string in front of the value, followed by a comma. This allows us to distinguish between the values are we printing.

1.3. Scalar-vector multiplication

Scalar-vector multiplication and division. If `a` is a number and `x` is a numpy array (vector), you can express the scalar-vector product either as `a*x` or `x*a`.

In [ ]:
```
import numpy as np
x = np.array([1,2,3])
print(2.2*x)
```

```
[2.2 4.4 6.6]
```

You can carry out scalar-vector division as `x/a`.

In [ ]:
```
import numpy as np
x = np.array([1,2,3])
print(x/2.2)
```

```
[0.45454545 0.90909091 1.36363636]
```

Remark: For Python 2.x, integer division is used when you use the operator `/` on scalars. For example, `5/2` gives you 2. You can avoid this problem by adding decimals to the integer, *i.e.*, `5.0/2`. This gives you 2.5.

Scalar-vector addition. In Python, you can add a scalar `a` and a numpy array (vector) `x` using `x+a`. This means that the scalar is added to each element of the vector. This is, however, NOT a standard mathematical notation. In mathematical notations, we should denote this as, e.g. $x + a\mathbf{1}$, where x is an n-vector and a is a scalar.

In [ ]:
```
import numpy as np
x = np.array([1,2,3,4])
print(x + 2)
```

```
[3 4 5 6]
```

Elementwise operations. In Python we can perform elementwise operations on numpy arrays. For numpy arrays of the same length `x` and `y`, the expressions `x * y`, `x / y` and `x ** y` give the resulting vectors of the same length as x and y and ith element $x_i y_i$,

x_i/y_i, and $x_i^{y_i}$ respectively. (In Python, scalar a to the power of b is expressed as `a**b`)

As an example of elementwise division, let's find the 3-vector of asset returns r from the (numpy arrays of) initial and final prices of assets (see page 22 in VMLS).

In [ ]:
```
import numpy as np
p_initial = np.array([22.15, 89.32, 56.77])
p_final = np.array([23.05, 87.32, 53.13])
r = (p_final - p_initial) / p_initial
r
```

Out[ ]:
```
array([ 0.04063205, -0.0223914 , -0.06411837])
```

Linear combination. You can form a linear combination is Python using scalar-vector multiplication and addition.

In [ ]:
```
import numpy as np
a = np.array([1,2])
b = np.array([3,4])
alpha = -0.5
beta = 1.5
c = alpha*a + beta*b
print(c)
```

```
[4. 5.]
```

To illustrate some additional Python syntax, we create a function that takes a list of coefficients and a list off vectors as its argument (input), and returns the linear combination (output).

In [ ]:
```
def lincomb(coef, vectors):
    n = len(vectors[0])
    comb = np.zeros(n)
    for i in range(len(vectors)):
        comb = comb + coef[i] * vectors[i]
    return comb

lincomb([alpha, beta], [a,b])
```

Out[ ]:
```
array([4., 5.])
```

A more compact definition of the function is as follows.

In [ ]:
```
def lincomb(coef, vectors):
    return sum(coef[i]*vectors[i] for i in range(len(vectors)))
```

Checking properties. Let's check the distributive property

$$\beta(a+b) = \beta a + \beta b$$

which holds for any two n-vector a and b, and any scalar β. We'll do this for $n = 3$, and randomly generated a, b, and β. (This computation does not show that the property always holds; it only show that it holds for the specific vectors chosen. But it's good to be skeptical and check identities with random arguments.) We use the `lincomb` function we just defined.

In [ ]:
```
import numpy as np
a = np.random.random(3)
b = np.random.random(3)
beta = np.random.random()
lhs = beta*(a+b)
rhs = beta*a + beta*b
print('a :', a)
print('b :', b)
print('beta :', beta)
print('LHS :', lhs)
print('RHS :', rhs)
```

```
a : [0.81037789 0.423708   0.76037206]
b : [0.45712264 0.73141297 0.46341656]
beta : 0.5799757967698047
LHS : [0.73511963 0.6699422  0.70976778]
RHS : [0.73511963 0.6699422  0.70976778]
```

Although the two vectors `lhs` and `rhs` are displayed as the same, they might not be exactly the same, due to very small round-off errors in floating point computations. When we check an identity using random numbers, we can expect that the left-hand and right-hand sides of the identity are not exactly the same, but very close to each other.

1.4. Inner product

Inner product. The inner product of n-vector x and y is denoted as $x^T y$. In Python the inner product of `x` and `y` can be found using `np.inner(x,y)`

In [ ]:
```
import numpy as np
x = np.array([-1,2,2])
y = np.array([1,0,-3])
print(np.inner(x,y))
```

```
-7
```

Alternatively, you can use the `@` operator to perform inner product on numpy arrays.

In [ ]:
```
import numpy as np
x = np.array([-1,2,2])
y = np.array([1,0,-3])
x @ y
```

Out[ ]:
```
-7
```

Net present value. As an example, the following code snippet finds the net present value (NPV) of a cash flow vector `c`, with per-period interest rate `r`.

In [ ]:
```
import numpy as np
c = np.array([0.1,0.1,0.1,1.1]) #cash flow vector
n = len(c)
r = 0.05 #5% per-period interest rate
d = np.array([(1+r)**-i for i in range(n)])
NPV = c @ d
print(NPV)
```

```
1.236162401468524
```

In the fifth line, to get the vector `d` we raise the scalar `1+r` element-wise to the powers given in the range `range(n)` which expands to `0, 1,2, ..., n-1`, using list comprehension.

Total school-age population. Suppose that the 100-vector x gives the age distribution of some population, with x_i the number of people of age $i - 1$, for $i = 1, \ldots, 100$. The