Vectors and Arrays

- A vector is a set of values whose meaning is context sensitive.
- (1, 3) could represent a point on a plane by specifying the x and y coordinates.
- (-2, 7, 10) could represent a point in three dimensional space by specifying the x, y and z coordinates.
- $(x_1, x_2, x_3, \ldots, x_n)$ could be the solution to n equations with n unknowns.
- A vector could represent a vehicle's direction, velocity, acceleration and position.

- Vectors in space have a length and direction.
- The length $||v||$ of a vector $v = (v_0, v_1, \ldots, v_{n-1})$ is

- Let $u = (u_0, u_1, \ldots u_{n-1})$ and $v = (v_0, v_1, \ldots, v_{n-1})$ be two vectors each with n elements.
- $u+v = (u_0+v_0, u_1+v_1, \ldots, u_{n-1}+v_{n-1})$
- $u-v = (u_0-v_0, u_1-v_1, \ldots, u_{n-1}-v_{n-1})$
- $u \cdot v = (u_0v_0 + u_1v_1 + \ldots + u_{n-1}v_{n-1})$ which is known as the inner, dot or scalar product
- A scalar is a single value such as 42.

- If $v = (v_0, v_1, \ldots, v_{n-1})$ then $f(v) = (f(v_0), f(v_1), \ldots, f(v_{n-1}))$
- e.g. $\sin(v) = (\sin(v_0), \sin(v_1), \ldots, \sin(v_{n-1}))$
- e.g.
- e.g. $v + 4 = (v_0+4, v_1+4, \ldots, v_{n-1}+4)$, we add the value 4 to each element of the vector
- Suppose we create tuples to represent vectors.

```
In [1]: a = (1, 2, 3)
In [2]: b = (4, 5, 6)
In [3]: print(a + b)
(1, 2, 3, 4, 5, 6)
```

- We get something like union, but NOT addition!

- Suppose we use lists to implement vectors.

```
In [1]: a = [1, 2, 3]
In [2]: b = [4, 5, 6]
In [3]: print(a + b)
[1, 2, 3, 4, 5, 6]
```

- We get something like union, but NOT addition!

```
In [4]: a + 4
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    a + 4
TypeError: can only concatenate list (not
"int") to list
```

- We can not add a scalar to a list!

- We cannot use lists or tuples to implement vectors directly.
- Let's try using **arrays** to implement vectors.
- To use arrays in Python you must import the **numpy** module, usually as follows:

```
import numpy as np
```

- To create two vectors enter:

```
In [1]: a = np.array([1, 2, 3])
In [2]: print(type(a))
<class 'numpy.ndarray'>
In [3]: a
array([1, 2, 3])
In [4]: b = np.array([4, 5, 6])
In [5]: b
array([4, 5, 6])
```

```
In [1]: a
array([1, 2, 3])
In [2]: b
array([4, 5, 6])
In [3]: a + b
array([5, 7, 9])
In [4]: a - b
array([-3, -3, -3])
In [5]: a * b
array([ 4, 10, 18])
In [6]: a + 4
array([5, 6, 7])
In [6]: b**2
array([16, 25, 36])
```

•Rather than using ***np.array*** and ***range*** to create an array use ***np.arange***.

```
In [1]: np.array(range(11))
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
In [2]: np.arange(11)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
In [3]: np.arange(2,11)
array([ 2,  3,  4,  5,  6,  7,  8,  9, 10])
```

•With ***np.arange*** you can use floats.

```
In [4]: np.arange(0, .6, .1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5])
In [5]: np.array([e * .1 for e in range(6)])
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5])
```

- When using arrays use the functions from numpy.

```
In [1]: from math import pi
In [2]: import numpy as np
In [3]: a = np.array([pi/n for n in (2, 4, 6)])
In [4]: a
array([ 1.57079633,  0.78539816,  0.52359878])
In [5]: np.sin(a)
array([ 1.        ,  0.70710678,  0.5       ])
In [6]: a = np.array([1.5, 12, 'hi'])
In [7]: a
array(['1.5', '12', 'hi'],
      dtype='<U32')
```

- Notice that the float 1.5 and the int 12 were converted to strings.
- **All the elements of an array MUST be of the same type!**

•To create array elements that are evenly spaced use the ***linspace*** function which is of the form:

•linspace(first element, last element, number of elements)

```
In [1]: firstElement = .1
In [2]: lastElement = .5
In [3]: numElements = 9
In [4]: np.linspace(firstElement,lastElement,numElements)
array([ 0.1 ,  0.15,  0.2 ,  0.25,  0.3 ,  0.35,  0.4 ,
0.45,  0.5 ])
```

Notice that the default type of the array elements is ***float***.

•numElements =[(lastElement – firstElement) / stepSize] + 1

•stepSize = (lastElement – firstElement) / (numElements - 1)

•e.g. (0.5 - 0.1) / (9 - 1) = 0.4 / 8 = 0.05

•Array elements are indexed using [], the same as for ***lists*** and ***tuples***.

```
In [1]: ar = np.array([1, 2, 3, 4, 5])
In [2]: ar[0], ar[1], ar[2], ar[3], ar[4]
(1, 2, 3, 4, 5)
```

•Array elements are variables, their values may be changed and used in expressions.

```
In [3]: ar[0] = (ar[1] + ar[2]) * ar[3]
In [4]: print(ar[0])
20
```

- The first element of an array is at position 0 and the last element is at position one less than the length of the array

```
In [1]: ar
array([1, 2, 3, 4, 5])
In [2]: len(ar)
5
In [3]: print(ar)
[1 2 3 4 5]
>>> ar[len(ar)-1]
5
In [4]: for i, a in enumerate(ar):
            print('ar[%d]=%g' % (i, a))
ar[0]=1
ar[1]=2
ar[2]=3
ar[3]=4
ar[4]=5
```

•Slices work for arrays but unlike for lists and tuples the slice is **not** a copy of the selected array elements.

```
In [1]: import numpy as np
In [2]: ar = np.array([1, 2, 3, 4])
In [3]: ar
array([1, 2, 3, 4])
In [4]: bb = ar[1:-1]
In [5]: bb
array([2, 3])
In [6]: bb[0] = 5
In [7]: bb[1] = 9
In [8]: bb
array([5, 9])
In [9]: ar
array([1, 5, 9, 4])
```

•Changing the value of an element in **bb** changes the value of the corresponding element in **ar**.

```
In [1]: ar = np.linspace(0, 3, 31)
In [2]: ar
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,
0.7,  0.8,  0.9,  1. , 1.1,  1.2,  1.3,  1.4,  1.5,
1.6,  1.7,  1.8,  1.9,  2. ,  2.1, 2.2,  2.3,  2.4,
2.5,  2.6,  2.7,  2.8,  2.9,  3. ])
In [3]: ar[:]
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,
0.7,  0.8,  0.9,  1. , 1.1,  1.2,  1.3,  1.4,  1.5,
1.6,  1.7,  1.8,  1.9,  2. ,  2.1, 2.2,  2.3,  2.4,
2.5,  2.6,  2.7,  2.8,  2.9,  3. ])
In [4]: ar[:2]
array([ 0. ,  0.1])
In [5]: ar[::5]
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ])
In [6]: ar[2:-2:6]
array([ 0.2,  0.8,  1.4,  2. ,  2.6])
```

•Create an array from an existing array using list comprehension.

```
In [1]: from math import exp
In [2]: x = np.linspace(0, 5, 6)
In [3]: x
array([ 0.,  1.,  2.,  3.,  4.,  5.])
In [4]: y = np.array([exp(xVal) for xVal in x])
In [5]: y
array([1., 2.71828183, 7.3890561, 20.08553692,
       54.59815003, 148.4131591 ])
```

•Create an array from another array using ***vectorization***.

```
In [1]: from numpy import exp
In [2]: x
array([ 0.,  1.,  2.,  3.,  4.,  5.])
In [3]: y = exp(x)
In [4]: y
array([1., 2.71828183, 7.3890561, 20.08553692,
       54.59815003, 148.4131591])
In [5]: type(y)
<class 'numpy.ndarray'>
```

•The functions in ***numpy*** implement ***vector arithmetic*** for arrays.

•It is not necessary to use loops to perform vector operations on arrays when the functions from ***numpy*** are used.

•Vector operations are much faster than loops.

- Replacing a loop, as shown above, with a vector / array expression is called ***vectorization***.
- It is important to understand that the functions imported from *numpy* are different than those imported from *math*.
- The functions from *math* do not allow arrays as arguments.
- The functions from *numpy* accept both arrays and real numbers as arguments.
- If a program is not using arrays use the functions from *math*.
- If a program is using arrays use the functions from *numpy* and ***vectorization*** to improve the efficiency of the program.

- Use ***vectorization*** to compute degrees Fahrenheit from degrees Celsius.

```
In [1]: celsius = np.linspace(-20, 20, 9)
In [2]: celsius
array([-20., -15., -10.,  -5.,   0.,   5.,
10.,  15.,  20.])
In [3]: fahrenheit = 9/5 * celsius + 32
In [4]: fahrenheit
array([ -4.,   5.,  14.,  23.,  32.,  41.,
50.,  59.,  68.])
```

- Determine the current position **x** of an object at time **t** given its initial position $\mathbf{x_0}$, initial velocity $\mathbf{v_0}$, its rate of acceleration **a**.
- $\mathbf{x_0}$ and $\mathbf{v_0}$ are the position and velocity of the object at time **t**=0.
- Use the following equation to compute the position of the object at time **t**.

- Create a table of positions from time t_0 to t_n.
- This can be done using arrays and ***vectorization***.

  (positionVector.py)

```
from time import ctime
import numpy as np

def positions(x0, v0, a, stop, intervals):
    tValues = np.linspace(0, stop, intervals+1)
    xValues = x0 + v0 * tValues + \
              0.5 * a * tValues**2
    return tValues, xValues

def displayPositions(tValues, xValues):
    print('\n%7s %10s' % ('T', 'X'))
    for t, x in zip(tValues, xValues):
        print('%10.4f %12.6f' % (t, x))

def getPosNumber(prompt):
    # exercise for the reader
```

```
def displayTerminationMessage():
    print("""
Programmed by Stew Dent.
Date: %s.
End of processing.""" % ctime())

def main():
    print('\n' + '-' * 80)

    x0 = float(getPosNumber('Enter the initial position in meters > 0: '))
    v0 = float(getPosNumber('Enter the initial velocity in m/s > 0: '))
    a = float(getPosNumber(
             'Enter the initial acceleration in m/s^2 > 0: '))
    stop = float(getPosNumber(
             'Enter the time at which to stop in seconds > 0: '))
    intervals = int(getPosNumber('Enter the number of intervals > 0: '))

    tValues, xValues = positions(x0, v0, a, stop, intervals)
    displayPositions(tValues, xValues)
    displayTerminationMessage()

main()
```

- Sample output from this programs follows on the next slide.

```
Enter the initial position in meters > 0: 1
Enter the initial velocity in m/s > 0: 1
Enter the initial acceleration in m/s^2 > 0: 1
Enter the time at which to stop in seconds > 0: 5
Enter the number of intervals > 0: 20

      T          X
    0.0000     1.000000
    0.2500     1.281250
    0.5000     1.625000
    0.7500     2.031250
    1.0000     2.500000
    1.2500     3.031250
    1.5000     3.625000
    1.7500     4.281250
    2.0000     5.000000
    2.2500     5.781250
    2.5000     6.625000
    2.7500     7.531250
    3.0000     8.500000
    3.2500     9.531250
    3.5000    10.625000
    3.7500    11.781250
    4.0000    13.000000
    4.2500    14.281250
    4.5000    15.625000
    4.7500    17.031250
    5.0000    18.500000

Programmed by Stew Dent.
Date: Fri Jun 8 12:51:13 2018.
End of processing.
```

•Rewrite the program to compute the trajectory of a projectile using ***vectorization***. (trajectoryVector.py)

•This is a quadratic equation of the form: $ax^2 + bx + c$

•Therefore we can find the value of x for which y = 0.

•There are two roots to a quadratic equation, we want the one for which x is greater than zero.

•Use this as the upper bound for the x values.

```
from math import pi, cos, tan, sqrt
import numpy as np

v0 = 5.0          # m/s
y0 = 1.           # m
theta = 60        # degrees
intervals = 20
# display the values of the variables
print("""
v0         = %.1f m/s
y0         = %.1f m
theta      = %d degrees
intervals = %d """ % (v0, y0, theta, intervals))
xValues, yValues = trajectory(v0, y0, theta, intervals)
displayTrajectory(xValues, yValues)
```

```
def trajectory(v0, y0, theta, intervals):
    theta = theta * pi / 180  # convert to radians
    G = 9.81
    a = -G / (2 * v0**2 * cos(theta)**2)
    b = tan(theta)
    c = y0
    distance = (-b - sqrt(b*b - 4*a*c)) / (2 * a)
    #root2 = (-b + sqrt(b*b - 4*a*c)) / (2 * a)
    xValues = np.linspace(0., distance, intervals+1)
    yValues = ((a * xValues) + b) * xValues + c
    return (xValues, yValues)

def displayTrajectory(xValues, yValues):
    print("\n%8s %10s" %('X', 'Y'))
    for x, y in zip(xValues, yValues):
        print("%8.4f %10.6f" % (x, y))
```

```
gravity   = 9.81 m/s^2
v0        = 5.0 m/s
y0        = 1.0 m
theta     = 60 degrees
intervals = 20

      X          Y
  0.0000   1.000000
  0.1341   1.218161
  0.2682   1.408095
  0.4023   1.569801
  0.5364   1.703280
  0.6705   1.808531
  0.8046   1.885555
  0.9387   1.934351
  1.0728   1.954919
  1.2069   1.947261
  1.3410   1.911374
  1.4751   1.847261
  1.6092   1.754919
  1.7434   1.634351
  1.8775   1.485555
  2.0116   1.308531
  2.1457   1.103280
  2.2798   0.869801
  2.4139   0.608095
  2.5480   0.318161
  2.6821  -0.000000
```

- Rewrite the program to calculate the area under a circle to compute the value of π. (`areaCircleVector.py`)

```
from time import ctime
from numpy import sum, minimum, sqrt, pi
import numpy as np
```

- Notice that those functions that must be used with arrays are imported from *numpy*.

```
def getPosInt(prompt):
    while True:
        number = input(prompt)
        if number != '':
                 try:
                number = eval(number)
            except:
                print('Invalid input!')
                 else:
                if type(number) is int:
                    if number > 0:
                        break
                    else:
                        print(number, 'is not a positive integer!’)
                          else:
                    print(number, 'is not an integer!’)
         else:
            print('Missing input!')
return number

main()
```

- The output from the program is:

```
Enter the number of intervals (>0): 1000000

Vectorized value of pi is 3.141592652413872
Calculated value of pi is 3.141592652413756
                 math.pi is 3.141592653589793
```

- The calculated values for π are less than the real value of π, why is that?

- Suppose we have several equations of the form:

$$c_0a + c_1b + c_2c + c_3d = k$$

- For example:

$$2a – b - 3c – d = 0$$

$$3a + b – 2c + d = 10$$

$$2a + b + c + d = 7$$

- We would like to know if a=1, b=2, c=-1 and d=3 is a solution for any of these equations.
- Let one array hold the coefficients and another array hold the solution.
- Multiply the two vectors and add together the resulting elements.
- If the result equals k we have a solution to the equation.

(equationVerification.py)

```
from time import ctime
import numpy as np
def main():
    solution = np.array([1, 2, -1, 3])
    c0 = float(input('Enter the first coefficient: '))
    c1 = float(input('Enter the second coefficient: '))
    c2 = float(input('Enter the third coefficient: '))
    c3 = float(input('Enter the fourth coefficient: '))
    k = float(input('Enter the constant: '))
    equation = np.array([c0, c1, c2, c3])
    if k == np.sum(solution * equation):
      print('\na=1, b=2, c=-1, d=3 is a solution to the'
            + ' equation.')
    else:
      print('\na=1, b=2, c=-1, d=3 is not a solution to the'
              + ' equation.')
    print("""
Programmed by Stew Dent.
Date: %s.
End of processing.""" % ctime())

main()
```

```
Enter the first coefficient: 2
Enter the second coefficient: -1
Enter the third coefficient: -3
Enter the fourth coefficient: -1
Enter the constant: 0

a=1, b=2, c=-1, d=3 is a solution to the equation.

Programmed by Stew Dent.
Date: Fri Jun 2 13:13:24 2018.
End of processing.
```