

# Memory layout

This lesson explains the memory layout using NumPy.

## WE'LL COVER THE FOLLOWING



- Layouts
  - Item layout
    - Example 1
    - Example 2
  - Flattened item layout
    - Example 1
    - Example 2
  - Memory layout (C order, big endian)
    - Example 1
    - Example 2

The [NumPy documentation](#) defines the ndarray class very clearly:

*An instance of class **ndarray** consists of a contiguous one-dimensional segment of computer memory (owned by the array, or by some other object), combined with an indexing scheme that maps  $N$  integers into the location of an item in the block.*

Said differently, an array is mostly a contiguous block of memory whose parts can be accessed using an indexing scheme. Such indexing scheme is in turn defined by a [shape](#) and a [data type](#) and this is precisely what is needed when you define a new array:

```
import numpy as np
Z = np.arange(9).reshape(3,3).astype(np.int16)
print(Z)
```



```
print(Z.itemsize)# returns size of Z in bytes
print(Z.shape)# returns the x dimension and y dimension of Z
print(Z.ndim)# dimension in Z i.e (2 in this case) since the array is 2D
```



Here, we know that itemsize is 2 bytes ( `int16` ), the shape is (3,3) and the number of dimensions is 2.

To calculate the dimension we can also use `len(Z.shape)` .

Furthermore, we can deduce the `strides` of the array that define the number of bytes to step in each dimension when traversing the array.

```
import numpy as np
Z = np.arange(9).reshape(3,3).astype(np.int16)
stride = Z.shape[1]*Z.itemsize, Z.itemsize # store stride of Z
print("Stride(as np.int16):",stride)
print("Z.strides:",Z.strides)
Z = np.arange(9).reshape(3,3).astype(np.int32)
stride= Z.shape[1]*Z.itemsize, Z.itemsize #stores stride of Z
print("Stride(as np.int32):",stride)
print("Z.strides:",Z.strides)
```



Here in this example, we have to skip 2 bytes (1 value) to move to the next column, but 6 bytes (3 values) to get to the same position in the next row. As such, the **strides** for the array `Z` will be **(6, 2)**.

With all this information, we know how to access a specific item (designed by an index tuple) and more precisely, how to compute the start and end offsets:

```
import numpy as np
Z = np.arange(9).reshape(3,3).astype(np.int16)
offset_start = 0
for i in range(Z.ndim):
    offset_start += Z.strides[i] * i #compute the start offset of Z
    offset_end = offset_start + Z.itemsize #compute the end offset of Z

print("Starting offset:", offset_start)
print("Ending offset:", offset_end)
```



Let's see if this is correct using the [tobytes](#) conversion method that construct Python bytes containing the raw data bytes in the array:

```
import numpy as np
Z = np.arange(9).reshape(3, 3).astype(np.int16)
index = 1, 1
print(Z[index].tobytes())
#b'\x04\x00'
offset_start = 0
for i in range(Z.ndim):
    offset_start += Z.strides[i] * index[i]
    offset_end = offset_start + Z.itemsize

print(Z.tobytes()[offset_start:offset_end])#b'\x04\x00'
```

## Layouts #

This array can be actually considered from different perspectives (i.e. layouts):

### Item layout #

Consider item layout as a 2-Dimensional Matrix with x rows and y columns.

#### Example 1 #

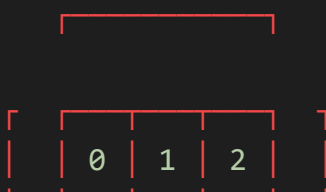
Consider the following example:

```
Z = np.arange(9).reshape(3, 3)
```

Here takes 9 values from 0-8 and is reshaped in 2-D matrix format having dimensions (3 \* 3).

Z has the following item layout:

shape[1]  
(=3)



```

shape[0] | 3 | 4 | 5 | len(Z)
(=3)     |---|---|---| (=3)
         | 6 | 7 | 8 |
         |---|---|---|

```

## Example 2 #

Consider the following example

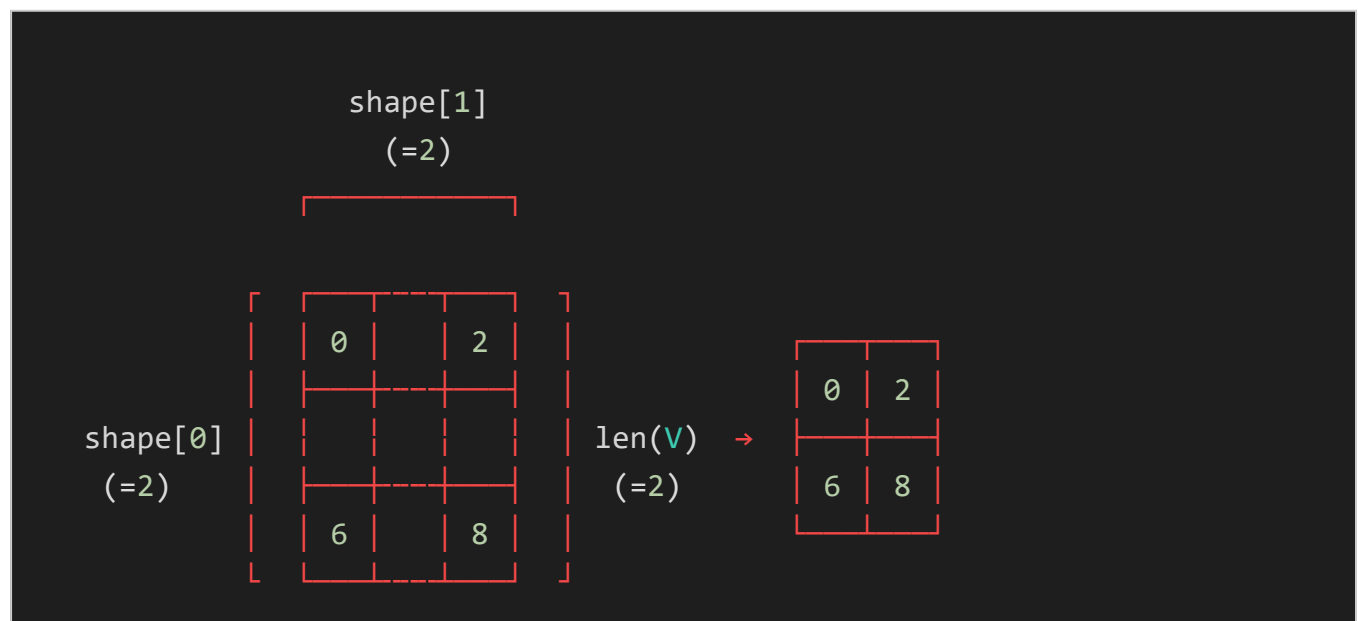
```

Z = np.arange(9).reshape(3,3)
V= Z[:,2,:2])

```

Here Z takes up 9 values from 0-8 and arranges them in a 3 \* 3 matrix V takes values from corners from the grid. i.e V has 4 values.

V has the following item layout:



## Flattened item layout #

Consider flattened item layout as 1-Dimensional Matrix with 1 row and n columns.

## Example 1 #

Consider the following example:

```

Z = np.arange(9)

```

It makes 9 indices in the computer's memory and places values from 0 to 8.

Z has the following flattened item layout:

Z has the following flattened item layout.



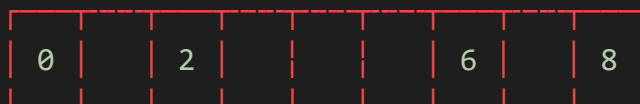
`Z.size`  
(=9)

## Example 2 #

Consider the following example

```
Z = np.arange(9).reshape(3,3).astype(np.int16)
V = Z[::2,::2]
V=V.reshape(1,4)
```

Here Z takes up 9 values from 0-8 and arranges them in a 3 \* 3 matrix V takes values from corners from the grid. i.e V has 4 values. V has the following flattened item layout:



→



`V.size`  
(=4)

## Memory layout (C order, big endian) #

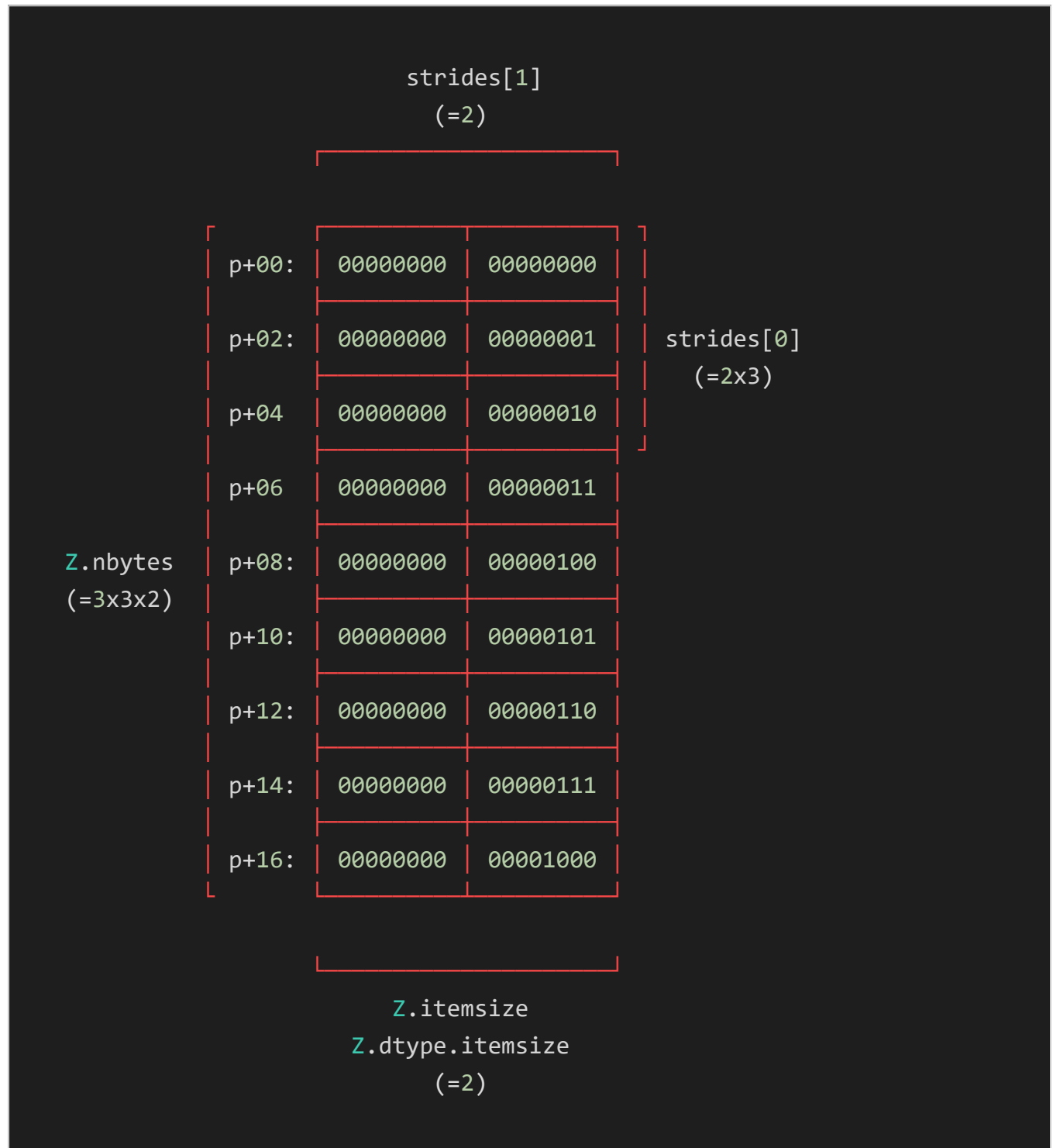
Consider memory layout with rows equal to the number of bytes and columns equal to the number of bytes divided by 8 (i.e Z.itemsize).

## Example 1 #

Consider the following example:

```
Z = np.arange(9).reshape(3, 3).astype(np.int16)
```

Here, the number of rows is 16 and the number of columns is 2. The total number of bytes is  $3 * 3 * 2$  where  $3 * 3$  is the size of the grid and each cell takes 2 bytes, so total 18 bytes. Z has the following memory layout:



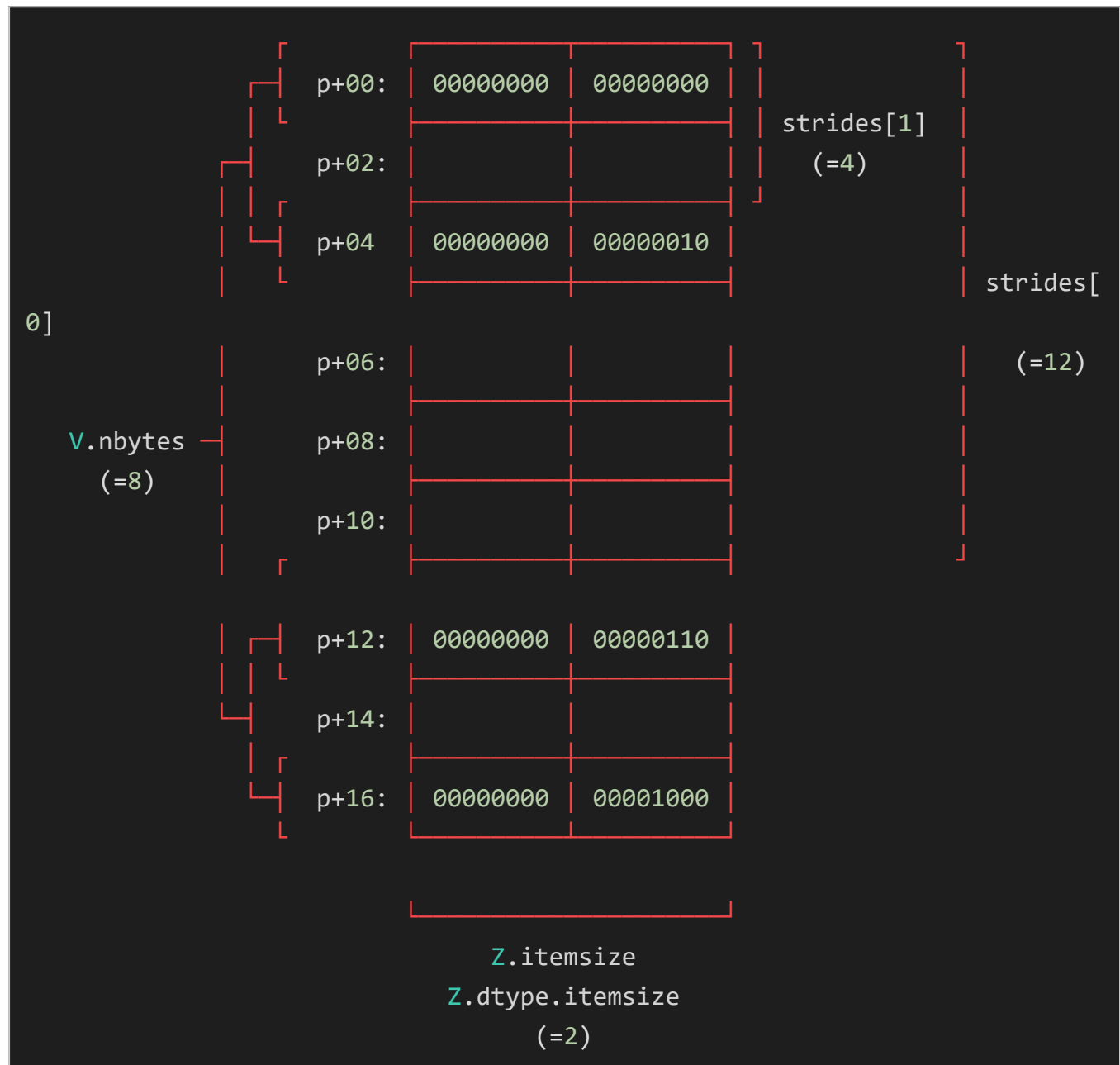
## Example 2 #

Consider the following example

```
Z = np.arange(9).reshape(3,3).astype(np.int16)
V = Z[:, :2, ::2]
```

Here we take a slice of `Z`, the result is a view of the base array `Z`. In the

memory layout below, since the array takes up only 4 value. and each value is 2 bytes so the total bytes are  $2 * 4 = 8$  bytes. V has the following memory layout:



Such a view is specified using a shape, a dtype and strides because strides cannot be deduced anymore from the dtype and shape only.

Solve this Quiz!

1

What is the output of the following code?

```
Z = np.arange(9).reshape(3,3).astype(np.int32)
print(Z.itemsize())
```

```
printf("%d\n", *p);
```

COMPLETED 0%

1 of 2



Now, that we have viewed memory layouts, we'll look at views and copies in the next lesson.