Some fundemental elements of programming II

Conditional Tests, Boolean Logic, and Other Control Flow

As we said before, the core of data science is computer programming. To really explore data, we need to be able to write code to 1) wrangle data into a suitable shape for analysis and 2) do the actual analysis and visualization.

If data science didn't involve programming – if it only involved clicking buttons in a statistics program like SPSS – it wouldn't be called data science. In fact, it wouldn't even be a "thing".

In this tutorial we are going to start looking at a few more core elements of computer programs.

Learning outcomes:

- Comparisons and Logical operators
- Numpy Arrays Comparisons and Logical operators
- `If, then` statements
- `While` Loops

Boo! Logical Tests and Boolean Operators

Believe it or not, everything that happens on your phone or computer comes down to lots (and I mean **LOTS**) of little decisions based on one or two inputs that can be either "True" or "False", and an output that can also be "True" or "False". Seriously, everything on any digitial device – from Tik Tok videos to your Python code – comes down to a whole bunch of truths and falsehoods (ones and zeros) that are themselves the result of decisions based on other truths and falsehoods. The "decision makers" are actual physical (but teeny teeny tiny) devices that are combinations of things called *transistors* (https://en.wikipedia.org/wiki/Transistor). Transistors perform conditional tests and logical operations on data.

Two are the primary operations performed by transistors:

- ***Comparison*** operations like `==` (equals) and `>` (greater than) that yield `True` or `False`
- ***Logical*** operations that use ***Boolean logic***, which compares two logical inputs and returns `True` or `False` like `A and B` (`True` only if both A and B are `True`) and `A or B` (`True` if either A or B – or both – are `True`).

Let's play with this. It might seem a bit silly and obvious now, but the power of logical tests will reveal itself soon.

Comparison operators

These are operators that test a single value. Imagine wanting to ask, whether the number of lives my cat has (9 for what I was told yesterday, when I was born (https://en.wikipedia.org/wiki/Cat#Superstitions_and_rituals)) is different than the number of lives of Schrödinger's cat (https://en.wikipedia.org/wiki/Schr%C3%B6dinger%27s_cat) has. Number to number type of questions.

Let's set a variable `x` to 11. (Why only go to ten when you can go to 11?)

In [ ]:
```
1 x = 11
2 x
```

Now let's do some logical tests on our variable `x`. Let's see if `x` is less than `42`.

In [ ]:
```
1 x < 42
```

Now you test if `x` is greater than `42`.

In [ ]:
```
1
```

We can also test for equality. Is `x` equal to `42`?

In [ ]:
```
1 x == 42
```

In [ ]:
```
1 x == 11
```

Finally, we can test for *inequality*. (We test whether it is true that x is *not* equal to a specific number).

In [ ]:
```
1 x != 42
```

The exclamation point here means "not", so the experession `x != 42` can be read as "is x not equal to 42?"

And the answer is "That's `True`! The variable `x` is not equal to 42!"

Now you test `x` to see if it's not equal to `11`. Is it?

In [ ]:
```
1
```

As you might have noticed, all these operations are *built in.* This means that we did not have to import any specific package to access the operations. Python provides these operations as they are core functionality, the bread and butter of most users, or better said, of most programmers. Like you!

Logical operators

So far we have been dealing with testing operations on single numbers. Often times it is important to be able to test multiple operations and compare them, say if `a > 0` **`and`** `b < 0`. Operations that compare or combine two statements are called **logical**. Logical operations such as `and` and `or` are extremely important and widely used in computer programming, mathematics, neuroscience and in real life.

Python provides binary operations `built in`, so there is not requirement to import a specific library.

Imagine wanting to compare the number of cake slices eaten per day by the average individual in three different countries.

In [ ]:
```
# average number of cake slices eaten in
USA = 3  # the United States of America
IT = 2   # Italy
CA = 4   # Canada
```

Imagine wanting to know if BOTH the USA AND Canada eat more cake than Italy. We can first compare if the average citozen eat more cake in Italy or the USA:

In [ ]:
```
(USA > IT)
```

Alright, it looks like more cake is eaten in the USA. What about Canada?

In [ ]:
```
(CA > IT)
```

If we wanted to compare both the USA and Canada at the same time, in python we could conveniently write the operation as follows:

In [ ]:
```
(USA > IT) and (CA > IT)
```

The statment about is *only* true if *both* statements are true. Let's test it.

In [ ]:
```
# We will use a temporary value for canada and
# then repeat the logical operation with the new value
CA_temp = 1
(USA > IT) and (CA_temp > IT)
```

OK what happened there is that whereas the first statement was true (3 > 2 slices of cake) the second was not true (1 is not more then 2 slices of cake) and the whole statement returned `False`. The `and` operator returns `True` only if all composing statements return `True`.

Another logical operation of Key value `OR`. `OR` returns `True` if only one of the two statements is `True`, even if the other is `False`. We can try it:

In [ ]:
```
CA_temp = 1
(USA > IT) or (CA_temp > IT)
```

What do you expect would be the result if you were to run `or` between the original statements:

- `(USA > IT)`
- `(CA > IT)`

Try this out:

In [ ]:
```

```

Another helpful operator, often used in similar questions is `not` . The `not` operator is a modifier that changes the value of the output of other operators suchas `>` , `=` , `and` , etc.

For example, if `not` is used, the number of slices of cake eaten by the average citizen in Canada is **not** more than those eaten in the USA:

In [ ]:
```
not(CA > USA)
```

Whereas this is is obeviously `True`

In [ ]:
```
(CA > USA)
```

So, I like to think about `not` as a modifier. It can become useful in many cases, especially when the output of two or more statments needs to be modified (flipped) for the code to advance. For example, the following code shows how three boolean statements (all set to `True`) can be modified but a boolean `not` to save my health.

In [ ]:
```
# Save my belly
ihaveeatencake = True
itislatenight = True
ididnotexercisetoday = True

INeedToEatCake = not(ihaveeatencake and itislatenight and ididnotexerci
INeedToEatCake
```

if, then statements

Fundamental to any mature software or code system is the ability to express conditional statements such as `if` , `then` statements. These are among the most basic building blocks of coding, as they allow controlling the flow and allowing a certain operations to occur. For example, operation `a` might be performed only `if` a specific condition 'C' is met. Say, I can eat my cake only if I have been diligent and went out for a nice and long run today.

A couple of basic numerical examples can get us started.

In [ ]:
```
x = 3
a_big_number = 100

if x > 5 :
    print('Yes, it is a big number!')
else :
    print('Nope, small!')
```

We can even add another test using the elif ("else if") statement. When measuring the temperature in Austin TX, we would say:

In [ ]:
```
current_temp = 70

if current_temp >= 90 :
    print('Too hot!')
elif current_temp <= 50 :
    print('Too cold!')
else :
    print('Just right!')
```

`If, then` statements can be combined with logical operators also and help manage complex decisions in a matter of a few lines:

In [ ]:
```
current_temp = 45
if not( (current_temp >= 90) or (current_temp <= 50) ) :
    print('I will ride my bicycle to school!')
else :
    print('Too cold! I will take the car or walk.')
```

To practice with `if , then` statements, write code that when asked if we should eat cake returns `True` only if we have eaten less then 2 slices of cake today and no cake yesterday and the day before yesterday. The code should otherwise tell us to eat soup.

In [ ]:
```

```

In [ ]:
```

```

Again, all these operations are *built in* this means that we did not have to import any specific package. Python provides these basic operations as they are the bread and butter of most users, or better said, of most programmers, like you.

Operators on NumPy arrays

There are other types of operators that do not come standard with Python but that are part of other packages and need to be imported. These operators behave differently.

When dealing with arrays, instead of individual numbers, things look slightly different. For example, if we wanted to perform a logical operation between two sets of numbers, e.g., two arrays, operations (= , > , etc) will work sometimes but not others.

Let's take a look at how we would perform comparisons and logical operations with NumPy arrays.

In [3]:
```
import numpy as np # We import NumPy as we are working on arrays
```

In [ ]:
```
myRnds = np.random.randn(1, 5) # we create an array of random numbers
myRnds
```

Now, imagine we wanted to know whether each number stored in the Array `myRnds` is positive.

In [ ]:
```
myRnds > 0
```

If we wanted to find out whether any of the numbers in an array are positive, we would use the numpy array method `any` :

In [ ]:
```
logical_array = (myRnds > 0)
np.any(logical_array)
```

If we wanted to test whether all the values in an array are positive, we would use the method `all` .

In [ ]:
```
np.all(logical_array)
```

Because both `all` and `any` apply to numpy atrays, they can also be called as methods of a NumPy Arrays. For example:

In [ ]:
```
logical_array.any()
```

In [ ]:
```
logical_array.all()
```

Numpy arrays also allow comparing values element-wise. This means that we could compare each element of one array with the corresponding element of another array. If the twovectors have the same size.

```
[1, 2, 3] = [1, 4, 3]
```

Would compare 1 to 1, 2 to 4 and 3 to 3.

In [ ]:
```
array_one = np.random.randn(1,5);
array_two = np.random.randn(1,5);
np.logical_and(array_one, array_two)
```

What happens if the two arrays have different size, though?

In [ ]:
```
vector_one = np.random.randn(1,6);
vector_two = np.random.randn(1,5);
np.logical_and(vector_one, vector_two)
```

The not and or operators also exist for numpy arrays:

In [ ]:
```
vector_one = np.random.randn(1,5);
vector_two = np.random.randn(1,5);
np.logical_or(vector_one, vector_two)
```

In [ ]:
```
vector_one = np.random.randn(1,5) > 0;
vector_two = np.random.randn(1,5) > 0;
np.logical_not(vector_one, vector_two)
```

Control Flow: while loops

The final concept and coding skill we will learn today is the `while` loop.

while loops

In tutorial 12, we have taclked the idea of `for` loops. For loops are excellent if we have a good idea of when to stop. If, for example, we are computing a sum between the number 10 and 20, we know already that our for-loop will last for 10 numbers.

In [ ]:
```
aList = np.arange(10,20);
j = 0
for i in aList :
    s = i + aList[j]
    print('The current sum at element ', i, ' is ', s, '(j is',j,')')
    j = j+1
```

In many situations the length of a loop might be unkown before entering the loop. Instead, it might be known that the loop will need to be ended, if a specific condition is met. For example, if we were not in the position to know how many people live in the city of Austin, TX, but we wanted to ask each one of them if they have eaten any cake today, we could could use a while loop. We would continue asking while meeting people that we hve not met before.

In reality, often times it is possible to rewrite a `while` loop as a `for` loop. But there are useful differences between the two loops and for that reason they have both survived.

To show how a `while` loop would work, imagine if I wanted to know if I have eaten more than 14 slices of pizza in the past two weeks (why pizza now? Because we can, pizza is good, too much cake already anyways). Let's assume that I eat 1 slice of pizza per day:

In [ ]:
```
day = 1
pizza = 0
while day < 14:
  pizza = pizza + 1
  print('today is day #,', day, 'so far I ate', pizza, 'pizzas')
  day += 1
```

The syntax of a `while` loop is similar but different than a `for` loop. In this, because of the type of question i asked, I could have asked the same question using a `for` loop. Try practicing,

rewrite the above statement using a `for` loop instead of the `while` loop.

In [ ]:
```
1
```

In general, `while` loops are used when the number of times it will be necessary to iterate is unkown, but the condition which determines the end of the loop is known. The `for` loop instead is used when the number of times the loop will have titerate over is known. Independently of whether the condition to exit the iteration is known or not.

Sometimes, `while` loops provide convenient ways to specify conditions explicitly. For example, we can break a `while` loop in the middle of it if a condition is met:

In [ ]:
```
# here we break the loop (using the break command)
# if the counter is equal to a certain exit value
counter = 1
exit_val = 6
while counter < 20:
  print('The counter is',counter,'The exit value is', exit_val)
  if counter == exit_val:
    break
  counter += 1
```

Try rewriting the above loop as a `for` loop. Is the code easier to read than the one above?

In [ ]:
```
1
```

In [ ]:
```
# here we continue the loop (using the continue command)
# if the counter is equal to a certain value
counter = 0
val = 6
while counter < 20:
   counter += 1
   if counter == val:
    continue
   print('The counter is',counter,'The value is', val)
```

In reality, we could have used the `brake` command also in combination with a `for` loop. So, the situations where a while and for loop diverge are limited. Yet, one or the other are often times preferable, especially for clarity of programming style (e.g., your colleagues reading your code will be happier, if you write simple, easy to read code).

Below one last example with the `while` loop. The loop offers the ability to end with an `else` just like an `if` statement.

In [ ]:
```
counter = 0

while counter < 10 :
    print("We write INSIDE becuase we are inside the while loop.")
    counter = counter + 1
else:
    print("We write END because we have exited the while loop and are i
```

Try writing a while loop that exits if a random number is positive. How would you do that?

In [ ]:
```

```