total number of people with age between 5 and 18 (inclusive) is given by

$$x_6 + x_7 + \cdots + x_{18} + x_{19}$$

We can express this as $s^T x$ where s is the vector with entries one for $i = 6, \ldots, 19$ and zero otherwise. In Python, this is expressed as

In [ ]:
```
s = np.concatenate([np.zeros(5), np.ones(14), np.zeros(81)])
school_age_pop = s @ x
```

Several other expressions can be used to evaluate this quantity, for example, the expression `sum(x[5 : 19 ])`, using the Python function `sum`, which gives the sum of entries of vector.

1.5. Complexity of vector computations

Floating point operations. For any two numbers a and b, we have $(a+b)(a-b) = a^2-b^2$. When a computer calculates the left-hand and right-hand side, for specific numbers a and b, they need not be exactly the same, due to very small floating point round-off errors. But they should be very nearly the same. Let's see an example of this.

In [ ]:
```
import numpy as np
a = np.random.random()
b = np.random.random()
lhs = (a+b) * (a-b)
rhs = a**2 - b**2
print(lhs - rhs)
```

```
4.336808689942018e-19
```

Here we see that the left-hand and right-hand sides are not exactly equal, but very very close.

Complexity. You can time a Python command using the `time` package. The timer is not very accurate for very small times, say, measured in microseconds (10^{-6} seconds). You should run the command more than once; it can be a lot faster on the second or subsequent runs.

In [ ]:
```
import numpy as np
import time
a = np.random.random(10**5)
```

```
b = np.random.random(10**5)
start = time.time()
a @ b
end = time.time()
print(end - start)
```

```
0.0006489753723144531
```

In [ ]:
```
start = time.time()
a @ b
end = time.time()
print(end - start)
```

```
0.0001862049102783203
```

The first inner product, of vectors of length 10^5, takes around 0.0006 seconds. This can be predicted by the complexity of the inner product, which is $2n-1$ flops. The computer on which the computations were done is capable of around 1 Gflop/s. These timings, and the estimate of computer speed, are very crude.

Sparse vectors. Functions for creating and manipulating sparse vectors are contained in the Python package `scipy.sparse`, so you need to import this package before you can use them. There are 7 classes of sparse structures in this package, each class is designed to be efficient with a specific kind of structure or operation. For more details, please refer to the `sparse` documentation.

Sparse vectors are stored as sparse matrices, i.e., only the nonzero elements are stored (we introduce matrices in chapter 6 of VMLS). In Python you can create a sparse vector from lists of the indices and values using the `sparse.coo_matrix` function. The `scipy.sparse.coo_matrix()` function create a sparse matrix from three arrays that specify the row indexes, column indexes and values of the nonzero elements. Since we are creating a row vector, it can be considered as a matrix with only 1 column and 100000 columns.

In [ ]:
```
from scipy import sparse
I = np.array([4,7,8,9])
J = np.array([0,0,0,0])
V = np.array([100,200,300,400])
A = sparse.coo_matrix((V,(I,J)),shape = (10000,1))
A
```

Out[ ]:
```
<10000x1 sparse matrix of type '<class 'numpy.int64'>'
  with 4 stored elements in COOrdinate format>
```

We can perform mathematical operations on the sparse matrix `A`, then we can view the result using `A.todense()` method.

From this point onwards, in our code syntax, we assume you have imported `numpy` package using the command `import numpy as np`.