Chapter 1.1

In [3]:
```
# page 2

x = [-1.1, 0.0, 3.6, -7.2] # vectors can be numbers, positive, negative, decimal
len(x) # built in counter, basically a summation notation
```

Out[3]:
```
4
```

In [4]:
```
y = [1, 0.5, 0.1, 2] # example vector 2

x*y # you can't multiply 2 vectors, and that's reflected in python as well!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_15492\3597416476.py in <module>
      1 y = [1, 0.5, 0.1, 2] # example vector 2
      2
----> 3 x*y # you can't multiply 2 vectors, and that's reflected in python as well!

TypeError: can't multiply sequence by non-int of type 'list'
```

In [5]:
```
import numpy as np

# testing for vector equality but trying different examples

x = np.array([-1.0, 0.0, 3.6, -7.2])
y = np.array([-1, 0, 3.6, -7.20])

x == y

# so comparing vectors is floating point agnostic?
```

Out[5]:
```
array([ True,  True,  True,  True])
```

In [6]:
```
# experimenting with nested vectors in python

a = np.array([1, 2, 3, [3, 4]])
b = np.array([1, 2, 3, 5])

a*b
```

```
C:\Users\raiu\AppData\Local\Temp\ipykernel_15492\137072018.py:3: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.
  a = np.array([1, 2, 3, [3, 4]])
```

Out[6]:
```
array([1, 4, 9, list([3, 4, 3, 4, 3, 4, 3, 4, 3, 4])], dtype=object)
```

In [7]:
```
# cannot add different size vectors in python

c = np.array([1, 2, 3])
d = np.array([1, 2, 3, 5])

c + d
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_15492\494062773.py in <module>
      4 d = np.array([1, 2, 3, 5])
      5
----> 6 c + d

ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

In [8]:
```
# but can just add them together if they're the same size, follows the linear algebra rules
c = np.array([1, 2, 3])
d = np.array([9, 6, 9])

c + d
```

Out[8]:
```
array([10,  8, 12])
```

In [9]:
```
# page 6

# this is different than concatenate
e = np.concatenate((c,d))
print(e)
```

```
[1 2 3 9 6 9]
```

In [10]:
```
# page 6

# vector slicing works to access certain elements in vector
# how does this work a little differently than normal python array?
# numpy array is wrapped in parantheses before the brackets

w = np.array([1,8,3,2,1,9,7])
u = w[1:4]
print(u)
```

```
[8 3 2]
```

In [11]:
```
w[3:6] = [100, 200, 300]
print(w)

# why does the output not have commas? is that something with numpy?
# we took the original array and replaced i = 3:6 with 3 new elements
# notice the extra space between the digits added and original digits?
```

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

In [12]:
```
w[3:7] = ['a', 'b', 'c', 'd']
print(w)

# cannot be letters since numpy array is numbers?
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_15492\3212277145.py in <module>
----> 1 w[3:7] = ['a', 'b', 'c', 'd']
      2 print(w)
      3
      4 # cannot be letters since numpy array is numbers?

ValueError: invalid literal for int() with base 10: 'a'
```

In [13]:
```
w[3:7] = [99, 91, 98, 0.01]
print(w)

# must have enough elements to satisfy from 3 -> 7
# why did the last number print as "0"?
```

```
[ 1  8  3 99 91 98  0]
```

Chapter 1.2 Vector Addition

In [14]:
```
# page 10

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]
```

In [15]:
```
x = np.array([1.001,2.99,3.999])
y = np.array([1,2,3])
print('Sum of arrays:', x+y)
print('Difference of arrays:', x-y)

# earlier, it rounded my digits, but seems like it's accurate now?
```

```
Sum of arrays: [2.001 4.99  6.999]
Difference of arrays: [0.001 0.99  0.999]
```

Chapter 1.3 Scalar Vector Multiplication

In [16]:
```
# page 11
x = np.array([1,2,3])
print(x/2.2)
```

```
[0.45454545 0.90909091 1.36363636]
```

In [17]:
```
x = np.array([1,2,3])
print(x/0.012)

# odd formatting, i assume from np
```

```
[ 83.33333333 166.66666667 250.        ]
```

In [18]:
```
x = np.array([1,2,3])
print(x/0.01222222222222222222222222222222)

# testing for floating point accuracy
```

```
[ 81.81818182 163.63636364 245.45454545]
```

In [19]:
```
x = np.array([1,2,3])
print(x/0.012222222222222222222222222244444444444444444444444444444)

# why does it seem like it's not that accurate?
```

```
[ 81.81818182 163.63636364 245.45454545]
```

In [20]:
```
# page 12
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.]
```

In [21]:
```
o = np.array([222,2222])
p = np.array([2.000000001,0.0000001])
triple = -0.005
penta = 1.0000505
e = triple*o + penta*p
print(c)
```

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

In [22]:
```
# take the steps above and turn it into a function
def lincomb(coef, vectors): # input is the coefficient + vectors
    n = len(vectors[0]) # n = length of first vector
    comb = np.zeros(n) # combination is zeros? what is zeros? must be something from numpy library
    for i in range(len(vectors)): # for loop to intreate the vector length
        comb = comb + coef[i] * vectors[i] # then update teh combination
    return comb

lincomb([alpha, beta], [a,b]) # the coefficient is passed as list as 1 argument
# the vectors is passed as a list for another argument
```

Out[22]:
```
array([4., 5.])
```

In [23]:
```
# experimenting with different number to see how this changes the linear comb result

r = np.array([1,2])
t = np.array([3,4])
alpha = -1.111111111
beta = 2.666666

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

# comb = comb + coef[i] * vectors[i] <-- this is the bulk of the logic
```

Out[23]:
```
array([6.88888689, 8.44444178])
```

Chapter 1.4 Inner Product

In [24]:
```
# page 14

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

# inner product is already a method in numpy! don't have to rebuild from scratch every time
```

```
-7
```

In [25]:
```
# page 14

x = np.array([-1,2,2])
y = np.array([1,0,-3])
x @ y

# "@" is used to perform inner product on numpy arrays, but they have to be in np.array format
```

Out[25]:
```
-7
```

In [26]:
```
(x/2)@ y

# cannot do any operation before inner product
```

Out[26]:
```
-3.5
```

Chapter 1.5 Complexity of Vector Computation

In [27]:
```
# page 15

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)

# the left hand side and the right hand side are not exactly perfectly equal
# and this is due to the # of floating point operations (and how it rounds off
# errors)

# this is super relevant to what im doing as my full-time job because we're
# building hardware to process FP8 on ASIC dedicated to transformers
```

```
0.0
```

In [28]:
```
# can compare complexity

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)

# this is supposed to be "0.0006489753723144531"
```

```
0.0079446912414550078
```

In [29]:
```
start = time.time()
a @ b
end = time.time()
print(end - start)

# this is supposed to be "0.00018620491027832031" according to the book, but it rounded down?
```

```
0.00049686431884765625
```

The first inner product, of vectors of length 105 , 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

In [ ]:
```

```