COMP 1023 Introduction to Python Programming

Numerical Python (NumPy)

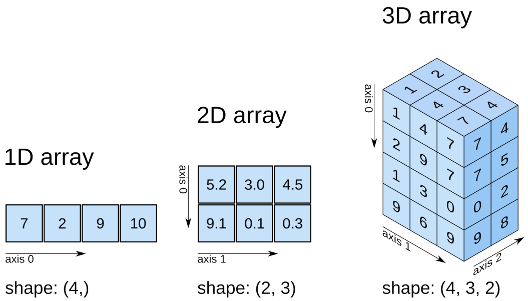
Dr. Cecia Chan, Prof. SC Cheung, Dr. Alex Lam, Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

NumPy Overview

- NumPy stands for Numerical Python and serves as the fundamental library for numerical and scientific calculations in the Python programming language.
- It offers a high-efficiency multidimensional array structure along with various tools for manipulating these arrays.

For this course, we will be utilizing NumPy version 2.3.1!



Understanding Arrays

- A NumPy array represents a grid of values that are all of the same data type and are indexed by a tuple of non-negative integers.
- The number of dimensions in the array is referred to as the rank.
- The shape of an array is defined as a tuple of integers that specifies the size of the array in each dimension.
- We can create NumPy arrays from nested lists in Python and retrieve elements using square brackets.

```python
import numpy as np  # Filename: array_example.py

a = np.array([10, 20, 30])            # Create a rank 1 array
print(type(a))                        # Output: <class 'numpy.ndarray'>
print(a.shape)                        # Output: (3,)
print(a[0], a[1], a[2])               # Output: 10 20 30
a[0] = 50                             # Modify an element of the array
print(a)                              # Output: [50 20 30]
b = np.array([[7, 8, 9], [10, 11, 12]]) # Create a rank 2 array
print(b.shape)                        # Output: (2, 3)
print(b[0, 0], b[0, 1], b[1, 0])      # Same as b[0][0], b[0][1], b[1][0]. Output: 7 8 10
```

Creating Arrays

- NumPy offers a variety of functions to create arrays.

```
# Filename: array_generation.py
import numpy as np

a = np.zeros((3,3))           # Generate an array filled with zeros
print(a)                      # Output: [[ 0.   0.   0.]
                              #          [ 0.   0.   0.]
                              #          [ 0.   0.   0.]]
b = np.ones((2,3))            # Generate an array filled with ones
print(b)                      # Output: [[ 1.   1.   1.]
                              #          [ 1.   1.   1.]]

c = np.full((3,2), 5)         # Generate an array with a constant value
print(c)                      # Output: [[ 5   5]
                              #          [ 5   5]
                              #          [ 5   5]]
d = np.eye(3)                 # Generate a 3x3 identity matrix
print(d)                      # Output: [[ 1.   0.   0.]
                              #          [ 0.   1.   0.]
                              #          [ 0.   0.   1.]]
e = np.random.rand(3,2)       # Generate an array with random values
print(e)                      # Output: [[ 0.12345678  0.87654321]
                              #          [ 0.23456789  0.76543210]
                              #          [ 0.34567890  0.65432109]]
```

Indexing and Slicing Arrays

Expression	Description
a[i]	Retrieve the element at index i, where i is an integer (indexing begins at 0).
a[-i]	Retrieve the i-th element from the end of the array, where i is an integer. The last element is accessed as -1, the second last element as -2, and so forth.
a[i:j]	Retrieve elements starting from index i up to, but not including, index j.
a[:] or a[0:]	Retrieve all elements along the specified axis.
a[:i]	Retrieve elements from index 0 up to, but not including, index i.
a[i:]	Retrieve elements starting from index i to the last element of the array.
a[i:j:n]	Retrieve elements from index i to j (exclusive), with a step size of n.
a[::-1]	Retrieve all elements in reverse order.

Indexing and Slicing Arrays

- Just like Python lists, NumPy arrays support slicing. Since arrays can be multidimensional, you need to specify a slice for each dimension.

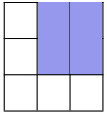
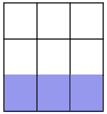
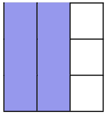
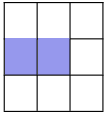
```python
# Filename: array_indexing_slicing_example.py
import numpy as np

# Create a rank 2 array with shape (3, 4)
# [[10  20  30  40]
#  [50  60  70  80]
#  [90 100 110 120]]
a = np.array([[10, 20, 30, 40], [50, 60, 70, 80], [90, 100, 110, 120]])
# Use slicing to extract a subarray containing the first 2 rows
# and columns 2 and 3; b will be the following array of shape (2, 2):
# [[30 40]
#  [70 80]]
b = a[:2, 2:4]
# A slice of an array is a view of the original data, so changing it
# will also change the original array.
print(a[0, 2])   # Output: 30
b[0, 0] = 99     # b[0, 0] refers to the same data as a[0, 2]
print(a[0, 2])   # Output: 99
```

Indexing and Slicing Arrays

- You can also combine integer indexing with slice indexing.
- However, this will result in an array of lower rank compared to the original array.

```
# Filename: array_indexing_slicing_example2.py
import numpy as np

# Create a rank 2 array with shape (3, 4)
# [[ 11  12  13  14]
#  [ 15  16  17  18]
#  [ 19  20  21  22]]
a = np.array([[11, 12, 13, 14], [15, 16, 17, 18], [19, 20, 21, 22]])

# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the original:
row_r1 = a[2, :]    # Rank 1 view of the third row of a
row_r2 = a[2:3, :]  # Rank 2 view of the third row of a
print(row_r1, row_r1.shape)  # Output: [19 20 21 22] (4,)
print(row_r2, row_r2.shape)  # Output: [[19 20 21 22]] (1, 4)

# The same distinction applies when accessing columns of an array:
col_r1 = a[:, 2]
col_r2 = a[:, 2:3]
print(col_r1, col_r1.shape)  # Output: [13 17 21] (3,)
print(col_r2, col_r2.shape)  # Output: [[13]
                             #           [17]
                             #           [21]] (3, 1)
```

Array Indexing and Slicing

	Expression	Shape
	arr[:2, 1:]	(2, 2)
	arr[2]	(3,)
	arr[2, :]	(3,)
	arr[2:, :]	(1, 3)
	arr[:, :2]	(3, 2)
	arr[1, :2]	(2,)
	arr[1:2, :2]	(1, 2)

Integer Array Indexing

- Integer array indexing enables you to create arbitrary arrays using data from another array.

```
# Filename: integer_array_indexing_example.py
import numpy as np

a = np.array([[7, 8], [9, 10], [11, 12]])

# An example of integer array indexing.
# The returned array will have shape (3,)
print(a[[0, 1, 2], [1, 0, 1]])  # Output: [ 8  9 12]

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 1], a[1, 0], a[2, 1]]))  # Output: [ 8  9 12]

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[1, 1], [0, 0]])  # Output: [9 9]

# Equivalent to the previous integer array indexing example
print(np.array([a[1, 0], a[1, 0]]))  # Output: [9 9]
```

Integer Array Indexing

- A useful technique with integer indexing is selecting or modifying one element from each row of a matrix.

```
import numpy as np  # Filename: integer_array_indexing_example2.py

# Create a new array from which we will select elements
a = np.array([[13, 14, 15], [16, 17, 18], [19, 20, 21], [22, 23, 24]])

print(a)  # Output: [[13 14 15]
          #           [16 17 18]
          #           [19 20 21]
          #           [22 23 24]]

b = np.array([1, 0, 2, 1])  # Create an array of indices
# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Output: [14 16 21 23]
# Modify one element from each row of a using the indices in b
a[np.arange(4), b] += 5

print(a)  # Output: [[13 19 15]
          #           [21 17 18]
          #           [19 20 26]
          #           [22 28 24]]
```

Boolean Array Indexing

- Boolean array indexing allows you to select arbitrary elements from an array.
- This type of indexing is often used to extract elements that meet a specific condition.

```python
import numpy as np  # Filename: boolean_array_indexing_example.py

a = np.array([[7, 8], [9, 10], [11, 12]])

bool_idx = (a < 10)   # Identify elements of a that are less than 10; this creates a numpy
                      # array of Booleans with the same shape as a, where each entry in bool_idx
                      # indicates whether the corresponding element of a is < 10.

print(bool_idx)      # Output: [[ True  True]
                      #         [ True False]
                      #         [False False]]

# Use boolean array indexing to create a rank 1 array of the elements of a
# corresponding to the True values in bool_idx
print(a[bool_idx])  # Output: [7 8 9]

# All of the above can be done in one concise statement:
print(a[a < 10])    # Output: [7 8 9]
```

Data Types in NumPy

- Each NumPy array consists of a grid of elements of a same type.
- NumPy offers a wide range of data types that can be utilized to create arrays.
- When you create an array, NumPy attempts to determine the appropriate data type, but functions that create arrays often include an optional parameter to explicitly define the data type.

```
# Filename: dtype_example.py

import numpy as np

x = np.array([3, 4])    # Allow NumPy to infer the data type
print(x.dtype)          # Output: int64

x = np.array([3.5, 4.5])    # Allow NumPy to infer the data type
print(x.dtype)              # Output: float64

x = np.array([3, 4], dtype=np.float32)    # Specify a specific data type
print(x.dtype)                            # Output: float32
```

Data Types

Category	Data Type	Description
Numeric	`np.int8`	8-bit signed integer
	`np.int16`	16-bit signed integer
	`np.int32`	32-bit signed integer
	`np.int64`	64-bit signed integer
Unsigned Integer	`np.uint8`	8-bit unsigned integer
	`np.uint16`	16-bit unsigned integer
	`np.uint32`	32-bit unsigned integer
	`np.uint64`	64-bit unsigned integer
Floating Point	`np.float16`	16-bit floating point
	`np.float32`	32-bit floating point
	`np.float64`	64-bit floating point
	`np.float128`	128-bit floating point
Complex	`np.complex64`	64-bit complex number
	`np.complex128`	128-bit complex number
	`np.complex256`	256-bit complex number
Boolean	`np.bool_`	Boolean type (True/False)
String	`np.str_`	Variable-length string
	`np.bytes_`	Byte string
Object	`np.object_`	General-purpose type for arbitrary objects
Datetime	`np.datetime64`	Represents dates and times
	`np.timedelta64`	Represents time differences
Void	`np.void`	Represents flexible type

Mathematical Operations on Arrays

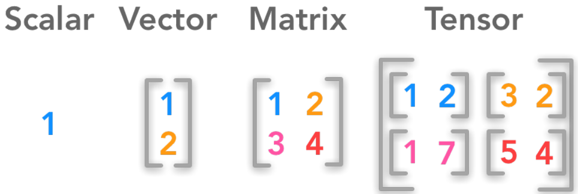
- Basic mathematical operations are performed element-wise on arrays, and are accessible both as operator overloads and as functions within the NumPy library.

```
# Filename: array_math_example.py
import numpy as np

x = np.array([[2, 3], [4, 5]], dtype=np.float64)
y = np.array([[6, 7], [8, 9]], dtype=np.float64)

# Elementwise sum; both yield the array
# [[ 8. 10.]
#  [12. 14.]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both yield the array
# [[-4. -4.]
#  [-4. -4.]]
print(x - y)
print(np.subtract(x, y))
```

```
# Elementwise product; both yield the array
# [[12. 21.]
#  [32. 45.]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both yield the array
# [[0.33333333 0.42857143]
#  [0.5        0.55555556]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; yields the array
# [[1.41421356 1.73205081]
#  [2.         2.23606798]]
print(np.sqrt(x))
```

Vectors, Matrices, and Tensors

- Vectors are one-dimensional arrays of numerical values.
- Matrices are two-dimensional arrays of numerical values.
- Tensors are arrays of numbers with three or more dimensions.
- These structures are essential for representing numerical data and play a significant role in performing various operations.
- As a result, they are crucial in fields like statistics, machine learning, and computer science.



$$1 \quad \begin{bmatrix}1\\2\end{bmatrix} \quad \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \quad \begin{bmatrix}\begin{bmatrix}1 & 2\end{bmatrix} & \begin{bmatrix}3 & 2\end{bmatrix}\\ \begin{bmatrix}1 & 7\end{bmatrix} & \begin{bmatrix}5 & 4\end{bmatrix}\end{bmatrix}$$

Dot Product

- The dot product is defined as the sum of the products of corresponding elements in two vectors (arrays) of the same size.
- The result of the dot product is a single scalar value.
- Now, let's explore how to calculate the dot product.

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = (1\text{x}10 + 2\text{x}20 + 3\text{x}30) = 140$$

Matrix Multiplication

- Matrix multiplication is a crucial operation in the field of artificial intelligence.
- It serves as the matrix equivalent of the dot product between two matrices.
- The result of matrix multiplication is a matrix where each element corresponds to the dot products of vector pairs from the two matrices.
- Let's explore how to perform matrix multiplication.

$$\underset{2 \text{ X } 3}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}} \text{ X } \underset{3 \text{ X } 2}{\begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}}$$

$$= \begin{bmatrix} 1\text{X}10 + 2\text{X}20 + 3\text{X}30 & 1\text{X}11 + 2\text{X}21 + 3\text{X}31 \\ 4\text{X}10 + 5\text{X}20 + 6\text{X}30 & 4\text{X}11 + 5\text{X}21 + 6\text{X}31 \end{bmatrix}$$

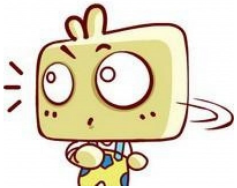
$$= \begin{bmatrix} 10 + 40 + 90 & 11 + 42 + 93 \\ 40 + 100 + 180 & 44 + 105 + 186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$

Mathematical Operations on Arrays

- The dot product can be calculated using the dot function or method.
- Matrix multiplication can be performed in Python using one of the following approaches:
  - dot function/method
  - matmul method
  - @ operator (available in Python 3.5 and above, which is equivalent to the matmul method)

Distinction between dot and matmul for matrix multiplication

When dealing with multi-dimensional arrays (N-dimensional arrays where $N > 2$), the results from the two methods might vary slightly.
Recommendation: It is recommended to use matmul or the @ operator for performing matrix multiplications.



Mathematical Operations on Arrays

- The dot operation is accessible both as a function in the NumPy library and as an instance method of array objects.

```
import numpy as np  # Filename: dot_product_example.py

x = np.array([[2, 4], [6, 8]])
y = np.array([[1, 3], [5, 7]])
v = np.array([8, 9])
w = np.array([10, 11])

# Dot product of vectors; both yield 170
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both yield the rank 1 array [52 120]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both yield the rank 2 array
# [[22 34]
#  [46 74]]
print(x.dot(y))
print(np.dot(x, y))
```

Matrix Operations

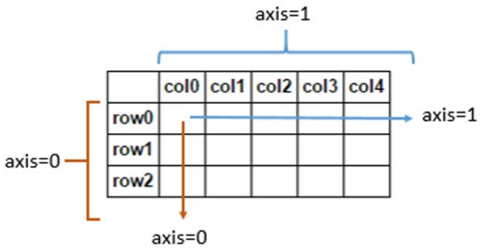
- matmul is accessible exclusively as a function in the NumPy library.

```
import numpy as np  # Filename: matrix_operations.py

a = np.array([[2, 3], [5, 7]])
b = np.array([[4, 1], [6, 3]])
u = np.array([8, 9])
v = np.array([10, 11])

# Inner product of vectors; both yield 170
print(np.matmul(u, v))
print(u @ v)

# Matrix / vector product; both yield the rank 1 array [43 103]
print(np.matmul(a, u))
print(a @ u)

# Matrix / matrix product; both yield the rank 2 array
# [[26 11]
#  [62 26]]
print(np.matmul(a, b))
print(a @ b)
```

Functions in NumPy

- NumPy offers a variety of functions to facilitate calculations on arrays; one of the key functions is the mean.

```
import numpy as np  # Filename: mean_func.py

y = np.array([[5, 10], [15, 20]])

print(np.mean(y))           # Calculate the mean of all elements; output 12.5
print(np.mean(y, axis=0))  # Calculate the mean of each column; output [10. 15.]
print(np.mean(y, axis=1))  # Calculate the mean of each row; output [7.5 17.5]
```



Matrix Transposition

- In addition to performing mathematical operations on arrays, we often need to reshape or adjust their structure.
- A fundamental operation in this category is transposing a matrix (or 2D array). You can accomplish this by utilizing the T attribute of an array, the transpose() function, or the transpose() method available on the array.
- The T attribute or NumPy's transpose() function effectively switches the dimensions of the specified array, transforming rows into columns and vice versa.

```
import numpy as np  # Filename: transpose_example.py

a = np.array([[5, 6], [7, 8]])
print(a)                # Output [[5 6]
                        #         [7 8]]
print(a.T)              # Output [[5 7]
                        #         [6 8]]
print(np.transpose(a))  # Output [[5 7]
                        #         [6 8]]
print(a.transpose())    # Output [[5 7]
                        #         [6 8]]
```

Matrix Transposition

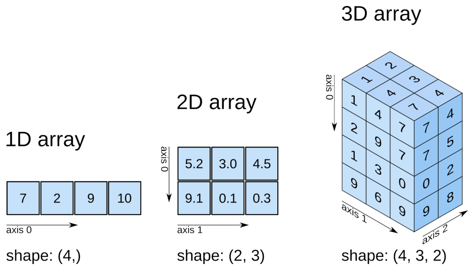
Is it possible to transpose a multi-dimensional array? Absolutely.

- The transpose function includes the axes parameter, which allows you to rearrange the dimensions of the array based on the specified axes.

Syntax

```
np.transpose(<array>, <axes>)
```

- Parameters:
  - `array`: the array you wish to transpose
  - `axes`: Defaults to None. When set to None or omitted, it will reverse the dimensions of the array `array`. If provided, it should be a tuple or list containing a permutation of $[0, 1, \ldots, N-1]$, where N represents the number of dimensions in `array`.



Matrix Transposition

```
# Filename: transpose_example2.py

import numpy as np
# Create a 3D array
# (2 layers, 4 rows, and 3 columns)
# with numbers 0 to 23.
array = np.array([[[ 0,  1,  2],
                   [ 3,  4,  5],
                   [ 6,  7,  8],
                   [ 9, 10, 11]],
                  [[12, 13, 14],
                   [15, 16, 17],
                   [18, 19, 20],
                   [21, 22, 23]]])

# The transpose function can rearrange the axes of the array.
# For instance, if we want to
# move the current last axis (i.e., 2)
# to the front (as axis 0),
# shift the current first axis (i.e., 0)
# to the middle (as axis 1),
# and place the current second axis (i.e., 1)
# at the end (as axis 2).
# We specify the order in transpose as [2, 0, 1].
print(np.transpose(array, [2, 0, 1]))
# Equivalent output using the line below
print(array.transpose([2, 0, 1]))

# Output will be [[[ 0  3  6  9]
#                  [12 15 18 21]]
#
#                 [[ 1  4  7 10]
#                  [13 16 19 22]]
#
#                 [[ 2  5  8 11]
#                  [14 17 20 23]]]
```

Matrix Transposition

- It is important to note that taking the transpose of a rank 1 array (or 1D array) has no effect.

Important Note

```
# Filename: transpose_example3.py

import numpy as np

vector = np.array([1, 2, 3])
print(vector)                # Output [1 2 3]
print(vector.T)              # Output [1 2 3]
print(np.transpose(vector))  # Output [1 2 3]
```

Array Reshaping

- Reshaping means changing the shape of an array.
- Remember that the shape of an array indicates the number of elements across each dimension.
- Through reshaping, we can add or remove dimensions or change the number of elements in each dimension.

Syntax

```
numpy.reshape(<array>, <newshape>)
```

- Parameters:
  - `array`: The array that you wish to reshape.
  - `newshape`: An integer or a tuple of integers.
    The new shape must be compatible with the original shape. If provided as a single integer, the result will be a 1D array of that specified length. One dimension can be set to -1, allowing its size to be automatically determined based on the total number of elements and the other specified dimensions.

Array Reshaping

```python
# Filename: reshape_example.py

import numpy as np

array1 = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120])
array2 = array1.reshape(4, 3)   # Reshape array1 into a 2D array with 4 rows and 3 columns
# array2 = array1.reshape(4, -1) # This line achieves the same result as array1.reshape(4, 3)
print(array2) # Output [[ 10  20  30]
              #         [ 40  50  60]
              #         [ 70  80  90]
              #         [100 110 120]]
print("Shape of array2:", array2.shape) # Output Shape of array2: (4, 3)
```

Challenge

How can you create the following array in a single line of code?

```python
array = np.array([ [ [ 0,  1,  2,  3], [ 4,  5,  6,  7], [ 8,  9, 10, 11] ],
                   [ [12, 13, 14, 15], [16, 17, 18, 19], [20, 21, 22, 23] ] ])
# Answer: array = np.arange(24).reshape(2, 3, 4)
```

Expanding Array Dimensions

- `np.newaxis` and `np.expand_dims()` enable us to add new dimensions to an array by introducing additional axes.
- Important: `numpy.newaxis` acts as an alias for `None`.

Syntax

```
numpy.newaxis
```

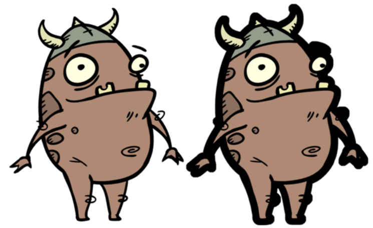
- How to use:
  - Inserting `numpy.newaxis` within [ ] adds a new dimension of size 1 at the specified location.

```
numpy.expand_dims(arr, axis)
```

- Parameters:
  - `arr`: The array to be expanded.
  - `axis`: An integer or a tuple of integers indicating the position in the expanded axes where the new axis (or axes) will be inserted.

Expanding Array Dimensions with np.newaxis

```
# Filename: newaxis_example.py
import numpy as np

array1 = np.array([10, 20, 30, 40, 50])
print(array1)  # Output: [10 20 30 40 50]

# Convert 1D array to a row vector
array2 = array1[np.newaxis]
print(array2)  # Output: [[10 20 30 40 50]]

# Convert 1D array to a row vector using None
array3 = array1[None]
print(array3)  # Output: [[10 20 30 40 50]]

array4 = np.array([[7, 8, 9], [10, 11, 12]])
print(array4)  # Output: [[ 7  8  9]
               #          [10 11 12]]

array5 = array4[np.newaxis]
print(array5)  # Output: [[[ 7  8  9]
               #           [10 11 12]]]
```



Expanding Array Dimensions with np.newaxis

```
# Filename: newaxis_example2.py
import numpy as np

array1 = np.array([[7, 8, 9], [10, 11, 12]])
print(array1)  # Output: [[ 7  8  9]
               #          [10 11 12]]

# np.newaxis adds a new dimension of size 1, i.e., 1 column
array2 = array1[:, :, np.newaxis]  # Equivalent to array2 = array1[..., np.newaxis]

print(array2)  # Output: [[[ 7]
               #           [ 8]
               #           [ 9]]
               #          [[10]
               #           [11]
               #           [12]]]

print(array2.shape) # Output: (2, 3, 1), indicating 2 layers, 3 rows, and 1 column
```

Expanding Array Dimensions with np.newaxis

```python
# Filename: newaxis_example3.py
import numpy as np

array1 = np.array([[5, 6, 7], [8, 9, 10]])
print(array1)  # Output: [[ 5  6  7]
               #          [ 8  9 10]]

# np.newaxis adds a new dimension of size 1, i.e., 1 row
# highest dimension, i.e., row
array2 = array1[np.newaxis, :, :]

print(array2)  # Output: [[[ 5  6  7]
               #           [ 8  9 10]]]

print(array2.shape) # Output: (1, 2, 3), indicating 1 layer, 2 rows, and 3 columns
```

Expanding Array Dimensions with np.expand_dims

```
# Filename: expand_dims_example.py
import numpy as np

array1 = np.array([[7, 8, 9], [10, 11, 12]])
print(array1)  # Output: [[ 7  8  9]
               #          [10 11 12]]

# 1 means adding a new dimension of size 1, after the 1st dimension
array2 = np.expand_dims(array1, axis=1)

print(array2)  # Output: [[[ 7  8  9]]
               #          [[10 11 12]]]

print(array2.shape) # Output: (2, 1, 3), indicating 2 layers, 1 row, and 3 columns
```

Expanding Array Dimensions with np.expand_dims

```
# Filename: expand_dims_example2.py
import numpy as np

array1 = np.array([[13, 14, 15], [16, 17, 18]])
print(array1)  # Output: [[13 14 15]
               #          [16 17 18]]

# 2 means adding a new dimension of size 1, after the 2nd dimension
array2 = np.expand_dims(array1, axis=2)

print(array2)  # Output: [[[13]
               #           [14]
               #           [15]]
               #          [[16]
               #           [17]
               #           [18]]]

print(array2.shape) # Output: (2, 3, 1), indicating 2 layers, 3 rows, and 1 column
```

View and Copy

- In Python, the terms view and copy refer to how data is accessed and modified.
- View
  - A view is a reference to the original data.
  - Changes made through a view affect the original data.

```
import numpy as np   # Filename: view_example.py
arr = np.array([1, 2, 3])
view_arr = arr[1:3]  # Creates a view
view_arr[0] = 10     # Modifies arr
print(arr)           # Print [ 1 10 3]
```

- Copy
  - A copy creates a new object with the same data.
  - Modifications to a copy do not affect the original object.
  - Useful for preserving the original data while making changes.

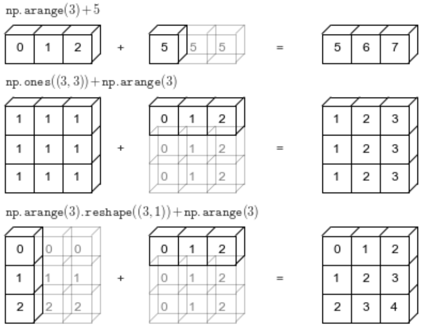
```
import numpy as np  # Filename: copy_example.py
import copy
original = np.array([1, 2, 3])
copy_list = copy.copy(original)  # Creates a shallow copy
copy_list[0] = 10                # Does not affect original
print(original)                  # Print [1 2 3]
```

NumPy Operations: View vs. Copy

- Operations that produce a view:
  - `arr[1:3]` - Slicing an array.
  - `arr.reshape(new_shape)` - Reshaping an array without changing data.
  - `arr.transpose()` - Transposing an array.
  - `arr.flatten()` - Returns a flattened view of the array.
  - `arr[:, :]` - Subsetting the array.
  - `arr.view()` - Creates a view of the array.
  - `arr[np.newaxis]` - Adds a new axis, creating a view.
  - `np.expand_dims(arr, axis)` - Expands the shape of the array, creating a view.
- Operations that produce a copy:
  - `np.copy(arr)` - Creates a deep copy of the array.
  - `arr.copy()` - Method to create a copy of the array.
  - `arr.astype(new_dtype)` - Converts to a new data type, creating a copy.
  - `np.array(arr)` - Creating a new array from an existing one.
  - `arr.tolist()` - Converts the array to a list, creating a copy.
  - `arr[np.arange(n)]` - Indexing with an array produces a copy.

Broadcasting

- Broadcasting is a powerful feature that enables NumPy to handle arrays of varying shapes during arithmetic operations.
- Often, we have a smaller array and a larger array, and we wish to apply the smaller array multiple times to perform operations on the larger array.



Broadcasting addresses the challenge of performing arithmetic between arrays of different shapes by extending the smaller array along the last mismatched dimension.

Broadcasting - Motivation

```
# Filename: broadcasting_example.py
import numpy as np

# We will add the vector v to each row of the matrix x, storing the result in y
x = np.array([[2, 3, 4], [5, 6, 7], [8, 9, 10], [11, 12, 13]])
v = np.array([2, 1, 2])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x using an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

print(y)  # Output: [[ 4  4  6]
          #          [ 7  7  9]
          #          [10 10 12]
          #          [13 13 15]]
```

The above method works; however, when the matrix x is significantly large, using an explicit loop can be inefficient.

Broadcasting - Motivation

- It is important to note that adding the vector *v* to each row of the matrix *x* is equivalent to creating a matrix *vv* by stacking multiple copies of *v* vertically, followed by performing element-wise addition of *x* and *vv*. This can be implemented as follows:

```python
# Filename: broadcasting_example2.py
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[2, 3, 4], [5, 6, 7], [8, 9, 10], [11, 12, 13]])
v = np.array([2, 1, 2])
vv = np.tile(v, (4, 1))    # Stack 4 copies of v on top of each other
print(vv)                  # Output: [[2 1 2]
                           #          [2 1 2]
                           #          [2 1 2]
                           #          [2 1 2]]
y = x + vv                 # Add x and vv elementwise

print(y)                   # Output: [[ 4  4  6]
                           #          [ 7  7  9]
                           #          [10 10 12]
                           #          [13 13 15]]
```

Broadcasting

- NumPy broadcasting enables us to execute this operation without generating multiple instances of v.
- Take a look at the following example that utilizes broadcasting:

```
import numpy as np  # Filename: broadcasting_demo.py

# We will add the vector v to every row of the matrix x,
# and store the results in the matrix y
x = np.array([[2, 3, 4], [5, 6, 7], [8, 9, 10], [11, 12, 13]])
v = np.array([2, 1, 2])
y = x + v  # Add v to each row of x using broadcasting
print(y)   # Print [[ 4  4  6]
           #        [ 7  7  9]
           #        [10 10 12]
           #        [13 13 15]]
```

The statement `y = x + v` operates seamlessly even though x has a shape of (4, 3) and v has a shape of (3,); broadcasting allows this line to function as if v had a shape of (4, 3), with each row being a duplicate of v, thus enabling element-wise addition.

Broadcasting Principles

Broadcasting two arrays together follows these rules:

1. If the arrays differ in rank, add 1s to the shape of the lower rank array until both shapes are of equal length.
2. Two arrays are considered compatible in a dimension if they have the same size in that dimension, or if one of the arrays has a size of 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.

Note

- After broadcasting, each array acts as if it had a shape that matches the element-wise maximum of the shapes of the two original arrays.
- In any dimension where one array has a size of 1 and the other has a size greater than 1, the first array behaves as if it were replicated along that dimension.

Exploring Broadcasting Principles

- Consider two arrays `A = np.array([1, 2, 3])` and `B = np.array([2])`. Can we execute `A * B`?
  1. Do they share the same rank? Yes, both A and B have a rank of 1.
  2. Are they compatible across all dimensions? Yes, since array B has a size of 1 in that dimension.
  3. Therefore, they are compatible.

```python
# Filename: broadcasting_rule1.py
import numpy as np

A = np.array([1, 2, 3])
print(A.ndim)     # Print 1
print(A.shape)    # Print (3,)
B = np.array([2])
print(B.ndim)     # Print 1
print(B.shape)    # Print (1,)
print(A * B)      # Print [2 4 6]
```

Array	Shape		
A	(	3,	)
B	(	1,	)
		B has a size of 1	
$A * B$	(	3,	)

Exploring Broadcasting Principles

- Consider two arrays `A = np.array([1, 2, 3])` and `B = np.array([[4, 4, 4], [3, 3, 3]])`. Is it possible to execute `A * B`?
  1. Do they share the same rank? No, they do not; the rank of A is 1, while the rank of B is 2. However, we can add 1s to the shape of A (the lower rank array) until both shapes are of equal length.
  2. Are they compatible across all dimensions? Yes, because array A has a size of 1 in that dimension.
  3. Thus, they are compatible.

```python
# Filename: broadcasting_rule2.py
import numpy as np

A = np.array([1, 2, 3])
print(A.ndim)  # Print 1
print(A.shape) # Print (3,)
B = np.array([[4, 4, 4], [3, 3, 3]])
print(B.ndim)  # Print 2
print(B.shape) # Print (2, 3)
print(A * B)   # Print [[ 4  8 12]
               #        [ 3  6  9]]
```

Array	Shape			
A	(	1 (Prepended),	3	)
B	(	2,	3	)
		A has size 1	Same size	
$A * B$	(	2,	3	)

Exploring Broadcasting Compatibility

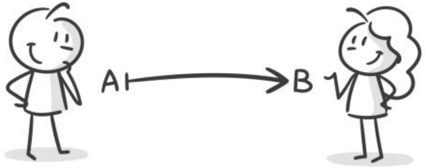
For each of the following pairs, determine whether they are compatible. If they are, what will be the size of the resulting array after performing $A * B$? Let's consider the following pairs:

1. Pair 1
   - A: Shape - 5×4
   - B: Shape - 1
2. Pair 2
   - A: Shape - 5×4
   - B: Shape - 4
3. Pair 3
   - A: Shape - $15 \times 3 \times 5$
   - B: Shape - $15 \times 1 \times 5$
4. Pair 4
   - A: Shape - $15 \times 3 \times 5$
   - B: Shape - 3×5
5. Pair 5
   - A: Shape - $15 \times 3 \times 5$
   - B: Shape - 3×1
6. Pair 6
   - A: Shape - $16 \times 6 \times 7$
   - B: Shape - 16×6

1. Yes. Resulting Shape - 5×4
2. Yes. Resulting Shape - 5×4
3. Yes. Resulting Shape - $15 \times 3 \times 5$
4. Yes. Resulting Shape - $15 \times 3 \times 5$
5. Yes. Resulting Shape - $15 \times 3 \times 5$
6. No compatibility.

Broadcasting in Action

- Broadcasting techniques are effective for streamlining calculations.
- We will explore the following examples to demonstrate their effectiveness:
  1. Centering a dataset (Gradebook scenario)
  2. Calculating Pairwise Distances (Euclidean distance between rows in two arrays)



Example 1: Centering a Dataset

- Imagine we have a gradebook for 5 students, each having taken 3 exams. We store these scores in a 5×3 array.

```
import numpy as np  # Filename: centering_a_dataset.py
scores = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15]])
```

- Next, we calculate the mean score for each exam using the `mean` function along the first dimension.

```
score_mean = scores.mean(0)  # 0 indicates the first dimension
print(score_mean)            # Print array([7., 8., 9.])
```

- We can center the `scores` array by subtracting the mean (this utilizes broadcasting):

```
scores_centered = scores - score_mean
print(scores_centered)     # Print array([[-6., -6., -6.],
                           #              [-3., -3., -3.],
                           #              [ 0.,  0.,  0.],
                           #              [ 3.,  3.,  3.],
                           #              [ 6.,  6.,  6.]])
```

- To verify that our calculations are accurate, we check the mean of the centered array to ensure it equals zero.

```
print(scores_centered.mean(axis=0))    # Print array([0., 0., 0.])
```

Example 2: Pairwise Distances

- Consider two 2D arrays, x and y.
- Array x has a shape of (M, D) while array y has a shape of (N, D), indicating they have different numbers of rows but share the same number of columns.
- Our goal is to calculate the Euclidean distance between every pair of rows from these two arrays.
- Specifically, if a row from x is represented by D values $(x_0, x_1, \ldots, x_{D-1})$, and a row from y is represented by D values $(y_0, y_1, \ldots, y_{D-1})$, we aim to compute the Euclidean distance between these two rows as follows:

$$\sqrt{(x_0 - y_0)^2 + (x_1 - y_1)^2 + \ldots + (x_{D-1} - y_{D-1})^2}$$

Example 2: Pairwise Distances (Continued)

```python
# Filename: pairwise_distance.py
import numpy as np

# A shape-(3, 3) array
x = np.array([[ 8.54,  1.54,  8.12],
              [ 3.13,  8.76,  5.29],
              [ 7.73,  6.71,  1.31]])

# A shape-(2, 3) array
y = np.array([[ 8.65,  0.27,  4.67],
              [ 7.73,  7.26,  1.95]])
```

Questions

How many distance values do we need to compute?
Answer: 6 distances, one for each combination of rows from x and y.

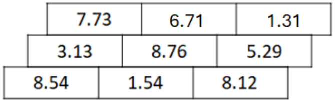
If x has a shape of $(M, 3)$ and y has a shape of $(N, 3)$, how many distance values will we calculate?
Answer: $M \times N$

Example 2: Pairwise Distances (Continued)

```python
reshaped_x = x.reshape(3, 1, 3)
reshaped_y = y.reshape(1, 2, 3)
diffs = reshaped_x - reshaped_y
print("Shape of diffs", diffs.shape)    # Print (3, 2, 3)
```

8.54	1.54	8.12
3.13	8.76	5.29
7.73	6.71	1.31

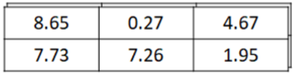
x



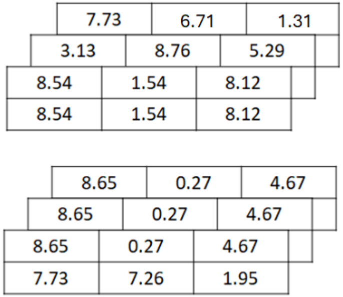
x.reshape(3, 1, 3)

8.65	0.27	4.67
7.73	7.26	1.95

y



y.reshape(1, 2, 3)



x.reshape(3, 1, 3) - y.reshape(1, 2, 3)

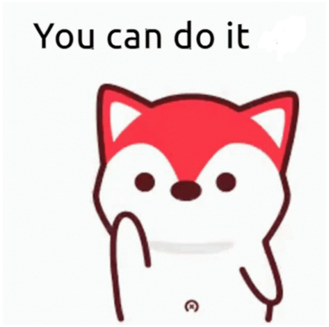
It is essential to note that, through broadcasting, `diffs[i,j]` represents `x[i] - y[j]`.

Example 2: Pairwise Distances (Continued)

```
print(diffs)  # Print [[[-0.11  1.27  3.45]
              #         [ 0.81 -5.72  6.17]]
              #        [[-5.52  8.49  0.62]
              #         [-4.6   1.5   3.34]]
              #        [[-0.92  6.44 -3.36]
              #         [ 0.   -0.55 -0.64]]]

dists = np.sqrt(np.sum(diffs**2, axis=2))  # axis=2 refers to the column axis

print(dists)  # Print [[ 3.67797499  8.45241977]
              #        [10.14568381  5.87925165]
              #        [ 7.32185769  0.84386018]]

print(dists.shape)  # Print (3, 2), indicating 3 rows and 2 columns
```

Question: Pairwise Distances

- Are you able to create a program that calculates the pairwise distances between two arrays, using np.newaxis instead of the reshape function?
- Likewise, can you develop a program that computes the pairwise distances of two arrays, utilizing np.expand_dims in place of the reshape function?

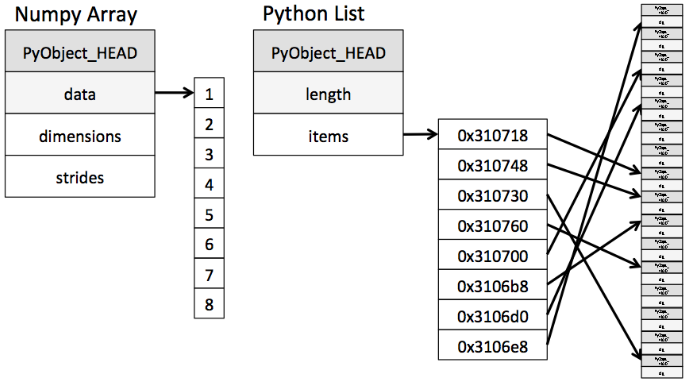


Why NumPy Arrays are Better?

- NumPy arrays consume less memory than Python lists.
- NumPy arrays are faster compared to Python lists.
- NumPy arrays are more convenient to use.

NumPy Array vs Python List

- A NumPy array contains a single reference to one contiguous block of data.
- A Python list contains a reference to a block of references, each of which references to a full Python object.



Memory Consumption for NumPy Arrays and Lists

```
# Filename: memory_consumption.py
import numpy as np, sys, gc  # Import numpy package, system, and gc module

def actualsize(input_object):
    memory_size = 0                   # memory_size: the actual memory size of "input_object"
                                      # Initialize it to 0
    ids = set()                       # ids: An empty set to store all the ids of objects in "input_object"
    objects = [input_object]          # objects: A list with "input_object" (traverse from "input_object")
    while objects:                    # While "objects" is non-empty
        new = []                      # new: An empty list to keep the items linked by "objects"
        for obj in objects:           # obj: Each object in "objects"
            if id(obj) not in ids:    # If the id of "obj" is not in "ids"
                ids.add(id(obj))      # Add the id of the "obj" to "ids"
                memory_size += sys.getsizeof(obj) # Use getsizeof to determine the size of "obj"
                                      # and add it to "memory_size"
                new.append(obj)       # Add "obj" to "new"
        objects = gc.get_referents(*new)  # Update "objects" with the list of objects directly
                                          # referred to by *new
    return memory_size                # Return "memory_size"

L = list(range(0, 1000))              # Define a Python list of 1000 elements
A = np.arange(1000)                   # Define a NumPy array of 1000 elements
# Print size of the whole list, it prints "Size of the whole list in bytes: 36056"
print("Size of the whole list in bytes:", actualsize(L))
# Print size of the whole NumPy array, it prints "Size of the whole NumPy array in bytes: 8112"
print("Size of the whole NumPy array in bytes:", actualsize(A))
```

Time Comparison Between NumPy Arrays and Python Lists

```
# Filename: time_comparison.py
import numpy as np       # Import required packages
import time as t

size = 1000000           # Size of arrays and lists

list1 = range(size)      # Declare lists
list2 = range(size)
array1 = np.arange(size) # Declare arrays
array2 = np.arange(size)

# Capture time before the multiplication of Python lists
initialTime = t.time()
# Multiply elements of both lists and store in another list
resultList = [(a * b) for a, b in zip(list1, list2)]
# Calculate execution time, it prints "Time taken by Lists: 0.13024258613586426 s"
print("Time taken by Lists:", (t.time() - initialTime), "s")

# Capture time before the multiplication of NumPy arrays
initialTime = t.time()
# Multiply elements of both NumPy arrays and store in another NumPy array
resultArray = array1 * array2
# Calculate execution time, it prints "Time taken by NumPy Arrays: 0.006006956100463867 s"
print("Time taken by NumPy Arrays:", (t.time() - initialTime), "s")
```

Effect of Operations on NumPy Arrays and Python Lists

```python
# Filename: effect_of_operations.py
import numpy as np # Import NumPy package

ls = [1, 2, 3]        # Declare a list
arr = np.array(ls)  # Convert the list into a NumPy array

try:
    ls = ls + 4     # Add 4 to each element of the list
except TypeError:
    print("Lists don't support list + int")

# Now on the array
try:
    arr = arr + 4  # Add 4 to each element of the NumPy array
    print("Modified NumPy array: ", arr)  # Print the NumPy array
except TypeError:
    print("NumPy arrays don't support array + int")
```

We won't cover try-except in this course.

Output:

```
Lists don't support list + int
Modified NumPy array: [5 6 7]
```

Key Terms and Acknowledgment

- Array indexing and slicing
- Boolean array indexing
- Broadcasting
- Dot product
- expand_dims
- Integer array indexing
- Matrix
- Matrix multiplication
- newaxis
- NumPy array
- Rank
- Reshaping
- Shape
- Sum function
- Tensor
- Transpose
- Vector

Acknowledgment

The content of these slides has been developed with reference to the Python NumPy tutorial available at `https://cs231n.github.io/python-numpy-tutorial/`.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

1. NumPy is a ________________ for Numerical Python and is the core library for numeric and scientific computing in Python.
2. A NumPy array is a ________________ of values, all of the same type, and is indexed by a tuple of non-negative integers.
3. The ________________ of an array is a tuple of integers giving the size of the array along each dimension.
4. We can initialize NumPy arrays from ________________ Python lists and access elements using square brackets.
5. The function ________________ creates an array of all zeros.

Answer: 1. library, 2. grid, 3. shape, 4, nested, 5. np.zeros.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

6. The function ________________ creates an array of all ones.
7. The function ________________ creates a constant array filled with a specified value.
8. The ________________ of an array is the number of dimensions.
9. NumPy arrays can be ________________, which allows for operations on arrays of different shapes.
10. Broadcasting solves the problem of arithmetic between arrays of differing shapes by ________________ the smaller array along the last mismatched dimension.
11. The dot product is the ________________ of products of values in two same-sized vectors (arrays).

Answer: 6. np.ones, 7. np.full, 8. rank, 9. broadcasted, 10. replicating, 11. sum.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

12. The output of the dot product is a ________________.
13. The ________________ method is used to change the shape of an array.
14. The ________________ function allows you to construct arbitrary arrays using the data from another array.
15. The ________________ function allows you to add a new axis to an array.
16. The ________________ attribute of an array object is used to reverse the dimensions of the given array.

Answer: 12. scalar, 13. reshape, 14. integer array indexing, 15. np.newaxis, 16. T.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

17. The ________________ is an important operation for artificial intelligence and is a matrix version of the dot product.
18. NumPy provides many useful functions for performing computations on arrays; one of the most useful is the ________________ function.
19. The ________________ function computes the sum of all elements in an array.
20. When using integer array indexing, you can reuse the same element from the source array, which allows for ________________ from multiple rows of a matrix.

Answer: 17. matrix multiplication, 18. sum, 19. np.sum, 20. selecting or mutating.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

21. A ________________ is a reference to the original data, meaning changes made through it will affect the original data.
22. To create a new object with the same data without affecting the original, you use a ________________.
23. Changes made to a ________________ do not affect the original object.

Answer: 21. view, 22. copy, 23. copy.

Practice Problems (Print the result)

1. How to create array A of size 15, with all zeros?
2. How to find memory size of array A?
3. How to create array B with values ranging from 20 to 60?
4. How to create array C of reversed array of B?
5. How to create 4$\times$4 array D with values from 0 to 15 (from top to bottom, left to right)?
6. How to find the dimensions of array E $[[3, 4, 5], [6, 7, 8]]$?
7. How to find indices for non-zero elements from array F $[0, 3, 0, 0, 4, 0]$?
8. How to create 3$\times$3$\times$3 array G with random values?
9. How to find maximum values in array H $[1, 13, 0, 56, 71, 22]$?
10. How to find minimum values in array H?
11. How to find mean values of array H?
12. How to find standard deviation of array H?
13. How to find median in array H?
14. How to transpose array D?

Practice Problems (Print the results)

15. How to append array [4, 5, 6] to array I [1, 2, 3]?
16. How to member-wise add, subtract, multiply and divide two arrays J [1, 2, 3] and K [4, 5, 6]?
17. How to find the total sum of elements of array I?
18. How to find natural log of array I?
19. How to use build an array L with [8, 8, 8, 8, 8] using full/repeat function?
20. How to sort array M [2, 5, 7, 3, 6]?
21. How to find the indices of the maximum values in array M?
22. How to find the indices of the minimum values in array M?
23. How to find the indices of elements in array M that would sort?
24. How to find the inverse of array N = [[6, 1, 1], [4, -2, 5], [2, 8, 7]] in NumPy?
25. How to find absolute value of array N?
26. How to extract the third column (from all rows) of the array O [[11, 22, 33], [44, 55, 66], [77, 88, 99]]?
27. How to extract the sub-array consisting of the odd rows and even columns of P [[3, 6, 9, 12], [15, 18, 21, 24], [27, 30, 33, 36], [39, 42, 45, 48], [51, 54, 57, 60]]?

Practice Problems (Answers)

1. `A = np.zeros(15); print(A)`
2. `print(A.size * A.itemsize)`
3. `B = np.arange(20, 61); print(B)`
4. `C = B[::-1]; print(C)`
5. `D = np.arange(16).reshape(4, 4); print(D)`
6. `E = np.array([[3, 4, 5], [6, 7, 8]]); print(E.shape)`
7. `F = np.array([0, 3, 0, 0, 4, 0]); print(F.nonzero())`
8. `G = np.random.random((3, 3, 3)); print(G)`
9. `H = np.array([1, 13, 0, 56, 71, 22]); print(H.max())`
10. `print(H.min())`
11. `print(H.mean())`
12. `print(H.std())`
13. `print(np.median(H))`
14. `print(np.transpose(D))`
15. `I = np.array([1, 2, 3]); I = np.append(I, [4, 5, 6]); print(I)`

Practice Problems (Answers)

16.
```
J = np.array([1, 2, 3]); K = np.array([4, 5, 6]);
print(J + K); print(J - K); print(J * K); print(J / K)
```
17.
```
print(np.sum(I))
```
18.
```
print(np.log(I))
```
19.
```
L = no.full(5, 8); print(L)
# L = np.repeat(8, 5); print(L)
```
20.
```
M = np.array([2, 5, 7, 3, 6]); print(np.sort(M))
```
21.
```
print(M.argmax())
```
22.
```
print(M.argmin())
```
23.
```
print(M.argsort())
```
24.
```
N = np.array([[6, 1, 1], [4, -2, 5], [2, 8, 7]]); print(np.linalg.inv(N))
```
25.
```
print(np.abs(N))
```
26.
```
O = np.array([[11, 22, 33], [44, 55, 66], [77, 88, 99]])
print(O[:,2])
```
27.
```
P = np.array([[3, 6, 9, 12], [15, 18, 21, 24], [27, 30, 33, 36],
              [39,  42, 45, 48], [51, 54, 57, 60]])
print(P[::2, 1::2])
```

That's all!

Any questions?