1. Explain the purpose and advantages of NumPy in scientific computing and data analysis. How does it enhance Python's capabilities for numerical operations?

Answer:

**Purpose of NumPy:**

**Efficient Array Computations:** NumPy provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently.

**Foundation for Scientific Libraries**: It serves as the foundation for many other scientific libraries in Python, such as SciPy, Pandas, and Matplotlib, making it essential for scientific computing.

**Data Analysis:** NumPy is widely used in data analysis for its ability to handle large datasets and perform complex calculations quickly.

**Advantages of NumPy**

**Performance:** NumPy arrays are more efficient than Python lists because they are implemented in C and use contiguous memory blocks, which leads to faster execution of operations.

**Vectorization:** It supports vectorized operations, which means you can perform element-wise operations on arrays without using explicit loops, leading to more concise and readable code.

**Broadcasting:** NumPy’s broadcasting feature allows you to perform arithmetic operations on arrays of different shapes, making it easier to write and understand code.

**Comprehensive Functions:** It includes a wide range of mathematical, statistical, and linear algebra functions, which are optimized for performance.

**Integration with Other Libraries:** NumPy integrates seamlessly with other scientific libraries, enhancing Python’s capabilities for numerical and scientific computing.

**Enhancing Python’s Capabilities**

**Memory Efficiency:** NumPy arrays consume less memory compared to Python lists, especially for large datasets.

**Speed**: Operations on NumPy arrays are executed much faster due to the optimized C implementation.

**Ease of Use:** The syntax and functions provided by NumPy are designed to be intuitive and easy to use, making it accessible for both beginners and experienced programmers.

**Interoperability:** NumPy arrays can be used with a variety of other Python libraries, facilitating a smooth workflow for data analysis and scientific computing.

1. Compare and contrast np.mean() and np.average() functions in NumPy. When would you use one over the other?

Answer:

|  |  |
| --- | --- |
| **np.mean()** | **np.average()** |
| Purpose: Computes the arithmetic mean (average) of the elements along the specified axis. | Purpose: Computes the weighted average of the elements along the specified axis. |
| Syntax: np.mean(a, axis=None, dtype=None, out=None, keepdims=<no value>) | Syntax: np.average(a, axis=None, weights=None, returned=False) |
| Parameters:  a: Input array.  axis: Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.  dtype: Data type used in computing the mean.  out: Alternative output array in which to place the result.  keepdims: If True, the reduced axes are left in the result as dimensions with size one. | Parameters:  a: Input array.  axis: Axis or axes along which to average a. The default is to average over the flattened array.  weights: An array of weights associated with the values in a. Each value in a contributes to the average according to its associated weight.  returned: If True, the sum of the weights is returned along with the average. |
| Use Case: Use np.mean() when you need a straightforward calculation of the mean without considering weights. | Use Case: Use np.average() when you need to compute a weighted average, where different elements contribute differently to the final average. |
| np.mean() does not support weights; it calculates the simple arithmetic mean. | np.average() supports weights, allowing for the calculation of a weighted mean. |
| np.mean() returns only the mean. | np.average() can return both the weighted average and the sum of the weights if the returned parameter is set to True. |
| Use np.mean():  When you need a simple average of the array elements.  When weights are not a factor in your calculation. | Use np.average():  When you need to calculate a weighted average.  When different elements in your array have different levels of importance or frequency. |
| Eg: import numpy as np  # Example array  data = np.array([1, 2, 3, 4, 5])  # Using np.mean()  mean\_value = np.mean(data)  print("Mean:", mean\_value) # Output: Mean: 3.0 | Eg: import numpy as np  # Example array  data = np.array([1, 2, 3, 4, 5])  # Using np.average() without weights  average\_value = np.average(data)  print("Average:", average\_value) # Output: Average: 3.0  # Using np.average() with weights  weights = np.array([1, 2, 3, 4, 5])  weighted\_average = np.average(data, weights=weights)  print("Weighted Average:", weighted\_average) # Output: Weighted Average: 3.6666666666666665 |

1. Describe the methods for reversing a NumPy array along different axes. Provide examples for 1D and 2D arrays.

Answer:

Reversing a NumPy array can be done using slicing and the np.flip() function. Here are the methods for reversing arrays along different axes with examples for both 1D and 2D arrays:

Reversing a 1D Array

Using Slicing:

Python

import numpy as np

# Create a 1D array

arr\_1d = np.array([1, 2, 3, 4, 5])

# Reverse the array using slicing

reversed\_arr\_1d = arr\_1d[::-1]

print("Reversed 1D array using slicing:", reversed\_arr\_1d)

# Output: [5 4 3 2 1]

Using np.flip():

Python

# Reverse the array using np.flip()

reversed\_arr\_1d\_flip = np.flip(arr\_1d)

print("Reversed 1D array using np.flip():", reversed\_arr\_1d\_flip)

# Output: [5 4 3 2 1]

Reversing a 2D Array

Reversing along Rows (Axis 0):

Python

# Create a 2D array

arr\_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Reverse the array along rows using slicing

reversed\_arr\_2d\_rows = arr\_2d[::-1, :]

print("Reversed 2D array along rows using slicing:\n", reversed\_arr\_2d\_rows)

# Output:

# [[7 8 9]

# [4 5 6]

# [1 2 3]]

# Reverse the array along rows using np.flip()

reversed\_arr\_2d\_rows\_flip = np.flip(arr\_2d, axis=0)

print("Reversed 2D array along rows using np.flip():\n", reversed\_arr\_2d\_rows\_flip)

# Output:

# [[7 8 9]

# [4 5 6]

# [1 2 3]]

Reversing along Columns (Axis 1):

Python

# Reverse the array along columns using slicing

reversed\_arr\_2d\_cols = arr\_2d[:, ::-1]

print("Reversed 2D array along columns using slicing:\n", reversed\_arr\_2d\_cols)

# Output:

# [[3 2 1]

# [6 5 4]

# [9 8 7]]

# Reverse the array along columns using np.flip()

reversed\_arr\_2d\_cols\_flip = np.flip(arr\_2d, axis=1)

print("Reversed 2D array along columns using np.flip():\n", reversed\_arr\_2d\_cols\_flip)

# Output:

# [[3 2 1]

# [6 5 4]

# [9 8 7]]

Summary

Slicing: Use [::-1] to reverse a 1D array, [::-1, :] to reverse rows, and [:, ::-1] to reverse columns in a 2D array.

np.flip(): Use np.flip(arr) to reverse a 1D array, np.flip(arr, axis=0) to reverse rows, and np.flip(arr, axis=1) to reverse columns in a 2D array.

1. How can you determine the data type of elements in a NumPy array? Discuss the importance of data types in memory management and performance.

Answer:

To determine the data type of elements in a NumPy array, you can use the dtype attribute. Here’s how you can do it:

Determining the Data Type

Using dtype Attribute:

Python

import numpy as np

# Create a NumPy array

arr = np.array([1, 2, 3, 4, 5])

# Determine the data type of the array elements

data\_type = arr.dtype

print("Data type of array elements:", data\_type)

# Output: Data type of array elements: int64

Using astype Method: You can also convert the data type of an array using the astype method and then check the dtype:

Python

# Convert array to float type

arr\_float = arr.astype(float)

print("Data type of array elements after conversion:", arr\_float.dtype)

# Output: Data type of array elements after conversion: float64

Importance of Data Types in Memory Management and Performance

Memory Management:

Efficient Storage: Different data types consume different amounts of memory. For example, an int32 type uses 4 bytes per element, while an int64 type uses 8 bytes. Choosing the appropriate data type can save memory, especially when dealing with large datasets.

Reduced Memory Footprint: Using smaller data types (e.g., int8, float32) can significantly reduce the memory footprint of an array, which is crucial for memory-constrained environments.

Performance:

Faster Computations: Operations on smaller data types are generally faster because they require less memory bandwidth and can take advantage of CPU cache more effectively.

Optimized Algorithms: NumPy’s internal algorithms are optimized for specific data types, leading to faster execution of mathematical operations and functions.

Vectorization: NumPy leverages vectorized operations, which are highly optimized for performance. The choice of data type can impact the efficiency of these operations.

Example

Consider a scenario where you need to store a large array of integers. Using int8 instead of int64 can save a significant amount of memory:

Python

# Create a large array of integers

large\_arr = np.arange(1000000, dtype=np.int8)

print("Memory usage with int8:", large\_arr.nbytes) # Output: Memory usage with int8: 1000000 bytes

# Convert to int64

large\_arr\_int64 = large\_arr.astype(np.int64)

print("Memory usage with int64:", large\_arr\_int64.nbytes) # Output: Memory usage with int64: 8000000 bytes

1. Define ndarrays in NumPy and explain their key features. How do they differ from standard Python lists?

Definition of ndarray

In NumPy, an ndarray (short for N-dimensional array) is a powerful data structure used for storing and manipulating large, multi-dimensional arrays and matrices of numeric data. It is the core data structure of the NumPy library.

**Key Features of ndarray**

**Homogeneous Data**: All elements in an ndarray must be of the same data type, which ensures efficient storage and computation.

**Multi-dimensional:** ndarray can have any number of dimensions, making it suitable for representing complex data structures like matrices, tensors, and higher-dimensional data.

Efficient Memory Layout: ndarray uses contiguous blocks of memory, which enhances performance by leveraging CPU cache and reducing memory overhead.

**Broadcasting:** Supports broadcasting, allowing operations on arrays of different shapes without the need for explicit loops.

**Vectorized Operations:** Enables element-wise operations and mathematical functions to be applied directly to arrays, leading to concise and efficient code.

**Comprehensive Methods:** Provides a wide range of methods for array manipulation, including reshaping, slicing, indexing, and aggregation functions.

**Interoperability:** Integrates seamlessly with other scientific libraries in Python, such as SciPy, Pandas, and Matplotlib.

Differences Between ndarray and Python Lists

Data Type:

ndarray: Homogeneous; all elements must be of the same type.

Python Lists: Heterogeneous; can contain elements of different types.

Performance:

ndarray: Optimized for numerical operations, leading to faster computations due to efficient memory usage and vectorized operations.

Python Lists: Slower for numerical operations as they are not optimized for such tasks.

Memory Efficiency:

ndarray: Uses contiguous memory blocks, which reduces memory overhead and improves cache performance.

Python Lists: Use pointers to objects, leading to higher memory overhead and less efficient memory usage.

Functionality:

ndarray: Provides a rich set of methods for mathematical and statistical operations, reshaping, slicing, and more.

Python Lists: Limited built-in methods for numerical operations; often requires additional loops or list comprehensions.

Dimensionality:

ndarray: Supports multi-dimensional arrays (e.g., 2D, 3D, etc.).

Python Lists: Primarily one-dimensional, though nested lists can simulate multi-dimensional arrays but with less efficiency and more complexity.

Example

Here’s a simple comparison of creating and manipulating a 2D array using ndarray and a nested Python list:

Python

import numpy as np

# Creating a 2D ndarray

nd\_array = np.array([[1, 2, 3], [4, 5, 6]])

print("NumPy ndarray:\n", nd\_array)

# Creating a 2D Python list

py\_list = [[1, 2, 3], [4, 5, 6]]

print("Python list:\n", py\_list)

# Accessing elements

print("Element at (1, 2) in ndarray:", nd\_array[1, 2])

print("Element at (1, 2) in list:", py\_list[1][2])

# Performing element-wise addition

nd\_array\_sum = nd\_array + 1

print("Element-wise addition with ndarray:\n", nd\_array\_sum)

# Performing element-wise addition with list (requires loop)

py\_list\_sum = [[element + 1 for element in row] for row in py\_list]

print("Element-wise addition with list:\n", py\_list\_sum)

1. Analyze the performance benefits of NumPy arrays over Python lists for large-scale numerical operations.

Answer:

NumPy arrays offer significant performance benefits over Python lists, especially for large-scale numerical operations. Here are some key reasons why NumPy arrays are more efficient:

1. Memory Efficiency

Contiguous Memory Allocation: NumPy arrays are stored in contiguous blocks of memory, which reduces memory overhead and improves cache performance. This is in contrast to Python lists, which store references to objects, leading to higher memory usage.

Fixed Data Types: NumPy arrays have a fixed data type for all elements, allowing for more efficient storage and access. Python lists can store elements of different types, which adds overhead.

2. Speed and Performance

Vectorized Operations: NumPy supports vectorized operations, allowing you to perform element-wise operations without explicit loops. This leverages low-level optimizations and results in faster execution.

Optimized Algorithms: NumPy’s internal algorithms are highly optimized for numerical computations, often implemented in C or Fortran, which are faster than Python’s interpreted code.

3. Broadcasting

Automatic Expansion: NumPy’s broadcasting feature allows operations on arrays of different shapes without the need for explicit loops or manual expansion. This simplifies code and improves performance.

4. Efficient Mathematical Functions

Built-in Functions: NumPy provides a wide range of mathematical functions that are optimized for performance. These functions operate directly on arrays and are much faster than equivalent operations on Python lists.

Example: Performance Comparison

Let’s compare the performance of NumPy arrays and Python lists for a simple operation like element-wise addition:

Python

import numpy as np

import time

# Create large arrays/lists

size = 1000000

np\_array = np.arange(size)

py\_list = list(range(size))

# Element-wise addition using NumPy

start\_time = time.time()

np\_result = np\_array + 1

np\_time = time.time() - start\_time

# Element-wise addition using Python list

start\_time = time.time()

py\_result = [x + 1 for x in py\_list]

py\_time = time.time() - start\_time

print(f"NumPy time: {np\_time:.6f} seconds")

print(f"Python list time: {py\_time:.6f} seconds")

AI-generated code. Review and use carefully. More info on FAQ.

Results

In most cases, you will find that the NumPy operation is significantly faster than the equivalent operation on a Python list. This is due to the reasons mentioned above, including memory efficiency, vectorized operations, and optimized algorithms.

Summary

Memory Efficiency: NumPy arrays use less memory and have better cache performance.

Speed: NumPy operations are faster due to vectorization and optimized algorithms.

Ease of Use: Broadcasting and built-in functions simplify code and improve performance.

1. Compare vstack() and hstack() functions in NumPy. Provide examples demonstrating their usage and output.

Answer:

|  |  |
| --- | --- |
| np.vstack() | np.hstack() |
| Purpose: Stacks arrays in sequence vertically (row-wise). | Purpose: Stacks arrays in sequence horizontally (column-wise). |
| Usage: Useful when you want to combine multiple arrays along the vertical axis (i.e., add rows). | Usage: Useful when you want to combine multiple arrays along the horizontal axis (i.e., add columns). |
| Example of np.vstack()  Python  import numpy as np  # Create two 1D arrays  arr1 = np.array([1, 2, 3])  arr2 = np.array([4, 5, 6])  # Stack arrays vertically  vstack\_result = np.vstack((arr1, arr2))  print("Vertical Stack (vstack):\n", vstack\_result)  # Output:  # [[1 2 3]  # [4 5 6]]  # Create two 2D arrays  arr3 = np.array([[1, 2, 3], [4, 5, 6]])  arr4 = np.array([[7, 8, 9], [10, 11, 12]])  # Stack 2D arrays vertically  vstack\_result\_2d = np.vstack((arr3, arr4))  print("Vertical Stack 2D (vstack):\n", vstack\_result\_2d)  # Output:  # [[ 1 2 3]  # [ 4 5 6]  # [ 7 8 9]  # [10 11 12]] | Example of np.hstack()  Python  # Create two 1D arrays  arr1 = np.array([1, 2, 3])  arr2 = np.array([4, 5, 6])  # Stack arrays horizontally  hstack\_result = np.hstack((arr1, arr2))  print("Horizontal Stack (hstack):\n", hstack\_result)  # Output:  # [1 2 3 4 5 6]  # Create two 2D arrays  arr3 = np.array([[1, 2, 3], [4, 5, 6]])  arr4 = np.array([[7, 8, 9], [10, 11, 12]])  # Stack 2D arrays horizontally  hstack\_result\_2d = np.hstack((arr3, arr4))  print("Horizontal Stack 2D (hstack):\n", hstack\_result\_2d)  # Output:  # [[ 1 2 3 7 8 9]  # [ 4 5 6 10 11 12]] |
| Axis of Stacking:  np.vstack(): Stacks arrays along the vertical axis (adds rows). | Axis of Stacking:  np.hstack(): Stacks arrays along the horizontal axis (adds columns). |
| Output Shape:  np.vstack(): The resulting array has more rows. | Output Shape:  np.hstack(): The resulting array has more columns. |
| Use np.vstack() when you need to stack arrays vertically, adding rows to the resulting array. | Use np.hstack() when you need to stack arrays horizontally, adding columns to the resulting array. |

1. Explain the differences between fliplr() and flipud() methods in NumPy, including their effects on various array dimensions.

Answer:

|  |  |
| --- | --- |
| np.fliplr() | np.flipud() |
| Purpose: Flips the array in the left/right direction (i.e., horizontally). | Purpose: Flips the array in the up/down direction (i.e., vertically). |
| Effect: Reverses the order of columns in a 2D array. | Effect: Reverses the order of rows in a 2D array. |
| Usage: Useful when you need to mirror an array horizontally. | Usage: Useful when you need to mirror an array vertically. |
| Example of np.fliplr()  Python  import numpy as np  # Create a 2D array  arr\_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  # Flip the array horizontally  fliplr\_result = np.fliplr(arr\_2d)  print("Original array:\n", arr\_2d)  print("Array after np.fliplr():\n", fliplr\_result)  # Output:  # Original array:  # [[1 2 3]  # [4 5 6]  # [7 8 9]]  # Array after np.fliplr():  # [[3 2 1]  # [6 5 4]  # [9 8 7]] | Example of np.flipud()  Python  # Flip the array verticallflipud\_result = np.flipud(arr\_2d)  print("Original array:\n", arr\_2d)  print("Array after np.flipud():\n", flipud\_result)  # Output:  # Original array:  # [[1 2 3]  # [4 5 6]  # [7 8 9]]  # Array after np.flipud():  # [[7 8 9]  # [4 5 6]  # [1 2 3]] |
| np.fliplr(): Flips arrays horizontally, reversing the order of columns. | np.flipud(): Flips arrays vertically, reversing the order of rows. |

1. Discuss the functionality of the array\_split() method in NumPy. How does it handle uneven splits?

The np.array\_split() method in NumPy is used to split an array into multiple sub-arrays. It is particularly useful when you need to divide an array into a specified number of sub-arrays, even if the array cannot be evenly divided.

**Functionality of np.array\_split()**

Purpose: To split an array into multiple sub-arrays.

Syntax: np.array\_split(ary, indices\_or\_sections, axis=0)

ary: The input array to be split.

indices\_or\_sections: If an integer, it indicates the number of equal-sized sub-arrays to create. If a list of indices, it specifies the points at which to split the array.

axis: The axis along which to split the array. The default is 0 (split along rows).

Handling Uneven Splits

When the array cannot be evenly divided, np.array\_split() ensures that the sub-arrays are as equal in size as possible. The function distributes the elements such that the first few sub-arrays have one more element than the remaining sub-arrays.

Example of np.array\_split()

Even Split

Python

import numpy as np

arr = np.arange(10) # Create an array

# Split the array into 5 equal parts

split\_arr = np.array\_split(arr, 5)

print("Even split into 5 parts:", split\_arr)

# Output: [array([0, 1]), array([2, 3]), array([4, 5]), array([6, 7]), array([8, 9])]

Uneven Split

Python

split\_arr\_uneven = np.array\_split(arr, 3) # Split the array into 3 parts (uneven split)

print("Uneven split into 3 parts:", split\_arr\_uneven)

# Output: [array([0, 1, 2, 3]), array([4, 5, 6]), array([7, 8, 9])]

In the uneven split example, the first sub-array has 4 elements, while the remaining sub-arrays have 3 elements each.

Example with 2D Array

Python

arr\_2d = np.arange(16).reshape(4, 4) # Create a 2D array

split\_arr\_2d = np.array\_split(arr\_2d, 3, axis=0) # Split the 2D array into 3 parts along the rows

print("2D array split into 3 parts along rows:")

for sub\_arr in split\_arr\_2d:

print(sub\_arr)

# Output:

# [[0 1 2 3]

# [4 5 6 7]]

# [[ 8 9 10 11]]

# [[12 13 14 15]]

# Split the 2D array into 3 parts along the columns

split\_arr\_2d\_cols = np.array\_split(arr\_2d, 3, axis=1)

print("2D array split into 3 parts along columns:")

for sub\_arr in split\_arr\_2d\_cols:

print(sub\_arr)

# Output:

# [[ 0 1]

# [ 4 5]

# [ 8 9]

# [12 13]]

# [[ 2]

# [ 6]

# [10]

# [14]]

# [[ 3]

# [ 7]

# [11]

# [15]]

Summary

np.array\_split(): Splits an array into multiple sub-arrays, handling both even and uneven splits.

Uneven Splits: Distributes elements such that the first few sub-arrays have one more element than the remaining sub-arrays.

Flexibility: Can split arrays along any specified axis, making it versatile for different data structures.

This method is particularly useful when working with data that needs to be divided into chunks for parallel processing or batch operations. If you have any more questions or need further examples, feel free to ask!

1. Explain the concepts of vectorization and broadcasting in NumPy. How do they contribute to efficient array operations?

In NumPy, two key concepts that contribute to efficient array operations are vectorization and broadcasting. These mechanisms allow operations on large datasets without the need for explicit loops, leading to performance improvements.

1. Vectorization

Vectorization refers to the process of applying operations directly on entire arrays or matrices, rather than iterating over individual elements with loops. It leverages low-level optimizations (often using highly efficient C or Fortran libraries) to speed up computations.

Example (Without Vectorization):

python

Copy code

import numpy as np

a = np.array([1, 2, 3, 4])

b = np.array([5, 6, 7, 8])

result = np.zeros(4)

for i in range(len(a)):

result[i] = a[i] \* b[i] # Element-wise multiplication using a loop

print(result)

Example (With Vectorization):

python

Copy code

result = a \* b # Element-wise multiplication directly

print(result)

The second example (vectorized) is much more efficient since it removes the need for an explicit Python loop, allowing NumPy to apply the operation using optimized C code.

Why Vectorization is Efficient:

No explicit loops: Loops in Python have high overhead. Vectorized operations are faster because NumPy executes operations in low-level compiled languages.

Parallelization: Many vectorized operations can be parallelized, using CPU vector instructions to process multiple elements simultaneously.

Memory locality: Vectorized operations often have better memory access patterns, which can reduce cache misses and speed up processing.

2. Broadcasting

Broadcasting is a powerful feature that allows NumPy to perform arithmetic operations on arrays of different shapes, by automatically expanding the smaller array to match the shape of the larger one. Instead of explicitly reshaping or duplicating arrays, broadcasting simplifies code and improves performance.

Example:

python

Copy code

a = np.array([1, 2, 3, 4]) # Shape: (4,)

b = 2 # Scalar (Shape: ())

result = a \* b # Broadcasting allows element-wise multiplication without reshaping `b`

print(result)

Here, the scalar b is "broadcast" to the shape of a, as if it were an array [2, 2, 2, 2], so the operation proceeds element-wise. Broadcasting can work in more complex scenarios as well, like when the shapes are partially compatible.

Example (2D Array Broadcasting):

python

Copy code

a = np.array([[1, 2, 3], [4, 5, 6]]) # Shape: (2, 3)

b = np.array([1, 2, 3]) # Shape: (3,)

result = a + b # Broadcasting allows row-wise addition

print(result)

In this example, b is broadcast to match the shape of a, and NumPy performs the addition row-wise.

Rules for Broadcasting:

If the two arrays differ in their number of dimensions, the smaller array is padded with ones on the left.

If the shapes of the two arrays differ in a dimension, the smaller size is expanded to match the larger size if the smaller size is 1.

If any dimension sizes do not match and neither is 1, NumPy throws an error.

How Vectorization and Broadcasting Improve Efficiency:

Fewer Loops: Instead of writing slow, explicit Python loops, operations are handled internally in efficient, optimized C loops.

Memory Efficiency: Broadcasting avoids the need to create large intermediate arrays (like expanding scalars or small arrays), saving memory.

Parallelization: Both vectorized operations and broadcasting can leverage hardware parallelism (such as SIMD instructions), speeding up computation.

Together, vectorization and broadcasting allow for more concise, readable, and significantly faster code in NumPy, particularly when dealing with large datasets.

**Practical Assignment:**

[https://github.com/Anmol2439/Assignment-/blob/main/Assignment\_numpy.ipynb](mailto:https://github.com/Anmol2439/Assignment-/blob/main/Assignment_numpy.ipynb)