Array-Oriented Programming with NumPy - Part 1

1. Introduction

2. Creating an array using different approaches (Constructors)

3. Indexing and slicing (Getter and Setter)

4. NumPy calculation methods (Reduction)

1. Introduction

The `NumPy` (Numerical Python) library is the favored Python array implementation. It provides a high-performance, feature-rich -dimensional array type called `array`. Array operations are typically one or two orders of magnitude faster than those on `lists`.

The `NumPy` (Numerical Python) library is the favored Python array implementation. It provides a high-performance, feature-rich -dimensional array type called `array`. Array operations are typically one or two orders of magnitude faster than those on `lists`.

Although the built-in `lists` can also possess multiple dimensions and be processed using nested loops. A key advantage of NumPy is "array-oriented programming," which employs ***functional-style programming*** and ***internal iteration*** to make array manipulation concise and straightforward, reducing the likelihood of bugs that can arise from explicitly programmed loops.

The `NumPy` (Numerical Python) library is the favored Python array implementation. It provides a high-performance, feature-rich -dimensional array type called `array`. Array operations are typically one or two orders of magnitude faster than those on `lists`.

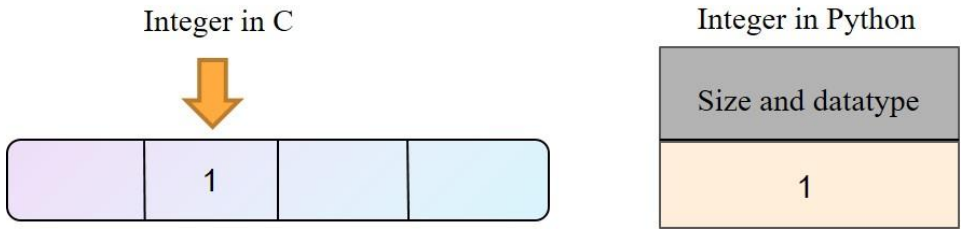
Although the built-in `lists` can also possess multiple dimensions and be processed using nested loops. A key advantage of NumPy is "array-oriented programming," which employs ***functional-style programming*** and ***internal iteration*** to make array manipulation concise and straightforward, reducing the likelihood of bugs that can arise from explicitly programmed loops.

```
V = [1, 9, 2, 8]
s = 3
R = []
for e in V:
    R.append(e*s)
```

```
V = np.array([1, 9, 2, 8])
s = 3
R = s*V
```

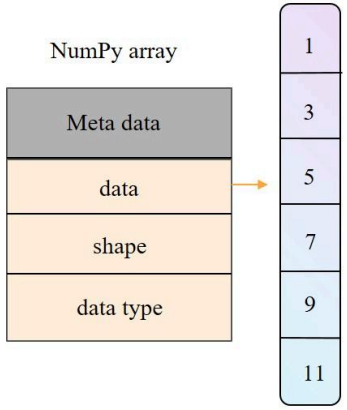
In `Python`, we don't have to declare types or handle memory by hand. Every variable holds more than just the value itself— they also include additional information about the value's type and size:

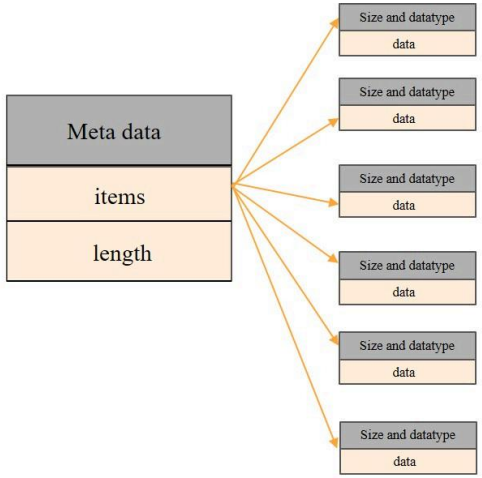
In `Python`, we don't have to declare types or handle memory by hand. Every variable holds more than just the value itself— they also include additional information about the value's type and size:

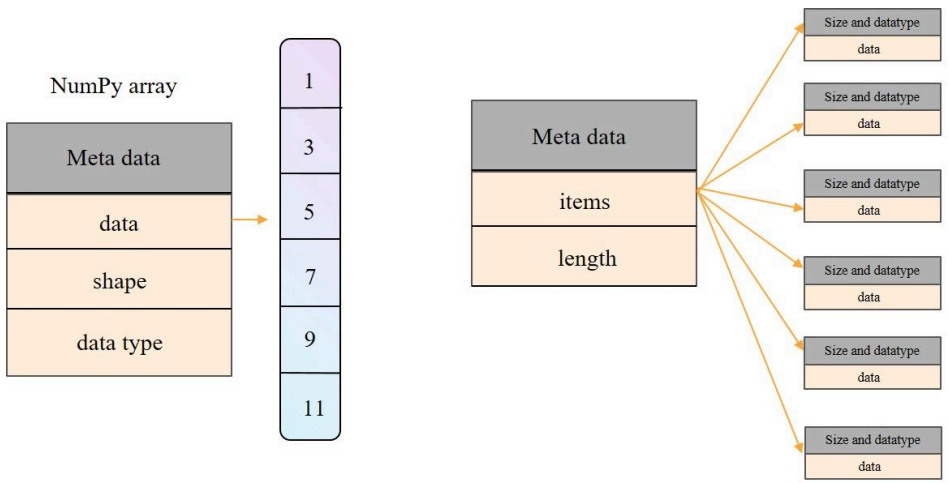


Likewise, a `Python list` is very flexible: it can hold `objects` of many different types. But that flexibility comes at a price — because the interpreter has to know what each element is, every item carries its own notes about type, size, and other details.

When all elements happen to share the same type, most of that extra data is just repeated over and over! A fixed-type `NumPy array` avoids this overhead by recording the type only once and storing all the raw values in one tightly packed block of memory, making it far more efficient than a dynamic-type `list` for large, uniform data.


NumPy array
Meta data
data
shape
data type
1
3
5
7
9
11


Meta data
items
length
Size and datatype
data
Size and datatype
data
Size and datatype
data
Size and datatype
data
Size and datatype
data
Size and datatype
data



From the figure, we can see that at the implementation level, the `array` primarily consists of a single pointer to a contiguous data block. In contrast, the `Python` `list` features a pointer to a block of pointers, each of which points to a `Python` object, such as a `Python` `integer`.

All in all, the primary benefit of the `list` is its flexibility. Since each `list` element is a comprehensive structure containing data and type information, the `list` can accommodate data of any type. While fixed-type `NumPy` `arrays` do not offer this level of adaptability

- They are significantly more efficient for storing and manipulating data.
- In addition, we know that every object consists of data and methods. The `array` object of the `NumPy` package not only provides efficient storage of array-based data but adds to this efficient operation on that data.

In the first step, we need to install `NumPy` as follows:

In the first step, we need to install `NumPy` as follows:

In [2]:
```
package_name = "numpy"

try:
    __import__(package_name)
    print(f"{package_name} is already installed.")
except ImportError:
    print(f"{package_name} not found. Installing...")
    %pip install {package_name}
```

```
numpy is already installed.
```

The official `NumPy` documentation recommends importing the `numpy` module as `np` so that we can access its methods with `np.`:

The official `NumPy` documentation recommends importing the `numpy` module as `np` so that we can access its methods with `np.`:

In [3]:
```
import numpy as np
```

In [4]:
```
display_quiz(path+"list_array.json", max_width=800)
```

What is printed by the following statements?

```
import numpy as np
lst1 = [1, 2, 3]
arr1 = np.array([1, 2, 3])
print(lst1 * 2)
print(arr1 * 2)
```

```
[2, 4, 6]
[2 4 6]
```

```
[1, 2, 3, 1, 2, 3]
[2, 4, 6]
```

```
[1, 2, 3, 1, 2, 3]
[2 4 6]
```

```
[1, 2, 3]
[2 4 6]
```

2. Creating `array` using different approaches (Constructors)

2.1 Creating `array` from fix sequence

The `numpy` module offers numerous functions to create arrays. In this case, we employ the `array()` function, which accepts a sequence of elements and returns a new `array` containing the input elements. For instance, let's pass a `list`:

The `numpy` module offers numerous functions to create arrays. In this case, we employ the `array()` function, which accepts a sequence of elements and returns a new `array` containing the input elements. For instance, let's pass a `list`:

In [5]:
```
import numpy as np
numbers = np.array([2, 3, 5, 7, 11])
numbers, type(numbers)
```

Out[5]:
```
(array([ 2,  3,  5,  7, 11]), numpy.ndarray)
```

The `numpy` module offers numerous functions to create arrays. In this case, we employ the `array()` function, which accepts a sequence of elements and returns a new `array` containing the input elements. For instance, let's pass a `list`:

In [5]:
```
import numpy as np
numbers = np.array([2, 3, 5, 7, 11])
numbers, type(numbers)
```

Out[5]:
```
(array([ 2,  3,  5,  7, 11]), numpy.ndarray)
```

The `array()` function copies its argument's contents into the `array`. Note that the type is `numpy.ndarray` and all the output will prefix the data with the keyword `array`.

The `array()` function copies its argument's dimensions. Let's create an `array` from a two-row-by-three-column nested `list`:

The `array()` function copies its argument's dimensions. Let's create an `array` from a two-row-by-three-column nested `list`:

In [6]:
```
np.array([[1, 2, 3], [4, 5, 6]]), type(np.array([[1, 2, 3], [4, 5, 6]]))
```

Out[6]:
```
(array([[1, 2, 3],
        [4, 5, 6]]),
 numpy.ndarray)
```

The `array()` function copies its argument's dimensions. Let's create an `array` from a two-row-by-three-column nested `list`:

In [6]:
```
np.array([[1, 2, 3], [4, 5, 6]]), type(np.array([[1, 2, 3], [4, 5, 6]]))
```

Out[6]:
```
(array([[1, 2, 3],
        [4, 5, 6]]),
 numpy.ndarray)
```

A 2D array is a sequence of 1D arrays that represent each row.

`array` Attributes

The `array` function determines an array's element type from its argument's elements. We can check the element type with an array's `dtype` attribute:

The `array` function determines an array's element type from its argument's elements. We can check the element type with an array's `dtype` attribute:

In [7]:
```
integers = np.array([[1, 2, 3], [4, 5, 6]])
floats = np.array([0.0, 0.1, 0.2, 0.3, 0.4])

integers.dtype, floats.dtype
```

Out[7]:
```
(dtype('int32'), dtype('float64'))
```

The `array` function determines an array's element type from its argument's elements. We can check the element type with an array's `dtype` attribute:

In [7]:
```
integers = np.array([[1, 2, 3], [4, 5, 6]])
floats = np.array([0.0, 0.1, 0.2, 0.3, 0.4])

integers.dtype, floats.dtype
```

Out[7]:
```
(dtype('int32'), dtype('float64'))
```

In the upcoming section, we will notice that several array-creation functions include a `dtype` keyword argument, allowing us to define an array's element type.

The attribute `ndim` contains an array's number of dimensions and the attribute `shape` contains a `tuple` specifying an array's dimensions:

The attribute `ndim` contains an array's number of dimensions and the attribute `shape` contains a `tuple` specifying an array's dimensions:

In [8]:
```
print(integers.ndim)
print(floats.ndim)
```

```
2
1
```

The attribute `ndim` contains an array's number of dimensions and the attribute `shape` contains a `tuple` specifying an array's dimensions:

In [8]:
```
print(integers.ndim)
print(floats.ndim)
```

```
2
1
```

In [9]:
```
print(integers.shape)
print(floats.shape)
```

```
(2, 3)
(5,)
```

The attribute `ndim` contains an array's number of dimensions and the attribute `shape` contains a `tuple` specifying an array's dimensions:

In [8]:
```
print(integers.ndim)
print(floats.ndim)
```

```
2
1
```

In [9]:
```
print(integers.shape)
print(floats.shape)
```

```
(2, 3)
(5,)
```

Here, integers have 2 rows and 3 columns (6 elements) and floats are one-dimensional, containing 5 floating numbers.

We can view an array's total number of elements with the attribute `size` and the number of bytes required to store each element with `itemsize`:

We can view an array's total number of elements with the attribute `size` and the number of bytes required to store each element with `itemsize`:

In [10]:
```
print(integers.size)
print(integers.itemsize)
print(floats.size)
print(floats.itemsize)
```

```
6
4
5
8
```

We can view an array's total number of elements with the attribute `size` and the number of bytes required to store each element with `itemsize`:

In [10]:
```
print(integers.size)
print(integers.itemsize)
print(floats.size)
print(floats.itemsize)
```

```
6
4
5
8
```

Note that the `size` of the integers is the result of multiplying the values in the `tuple` — two rows with three elements each, totaling six elements. In each instance, `itemsize` is 4 because integers comprise `int32` values, and as floats consist of `float64` values.

2.2 Filling `array` with specific values

`NumPy` offers the functions `zeros()`, `ones()`, and `full()` for creating arrays filled with 0s, 1s, or a specified value, respectively. By default, `zeros()` and `ones()` generate arrays containing `float64` values. We will demonstrate how to customize the element type shortly. The first argument for these functions should be either an `integer` or a `tuple` of integers defining the desired dimensions. When given an integer, each function returns a one-dimensional array containing the specified number of elements:

`NumPy` offers the functions `zeros()`, `ones()`, and `full()` for creating arrays filled with 0s, 1s, or a specified value, respectively. By default, `zeros()` and `ones()` generate arrays containing `float64` values. We will demonstrate how to customize the element type shortly. The first argument for these functions should be either an `integer` or a `tuple` of integers defining the desired dimensions. When given an integer, each function returns a one-dimensional array containing the specified number of elements:

In [11]:
```
np.zeros(5)
```

Out[11]:
```
array([0., 0., 0., 0., 0.])
```

When provided with a `tuple` of integers, these functions return a multidimensional array featuring the specified dimensions. We can define the array's element type using the `dtype` keyword argument for the `zeros()` and `ones()` functions:

When provided with a `tuple` of integers, these functions return a multidimensional array featuring the specified dimensions. We can define the array's element type using the `dtype` keyword argument for the `zeros()` and `ones()` functions:

In [12]:
```
np.ones((2, 4), dtype=np.int32)
```

Out[12]:
```
array([[1, 1, 1, 1],
       [1, 1, 1, 1]])
```

The `array` returned by `full()` contains elements with the second argument's value and type:

The `array` returned by `full()` contains elements with the second argument's value and type:

In [13]:
```
np.full((3, 5), 13+2j), np.full((3, 5), 13+2j).dtype
```

Out[13]:
```
(array([[13.+2.j, 13.+2.j, 13.+2.j, 13.+2.j, 13.+2.j],
        [13.+2.j, 13.+2.j, 13.+2.j, 13.+2.j, 13.+2.j],
        [13.+2.j, 13.+2.j, 13.+2.j, 13.+2.j, 13.+2.j]]),
 dtype('complex128'))
```

2.3 Creating `array` from sequence generated by different methods

Creating sequence with fix step by `arange()`

We can employ `NumPy`'s `arange()` function to create integer ranges, similar to using the built-in `range()` function. The first two arguments of the function determine the starting and ending values of the range, with the ending value excluded from the array. The optional third argument represents the step size which has a default value of 1:

We can employ `NumPy`'s `arange()` function to create integer ranges, similar to using the built-in `range()` function. The first two arguments of the function determine the starting and ending values of the range, with the ending value excluded from the array. The optional third argument represents the step size which has a default value of 1:

In [14]:
```
np.arange(5)
```

Out[14]:
```
array([0, 1, 2, 3, 4])
```

We can employ `NumPy`'s `arange()` function to create integer ranges, similar to using the built-in `range()` function. The first two arguments of the function determine the starting and ending values of the range, with the ending value excluded from the array. The optional third argument represents the step size which has a default value of 1:

In [14]:
```
np.arange(5)
```

Out[14]:
```
array([0, 1, 2, 3, 4])
```

In [15]:
```
np.arange(5, 10)
```

Out[15]:
```
array([5, 6, 7, 8, 9])
```

We can employ `NumPy`'s `arange()` function to create integer ranges, similar to using the built-in `range()` function. The first two arguments of the function determine the starting and ending values of the range, with the ending value excluded from the array. The optional third argument represents the step size which has a default value of 1:

In [14]:
```
np.arange(5)
```

Out[14]:
```
array([0, 1, 2, 3, 4])
```

In [15]:
```
np.arange(5, 10)
```

Out[15]:
```
array([5, 6, 7, 8, 9])
```

In [16]:
```
np.arange(10, 1, -2)
```

Out[16]:
```
array([10,  8,  6,  4,  2])
```

We can employ `NumPy`'s `arange()` function to create integer ranges, similar to using the built-in `range()` function. The first two arguments of the function determine the starting and ending values of the range, with the ending value excluded from the array. The optional third argument represents the step size which has a default value of 1:

In [14]:
```
np.arange(5)
```

Out[14]:
```
array([0, 1, 2, 3, 4])
```

In [15]:
```
np.arange(5, 10)
```

Out[15]:
```
array([5, 6, 7, 8, 9])
```

In [16]:
```
np.arange(10, 1, -2)
```

Out[16]:
```
array([10,  8,  6,  4,  2])
```

Note that it is the same as `range()`, which takes three arguments `numpy.arange(start, stop, step)` and the first and third arguments can be omitted.

Creating sequence with fix sample number by `linspace()`

Additionally, we can generate evenly spaced floating-point ranges using `NumPy`'s `linspace()` function. The first two arguments of the function determine the starting and ending values of the range, with the ending value included in the `array`. The optional keyword argument `num` designates the number of evenly spaced values to create:

Additionally, we can generate evenly spaced floating-point ranges using `NumPy`'s `linspace()` function. The first two arguments of the function determine the starting and ending values of the range, with the ending value included in the `array`. The optional keyword argument `num` designates the number of evenly spaced values to create:

In [17]:
```
np.linspace(0.0, 1.0, num=5)
```

Out[17]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Reshaping an `array`

We can also first create an `array` using the previous methods and then utilize the `array` method `reshape()` to convert the one-dimensional array into a multidimensional array. Let's generate an array containing values from 1 to 20 and then reshape it into a matrix with four rows and five columns:

We can also first create an `array` using the previous methods and then utilize the `array` method `reshape()` to convert the one-dimensional array into a multidimensional array. Let's generate an array containing values from 1 to 20 and then reshape it into a matrix with four rows and five columns:

In [18]:
```
np.arange(1, 21).reshape(4, 5)
```

Out[18]:
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20]])
```

We can also first create an `array` using the previous methods and then utilize the `array` method `reshape()` to convert the one-dimensional array into a multidimensional array. Let's generate an array containing values from 1 to 20 and then reshape it into a matrix with four rows and five columns:

In [18]:
```
np.arange(1, 21).reshape(4, 5)
```

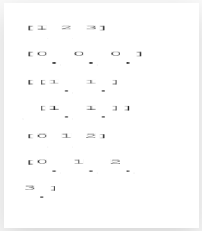
Out[18]:
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20]])
```

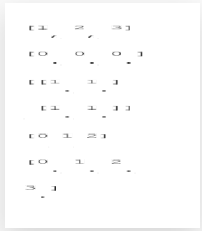
Note the ***chained method*** calls in the previous example. Initially, `arange()` generates an array containing values 1 to 20. Then, we invoke `reshape()` on that array to obtain the displayed 4-by-5 array. We can `reshape()` any array as long as the new shape contains the same number of elements as the original. Thus, a six-element one-dimensional array can be transformed into a 3-by-2 or 2-by-3 array, and vice versa!

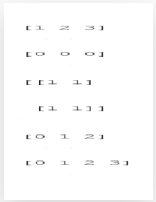
In [19]:
```
display_quiz(path+"constructors.json", max_width=850)
```

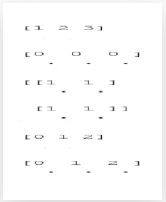
What is printed by the following statements?

```
import numpy as np
a = np.array([1, 2, 3])
b = np.zeros(3)
c = np.ones((2, 2))
d = np.arange(3)
e = np.linspace(0, 3, 4)
print(a)
print(b)
print(c)
print(d)
print(e)
```









Example 1: `List` vs. `array` performance: Introducing `%%timeit`

Most `array` operations execute significantly faster than corresponding `list` operations. To demonstrate, we'll use the `%%timeit` magic command, which benchmarks the average duration of operations.

Most `array` operations execute significantly faster than corresponding `list` operations. To demonstrate, we'll use the `%%timeit` magic command, which benchmarks the average duration of operations.

In [20]:
```
import random
```

Most `array` operations execute significantly faster than corresponding `list` operations. To demonstrate, we'll use the `%%timeit` magic command, which benchmarks the average duration of operations.

In [20]:
```
import random
```

Here, let's use the `random` module's `randint()` function with a list comprehension to create a list of six million die rolls and time the operation using `%%timeit`:

Most `array` operations execute significantly faster than corresponding `list` operations. To demonstrate, we'll use the `%%timeit` magic command, which benchmarks the average duration of operations.

In [20]:
```
import random
```

Here, let's use the `random` module's `randint()` function with a list comprehension to create a list of six million die rolls and time the operation using `%%timeit`:

In [21]:
```
%%timeit
rolls_list = [random.randint(1, 6) for i in range(0, 6_000_000)] #_ is use to
```

```
3.66 s ± 11.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Now, let's use the `randint()` function from the `numpy.random` module to create an array

Now, let's use the `randint()` function from the `numpy.random` module to create an array

In [22]:
```
%%timeit
rolls_array = np.random.randint(1, 7, 6_000_000)
```

```
44.1 ms ± 111 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

3. Indexing and slicing (Getter and Setter)

One-dimensional `arrays` can be *indexed* and *sliced* using the same syntax and techniques applied when handling other sequence data types, such as built-in `lists` or `tuples`.

One-dimensional `arrays` can be *indexed* and *sliced* using the same syntax and techniques applied when handling other sequence data types, such as built-in `lists` or `tuples`.

To select an element in a two-dimensional array, specify two indices containing the element's row and column indices in square brackets:

One-dimensional `arrays` can be *indexed* and *sliced* using the same syntax and techniques applied when handling other sequence data types, such as built-in `lists` or `tuples`.

To select an element in a two-dimensional array, specify two indices containing the element's row and column indices in square brackets:

In [23]:
```
grades = np.array([[87, 96, 70], [60, 87, 90],
                   [94, 77, 92], [100, 81, 82]])
grades
```

Out[23]:
```
array([[ 87,  96,  70],
       [ 60,  87,  90],
       [ 94,  77,  92],
       [100,  81,  82]])
```

One-dimensional `arrays` can be *indexed* and *sliced* using the same syntax and techniques applied when handling other sequence data types, such as built-in `lists` or `tuples`.

To select an element in a two-dimensional array, specify two indices containing the element's row and column indices in square brackets:

In [23]:
```
grades = np.array([[87, 96, 70], [60, 87, 90],
                   [94, 77, 92], [100, 81, 82]])
grades
```

Out[23]:
```
array([[ 87,  96,  70],
       [ 60,  87,  90],
       [ 94,  77,  92],
       [100,  81,  82]])
```

In [24]:
```
grades[0, 1]  # row 0, column 1
```

Out[24]:
```
96
```

To select a single row, we can specify only one index in square brackets:

To select a single row, we can specify only one index in square brackets:

In [25]:
```
grades, grades[1]
```

Out[25]:
```
(array([[ 87,  96,  70],
        [ 60,  87,  90],
        [ 94,  77,  92],
        [100,  81,  82]]),
 array([60, 87, 90]))
```

To select a single row, we can specify only one index in square brackets:

In [25]:
```
grades, grades[1]
```

Out[25]:
```
(array([[ 87,  96,  70],
        [ 60,  87,  90],
        [ 94,  77,  92],
        [100,  81,  82]]),
 array([60, 87, 90]))
```

To select multiple sequential rows, use slice notation:

To select a single row, we can specify only one index in square brackets:

In [25]:
```
grades, grades[1]
```

Out[25]:
```
(array([[ 87,  96,  70],
        [ 60,  87,  90],
        [ 94,  77,  92],
        [100,  81,  82]]),
 array([60, 87, 90]))
```

To select multiple sequential rows, use slice notation:

In [26]:
```
grades[0:2]
```

Out[26]:
```
array([[87, 96, 70],
       [60, 87, 90]])
```

To select a single row, we can specify only one index in square brackets:

In [25]:
```
grades, grades[1]
```

Out[25]:
```
(array([[ 87,  96,  70],
        [ 60,  87,  90],
        [ 94,  77,  92],
        [100,  81,  82]]),
 array([60, 87, 90]))
```

To select multiple sequential rows, use slice notation:

In [26]:
```
grades[0:2]
```

Out[26]:
```
array([[87, 96, 70],
       [60, 87, 90]])
```

To select multiple non-sequential rows, use a list of row indices which is called ***fancy indexing***:

To select a single row, we can specify only one index in square brackets:

In [25]:
```
grades, grades[1]
```

Out[25]:
```
(array([[ 87,  96,  70],
        [ 60,  87,  90],
        [ 94,  77,  92],
        [100,  81,  82]]),
 array([60, 87, 90]))
```

To select multiple sequential rows, use slice notation:

In [26]:
```
grades[0:2]
```

Out[26]:
```
array([[87, 96, 70],
       [60, 87, 90]])
```

To select multiple non-sequential rows, use a list of row indices which is called ***fancy indexing***:

In [27]:
```
grades[[1, 3]]
```

Out[27]:
```
array([[ 60,  87,  90],
       [100,  81,  82]])
```

Let's select only the elements in the first column:

Let's select only the elements in the first column:

In [28]:
```
grades, grades[:, 0]
```

Out[28]:
```
(array([[ 87,  96,  70],
        [ 60,  87,  90],
        [ 94,  77,  92],
        [100,  81,  82]]),
 array([ 87,  60,  94, 100]))
```

Let's select only the elements in the first column:

In [28]:
```
grades, grades[:, 0]
```

Out[28]:
```
(array([[ 87,  96,  70],
        [ 60,  87,  90],
        [ 94,  77,  92],
        [100,  81,  82]]),
 array([ 87,  60,  94, 100]))
```

The 0 after the comma signifies that we are selecting only column 0. The `:` before the comma indicates which rows within that column to choose. In this instance, `:` is a slice representing all rows. We can also select consecutive columns using a slice:

Let's select only the elements in the first column:

In [28]:
```
grades, grades[:, 0]
```

Out[28]:
```
(array([[ 87,  96,  70],
        [ 60,  87,  90],
        [ 94,  77,  92],
        [100,  81,  82]]),
 array([ 87,  60,  94, 100]))
```

The 0 after the comma signifies that we are selecting only column 0. The `:` before the comma indicates which rows within that column to choose. In this instance, `:` is a slice representing all rows. We can also select consecutive columns using a slice:

In [29]:
```
grades[:, 1:3]
```

Out[29]:
```
array([[96, 70],
       [87, 90],
       [77, 92],
       [81, 82]])
```

or specific columns with fancy indexing using a list of column indices:

or specific columns with fancy indexing using a list of column indices:

In [30]:
```
grades, grades[:, [0, 2]]
```

Out[30]:
```
(array([[ 87,  96,  70],
        [ 60,  87,  90],
        [ 94,  77,  92],
        [100,  81,  82]]),
 array([[ 87,  70],
        [ 60,  90],
        [ 94,  92],
        [100,  82]]))
```

`array` is mutable. Therefore, if we want to modify the value of the array, we can use the previous method and put the result on the left-hand side:

`array` is mutable. Therefore, if we want to modify the value of the array, we can use the previous method and put the result on the left-hand side:

In [31]:
```
print(grades)
grades[3, 2] = 42
grades
```

```
[[ 87  96  70]
 [ 60  87  90]
 [ 94  77  92]
 [100  81  82]]
```

Out[31]:
```
array([[ 87,  96,  70],
       [ 60,  87,  90],
       [ 94,  77,  92],
       [100,  81,  42]])
```

Views: Shallow copies

Views are objects that see the data in other objects, instead of having their own copies of the data. Views are also referred to as ***shallow copies***. Several `array` methods and slicing operations generate views of an `array`'s data. The `array` method `view()` returns a new `array` object with a view of the original `array` object's data. First, let's create an `array` and a view of that `array`:

Views are objects that see the data in other objects, instead of having their own copies of the data. Views are also referred to as ***shallow copies***. Several `array` methods and slicing operations generate views of an `array`'s data. The `array` method `view()` returns a new `array` object with a view of the original `array` object's data. First, let's create an `array` and a view of that `array`:

In [32]:
```
numbers = np.arange(1, 6)
numbers2 = numbers.view()
```

Views are objects that see the data in other objects, instead of having their own copies of the data. Views are also referred to as ***shallow copies***. Several `array` methods and slicing operations generate views of an `array`'s data. The `array` method `view()` returns a new `array` object with a view of the original `array` object's data. First, let's create an `array` and a view of that `array`:

In [32]:
```
numbers = np.arange(1, 6)
numbers2 = numbers.view()
```

We can use the built-in `id()` function to verify that `numbers` and `numbers2` are different objects:

Views are objects that see the data in other objects, instead of having their own copies of the data. Views are also referred to as ***shallow copies***. Several `array` methods and slicing operations generate views of an `array`'s data. The `array` method `view()` returns a new `array` object with a view of the original `array` object's data. First, let's create an `array` and a view of that `array`:

In [32]:
```
numbers = np.arange(1, 6)
numbers2 = numbers.view()
```

We can use the built-in `id()` function to verify that `numbers` and `numbers2` are different objects:

In [33]:
```
id(numbers), id(numbers2)
```

Out[33]:
```
(2335501871888, 2335501872368)
```

`NumPy` also has a handy function called `shares_memory()` that can be utilized in this scenario:

`NumPy` also has a handy function called `shares_memory()` that can be utilized in this scenario:

In [34]:
```
np.shares_memory(numbers, numbers2)
```

Out[34]:
```
True
```

`NumPy` also has a handy function called `shares_memory()` that can be utilized in this scenario:

In [34]:
```
np.shares_memory(numbers, numbers2)
```

Out[34]:
```
True
```

To prove that `numbers2` views the same data as `numbers`, let's modify an element in `numbers`, then display both arrays:

`NumPy` also has a handy function called `shares_memory()` that can be utilized in this scenario:

In [34]:
```
np.shares_memory(numbers, numbers2)
```

Out[34]:
```
True
```

To prove that `numbers2` views the same data as `numbers`, let's modify an element in `numbers`, then display both arrays:

In [35]:
```
numbers[1] *= 10
numbers
```

Out[35]:
```
array([ 1, 20,  3,  4,  5])
```

`NumPy` also has a handy function called `shares_memory()` that can be utilized in this scenario:

In [34]:
```
np.shares_memory(numbers, numbers2)
```

Out[34]:
```
True
```

To prove that `numbers2` views the same data as `numbers`, let's modify an element in `numbers`, then display both arrays:

In [35]:
```
numbers[1] *= 10
numbers
```

Out[35]:
```
array([ 1, 20,  3,  4,  5])
```

In [36]:
```
numbers2
```

Out[36]:
```
array([ 1, 20,  3,  4,  5])
```

Similarly, changing a value in the view also changes that value in the original array:

Similarly, changing a value in the view also changes that value in the original array:

In [37]:
```
numbers2[1] /= 5
numbers, numbers2
```

Out[37]:
```
(array([1, 4, 3, 4, 5]), array([1, 4, 3, 4, 5]))
```

Similarly, changing a value in the view also changes that value in the original array:

In [37]:
```
numbers2[1] /= 5
numbers, numbers2
```

Out[37]:
```
(array([1, 4, 3, 4, 5]), array([1, 4, 3, 4, 5]))
```

Slices also create views. Let's make `numbers2` a slice that views only the first three elements of numbers:

Similarly, changing a value in the view also changes that value in the original array:

In [37]:
```
numbers2[1] /= 5
numbers, numbers2
```

Out[37]:
```
(array([1, 4, 3, 4, 5]), array([1, 4, 3, 4, 5]))
```

Slices also create views. Let's make `numbers2` a slice that views only the first three elements of numbers:

In [38]:
```
numbers2 = numbers[0:3]
numbers2
```

Out[38]:
```
array([1, 4, 3])
```

Now, let's modify an element both arrays share, then display them. Again, we see that `numbers2` is a view of `numbers`:

Now, let's modify an element both arrays share, then display them. Again, we see that `numbers2` is a view of `numbers`:

In [39]:
```
numbers[1] *= 20
numbers
```

Out[39]:
```
array([ 1, 80,  3,  4,  5])
```

Now, let's modify an element both arrays share, then display them. Again, we see that `numbers2` is a view of `numbers`:

In [39]:
```
numbers[1] *= 20
numbers
```

Out[39]:
```
array([ 1, 80,  3,  4,  5])
```

In [40]:
```
numbers2
```

Out[40]:
```
array([ 1, 80,  3])
```

Now, let's modify an element both arrays share, then display them. Again, we see that `numbers2` is a view of `numbers`:

In [39]:
```
numbers[1] *= 20
numbers
```

Out[39]:
```
array([ 1, 80,  3,  4,  5])
```

In [40]:
```
numbers2
```

Out[40]:
```
array([ 1, 80,  3])
```

Note that this behavior is different from `list`, where the slicing will create a new sub `list`!

Deep Copies

While views are distinct `array` objects, they save memory by sharing element data with other `arrays`. Nonetheless, when dealing with mutable values, it is occasionally essential to create a ***deep copy*** containing independent copies of the original data.

While views are distinct `array` objects, they save memory by sharing element data with other `arrays`. Nonetheless, when dealing with mutable values, it is occasionally essential to create a ***deep copy*** containing independent copies of the original data.

The `array` method `copy()` returns a new `array` object with a deep copy of the original `array` object's data. First, let's create an `array` and a deep copy of that `array`:

While views are distinct `array` objects, they save memory by sharing element data with other `arrays`. Nonetheless, when dealing with mutable values, it is occasionally essential to create a ***deep copy*** containing independent copies of the original data.

The `array` method `copy()` returns a new `array` object with a deep copy of the original `array` object's data. First, let's create an `array` and a deep copy of that `array`:

In [41]:
```
numbers = np.arange(1, 6)
numbers2 = numbers.copy()
```

To prove that `numbers2` has a separate copy of the data in `numbers`, let's modify an element in `numbers`, then display both arrays:

To prove that `numbers2` has a separate copy of the data in `numbers`, let's modify an element in `numbers`, then display both arrays:

In [42]:
```
numbers[1] *= 5
numbers
```

Out[42]:
```
array([ 1, 10,  3,  4,  5])
```

To prove that `numbers2` has a separate copy of the data in `numbers`, let's modify an element in `numbers`, then display both arrays:

In [42]:
```
numbers[1] *= 5
numbers
```

Out[42]:
```
array([ 1, 10,  3,  4,  5])
```

In [43]:
```
numbers2
```

Out[43]:
```
array([1, 2, 3, 4, 5])
```

In [44]:
```
display_quiz(path+"view_copy.json", max_width=850)
```

What is printed by the following statements?

```
import numpy as np

a = np.array([1, 2, 3])
b = a               # b references the same array as a
c = a.view()        # c is a view of a
d = a.copy()        # d is a copy of a

b[0] = 10           # modifies a and b
c[1] = 20           # modifies a and c
d[2] = 30           # modifies only d

print(a)
print(b)
print(c)
print(d)
```

```
[10  2  3]
[10  2  3]
[10 20  3]
[ 1  2 30]
```

```
[10 20  3]
[10 20  3]
[10 20  3]
[ 1  2 30]
```

```
[ 1  2  3]
[10 20  3]
[10 20  3]
[10 20 30]
```

```
[10 20  3]
[10 20  3]
[ 1  2  3]
[ 1  2 30]
```

More about Reshaping and Transposing

We've used `array` method `reshape()` to produce two-dimensional arrays from one-dimensional ranges. `NumPy` provides various other ways to reshape arrays.

Both the `reshape()` and `resize()` array methods allow us to alter an array's dimensions. The `reshape()` method returns a view (shallow copy) of the original array with updated dimensions, leaving the original array unaltered:

Both the `reshape()` and `resize()` array methods allow us to alter an array's dimensions. The `reshape()` method returns a view (shallow copy) of the original array with updated dimensions, leaving the original array unaltered:

In [45]:
```
grades = np.array([[87, 96, 70], [99, 87, 90]])
grades
```

Out[45]:
```
array([[87, 96, 70],
       [99, 87, 90]])
```

Both the `reshape()` and `resize()` array methods allow us to alter an array's dimensions. The `reshape()` method returns a view (shallow copy) of the original array with updated dimensions, leaving the original array unaltered:

In [45]:
```
grades = np.array([[87, 96, 70], [99, 87, 90]])
grades
```

Out[45]:
```
array([[87, 96, 70],
       [99, 87, 90]])
```

In [46]:
```
grades2 = grades.reshape(1, 6)
```

Both the `reshape()` and `resize()` array methods allow us to alter an array's dimensions. The `reshape()` method returns a view (shallow copy) of the original array with updated dimensions, leaving the original array unaltered:

In [45]:
```
grades = np.array([[87, 96, 70], [99, 87, 90]])
grades
```

Out[45]:
```
array([[87, 96, 70],
       [99, 87, 90]])
```

In [46]:
```
grades2 = grades.reshape(1, 6)
```

In [47]:
```
grades2[0, 0] = 0
grades2, grades
```

Out[47]:
```
(array([[ 0, 96, 70, 99, 87, 90]]),
 array([[ 0, 96, 70],
        [99, 87, 90]]))
```

A widely used technique involves using `-1` to specify the shape in `reshape()`. The length of the dimension set to `-1` is automatically deduced based on the specified values of other dimensions:

A widely used technique involves using `-1` to specify the shape in `reshape()`. The length of the dimension set to `-1` is automatically deduced based on the specified values of other dimensions:

In [48]:
```
grades, grades.reshape(-1, 3) # Same as grades.reshape(2, 3)
```

Out[48]:
```
(array([[ 0, 96, 70],
        [99, 87, 90]]),
 array([[ 0, 96, 70],
        [99, 87, 90]]))
```

A widely used technique involves using `-1` to specify the shape in `reshape()`. The length of the dimension set to `-1` is automatically deduced based on the specified values of other dimensions:

In [48]:
```
grades, grades.reshape(-1, 3) # Same as grades.reshape(2, 3)
```

Out[48]:
```
(array([[ 0, 96, 70],
        [99, 87, 90]]),
 array([[ 0, 96, 70],
        [99, 87, 90]]))
```

Method `resize()`, on the other hand, modifies the original `array`'s shape in-place:

A widely used technique involves using `-1` to specify the shape in `reshape()`. The length of the dimension set to `-1` is automatically deduced based on the specified values of other dimensions:

In [48]:
```
grades, grades.reshape(-1, 3) # Same as grades.reshape(2, 3)
```

Out[48]:
```
(array([[ 0, 96, 70],
        [99, 87, 90]]),
 array([[ 0, 96, 70],
        [99, 87, 90]]))
```

Method `resize()`, on the other hand, modifies the original `array`'s shape in-place:

In [49]:
```
grades.resize(1, 6)
grades
```

Out[49]:
```
array([[ 0, 96, 70, 99, 87, 90]])
```

We can also do the opposite operation, which takes a multidimensional array and flatten it into a single dimension with the methods `flatten()`. Method `flatten()` deep copies the original array's data:

We can also do the opposite operation, which takes a multidimensional array and flatten it into a single dimension with the methods `flatten()`. Method `flatten()` deep copies the original array's data:

In [50]:
```
grades = np.array([[87, 96, 70], [99, 87, 90]])
grades
```

Out[50]:
```
array([[87, 96, 70],
       [99, 87, 90]])
```

We can also do the opposite operation, which takes a multidimensional array and flatten it into a single dimension with the methods `flatten()`. Method `flatten()` deep copies the original array's data:

In [50]:
```
grades = np.array([[87, 96, 70], [99, 87, 90]])
grades
```

Out[50]:
```
array([[87, 96, 70],
       [99, 87, 90]])
```

In [51]:
```
flattened = grades.flatten()
flattened
```

Out[51]:
```
array([87, 96, 70, 99, 87, 90])
```

We can also do the opposite operation, which takes a multidimensional array and flatten it into a single dimension with the methods `flatten()`. Method `flatten()` deep copies the original array's data:

In [50]:
```
grades = np.array([[87, 96, 70], [99, 87, 90]])
grades
```

Out[50]:
```
array([[87, 96, 70],
       [99, 87, 90]])
```

In [51]:
```
flattened = grades.flatten()
flattened
```

Out[51]:
```
array([87, 96, 70, 99, 87, 90])
```

In [52]:
```
flattened[0] = 100
grades # Original array does not change
```

Out[52]:
```
array([[87, 96, 70],
       [99, 87, 90]])
```

Additionally, we can transpose an `array`'s rows and columns, the `T` attribute returns a transposed view of the array.

Assume that the original `grades array` presents two students' grades (the rows) across three exams (the columns). Let's transpose the rows and columns to examine the data as the grades for three exams (the rows) taken by two students (the columns):

Additionally, we can transpose an `array`'s rows and columns, the `T` attribute returns a transposed view of the array.

Assume that the original `grades` `array` presents two students' grades (the rows) across three exams (the columns). Let's transpose the rows and columns to examine the data as the grades for three exams (the rows) taken by two students (the columns):

In [53]:
```
grades.T
```

Out[53]:
```
array([[87, 99],
       [96, 87],
       [70, 90]])
```

Additionally, we can transpose an `array`'s rows and columns, the `T` attribute returns a transposed view of the array.

Assume that the original `grades` `array` presents two students' grades (the rows) across three exams (the columns). Let's transpose the rows and columns to examine the data as the grades for three exams (the rows) taken by two students (the columns):

In [53]:
```
grades.T
```

Out[53]:
```
array([[87, 99],
       [96, 87],
       [70, 90]])
```

Transposing does not modify the original array:

Additionally, we can transpose an `array`'s rows and columns, the `T` attribute returns a transposed view of the array.

Assume that the original `grades` `array` presents two students' grades (the rows) across three exams (the columns). Let's transpose the rows and columns to examine the data as the grades for three exams (the rows) taken by two students (the columns):

In [53]:
```
grades.T
```

Out[53]:
```
array([[87, 99],
       [96, 87],
       [70, 90]])
```

Transposing does not modify the original array:

In [54]:
```
grades
```

Out[54]:
```
array([[87, 96, 70],
       [99, 87, 90]])
```

Finally, we can combine `arrays` by adding more columns or more rows — known as horizontal stacking and vertical stacking. Let's first create another 2-by-3 `array` of grades:

Finally, we can combine `arrays` by adding more columns or more rows — known as horizontal stacking and vertical stacking. Let's first create another 2-by-3 `array` of grades:

In [55]:
```
grades2 = np.array([[94, 77, 90], [100, 81, 82]])
grades2
```

Out[55]:
```
array([[ 94,  77,  90],
       [100,  81,  82]])
```

Finally, we can combine `arrays` by adding more columns or more rows — known as horizontal stacking and vertical stacking. Let's first create another 2-by-3 `array` of grades:

In [55]:
```
grades2 = np.array([[94, 77, 90], [100, 81, 82]])
grades2
```

Out[55]:
```
array([[ 94,  77,  90],
       [100,  81,  82]])
```

Suppose `grades2` represents three more exam grades for the two students in the `grades` array. We can merge `grades` and `grades2` using `NumPy`'s `hstack()` (horizontal stack) function by passing a `tuple` containing the arrays to combine. The extra parentheses are necessary because `hstack()` expects a single argument:

Finally, we can combine `arrays` by adding more columns or more rows — known as horizontal stacking and vertical stacking. Let's first create another 2-by-3 `array` of grades:

In [55]:
```
grades2 = np.array([[94, 77, 90], [100, 81, 82]])
grades2
```

Out[55]:
```
array([[ 94,  77,  90],
       [100,  81,  82]])
```

Suppose `grades2` represents three more exam grades for the two students in the `grades` array. We can merge `grades` and `grades2` using `NumPy`'s `hstack()` (horizontal stack) function by passing a `tuple` containing the arrays to combine. The extra parentheses are necessary because `hstack()` expects a single argument:

In [56]:
```
np.hstack((grades, grades2))
```

Out[56]:
```
array([[ 87,  96,  70,  94,  77,  90],
       [ 99,  87,  90, 100,  81,  82]])
```

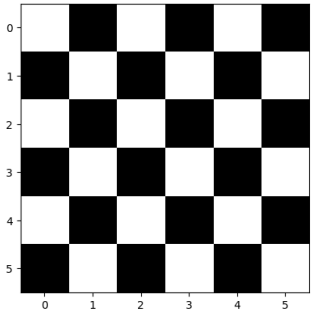
Moving forward, let's suppose that `grades2` represents the grades of two additional students on three exams. In this scenario, we can combine `grades` and `grades2` using `NumPy`'s `vstack()` (vertical stack) function:

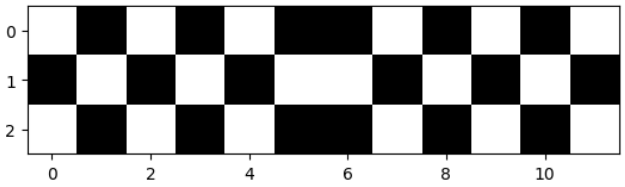
Moving forward, let's suppose that `grades2` represents the grades of two additional students on three exams. In this scenario, we can combine `grades` and `grades2` using `NumPy`'s `vstack()` (vertical stack) function:

In [57]:
```
np.vstack((grades, grades2))
```

Out[57]:
```
array([[ 87,  96,  70],
       [ 99,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

Exercise 1: Suppose we are developing a chess game and the chess game provide two special checkerboards as follows:





We decide to use 1 to represent the white square and 0 to represent the black square. Write a program to create two 2D arrays to represent the two checkerboards as follows:

```
[[1, 0, 1, 0, 1, 0],
 [0, 1, 0, 1, 0, 1],
 [1, 0, 1, 0, 1, 0],
 [0, 1, 0, 1, 0, 1],
 [1, 0, 1, 0, 1, 0],
 [0, 1, 0, 1, 0, 1]]
```

```
[[1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1],
 [0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0],
 [1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1]]
```

Note you should not directly hardcode the above arrays. You should use `Numpy` methods to create the arrays. After you have finished the exercise, you can print out the checkerboard using the following code cell.

In [ ]:

```
# Your answer here
chb1 = np.ones((___,___), dtype=int)
chb1[___, ___] = 0
chb1[___, ___] = 0
chb1
```

In [ ]:
```
# Your answer here
chb1 = np.ones((___,___), dtype=int)
chb1[___, ___] = 0
chb1[___, ___] = 0
chb1
```

In [ ]:
```
# Your answer here
chb2 = np.____((chb1[__,___], chb1[___,___]))
chb2
```

In [ ]:

```
# Plot the checkerboard
package_name = "matplotlib"

try:
    __import__(package_name)
    print(f"{package_name} is already installed.")
except ImportError:
    print(f"{package_name} not found. Installing...")
    %pip install {package_name}

import matplotlib.pyplot as plt
plt.imshow(chb2, cmap='gray')
plt.show()
```

4. `NumPy` calculation methods (Reduction)

An `array` includes several methods that carry out computations based on its contents. **By default, these methods disregard the array's shape and utilize all the elements in the calculations.**

An `array` includes several methods that carry out computations based on its contents. **By default, these methods disregard the array's shape and utilize all the elements in the calculations.**

For instance, when computing the mean of an array, it sums all of its elements irrespective of its shape, and then divides by the total number of elements. **We can also execute these calculations on each dimension.** For example, in a two-dimensional array, we can determine the mean of each row and each column.

In [58]:
```
grades = np.array([[87, 96, 70], [100, 87, 90],
                   [94, 77, 90], [100, 81, 82]])
grades
```

Out[58]:
```
array([[ 87,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

In [58]:
```
grades = np.array([[87, 96, 70], [100, 87, 90],
                   [94, 77, 90], [100, 81, 82]])
grades
```

Out[58]:
```
array([[ 87,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

We can use methods to calculate `sum()`, `min()`, `max()`, `mean()`, `std()` (standard deviation) and `var()` (variance) — each is a functional-style programming reduction:

In [58]:
```
grades = np.array([[87, 96, 70], [100, 87, 90],
                   [94, 77, 90], [100, 81, 82]])
grades
```

Out[58]:
```
array([[ 87,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

We can use methods to calculate `sum()`, `min()`, `max()`, `mean()`, `std()` (standard deviation) and `var()` (variance) — each is a functional-style programming reduction:

In [59]:
```
print(grades.sum())
print(grades.min())
print(grades.max())
print(grades.mean())
print(grades.std())
print(grades.var())
```

```
1054
70
100
87.83333333333333
8.792357792739987
77.30555555555556
```

Calculations by Row or Column

Numerous calculation methods can be applied to specific `array` dimensions, referred to as the `array`'s ***axes***. These methods accept an `axis` keyword argument that designates the dimension to be utilized in the calculation, providing a convenient means to perform computations by row or column in a two-dimensional `array`.

Suppose we want to find the maximum grade for each exam, represented by the columns of `grades`. By specifying `axis=0`, the calculation is performed on all the row values within each column:

Suppose we want to find the maximum grade for each exam, represented by the columns of `grades`. By specifying `axis=0`, the calculation is performed on all the row values within each column:

In [60]:
```
grades, grades.max(axis=0), grades.argmax(axis=0)
```

Out[60]:
```
(array([[ 87,  96,  70],
        [100,  87,  90],
        [ 94,  77,  90],
        [100,  81,  82]]),
 array([100,  96,  90]),
 array([1, 0, 1], dtype=int64))
```

Suppose we want to find the maximum grade for each exam, represented by the columns of `grades`. By specifying `axis=0`, the calculation is performed on all the row values within each column:

In [60]:
```
grades, grades.max(axis=0), grades.argmax(axis=0)
```

Out[60]:
```
(array([[ 87,  96,  70],
        [100,  87,  90],
        [ 94,  77,  90],
        [100,  81,  82]]),
 array([100,  96,  90]),
 array([1, 0, 1], dtype=int64))
```

Here, 100 is the maximum value in the first column and its corresponding index (row) is 1 (if there are duplicate elements, the index of the first element will be reported). 96 and 90 are the maximum values in the second and third columns, respectively.

Suppose we want to find the maximum grade for each exam, represented by the columns of `grades`. By specifying `axis=0`, the calculation is performed on all the row values within each column:

In [60]:
```
grades, grades.max(axis=0), grades.argmax(axis=0)
```

Out[60]:
```
(array([[ 87,  96,  70],
        [100,  87,  90],
        [ 94,  77,  90],
        [100,  81,  82]]),
 array([100,  96,  90]),
 array([1, 0, 1], dtype=int64))
```

Here, 100 is the maximum value in the first column and its corresponding index (row) is 1 (if there are duplicate elements, the index of the first element will be reported). 96 and 90 are the maximum values in the second and third columns, respectively.

In [61]:
```
grades, grades.mean(axis=0)
```

Out[61]:
```
(array([[ 87,  96,  70],
        [100,  87,  90],
        [ 94,  77,  90],
        [100,  81,  82]]),
 array([95.25, 85.25, 83.  ]))
```

Similarly, specifying `axis=1` performs the calculation on all the column values within each individual row. To determine each student's average grade for all exams, we can use:

Similarly, specifying `axis=1` performs the calculation on all the column values within each individual row. To determine each student's average grade for all exams, we can use:

In [62]:
```
grades.mean(axis=1)
```

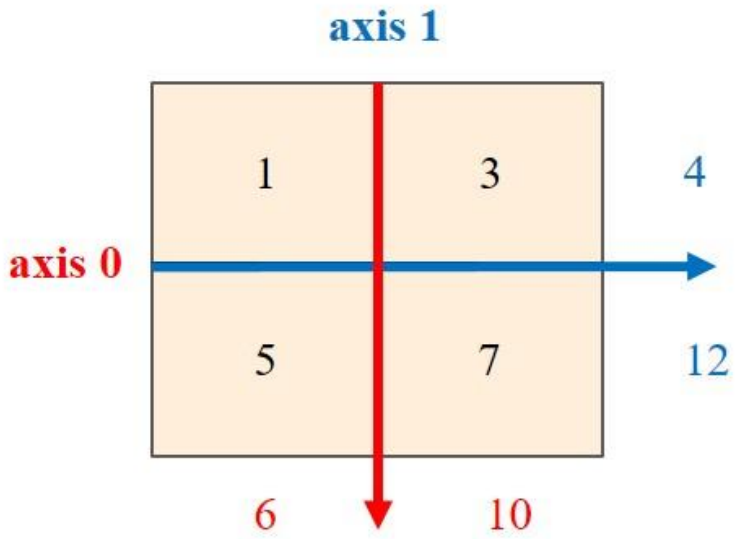
Out[62]:
```
array([84.33333333, 92.33333333, 87.        , 87.66666667])
```

Similarly, specifying `axis=1` performs the calculation on all the column values within each individual row. To determine each student's average grade for all exams, we can use:

In [62]:
```
grades.mean(axis=1)
```

Out[62]:
```
array([84.33333333, 92.33333333, 87.        , 87.66666667])
```

This generates four averages — one for the values in each row. Therefore, 84.33333333 is the average of row 0's grades (87, 96, and 70), and the other averages correspond to the remaining rows. For more methods, refer to https://numpy.org/doc/stable/reference/arrays.ndarray.html.


axis 1
axis 0
1
3
4
5
7
12
6
10

Exercise2: Find the maximum and minimum values of the function $f(x) = x^2$ on the interval $[-3, 5]$ by substituting 1000 evenly spaced numbers between -3 and 5 into the function. What is the corresponding x value for the maximum and minimum values and how do they compare with the actual values?

Hint: You may find `np.linspace()`, `np.max()/np.min()` and `np.argmax()/np.argmin()` useful.

In [ ]:

```
# Your answer here
N = 1000   # Number of points to sample in the interval
x = np._____(___,___, num=N)  # Create 1000 evenly spaced values from -3 to 5
y = ____  # # Compute y = x² for every x in the array

y_max = np.___(y) # Largest value of y (the maximum of the parabola on this i
y_min = np.___(y) # Smallest value of y (the minimum of the parabola on this
x_max = x[np.___(y)] # x-value at which y reaches its maximum
x_min = x[np.___(y)] # x-value at which y reaches its minimum

print("max y=", y_max, "x=", x_max)
print("min y=", y_min, "x=", x_min)
```

In [63]:
```
from jupytercards import display_flashcards
fpath= "flashcards/"
display_flashcards(fpath + 'ch9-1.json')
```

Array-Oriented Programming

Next

>