Q1. What are the benefits of the built-in array package, if any?

Built-in array module defines an object type which can efficiently represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.

Q2. What are some of the array package's limitations?

Insertion and deletion operations are costly in arrays as elements are stored in contiguous memory. If the size of the declared array is more than the required size then, it can lead to memory wastage.

Q3. Describe the main differences between the array and numpy packages.

There are several important differences between NumPy arrays and the standard Python sequences:

* NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
* The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
* NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python’s built-in sequences.
* A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today’s scientific/mathematical Python-based software, just knowing how to use Python’s built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

The points about sequence size and speed are particularly important in scientific computing. As a simple example, consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data are stored in two Python lists, a and b, we could iterate over each element:

Q4. Explain the distinctions between the empty, ones, and zeros functions.

**Empty Array - Using numpy.empty**

This is used to create an uninitialized array of specified shape and dtype.

The syntax to use this constructor is as follows:

numpy.empty(shape, dtype, order)

Following is a description of parameters by this constructor:

shape: This parameter is used to indicate the desired shape of the specified array.

dtype: This parameter indicates the data type of the array items. The default value of this parameter is float.

order: The default order or value of this option is the c-style row-major order. This parameter can be set to F for FORTRAN-style (column-major order)

**Zero Value Array - Using numpy.zeroes**

This is used to return a new array of the specified size and each array item will be initialized with 0.

The syntax to use this constructor is as follows:

numpy.zeros(shape, dtype, order)

Following is a description of parameters by this constructor:

shape: This parameter is used to indicate the desired shape of the specified array.

dtype: This parameter indicates the data type of the array items. The default value of this parameter is float.

order: The default order or value of this option is the c-style row-major order. This parameter can be set to F for FORTRAN-style (column-major order)

**One Value Array - Using numpy.ones**

This is used to return a new array of the specified size and each array item will be initialized as 1.

The syntax to use this constructor is as follows:

numpy.ones(shape, dtype, order)

Following is a description of parameters by this constructor:

shape: This parameter is used to indicate the desired shape of the specified array.

dtype: This parameter indicates the data type of the array items. The default value of this parameter is float.

order: The default order or value of this option is the c-style row-major order. This parameter can be set to F for FORTRAN-style (column-major order)

Q5. In the fromfunction function, which is used to construct new arrays, what is the role of the callable argument?

numpy.fromfunction() function construct an array by executing a function over each coordinate and the resulting array, therefore, has a value fn(x, y, z) at coordinate (x, y, z).

Syntax : numpy.fromfunction(function, shape, dtype)

Parameters :

function : [callable] The function is called with N parameters, where N is the rank of shape. Each parameter represents the coordinates of the array varying along a specific axis.

shape : [(N, ) tuple of ints] Shape of the output array, which also determines the shape of the coordinate arrays passed to function.

dtype : [data-type, optional] Data-type of the coordinate arrays passed to function.

Return : The output array.

# Python program explaining

# numpy.fromfunction() function

# importing numpy as geek

import numpy as geek

gfg = geek.fromfunction(lambda i, j: i \* j, (4, 4), dtype = float)

print(gfg)

A callable object is an object that can accept some arguments (also called parameters) and possibly return an object (often a tuple containing multiple objects). A function is the simplest callable object in Python, but there are others, such as classes or certain class instances.

Q6. What happens when a numpy array is combined with a single-value operand (a scalar, such as an int or a floating-point value) through addition, as in the expression A + n?

Adding values at the end of the array is a necessary task especially when the data is not fixed and is prone to change. For this task we can use numpy. append(). This function can help us to append a single value as well as multiple values at the end of the array.

when a numpy array is combined with a single-value operand (a scalar, such as an int or a floating-point value) through addition, as in the expression A + n, it will add all the elements of the array with the interger or floating-point value n.

Code:

# importing the module

import numpy as np

# creating an array

arr = np.array([1, 8, 3, 3, 5])

arr+ 7

array([ 8, 15, 10, 10, 12])

Q7. Can array-to-scalar operations use combined operation-assign operators (such as += or \*=)? What is the outcome?

Yes array-to-scalar operations use combined operation-assign operators (such as += or \*=).

The outcome is that all the elements of the array are added or multiplied by the scalar and stored in the same variable.

Code:

import numpy as np

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

a+=2

a

array([3, 4, 5])

Q8. Does a numpy array contain fixed-length strings? What happens if you allocate a longer string to one of these arrays?

The dtype of any numpy array containing string values is the maximum length of any string present in the array. Once set, it will only be able to store new string having length not more than the maximum length at the time of the creation.

If we try to reassign some another string value having length greater than the maximum length of the existing elements, it simply discards all the values beyond the maximum length.

Let’s first visualize the problem with creating an arbitrary length numpy array of string type.

# importing numpy as np

import numpy as np

# Create the numpy array

country = np.array(['USA', 'Japan', 'UK', '', 'India', 'China'])

# Print the array

print(country)



As we can see in the output, the maximum length of any string length element in the given array is 5. Let’s try to assign a value having greater length at the place of missing value in the array.

# Assign 'New Zealand' at the place of missing value

country[country == ''] = 'New Zealand'

# Print the modified array

print(country)



As we can see in the output, ‘New Z’ has been assigned rather than ‘New Zealand’ because of the limitation to the length. Now, let’s see the ways in which we can overcome this problem.

Create a numpy array of arbitrary length.

Solution : While creating the array assign the ‘object’ dtype to it. This lets you have all the behaviors of the python string.

# importing the numpy library as np

import numpy as np

# Create a numpy array

# set the dtype to object

country = np.array(['USA', 'Japan', 'UK', '', 'India', 'China'], dtype = 'object')

# Print the array

print(country)

Create a numpy array of arbitrary length.

Solution : We will use the numpy.astype() function to change the dtype of the given array object.

Now we will change the dtype of the given array object using numpy.astype() function. Then we will assign an arbitrary length string to it.

# Change the dtype of the country

# object to 'U256'

country = country.astype('U256')

# Assign 'New Zealand' to the missing value

country[country == ''] = 'New Zealand'

# Print the array

print(country)

Q9. What happens when you combine two numpy arrays using an operation like addition (+) or multiplication (\*)? What are the conditions for combining two numpy arrays?

If we use another array instead of a scalar, the elements of both arrays will be component-wise combined:

import numpy as np

A = np.array([ [11, 12, 13], [21, 22, 23], [31, 32, 33] ])

B = np.ones((3,3))

print("Adding to arrays: ")

print(A + B)

print("\nMultiplying two arrays: ")

print(A \* (B + 1))

OUTPUT:

Adding to arrays:

[[12. 13. 14.]

[22. 23. 24.]

[32. 33. 34.]]

Multiplying two arrays:

[[22. 24. 26.]

[42. 44. 46.]

[62. 64. 66.]]

Q10. What is the best way to use a Boolean array to mask another array?

how to mask an array using another array in Python. When working with data arrays or data-frames masking can be extremely useful. Masks are an array that contains the list of boolean values for the given condition. The masked array is the arrays that have invalid or missing entries.

Using Masking of arrays we can easily handle the missing, invalid, or unwanted entries in our array or dataset/dataframe. Masking is essential works with the list of Boolean values i.e, True or False which when applied to an original array to return the element of interest, here True refers to the value that satisfies the given condition whereas False refers to values that fail to satisfy the condition.

We can mask the array using another by using the following functions:-

numpy.ma.masked\_where(condition, arr)

numpy.ma.getmask(arr)

numpy.ma.masked\_array(arr, mask=)

where,

condition: condition for masking

arr: arr to be masked

mask: result of masked array

Steps Required:

* Import the library.
* Create a function for masking.
* Masking can be done by following two approaches:-
* Using masked\_where() function: Pass the two array in the function as a parameter then use numpy.ma.masked\_where() function in which pass the condition for masking and array to be masked. In this we are giving the condition for masking by using one array and masking the another array for that condition.
* Using masked\_where(), getmask() and masked\_array() function: Pass the two array in the function as a parameter then use numpy.ma.masked\_where() function in which pass the condition for masking and array to be masked in this we are using the same array for which we are giving condition for making and the array to be masked and store the result in the variable, then use numpy.ma.getmask() function in which pass the result of marked\_where function and store it in the variable named as ‘res\_mask’. Now mask another array using the created mask, for this, we are using numpy.ma.masked\_array() function in which pass the array to be made and the parameter mask=’res\_mask’ for making the array using another array and store it in a variable let be named as ‘masked’.
* Then return the masked from the function.
* Now create the main function
* Create two arrays one for masking another.
* Then call the function as we have created above and pass both the arrays in the function as a parameter and store the result in a variable let named ‘masked’.
* Now for getting the array as a 1-d array we are using numpy.ma.compressed() which passes the masked as a parameter.
* Then print the Masked array.

Masking the first array using the second array:

In the above example, we are masking the first array using the second array on the basis of the condition that each element of the first array mod 7 is true, those elements which satisfy the condition at that index elements are masked in the first array.

Since we have the array1 = [1,2,4,5,7,8,9] and array2 = [10,12,14,5,7,0,13], we have given the condition array2%7 so in array2 element 14, 7 and 0 satisfies the condition, and they are present at index 2,4 and 5 so at the same index in array1 elements are masked so the resultant array we have [4 7 8].

# importing the library

import numpy as np

# function to create masked array

def masking(ar1, ar2):

# masking the array1 by using array2

# where array2 mod 7 is true

mask = np.ma.masked\_where(ar2%7,ar1)

return mask

# main function

if \_\_name\_\_ == '\_\_main\_\_':

# creating two arrays

x = np.array([1,2,4,5,7,8,9])

y = np.array([10,12,14,5,7,0,13])

# calling masking function to get

# masked array

masked = masking(x,y)

# getting the values as 1-d array which

# are non masked

masked\_array = np.ma.compressed(mask)

# printing the resultant array after masking

print(f'Masked Array is:{masked\_array}')

Q11. What are three different ways to get the standard deviation of a wide collection of data using both standard Python and its packages? Sort the three of them by how quickly they execute.

There are many Python statistics libraries out there:

Python’s statistics is a built-in Python library for descriptive statistics. You can use it if your datasets are not too large or if you can’t rely on importing other libraries.

NumPy is a third-party library for numerical computing, optimized for working with single- and multi-dimensional arrays. Its primary type is the array type called ndarray. This library contains many routines for statistical analysis.

SciPy is a third-party library for scientific computing based on NumPy. It offers additional functionality compared to NumPy, including scipy.stats for statistical analysis.

pandas is a third-party library for numerical computing based on NumPy. It excels in handling labeled one-dimensional (1D) data with Series objects and two-dimensional (2D) data with DataFrame objects.

Matplotlib is a third-party library for data visualization. It works well in combination with NumPy, SciPy, and pandas.

Note that, in many cases, Series and DataFrame objects can be used in place of NumPy arrays. Often, you might just pass them to a NumPy or SciPy statistical function. In addition, you can get the unlabeled data from a Series or DataFrame as a np.ndarray object by calling .values or .to\_numpy().

Start by importing all the packages you’ll need:

>>> import math

>>> import statistics

>>> import numpy as np

>>> import scipy.stats

>>> import pandas as pd

Standard Deviation

The sample standard deviation is another measure of data spread. It’s connected to the sample variance, as standard deviation, 𝑠, is the positive square root of the sample variance. The standard deviation is often more convenient than the variance because it has the same unit as the data points. Once you get the variance, you can calculate the standard deviation with pure Python:

>>> std\_ = var\_ \*\* 0.5

>>> std\_

11.099549540409285

Although this solution works, you can also use statistics.stdev():

>>> std\_ = statistics.stdev(x)

>>> std\_

11.099549540409287

Of course, the result is the same as before. Like variance(), stdev() doesn’t calculate the mean if you provide it explicitly as the second argument: statistics.stdev(x, mean\_).

You can get the standard deviation with NumPy in almost the same way. You can use the function std() and the corresponding method .std() to calculate the standard deviation. If there are nan values in the dataset, then they’ll return nan. To ignore nan values, you should use np.nanstd(). You use std(), .std(), and nanstd() from NumPy as you would use var(), .var(), and nanvar():

>>> np.std(y, ddof=1)

11.099549540409285

>>> y.std(ddof=1)

11.099549540409285

>>> np.std(y\_with\_nan, ddof=1)

nan

>>> y\_with\_nan.std(ddof=1)

nan

>>> np.nanstd(y\_with\_nan, ddof=1)

11.099549540409285

Don’t forget to set the delta degrees of freedom to 1!

pd.Series objects also have the method .std() that skips nan by default:

>>> z.std(ddof=1)

11.099549540409285

>>> z\_with\_nan.std(ddof=1)

11.099549540409285

The parameter ddof defaults to 1, so you can omit it. Again, if you want to treat nan values differently, then apply the parameter skipna.

The population standard deviation refers to the entire population. It’s the positive square root of the population variance. You can calculate it just like the sample standard deviation, with the following differences:

Find the square root of the population variance in the pure Python implementation.

Use statistics.pstdev() instead of statistics.stdev().

Specify the parameter ddof=0 if you use NumPy or pandas. In NumPy, you can omit ddof because its default value is 0.

As you can see, you can determine the standard deviation in Python, NumPy, and pandas in almost the same way as you determine the variance. You use different but analogous functions and methods with the same arguments.

The statistics module in python provides functions called stdev() and pstdev() to calculate the standard deviation of a sample dataset. The stdev() function in python only calculates the sample standard deviation whereas the pstdev() function calculates the population standard deviation.

The numpy module of Python provides a function called numpy.std(), used to compute the standard deviation along the specified axis.

NumPy modules are more faster than Python modules.

12. What is the dimensionality of a Boolean mask-generated array?

What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is True .