Lecture 6: Logical Statements and Boolean Operators

The purpose of todays lecture is to introduce the use of *Logical Statements* and *Boolean Operators* in Programming.

A simple definition of a logical statement:

- A logical statement is a statement that produces an output which either has the value **True** or **False**.
- From a coding perspective, we could also map those two states onto a binary variable which takes on the value 1 or 0.
- In practice these two representations can be used interchangeability and implicitly.
- This type of variable is known as a **Boolean** variable.
- In scientific applications in Psychology/Neuroscience we will make use of logical statements to develop boolean variables as

1. logical *indexing* into data arrays in data analysis applications.
2. conditional* code execution.

In [31]:

```
import numpy as np
from numpy import random
from matplotlib import pyplot as plt
import pandas as pd
```

Comparison Operators

- A logical statement almost always involves comparisons. Examples of comparisons that come to mind might be

1. `==` - equal
2. `!=` - not equal
3. `>` - greater than
4. `>=` - greater than or equal
5. `<` - less than
6. `<=` - less than or equal

This set of examples looks at simple logical statements on variables, that returns either **True** or **False**

In [32]:

```
x = 5   #This is a statement defining the variable x with the value 5.
x == 5   #Test of Equality, returns the value True
```

Out[32]:

```
True
```

In [33]:

```
x = 5
test = (x == 5)  #Equal, True Statement, correct syntax,readable formatting
print(test)
```

```
True
```

In [34]:

```
x = 5
test = (x != 5)  #Not Equal, False Statement
print(test)
```

```
False
```

- The internal representation of the Boolean variable is **both** True/False and 1/0

In [35]:

```
x = 5
test = (x == 5)
print(test)
istestone = (test == 1) # Here I test if the boolean variable
                        #with value True is the same as 1
print(istestone)
```

```
True
True
```

In [36]:

```
x = 5
test = (x != 5)  #Not Equal, False Statement
print(test)
istestzero = (test == 0) # Here I test if the boolean variable
                         #with value False is the same as 0
print(istestzero)
```

```
False
True
```

Logical Statements on numpy arrays

- When applying logical statements to an array, we return a numpy **Boolean** array of equal size containing Boolean variables.
- Because the Boolean array is a numpy array, we can do *math* with it.

In [37]:

```
x = np.array([1, 2, 3, 4, 5]) #make an array from a list 1,2,3,4,5
print(x)
test = (x < 3)  # use the less than logical operation
print(test) #print the output of the numpy array
test.dtype  # with numpy arrays we use the dtype method
            #to find out the data type.
```

```
[1 2 3 4 5]
[ True  True False False False]
```

Out[37]:

```
dtype('bool')
```

In [38]:

```
x = np.array([1, 2, 3, 4, 5]) #make an array from a list 1,2,3,4,5
print(x)
#How many elements of the array are less than 3?
test = (x < 3)  # use the less than logical operation
print(test)
nless = np.sum(test) #Here I sum the boolean array,
            #implicitly converting the True to 1 and False to 0
print(nless)
```

```
[1 2 3 4 5]
[ True  True False False False]
2
```

In [39]:

```
x = np.array([1, 2, 3, 4, 5]) #make an array from a list 1,2,3,4,5
print(x)
#How many elements of the array are less than 3?
test_less = (x < 3)  # use the less than logical operation
print(test_less)
test_greater = (x > 3)  # greater than
print(test_greater)
```

```
[1 2 3 4 5]
[ True  True False False False]
[False False False  True  True]
```

In [40]:

```
x = np.array([1, 2, 3, 4, 5]) #make an array from a list 1,2,3,4,5
print(x)
#How many elements of the array are less than 3?
test_less = (x < 3)  # use the less than logical operation
print(test_less)
test_greater_eq = (x >= 3)  # greater than or equal
print(test_greater_eq)
```

```
[1 2 3 4 5]
[ True  True False False False]
[False False  True  True  True]
```

In [41]:

```
x = np.array([1, 2, 3, 4, 5]) #make an array from a list 1,2,3,4,5
print(x)
test_less_eq = (x <= 3)  # less than or equal
print(test_less_eq)
test_greater = (x > 3)  # greater than
print(test_greater)
```

```
[1 2 3 4 5]
[ True  True  True False False]
[False False False  True  True]
```

- The complement of == is !=

In [42]:

```
x = np.array([1, 2, 3, 4, 5]) #make an array from a list 1,2,3,4,5
print(x)
test_eq = (x == 3)  # equal
print(test_eq)
test_neq = (x != 3)  # not equal
print(test_neq)
```

```
[1 2 3 4 5]
[False False  True False False]
[ True  True False  True  True]
```

Logical complement, or logical_not

- The complement of a boolean variable is the opposite state. The complement of **True** is **False** and the complement of **False** is **True**
- There are two ways to take the complement of a Boolean array. One is the numpy method `logical_not` or by using the operator `~`

In [43]:

```
x = np.array([1, 2, 3, 4, 5]) #make an array from a list 1,2,3,4,5
print(x)
test_greater = (x > 3)  # greater than
test_not_greater = np.logical_not(test_greater)
print('test greater = ', test_greater)
print('test greater complement  = ',test_not_greater)
```

```
[1 2 3 4 5]
test greater =  [False False False  True  True]
test greater complement  =  [ True  True  True False False]
```

Rules of Logic - Set Theory Perspective

- The rules of logic and set theory are important to understand in meaningfully evaluation logical statements.
- We can view the array `x =[1 2 3 4 5]` as a **set**.
- Any of the logical statements uses a comparison operation to divide the set into two **subsets**, one where the logical statement is **True**, and another where the logical statement is **False**.
- **The most important thing to remember is that the two subsets must combine to the original set**

In [44]:

```
x = np.array([1, 2, 3, 4, 5]) #make an array from a list 1,2,3,4,5
print(x)
test_equal = (x == 3)  # equal
test_not_equal = (x != 3) #not equal
print('test equal  = ', test_equal)
print('test equal complement  = ',test_not_equal)
```

```
[1 2 3 4 5]
test equal  =  [False False  True False False]
test equal complement  =  [ True  True False  True  True]
```

Creating Boolean Arrays

- Sometimes we need to manually create Boolean variable or array.
- A Boolean variable can be obtained like this

In [45]:

```
#Boolean variables
boolean_var1 = True
print(boolean_var1)
boolean_var2 = bool(0)
print(boolean_var2)
```

```
True
False
```

- Boolean Arrays are similar

In [46]:

```
#Boolean Arrays
boolean_array1 = [True,False,False,True]
print(boolean_array1)
boolean_numpy_array1 = np.array(boolean_array1)
print(boolean_numpy_array1)
```

```
[True, False, False, True]
[ True False False  True]
```

In [47]:

```
#Boolean arrays from 0 and 1
integer_array = np.array([0,1,1,0])
print(integer_array)
boolean_array3 = np.array([0,1,1,0],dtype=bool)  # force the data type to boolean
print(boolean_array3)
```

```
[0 1 1 0]
[False  True  True False]
```

Conversion to a Boolean Array - I can turn anything into a boolean array.

- What are the rules?

1. 0, None, False or empty strings ARE **False**
2. Values other than 0, None, False or empty strings ARE **True.**

In [48]:

```
strange_list = [1, 0.5, 0, 0.0, None, 'a', '', ' ',True, False]
bool_arr = np.array(strange_list, dtype=bool)
print(strange_list)
print(bool_arr)
```

```
[1, 0.5, 0, 0.0, None, 'a', '', ' ', True, False]
[ True  True False False False  True False  True  True False]
```

- Interpreted languages like Python, Matlab, R are easy to use because you can be careless and things still kind of work.
- **This is their strength AND their weakness**
- Its really easy to get code to run
- Its really easy to have a mistake in your code and have your code seem to work in these languages.

Logical Statements to Compare Arrays

- Logical Operators can be used to compare arrays of equal size and shape.
- Such comparisons proceed on an element by element basis and return a Boolean array of the same size

In [49]:

```
rng = random.default_rng(seed = 21)
array_a = rng.integers(0,10,8) #array of random integers
array_b = rng.integers(0,10,8)
test = (array_a > array_b) # is a greater than b
print(array_a)
print(array_b)
print(test)
```

```
[3 7 3 6 4 7 3 0]
[2 6 6 9 2 4 9 1]
[ True  True False False  True  True False False]
```

- You can embed mathematical calculations into logical statements.

In [50]:

```
x = np.array([1,2,3,4,5])
y = 2**x
z = x**2
test = (y == z)
print(x)
print(y)
print(z)
print(test)
```

```
[1 2 3 4 5]
[ 2  4  8 16 32]
[ 1  4  9 16 25]
[False  True False  True False]
```

- You can do it in 1 step if you want.

In [51]:

```
onesteptest = (2**x == x**2)
print(onesteptest)
```

```
[False  True False  True False]
```

Boolean Operators

- We can combine Logical Statements using **Boolean operators** to express more complex logical expressions.
- Boolean Operators are operators that work on Boolean variables to create **compound** logical statements.
- This is equivalent to the idea arithmetic operators are operators that work on floating point numbers.

`Not (~)` operator

- This Boolean operator flipped the value of the Boolean variable between True and False.

In [52]:

```
bool_array1 = np.array([0,0,0,1,1,0,1,0],dtype=bool)
print(bool_array1)
bool_array2 = ~bool_array1
print(bool_array2)
```

```
[False False False  True  True False  True False]
[ True  True  True False False  True False  True]
```

Table of Boolean Operators.

- The following table summarizes the bitwise Boolean operators and their equivalent ufuncs:
- There are 3 logical operations: and, or, not

Operator	Meaning	Example
and	True if both the operands are true	x & y
or	True if either of the operands is true	x | y
not	True if operand is false (complements the operand)	~x

VERY IMPORTANT POINT

- You will read online about the `and` and `or` operators built in to Python.
- Many students in my classes have wasted a lot of time trying to use this.
- They dont work properly on `numpy` arrays.
- Please use `&` for **and** `|` for **or**.

Combining Boolean operators on arrays can lead to a wide range of efficient logical operations

- Example 1 - & (AND)

In [53]:

```
rng = random.default_rng(seed = 1967)
a = rng.integers(-9,10,12) #make integers from -9 to 9
print(a)
x = (a < 5) #find the integers less than 5
print('x = ',x) #This is a Boolean which tells where in a I can find x
y = (a > -5) #find the integers greater than -5
print('y = ',y) #This is a Boolean which tells me where in a I can find y
```

```
[ 3  9  4  5 -1 -5 -1 -2  0 -9  1 -7]
x =  [ True False  True False  True  True  True  True  True  True  True
True]
y =  [ True  True  True  True  True False  True  True  True False  True F
alse]
```

- To find the integers between -5 and 5
- we need to take the **intersection** of the subset of a represented by x with the subset of a represented by y.
- In math (or logic), we would write this is as $x \cap y$
- In python, we can carry this out with an `&` operator

In [54]:

```
z1 = x & y #The & symbol requires that both conditional statements be true
print(z1)
```

```
[ True False  True False  True False  True  True  True False  True False]
```

- Example 2 - | (OR)

In [55]:

```
a = rng.integers(-9,10,12) #make integers from -9 to 9
print(a)
x = (a > 5) #find the integers greater than 6
print('x = ',x) #This is a Boolean which tells where in a I can find x
y = (a < -5) #find the integers less than -6
print('y = ',y) #This is a Boolean which tells me where in a I can find y
```

```
[ 6  2  8 -3  5 -8  0  5  2 -4  8 -8]
x =  [ True False  True False False False False False False False  True F
alse]
y =  [False False False False False  True False False False False False
True]
```

- To find the integers *either* greater than 5 *or* less than -5
- We need to take the **union** of the subset of a represented by x with the subset of a represented by y.
- In math (or logic), we would write this is as $x \cup y$
- In python, we can carry this out with an | operator

In [56]:

```
z1 = (x | y) #The | symbol requires that either one of the conditional statements be true
print(z1)
```

```
[ True False  True False False  True False False False False  True  True]
```

Working with Boolean Arrays

- Why do we want to make use of Boolean arrays?
- Here I will show some useful operations that Boolean arrays will allow us to do.
- I will show an example of how we make use of Boolean arrays to organize data.

Counting entries

- To count the number of **True** entries in a Boolean array, `np.count_nonzero` is useful:

In [57]:

```
x = rng.integers(0,9,12)
print(x)
print(x<6)
# how many values less than 6?
n6 = np.count_nonzero(x < 6)
print(n6)
```

```
[8 5 4 3 0 3 5 8 6 7 4 8]
[False  True  True  True  True  True  True False False False  True False]
7
```

- We see that `count_nonzero` counted the number of entries that were **True** in the entire matrix.
- Python interprets **True** as having the numeric value of 1, and **False** as zero.
- Another way to get at this information is to use `np.sum`:

In [58]:

```
nsum6 = np.sum(x < 6)
print(nsum6)
```

```
7
```

In [59]:

```
y = x >= 6
nsumg6 = np.sum(y) # count the number greater than 6
print(nsumg6)
nsuml6 = np.sum(~y) # count hte number less than 6
print(nsuml6)
```

```
5
7
```

Global array tests

- np.any can be used to test the entire array for whether *any* element is **True**

In [60]:

```
# are there any values greater than 8?
print(x)
np.any(x > 8)
```

```
[8 5 4 3 0 3 5 8 6 7 4 8]
```

Out[60]:

```
False
```

In [61]:

```
# are there any values less than zero?
print(x)
np.any(x < 0)
```

```
[8 5 4 3 0 3 5 8 6 7 4 8]
```

Out[61]:

```
False
```

- `np.all` can be used to test the entire array for whether *all* elements are **True**

In [62]:

```
# are all values less than 10?
print(x)
np.all(x < 10)
```

```
[8 5 4 3 0 3 5 8 6 7 4 8]
```

Out[62]:

```
True
```

In [63]:

```
# are all values equal to 6?
print(x)
np.all(x == 6)
```

```
[8 5 4 3 0 3 5 8 6 7 4 8]
```

Out[63]:

```
False
```

Boolean Indexing

- We can make use of the Boolean variable that is the results of a logical statement as the **index** into a variable to extract the variables that meet the condition of the logical statement.
- We often will want to return the **index** into an array that meets a logical condition.

In [64]:

```
my_array = rng.integers(0,10,20)
print(my_array)
#test if the array entries are bigger than 5
test = (my_array > 5)
print(test)
#how many values are there greater than 5
count = np.sum(test)
print(count)
my_greater_than_5_array = my_array[test]
print(my_greater_than_5_array)
np.size(my_greater_than_5_array)
```

```
[7 9 8 4 6 6 7 2 3 5 2 5 0 4 0 2 4 7 8 0]
[ True  True  True False  True  True  True False False False False False
 False False False False False  True  True False]
8
[7 9 8 6 6 7 7 8]
```

Out[64]:

```
8
```

Implicit Boolean indexing

- We can do the Boolean indexing implicitly, and I think it makes it more readable.

In [65]:

```
onestep_greater_than_5 = my_array[my_array > 5]
print(onestep_greater_than_5)
```

```
[7 9 8 6 6 7 7 8]
```

- Suppose my task is to find the mean of all entries in the array greater than 5.
- I can do this with a simple logical statement.
- Its really important to look at this and see it in spoken language terms.

In [66]:

```
bigmean = np.mean(my_array[my_array > 5])
print(bigmean)
```

```
7.25
```

- This will prove very useful in data summary and visualization at the end of the quarter.