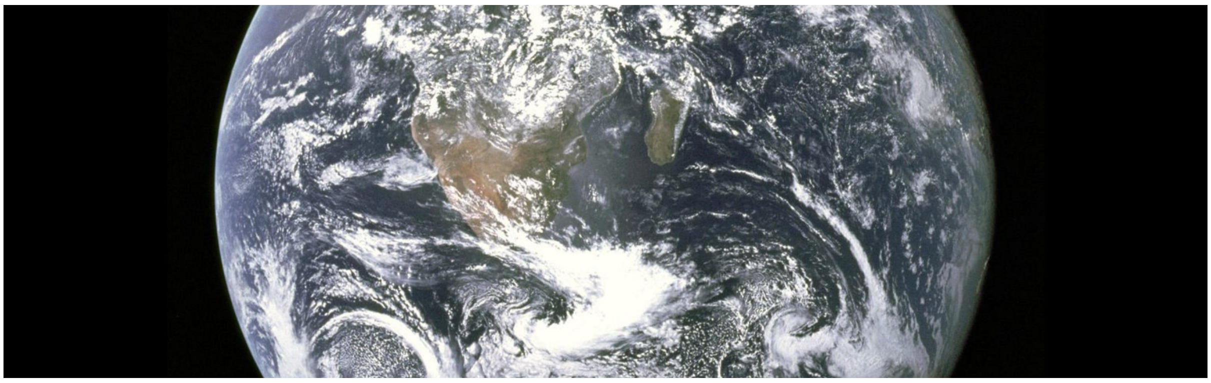

4. DataFrames In Depth

Plan each word out in advance.

dataframe.max(numeric_only)

df.describe(include=[])

converge to loc in all examples!

Did changing astype to float affect size?

introduce datadictionary at the beginning

https://www.kaggle.com/mauryashubham/english-premier-league-players-dataset

https://www.dataquest.io/blog/settingwithcopywarning/

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.ge.html

https://stackoverflow.com/questions/46210678/whats-the-difference-between-transform-vs-applymap-for-pandas-dataframe

▼ **Lecture 0** - Section Intro [**2:23**]

Welcome to our second section dedicated to the pandas dataframe, where we will take a deeper dive into the main data structure in pandas. This is the longest section in the course and could probably be a course on its own. But don't let that discourage you. We will learn a lot and familiarize ourselves more advanced pandas.

In the first part of the section, we will take a closer look at advanced indexing using binary operators and multiple boolean conditions, and also explore some computer science concepts around bitwise operators and the way modern computers represent numbers.

In the section part, we'll discuss sorting, looking up values, as well as more advanced methods to reorder our records according to customized logic that stands distinct from the typical sorting in ascending or descending order.

In the third part, we'll shift our attention to pruning operations. Mostly related to duplicate values, NAs, but also general dataframe reshaping, for example removing rows, columns, or even completely reindexing our dataset. The reality for most data scientists is that most data will require significant transformation before it's ready for analysis and further modelling. And in this part, we'll take a very close look at most of those operations.

In the fourth part, we will cover several dataframe transformation techniques; we are going to explore same-shape as well as aggregation-based approaches to transforming our dataset according to highly customized functions that we define as well as built-in pandas, numpy, and python functions.

Finally, in the fifth part of this section, we will delve into some advanced considerations on dataframe views compared to copies, we'll see how dataframes are stored in memory as well as what that implies for how we interact with our data.

Throughout this section we will get plenty of hands-on practice with skill challenges followed by detailed solution videos.

I'm very excited to welcome you to this long, but absolutely foundational, part of the course, and I'll see you in the very next lecture.

▼ **Lecture 1** - Introducing A New Dataset [**3:56**]

The EPL is one of the most prestigious and popular club soccer leagues in Europe and the World.

I mean even if sports is not really your thing, think of the structure of this dataset as a list of employees with various attributes. At the end of the day, all these players are just very famous employees.

So we will work with data in a tabular form that is very similarly structured, even though the content might differ. If on the other hands you like soccer, all the better.

`players = pd.read_csv(data_epl)`

First things first, let's take a look at the structure of our dataframe using the info() method.

`players.info()`

Okay, we see we have 17 columns, 9 of which are ints, 3 floats, and 5 object (strings).

▼ **Lecture 2** - Quick Review: Indexing With Boolean Masks [**3:42**]

We've kind of talked and discussed indexing with boolean masks before, but it's time to explore some new combinations using methods that we are already familiar with.

Remember, the key concept when using boolean masks is to generate a sequence of Trues and Falses, where True indicates that we want the record back. And, as we have seen, this concept is the same for indexing both series and dataframes.

So let's start with a question. What are the players that have a market value of greater than 40 million? This is a great opportunity to use boolean indexing.

Our first step will be to generate a sequence of booleans. And I'll do that using an expression that compares the the player market value series to the integer 40

`players.market_value > 40`

Okay, that was easy.

So we now have a pandas series of booleans which has a length of 461. Notice that this matches the number of rows, or the index length, for our main dataframe. If you recall, this is a very important point. The length of the boolean mask should always match the length of the axis we're indexing.

Excellent, now what do we do with this series of booleans? We simply pass it to square brackets or the loc indexer to extract from the main dataframe.

`players.loc[players.market_value>40]`

So these are the players that have a market value of 40m or more. There's what? 12 of them?

`players[players.market_value>40].shape`

That's great. In this short lecture, we reviewed boolean indexing; we generated a boolean sequence using a comparison expression, and then passing to square brackets.

There are a several other powerful methods which enable us to create boolean sequences quickly in pandas, and they are quite useful to boolean masking.

We'll explore them in the next couple of lectures.

▼ **Lecture 3** - More Approaches To Boolean Masking [**10:27**]

Let's look at a couple of other methods that allow us to generate boolean series on the fly.

Say we're interested in looking at all the defenders.

Um, in soccer as in other sports, there are position which are identified by some semi-technical terms that effectively describe where the player is located in the field and what role they play. Let me give you a quick visualization, which I found o

https://www.vocabulary.cl/pictures/football-soccer-positions-pitch.jpg

So this portrays what's probably the most common formation in soccer. It's called the 4-4-2.

Because we have four defenders, four midfielders, and 2 forwards. Yeah, somehow the goalie doesn't make it in the formation description because I guess there's always a goalie. Managers could change the formation around..go from 4-4-2 to 5-3-2 but there's always a goalie.

Cool, so say we want to look at the subset of our dataset that pertains to defenders only. Back to our notebook, let's first see how many different positions we're looking at

players.position.unique()

Okay, we have 13 (add .size then remove). Out of all these, we're interested in the defenders. That would be the CB, RB, LB. So we want to extract from our dataframe on the condition that the player's position is one of these values. This is a great use case for the series isin() method

players.position.isin(['CB', 'RB', 'LB'])

Beautiful, so we now have a pandas series of booleans which indicates whether each player in the dataframe is a defender, and now we could simply pass this to the main dataframe using square brackets or the loc indexer

players.loc[players.position.isin(['CB', 'RB', 'LB'])]

That's great. Looks like our dataframe has 154 defenders. So whenever we're looking to express a membership condition, as we did above where we're checking whether a given value is contained in collection of values, whenever we have such cases the isin() method is a great choice.

Okay, what else?

Let's look at ranges of values. We saw in the previous lecture how to extract players with a market value of greater than 40, but what if instead we're interested in a range of values, for example a market value from 40 to 50 but not more or less.

The series between() method would be a really good choice here… I'll start with an exclusive range

players.market_value.between(40, 50, False)

Notice, how we get a series of length 461, containing only booleans. Remember the length is the same as our index length. I know I keep repeating this but it's a very important concept when using boolean masking. The length of the sequence of booleans should match the length of the axis being indexed.

And now to do the actual indexing or selection, we simply pass this series within square brackets or loc

players.loc[players.market_value.between(40, 50, False)]

So there appear to be only three players with a market value greater than 40 and less than 50 million.

Let's convert this to an inclusive range by setting the inclusive parameter to True, and we now get some more players. It appears that there's quite a lot of players with a market value of exactly 40 or 50.

Nice. What else? What are some other ways of creating boolean sequences on the fly?

Well, there's comparison operators. For example, if we're interested in players that are 25 years of age or younger, we could start with generating our boolean mask. In this case I would say

players.age<=25

But in pandas we have an alternative to using these arithmetic operators, in that we could also rely on comparator wrapper methods. For our example here, instead of ≤ we could say

players.age.le(25)

This is simply a convenience method; there's really no differences between these two, expect for the fact that using the comparator methods we have the option of fillining in NAs. But it's really not a big deal. For the most part, I would consider these two as equivalent.

To see that, we could say:

players.age.le(25).equals(players.age<=25)

And by the way there are several other such methods, in fact there's one for each comparison operator

slide switch

Okay. In this lecture we saw a couple of other series methods that help us generate sequences of booleans on the fly. We explored the isin method, the between method, and lastly the comparator methods.

But through all this, we've been practicing dataframe indexing using single conditions. What if we wanted to combine several of them? Say, players 25 or younger *and* having a market value of between 40 and 50 million. Let's start tackling conditions like this in the next lecture.

▼ **Lecture 4** - Binary Operators With Booleans [**10:29**]

Before we get into combining multiple conditions, I thought it'd be really useful to get extra clear about how booleans combine. Remember, for the indexing we've done thus far, our first step has always been to create a series of booleans. So maybe before we jump headfirst into working with multiple series of booleans, I thought it'd be beneficial to explore how they behave in isolation. So let's do that.

First, let's talk about binary operators. Binary operators are just like other operators, but they work on the binary representation of the value, on the individual bits. So they allow us to do operations, comparison, complements on a bit by bit basis, that's why they're also known as bitwise operators.

The two that we will find the most helpful when working with pandas are the binary OR and AND operators. And these behave just like we would expect. Let's start with or, which is represented by this vertical bar notation. Say we have a true and false.

True or false is True. False or True is also True. False or False is False. True or True is True. Think of the the or operator as searching for true... here we have it, here we have it, here we don't...

Hopefully no surprises here.

The next really useful operator is the AND binary operator, represented by the &. Let's take a look at this.

(4 examples)

Think of the AND operator as always true unless there is a false. A single false is enough to trigger a false. We could have a lot true's, but that single false will result in the entire expression being false, even if the false is preceded and followed by several Trues.

Good. And obviously we could use both of these operators as part of one expression. For instance,

(example)

Okay, I hope a lot of this is review and by now pretty clear. Let's bring it up a notch and start looking at how pandas series of booleans combine. I'll start by creating two series of booleans.

(t containing True, and f containing False) (so t and f)

And now, let's combine them using a binary operator.

t&f

What we see is that series of booleans combine just like the booleans themselves. A series containing a true when combined using the binary and operator with a series containing a false produces a new series containing a false. And the or operator behaves exactly the same way.

Maybe this is a bit underwhelming. This is nothing impressive. Where it gets really powerful though, is when we a have long sequence of booleans

Let's convert our series into longer ranges of boolean. One containing 10 falses, and the other containing 10 items of alternating trues and falses.

(long sequence)

And now lets go ahead and combine them

t & f ... t | f

We notice that the exact same dynamics apply. So maybe still not that impressive.

However, the important thing to note here is that when we combine the two, the combination is done label-to-label, not based on order. So let's create another example, this time with labels

(example with labels)

Take a closer look at what happened here. t & f returns True for the first element, when it's combining the two series, one of which has False under the first element. So why is that?

The reason for this is that the order of the elements is not important at all. What's important is the label that the value is associated with. So a is combined with a, b with b, and c with c, which is why we get a true value.

I wanted to demonstrate this because it'll really help illustrate what happens behind the scenes when we combine conditions, expressed as boolean series, to index our dataframe. We will discuss this in a couple of lectures. But before we get to that, in the next bonus lecture, we'll talk talk about about two other binary operators which are less widely used, but I find pretty good interesting to know. As always with bonus lectures, this is something I would recommend you watch but it's not something that detracts from your progress in the rest of the course.

▼ **Lecture 5** - BONUS - XOR and Complement Binary Ops [**12:57**]

In the previous lecture we discussed the binary or and and operators, and without doubt those two are the ones you will see most frequently in pandas and elsewhere. But there are several other binary ops, and in this bonus lecture we will cover another pair.

Let's start here with XOR, which stands for exclusive OR. How is this different from or? Well, as the name suggests, it's exclusive. In plain language, it is true only when the inputs are different, and false when they are similar or alike.

Maybe it's easier to understand this using some examples. The XOR binary operator in python is represented using the ^ symbol, also known as the circumflex, and so

True ^ False

This is True because the inputs are different.

False ^ False

is false because the inputs are alike, and also

True ^ True

is false because the inputs are alike as well. So that's XOR. And, certainly we could combine this with other operators. For example,

True ^ (False | False & True) | False

If you want, pause the video and think about what this would return.

Let's quickly go through this starting with the expression within the parentheses... explain

Good. We could use XOR to combine boolean series in dataframe indexing, but XOR is also very practically useful in computer science. For example, XOR gates are used to implement binary addition in computers. Now, I love to get into details whenever possible but this would perhaps be too much of a detour to go over, so instead let's talk about another binary operator, and that's the complement, or two's complement.

In python it is represented using the tilde (~) symbol, and in pandas and numpy you will find it extremely useful in negating boolean series, or in other words finding the opposite

Actually, when applied to plain booleans in python, this might give what appear to be surprising results. For example,

~True gives us -2

~False gives us -1

Why is this? In what world does this make sense?

I don't wanna go too much off topic here but this has to do with how computers represent integers. Specifically, the two's complement system of representing integers, which is widely popular, in fact I think that's an understatement; this is how almost all modern computers represent integers. Think of it as a scheme or system for deriving a binary representation of integer numbers.

slide switch

In two's complement, zero is represented using all 0s, and then for positive numbers we start counting up. So here we have our own little two's complement number representation using 8-bits. By the way, using 8 bits we could represent 256 numbers in total using this system, so that would be something like negative 127 to positive 128.

But in here, we're only showing 6 numbers to save space and focus on the key concept.

Great. So if this is the number system we use to represent integers in a computer that only understands bits.

In addition to this representation scheme, two's complement also refers to the operator by the same nam, the one represented by the tilde.

Think of the two's complement operator, or the tilde, as negating all the bits. This is also known as the binary complement.

notebook switch

So at this point, you may be wondering "how is this related to ~True" being equal to negative two? Remember, in python booleans are integers so ~True is the same as saying tilde positive integer 1.

And what does the tilde operator do? it inverts the bits representing the number 1.

Okay, so now that we know how the number 1 is represented let's go back and invert those bits.

slide switch

We're starting with one, which is represented in 8bit binary as 00000001. And next we invert all the bits, so we end up with 11111110. And that in our scheme corresponds to -2. And that my friend is why ~True returns -2.

How about ~False? Well, that's the same as ~0, which is the same as inverting the binary representation of zero. So we go from 8 zeros to 8 ones, and in our scheme 8 ones correspond to negative one, which is exactly what we get here.

https://stackoverflow.com/questions/1049722/what-is-2s-complement

That's pretty awesome.

Okay, let's move back to pandas. In pandas we will use the complement binary operator to negate our boolean conditions. This operator will turn our Trues into Falses, and vice versa. Let's take a look at some examples. If we had a series consisting of 3 booleans... and we negated it, the result would be a series that has the exactly opposite boolean. This feature will be very useful in defining negative conditions.

example then negate

Excellent, this has actually been a pretty intense lecture. We talked about the xor and complement binary operators and then we went over the binary representation of numbers to really understand some surprising results that we were getting.

I'm sorry if you found this a bit too much. At the end of the day, this is a pandas course, but I do think it's important to go over computer science and other concepts as much as possible if they help improve our understanding.

Before moving on, let's take a second to summarize the four binary operators we discussed in the last two lectures.

slide switch

▼ **Lecture 6** - Combining Conditions [**7:52**]

Okay, it's time to look at indexing using multiple conditions. So far, we've used pandas series of booleans to do our indexing. For a quick refresher, if we want to look at all left back players (that's defenders that play on the left side of the field), we would first create a series a of booleans like so

`players.position == 'LB'`

and then use that in square bracket indexing. So far so good.

Now what if we wanted to restrict by another condition and do this multi-condition selection, if you will, in one go. Well, to do that, we simply combine several boolean series using the binary operators we covered in the previous two lectures.

For example, if we want to only look at left backs who are 25 or younger, perhaps we could go straight inside the square brackets and add a new condition using the binary & operator

`players[players.position == 'LB' & players.age <= 25]`

Now, there is gotcha here, something we have to watch out for. If we run this the way it is, we will run into a strange TypeError, and the reason is the way pandas interprets this. It thinks we're asking it to combine the string 'LB' with the boolean array from players ages. To avoid this problem, I would always suggest wrapping our conditions in parentheses.

`players[(players.position == 'LB') & (players.age <= 25)]`

In this case the expressions within the parentheses are first evaluated, each producing a sequence of booleans, and only after are the two resulting boolean series combined.

That's great. So we now have indexed our dataframe for the subset of players that are left backs and 25 years old or younger.

To improve the readability of this code a bit, we could split this into multiple lines

```
players[
    (players.position == 'LB') &
    (players.age <= 25)
]
```

And, it goes without saying, that we could add even more conditions. For instance, next let's filter for those players that have a market value of at least 10 million

add (players.market_value >= 10)

That's nice. And we could also certainly mix and combine binary operators. So if we're not interested in Arsenal or Tottenham results at all, we could add a fourth condition that specifically excludes those clubs.

add ~(players.club.isin(['Tottenham', 'Arsenal']))

Excellent. This is how easily we could combine boolean masks to do some pretty powerful dataframe indexing.

If like some people you don't like the look of this, if you feel it's getting a bit out of hand with all of these conditions in here, there's always the option of taking them out of the square brackets and assigning them to specific variables.

We'll talk about that quick refactor in the next lecture.

▼ **Lecture 7** - Conditions As Variables [**4:44**]

We ended the previous lecture with a multiline expression that several people think is quite difficult to read. One suggestion I have to avoid this being case is to always consider refactoring your conditions into standalone variables, especially when you need to compose an indexing logic that involves several of them.

Let's take a look at that. And hey, maybe let's use this as an opportunity to get some more practice and think up some other conditions.

Let's say we're now targetting Arsenal right backs or Chelsea goalkeepers. We'll start with the first condition, that the player comes from the Arsenal club.

players.club == 'Arsenal'

arsenal_player = players.club == 'Arsenal'

So this generates a series of booleans as we've seen. And let's now assign the resulting series to a variable. Notice that we're using two different operators here in the same line. The equals operator is the operator that handles the assignment; it's called, surprise surprise, an **assignment operator**. The expression on the right is first evaluated and then the result is assigned to the variable on the left. The equals-equals operator is a **comparison operator**. What it does is it compares each value in the series to the string 'Arsenal' returning True when it sees a match, and False otherwise.

Next, let's work on our right back condition.

right_back = players.position == 'RB'

For the Chelsea part of the question, let's do it a bit differently by combining both conditions and assigning the resulting output to one variable.

chelsea_and_GK = (players.club == 'Chelsea') & (players.position == 'GK')

In the end, all that remains to index the dataframe is for us to pass these conditions within square brackets or the loc indexer.

players.loc[(arsenal_player & right_back) | chelsea_and_GK]

And there we have it. In summary, to avoid long, hard-to-read combinations of indexing criteria, we could always refactor the conditions to some more basic primitives and then combine those using binary operators to extract from the dataframe. Excellent.

▼ **Lecture 8** - Skill Challenge [**1:07**]

Alright, so let's put all of this to practice now. Our goal in this exercise is to identify the subset of players that have these characteristics in common:

- they're English, and
- their market value is more than twice the average market value in the league, and
- they either have more than 4000 page views or they are a new signing, but not both

Take a moment to think this through. Have a go at it and in the next lecture we will tackle it together.

▼ **Lecture 9** - Solution [**6:57**]

Alright, so let's get going here. I'll begin by outputting the first couple of records from our dataframe to have as a reference.

players.head()

And then I'll go ahead and create a variable for each of these conditions. Let's take a look here, so starting with the first, we have to check that the player's nationality is English using the comparative equals operator.

english = players.nationality == 'England'

Then for the second condition we have to compare the players' market value to the average market values of the entire dataset, and specifically we have to ensure that the players market value exceeds the average by more than twice. Hm, that's a bit tricky. So let's go through this carefully.

above_average = players.market_value > players.market_value.mean() * 2

Now for the third condition we have (read it out). Notice how this starts like a regular OR condition but then it says that either one of the other should be true, but not both. Whenever we see language this it's a great use case for the XOR operator. So let's use that.

popular_xor_new = (players.page_views > 4000) ^ (players.new_signing == 1)

At this point, we've isolated our conditions in separate variables. And remember these variables point to three separate boolean series. So even though they capture different logic.. for example, one is checking for nationality, the other for market value being over the mean, and the third is xoring page views with new signing. Even though they're so diverse in terms of what player aspects they are restricting, all of them have two characteristics in common:

first, they're all boolean series... so a sequence of Trues and Falses

.head() one of them

second, they're all equal in length to the dataframe index. Remember this is a critical requirement for boolean indexing

print the shape of all including index... so they're series becase ndim is 1, there is only one dimension, and that is equal to the index length

Now that we have our logic condensed in three boolean series, all that remains is to combine these series and index from our dataframe. Before we do that we have to confirm how these conditions combine with each other, and I believe it's all and (scroll up and highlight condition1 *and* condition2 *and* condition3). Fair enough, so we'll put them together using the &.

players.loc[english & above_average & popular_xor_new]

Beautiful.

▼ **Lecture 10** - 2d Indexing [**10:00**]

So far we've discussed boolean indexing in the context of extracting rows. Right? Our indexing has only operated on the index axis, on axis 0. This is the default behaviour in pandas, but obviously when working with dataframes we're working with a two dimensional data structure, so we certainly have the option of indexing along the other axis too. In this lecture we'll take a look at doing that.

First, let's create a condition for our example here. I'll go with Chelsea youngsters

chelsea_23under = (players.club == 'Chelsea') & (players.age <= 23)

And I'll pass that to square brackets and we get chelsea players who are 23 or younger. This should hopefully be something that by now we are pretty familiar with

players[chelsea_23under]

But notice how all the columns in this dataframe slice are being selected. What if instead we were interested in only the position or position and market_value categories. How would we do that selection at once?

There's a couple of ways to go about this. First, we convert this from square brackets to the loc indexer, and notice that it continues to work exactly the same way, the reason being that loc and plain square brackets operate identically when it comes to indexing rows using boolean arrays.

And now that we are working with the loc indexer, we could add our column condition as the second parameter. I'll go right in, add a comma, and say

players.loc[chelsea_23under, ['position', 'market_value']]

And there we have it. Honestly, this is the most intuitive way to go about it. But it's not the only one. If our conditions get messy or complicated.. passing in a python list of labels like we're doing here won't work.

For example, if we wanted all the columns that begin with the letter 'p'.. yeah that's not the kind of condition we could key in like this. To do something like that, we would need to create a separate variable to hold a pandas series of booleans that check for this condition along the column axis.

In this case, to select all the columns that start with the letter 'p' we could use a string method which we will discuss later in more detail as well, but for now notice that all it does is it returns True when the string begins with the letter p and False otherwise.

p_cols = players.columns.str.startswith('p')

Finally, we pass this boolean series as the second attribute in our loc indexer. And there we go, we have all the columns that begin with p, for the chelsea players who are younger than 23.

In the previous lecture we noted that to do dataframe selection along the index axis the length of the boolean series needed to be the same as the length of index itself. And sure enough that's the case for our chelsea_23under condition.

print(chelsea_23under.shape)

print(players.shape)

Similar to this, to do selection along the column axis like we are doing here for columns that begin with p using the p_cols variable. To effectively do such horizontal indexing, we have to make sure that the length of that boolean series matches the length of the column axis for the underlying dataframe.

From the dataframe shape we have here, we note that that's 17, so let's confirm for p_cols... and sure enough, it is

print(p_cols.shape)

That's great.

One last point we should be aware of here is that there is an alternative way of indexing along the second dimension, that is pretty common in existing codebases, and that is to chain two square brackets like so.

The idea being that the first selects a dataframe slice containing all the columns, and the second condition selects a single column from that.

players[chelsea_23under]['position']

Now, this may appear to be equivalent to

players.loc[chelsea_23under, 'position']

But in reality what happens behind the scenes ensures that it will always be slower. And that has to do with the fact the the dunder **get_item** method gets called twice in this first example, and only once in the second. Just something to keep in mind. When you have a choice between these two syntaxes, please prefer this second one here.

▼ **Lecture 11** - Fancy Indexing With lookup() [**8:29**]

start with slide

Throughout this section and the previous ones we have seen how flexible indexing is in pandas. We've seen that using square brackets alone we could slice the dataframe, select from it using boolean masks, even get a lot more specific about exactly what labels from each axis we want back.

This fourth approach here, which is something we have have used several times before, including in the previous lecture, is also known as fancy indexing.

Now, the name is fancy, but it simply refers to passing multiple labels all at once. In other words, fancy indexing is very similar to basic indexing but instead of using single labels, we specify a list or tuple of them. So this is basic, this is fancy.

notebook switch

In this lecture we'll talk about a pandas method that is dedicated to doing fancy indexing item selection. It will allow us to step it up a notch.

next slide

The method is called lookup(). It takes specific row and column labels and returns the values for each corresponding pair of labels.

So in this example here, we are looking up the value associated with the index label 450 and the column label 'age'. And we get back an array containing the value in those label coordinates, so 30.

Okay, now that we have the basic concept down, let's take a closer look at this method.

notebook switch

So we have here a traditional label based extraction with loc... and we're doing fancy indexing by passing a list of index labels and a tuple of column labels.

players.loc[[0, 132], ('name', 'market_value')] # fancy

The output we get back is a dataframe slice. Now, let's compare this with what we would get from a lookup method using the same labels

players.lookup([0,132], ('name', 'market_value'))

Hm, so we have an array of two items, containing only the values that correspond to those row and column labels specified. Again, think of these as label coordinates. We specify a row, we specify a column, and for that pair of labels we get back a single value.

explain how the values correspond

What would happen if we swapped the column coordinates around? Well, we would get market value for the player with the index label 0 and name for the player with label 132.

show the swap

Okay, good.

In reality though, the lookup method is most useful when we already have some collection of labels that we want to select from the dataframe.

To demonstrate that, let's say that we have a list of player names and for each we want to source a set of attributes.

names = ['Petr Cech', 'Mesut Ozil', 'Alexis Sanchez']

attributes = ['age', 'market_value', 'page_views']

So our goal here is to select peter cech's age, ozil's market value, and sanchez page_views.

Okay. Let's try the obvious

players.lookup(names, attributes)

Hm... this gives us a key error, starting with the first label in our series. Because guess what, our current index labels are not player names but rather the default rangeindex.

players.head()

So what we need to do before the lookup is change the index of our dataframe to player names... and this does not need to be done in place. We could simply do it on the fly for the purpose of this lookup

players.set_index('name').lookup(names, attributes)

And sure enough we now get the exact details we were looking for.

So that's lookup. Think of it as a more direct and performant way to do fancy indexing at scale. It gives us the flexibility to select the precise values in a dataframe using a collection of row and column labels, kind of like hunting down values using their label coordinates.

▼ **Lecture 12** - Sorting By Index Or Column [**6:59**]

In a previous section we explored the sort_values method which helped us sort our row labels based on different columns that we specified in the by parameter.

For a quick refresher, if we want to sort the players by market value in descending order, we simply say

players.sort_values(by='market_value', ascending=False)

That is great. But this sorting by values is not the only way to sort. We occasionally also find the need to sort by index. And this is usually the case for indices that we create, not the pandas default ones.

For instance, our players dataframe currently has this integer based index. It's this immutable rangeindex object as we've seen before. So there's nothing exciting here really. But say we promote our player names to the be the index for this dataframe. As we know, to do that, we simply say... and I'll set the inplace parameter to true so that the changes affect the underlying dataframe

players.set_index('name', inplace=True)

Okay. Now our dataframe has player names acting as the index.

players.index

We now have this brand new index, it's a pandas Index object, consisting of player names, and because it contains strings it has an object dtype.

That's great, but notice that our index is not ordered anymore. Right? These names are all over the place. A good practice when working with tabular data, especially in the context of databases, is to always maintain an ordered index. Fortunately in pandas it's pretty quick and easy to fix this by saying

players.sort_index().head(10)

And there we have it: a copy of our dataframe, now sorted by the index values, in this case player names. If we want the changes to stick, we set the inplace parameter to true so that the underlying dataframe is modified.

This exact method also helps us sort our columns, which is something we haven't discussed yet. Let's take a look that.

Let's suppose that in addition to ordering our index, we also want to alphabetically order our columns too. Turns out we could also do that using the sort_index() method. The only thing we need to change is the axis parameter. We say

players.sort_index(axis=1)

And we're good to go. We've seen how to sort the dataframe along both axes, and we've done this inplace. Before moving on, I'll reset index so that if you haven't been following along, which I definitely recommend you always do, but if you haven't, I want us to both start the next lecture fresh from the same base dataframe.

So I will use the reset index to demote the player names from acting as the dataframe index

players.reset_index()

What this method does is it takes the current index and it "downgrades" it to be a dataframe column.

Funnily enough, if we run it again, we will see this default rangeindex added as a new column and replaced with a brand new rangeindex. Let's not do that though, it's unnecessary clutter. Let's only reset it once to remove the player names from the index, and let's do it in place this time.

Great. See you in the next one.

▼ **Lecture 13** - Sorting vs. Reordering [**12:29**]

We've talked a lot about sorting, and we've already sorted by values, index and even columns. But the sorting we've seen has been a bit restrictive. I mean, we had plenty of options at our disposal. Both the sort_values() and sort_index() methods contain several parameters that enable us to customize the specifics of how we want the sort to be carried out

walk through docs showing how many paramters are available

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_values.html#pandas.DataFrame.sort_values

But at the end of the day there are only two ways to fundamentally sort using these methods. We either go in ascending or alphabetical order, or descending or reverse alphabetical order. What if we wanted to more precisely reorder the rows or columns in a dataframe… according to some very specific order that we specify

For that, we could use the reindex() method. And it works like this.

I'm going to begin by isolating a chunk of our dataframe. I'll use the location based indexer, for a change, iloc, and for our example here I'll select the first four rows and columns.

players_lite = players.iloc[:4, :4]

Now, say we wanted to precisely order the rows here according to some highly specific requirement.

Suppose we're working for a soccer scout he or she is targeting these four players and wants the relevant data to be reflected in a very specific form, because she's picky

type requirement

There's actually no way to achieve this using sort_values, sort_index or a combination of the two. But it turns out it's quite easy with reindex. We only need to say:

players_lite.reindex(index=[2, 1, 3, 0], columns=['age', 'name', 'position', 'club'])

And that's it! This is a nifty little method that's actually quite powerful for specific cases like this when we need to do some precision reordering along either axis. Here we're doing it for both the index and columns, but that's not necessary, we could drop one or the other.

Technically speaking, we're not reordering our players_lite dataframe. In fact, we're getting a copy of it that includes the index and column labels we specify. For this reason, it's not even necessary for reindex to have the full range of labels, like we're doing in our little dataframe.. where we're using all four.

In fact, because of that, we could craft this precise dataframe by applying reindex on our main players dataframe directly

```
players.reindex(index=[2, 1, 3, 0])
```

Right? Notice that Pandas is not complaining that we're only specifying 4 of the 461 index labels.

Also note that because we didn't provide any column labels, we got everything back. So let's change that in order to meet our requirement above. We'll say

```
players.reindex(index=[2, 1, 3, 0], columns=['age', 'name', 'position', 'club'])
```

Okay, nice.

What if our soccer scout boss changes her mind and instead of these 4 columns, she's now asking for all the data on these players. And she wants it organized in alphabetical order.

We have a couple of options. First, we need to remove the columns parameter we have here, so that we get back everything.

```
players.reindex(index=[2, 1, 3, 0])
```

Now, we have all the columns, but they're not ordered. To quickly fix that we could simply chain on a sort_index() with axis set to 1.

```
players.reindex(index=[2, 1, 3, 0]).sort_index(axis=1)
```

And that would do the job quite nicely.

By the way, we could also avoid method chaining in this case, and rely only on the reindex() method. Another way to achieve exactly the same thing with reindex is to pass to the columns parameter a list of all column labels in alphabetical order.

But we don't wanna sit here and type out all the column names in order...age, age_cat, big_club, club. So a quick alternative would be to pass a sorted python list of column names.

Let's take a very quick aside here to demonstrate how we're going to do this.

```
# === ASIDE ===
```

To access the columns we say:

```
players.columns
```

This is a pandas Index object, yes. But it is also an iterable. We could verify that by calling iter() on this object, and we get the reference in memory. But the fact that we get a reference indicates that it is an iterable.

If instead we tried something like True, we would get an exception thrown, because booleans are not iterable. The point here is that the pandas Index object is an iterable.

And there is a pretty convenient method in python that returns a new sorted list from an iterable, and clearly enough, it's just called sorted(). I'll link the documentation if you want to read more about it, but if we simply pass our players column to sorted() we get back a python list of alphabetically sorted columns labels.

That's really great, because it's exactly what we need for our columns parameter in the reindex method. So we'll end the aside here

```
# === END ASIDE ===
```

..and I'll go back and say:

```
players.reindex(index=[2, 1, 3, 0], columns=sorted(players.columns))
```

So yeah, in pandas, as in programming in general there is more than one way to go about things.

Now typically, I've seen a lot of people use reindex for precision reordering of columns and rows, which is why I wanted to demonstrate that use case first. But it's worth noting that it technically could be used to carve out a slice of the dataframe containing any set of index and column labels in the order that we specify. In other words, not necessarily a full reordering of the dataset.

We could go in and slice our sorted columns to only get back the first 6, for example, alongside 4 rows. That's totally fine with reindex. In fact, we see how how it could serve as an alternative way to select or build very specific dataframe slices. I gotta warn you though that it's not the most effective way to do selection.

```
players.reindex(index=[2, 1, 3, 0], columns=sorted(players.columns)[:6])
```

Lastly, and perhaps most importantly, this method is very useful when working with dataframes containing hierarchical indices, which is a very exciting topic that we will explore in a future section.

▼ **Lecture 14** - BONUS: Another Way [**2:13**]

players.reindex(index=[2,1,3,0], columns=sorted(players.columns))

sorted(players.columns)

players.columns.sort_values()

players.reindex(index=[2,1,3,0], columns=players.columns.sort_values())

▼ **Lecture 15** - BONUS: Please Avoid Sorting Like This [**3:37**]

I wanted to do a quick video on something that I've seen more times than I wish was the case and that I think is definitely an anti-pattern, something you should try to avoid.

It has to do with column sorting. So let's take a new slice here for a quick example, and I'll assign this to a temporary variable.

```
df = players.iloc[:6, :6]
```

```
df
```

Say we're now back to the question of "how do we sort the columns alphabetically"?

What I've seen some people do is they change the shape of the dataframe, by specifically transposing it. In pandas this could be done using the swapaxes function like so

```
df.swapaxes(1, 0) # swap axis one (columns) with axis zero (rows)
```

...there's an even easier way which is to use the transpose attribute.

```
df.T
```

Remember, we're trying to sort the columns. But after the transpose, the columns become our row, so now we could simply use the sort_index()

```
df.T.sort_index()
```

Okay, and now that our index is sorted we transpose back once again

```
df.T.sort_index().T
```

And we end up with our original dataframe, but now with alphabetically sorted columns.

I'll admit, this is kind of creative but please don't use it. First, it's kind of difficult to read and reason about. We're transposing twice, sorting once, it's kind of a mess. Second, it's not really performant because it's so indirect and it involves a lot of temporary output. Third, there's really no reason to rely on this approach when we could achieve the same thing as easily as

```
df.sort_index(axis=1)
```

And if you didn't know that this was a valid parameter, you could always do a quick check on the pandas documentation. Supported params are listed up top following a brief description of the method.

That's really it for this one. Part of being a good data analyst or developer is also knowing what not to do. And, in pandas, this is definitely one of those things we should avoid.

▼ **Lecture 16** - Skill Challenge [**1:16**]

1.

Sort the players in the *players* dataframe by age in ascending order. Who is the youngest footballer in the EPL?

2.

Set the *club* column as the index of the dataframe. Then sort the dataframe index in alphabetical order. Make sure these changes are applied to the underlying dataframe and carry over to the next question.

3.

Sort the dataframe values by *club* and *market_value* where the club is alphabetical (Arsenal first) and the mark

players.set_index('club').sort_values(by=['club','market_value'], ascending=[True, False])

▼ **Lecture 17** - Solution [**4:03**]

1. Okay, the straightforward way to answer this first question is to simply use the sort_values() method, pass age as an argument to the by parameter, and set ascending to True. This sorts the entire dataframe from where we could immediately see who the youngest player is. It's simply the first record.

   players.sort_values(by=['age'], ascending=True)

   By the way, we could also find the youngest player without sorting the dataframe by combining the idxmin() method with an indexer

   players.iloc[players.age.idxmin()]

2. Fair enough. For this second part, I'm going to use the set_index method to specify club as the index. And then I will sort the index of the resulting dataframe using the sort_index() method. That's very readable, we set, and then we sort.

   players.set_index('club').sort_index(inplace=True)

3. Now this is a bit tricky. First because we are sorting by two values, and second because the sorting direction is opposite. We want clubs in alphabetical order but the players within each club in descending order by market value.

The first part is relatively easy. Instead of passing a single string, we pass a list of column labels. Now for the second part, we know at first blush that this ascending parameter is involved. But whereas typically we set it to either True or False, in this case, neither is strictly correct because we have to specify different directions for each of our columns. So to accommodate this requirement we now have to pass in a list of booleans.

players.sort_values(by=['club', 'market_value'], ascending=[True, False])

▼ **Lecture 18** - Identifying Dupes [**10:40**]

Very often in the real world we work with data that is not perfect. In fact this is most often the case, and it's part of the reason why tools like pandas are so prominent and useful.

Our players dataframe for example contains some duplicate records, players that have the exact some values across all column labels. The easy way to identify such records in pandas is to use the duplicated() method. We say

players.duplicated()

And this returns a boolean series, which we could then pass to a square bracket indexer to get the actual player records that are duplicated

players[players.duplicated()]

Okay, that's good for a start, but there's a couple of notions here which we could explore and play around with.

First is related to the question of what really constitutes a duplicate value. If we use the method the way we are in this example, we're saying that the values across all columns need to be the same for two records to be considered duplicates.

But we could also loosen up this restriction by specifying a subset of columns that should be considered. For example, say we want the combination of club, age, position, and market value to define a unique player.

In other words, from each club, we recognize as original entries only the unique age-position-market value combinations. All the other records that repeat these 4 fields will be considered duplicates. To specify this in code, we use the subset parameter of the duplicated method

players.duplicated(subset=['club', 'age', 'position', 'market_value'])

So here we are saying that if Arsenal for example has two 23 year old right backs with a market value of 20 million, one will be considered the original entry and the other will be be duplicate.

This is great, but it brings up another question: which one is the duplicate? Right? If we have several repeating values, how do we distinguish between the original and the duplicate?

By default, pandas will treat the first ordinal record as the original one, and any subsequent ones as duplicated. But this also could easily be changed using the keep parameter.

```
players[players.duplicated(subset=['club', 'age', 'position', 'market_value'], keep='last')].shape
```

Keep takes three values. The default is "first". And that means, treat the first item out of a set of repeating values as the original value, and everything that follows as duplicate.

Another option is "last", which is the exact opposite and means that, in a set of repeating values, the last will be considered as the original and everything preceding it will be marked as a duplicate.

And, interestingly, a third option is False, as in the boolean false. Setting the keep parameter to False indicates to pandas that we want all repeated values to be treated as duplicates. So both of those Arsenal right backs with a market value of 22 million will be marked as duplicates.

```
players[players.duplicated(subset=['club', 'age', 'position', 'market_value'], keep=False)].shape
```

If we rerun our method with keep set to False we see that not surprisingly the number of records which we have identified as duplicates has increased now from 7 to 13 now, and, again, that's because setting keep to False means that we want all repeated values to be classified as duplicates... which is why we see them show up as output in

So that's the duplicated method. You'll find it widely used in pandas.

▼ **Lecture 19** - Removing Duplicates [**6:13**]

Alright, let's go back to our original duplicated records.

```
players[players.duplicated()]
```

As discussed in the previous lecture, these records are identical across all columns.

Now what appears to be duplicated input is not always incorrect or bad data, but for certain contexts like ours here it is certainly misleading. There is no good reason to count the same player multiple times. For example, if we were to calculate the mean market value for all players in the league, we would be miscalculating the true mean by counting certain players more than once.

Let's take Alex here, who has a market value of 22M. Our dataset average is

```
players.market_value.mean()
```

just 11.13 or so. But this average counts Alex three time and Granit twice. Right? Remember the defaults in the duplicated method. When we call this method without any arguments, it's as if we have

`subset = None`, meaning compare all columns in the dataset; not just a subset, and

`keep = 'first'`, meaning treat the first record as the original one, and everything after it as duplicated.

Now, both Alex and Granit have been identified as duplicates. And they both have market values that are significantly above the mean, which means that they are biasing our dataset average upward.

In order to correct for this, and calculate the true league average market value we have to count these players only once. That is to say that we somehow need to exclude these duplicates.

One pretty useful method which follows a very similar function signature is the drop_duplicates() method. This returns a copy of the dataframe where the duplicated values have been removed.

Also, just like the duplicated() method the drop_duplicates() method has a keep parameter that defaults to 'first'.

```
players.drop_duplicates(keep='first')
```

Now, let's see the change in the dataset mean that comes from excluding these players. And I'll go ahead and do this in one go, by extracting market_value and then calculating the mean.

```
players.drop_duplicates(keep='first').market_value.mean()
```

So our mean is now 11.01. It dropped by .11 or about 1% or so. Not a huge change in this case, but you could imagine it being a lot worse if we have more duplicates or more extreme ones.

In summary, duplicated() and drop_duplicates() are two pretty useful methods to identify and exclude duplicated records from dataframes. It's worth noting, that whether a duplicated value is bad data or not, whether it's misleading or not, really depends on the nature of the data we're working with.

In our case, we are working with a dataset that includes all EPL players, and we identified several that appeared multiple times. For our specific context, counting the same people multiple times was not very meaningful.

But if we are working with other datasets that either because of their nature or sheer size are bound to have repeating values, for example a massive database of chess moves, census data, website activity patterns... it may be the case that duplicate records in those cases carry meaning, and therefore, it may be an analytical misstep to blindly drop all duplicates. Just something to be aware of. Duplicated values should not always be discarded. It all depends on context.

▼ **Lecture 20** - Removing DataFrame Rows [**2:58**]

In the previous lecture, we saw how to remove duplicated rows in one go using the drop_duplicates() method.

Another approach would be to first identify the records we want to remove and then exclude all or some of them separately. For example,

`players[players.duplicated()]`

The duplicated records are those in index labels 18, 19 and 154

Now, what if we want to remove 19 but keep the other two. Using drop_duplicates would not help here, but we could simply rely on the more generic drop() method. To drop the record with index label 19 we say...

`players.drop(labels=19... and importantly we specify... axis=0)`

Another way of achieving the same thing would be to use the index parameter instead

`players.drop(index=19)`

And if we wanted to remove several rows at once, we simply pass in a list of labels

`players.drop(index=[19, 20, 21, 231, 10])`

This method returns a copy of the dataframe, it doesn't modify the underlying data unless we set the inplace parameter to True.

So that's an alternative way to remove unwanted rows, whether they are duplicates or just something that does not belong in our dataset. We could always use the drop method, specifying the unwanted row labels.

▼ **Lecture 21** - BONUS - Removing Columns [**3:02**]

I just realized that in the previous lecture we only talked about using the drop() method to remove rows, but an equally valid use case is removing columns. And fortunately the syntax for that is nothing earth-shatteringly new. In fact take a second to guess how we would go about doing this using drop().

So, if we used this syntax to remove rows, how would we go about removing columns?

..okay, if you thought of changing the axis parameter from 0 to 1, to go from row to column, you'd be absolutely right!

`players.drop(labels=[19, 20], axis=1)`

Now, obviously this is going to fail because now that we switched to the column dimension, we have to provide column labels. So I'll say 'age' and 'market_value'

`players.drop(labels=['age', 'market_value'], axis=1)`

And there we have it. I think we've mentioned this before but an alternative to saying 0 and 1 is to simply type 'rows' or 'columns'. This is entirely up to personal taste. I have a preference for the 0 and 1 syntax, but a lot of folks prefer sticking to 'rows' or 'columns'.

Excellent. One last thing. Another way to drop these labels is to pass them to the columns parameter directly

`players.drop(columns=[19,20])`

So that was a quick bonus lecture on removing columns.

▼ **Lecture 22** - BONUS - Another Way: pop() [**4:13**]

Let's look at yet another way of removing columns and that is the pop() method. This is pretty easy to use; we only need to specify a column label that we want removed from the dataframe. I'll say club...

`players.pop('club')`

That's great. Three things to note about this method.

First, what we get back is not the dataframe with the column removed, as we did when we used the drop() method. Instead, when using pop() we get back the removed column itself. In the form of a series. Right? So that's one thing to note.

The second point is that pop() works on a single column at a time, which is to say that we could not use the pop() method to remove more than one column at once. Obviously if we keep calling the method with different column labels, we definitely end up removing several columns, but the point is that this needs to be done one column at a time, which is not very efficient.

The third point is that the method works on the underlying dataframe directly. In other words it's as if it always operates with inplace set to True. So if we pop a column, we're modifying the underlying dataframe to no longer include that column.

So always be careful when using pop(). I mean it's very cute and convenient but it also does have some side effects that we should be aware of. If you want to remove a column and capture the removed sequence of values for whatever use, this is the perfect method for that. But if you're planning on removing more than one column or you're exploring around and don't want the changes to affect the underlying dataframe yet, then I would advise using the drop method instead, as we discussed in the previous lecture. That's it for this one.

▼ **Lecture 23** - BONUS - A Sophisticated Alternative [**5:12**]

Okay, in this bonus lecture we'll mention another way to remove rows or columns, that takes quite a different approach.

We talked about drop_duplicates(), dropping rows or columns, or popping columns one by one. Another less popular, and perhaps "not that good of an alternative" is to reindex the dataframe by leaving out several unwanted columns.

Remember the reindex method? It crafts a new dataframe for us that only reflects the the labels we specify.

If we reindex the players dataframe without any arguments, we get back its copy

`players.reindex()`

If we reindex it looking for the rows labels 0, 3, and 9, we get back only those rows

`players.reindex(index=[0, 3, 9])`

Now say we have a list of index labels that we want removed, so to speak. We want to access a dataframe that does not include these labels. Let's call them unwanted.

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

At this point we could reindex our dataframe in a way that leaves these labels out, or excluded. So, to put this in code, we'd say

`players.reindex(index=set(players.index).difference(unwanted))`

What we're doing here is we're taking a set difference between the set of all index labels.. remember that's the range of integers going from 0 to 464... so we're taking the difference between that set and the set of numbers in the unwanted_rows list, which is 1, 2, 3, and 4

And that set difference is being passed on to the index parameter. So effectively we are using reindex to remove rows.

Obviously we could do something similar to remove columns at the same time. So if we have some unwanted columns...

`unwanted_cols = ['name', 'position', 'position_cat']`

..we could pass a set difference between the column labels and our unwanted labels as an argument to the columns parameter...

`players.reindex(index=set(players.index).difference(unwanted_rows),`

`columns=set(players.columns).difference(unwanted_cols))`

So now in one method call we're excluding both rows and columns. Technically, we're not removing anything, we're just creating a copy of the dataframe that leaves out certain labels.

To be honest, I'm a bit conflicted about using reindex this way because it's not the right way to go about things. I think the more pandorable way to remove rows or columns is the drop method.

With that said, I think it's still helpful to explore alternative ways of solving the same problem. It helps us establish these connections across methods and get a better sense of pandas more generally. For example, we now know that we could replicate the functionality of the drop() method using reindex().

▼ **Lecture 24** - Null Values In DataFrames [**7:13**]

So we've talked about dealing with duplicates in dataframes, now let's cover another usual suspect: NaN's, null values or simply gaps in the data. When we were working with series, we did see the isnull() or its alias, isna()

methods which returned series of booleans.

For a quick refresher, if we isolate a series from our dataframe consisting of all player ages.

```
players.age
```

Now, let's assign this to the variable ages.

```
ages = players.age
```

...and if we want to see whether this series has any NaN's, we simply chain the isna() method

```
ages.isna()
```

That gives us a long series of booleans, not very reader-friendly, but as we've seen, quite useful. Because we could pass this right into an indexer to select the records that have NA's

```
ages[ages.isna()]
```

And in this case there is none. It appears that we have ages for all of our players, and so we simply get back an empty series. Hopefully this has been just a review. If you're not too sure about any of this please refer to the series sections where this is covered in detail.

Now, this approach is great for a one dimensional labeled sequence of values, in other words series. But what do we do for dataframes? Are we supposed to repeat this process for all our columns?

Fortunately, the answer is no because the isna() method also works on dataframes. Let's take a look at that by simply calling this method on the dataframe

```
players.isna()
```

Ugh. That is NOT pretty. So we get back a new dataframe where each value has been replaced by a boolean indicating whether the value is NA or not.

To quickly count how many NAs there are in this dataframe, if any at all, we could use a very performant numpy function simply called count_nonzero.

```
np.count_nonzero(players.isna())
```

This is going to give us a count of the truthful values in this dataframe. But remember, a True in the dataframe returned by isna() indicates an NA value.

So it seems that we have 4 NA's in our players dataset. Now which records are those? Could we simply use the technique we saw with series earlier in the lecture to identify those records? In other words, could we take this "players.isna()"... and pass it to an indexer

```
players[players.isna()]
```

Well, the short answer is no, this won't work. We can't index like this.

First we need to extract the values from this boolean dataframe to create a boolean array of arrays

```
players.isna().values
```

So that is going to give us a list of boolean lists which we could then simply pass to an indexer to identify the records that contain at least one NA

```
players[players.isna().values]
```

Nice. So four records from our players dataframe have at least one NA.

But if we look closely at this, we realize that it's only 3 unique players, because one of them, Granit, who as we see here has an NA position and an NA market_value, is showing up twice.

Another way to say this is that each record, or row, will appear as many times as there are NA's in it. To get rid of this duplication we could simply chain a drop_duplicates() to the end.

```
players[players.isna().values].drop_duplicates()
```

That's great. We've seen how to count NAs and even identify the records that contain nulls in dataframes. Next, we'll turn our attention to the problem of what do we do with these observations, with these gaps in the data.

▼ **Lecture 25** - Dropping And Filling DataFrame NAs [**7:49**]

Alright, so let's talk about handling NAs in dataframes. Now, null values usually indicate a gap in the data. This could be information that was lost, not available, or simply incorrectly recorded. So one inclination we could have is to fill or replace these null values with something more meaningful. And the fillna() method is really useful for that. We did see this method in the series section, and the good news is that it works in a very similar way for dataframes too.

```
players.fillna('some meaningful replacement value')
```

In the most basic scenario we simply specify the value that we want to replace our nulls with. And this method will return a copy of the dataframe where the nulls have been replaced with this new value.

Now, I'm going to isolate the index labels that we knew had some nulls. That way we could take a closer look at the replacements.

```
players.fillna('some meaningful replacement value').loc[[30, 192, 195]]
```

Notice that by doing this, we're simply plugging this new value wherever we see an NA. In reality, our needs might be a bit more sensitive to the context. For example, a null in the market_value column should maybe, at the very least, be replaced by a numeric value.

To be able to do that, we could use the dictionary syntax of the fillna method. Instead of providing a generic fill value, we're now passing column-specific defaults

```
players.fillna({'market_value': 100, 'position': 'RM'}).loc[[30, 192, 195]]
```

So we're saying replace market value with 100 and position with 'RM'... is 'RM' even an option?

```
players.position.unique()
```

Yes it is, this stands for right midfielder. Good.

And by the way, we could get even more dynamic here and say for example, replace market value with the average market value across the entire dataset.

```
players.fillna({'market_value': players.market_value.mean(), 'position': 'RM'}).loc[[30, 192, 195]]
```

Very nice.

So these are just a couple of options to replace nulls in dataframes. By using the fillna() method we could replace in a common value for all our nulls, or using a dictionary we could specify different values for different column labels.

This is great is we want to plug in the gaps in our data, but an entirely different approach to dealing with null values is to simply exclude them. And for that we have the dropna() method, which we covered quite a bit in the series sections. So we go

```
players.dropna()
```

And we get back a new dataframe, which excludes rows containing null values. We could confirm that by indexing for the labels that we know to contain nulls.

players.dropna().loc[[30, 192, 195]]

And... they're not here; they've been dropped.

Excellent. Right now we're dropping index labels, so entire dataframe rows are removed if any of the values contains a single null. Which is to say that we're relying on default values for two parameters.

```
players.dropna(axis=1, how='any').loc[[30, 192, 195]]
```

If we switch axis from 0 to 1, we would be dropping entire columns if a single value is null. So notice how there's no market_value or position columns in our dataframe anymore.

And if we change the argument to the how paramter from 'any' to 'all', we notice that those two columns are back and nothing is dropped because the condition for dropping nulls is not met in this case. With how set to 'all', a given column would only be dropped if it contained nulls exclusively.

So those are some ways to deal with null values. Generally speaking, we either replace them or remove the records associated with them, and the correct approach varies depending on the analytical context.

▼ **Lecture 26** - BONUS - Methods And Axes With fillna() [**10:05**]

When we discussed the fillna() method in the previous lecture, we didn't explore some exciting parameters that the method supports. Most importantly, I think we should be aware of the built-in methods and the way they interact with the axis parameter. So in this bonus lecture we'll take a deeper dive into those.

```
players[players.isna().values].drop_duplicates()
```

First things first, notice that nothing we did in the previous lecture modified the underlying dataframe. All the filling or dropping we did was not inplace, and so the changes did not affect our original dataframe. A the beginning of this bonus lecture, we still have those three unique records containing 4 NA values.

Now, previously we used fillna with strings or dictionaries to do replace NAs with new values we specified

    # players.fillna('some meaningful replacement value').loc[[30, 192, 195]]

    # players.fillna({ 'market_value': 100, 'position': 'RM' }).loc[[30, 192, 195]]

An alternative would be to not provide any value at all and instead just pass, what's known as a method. So we could say

players.fillna(method='ffill').loc[[30, 192, 195]]

That's really interesting. What we see is that the null values are gone and in their place we have what appear to be reasonable values. So what exactly is happening here?

What pandas is doing here is it's copying forward the previous non-missing value to forward fill our NA value... that's what ffill stands for. So if we take a look at the previous index label, 29, we notice that the record labeled 30 which had an NA under position and another one under market_value now reflects the non-null values of the previous row, or the one labeled 29.

players.fillna(method='ffill').loc[[29, 30, **191**, 192, **194**, 195]]

And the same goes for our other two records. They are now filled with the non-null value of their respective preceding records. In the case of 192, that would be 191, and for 195, 194...

...and remember this is because we have a dataframe that is sorted in ascending order by index.

Now, a silent parameter here is the axis, which we're currently defaulting to 0, or in other words to the index dimension

players.fillna(method='ffill', axis=0).loc[[29, 30, **191**, 192, **194**, 195]]

When the axis is 0, the filling happens vertically. So we copy forward from the previous row record. You could visualize the copying alignment as going parallel to the index we specify in the axis. So when we say axis 0, we mean the index, and in this case the copying happens vertically.

If we switch the axis to 1, we now start targeting the column dimension.

players.fillna(method='ffill', axis=1).loc[[30, 192, 195]]

So the copying of, or the filling of, NA values now happens horizontally. For example, the position column for the player labelled 30 is now copied over from the closest column to its left, which is age. And likewise for market_value for records 192 and 195, we're getting positon_cat copied over.

Honestly, this doesn't make a lot of sense for our dataset, but it's worth knowing that this kind of fillvalue behavior is a possibility.

slide switch

So think of the axis parameter as aligning our copying either along the index when the axis is 0 or along the columns when the axis is 1. The method parameter then determines the direction. When we say 'ffill' we mean copy from top to bottom if the axis is 0, or left to right if the axis is 1.

The other option is 'bfill', backward fill, and by this we mean the exact opposite direction. So backfill along axis 1 copies from right to left horizontally, and backfill along axis 0 copies from bottom to top vertically.

Okay this is pretty cool, but it may be a bit overwhelming to take in all at once. I would recommend not trying to learn this by heart. Try to develop a sense for how it comes together, and with practice you'll become increasingly comfortable with these notions.

Again, a good rule of thumb is to visualize the copying direction as going parallel to the index specified in the axis. So when we say axis 1, we mean the column, and in this case the copying happens horizontally.

Great. These filling methods are particularly useful when working with time series data because they allow us to plug the gaps in our series with relatively reasonable values.

In our players dataset, when we have a player with a missing market value, and we say, "oh just forward fill the previous player's market value", we might be way off. Chances are it won't really be a good guess. But if we're tracking the daily price of gold for example, and we find in our series a couple of gaps, say some days for which we don't have a price, then forward filling or backward filling from the previous or the next day is a reasonably okay approach.

I just wanted to bring that up in case these replacements we did here seemed arbitrary. We will see better uses of this method in future sections.

Oh and by the way, an alias for 'ffill', forward fill, is 'pad'. And 'bfill' is also equivalent to 'backfill'. These pairs are identical although I personally prefer the 'ffill'/'bfill' duo.

Word of the lecture: duo which means a pair. I think it means the number two in latin. Why say it in English, when you could say it in Latin, right?

But yeah, I prefer 'ffill'/'bfill', but 'pad'/'backfill' is a functionally identical. Okay, let's move on.

▼ **Lecture 27** - Skill Challenge [**1:37**]

1. From our players dataframe remove the rows labeled 2, 10, 21 and the market_value columns. Do not modify the underlying dataframe. Assign the result to df2.
2. Does the nationality column in df2 contain any NA values? How many unique nationalities are there?
3. Starting from df2, isolate a dataframe slice of players that contains only the unique age-position combinations for each club. Do not include the club column itself.

▼ **Lecture 28** - Solution [**7:00**]

1

```
players.drop(labels=[2,10,21], axis=0) \
    .drop(labels='market_value', axis=1)
```

or

```
players.drop(index=[2,10,21], columns='market_value')
```

2

```
df2.nationality.isna().sum()
df2.nationality[df2.nationality.isna()]
```

part2

```
df2.nationality.drop_duplicates().size
```

or

```
df2.nationality.nunique()
```

3

```
df2.drop_duplicates(subset=['age', 'club', 'position'], keep='first').loc[:, ['age', 'position']]
```

▼ **Lecture 29** - Calculating Aggregates With agg() [**9:23**]

Let's now take a look at a method that will become very useful when we start talking about grouping our values along several attributes. We'll see a lot more of this in future sections, but let me give you a quick introduction to agg(). It's short for aggregate and what it does is it accumulates or aggregates the output of a function that we specify and then collapses the entire axis into that single value.

Okay, maybe that sounds more complicated than it is, so let's just run agg() on the entire dataframe and see what we get... say we want to apply the mean function.

```
players.agg('mean')
```

So that gets us a series where each label is a numeric column from our dataframe, and each value is the mean or average of those columns. In fact, let's confirm one of these values. I'll pick new_signing, which indicates whether the player was signed in this season or not.

```
players.new_signing.mean()
```

And yes, it is the same number that we see.

So this is the aggregate method in a nutshell. We could certainly pass other functions as this first positional argument. For example, if we want the minimum value by column we could say min

```
players.agg('min')
```

By the way, instead of this string, we could also just pass the the built-in python min function, by simply removing the quotes, or even the numpy version of minimum.

players.agg('min') # with min and np.min too

All of these give us exactly the same output.. which apparently now includes strings as well.

And the reason for that is that it is technically possible to apply comparative operators to strings, and that works by comparing the position of their respective codepoints. I don't wanna go into too much detail here but let's do a quick aside

===ASIDE

So technically in Python we could say 'a' < 'b' and we get a True. And why is that? Well, colloquially we know that 'a' comes before 'b'

But the true reason is that each of the characters is associated with an integer known as its Unicode codepoint. So for example 'a' corresponds to 97 and 'b' to 98, and so on.

ord('a')

So technically when we compare the character 'a' to the character 'b', python does not complain, because it does have something to work with.

Similarly if we had an sequence of strings stored in a list, and then we run a min or a max function on that list, we still get some meaningful output back.

a = ['a', 'b', 'd']

min(a)

But notice that this flexibility in syntax ends the moment we start mixing types by throwing in a truly numeric data into the comparison. So if we add a float or an integer to this list of strings, I'll get a type error.

try it with string

And the reason this is because the comparative operator is not defined for comparisons between int and str.

'a' > 100.0

===END ASIDE

Okay, so let's end the aside here. But I hope it's now clear why when we say min in the agg function, we got back some strings.

If we want more control over this, and actually prefer to get rid of these strings, we could choose to prefilter the columns we aggregate on by using the select_dtypes() method. And I'll go ahead and use a numpy type here to cover all numeric types in our dataframe.

players.select_dtypes(np.number)

This returns a dataframe where only the numeric columns are included. And to that dataframe we could then simply chain on the agg function.

players.select_dtypes(np.number).agg('min')

Beautiful. So we now have the min applied only where we want it.

Another cool thing about agg is that we could pass in a list of functions and get back several aggregates. So if we go back to our expression here and instead of 'min', we say

players.select_dtypes(np.number).agg(['min', 'max', 'mean'])

We now have a dataframe with min and max and average values across all numeric columns in the dataframe. Which is pretty neat for the amount of code that we wrote.

So that's agg method. It's an alias for aggregate, by the way. So substituting it in does not change the functionality whatsoever, but in most of the code I've seen the agg version is almost exclusively used.

One important takeaway from this lecture is that agg reshapes our data. Right? We start with a dataframe and then we get a series, or if we pass multiple functions in, we get another dataframe back where each column data is reflected as collapsed or cumulated.

But in both cases the data is being reshaped, a function is applied, and a new data structure with the aggregate output is returned. That is the essence.

▼ **Lecture 30** - Same-shape Transforms [**14:43**]

Pandas also has a dedicated method used for applying custom functions to the entire dataframe. And the beauty of this method is that it guarantees that the shape of the dataframe will not change. It works like this.

Um, let's say we want to convert our market value and fpl values from one currency to another, maybe from US dollar to euro, which is the common currency of the European monetary union. To convert from one currency to another, we need to have a conversion rate, or an fx rate as it's called. Due to differences in denomination and central bank monetary policy, one USD is not equal to one euro. To get the fx rate, we could simply google "usdtoeur" and I see here that one US dollar is worth about 0.91 euro as of mid 2020.

So let's get down to using this. I'll start by indexing for these player value columns using the loc label-based indexer.

```
players.loc[:, ['market_value', 'fpl_value']]
```

So far so good, we have a dataframe of shape 465 by 2, containing the two columns we're interested in converting to euro. So now, to this dataframe, we need to apply the transformation.

For this example, I'll use the transform method with an inline lambda function... think of the x here as representing an entire column. So market_value is passed in as x, it's multiplied by 0.9, then fpl_value is passed in, it becomes the new x, it's multiplying by 0.9, and the method execution ends

```
players.loc[:, ['market_value', 'fpl_value']].transform(lambda x: x * 0.9)
```

Nice. So we see our value columns have now been converted. This is a basic example. And to be honest, we could just straight up use the multiplication operator to do this conversion.

```
players.loc[:, ['market_value', 'fpl_value']] * 0.9
```

I think the same mechanics will apply here, and with even less overhead, so this second approach might even be a bit more performant that the one based on the transform method.

But say we had a different task, um, say we want to apply a string transformation to all string columns and we don't want to or we can't specify what this string function should be. We want the transformation to be randomly selected at runtime. As far as I know there is no built-in pandas or python methods that allow us to do this. So let's begin by creating our own function.

Before we do that actually, I'm going to take a quick aside to introduce two concepts that we will end up using.

===ASIDE

The first is choice and this is a method that allows us to randomly sample from a sequence of values. I'm going to use the one from the random module in python but there are alternatives.

```
from random import choice
```

Let's create a list of names to sample from

```
names = ['Bud', 'Brooke', 'Paleo']
```

Is that even a name? Never mind. So now if we pass this list to choice, choice returns one of the items. Simple enough.

The second concept I wanted to bring up is something we haven't discussed at length yet and those are the built in pandas string wrapper functions. We'll spend plenty of time exploring them in a future section. For now, notice that in pure python, we could say something like

```
'Andy'.upper()
```

but if we try applying something similar to a pandas series, and I'll go ahead and quickly create one here

```
ser = pd.Series(['Bud', 'Brooke', 'Paleo'])
```

```
set.upper()
```

...we quickly realize that series have no such method. To be able to do this in a pandas series context, we simply add the str accessor between the method and the series it is being applied to.

```
ser.str.lower()
```

And this pattern is followed by the other text transformation methods in python. So for exmaple, if we wanted title case we'd say

```
ser.str.title()
```

..there's also swapcase, which flips the lowercase to uppercase and vice versa

```
ser.str.swapcase()
```

===END ASIDE

So that's the end of our aside. We covered choice which is used to select random items from a non-empty sequence, and we also explored how to access and apply string methods directly on pandas series.

Now it's time to get back to defining our random transformation function. Let's spend a couple of seconds writing down our goal:

the function should:

- apply a random string capitalization method
- to a sequence of values, and
- return the transformed sequence

Let's call our function random_case... and just pass on the definition for now

```
def random_case(x):
    pass
```

In the end this function will be called from within transform, and perhaps we should only focus on string columns. We'll do that using the select_dtypes method.

```
players.select_dtypes(include=object).transform(random_case)
```

Okay. So let's get back to the function body and flesh it out. The function takes a single argument, remember x will be assigned to a column by transform, so within the function itself, x represent a column.

Now, we're supposed to apply a random random case string method to this column. And we want to change the case unpredictably. Right? We want it to be randomly decided at when the function runs. To do that I'll start by creating an array of bound function references representing the possible transformations.

```
funcs = [x.str.swapcase, x.str.lower, x.str.title, x.str.upper]
```

And then let's use choice to select from this array.. invoke the function.. and finally return the output

```
choice(funcs)()
```

Very nice. If we go back to transform and call it now... we see that the case is changing randomly.

```
players.select_dtypes(include=object).transform(random_case).head()
```

In fact let's do this a couple of times so that we see this more clearly. We'll wrap our transform in quick for loop, and maybe do it 4 times

```
for i in range(4):
    print(players.select_dtypes(include=object).transform(random_case).head())
```

Excellent. So we see that the case for our dataframe string columns is changing in odd, unpredictable ways, exactly as we intended it to.

So that was a quick introduction to the dataframe transform method. Notice that unlike the agg() method we saw in the previous lecture, transform does not reshape the dataframe. It simply applies a function in place without changing the size or shape of the resulting dataframe.

In terms of what that function is? Well it could be anything. We started with an in-line lambda function that did a simple multiplication and then we stepped it up a notch to define a custom function that applied string methods randomly at runtime. So the possibilities are endless here.

▼ **Lecture 31** - More Flexibility With apply() [**13:14**]

Okay, it's time to revive the apply() method, which we did discuss and practice in our series section, but I feel like we should at least briefly mention how it behaves with dataframes.

So let's talk about apply() in a two dimensional context.

First, when would we use the apply() method? The short answer is whenever we want to apply a given function to an entire row or column. The dataframe apply method gives us the opportunity to apply a function across the entire dataframe, one row or one column at a time. Let's demonstrate that.

Say we want to modify all our floating point columns, and round them to the closest integer. We could start by defining a function called round_floats, it will have a single parameter, let's call it x.. and within the function body we'll check whether it is a float, and if it is, we'll return a rounded value

```
def round_floats(x):
    if x.dtype == np.float64:
        return round(x)
return x
```

Clear enough, right? Remember this function will be applied to the entire dataframe, so to avoid any type errors, we are first checking the dtype of the incoming argument. X will be assigned to an entire row or column, which means that x will be a series when this function runs. And as a series it will have the dtype attribute. So before applying the rounding, we will check that the dtype is float. If not, we will simply return the series unchanged.

```
players.head()
```

```
players.apply(round_floats).head()
```

Nice. So now our floating point columns are all rounded. It's not that easy to see here. Perhaps we should narrow down the the view to only target our floating point columns.

```
players.select_dtypes(np.float64).head()
```

And now let's apply the function we defined above to do the rounding

```
players.select_dtypes(np.float64).apply(round_floats).head()
```

Great. As we see, the values are now rounded to the closest integer. Where we had 9.5 before, we now have 10, 7.5 → 8.0 and so on.

So that the apply method. One thing that may jump out right away is that this is awfully similar to the transform method we discussed in the previous lecture. I mean, what we're doing is defining a function and then using a method to apply that function to each column. In fact, we could go in and swap apply with transform and everything would continue to work exactly the same way.

```
players.select_dtypes(np.float64).transform(round_floats).head()
```

So one question is why does this method even exist then, considering that it is not an alias, or another name for transform?

The reason is that apply is a bit more flexible and powerful, it has more general scope, specifically in that it support aggregations as well as same-shape transforms.

You could think of the dataframe apply method as a combination of transform() and aggregate() in that a single function, a single piece of syntax, supports both types of operations.

So first we have...

```
# apply as aggregate
```

For example, if we wanted to aggregate our numeric columns, we saw that the prototypical way of doing that was to use the agg() method, passing in 'mean'.

```
players.agg('mean')
```

But just as well, we could say apply 'mean'.

```
players.apply('mean')
```

In this case apply acts exactly like aggregate, and it produces identical output too. By the way, if you're wondering, we would not be able to do something similar with transform()

```
players.transform('mean')
```

...and the error in fact is quite illustrative. It says "transforms cannot produce aggregated results"

slide switch

So the key point here is that apply is a bit more advanced and flexible in that it supports two kinds of operations. The ones that change the shape and dimensions of the underlying dataset—so that would be things like aggregate—and also in-place transformations that return identically shaped dataframes.

When we call apply(), the method first tries to determine whether an aggregation will take place or not. If yes, it proceeds in a way similar to agg(). If not, it behaves similarly to transform().

notebook switch

Very nice. One last thing. In all of these methods, we could control how we want the transformation to be applied.. columnwise or rowwise by changing the axis parameter. In our earlier example (scroll to round_floats), because we were simply rounding our float values, it doesn't really matter if we traverse the dataframe by column or by row. We're doing a basic in place transformation that will be identical no matter how we slice it.

But when we aggregate to calculate the mean, for example, changing the axis produces fundamentally different results.

For instance by default the axis is 0, corresponding to the index labels, and as we saw that means that we aggregate values vertically.

`players.apply('mean', axis=1)`

So here we get a mean age of 26.77 which is exactly equal to

`players.age.mean()`

But if we flip the axis to 1, or the column axis... we switch to aggregating horizontally

`players.apply('mean', axis=1)`

which, first, explains why we now get 465 averages. Right? We're now calculating horizontal averages by row, and that's how many rows there are in our dataframe. Now, these averages do not make a lot of sense in our players dataset, because they represent the average of market value, age, fpl value, page views, and so on. So they're not very meaningful.

But let's ty replicate one of them just so we understand the dynamics perfectly. Say, we pick this 461st one.

First we need to extract the record with the label 460.

`players.loc[460, :]`

That gives us a series containing strings and numeric values. Which means we can't just apply a mean here because this series has non-numeric data types. If we attempt that we get a TypeError.

`players.loc[460, :].mean() # TypeError`

Also, we can't use the select_dtypes method, because that's only available on dataframes, and this one here is a series, it's a one-dimensional slice of the dataframe representing a single row. So let's refine it a bit to exclude object dtypes.

We could do that dynamically using a list comprehension to generate a python list of booleans

`players.loc[460, [dtype != object for dtype in players.dtypes]]`

Looking better. And why did this work? Well, we're using the loc indexer, to select the row with the index label 460, but not all its columns. Which columns do we want? Well, we're dynamically checking the dtype for each of them, and only selecting those that are not of object dtype. So this part (highlight) produces a python list of trues and falses, where the trues correspond to the numeric columns, and are therefore selected.

At this point, because we're only working with numeric values, we could simply calculate the .mean()

`players.loc[0, [dtype != object for dtype in players.dtypes]].mean()`

And there we have it. Does it match what we got from apply?

`players.apply('mean', axis=1).loc[460]`

Yes it does. Very nice.

So that's the apply() method. This lecture turned out to be much longer than I intended, but I hope it's been useful. We will summarize all of this again at the end of the next lecture, but for your own mental map, for now, just remember that apply() is like a combination of transform() and agg(), in that it could do either operation while traversing either axis.

▼ **Lecture 32** - Element-wise Operations With applymap() [**13:35**]

Phew! That was a couple of long, intensive back to back lectures. I hope you're hanging in there and learning a lot. We'll introduce one last concept here before we move on to some practice problems.

So the methods we've discussed over the past couple of lectures, agg(), transform() and apply(), they operate on entire columns or rows at once. They take advantage of this concept of vectorized operations which is really a numpy feature that allows us to realize huge performance gains by applying a given operation on a set of values, all at once, instead of operating on them individually. This set or group of values is called a vector, from linear algebra.

But in numpy this is only made possible by what's known as **Single Instruction Multiple Data (SIMD)** processors. SIMD allows a processing unit, which could be a CPU or a GPU, to perform the same operation, on multiple data points, in a single processor cycle.

switch to slide - explain

However, not all functions could be vectorized. So we can't expect all functions to accept an array or numpy ndarray and return another array or value. Occasionally, we might be working with a kind of function whose logic depends on accepting and operating on a individual values, in other words operating one item at a time. And for such cases pandas offers the applymap() method.

Let's come up with an example here. Say it's another year and we need to adjust our market value and fantasy player values, the fpl values, for inflation. You've probably heard of inflation before, and if I had to put it in as few words as possible, it's a general increase in the price of things. Most central banks in the world try to maintain small positive inflation, I think the golden target in developed economies is about two percent a year.

Let's say that for now we need to adjust these two columns for inflation.

```
infaltion = 1.02
```

I'll start by isolating them in a new mini dataframe.

```
mini_df = players.loc[:, ['market_value', 'fpl_value']]
```

Now to adjust this mini dataframe of values, we could simply multiply by inflation. We don't actually need a function. And this is a highly efficient, vectorized operation. It operates on the entire column at once.

But unfortunately not all functions could be vectorized. Let's assume, for whatever reason, we wanted to know precisely when each 100th value was adjusted for inflation. So every 100 values or so we want the function to print out the timestamp when the transformation took place.

Let's create a quick function here that does just that.

```
from datetime import datetime

counter = 0

def log_and_transform(x):

    global counter

    counter += 1

    if counter % 100 == 0:

        print(f"It's {datetime.now()} and I just adjusted the {counter}th value.")

    return x * inflation
```

The syntax we're using here is called literal string interpolation, or f-strings, and it was added with python 3.6. You could check your python version by importing the sys module and looking at version

```
import sys

sys.version
```

So make sure you have python 3.6 or higher if you want to use this syntax.

Okay, it's time to test this function

```
mini_df.applymap(log_and_transform)
```

That's pretty cool. We're now getting a little notification for every 100th value that is adjusted. And this is only made possible because we are applying this function element-wise. So one value at a time is being passed through our logger.

Our logger function is impossible to vectorize without losing functionality. For example, it only fulfills its logging function if values are passed in one at a time. Because if we apply the inflation adjustment in a vectorized way, the entire vector of column values is adjusted at once, so our counter never hits anything close to 100.

An example of doing this in a vectorized way is to use the apply function instead.

```
mini_df.apply(log_and_transform)
```

Now, our logger doesn't get individual values, but rather entire columns, and so this criterion is never met.

Nowadays as we discussed there's hardware level support for vectorized operations, and they significantly speed up our work, which is why we should always be prefer them over elementwise operations.

But as we saw here sometimes our hands are tied and the functionality we need to implement is not conducive to vectorization.

And it is exactly for such cases that we fall back to the applymap() method.

Now, we've really covered a lot over the past couple of lectures so let's take a moment to put this in context.

Again, one of the biggest advantages of relying on numpy and pandas for our data analysis and manipulation is that it offers us a very powerful and performant set of tools to conduct vectorized operations on our set of data. And we saw examples of vectorized ops with apply, agg, and transform. But in contrast to these vectorized operations, the applymap() method gives us access to each element, and so it gives us an opportunity to work with transform each element one at a time, instead of working on rows or columns all at once.

▼ **Lecture 33** - Skill Challenge [**2:07**]

1. Create a standalone function that

- accepts a single parameter x
- returns the string 'relatively unknown' if x is less than 220
- 'kind of popular' if x is between 220 and 600 (non-inclusive)
- 'popular' if x is between 600 and 2000 (non-inclusive)
- 'super-popular' otherwise

2. Apply the function from the step above to the players page_views column. Use a method that supports vectorized operations.

3. Add the output from the step above as a new column to the players dataframe. Name the column popularity.

4. How many "super-popular" players are there?

▼ **Lecture 34** - Solution [**4:56**]

```
# 1
def get_popularity(x):
  if x < 220:
    return 'relatively unknwon'
  elif x < 600:
    return 'kind of popular'
  elif x < 2000:
    return 'popular'
  else:
    return 'super-popular'

# 2
players.page_views.apply(get_popularity)

# 3
players['popularity'] = players.page_views.apply(get_popularity)

# 4
players[players.popularity == 'super-popular']
```

▼ **Lecture 35** - Setting DataFrame Values [**6:54**]

In the previous section, we explored a lot of techniques for extracting data from dataframes, and then in this section we took a deeper dive into applying transformations and aggregations.

But sometimes we don't want that large scale operation. Occasionally we just want to spot change our data. As in pick a value and modify just that single value. We could easily do that using the equals assignment operator.

players.head()

Say we wanted to change Theo Walcott's position from RW (right wing) to CM (central midfielder). Well, to do that, we could certainly use approaches that we are already quite familiar with. We are targeting the value at index label 3 and column label 'position', so using the loc indexer we could index select that value... and using the equal operator we set it to 'CM'

players.loc[3, 'position'] = 'CM'

If we take a second look at our dataframe now, we notice the change.

players.head()

Obviously an alternative to label based indexing is position-based indexing. Let's now use iloc to select Walcotts's position and set it back back to a 'RW'

players.iloc[3, 2] = 'RW'

Now this is cool, and it certainly works great, but if we're setting individual values it's best to use the highly performant at and iat indexers. Remember we talked about these in the previous section on dataframes.

at[] is the label indexer which is most similar to loc, and we could use it here to set the position back to right wing

players.at[3, 'position'] = 'CM'

Similarly instead of labels we could use positions with iat

players.iat[3, 2] = 'RW'

...and we've come full circle with Walcott still a right wing. So we have reviewed 4 ways to assign spot values in dataframes.

Now what was the point of the at and iat indexers? Remember, unlike loc and iloc, at and iat support only one type of syntax and so they have much less overhead in their function definitions and are incredibly fast as a result. Actually, let's go ahead and get a sense for how much faster they are.

I'll use the timeit magic function on loc and at

add %%timeit

Wow, so we see here the at is about 50 times faster than loc. That's 50 times, or 5000% faster. And the same goes for iat relative to iloc

uncommed %%timeits for iat and iloc

This is crazy. I mean we are talking about a difference in microseconds here but please do not underestimate how quickly this could become an important difference when working with much bigger datasets.

▼ **Lecture 36** - The SettingWithCopy Warning [**7:14**]

In the previous lecture we used loc and iloc as well as the highly performant at and iat indexers to set values in our dataframe. In this lecture I wanted to discuss a warning that pandas might throw at us when we try to update or set values in our dataframes. So let's take a look.

players.head()

Say we now want to change Petr Cech's page_views to '2001'. We saw how this could be done using loc, iloc, at, or iat, but a fifth approach to use square bracket indexing. And in fact you are very likely to come across code that does assignments like this.

To change Petr's page_views we could start by isolating the page_views column

players['page_views']

Nice. So we now have a series containing page views. And to isolate Petr Cech specifically, we could simply chain another square brackets to extract from this series

players['page_views'][2]

Okay, cool. So now that we've isolated the value we want to update, we could go ahead and set it to 2001 using the equals operator

players['page_views'][2] = 2001

Ugh and we get this interesting warning... (read it) So what's happening here?

The reason we get this warning is because pandas cannot guarantee that we are working with the actual underlying dataframe when we do indexing like this, which is known as "chained indexing". Depending on how the memory is structured we could be working with a copy of the data or a view to the underlying data. We will discuss this at length in the next lecture, but for now I think it's important to realize that if we are working with a copy of the dataframe, any change that we do to that copy does not change the underlying dataframe.

So pandas has this warning to let us know that it cannot guarantee that the assignment we're trying to do will actually happen in the underlying dataframe. In this case, it might or it might not depending on how the memory for this dataset is laid out.

Um, let's actually check whether it changed.

players.head()

Oh yeah it did change. So in this case, given the structure of our dataframe, we did get a view, and through our setting operation we managed to change the underlying dataframe.

Um, let's try to trigger this error in another way. Think of methods that we know return copies of the dataframe. Remember when we talked about drop_duplicates()?

players.drop_duplicates()

If we do not include the inplace parameter in this method call, we simply get a new dataframe (or a copy) from where the duplicates have been excluded. So we know for a fact that in this case we are working with a copy. Now, if we try to do some assignment using this copy... we would trigger the same error

players.drop_duplicates()['page_views'][2] = 3000

But in this case we also know for a fact that the assignment could not have affected the original dataframe, because we know that drop_duplicates returns a copy. We could confirm that real quick

players.head()

And as we see Petr's views are still stuck at 2000.

Awesome, we'll pick up this topic in the next lecture as well, but I wanted to mention that it is possible to turn this warning off by setting the chained assignment mode to None

pd.options.mode.chained_assignment = 'warn'

That said I would strongly recommend against doing this. I think it's a very useful warning and it serves to keep us mindful of whether we're working with a copy or a view especially when the difference is important for what we are trying to do, as we saw with assignments.

In the next lecture we'll look more closely at the difference between views and copies as well as come up with some rules to guarantee that we avoid seeing this warning again.

▼ **Lecture 37** - View vs Copy [**9:00**]

Alright, so we talked about the SettingWithCopy warning and we alluded to the difference between a copy and a view. In this lecture we will more closely explore this important concept and look at ways to avoid the uncertainty and the warning.

Whenever we index, extract slices, or apply a method to our dataframe we technically run into this question of whether we're working with a copy or a view.

slide switch

To briefly define these terms again. A view is like is like a window to the underlying data. It's like an opening that gives direct access to the data. What we see is what's really in the dataframe. And so if we modify what we see, we end up modifying the dataframe.

A copy on the other hand is a replica of what's in the dataframe. It looks very similar, it might even be exactly the same shape, including all the rows and all the columns. But it is still a copy. So if we change the copy, we don't affect the underlying dataframe.

I hope this concept of working with a view or a copy is clear, but how do we know if pandas is giving us one or the other.

switch to slide 2

In order to simplify our reasoning and practice, let's think of it in terms of these two basic principles:

- pandas loves to give us copies
- if you use loc/at or iloc/iat, you're guaranteed a view

So start by always assuming that you're working with a copy. Any method that includes an inplace parameter is by the way guaranteed to be giving us copies if inplace is false.

So methods like drop_duplicates, drop, dropna, all of those return copies of the dataframe. Their inplace parameter is set to False by default. Obviously any values you set on the output received from these methods will not impact the underlying dataframe. It cannot. Because we're only changing the copy.

Now say we are about to change a couple of values and we need to guarantee that we are working with a view? Right? We want to make sure the values in the underlying dataframe change, and we also want to avoid the error. How do we go about that?

The short answer is to always use the loc and iloc indexers, or the at and iat if we are setting a single value.

notebook switch

For example, if we want to change or set multiple values at once... say we'll set the position for the first 4 players in the dataframe to a bunch of other positions

`players.loc[0:3, 'position'] = ['CM', 'RW', 'CB', 'GK']`

Let's take a look at our dataframe to see if the changes took effect

`players.head()`

And, yes they did. Let's do another one. Let's replace all the players with an 'Aaron' first name with what is likely to have been their nickname at some point in their life.

First let's see how many such players there are

`players.loc[players.name.str.startswith('Aaron')]`

`players.loc[players.name.str.startswith('Aaron'), 'name'] = 'Ronny'`

Sorry for making this ridiculous, though somewhat likely assumption. So we're using a boolean mask to isolate players whose first name is Aaron, and then we select the name column. In the end I set it to Ronny. Did the change take place?

`players.loc[[15,157,176,455]]`

In both of these examples we managed to successfully change the values in the underlying dataframe without triggering the SettingWithCopy warning. Why is that? Well because there's no uncertainty over whether we get a copy or view when using the loc, iloc or at or iat indexer. Pandas guarantees us a view.

Importantly though, this guarantee only works if we use the indexers on the original dataframe. Notice how we're saying players.loc in both examples. If we break this rule and come up with some sort of mixed chained indexing like

`players['age'].iloc[1] = '12'`

`players.drop_duplicates().loc[3, 'position'] = 'CM'`

Yeah, these won't work, we're back to the same SettingWithCopyWarning.

And the problem here is that first link in the chain is not guaranteed to return a view. And then we're chaining an indexer that is guaranteed to return a view. But it's already too late because by that point we no longer know what we're working with.

So that's view vs copy. Very important concept to be aware of when working in pandas. Again my recommendation is to not turn the warning off, but instead use a couple of basic guiding principles to stay away from the pitfalls that the warning is meant to help us avoid.

https://stackoverflow.com/questions/23296282/what-rules-does-pandas-use-to-generate-a-view-vs-a-copy

▼ **Lecture 38** - Adding DataFrame Columns [**8:02**]

In this lecture we'll explore adding columns to our dataframe. Sometimes we need to expand our dataset by adding new attributes. We did this in a previous skill challenge, when we added the popularity column to our players dataset, but we return to this topic to cover it a bit more in depth.

As we saw, quite possibly the easiest way to add a column is to just use the assignment operator on square brackets with a label that does not exist yet. So if we want to create a new column called 'MVtoFPL' we could start by confirming that it does not already exist in our dataframe... just to make sure that we're not overwriting anything

`'MVtoFPL' in players`

And then we simply use square bracket assignment on this new label and initiate it at 1.0

`players['MVtoFPL'] = 1.0`

Great. Let's confirm it was added

`players.head()`

And it was! Awesome. Now let's update the value for this column to be the ratio of market value and fpl values columns. The syntax is identical except now we're setting to a different value...(code)...and let's verify

`players['MVtoFPL'] = players['market_value'] / players['fpl_value']`

Beautiful. So we've added a new column to our dataframe. This is probably the most common approach that folks use to add columns, but there are also a couple of methods that are pretty decent alternatives to the above.

A clean approach which is actually my favourite is to use the insert method. And for this example, I'll slice a small piece of the dataframe and assign it to a new variable called df_mini.

```
df_mini = players.iloc[:4, 1:5]
```

Let's take a look at that.

```
df_mini
```

So let's assume we're working with this dataframe and we'd like to add in some names. I want to add a new column here containing the name of each player. I'll start by creating a series that contains the names, and we could use either a dict or a python list but let's go with a list here

```
player_names = pd.Series(['Bronson', 'Bradley', 'Ronald', 'Ronnie'])
```

```
player_names
```

Now, to insert this brand new column to our mini dataframe, we say

```
df_mini.insert(0, 'nicknames', player_names)
```

The beauty of this method is that it allows us to not only add a column but also specify exactly where it should be placed, as in the precise position it should take in the dataframe. So that's what this first parameter here controls. Very nice.

Another method we could use to add new columns is the dataframe assign(). This is a method that, as the name suggests, assigns new columns to an existing dataframe and returns a new copy. The syntax is a bit interesting in that it doesn't expect any parameters per se, but instead expects us to go ahead and specify our column names and values as keyword arguments, so let's do that

```
df_mini.assign(career_goals=[12, 67, 179, 46])
```

Here we are adding a new column for the total goals that each player has scored in his career. And that's it.

By the way, using this method we could add multiple columns at once, something that can't be done with insert. So if we go back and add another keyword argument, it will simply be added to the end of the dataframe

```
df_mini.assign(career_goals=[12, 67, 179, 46], nationality=['American', 'British', 'Turkish', 'Indian'])
```

So those are three ways to add columns to dataframes. When we cover merging and joining multiple dataframes together.. in a future section which I'm very much looking forward to.. we will learn and practice much more efficient ways of doing these operations at scale, but that's a future topic, let's not jump ahead of ourselves.

▼ **Lecture 39** - Adding Rows To DataFrames [**9:58**]

In the previous lecture, we saw three ways of adding columns to our dataframe. But how do we go about adding rows?

Let's continue working with our df_mini here for simplicity. And now, let's say that we want to add a fifth record containing information on a new player. The first approach we'll discuss is to use the append() method.

Append could take a dataframe, a series, or a list of these. For this example, I'll use a series to encapsulate all of this new player's attributes in one labeled sequence.

```
cristiano = pd.Series({ 'nicknames': 'Cristiano',
    'age': 32,
    'position': 'RW',
    'club': 'Juventus',
    'position_cat': 1,
}, name=4)
```

Let's confirm that all is looking good.

```
cristiano
```

Nice. So now to add this player all we need to do is call append on the df_mini.

```
df_mini.append(cristiano)
```

And that's it. Notice how the name of this series become the index label on that new record.

What if we wanted to add a bunch of players at once? The append method certainly supports that. All we need to is create individual series for each of them and then pass a list of those series to the append method, like so

```
# df_mini.append([player_1, player_2, player_3])
```

But instead of a list of series, as we have above, let's go with a slightly different approach here, and spin up a new dataframe to contain several players. I'll call it other_players

```
other_players = pd.DataFrame({
    'nicknames': ['Gianliugi', 'Lionel'],
    'age': [37, 32],
    'club': ['Juventus', 'Barcelona'],
    'position': ['GK', 'CF'],
    'position_cat': [4, 2],
}, index=[5, 6])
```

Let's take a look at what we just built...

```
other_players
```

Very nice. So a simple dataframe containing two rows, one for each player.

And finally let's append it to df_mini.

```
df_mini.append(other_players)
```

Beautiful. So we've seen a nifty way of adding single as well as multiple rows to an existing dataframe using the append() method.

Something you're very likely to encounter as you work with code written by other pandas users is what's known as setting with enlargement. This is something we saw before when we added new columns by assigning inexistent labels to a value. Right?

```
# setting with enlargement
# df['inexistent_label'] = 'new value'
```

So we could use the same exact technique to add a new row to our dataframe. In other words by simply assigning a value to a loc or iloc view which doesn't exist, we end up adding a new row to our dataframe. And it looks something like this

```
df_mini
```

In our current df_mini dataframe, we have indices going up to 3. If we say loc 9... which is a nonexistent index.... and then we assign that to a new value... we just end up adding a new row to our dataframe.

```
df_mini.loc[9] = 'new row'
```

Now, everything looks fine but this is actually something we should strive to avoid.

The mechanics appear to be very similar to what we saw in the previous lecture where we added columns, and that's because the syntax is very similar. Notice that in both cases we begin by indexing a non-existent key using square brackets and then we follow that with an assignment to a new value.

So this approach to adding rows appears very similar to how we added columns, BUT it is in fact much more inefficient. It's way worse than adding columns, and it is also a slower way of adding rows than the append method.

Adding rows to a dataframe in general is a pretty expensive operation, in that it's not efficient. And this has to do with how pandas stores dataframes in memory. So you could take that as a fact and be mindful of this when working with pandas.

Or if you're interested in knowing a bit more about the deeper reasons for why this is, I'll add a quick bonus lecture next to go over exactly this.

But if you're not into that, if it's been a long section already, feel free to skip it. I don't take it personally, we're still friends.

▼ **Lecture 40** - BONUS - How Are DataFrames Stored In Memory [**4:06**]

So we said that appending rows is very inefficient in pandas. Whether we're setting with enlargement using square brackets or even directly using the append() method to add rows, each and every time we do one of these operations, the entire dataframe is copied over in memory.

And the reason for this has to do with how pandas stores data in memory. Let me try to help you visualize this.

The first thing we should note is that pandas under the hood relies on numpy ndarrays which in turn rely on C arrays.

So what happens is that our beautiful players dataframe here, is first split into chunks of homogenous datatypes. Right? Data of the same kind.

players.info(verbose=False)

Our players dataframe has 3 64 bit float columns, 8 64bit int columns, and 6 variable length object columns. These are chunked together and stored in contiguous blocks of memory, in other words they're stored right next to each other.

So this approach to structuring memory is very much column-driven. We're grouping columns based on their datatype and then we're storing those chunks together.

Notice how there's no concept of a dataframe row at this level. Every time we index a row in pandas, pandas needs to reach back in memory, extract the relevant value from each block and reconstruct a row object. And this is all orchestrated by what's known as the block manager class in pandas. But there's clearly a lot involved, because the concept of a dataframe row does not really exist in memory.

So now think about what happens when we append a row vs a column. When we add a column, pandas first determines its type, then it identifies the block it should be added to, and then it modifies that block only. The rest remain largely unchanged

When adding a row, however, we have to touch each and every block, all of them need to be copied, modified and reassigned back to memory. And this needs to happen for every row that we append.

This might not seem like a huge deal if we're working with a small dataset and only inserting new rows once or twice, but as the size of the dataframe increases or the frequency of our insertions increases, I guarantee you that there will be a performance hit.

The big takeaway is that dataframes should be seen as collections of columns, not rows. So if there's more than one way to address your use case, always try to avoid working on individual rows because these operations will be slower for memory-related reasons.

I hope this has shed some light on what we discussed in the previous lecture.

https://www.dataquest.io/blog/pandas-big-data/

▼ **Lecture 41** - Skill Challenge [**1:25**]

1. From the players dataframe select 4 columns and 4 rows, of no particular order. Assign the resulting 4×4 dataframe to df_random.
2. Extend df_random 1) vertically by adding a new row, and 2) horizontally by adding a new column. Do this as two separate operations.
3. Compare the relative performance of the operations above. Is adding a row or column faster? Is there a significant difference?

▼ **Lecture 42** - Solution [**5:53**]

```
# 1
df_random = players.sample(4).sample(4, axis=1)

# 2
# to add row
df_random.append(pd.Series({}), name=7)

# to add column
df_random.assign(new_column=['a', 'b', 'c', 'd'])

# 3
%%timeit
df_random.append(pd.Series({}, name=7))

%%timeit
df_random.assign(new_column=['a', 'b', 'c', 'd'])

# even with this
df_random.insert(1, 'new column', ['a', 'b', 'c', 'd'])
df_random.drop('new column', axis=1, inplace=True)
```