# Array Conditional Statements and numpy.where()

## Array Conditional Statements and `numpy.where()`

### Comparison and Bitwise Operations on Arrays

We can apply comparison operators to arrays:

```
[51]: a1 = np.array([1,  2, 3, 4, 5])

      a2 = np.array([2, 1, 5, 6, 4])

      a1 < a2
```

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

As you can see this gives us an array of booleans, each element representing the outcome of comparing the corresponding element of `a1` to `a2`.

What if we wanted to combine the boolean arrays with a logical operator? For example, if we want an array of booleans for the condition `a1` is less-than `a2` and greater than `2`. Unfortunately the boolean comparison operators we used in the If Statements won't work, for example using `and`:

```
[53]: a1 < a2 and a1 > 2
```

```
      ␣
   ↪---------------------------------------------------------------------------

          ValueError                                Traceback (most recent call␣
      ↪last)

          <ipython-input-53-74cb81b02f02> in <module>
      ----> 1 a1 < a2 and a1 > 2


          ValueError: The truth value of an array with more than one element is␣
      ↪ambiguous. Use a.any() or a.all()
```

In order to combine boolean arrays (without a loop) we need to use **bitwise** operators.

Bitwise operators treat numbers as a string of bits and act on them bit by bit. In the case of a boolean array, the operator acts on it element by element. The bitwise operators we are interested are:

Operator

Name

Analogous boolean operator

&

Bitwise and

and

|

Bitwise or

or

~

Bitwise complement

not

(See https://wiki.python.org/moin/BitwiseOperators for a more comprehensive list and explanation of bitwise operations.)

Returning to our original example:

```
[55]: (a1 < a2) & (a1 > 2)
```

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

Note that the comparisons must be grouped in brackets for this to work:

```
[56]: a1 < a2 & a1 > 2
```

```
      ␣
    ↪---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call␣
    ↪last)

        <ipython-input-56-c3606bc24b97> in <module>
      ----> 1 a1 < a2 & a1 > 2


        ValueError: The truth value of an array with more than one element is␣
    ↪ambiguous. Use a.any() or a.all()
```

**Example - Random Points in a Region**

We can use `np.where()` to check which points in an array lie inside or outside of region.
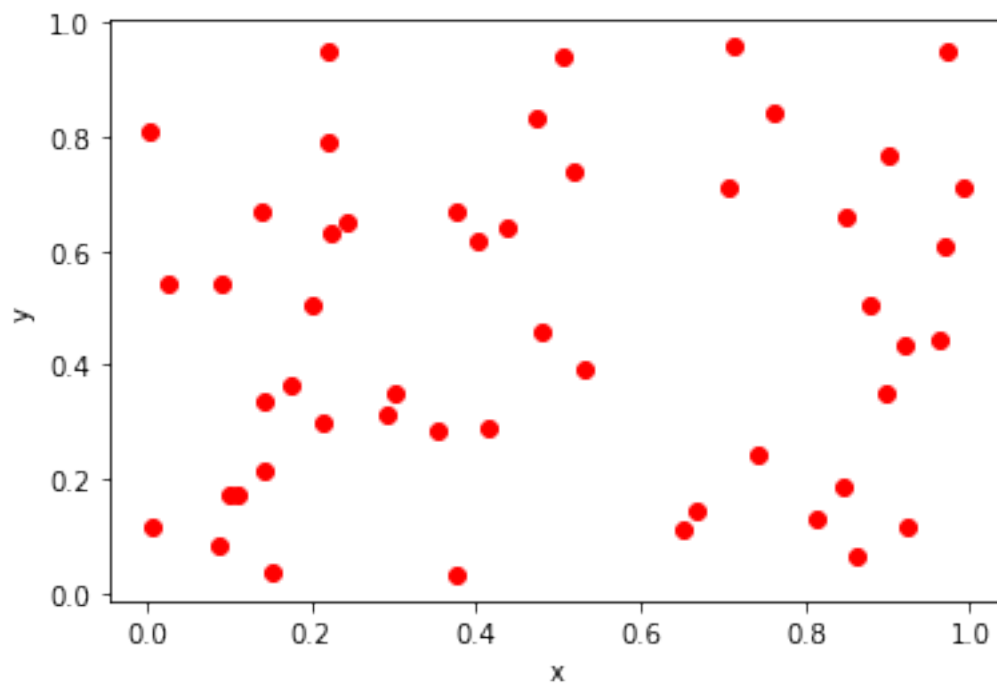
First let's generate an array of 50 random points in 2D space:

```
[46]: points = np.random.random((2, 50))

      plt.plot(points[0, :], points[1, :], 'ro')

      plt.xlabel('x')
      plt.ylabel('y')

      plt.show()
```



Note that axis 0 of `points` is used to represent the x and y values, and axis 1 represents points. i.e. for the points $(x_0, y_0), (x_1, y_1), \ldots, (x_{49}, y_{49})$, `points` is:

x0

x1

x2

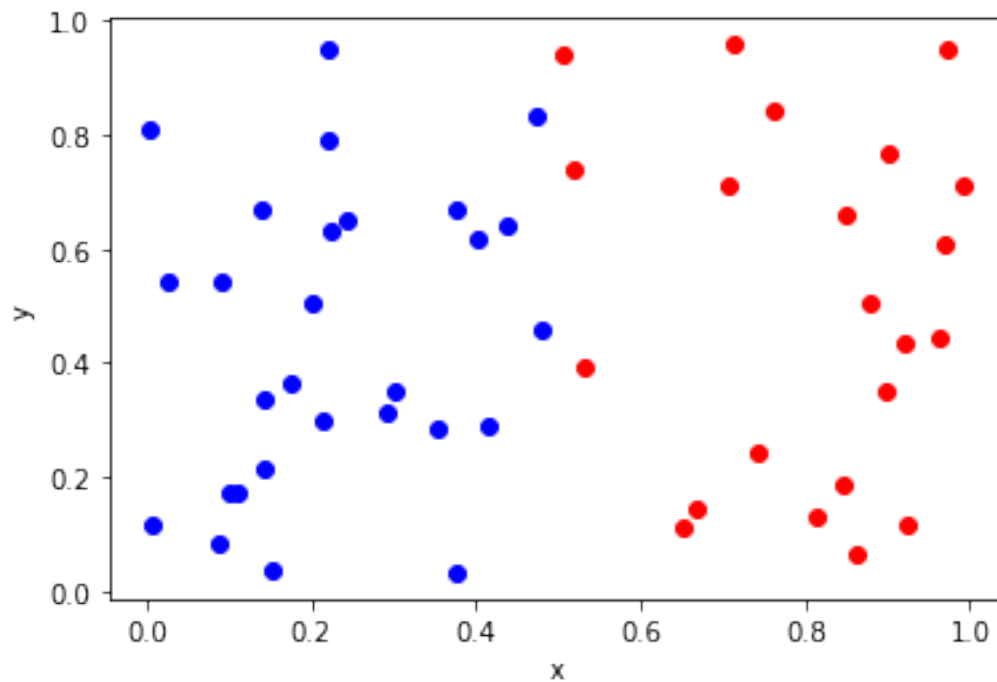x3

x4

...

x48

x49

y0

y1

y2

y3

y4

...

y48

y49

Now, let's plot the points which lie to the left of 0.5 as blue and the others as red:

```
[47]: is_left = points[0, :] < 0.5 #True where left of 0.5

plt.plot(points[0, is_left], points[1, is_left], 'bo')
plt.plot(points[0, ~ (is_left)], points[1, ~ is_left], 'ro')

plt.xlabel('x')
plt.ylabel('y')
```

[47]: Text(0, 0.5, 'y')

Note that, in the example above, we have used an array of booleans to **slice the elements of the array which are true**. We have also use the **bitwise compliment** to get the complement of our comparison result, there is no need to recalculate it.
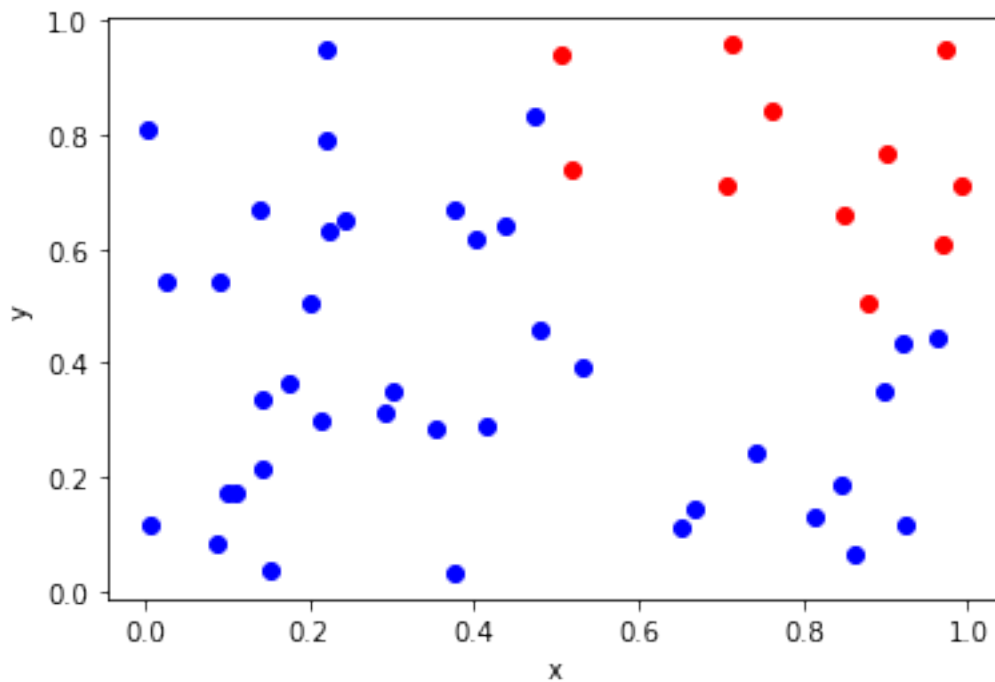
Now, lets plot the points right of 0.5 and above 0.5 (the top left square) as red and the rest as blue (remember the **bitwise and**):

```
[48]: #True if in top left square
      is_top_left = (points[0, :] > 0.5) & (points[1, :] > 0.5)

      plt.plot(points[0, is_top_left], points[1, is_top_left], 'ro')
      plt.plot(points[0, ~ (is_top_left)], points[1, ~ is_top_left], 'bo')

      plt.xlabel('x')
      plt.ylabel('y')
```

[48]: Text(0, 0.5, 'y')



**numpy.where()**

Documentation

numpy.where(condition[,x, y])

Returns elements chosen from x or y depending on the condition. If no x or y arguments are provided it returns and array of indices.

```
[17]: arr = np.arange(10, 20)

arr_where = np.where(arr > 15)

print('arr1:', arr)
print('Indices where arr1 is greater than 15:', arr_where)
print('The sub-array of arr1 that is greater than 15:', arr[arr1_where])
```

```
arr1: [10 11 12 13 14 15 16 17 18 19]
Indices where arr1 is greater than 15: (array([6, 7, 8, 9]),)
The sub-array of arr1 that is greater than 15: [16 17 18 19]
```

If both x and y is specified, the elements of the returned array come from x if `condition` is true, or from y if `condition` is false.

```
[22]: x = np.linspace(1, 5, 5)
#y = np.linspace(-5, -1, 5)
y = -x

condition = [True, False, True, True, False]

print('x:', x)
print('y:', y)
print('Condition:', condition)
print('x where True, y where False:', np.where(condition, x, y))
```

```
x: [1. 2. 3. 4. 5.]
y: [-1. -2. -3. -4. -5.]
Condition: [True, False, True, True, False]
x where True, y where False: [ 1. -2.  3.  4. -5.]
```

**Example - Piecewise defined functions**

One use for `np.where()` is to define a piecewise defined function that works on arrays.
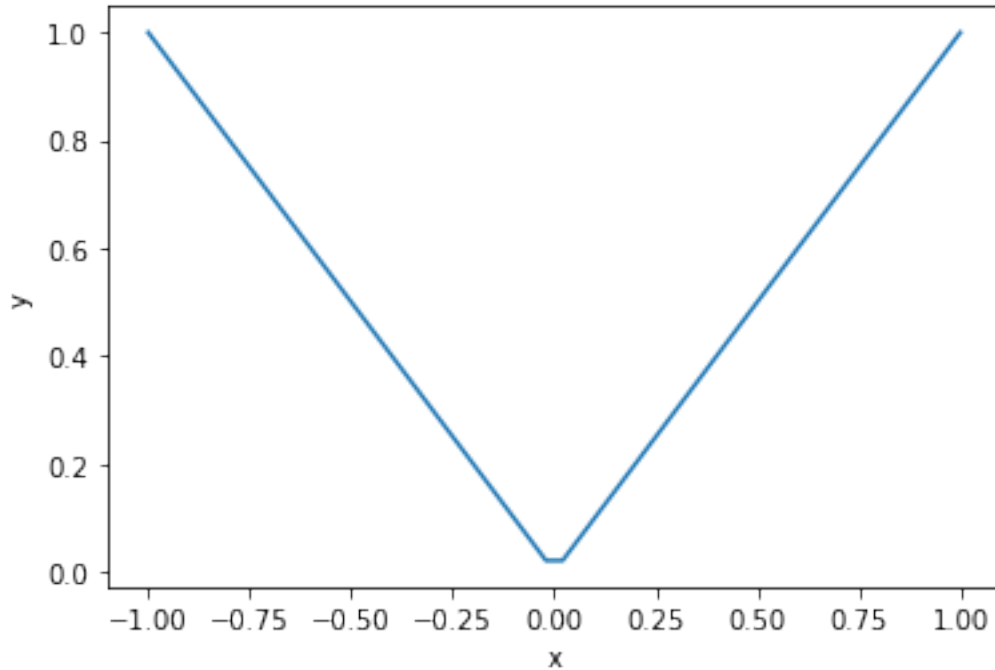
As a first example, let's use `np.where()` to plot the absolute value function (you should really use `np.abs()` for this):

$$y = \begin{cases} -x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

```
[24]: x = np.linspace(-1, 1)

y = np.where(x >= 0, x, -x)
```

```
plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Note that, in the plot above, the line does not reach zero, but flattens out to a value above it. This is because the array **x** does not contain the value 0, but values around it.

Now, consider the piecewise function:

$$f(x) = \begin{cases} -(x+1)^2 + 1 & \text{if } x < -1 \\ -x & \text{if } -1 \le x \ge 1 \\ (x-1)^3 - 1 & \text{if } x > 1 \end{cases}$$

where there are three regions. To handle this we can use 2 `np.where()` calls:

[34]: 
```
x = np.linspace(-3, 3)

#Left condition
y = np.where(x < -1, -(x+1)**2 + 1, -x)

#Right condition
y = np.where(x > 1, (x - 1)**3 - 1, y)

plt.plot(x, y)
```

```
plt.xlabel('x')
plt.ylabel('y')

plt.show()
```