Chapter6.2

So now, we're going to do some more things with strings,

because string manipulation is a lot of programs.

Number manipulation is one kind of program and

string manipulation is generally another thing that we do in programs.

So let's just... the +, we've been doing that,

where it sort of looks to the left,

looks to the right, concatenates.

Remember, there is no space in here.

If you say, like, print(x,y),

the two things come out and there's a space in between them.

But that's not what's going to happen here.

This + says concatenate these two things and it literally does it.

So if you want to put a space in between, you have to say, you know,

a concatenated with a space concatenated with

There and so we've explicitly put the space in.

So, string concatenation truly concatenates the strings together.

If we added the space automatically then you'd need a way to suppress that behavior.

So you'd need some other operation that concatenated strings without a space.

So we just say we've got one way of doing it,

and if you want a space, put the space in there.

The in, which is,

I love so much in the for loop,

is also usable as a logical operator.

The expression is a little bit different.

And so, instead of like, you know,

in, sort of variable in,

it's like this is asking that question.

And it's very much in a way like, you know,

double equals, which is a question, or not equals, you know,

these are leading back, giving us back True/False

or like less than or less than or equals, or something like that.

These are all questions.

Is this true? Yes or no?

And we use them in if statements.

And so, here we make the variable fruit,

and we ask the question: Is the string 'n' in fruit?

And so, in is an operator here.

It's like ==, but it's really looking through and saying, is the letter n in fruit?

And the answer is yes, it is,

'cause there's that n there,

and so we get back a True.

Is 'm' in the contents of the variable fruit?

The answer is no, so we get back a False.

And now it doesn't have to be a single character.

We can ask for a substring,

and say, is 'nan' inside fruit?

The answer is yes,

there is, and so we get a True back.

And so it's pretty smart.

It can scan, it finds whether or not these things are in there.

And we tend to build these things where we use them in if statements.

You know, if 'a' is in fruit,

then print Found it!

So in this case, that's an expression that evaluates to

True because a is in fruit and so this code executes and away you go.

Just a little note, if you're using the interactive interpreter,

and you're using – you actually have to throw a blank line here.

You don't need a blank line in real Python, like if you're writing in a file.

But, you know, if you type this and then you indent that, it works,

and then you hit enter here,

then it will actually run that whole block of code.

You have to give this blank line to convince it.

It's a situation where the interpreter, the chevron prompt,

is slightly different than a Python syntax in a Python script.

No biggie, but I bet by now you've probably figured that out.

OK, you can compare strings.

They make a lot of sense.

Equal sign, you know it just,

you know, compares character for character.

Less than and greater than have to do with

the character set of your computer

and the character set that Python is configured to use.

So less than, if you recall,

we did max and min,

and we learned that the uppercase letters are generally less than lowercase letters.

Right? So uppercase Z is less than lowercase a.

And that's going to happen if you do upper, greater than or less than.

Now, the thing that works,

for example, that is, you know, consistent,

is if you have something like Chuck with uppercase C and Glenn with uppercase G,

oops, and lowercase everything else, it's going to sort right,

because all the uppercase letters sort

the right way and the lowercase letters sort the right way.

And, but chuck and Glenn,

chuck and Glenn will sort the wrong way because the G is

going to sort before the c, and so...

But, it's ok. It makes some sense, it's all consistent.

But you can do this. But certainly == works just peachy fine.

So, those are the sort of the basic operations we can do,

but there's a whole bunch of additional capabilities

that are part of what we call the string library.

And it has to do with the idea that strings are objects.

And later, we'll learn what objects are and learn a lot of stuff.

Just for now, objects are these kinds of variables

that have capabilities that are kind of grafted onto or built into them.

And so, inside, once we put a string in greet,

Python knows that's a string.

If you use the type command,

it would say, oh, class str.

That str confers certain benefits and privileges that strings are

capable of doing that are different than what integers can do and

different than what files can do and other kinds of types.

Right? And one of the things they can do is,

you can say greet.lower().

Now, it's almost like saying,

like calling a function called lower() and passing greet into it.

But this is slightly different syntax.

This is, this is run a function lower() that's part of the string object,

of the string class,

that is going to give us back a lowercase copy.

So what this functionally does, it says,

make a copy of greet but all lowercase and return it to us,

and then we're going to store that into zap.

And so, if we print this out,

it's basically all lowercase.

And if we take a look at what's in greet,

we see the greet is unchanged,

because this was make a copy that's lowercase, a lowercase copy.

And so it doesn't change the original.

And even a constant is a legit object,

and we have it, a lower method inside of this.

So this just prints "Hi There".lower() which gives

us 'hi there' all lowercase. So we're good.

OK? And so even constants have this sort of built-in capability.

When we get there, we're going to call, you can look this up,

look at object method, method.

This thing is a method.

Or you can look ahead in an upcoming chapter and figure

out what methods are or you can just look on the Internet to figure out what methods are.

And we will cover this in much greater detail when we get there.

This is foreshadowing.

Now these are the dir and the type are things we've done before.

And so the type says it's a type str.

Class is an object-oriented term that

basically says this is a thing that's of the category string.

And now dir says,

what are strings capable of?

And there's actually a bunch of things. I don't show them all.

But these are a bunch of methods in the class str.

This is a light version of an object-oriented lecture.

And these are just the things you can do.

So it's stuff dot

blah blah blah ().

OK? replace, rfind, rstrip, they're all here.

There's a whole bunch of them.

And, this is, dir is not the best documentation for these things,

but Python, of course, has wonderful online documentation that explains it.

So they tell you what the parameters are,

and str is whatever variable like,

you know, x or y.

You know, and y.replace() blah old new,

y.rjust() for justification, or split.

We'll play with lots of these things.

So we'll take a look at some of the more common things that we

do with the string library. Capitalize,

which takes a string like, you know, abc,

and makes the first letter Abc,

or you could even have ABC as input and then output makes Abc capitalized.

Whenever it's done, the first letter's capitalized.

That's what capitalize does.

Why you want to do that?

Whatever. It's already built in.

You could write a for loop to do that,

but it's already built in,

so we'll take a look at some of these things.

One of the most common things that we do is use the find operator.

And it's kind of like in,

except that instead of returning True/False,

it returns where it found it.

So, the in says,

is the 'na' inside banana?

Or, find says, where inside the banana is 'na'?

So, we say fruit.find(),

it's a method within strings,

and we pass in 'na',

and then Python goes looking through here and says,

oh, there's an na right there,

starting in position 2.

Now, it doesn't say there's a bunch of them.

Later we'll figure out how if you really want to find a bunch of them,

you can call regular expressions. More foreshadowing.

But you find the first one and it comes back with a number 2.

So, the position is where the na is positioned within the banana is position 2.

But that's actually the third letter, so don't forget that.

If you look for something that's not there, no z,

you get back -1, so that's our little indicator,

or our flag that basically says, did not find it.

OK? So that's the find operation.

We already have played with the uppercase and the lowercase.

There is an upper that is effectively shouting.

Remember that greet doesn't change.

There's a lower that goes all to lowercase.

Sometimes, I tend to use these when I don't exactly know and I want to do an

if test to say if here's a string and here's a string and I want to ignore the case,

I say if the string to lower,

if the string lower is equal to the other string lower then I know that they're

both lower. That they match, ignoring case.

Search and replace.

So this is an example where we have,

you know, Hello Bob is in this variable.

And we're going to call the replace method inside of

the greet variable and give it two parameters.

In this case, we're going to give it an old and a new.

So that says go find all the Bobs and replace them with

Janes. It doesn't hurt greet, greet doesn't change,

it gives us back a copy and in that copy,

all those characters are replaced,

so the, what's in nstr is Hello Jane.

We can replace all the o's with X's.

So that goes here, and

it replaces that and replaces that.

Of course, greet's unchanged,

but then we get a copy of it with the o's replaced with X and put that in nstr,

and so that's where we get Hello Bob with X's instead

of o's. And it just shows that this is a sort of multi-replace.

It goes through and finds all of the o's and replaces them.

And there was only one Bob, but it would have found,

if there was more than one Bob,

it would have fixed all the Bobs and changed them to Janes.

Whitespace is something that we see.

The best way to think of whitespace is it's like spaces.

But there are other things that qualify as whitespace,

like newlines or tabs.

There's other characters that you'll find in strings,

especially if you start reading them from files,

and so they're sort of crufty bits.

The way that I think of whitespace is like here's

something printed out and it's like

abc def. Well, there's something here,

and you can't see it. It's like a clear letter. That's what whitespace is.

If this were a white piece of paper,

it would be whitespace.

It might be a tab, might be a bunch of spaces,

whatever, it's whitespace.

That's what whitespace means.

It affects mostly spaces but there are a few other characters that do it.

So here we have a string that's got spaces at the beginning and end.

And strip pulls off the characters from both the beginning and the ends.

Whoosh, whoosh.

It doesn't hurt the original variable,

it just gives us back a copy with nothing there.

And we can strip from the right side if we want and then we can strip from the left side.

And so that's a way of, you know,

otherwise, you'd be writing loops to get rid of the whitespace like,

oh, what if there are four characters or,

you know, four characters in the beginning and I'm going to throw them away?

Three characters in, throw them away.

I'd be writing a for loop,

do this other thing, concatenate these things together.

It's like you know, why didn't they write a library for it?

Ah, yes, they did. They did.

They wrote a library for it. And so, we're in good shape.

So, again, these libraries are

you've got to use them otherwise you'd be writing crazy loops.

We can ask it's a real common problem to be scanning

through a file and want to know only the lines that start with a prefix.

And there is a built-in method called startswith, line.startswith().

This takes a parameter,

what prefix we're looking for.

And in this case, we get True back because it does start with Please.

Does it start with a lowercase p?

And then we get back a False because, no,

it doesn't start with a lowercase p. So,

it is a True/False.

We tend to use that in the if... startswith(): do something to the line,

and that way we skip a bunch of lines

except the ones that start with the prefix that we're looking for.

Now, let's put some of this together. Some find and some slicing.

So slicing is the word for using that : operator. So let's take a look.

So, here is a big, long string.

And you're going to see a lot of these in the later chapters of the book.

We're obsessed with email messages in this class.

And so here is the first line of a bunch of email messages.

The format is the word From, space,

then an email address which includes a name and an @ sign which is the organization,

a space, and then a date and time which it was sent.

This is actually a real email message from a real person.

That's Stephen right there.

If you're ever in Cape Town, stop by UCT.

He's there, that's where he's at.

I've been to UCT and I said hi to Stephen.

People who've taken this course actually know Stephen.

That's crazy, right?

People who have taken this course are from South Africa and they know Stephen,

and they walk up and say, "Hey, Stephen!

You're in Dr. Chuck's lecture."

Yes, Stephen. You are in Dr. Chuck's lecture.

OK, but that's not the point.

We're learning Python. OK. So, what

I'm interested in is I want to extract this little bit from here to here.

I want to go one character after the @ sign up to,

but not including the next space.

So, we're going to take a couple of steps.

First, we're going to say, OK,

let's find the @ sign. Where is that?

Python goes and says, oh,

that's in position 21.

So it returns 21 back in there, and so we get 21.

That's the start. The character after that is where we want to start.

Now, the next thing I want to do is I want to say where is the next space after this?

Well, it turns out in find you can put up a second parameter in,

and say that's where to start.

So this is starting here and looking for a space and says,

oh, I just found you a space starting at the @ sign.

So down comes 31 into here.

We have basically 21 to 31,

which kind of boundary, that gives us the boundary.

And here's the fun part.

We want to slice this out.

So slicing is like chop, chop.

So we're going to go one beyond the @.

So that's going to give us the little u at position plus 1 through space position.

But it's not really space position. It's up to,

but not including the space position,

look how nice that came out, right?

So up to but not including the space position.

We get exactly what we want, not extra stuff,

and we get back the piece that we were trying to pull out.

So you'll see how we sort of

put these things together.

I must have written this same code 20,000 times in the last 30 years,

of search for something,

search for the start of something,

search for the end of something,

pull the thing out, search for the start of something,

search for the end of something, pull the thing out.

And we'll find that there are actually better ways to do this,

but this is kind of low-level,

sort of doing it the hard way in Python.

So this is just a little bit.

We're talking about Python 3,

but some of you may have to work in Python 2 from time to time.

And so, one of the real advantages of Python 3 is

that all the strings internally are what are called Unicode,

which means that they can represent a wide range of character sets.

In Python 2, strings sometimes have to go through conversions.

There are two kinds of things.

And so in Python 2, there were regular strings and Unicode strings.

And so, you would indicate a Unicode string by adding this u prefix.

And these were different.

And sometimes when you read these from files or wrote those to files,

you'd have to kind of go through some conversion and it was a little bit weird.

But the interesting thing is

that in Python 3 regular strings and Unicode strings are all just strings.

So, every string inside Python 3 is

capable of representing all character sets, and that's kind of cool.

There will still be some explicit conversion we'll have to do.

But the conversion in Python 3,

when we start talking to databases and reading data off of networks,

there will be conversions we'll have to do,

but those conversions will actually make

far more sense than the way you had to do it in Python 2.

So if you took my class in Python 2 and you're like,

"Here, use this buffer thing," and you say,

"Why should I use the buffer thing?"

I'm like, "Uh,'cause if you don't use the buffer thing, it won't work."

At least in Python 3, we have a sense of external data coming from outside your computer.

It needs to be dealt with in a certain way and it's quite predictable.

So Python 3 does a really much better job on character sets than Python 2.

It's possible in Python 2, but it wasn't as easy.

That's a quick run through strings.

We talked about the types and searching and looping.

There's a lot to it. We still haven't done anything

useful and that's what we're going to do in the next chapter.