:
        print(fruit)
```

3.5 Functions.

Functions are fundamental building blocks in Python, similar to mathematical functions that define relationships between inputs and outputs. In mathematics, a function such as

$$z = f(x, y)$$

maps inputs x and y to an output z.
However, functions in programming languages are more generalized and versatile; they can process various data types and perform a wide range of operations on inputs beyond simple mathematical relationships.
A function in programming is a self-contained block of code that encapsulates a specific task or a set of related tasks. It defines how inputs relate to outputs through the operations performed within the function. When a function is called, input arguments are provided, the program executes the defined code, and then returns the function's output. This structure enables functions to manage complex processes in a structured and reusable way. Most programming languages offer support for pre-built and user-defined functions to promote following:

1. **Code Abstraction and Re-usability:** A key principles in software development. Rather than repeating the same code across multiple locations in an application, we can define a single function that performs the desired task and call it wherever needed. This approach not only reduces redundancy but also simplifies maintenance; if a change is required, we modify it in one place—the function itself—rather than updating every instance where the code appears.

2. **Code Modularity:** By breaking down complex processes into smaller, self-contained functions that focus on specific tasks, we create modular code that is both organized and easier to understand. This modular structure enhances readability and maintainability, making it simpler to update or expand the program without disrupting its overall structure.

3.6 Anatomy of a Function:

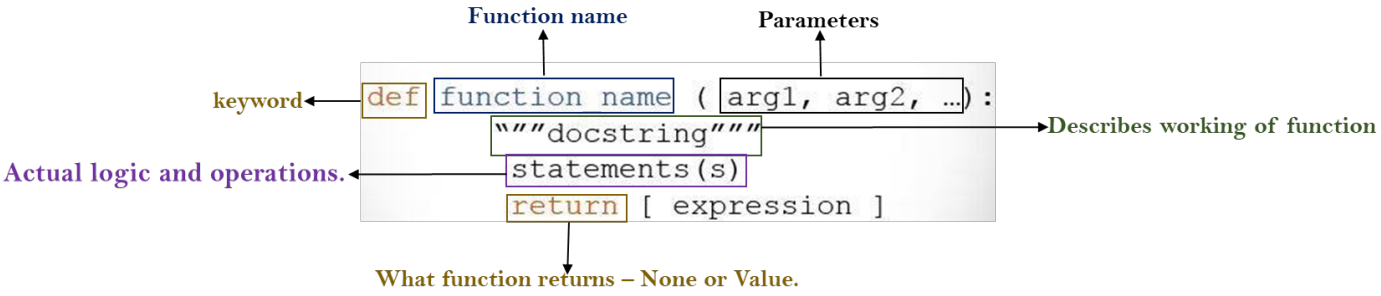


Fig: A Function.

Figure 1: Anatomy of a Function

Here's a brief overview of the main components of a function:

- **Function Name:** This is the identifier by which the function is called. It should be descriptive, conveying the function's purpose or the task it performs. In Python, function names typically follow snake_case formatting (e.g., `calculate_area`).

- **Input Parameters:** These are variables listed within the parentheses after the function name in the definition. Parameters allow functions to accept inputs, which are used within the function's body to perform calculations or operations. Parameters make functions more flexible and reusable, as they allow you to pass different values each time you call the function.
- **Docstring:** A docstring is an optional, multi-line comment placed just after the function definition. It provides documentation on the function's purpose, parameters, and output, helping others understand what the function does. In Python, a docstring is written within triple quotes (""" """) and can be accessed using the help() function.
- **Return Output:** The `return` statement specifies the output of the function, allowing it to send back a result to the part of the program where it was called. This enables further processing or storage of the function's output. If no `return` statement is specified, the function will `return` None by default.

How to write a Function Correctly? - Docstring are Most.

Correct way to Write Function

```
    def add_binary(a, b):
'''
Returns the sum of two decimal numbers in binary digits.

Parameters:
       a (int): A decimal integer
       b (int): Another decimal integer
Returns:
       binary_sum (str): Binary string of the sum of a and b
'''
binary_sum = bin(a+b)[2:]
return binary_sum
```

3.6.1 Built - in - Functions:

Python provides a wide range of built-in functions that are ready to use. These functions perform common tasks and are available without the need to import additional modules.Presented below is an example showcase; for more details, refer to the Python Reference on W3Schools

Example on Built - in - Function

```
print("Hello, World!") # Output: Hello, World!
length = len("Hello") # Output: 5
data_type = type(42) # Output: <class 'int'>
total = sum([1, 2, 3]) # Output: 6
```

3.6.2 Built - in - Methods:

Methods are similar to functions but are associated with objects. They act specifically on data types (e.g., `strings, lists, dictionaries` and are called using dot notation `object.method()`. Presented below is an example showcase; for more details, refer to the Python Reference on W3Schools

Example on Built - in - Methods

```
    # str.upper(): Converts all characters in a string to uppercase.
    message = "hello".upper() # Output: "HELLO"
    # list.append(): Adds an element to the end of a list.
    fruits = ["apple", "banana"]
    fruits.append("cherry") # Output: ["apple", "banana", "cherry"]
    # dict.get(): Returns the value associated with a key in a dictionary.
    info = {"name": "Alice", "age": 25}
    age = info.get("age") # Output: 25
```

3.6.3 User - Defined - Function:

User-defined functions are created by the user to perform specific tasks. These functions are defined using the `def` keyword and may include parameters and return values.
Here is an example function python that converts Celsius to Fahrenheit and vice versa.

Example of well-structured user-defined function

```
    def temperature_converter():
    """
    Converts temperature between Celsius and Fahrenheit.
    This function prompts the user to specify the conversion type (Celsius to Fahrenheit or
        Fahrenheit to Celsius)
    and then asks for the temperature value. It calculates and returns the converted temperature.
    Returns:
        float: The converted temperature value.
    """
    print("Choose conversion type:")
    print("1. Celsius to Fahrenheit")
    print("2. Fahrenheit to Celsius")
    # Get conversion choice from user
    choice = input("Enter 1 or 2: ")
    if choice == "1":
        # Celsius to Fahrenheit conversion
        celsius = float(input("Enter temperature in Celsius: "))
        fahrenheit = (celsius * 9/5) + 32
        print(f"{celsius}C is equal to {fahrenheit}F")
        return fahrenheit
    elif choice == "2":
        # Fahrenheit to Celsius conversion
        fahrenheit = float(input("Enter temperature in Fahrenheit: "))
        celsius = (fahrenheit - 32) * 5/9
        print(f"{fahrenheit}F is equal to {celsius}C")
        return celsius
    else:
        print("Invalid choice. Please enter 1 or 2.")
        return None
# Call the function
temperature_converter()
```

3.7 Global and Local Variables:

1. Global Variables:

A **global variable** is defined outside of any function and is accessible from any part of the program, including within functions. Global variables maintain their values throughout the program's execution and can be accessed or modified by any function unless explicitly declared as `nonlocal` or redefined within the function.

Global Varaible

```
x = 10 # Global Variable.
def print_global():
    print(x) # Accessing global variable
print_global() # Output: 10
```

In this example, the variable `x` is global and can be accessed inside the `print_global()` function.

2. Local Variables:

A **local variable** is defined within a function and can only be accessed within that function. It exists only during the function's execution, and once the function completes, the local variable is removed from memory.

Global Varaible

```
def print_local():
    y = 5 # Local variable
    print(y)

print_local() # Output: 5
print(y) # This would cause an error because y is not accessible outside the function
```

Here, `y` is a local variable within `print_local()` and cannot be accessed outside the function.

3. Accessing and Modifying Global Variables Inside a Function

To modify a global variable within a function, we must use the `global` keyword. Otherwise, assigning a new value to a global variable within a function will create a new local variable with the same name, leaving the global variable unchanged.

Global Varaible

```
x = 10 # Global variable
def modify_global():
    global x # Declaring x as global
    x = 20 # Modifying global x
modify_global()
print(x) # Output: 20
```

Using `global` here allows the function to modify the global variable `x` directly.

3.8 Exception and Error Handling.

In programming, errors can interrupt the normal flow of execution, potentially causing the program to crash. **Exception and Error handling** is a method to detect and respond to such errors gracefully, ensuring the program can handle unexpected situations without failing.

1. Types of Error:

Errors in Python can be broadly classified into:

- **Syntax Errors**: Occur when the Python parser finds code that violates Python's syntax rules. These are usually detected before the program runs.
```
# Example of Syntax Error
print("Hello World)  # Missing closing quote
```
- **Exceptions**: Occur at runtime and indicate unexpected conditions that disrupt program execution. Examples include division by zero, accessing an undefined variable, or trying to open a file that does not exist.
```
# Example of an Exception
result = 10 / 0  # Raises ZeroDivisionError
```

2. Handling Exceptions with `try-except`

To handle exceptions, Python uses the `try-except` structure. Code that may raise an exception is placed within the `try` block, and if an exception occurs, it is caught and handled in the `except` block.

- Example:
```
try:
    num = int(input("Enter a number: "))
    print("Result:", 10 / num)
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Invalid input. Please enter a number.")
```

In this example, `ZeroDivisionError` and `ValueError` are handled specifically, ensuring that the program continues even if the user inputs invalid data.

3. Using `finally` Block

The `finally` block is optional and executes regardless of whether an exception occurs or not. It is typically used for cleanup tasks, like closing files or releasing resources.

- Example:
```
try:
    file = open("example.txt", "r")
    data = file.read()
except FileNotFoundError:
    print("File not found.")
finally:
    file.close()  # Ensures the file is closed
```

The `finally` block here ensures that the file is closed, even if an exception occurs.

4. Raising Custom Exceptions

Python allows the creation of custom exceptions using the `raise` statement. This is useful for handling specific errors not covered by built-in exceptions.

- Example:

```
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    else:
        print("Valid age")
try:
    check_age(-1)
except ValueError as e:
    print(e)
```

This example raises a `ValueError` if an invalid age is provided.

3.8.1 To - Summarize:

- **Errors:** occur due to incorrect code structure and are identified before execution.
- **Common Errors in Python:**

Error	Description
`SyntaxError`	Occurs when the Python interpreter encounters incorrect syntax. Example: Missing colon in function definition.
`IndentationError`	Raised when the indentation is incorrect. Example: Mixing spaces and tabs in indentation.
`NameError`	Happens when a variable is used before it has been defined.
`TypeError`	Raised when an operation or function is applied to an object of inappropriate type. Example: Adding a string and an integer.
`IndexError`	Occurs when trying to access an index that is out of range in a list or tuple.
`KeyError`	Raised when trying to access a non-existent key in a dictionary.
`AttributeError`	Happens when an invalid attribute is accessed on an object. Example: Calling a method that does not exist.

Table 3: Common Errors in Python

- **Exceptions:** occur at runtime and can be handled using `try-except` blocks.
  - The `finally` block ensures that essential cleanup code is run, regardless of exceptions.
  - Custom exceptions can be created with `raise` to handle specific scenarios.
- Understanding and implementing exception handling is essential for writing resilient code that can gracefully handle unexpected situations.

- **Common Exceptions in Python:**

Exception	Description
`AssertionError`	Raised when an assertion statement fails. Example: `assert 2 + 2 == 5`
`ZeroDivisionError`	Occurs when attempting to divide by zero. Example: `10 / 0`
`ImportError`	Raised when an import statement fails because the module is not found.
`ValueError`	Happens when an operation receives an argument of the right type but inappropriate value. Example: `int("abc")`
`RuntimeError`	Raised when an error occurs that does not fit into other exception categories.
`FileNotFoundError`	Occurs when trying to open a non-existent file.
`KeyboardInterrupt`	Raised when the user interrupts program execution using Ctrl+C.

Table 4: Common Exceptions in Python

3.9 Reading and Understanding Error Message:

Errors are a natural part of programming and can arise for various reasons, such as:

1. Mistyped or incorrectly structured code:
   - e.g., forgetting a colon (:) at the end of a function definition.
2. Performing an invalid operation:
   - e.g., dividing by zero or trying to access an element in a list that does not exist.
3. Incorrect use of data types:
   - e.g., attempting to concatenate a string and an integer without proper conversion.
4. Referencing an undefined variable:
   - e.g., using a variable before assigning it a value.
5. Issues in program logic:
   - e.g., writing a loop that never terminates or implementing an incorrect formula.

Python provides error messages to help programmers identify and fix these issues.

Example - Error Message.

```
Traceback (most recent call last):
  File "nx_error.py", line 41, in <module>
    print friends_of_friends(rj, myval)
  File "nx_error.py", line 30, in friends_of_friends
    f = friends(graph, user)
  File "nx_error.py", line 25, in friends
    return set(graph.neighbors(user))#
  File "/Library/Frameworks//graph.py", line 978, in neighbors
     return list(self.adj[n])
TypeError: unhashable type: 'list'
```

Figure 2: An Example of Error Message.

How to Read an Error Message?

1. **Traceback Message:**

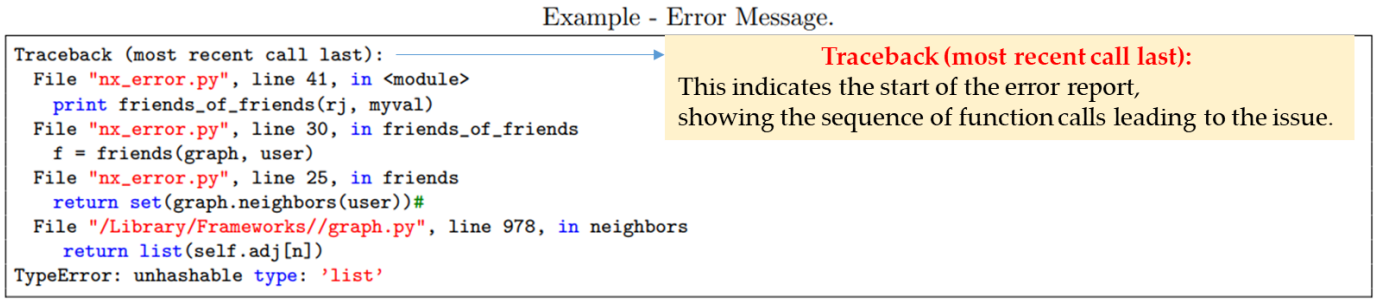


Figure 3: Analyzing Error Message - 1.

2. **The Call Stack (Files and Line Numbers):**

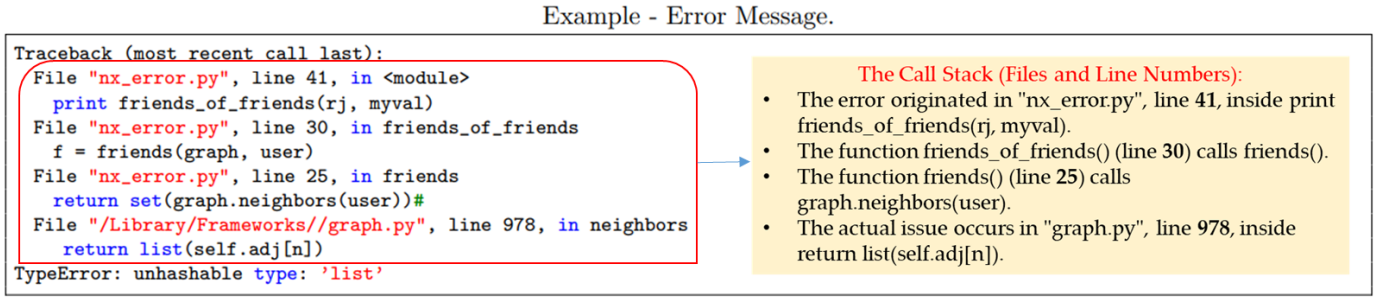


Figure 4: Analyzing Error Message - 2.

3. **Error Type and Error Message:**

Example - Error Message.

```
Traceback (most recent call last):
  File "nx_error.py", line 41, in <module>
    print friends_of_friends(rj, myval)
  File "nx_error.py", line 30, in friends_of_friends
    f = friends(graph, user)
  File "nx_error.py", line 25, in friends
    return set(graph.neighbors(user))#
  File "/Library/Frameworks//graph.py", line 978, in neighbors
     return list(self.adj[n])
TypeError: unhashable type: 'list'
```

Error Type: TypeError

- This tells us that there is a **type mismatch** in the operation.

Error Message: unhashable type: 'list'

- This means that a **list** was used in a place where a hashable object (like a string or integer) was expected.
- The issue likely arises because n (a user identifier) is a **list**,

but dictionary keys (used in self.adj[n]) must be immutable types like strings or integers.

Figure 5: Analyzing Error Message - 3.

4 TO - DO - Task

Please complete all the problem listed below.

4.1 Exercise on Functions:

Task - 1:

Create a Python program that converts between different units of measurement.

- The program should:
  1. Prompt the user to choose the type of conversion (e.g., length, weight, volume).
  2. Ask the user to input the value to be converted.
  3. Perform the conversion and display the result.
  4. Handle potential errors, such as invalid input or unsupported conversion types.
- Requirements:
  1. **Functions:** Define at least one function to perform the conversion.
  2. **Error Handling:** Use **try-except** blocks to handle invalid input (e.g., non-numeric values).
  3. **User Input:** Prompt the user to select the conversion type and input the value.
  4. **Docstrings:** Include a docstring in your function to describe its purpose, parameters, and return value.
- Conversion Options:
  1. **Length:**
     - Convert meters (m) to feet (ft).
     - Convert feet (ft) to meters (m).
  2. **Weight:**
     - Convert kilograms (kg) to pounds (lbs).
     - Convert pounds (lbs) to kilograms (kg).
  3. **Volume:**
     - Convert liters (L) to gallons (gal).
     - Convert gallons (gal) to liters (L).

Task - 2:

Create a Python program that performs various mathematical operations on a list of numbers.

- **The Program should:**
  1. Prompt the user to choose an operation (e.g., find the sum, average, maximum, or minimum of the numbers).
  2. Ask the user to input a list of numbers (separated by spaces).

3. Perform the selected operation and display the result.
4. Handle potential errors, such as invalid input or empty lists.

- **Requirements:**

1. **Functions:** Define at least one function for each operation (sum, average, maximum, minimum).
2. **Error Handling:** Use try-except blocks to handle invalid input (e.g., non-numeric values or empty lists).
3. **User Input:** Prompt the user to select the operation and input the list of numbers.
4. **Docstrings:** Include a docstring in each function to describe its purpose, parameters, and return value.

4.2 Exercise on List Manipulation:

1. Extract Every Other Element:

Write a Python function that extracts every other element from a list, starting from the first element.

- **Requirements:**
  - Define a function `extract_every_other(lst)` that takes a list `lst` as input and returns a new list containing every other element from the original list.
  - Example: For the input `[1, 2, 3, 4, 5, 6]`, the output should be `[1, 3, 5]`.

2. Slice a Sublist:

Write a Python function that returns a sublist from a given list, starting from a specified index and ending at another specified index.

- **Requirements:**
  - Define a function `get_sublist(lst, start, end)` that takes a list `lst`, a starting index `start`, and an ending index `end` as input and returns the sublist from `start` to `end` (inclusive).
  - Example: For the input `[1, 2, 3, 4, 5, 6]` with `start=2` and `end=4`, the output should be `[3, 4, 5]`.

3. Reverse a List Using Slicing:

Write a Python function that reverses a list using slicing.

- **Requirements:**
  - Define a function `reverse_list(lst)` that takes a list `lst` and returns a reversed list using slicing.
  - Example: For the input `[1, 2, 3, 4, 5]`, the output should be `[5, 4, 3, 2, 1]`.

4. Remove the First and Last Elements:

Write a Python function that removes the first and last elements of a list and returns the resulting sublist.

- **Requirements:**
  - Define a function `remove_first_last(lst)` that takes a list `lst` and returns a sublist without the first and last elements using slicing.
  - Example: For the input `[1, 2, 3, 4, 5]`, the output should be `[2, 3, 4]`.

5. Get the First n Elements:

Write a Python function that extracts the first `n` elements from a list.

- **Requirements:**
  - Define a function `get_first_n(lst, n)` that takes a list `lst` and an integer `n` as input and returns the first `n` elements of the list using slicing.
  - Example: For the input `[1, 2, 3, 4, 5]` with `n=3`, the output should be `[1, 2, 3]`.

6. Extract Elements from the End:

Write a Python function that extracts the last `n` elements of a list using slicing.

- **Requirements:**
  - Define a function `get_last_n(lst, n)` that takes a list `lst` and an integer `n` as input and returns the last `n` elements of the list.
  - Example: For the input `[1, 2, 3, 4, 5]` with `n=2`, the output should be `[4, 5]`.

7. Extract Elements in Reverse Order:

Write a Python function that extracts a list of elements in reverse order starting from the second-to-last element and skipping one element in between.

- **Requirements:**
  - Define a function `reverse_skip(lst)` that takes a list `lst` and returns a new list containing every second element starting from the second-to-last, moving backward.
  - Example: For the input `[1, 2, 3, 4, 5, 6]`, the output should be `[5, 3, 1]`.

4.3 Exercise on Nested List:

1. Flatten a Nested List:

Write a Python function that takes a nested list and flattens it into a single list, where all the elements are in a single dimension.

- **Requirements:**
  - Define a function `flatten(lst)` that takes a nested list `lst` and returns a flattened version of the list.
  - Example: For the input `[[1, 2], [3, 4], [5]]`, the output should be `[1, 2, 3, 4, 5]`.

2. Accessing Nested List Elements:

Write a Python function that extracts a specific element from a nested list given its indices.

- **Requirements:**
  - Define a function `access_nested_element(lst, indices)` that takes a nested list `lst` and a list of indices `indices`, and returns the element at that position.
  - Example: For the input `lst = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]` with `indices = [1, 2]`, the output should be `6`.

3. Sum of All Elements in a Nested List:

Write a Python function that calculates the sum of all the numbers in a nested list (regardless of depth).

- **Requirements:**
  - Define a function `sum_nested(lst)` that takes a nested list `lst` and returns the sum of all the elements.
  - Example: For the input `[[1, 2], [3, [4, 5]], 6]`, the output should be `21`.

4. Remove Specific Element from a Nested List:

Write a Python function that removes all occurrences of a specific element from a nested list.

- **Requirements:**
  - Define a function `remove_element(lst, elem)` that removes `elem` from `lst` and returns the modified list.
  - Example: For the input `lst = [[1, 2], [3, 2], [4, 5]]` and `elem = 2`, the output should be `[[1], [3], [4, 5]]`.

5. Find the Maximum Element in a Nested List:

Write a Python function that finds the maximum element in a nested list (regardless of depth).

- **Requirements:**
  - Define a function `find_max(lst)` that takes a nested list `lst` and returns the maximum element.
  - Example: For the input `[[1, 2], [3, [4, 5]], 6]`, the output should be `6`.

6. Count Occurrences of an Element in a Nested List:

Write a Python function that counts how many times a specific element appears in a nested list.

- **Requirements:**
  - Define a function `count_occurrences(lst, elem)` that counts the occurrences of `elem` in the nested list `lst`.
  - Example: For the input `lst = [[1, 2], [2, 3], [2, 4]]` and `elem = 2`, the output should be `3`.

7. Flatten a List of Lists of Lists:

Write a Python function that flattens a list of lists of lists into a single list, regardless of the depth.

- **Requirements:**
  - Define a function `deep_flatten(lst)` that takes a deeply nested list `lst` and returns a single flattened list.
  - Example: For the input `[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]`, the output should be `[1, 2, 3, 4, 5, 6, 7, 8]`.

8. Nested List Average:

Write a Python function that calculates the average of all elements in a nested list.

- **Requirements:**
  - Define a function `average_nested(lst)` that takes a nested list `lst` and returns the average of all the elements.
  - Example: For the input `[[1, 2], [3, 4], [5, 6]]`, the output should be `3.5`.

5 About a Numpy.

What is Numpy?

NumPy (Numerical Python) is an open-source library that provides fast, pre-compiled functions for mathematical and numerical operations in Python. It serves as a foundational package for scientific computing, enabling efficient manipulation of large arrays and matrices. NumPy was created by **Travis Oliphant** in 2005 as an extension of the older Numeric and Numarray libraries, combining their strengths into a single, more powerful package. Its functionalities are further extended by **SciPy**, which provides additional algorithms for optimization, integration, and signal processing.

Why NumPy?

NumPy is widely used in data science, machine learning, and scientific computing due to its performance and flexibility. Key advantages include:

- **Efficient N-dimensional Array Object (`ndarray`):**
  1. The `ndarray` is NumPy's core data structure, optimized for storing and processing large datasets.
  2. Supports multiple dimensions (`1D, 2D, or higher`), making it suitable for matrices and tensors.
  3. Provides functions like `reshape()` and `flatten()` to modify array structure efficiently.
  4. The `resize()` function allows dynamic adjustments to array sizes.
- **Fast Broadcasting and Vectorized Operations:**
  1. **Broadcasting** allows operations between arrays of different shapes without explicit loops.
  2. **Vectorized operations** enable element-wise computations across entire arrays, eliminating the need for slow Python loops.
  3. These features significantly improve performance by leveraging optimized C and Fortran code under the hood.
- **Comprehensive Mathematical Functions:**
  1. NumPy provides an extensive collection of mathematical, logical, and statistical functions such as `mean(), median(), std(), sum(), min(), max(), log(), exp()`.
  2. The `numpy.linalg` module offers powerful linear algebra tools, including matrix multiplication, dot products, determinants, and eigenvalues.
- **Seamless Integration with Other Libraries:**
  1. NumPy is the backbone of many scientific libraries, such as **Pandas, SciPy, and scikit-learn**, ensuring smooth interoperability.
  2. While NumPy itself does not support GPU acceleration, libraries like **CuPy** offer a nearly identical API for running NumPy operations on NVIDIA GPUs, enabling accelerated computations.

6 Getting Started with NumPy:

6.1 Introduction to Array `ndarray`:

The central feature of Numpy is the array object class. Arrays are similar to lists in Python, except that **every element** of an array must be of the same **data type (dtype)**. Arrays enable operations with large amounts of numeric data to be performed efficiently, making them significantly faster and more memory-efficient than Python lists.

Example: Importing NumPy and Creating Arrays of Zeros:

Importing Numpy and Creating Arrays of Zero.

```
import numpy as np
# initialize an all zero array with size 2 X 3:
zeros_arr = np.zeros((2,3))
print(" A zeros array is \n", zeros_arr, "with dimensions", zeros_arr.shape, "\n")
```

This code initializes a 2×3 array of zeros and prints both the array and its dimensions.

1. NumPy Array - Types:

Numpy arrays can be classified as following based on there dimension or shape:

- **zero - dimensional arrays:** A scalar value represented as a NumPy array with no axes, created using `np.array(5)`.
- **one - dimensional arrays:** A linear array of elements, created using `np.array([1, 2, 3])`, which can represent either a single row or column.
- **Multi-dimensional (n-dimensional) arrays:** Arrays with multiple dimensions (axes). For example, a 2D array can be created using `np.array([[1, 2], [3, 4]])`. This concept extends to higher dimensions as needed.

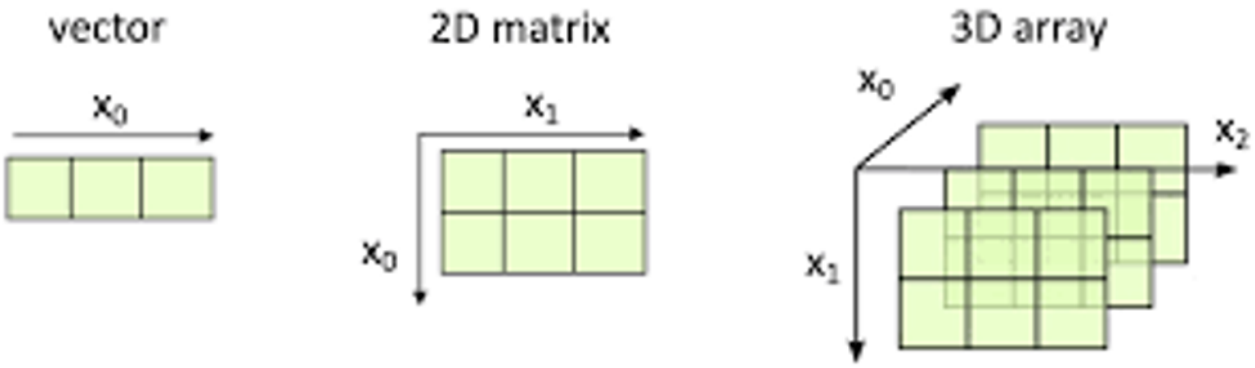


Figure 6: Array of Different Dimensions.

Example: Creating and Displaying Zero, One, and Multi-Dimensional Arrays:

Creating and Displaying Different Type of Arrays.

```
import numpy as np
# Create and display zero, one, and n-dimensional arrays
zero_dim_array = np.array(5)
one_dim_array = np.array([1, 2, 3])
n_dim_array = np.array([[1, 2], [3, 4]])
for arr in [zero_dim_array, one_dim_array, n_dim_array]:
    print(f"Array:\n{arr}\nDimension: {arr.ndim}\nData type: {arr.dtype}\n")
```

This code snippet demonstrates the creation of different types of arrays and prints their values, dimensions, and data types.

6.1.1 Some of the key attributes of `ndarray`:

1. `.ndim` **Number of Dimensions:** Returns the number of dimensions (axes) in the array.

   Example.

   ```
   # For a 1-D array:
   import numpy as np
   arr_1d = np.array([1, 2, 3, 4, 5])
   print(arr_1d.ndim) # Output: 1
   # For a 2-D array:
   arr_2d = np.array([[1, 2, 3],
                      [4, 5, 6]])
   print(arr_2d.ndim) # Output: 2
   #For a 3-D array:
   arr_3d = np.array([[[1, 2, 3],
                       [4, 5, 6]],

                      [[7, 8, 9],
                       [10, 11, 12]]])
   print(arr_3d.ndim) # Output: 3
   ```

2. `.shape` **Shape of the Array:** Returns a tuple representing the size of each dimension.

   Example.

   ```
   print(arr_1d.shape) # Output: (5,)
   print(arr_2d.shape) # Output: (2, 3)
   print(arr_3d.shape) # Output: (2, 2, 3)
   ```

3. `.size` **Total Number of Elements:** Returns the total number of elements in the array.

   Example.

   ```
   print(arr_1d.size) # Output: 5
   print(arr_2d.size) # Output: 6
   print(arr_3d.size) # Output: 12
   ```

4. `.dtype` **Data Type of Elements:** Returns the data type of the array elements.

   Example.

   ```
   print(arr_1d.dtype) # Output: int64 (or int32)
   arr_float = np.array([1.1, 2.2, 3.3])
   print(arr_float.dtype) # Output: float64
   ```

5. `.itemsize` **Size of Each Element in Bytes:** Returns the size (in bytes) of one array element.

Example.

```
print(arr_1d.itemsize) # Output: 8 (for int64)
```

6. `.nbytes` **Total Memory Consumption:** Returns the total memory (in bytes) used by the array.

Example.

```
print(arr_1d.nbytes) # Output: 40 (5 elements \times 8 bytes each for int64)
```

7. `.T` **Transpose of the Array:** Returns the transposed version of the array.

Example.

```
print(arr_2d.T)
# Output:
# [[1 4]
# [2 5]
# [3 6]]
```

8. `.flat` **Flattening the Array:** Returns an iterator over a 1D representation of the array.

Example.

```
for item in arr_2d.flat:
    print(item, end=" ") # Output: 1 2 3 4 5 6
```

9. `.real` **and** `.imag` **- Real and Imaginary Parts:** Used for complex number arrays.

Example.

```
arr_complex = np.array([1+2j, 3+4j])
print(arr_complex.real) # Output: [1. 3.]
print(arr_complex.imag) # Output: [2. 4.]
```

10. `.astype(dtype)` **Convert Data Type:** Converts the array elements to a specified data type.

Example.

```
arr_float = arr_2d.astype(float)
print(arr_float)
# Output:
# [[1. 2. 3.]
# [4. 5. 6.]]
```

6.1.2 Common Errors with Key Attributes

When working with `.shape` and `.size`, users often encounter the following errors:

1. **AttributeError: 'list' object has no attribute 'shape'**
   - **Cause:** `.shape` only works on NumPy arrays, not Python lists.
   - **Fix:** Convert the list to a NumPy array first.

Example - Attribute Error.

```
lst = [[1, 2], [3, 4]]
print(lst.shape)
# Error: List has no .shape attribute
#Fixing the Error:
arr = np.array(lst)
print(arr.shape) # Output: (2, 2)
```

2. **IndexError: tuple index out of range**
   - **Cause:** Trying to access `.shape[x]` where `x` exceeds the number of dimensions.
   - **Fix:** Check the length of `.shape` before accessing.

Example - Index Error.

```
arr = np.array([1, 2, 3])
print(arr.shape[1]) # Error: 1D array only has one dimension
# Fixing the Error
if len(arr.shape) > 1:
    print(arr.shape[1]) # Access only if it's valid
```

3. **TypeError: 'tuple' object is not callable**
   - **Cause:** Using `.shape()` instead of `.shape`.
   - **Fix:** Remove parentheses.

Example - Type Error.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape()) # Error: .shape is an attribute, not a function
print(arr.shape) # Correct usage
```

4. **ValueError: operands could not be broadcast together**
   - **Cause:** Trying to perform operations on arrays with incompatible shapes.
   - **Fix:** Ensure the arrays have compatible dimensions.

Example - ValueError.

```
a = np.array([[1, 2], [3, 4]])
b = np.array([1, 2, 3])
print(a + b) # Error: Shapes (2,2) and (3,) are not compatible
```

5. **MemoryError: Unable to allocate array**

- **Cause:** Attempting to create an excessively large array.
- **Fix:** Use efficient memory management techniques.

Example - MemoryError.

```
large_array = np.ones((100000, 100000)) # May cause MemoryError
```

6.1.3 NumPy `ndarray` - reshape:

The `reshape()` method in NumPy changes the shape of an array without altering its data, as long as the total number of elements remains the same before and after reshaping.

Example:

An array with shape $(2, 3)$ (6 elements) can be reshaped to any shape with 6 elements, like $(3, 2)$ or $(1, 6)$, but reshaping to $(4, 2)$ would raise an error because it requires 8 elements.

Example: Reshaping an Array

```
import numpy as np

# Create an array with shape (2, 3)
array = np.array([[1, 2, 3],
                  [4, 5, 6]])

# Reshape to (3, 2), keeping 6 elements
reshaped_array = array.reshape(3, 2)

print("Original Shape:", array.shape, "\nReshaped Shape:", reshaped_array.shape)
# Expected Outputs:
# Original Shape: (2, 3)
# Reshaped Shape: (3, 2)
```

The `reshape()` method in NumPy changes the shape of an array without altering its data, as long as the total number of elements remains the same before and after reshaping. This makes it a powerful tool for manipulating the structure of data without losing any of the underlying information.

Example:

An array with shape 2×3 i.e. **6 elements** can be reshaped to any shape with **6 elements**,like 3×2 or 1×6, but reshaping to 4×2 would raise an error because it requires **8 elements** which does not match the original **6 element.**

Potential Errors:

ValueError: If the reshaped array doesn't match the total number of elements in the original array, NumPy will raise a `ValueError`. Example:

.reshape - ValueError.

```
array = np.array([[1, 2, 3], [4, 5, 6]])
reshaped_array = array.reshape(4, 2)
# Error: cannot reshape array of size 6 into shape (4, 2)
```

7 How to Create an Array {ndarray}?

- **Using in-built function - Arrays with Evenly spaced values:**
  - `arange`: The syntax of arange is `arange([start,]stop[,step],[,dtype=None])`

Example - arange

```
import numpy as np
a = np.arange(1, 10)
print(a)
x = range(1, 10)
print(x) # x is an iterator
print(list(x))
# further arange examples:
x = np.arange(10.4)
print(x)
x = np.arange(0.5, 10.4, 0.8)
print(x)
```

  - `linspace`: The syntax is `linpspace(start,stop,num=50,endpoint=True,retstep=False)`

Example - linspace

```
import numpy as np
# 50 values between 1 and 10:
print(np.linspace(1, 10))
# 7 values between 1 and 10:
print(np.linspace(1, 10, 7))
# excluding the endpoint:
print(np.linspace(1, 10, 7, endpoint=False))
```

- **Creating or Initializing with Ones, Zeros and Empty:**
  - `np.ones`: Creates an array of ones as specified by shape.The syntax is: `np.ones(shape, dtype = None)`
  - `np.zeros`: Creates an array of zeros as specified by `shape`.The syntax is: `np.zeros(shape, dtype=None)`
  - `np.empty`: Creates an array of uninitialized values of the specified `shape`.The syntax is: `np.empty(shape, dtype=None)`.
  - `np.eye(N)`: Creates an identity matrix of `shape (N, N)` filled with ones on the diagonal and zeros elsewhere.

Example - Initalizing an Arrays.

```
import numpy as np
# Create arrays of specified shapes
ones_array = np.ones((2, 3)) # Shape: (2, 3)
zeros_array = np.zeros((3, 2)) # Shape: (3, 2)
empty_array = np.empty((2, 2)) # Shape: (2, 2)
identity_matrix = np.eye(3) # Shape: (3, 3)
print(ones_array, zeros_array, empty_array, identity_matrix, sep='\n\n')
```

- **By Manipulating Existing Array:**
  - `np.array`: It can be used to create an array from any array-like object, such as lists, tuples, or other arrays. This method converts the input to a NumPy array if it isn't one already. Example:

Example - Using ndarray.

```
import numpy as np
array_from_list = np.array([1, 2, 3]) # [1 2 3]
array_from_tuple = np.array((4, 5, 6)) # [4 5 6]
array_from_nested_list = np.array([[1, 2, 3], [4, 5, 6]]) # [[1 2 3] [4 5 6]]
print(array_from_list, array_from_tuple, array_from_nested_list, sep='\n')
```

  - **Using shape of an existing array.**
    - `np.zeros(existing_array.shape)`: Creates an array of zeros with the same shape as existing array.
    - `np.ones(existing_array.shape)` : Creates an array of ones with the same shape.
    - `np.empty(existing_array.shape)`: Creates an uninitialized array with the same shape, which may contain random values.
    - Example:

Example - Using shape of an existing array.

```
import numpy as np
# Existing Array of Shape:(2,3)
arr = np.array([[1, 2, 3], [4, 5, 6]])
# Creating Array with Shape of existing array:
zeros, ones, empty = np.zeros(arr.shape), np.ones(arr.shape), np.empty(arr.shape) #
    Shape: (2,3)
print(arr, zeros, ones, empty, sep='\n')
```

- **By Combining or joining array:**
  - **Horizontal Stacking:** The `np.hstack()` function stacks arrays horizontally (side by side). All arrays must have the **same number of rows**.
  - **Vertical Stacking:** The `np.vstack()` function stacks arrays vertically (on top of each other). All arrays must have the **same number of columns**.

Example - Implementing `np.hstack()` and `np.vstack()`

```
import numpy as np
arr1 = np.array([[1, 2], [3, 4]]) # Shape: (2, 2)
arr2 = np.array([[5, 6], [7, 8]]) # Shape: (2, 2)
# Stacking examples
hstacked = np.hstack((arr1, arr2))
# Output hstack: [[1 2 5 6] [3 4 7 8]]
vstacked = np.vstack((arr1, arr2))
# Output vstack[[1 2] [3 4] [5 6] [7 8]]
colstacked = np.column_stack((arr1[0], arr2[0]))
# Output column stack: [[1 5] [2 6]]
print(hstacked, vstacked, colstacked, sep='\n')
```

- **concatenation:** `np.concatenate((array1, array2), axis=0)` joins arrays along a specified axis.
  - `axis = 0`: Concatenate along the rows (vertical stacking).This means the arrays are stacked on top of each other. The number of columns must match across the arrays.
  - `axis = 1`: Concatenate along the column (horizontal stacking).This means the arrays are placed side by side. The number of rows must match across the arrays.
  - `axis = 2`: Concatenate along the third dimension (depth stacking). This is typically used for 3D arrays or higher-dimensional arrays, where the number of rows and columns must match, but new depth layers are added.
  - Example:

Example - Implementing np.concatenate()

```
import numpy as np

# Creating two 3D arrays
arr1 = np.array([[[1], [2]], [[3], [4]]]) # Shape: (2, 2, 1)
arr2 = np.array([[[5], [6]], [[7], [8]]]) # Shape: (2, 2, 1)
# Concatenate along axis 0 (vertical stacking)
concat_axis0 = np.concatenate((arr1, arr2), axis=0)
# Shape of concat_axis0: (4, 2, 1), arr1 is stacked on top of arr2
# Concatenate along axis 1 (horizontal stacking)
concat_axis1 = np.concatenate((arr1, arr2), axis=1)
# Shape of concat_axis1: (2, 4, 1), arr1 and arr2 are placed side by side
# Concatenate along axis 2 (depth stacking)
concat_axis2 = np.concatenate((arr1, arr2), axis=2)
# Shape of concat_axis2: (2, 2, 2), arr1 and arr2 are stacked in depth (third
    dimension)
# Displaying the results
print("Concatenation along axis 0 (Vertical stacking):\n", concat_axis0)
print("Shape of concatenated array along axis 0:", concat_axis0.shape)
print("\nConcatenation along axis 1 (Horizontal stacking):\n", concat_axis1)
print("Shape of concatenated array along axis 1:", concat_axis1.shape)
print("\nConcatenation along axis 2 (Depth stacking):\n", concat_axis2)
print("Shape of concatenated array along axis 2:", concat_axis2.shape)
```

7.1 Visualizing Array Combination Operations:

Array combination - whether through stacking or concatenation operations—is one of the most fundamental tasks in deep learning programming. These operations are frequently used to manipulate data, combine feature vectors, or merge different layers in neural networks. Mastery of these operations is crucial for efficient data handling and model design.Understanding the nuances of operations like `np.hstack(), np.vstack(), and np.concatenate()` will empower you to structure your data effectively for model input or layer construction.Above, we provide examples and sample code implementations to clarify the differences and applications of each operation.Below we provide an image to offer a more intuitive understanding of these operations.{Images are by Joshua Ebner.}

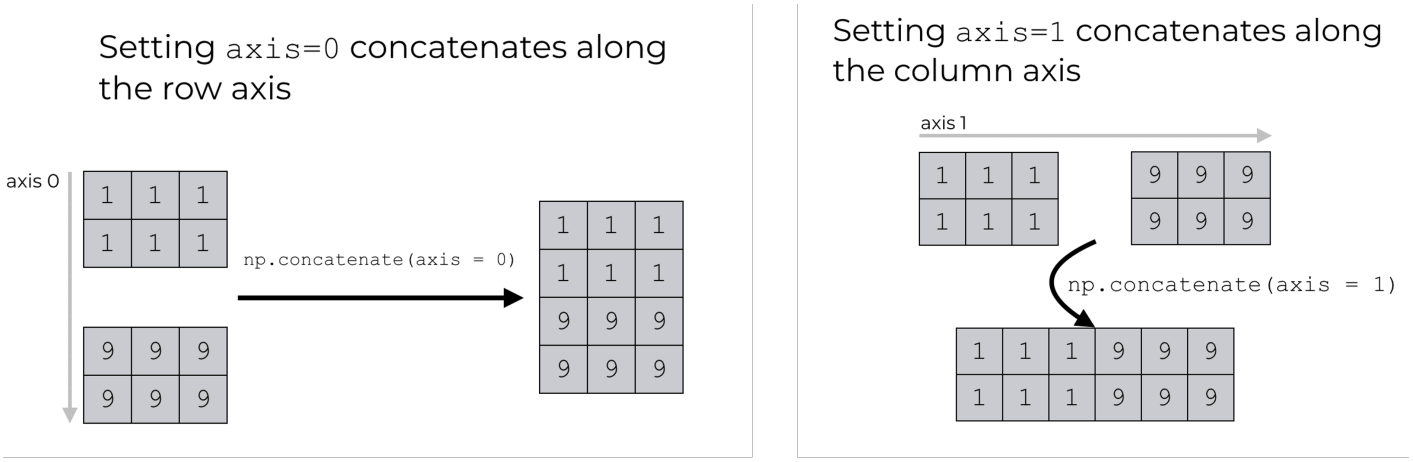


Image Stacking or Concatenation Visualizations.

Image By: Joshua Ebner.

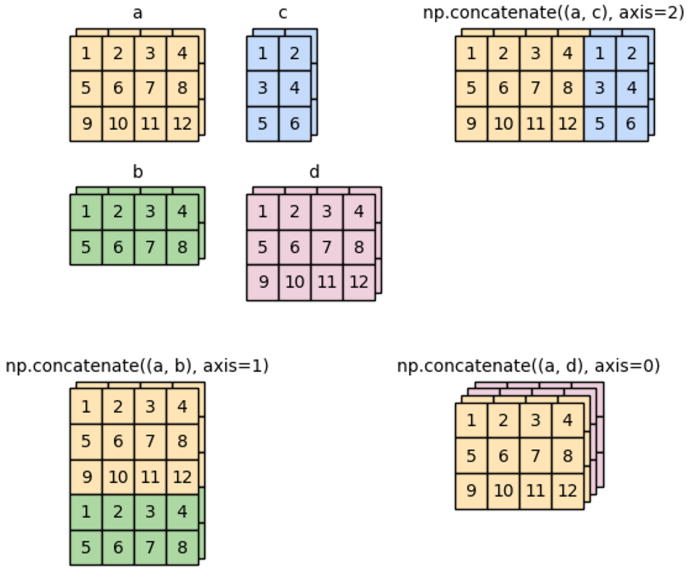


Image Concatenation along depth i.e. axis = 2.

7.2 Key Errors in Array Combining Operations: Causes and Solutions:

In this section, we explore common errors encountered during NumPy array stacking and concatenation operations. We focus on understanding the causes of these errors and provide practical solutions to resolve them. By the end, you'll have a clear understanding of how to avoid these issues and perform array manipulations effectively in NumPy.

1. Dimension Mismatch:

- **Error:**

Error - Dimension Mismatch.

```
ValueError: all the input arrays must have same number of dimensions
```

- **Cause:** This error occurs when the arrays you are trying to stack or concatenate do not have the same number of dimensions or have incompatible shapes.
  - For `np.hstack()`: All arrays must have the same number of rows (i.e., the same first dimension).
  - For `np.vstack()`: All arrays must have the same number of columns (i.e., the same second dimension).
  - For `np.concatenate()`: The arrays must have the same number of dimensions, and the dimensions other than the one along which you are concatenating must match.
- **Solution:** Make sure the dimensions of the arrays align properly. Check that the arrays have the correct shape (e.g., same number of rows for horizontal stacking, or same number of columns for vertical stacking). Example:

Dimension Mismatch Sample Solution.

```
import numpy as np
# Incorrect shapes
arr1 = np.array([[1, 2], [3, 4]]) # shape (2, 2)
arr2 = np.array([5, 6]) # shape (2,)
# Attempting horizontal stack
try:
    hstacked = np.hstack((arr1, arr2)) # Error due to dimension mismatch
except ValueError as e:
    print(f"Error: {e}")
# Solution: Reshape arr2 to match arr1's dimensions
arr2 = np.array([5, 6]).reshape(2, 1) # shape (2, 1)
hstacked = np.hstack((arr1, arr2)) # No error
print(hstacked)
```

2. Shape Misalignment for Stacking:

- **Error:**

Error - Shape Misalignment for Stacking.

```
ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

- **Cause:** This error occurs when the arrays you're trying to stack have incompatible shapes for the specific type of stacking.
  - For `np.hstack()`: The number of rows (dimension 0) must be the same.
  - For `np.vstack()`: The number of columns (dimension 1) must be the same.
  - For `np.concatenate()`: The shape must align for the dimension other than the one you're concatenating along.
- **Solution:** Ensure the number of rows or columns match the stacking direction. Example:

Shape Misalignment for Stacking Sample Solution.

```
import numpy as np
arr1 = np.array([[1, 2, 3], [4, 5, 6]]) # shape (2, 3)
arr2 = np.array([[7, 8], [9, 10]]) # shape (2, 2)
# Attempting horizontal stack (this will fail)
try:
    hstacked = np.hstack((arr1, arr2)) # Error due to shape mismatch
except ValueError as e:
    print(f"Error: {e}")
# Solution: Make sure the number of rows in both arrays is the same before stacking
arr2 = np.array([[7, 8, 9], [10, 11, 12]]) # shape (2, 3)
hstacked = np.hstack((arr1, arr2)) # Works fine
print(hstacked)
```

3. Inconsistent Data Types:

- **Error:**

Error - Inconsistent Data Types.

```
TypeError: Cannot interpret '-1' as a data type
```

- **Cause:** This error occurs if there's a mismatch in the data types of the arrays you're trying to stack or concatenate. If one array is of type `int` and another is of type `float`, NumPy will try to cast them, but sometimes it may not be able to do so automatically.
- **Solution:** Ensure that all arrays involved in stacking or concatenation have the same or compatible data types. Example:

Inconsistent Data Types Sample Solution.

```
arr1 = np.array([[1, 2], [3, 4]], dtype=int)
arr2 = np.array([5, 6], dtype=float)
# Attempting horizontal stack (this will work, but can lead to unexpected results)
hstacked = np.hstack((arr1, arr2))
print(hstacked)
# Solution: Convert the data types to match each other explicitly, if needed:
```

```
arr2 = np.array([5, 6], dtype=int) # Match the type of arr1
hstacked = np.hstack((arr1, arr2)) # Works fine
print(hstacked)
```

4. Too Many Dimensions in the Arrays:

- **Error:**

Error - Too Many Dimensions in the Arrays.

```
ValueError: setting an array element with a sequence
```

- **Cause:** This error arises if one or more arrays are too "nested" (have more than two dimensions) when trying to stack them.
- **Solution:** Ensure that the arrays are 1D or 2D (for proper horizontal or vertical stacking). If you're trying to stack higher-dimensional arrays, reshape them appropriately. Example:

Too Many Dimensions in the Arrays Sample Solution.

```
arr1 = np.array([[[1, 2], [3, 4]]]) # shape (1, 2, 2)
arr2 = np.array([[5, 6], [7, 8]]) # shape (2, 2)
# Attempting stacking with 3D and 2D arrays
try:
    vstacked = np.vstack((arr1, arr2)) # Error due to too many dimensions
except ValueError as e:
    print(f"Error: {e}")
# Solution: Reshape arr1 to be 2D (or flatten it) before stacking
arr1 = arr1.reshape(2, 2) # Now it's shape (2, 2)
vstacked = np.vstack((arr1, arr2)) # Works fine
print(vstacked)
```

5. Memory Error:

- **Error:**

Error - Memory Error.

```
MemoryError
```

- **Cause:** A `MemoryError` can occur if the arrays you're trying to stack or concatenate are too large, leading to memory constraints, especially when working with very large datasets.
- **Solution:** Consider breaking down large arrays into smaller ones or using a memory-efficient approach, such as sparse matrices or chunking the data into batches.

8 Array Slicing and Indexing.

Indexing allows you to access specific elements or sets of elements from an array based on their position or index. The indexing rules vary depending on the dimensions of the array:

1. **1D Array Indexing:** Access elements using a single index. You can use both positive and negative indexing. For example: `array[0]` for the first element, or `array[-1]` for the last element.
   - **Slicing:** You can extract a range of elements using slicing. Example: `array[start:end]`.

Technique	Syntax	Explanation
1D Arrays		
Basic Slicing	arr[start:stop]	Extracts elements in the specified range along one or more axes.
Slicing with Step	arr[start:stop:step]	Skips elements in the specified range by the given step size.
Slicing Multiple Axes	N/A	Slices multiple dimensions (rows, columns, etc.) simultaneously.
Negative Indices	arr[-n:]	Allows indexing from the end of the array (negative indices).
Slicing Entire Axis	arr[:]	Extracts all elements along one axis or dimension (entire array or sub-array).

Figure 7: Slicing 1-D Array

2. **2D Array Indexing:** Access elements using two indices: one for the row and one for the column. For example: `array[0, 1]` accesses the element in the first row and second column.
   - **Slicing:** Use the colon notation to slice rows and columns. Example: `array[start_row:end_row, start_col:end_col]`.

2D Arrays		
Basic Slicing	arr[start_row : end_row, start_col : end_ col]	Extracts elements in the specified range along one or more axes.
Slicing with Step	arr[start_row : end_row : step1, start_col : end_col : step2]	Skips elements in the specified range by the given step size.
Slicing Multiple Axes	arr[start_row : end_row, start_col : end_ col]	Slices multiple dimensions (rows, columns, etc.) simultaneously.
Negative Indices	arr[-n_rows:, -n_cols:]	Allows indexing from the end of the array (negative indices).
Slicing Entire Axis	arr[:, start_col:end_col]	Extracts all elements along one axis or dimension (entire array or sub-array).
Technique	Syntax	Explanation

Figure 8: Slicing 2-D Array

3. **3D Array Indexing:** For three-dimensional arrays, you use three indices: width, row, and column. Example: `array[0, 1, 2]` accesses the element at the first width, second row, and third column.

    - **Slicing:** You can slice each dimension using the colon notation.
    Example: `array[start_width:end_width, start_row:end_row, start_col:end_col]`.

3D Arrays		
Basic Slicing	arr[start_depth:end_depth, start_row:end_row, start_col:end_col]	Extracts elements in the specified range along one or more axes.
Slicing with Step	arr[start_depth:end_depth:step1, start_row:end_row:step2, start_col:end_col:step3]	Skips elements in the specified range by the given step size.
Slicing Multiple Axes	arr[start_depth:end_depth, start_row:end_row, start_col:end_col]	Slices multiple dimensions (rows, columns, etc.) simultaneously.
Negative Indices	arr[-n_depth:, -n_rows:, -n_cols:]	Allows indexing from the end of the array (negative indices).
Slicing Entire Axis	arr[:, :, start_col:end_col]	Extracts all elements along one axis or dimension (entire array or sub-array).

Figure 9: Slicing 3-D Array

4. **Advanced Indexing:**

    - **Boolean Indexing:** You can use boolean arrays to index elements that satisfy a condition. Example: `array[array > 5]`.
    - **Integer Indexing:** You can index with arrays of integers to select specific elements. Example: `array[[0, 2], [1, 3]]` selects specific row-column pairs.

Example Code:

Array Indexing and Slicing.

```
import numpy as np
# Creating a 2D array
array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Accessing elements using indexing
print(array[0, 1]) # Output: 2 (first row, second column)
print(array[-1, -1]) # Output: 9 (last row, last column)
# Slicing the array
print(array[0:2, 1:3]) # Output: [[2, 3], [5, 6]]
```

8.1 Index Out of Bounds Error:

Error - Index Out of Bound Error.

```
import numpy as np
# Creating a 2D array
array = np.array([[1, 2, 3], [4, 5, 6]])
# Attempting to access an element out of bounds
try:
    print(array[2, 1])
except IndexError as e:
    print(f"Error: {e}")
```

The Out will be:

Error - Index Out of Bound Error.

```
# The Output look like:
Error: index 2 is out of bounds for axis 0 with size 2
```

What does the Error mean?

- The array array has a shape of (2, 3), meaning it has 2 rows and 3 columns.
- When trying to access array[2, 1], you are requesting the third row (index 2), which does not exist (valid row indices are 0 and 1).
- This raises an IndexError, as you are trying to access an index outside the valid range of the array.

9 Array Mathematics with NumPy.

Numpy is the core library for numerical computing in Python, and its strength lies in its support for a variety of mathematical operations. At the heart of these capabilities are Universal Functions (ufuncs), Linear Algebra Operations, and efficient broadcasting and vectorization techniques. These allow for high-performance computations, especially when working with large datasets or complex mathematical models.

9.1 Universal Functions{ufuncs}:

In Numpy, **Universal Functions (ufuncs)** are functions that operate **element-wise** on arrays. They are optimized for performance and allow you to perform mathematical operations across entire arrays without the need for explicit loops.

Key Features of ufuncs:

- **Element-wise operations:** Ufuncs apply a function to every element of an array.
- **Performance:** Ufuncs are implemented in C, making them faster than Python loops.
- **Compatibility:** Ufuncs work with arrays of any shape.

Example - General ufuncs:

Example - Universal Functions.

```
import numpy as np
# Example of basic ufuncs: square and square root
arr = np.array([1, 4, 9, 16])
# Square every element
squared = np.square(arr)
# Square root of every element
sqrt = np.sqrt(arr)
print(f"Squared: {squared}")
print(f"Square root: {sqrt}")
```

Mathematical ufuncs:

Numpy provides a rich set of ufuncs areeee for operations such as:

- Trigonometric functions (np.sin, np.cos, np.tan)
- Exponentiation (np.exp, np.log)
- Absolute value (np.abs)
- Rounding (np.floor, np.ceil)

Category	Function	Description
Arithmetic	`np.add(x, y)`	Element-wise addition
	`np.subtract(x, y)`	Element-wise subtraction
	`np.multiply(x, y)`	Element-wise multiplication
	`np.divide(x, y)`	Element-wise division
	`np.mod(x, y)`	Element-wise modulus
	`np.power(x, y)`	Element-wise exponentiation
Trigonometric	`np.sin(x)`	Sine of each element
	`np.cos(x)`	Cosine of each element
	`np.tan(x)`	Tangent of each element
	`np.arcsin(x)`	Inverse sine
	`np.arccos(x)`	Inverse cosine
	`np.arctan(x)`	Inverse tangent
Exponential and Logarithmic	`np.exp(x)`	Exponential of each element (e^x)
	`np.log(x)`	Natural logarithm ($\ln x$)
	`np.log10(x)`	Base-10 logarithm
	`np.log2(x)`	Base-2 logarithm
Rounding	`np.floor(x)`	Rounds down to the nearest integer
	`np.ceil(x)`	Rounds up to the nearest integer
	`np.round(x)`	Rounds to the nearest integer
Comparison	`np.maximum(x, y)`	Element-wise maximum
	`np.minimum(x, y)`	Element-wise minimum
	`np.equal(x, y)`	Element-wise equality check
	`np.greater(x, y)`	Element-wise greater-than check
	`np.less(x, y)`	Element-wise less-than check

Table 5: Key Universal Functions (ufuncs) in NumPy

Example - Mathematical ufuncs:

Example - Mathematical ufuncs.

```
import numpy as np
# Trigonometric ufuncs
angles = np.array([0, np.pi / 4, np.pi / 2])
sine_values = np.sin(angles)
# Logarithmic ufuncs
log_values = np.log(arr)
```

These functions are vectorized, meaning they automatically apply the operation to each element of the array without the need for explicit loops.

9.2 Linear Algebra with `np.linalg`.

Numpy's **linalg** module provides powerful tools for performing linear algebra operations such as matrix multiplication, solving systems of linear equations, computing determinants, and eigenvalues.

Some Key Vector and Matrix Operations with `np.linalg.` module.

Basic Operations

Function	Description
`np.linalg.norm(x)`	Computes the norm (magnitude) of a vector or matrix.
`np.linalg.det(A)`	Computes the determinant of a square matrix.
`np.linalg.inv(A)`	Computes the inverse of a square matrix.

Solving Linear Systems

Function	Description
`np.linalg.solve(A, b)`	Solves the system of linear equations $Ax = b$.
`np.linalg.lstsq(A, b)`	Computes the least-squares solution of $Ax = b$.
`np.linalg.cond(A)`	Computes the condition number of a matrix.

Eigenvalues and Eigenvectors

Function	Description
`np.linalg.eig(A)`	Computes the eigenvalues and eigenvectors of a square matrix.
`np.linalg.eigvals(A)`	Computes only the eigenvalues of a square matrix.
`np.linalg.eigh(A)`	Computes eigenvalues and eigenvectors of a Hermitian matrix.

Singular Value Decomposition (SVD)

Function	Description
`np.linalg.svd(A)`	Computes the singular value decomposition of a matrix.
`np.linalg.pinv(A)`	Computes the Moore-Penrose pseudo-inverse of a matrix.
`np.linalg.matrix_rank(A)`	Computes the rank of a matrix.

Matrix Factorization

Function	Description
`np.linalg.cholesky(A)`	Computes the Cholesky decomposition of a positive-definite matrix.
`np.linalg.qr(A)`	Computes the QR decomposition of a matrix.
`np.linalg.lu(A)`	Computes the LU decomposition (requires SciPy).

Example - Operation with `np.linalg`.

Sample Use of np.linalg

```
import numpy as np
# Matrix multiplication
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
product = np.dot(A, B)
# Compute determinant
det = np.linalg.det(A)
# Compute inverse
inverse = np.linalg.inv(A)
# Compute eigenvalues and eigenvectors
eigvals, eigvecs = np.linalg.eig(A)
print(f"Matrix Product:\n{product}")
print(f"Determinant: {det}")
print(f"Inverse:\n{inverse}")
print(f"Eigenvalues: {eigvals}, Eigenvectors:\n{eigvecs}")
```

9.3 Broadcasting:

Broadcasting is a powerful concept in Numpy that allows arrays of different shapes to be used together in arithmetic operations. Broadcasting automatically "stretches" the smaller array across the larger array without the need for explicit replication, making the operations more memory-efficient.
Broadcasting allows vectorized operations between arrays of different shapes, provided their dimensions are compatible according to NumPy's broadcasting rules.

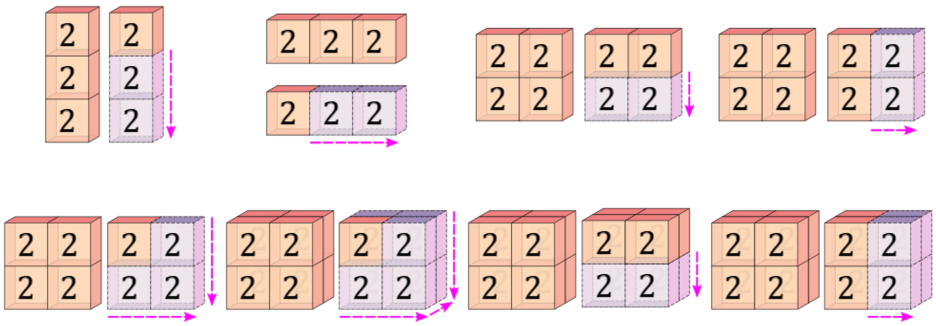
Rules of Broadcasting:

- If the arrays have different numbers of dimensions, the shape of the smaller-dimensional array is padded with ones on the left.
- If the size of the dimension of the larger array is not equal to the size of the smaller array, the smaller array is stretched to match the larger one.

Cautions!!!

Broadcasting does not work for all mismatched dimensions; it follows specific rules to expand the smaller array's dimensions without copying data. Two dimensions are compatible for broadcasting if:

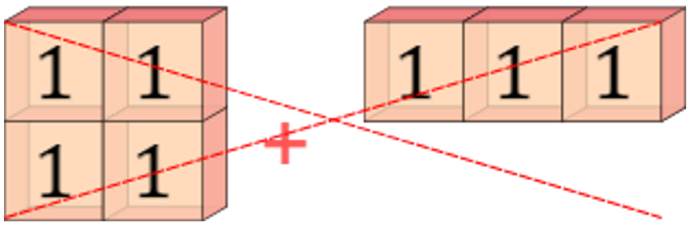
1. They are Equal or
2. One of them is 1 (which allows expansion).



Valid Broadcasting Operations.

Image by Pablo's blog on NumPy Fundamentals.

If neither of these conditions is met for any dimension, broadcasting fails and NumPy raises a ValueError.



Invalid Broadcasting Operations.

Image by Pablo's blog on NumPy Fundamentals.

Example - Valid Broadcasting Operation:

Valid Broadcasting - Column Matches.

```
import numpy as np
A = np.array([[1], [2], [3]]) # Shape (3,1) 3 rows, 1 column
B = np.array([10, 20, 30]) # Shape (3,) 1D array (3 elements)

C = A + B # Works! Result shape: (3,3)
print(C)
```

Why does this work?

- A is (3,1) and B is (3,) →NumPy treats B as (1,3)
- A expands to (3,3) by repeating its single column across all columns.

Valid Broadcasting - Row Matches.

```
A = np.array([[1, 2, 3]]) # Shape (1,3)
B = np.array([[4], [5], [6]]) # Shape (3,1)

C = A + B #Works! Result shape: (3,3)
print(C)
```

Why does this work?

- A expands from (1,3) to (3,3) (rows copied).
- B expands from (3,1) to (3,3) (columns copied).

Example - InValid Broadcasting Operation:

InValid Broadcasting - Dimension Mismatch.

```
A = np.ones((3,2)) # Shape (3,2)
B = np.ones((2,3)) # Shape (2,3)
C = A + B #ValueError: operands could not be broadcast together
```

Why does this fail?

- A has 3 rows, 2 columns, B has 2 rows, 3 columns.
- Neither the rows nor the columns are compatible.

Key Takeaways:

- Rows or columns don't need to match exactly, but they must be broadcastable i.e. one dimension must be equal or 1 to be expanded.
- NumPy expands dimensions from the rightmost axis.
- If neither rows nor columns match, broadcasting fails.

9.4 Vectorization:

Vectorization is the process of converting an operation that uses explicit loops into an operation that uses Numpy's array operations, which are optimized for performance. Vectorized operations are typically much faster than their loop-based equivalents.

Why Vectorization?

- **Efficiency:** It eliminates the need for loops, as Numpy performs the computation internally in compiled code.
- **Readability:** Code becomes more concise and easier to understand.
- **Memory efficiency:** Vectorized operations do not require the creation of additional arrays, as they work in place.

In layman's terms, vectorization speeds up mathematical operations by eliminating explicit loops and leveraging optimized array computations. Let's verify this with an example.

Python Loop vs. NumPy Vectorizations.

```
import numpy as np
import time
# Create two large arrays
size = 10**6
a = np.random.rand(size)
b = np.random.rand(size)
# Using a loop
start_time = time.time()
result_loop = np.zeros(size)
for i in range(size):
    result_loop[i] = a[i] + b[i]
end_time = time.time()
loop_time = end_time - start_time
print(f"Loop Execution Time: {loop_time:.5f} seconds")
# Using NumPy's vectorized operations
start_time = time.time()
result_vectorized = a + b # Element-wise addition
end_time = time.time()
vectorized_time = end_time - start_time
print(f"Vectorized Execution Time: {vectorized_time:.5f} seconds")
# Performance improvement factor
speedup = loop_time / vectorized_time
print(f"Vectorization is approximately {speedup:.2f} times faster!")
```

Expected Outcome.

```
Loop Execution Time: 0.52034 seconds
Vectorized Execution Time: 0.00325 seconds
```

Vectorization is approximately 160.10 times faster!

10 To - Do - NumPy

Please complete all the problems listed below:

10.1 Basic Vector and Matrix Operation with Numpy.

Problem - 1: Array Creation:

Complete the following Tasks:

1. Initialize an empty array with size $2X2$.
2. Initialize an all one array with size $4X2$.
3. Return a new array of given shape and type, filled with fill_value.{Hint: np.full}
4. Return a new array of zeros with same shape and type as a given array.{Hint: np.zeros_like}
5. Return a new array of ones with same shape and type as a given array.{Hint: np.ones_like}
6. For an existing list `new_list = [1,2,3,4]` convert to an numpy array.{Hint: np.array()}

Problem - 2: Array Manipulation: Numerical Ranges and Array indexing:

Complete the following tasks:

1. Create an array with values ranging from 10 to 49. {Hint:np.arrange()}.
2. Create a $3X3$ matrix with values ranging from 0 to 8.
   {Hint:look for np.reshape()}
3. Create a $3X3$ identity matrix.{Hint:np.eye()}
4. Create a random array of size 30 and find the mean of the array.
   {Hint:check for np.random.random() and array.mean() function}
5. Create a $10X10$ array with random values and find the minimum and maximum values.
6. Create a zero array of size 10 and replace 5^{th} element with 1.
7. Reverse an array `arr = [1,2,0,0,4,0]`.
8. Create a 2d array with 1 on border and 0 inside.
9. Create a $8X8$ matrix and fill it with a checkerboard pattern.

Problem - 3: Array Operations:

For the following arrays:
`x = np.array([[1,2],[3,5]])` and `y = np.array([[5,6],[7,8]])`;
`v = np.array([9,10])` and `w = np.array([11,12])`;
Complete all the task using numpy:

1. Add the two array.
2. Subtract the two array.
3. Multiply the array with any integers of your choice.
4. Find the square of each element of the array.
5. Find the dot product between: `v(and)w` ; `x(and)v` ; `x(and)y`.
6. Concatenate `x(and)y` along row and Concatenate `v(and)w` along column.
   {Hint:try `np.concatenate()` or `np.vstack()` functions.
7. Concatenate `x(and)v`; if you get an error, observe and explain why did you get the error?

Problem - 4: Matrix Operations:

- For the following arrays:
  `A = np.array([[3,4],[7,8]])` and `B = np.array([[5,3],[2,1]])`;
  Prove following with Numpy:
  1. Prove $A.A^{-1} = I$.
  2. Prove $AB \neq BA$.
  3. Prove $(AB)^T = B^T A^T$.
- Solve the following system of Linear equation using Inverse Methods.

$$\begin{aligned} 2x - 3y + z &= -1 \\ x - y + 2z &= -3 \\ 3x + y - z &= 9 \end{aligned}$$

  {Hint: First use Numpy array to represent the equation in Matrix form. Then Solve for: $AX = B$}
- Now: solve the above equation using `np.linalg.inv` function.{Explore more about "linalg" function of Numpy}

10.2 Experiment: How Fast is Numpy?

In this exercise, you will compare the performance and implementation of operations using plain Python lists (arrays) and NumPy arrays. Follow the instructions:

1. Element-wise Addition:
    - Using **Python Lists**, perform element-wise addition of two lists of size $1,000,000$. Measure and Print the time taken for this operation.
    - Using **Numpy Arrays**, Repeat the calculation and measure and print the time taken for this operation.
2. Element-wise Multiplication
    - Using **Python Lists**, perform element-wise multiplication of two lists of size $1,000,000$. Measure and Print the time taken for this operation.
    - Using **Numpy Arrays**, Repeat the calculation and measure and print the time taken for this operation.
3. Dot Product
    - Using **Python Lists**, compute the dot product of two lists of size $1,000,000$. Measure and Print the time taken for this operation.
    - Using **Numpy Arrays**, Repeat the calculation and measure and print the time taken for this operation.
4. Matrix Multiplication
    - Using **Python lists**, perform matrix multiplication of two matrices of size 1000x1000. Measure and print the time taken for this operation.
    - Using **NumPy arrays**, perform matrix multiplication of two matrices of size 1000x1000. Measure and print the time taken for this operation.

Fasten Your Seat Belt and Enjoy the Ride.