1. Vectors

1.1. Vectors

The simplest way to represent vectors in Python is using a *list* structure. A vector is constructed by giving the list of elements surrounded by square brackets, with the elements separated by commas. The assignment operator `=` is used to give a name to the list. The `len()` function returns the size (dimension).

In [ ]:
```
x = [-1.1, 0.0, 3.6, -7.2]
len(x)
```

Out[ ]:
```
4
```

Some common mistakes. Don't forget the commas between entries, and be sure to use square brackets. Otherwise, you'll get things that may or may not makes sense in Python, but are not vectors.

In [ ]:
```
x = [-1.1  0.0  3.6  -7.2]
```

Out[ ]:
```
  File "<ipython-input-1-18f108d7fe41>", line 1
    x = [-1.1  0.0  3.6  -7.2]
                 ^
SyntaxError: invalid syntax
```

Here Python returns a `SyntaxError` suggesting the expression that you entered does not make sense in Python. (Depending on the kind of error that occurred, Python may return other types of error messages. Some of the error messages are quite self-explanatory, some are not. See Appendix A for other types of error messages from Python and possible solutions.)

In [ ]:
```
y = (1,2)
```

Here `y` is a tuple consisting of two scalars. It is neither a list nor a vector.

Another common way to represent vectors in Python is to use a numpy array. To do so, we must first import the numpy package.

In [ ]:
```
import numpy as np
x = np.array([-1.1, 0.0, 3.6, -7.2])
len(x)
```

Out[ ]:
```
4
```

We can initialize numpy arrays from Python list. Here, to call the package we put `np.` in front of the array structure. *Note*: we have parentheses outside the square brackets. A major advantage of numpy arrays is their ability to perform linear algebra operations which make intuitive sense when we are working with vectors.

Indexing. A specific element x_i is extracted with the expression `x[i]` where `i` is the index (which runs from 0 to $n-1$, for an n-vector).

In [ ]:
```
import numpy as np
x = np.array([-1.1, 0.0, 3.6, -7.2])
x[2]
```

Out[ ]:
```
3.6
```

If we used array indexing on the left-hand side of an assignment statement, then the value of the corresponding element changes.

In [ ]:
```
x[2] = 20.0
print(x)
```

```
[-1.1  0.  20.  -7.2]
```

-1 is a special index in Python. It is the index of the last element in an array.

In [ ]:
```
x[-1]
```

Out[ ]:
```
-7.2
```

Assignment versus copying. The behavior of an assignment statement `y = x` where `x` is an array may be surprising for those who use other languages like Matlab or Octave. The assignment expression gives a new name `y` to the *same* array that is already referenced by `x` instead of creating a copy of `x`.

In [ ]:
```
import numpy as np
x = np.array([-1.1, 0.0, 3.6, -7.2])
y = x
x[2] = 20.0
print(y)
```

```
[-1.1  0.  20.  -7.2]
```

To create a new copy of array `x`, the method `copy` should be used.

In [ ]:
```
import numpy as np
x = np.array([-1.1, 0.0, 3.6, -7.2])
y = x.copy()
x[2] = 20.0
print(y)
```

```
[-1.1  0.   3.6 -7.2]
```

Vector equality. Equality of vectors is checked using the relational operator `==` (double equal signs). The Python expression evaluates to `True` if the expression on the left and right-hand side of the relational operator is equal, and to `False` otherwise.

In [ ]:
```
import numpy as np
x = np.array([-1.1, 0.0, 3.6, -7.2])
y = x.copy()
x == y
```

Out[ ]:
```
array([ True,  True,  True,  True])
```

In [ ]:
```
import numpy as np
x = np.array([-1.1, 0.0, 3.6, -7.2])
y = x.copy()
y[3] = 9.0
x == y
```

Out[ ]:
```
array([ True,  True,  True, False])
```

Here four evaluations (`True` or `False`) are shown because applying relational operator on numpy array performs element-wise comparison. However, if we apply the relational operator on list structures, the Python expression evaluates to `True` only when both sides have the same length and identical entries.

```
In [ ]: x = [-1.1, 0.0, 3.6, -7.2]
        y = x.copy()
        x == y
```

```
Out[ ]: True
```

```
In [ ]: x = [-1.1, 0.0, 3.6, -7.2]
        y = x.copy()
        y[3] = 9.0
        x == y
```

```
Out[ ]: False
```

Scalars versus 1-vectors. In the mathematical notations of VMLS, 1-vector is considered as a scalar. However, in Python, 1-vectors are not the same as scalars. For list structure, Python distinguishes 1-vector (list with only one element) `[2.4]` and the number `2.4`.

```
In [ ]: x = 2.4
        y = [2.4]
        x == y
```

```
Out[ ]: False
```

```
In [ ]: y[0] == 2.4
```

```
Out[ ]: True
```

In the last line, Python checks whether the first element of `y` is equal to the number `2.4`. For numpy arrays, Python compares the elements inside `np.array([2.4])` with the scalar. In our example, Python compares the first (and only) element of `y` to the number `2.4`. Hence, numpy 1-vectors behave like scalars.

```
In [ ]: import numpy as np
        x = 2.4
        y = np.array([2.4])
        x == y
```

```
Out[ ]: array([ True])
```

Block and stacked vectors. In Python, we can construct a block vector using the numpy function `concatenate()`. Remember you need an extra set of parentheses over the vectors that you want to concatenate. The use of numpy array or list structure does not create a huge difference here.

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

```
[ 1 -2  1  1  0]
```

Some common mistakes. There are few Python operations that appear to be able to construct a block or stacked vector but do not. For example, `z = (x,y)` creates a tuple of two vectors; `z = [x,y]` creates an array of the two vectors. Both of these are valid Python expression but neither of them is the stacked vector.

Subvectors and slicing. In VMLS, the mathematical notation $r : s$ denotes the index range $r, r+1, \ldots, s$ and $x_{r:s}$ denotes the slice of the vector x from index r to s. In Python, you can extract a slice of a vector using an index range as the argument. Remember, Python indices start from 0 to $n-1$. For the expressing `x[a:b]` for array `x`, the slicing selects the element from index `a` to index `b-1`. In the code below, `x[1:4]` select element from index 1 to 3, which means the second to the fourth elements are selected.

In [ ]:
```
import numpy as np
x = np.array([1,8,3,2,1,9,7])
y = x[1:4]
print(y)
```

```
[8 3 2]
```

You can also use index ranges to assign a slice of a vector. In the code below, we reassigned index 3 to 5 in array `x`.

In [ ]:
```
x[3:6] = [100,200,300]
print(x)
```

```
[  1   8   3 100 200 300   7]
```

As a result, you can see the 4th, 5th and 6th elements in the array are reassigned.

You can also use slicing to select all elements in the array starting from a certain index

In [ ]:
```
import numpy as np
x = np.array([1,8,3,2,1,9,7])
x[2:]
```

Out[ ]:
```
array([3, 2, 1, 9, 7])
```

In [ ]:
```
x[:-1]
```

Out[ ]:
```
array([1, 8, 3, 2, 1, 9])
```

Here the first expression selects all elements in `x` starting from index 2 (the third element). The second expression selects all elements in `x` except the last element.

Python indexing into arrays Python slicing and subvectoring is much more general than the mathematical notation we use in VMLS. For example, one can use a number range with a third argument, that gives the increment between successive indexes. For example, the index range `1:5:2` is the list of numbers $8, 2$. The expression `x[1:4:2]` extracts the second (index 1) to the fifth (index 4) element with an increment of 2. Therefore, the second and fourth entries of `x` are extracted. You can also use an index range that runs backward. The Python expression `x[::-1]` gives the reversed vector, *i.e.*, the vector with the same coefficients but in opposite order.

Vector of first differences. Here we use slicing to create an $(n-1)$-vector d which is defined as the first difference vector $d_i = x_{i+1} - x_i$, for $i = 1, \ldots, n-1$, where x is an n-vector.

In [ ]:
```
import numpy as np
x = np.array([1,8,3,2,1,9,7])
d = x[1:] - x[:-1]
print(d)
```

```
[ 7 -5 -1 -1  8 -2]
```

Lists of vectors. An ordered list of n-vectors might be denoted in VMLS as $a_1, \ldots, a_k$ or $a^{(1)}, \ldots, a^{(k)}$ or just as a, b, c. There are several ways to represent lists of vectors in Python. If we give the elements of the list, separated by commas, and surrounded by

square brackets, we form a one-dimensional arrays of vectors or a list of lists. If instead, we use parentheses as delimiters, we obtain a *tuple*.

In [ ]:
```
x = [1,0]
y = [1,-1]
z = [0,1]
list_of_vectors = [x,y,z]
list_of_vectors[1] #Second element of list
```

Out[ ]:
```
[1, -1]
```

In [ ]:
```
list_of_vectors[1][0] #First entry in second element of list
```

Out[ ]:
```
1
```

In [ ]:
```
tuple_of_vectors = (x,y,z)
tuple_of_vectors[1] #Second element of list
```

Out[ ]:
```
[1, -1]
```

In [ ]:
```
tuple_of_vectors[1][0] #First entry in second element of list
```

Out[ ]:
```
1
```

Note that the difference between `[x,y,z]` (an array of arrays or a list of lists) and a stacked vector of x, y, and z, obtained by concatenation.

Zero vectors. We can create a zero vector of size n using `np.zeros(n)`. The expression `np.zeros(len(x))` creates a zero vector with the same dimension as vector `x`.

In [ ]:
```
import numpy as np
np.zeros(3)
```

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

Unit vectors. We can create e_i, the ith unit vector of length n using index.

In [ ]:
```
import numpy as np
i = 2
n = 4
x = np.zeros(n)
x[i] = 1
print(x)
```

```
[0. 0. 1. 0.]
```

Ones vector. We can create a ones vector of size n using `np.ones(n)`. The expression `np.ones(len(x))` creates a ones vector with the same dimension as vector `x`.

```
In [ ]: import numpy as np
        np.ones(3)
```

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

Random vectors. Sometimes it is useful to generate random vectors to check our algorithm or test an identity. In Python, we generate a random vector of size n using `np.random.random(n)`.

```
In [ ]: np.random.random(2)
```

```
Out[ ]: array([0.79339885, 0.60803751])
```

Plotting. There are several external packages for creating plots in Python. One such package is `matplotlib`, which comes together in the Anaconda distribution. Otherwise, you can also add (install) it manually; see page vii. In particular, we use the module `pyplot` in `matplotlib`. Assuming the `matplotlib` package had been installed, you import it into Python using the command `import matplotlib.pyplot as plt`. (Now `plt` acts as an alias for `matplotlib.pyplot`.) After that, you can access the Python commands that create or manipulate plots.

For example, we can plot the temperature time series in Figure 1.3 in VMLS using the code below; the last line saves the plot in the file `temperature.pdf`. The result is shown in Figure 1.1.

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
        temps = [ 71, 71, 68, 69, 68, 69, 68, 74, 77, 82, 85, 86, 88, 86,
        ↪  85, 86, 84, 79, 77, 75, 73, 71, 70, 70, 69, 69, 69, 69, 67,
        ↪  68, 68, 73, 76, 77, 82, 84, 84, 81, 80, 78, 79, 78, 73, 72,
        ↪  70, 70, 68, 67 ]
        plt.plot(temps, '-bo')
        plt.savefig('temperature.pdf', format = 'pdf')
```

```
plt.show()
```

The syntax `-bo` indicates plotting with line (`-`) with circle marker (`o`) in blue (`b`). To show the plot in the interactive session or the notebook, we need to set the interactive output on with `plt.ion()` and then use the command `plt.show()`.

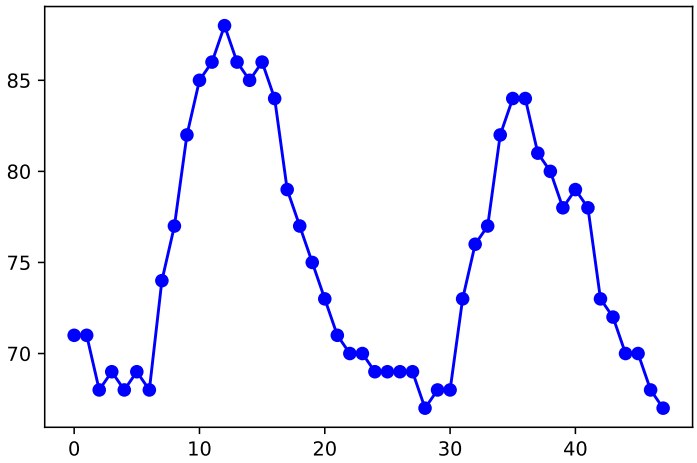

Figure 1.1.: Hourly temperature in downtown Los Angeles on August 5 and 6, 2015 (starting at 12:47AM, ending at 11:47PM).

1.2. Vector addition

Vector addition and subtraction. If `x` and `y` are numpy arrays of the same size, `x+y` and `x-y` give their sum and difference, respectively.

In [ ]:
```
import numpy as np
x = np.array([1,2,3])
y = np.array([100,200,300])
print('Sum of arrays:', x+y)
print('Difference of arrays:', x-y)
```

```
Sum of arrays: [101 202 303]
Difference of arrays: [ -99 -198 -297]
```

Sometimes when we would like to print more than one value, we may add a piece of