

As promised, here are the slides for the Python Masterclass course re-master (that really is a mouthful).

August 2021 Update

This is an update to the progress made - This document is now 670 pages and growing fast. It's already a huge document.

Well, what a strange time we live in with the virus and the associated changes to our lives. We hope you and your family are safe and well and that life is ok for you.

We've just moved the new section 7 into place on Udemy - This is the Dictionaries and Sets section - a total of 59 videos are in that section as well.

In addition we've released section 13 - Big O notation. So check that out.

The slides for those sections are in this document.

We still have sections 8 through 12 to remaster. Expect regular updates.

Make sure you check out the work in progress section at the end of the course (section 14 currently) where you will always find the latest remastered content, and also note that only the slides for the remastered portions of the course are in this document. Why? Because I never created slides for the older videos, they just got “baked” into the videos. So as each video is re-recorded new slides are created and added to this document.

This means you can expect more frequent re-mastered content and more frequent slide releases! Remember that these slides only cover the remastered content because the old version of the content did not actually have slides. Well, we used them but did not save them if that makes sense.

What is being remastered?

Well first, some history.

I originally conceived the idea of this Python Masterclass in 2016 based on strong demand from my existing students.

It's been updated over the years to cover newer versions of Python, and we have even added content for older versions to help Python developers understand the differences which become really important if you

have to support legacy code in the future.

Like any educational material, over time bugs are found, things need updating, and combining this with what I have learned about producing courses and using the feedback I have received from students like you lead us to make a decision to re-record the entire course. It's got past the point when I can just release an update video here and there.

What is getting re-recorded?

We are literally re-recording each and every video in the course. While the content will be substantially the same, there will be improvements. Firstly to the video recording quality itself. One of the issues students found was the font size. I'm using much larger fonts.

Secondly, these slides (now part of many videos) help you understand key concepts.

Third, in the case of certain videos that did not adequately explain the concepts covered to the level I would have liked, they are being enhanced with additional clarification and explanation.

The end result will be a much-improved course that is really going to help you become a Python Programmer, better than ever.

I sincerely believe this is the best content we have ever produced, and we are looking forward to getting the content released. The fact that you are reading this is a sign that I have already released some of this new content on Udemy.

When will the remaster be completed?

Keep in mind that it will take us some time to get the new content completed. As explained in the early videos in the course, we are releasing the new content one section at a time. This is the best way to reduce confusion between old and new content.

As new content is released this document will be updated. We will be sending out a regular email to everyone who requested access to the slides.

How do you get notifications about new content and new slides?

We will be sending out regular updates to the email list you used to get access to these slides. Just watch for regular emails letting you know what content is available (both new videos and updated slides).

Are you a pirate? (no judgment).

If you happened to download these slides and/or my course from the Internet, keep in mind that what you have is likely very out of date (both slides and videos). I won't judge your actions other than to say that I have spent hundreds of hours producing and supporting this course, and if you find the content valuable I would appreciate you buying your own copy of it.

Please consider clicking this [link](#) to get your very own copy of this course at the cheapest price I can make it available.

Unfortunately, I have to do my part to make it harder for people to illegally download my content. This is my full-time job and the course also supports my co-instructors, three students who get paid to support students in the course as well as my other team helping me with the video content. If you've seen a picture of me you know I like to eat! Staying on this email list is the best way to be notified about new content.

Current version

The current version of this slides document is v1.034 created on the 04th of August 2021.

Programming Tips and Career advice

I have a Youtube channel where I post regular videos too. In the past twelve months, there have been 100 programming tips and career advice related videos released. Having been a programmer for 40 years I know a thing or two about programming and the industry so check out my playlist of [Programming Tips and Career Advice Videos.](#)

Summary

Well, that's it for me. If you want to drop me a line you can contact me at my blog.

Visit [My Personal Blog here.](#)

I hope you enjoy the course and these slides.

Regards

Tim Buchalka



Python Masterclass Remaster Slides

Main Course Slides.

COMPLETE PYTHON MASTERCLASS
Python Masterclass Remaster Slides



What is Python

Python is an object-oriented, interpreted language that's easy to use, and runs on many operating systems including Windows, Mac OS X, Linux and more.

It's named after the cult comedy show **Monty Python's Flying Circus** (not after the snake), and you'll find various references to **Monty Python** sketches in the official documentation.

What is Python

Python supports basic data types such as numbers and strings, as well as more complex types like lists and dictionaries, that can greatly simplify data processing.

Python also supports several programming paradigms, and can be used for Procedural Programming, Functional Programming and Object Oriented Programming.

Don't worry if those terms don't mean much at the moment - we'll be using all 3 approaches in the course.

What is Python

One thing I will be doing, throughout the course, is encouraging you to google a lot.

To get you started, search for programming paradigm after watching this video, to find out more about what those terms mean.

What is Python

Data in Python is strongly typed (so attempting to add a number and a string will give an error) but is also dynamically typed, so you're freed from worrying about variable declarations.

Despite its ease of use, it's also a very powerful language - it's one of the 3 "official" languages at Google, for example, and the creator of Python (Guido van Rossum) was employed there for 7 years.

He was allowed to spend half his time developing Python while he was employed at Google.

One thing to note is that the Python team decided to break backwards compatibility when creating Python 3.

What is Python

Although it's still very similar to Python 2, they no longer guarantee that Python 2 programs will work with the Python 3 interpreter.

There will be no further development of Python 2. Version 2.7 is the final version, and will no longer be supported after the end of 2019.

Python 3 is the way ahead, and we won't be discussing Python 2 very much, in this course.

Printing in Python

Jargon	Meaning
literal	<p>A value of some type.</p> <p>Examples of numeric literals are: 1, 42, 98.04</p> <p>Examples of string literals are: "Hello, World!", "Guido van Rossum", "Python"</p>
function	<p>A named block of code that we can call, by using its name.</p> <p>We can write our own functions, or we can use functions that are built into Python (such as <code>print</code>).</p> <p>In Python, all functions return a value.</p>
argument	<p>A value passed to a function, in order to give it values to work with.</p> <p>There may be no arguments, or there may be 1 or more.</p> <p>Arguments appear in parentheses after the function name.</p> <p>If there are no arguments, you still have to type the parentheses.</p>
calling a function	<p>Using the function name to execute the function's code.</p> <p>When you call a function, you have to provide the arguments that the function expects.</p> <p>If it doesn't expect any arguments, don't put anything between the parentheses.</p>
return value	<p>The value that a function returns.</p> <p>We haven't talked about that yet, but it belongs in this slide, to make the slide complete.</p>
parameter	<p>Also called formal parameter.</p> <p>This is something else we haven't discussed yet, and we'll learn about parameters when we write our own functions.</p>

Variables and Types

When a program's running, everything the program needs ends up stored in the computer's memory. The program code itself will be stored in one area of memory, and the data it works on will also be stored somewhere in memory.

A variable is basically just a way to give a (meaningful) name to an area of memory, into which we can place certain values.

Variables and Types

```
6     greeting = "Hello"  
7     name = input("Please enter your name ")  
8     print(greeting + name)
```

When we create the variable called **greeting**, Python allocates an area of memory for us.

It knows to refer to that area whenever we talk about the variable **greeting**.

Because we created **greeting** by attaching it to a string value, Python also decided that **greeting** was a variable of type string, and ensures that we can only do things with it that make sense for strings.

Variables and Types

There are a few rules for variable names:

- Python variable names must begin with a letter (either upper or lower case) or an underscore _ character.
- They can contain letters, numbers or underscore characters (but cannot begin with a number).
- Python variables are case sensitive, so **greeting** and **Greeting** would refer to 2 different variables.
- Variables are created when they are first attached to a value, using =.

Python Data Types

Python has several **built-in** data types, that can be classed as:

- **numeric**
- **iterator**
- **sequence (which are also iterators)**
- **mapping**
- **file**
- **class**
- **exception**

Python Data Types

We'll be introducing all of these as the course progresses, but in the remainder of this section, we'll have a look at the numeric data types, and one of the sequence types.

We've already seen the sequence type that I'll discuss in this section: the **str** type, for strings.

Before we look at that, I'll start with the Numeric data types.

Numeric Data Types

Python 3 has three numeric data types:

- **int**
- **float**
- **complex**

Python 2 had another type, **long**, because it's **int** type couldn't store very large values. In Python 3, the **int** type replaces **long**.

Numeric Data Types

We're not going to discuss complex numbers (which contain a real and an imaginary part, based on the square root of minus one).

If you understand complex numbers, and want to use Python to manipulate them, then by the end of the course you'll understand Python well enough to be able to do so.

Complex number theory is an advanced branch of mathematics/engineering that we're not going to attempt to explain.

Integer & Float Data Types

The Python integer data type is called **int**.

Integers are just whole numbers - numbers having no fractional part; whereas **float** is another name for real numbers - numbers having a fractional part after the decimal point.

Integers can be considered just special cases of real numbers - but when represented in a computer, computations using integers are significantly faster than using floating point numbers.

Integer & Float Data Types

That's one of the main reasons for the distinction in programming languages.

Because of the way that integers are generally stored in the computer's memory, many programming languages have a limit to the size of an integer - about 9 trillion in European terms, 9 quintillion in American. The Python 3 **int** effectively has no maximum size.

I'll repeat that, for programmers who are coming to Python from other languages: There's no limit to the size of the values that you can store in a Python **int**. You'll run out of memory before you exceed the size limit - because there isn't one.

Integer & Float Data Types

Floating point numbers, or floats, are used to represent numbers having a fractional part.

The Python floating point type is called **float**, and some examples of floating point numbers are:

- 1.0
- 123.456
- 3.14159265358979323846264338327950288419716939937510582097494459230781640
628620899862803482534211706798214808651328230664709384460955058223172535
940812848111745028410270193852110555964462294895493038196442881097566593
344612847564823378678316527120190914564856692346034861045432664821339360
72602491412737245870066063155881

Integer & Float Data Types

The maximum float value on a 64 bit computer is 1.7976931348623157e+308 which means move the decimal point 308 places right.

The smallest float is 2.2250738585072014e-308, which has 307 zeros before the decimal point.

Python floats have 52 digits of precision, which should be adequate for most purposes. If you need more precise decimal numbers, Python 3 now includes a **Decimal** data type, but we won't be using it in this course.

In a later section, I'll be showing you one trick you can use, to avoid rounding errors in financial calculations.

Integer & Float Data Types

Because Python doesn't have type declarations (where you specify the type of a variable before you can use it), it's tempting to think that you don't have to understand the difference between the **int** and **float** types.

But it **is** something you need to consider when writing your programs.

That will make more sense when we look at some of the operations that can be performed on numbers, so let's switch back to the IDE and do that, in the next video.

Operator Precedence Acronyms

- PEMDAS **P**arentheses, **E**xponents, **M**ultiplication/**D**ivision, **A**ddition/**S**ubtraction
- BEDMAS **B**rackets, **E**xponents, **D**ivision/**M**ultiplication, **A**ddition/**S**ubtraction
- BODMAS **B**rackets, **O**rder, **D**ivision/**M**ultiplication, **A**ddition/**S**ubtraction
- BIDMAS **B**rackets, **I**ndex, **D**ivision/**M**ultiplication, **A**ddition/**S**ubtraction

So what's wrong with those acronyms?

Well, they all have the same problem of being ambiguous. Students sometimes interpret them as meaning that multiplication has higher precedence than division, and addition has higher precedence than subtraction.

That's not the case. Multiplication and division have equal precedence. Addition and subtraction also have equal precedence.

In an expression that mixes operations with equal precedence, they're evaluated from left to right.

Let's see that working in Python.

Slicing a sequence

parrot[0:6]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	o	r	w	e	g	i	a	n		B	l	u	e

Our first slice was [0:6].

The slice starts at index position 0.

Slicing a sequence

parrot[0:6]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	o	r	w	e	g	i	a	n		B	l	u	e

It extends up to (but not including) position 6.

Slicing a sequence

parrot[3:5]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	o	r	w	e	g	i	a	n		B	l	u	e

The next slice was [3:5].

The slice starts at index position 3.

Slicing a sequence

parrot[3:5]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	o	r	w	e	g	i	a	n		B	l	u	e

The next slice was [3:5].

The slice starts at index position 3.

It extends up to (but not including) position 5.

Slicing a sequence

parrot[0:9]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	o	r	w	e	g	i	a	n		B	l	u	e

Then we had the slice [0:9].

This slice starts at index position 0.

Slicing a sequence

parrot[0:9]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	o	r	w	e	g	i	a	n		B	l	u	e

Then we had the slice [0:9].

This slice starts at index position 0.

It extends up to (but not including) index position 9.

Slicing a sequence

parrot[:9]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	o	r	w	e	g	i	a	n		B	l	u	e

We can leave off the start value of 0, because the start defaults to the beginning of the sequence.

The colon is necessary, otherwise we'd be specifying the single character at position 9.

Using a Step in a Slice

parrot[0:6:2]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	o	r	w	e	g	i	a	n		B	l	u	e

The slice starts at index position 0.

Using a Step in a Slice

parrot[0:**6**:2]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	o	r	w	e	g	i	a	n		B	l	u	e

The slice starts at index position 0.

It extends up to (but not including) position 6.

Using a Step in a Slice

parrot[0:6:**2**]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N		r		e									

The slice starts at index position 0.

It extends up to (but not including) position 6.

We step through the sequence in steps of 2.

Using a Step in a Slice

parrot[0:6:3]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	o	r	w	e	g	i	a	n		B	l	u	e

The slice starts at index position 0.

Using a Step in a Slice

parrot[0:**6**:3]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N	o	r	w	e	g	i	a	n		B	l	u	e

The slice starts at index position 0.

It extends up to (but not including) position 6.

Using a Step in a Slice

parrot[0:6:3]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
N			W										

The slice starts at index position 0.

It extends up to (but not including) position 6.

We step through the sequence in steps of 3.

Python Sequence Types

Python 3 has 5 sequence types built in:

- The string type
- list
- tuple
- range
- bytes and bytearray

What is a Sequence?

A sequence is defined as an ordered set of items.

For example, the string "**Hello World**" contains 11 items, and each item is a character.

A list is also a sequence type. For example, here's a Python list of things you might need, when buying a new computer:

```
["computer", "monitor", "keyboard", "mouse", "mouse mat"]
```

That list contains 5 items, each of which is a string.

We'll be looking at lists in a later section.

What is a Sequence?

That last example was a list of strings.

In other words, it's a sequence where each item is also a sequence.

Make sure you fully understand the videos on indexing, because indexing is very important when dealing with sequences.

What is a Sequence?

Because a sequence is **ordered**, we can use indexes to access individual items in the sequence.

For example, if we have:

```
computer_parts = ["computer", "monitor", "keyboard", "mouse", "mouse mat"]
```

then

```
computer_parts[1]
```

will be the string "monitor".

What is a Sequence?

That's also a sequence, and we can index into that as well:

```
computer_parts[1][0]
```

will be the letter "m".

What is a Sequence?

We've been looking at strings in this section, but everything that you can do with the **str** sequence type can also be done with the other sequence types.

Well, everything with one exception. Not all sequence types can be concatenated or multiplied. **range** is an example of a sequence that can't be concatenated.

String Replacement Fields

When printing strings and numbers, it would often be convenient to display both values using a single call to print. For example, we may want to print a description of what a value is, before the value itself.

We've seen that numbers **can't** be concatenated with strings using +, as the presence of a number instructs Python to attempt addition, and that fails.

One approach we could take is to coerce our numbers into a string, using the **str** function. In Python, every data type can be coerced into a string representation.

I mentioned that Java did this automatically, when an attempt's made to concatenate a string and a number, and the same's possible in Python.

Replacement Fields

In our code, we used 8 replacement fields - numbered from 0 to 7.

These are replaced with the values in the parentheses after `.format`.

The first value goes into `{0}`, the second into `{1}`, and so on.

```
print("There are {0} days in {1}, {2}, {3}, {4}, {5}, {6} and {7}"  
    .format(31, "Jan", "Mar", "May", "Jul", "Aug", "Oct", "Dec"))
```

Replacement Fields

That produces the output that we saw, when we ran the program:

```
print("There are {0} days in {1}, {2}, {3}, {4}, {5}, {6} and {7}"  
      .format(31, "Jan", "Mar", "May", "Jul", "Aug", "Oct", "Dec"))  
  
There are 31 days in Jan, Mar, May, Jul, Aug, Oct and Dec
```

Python 2 String Interpolation

Python 2 had another way of formatting strings, called **string interpolation**.

If you're familiar with **printf** in C, it works very similar.

It's useful to understand how it works, but it's not recommended that you use it for any Python 3 programs. That's because it's been deprecated, and will be removed from the language at some point.

Python 2 String Interpolation

I'm covering it here for 2 main reasons:

- You may end up working on a project that's still using Python 2.
- There's a lot of example code available on the internet, so it's useful for you to understand some of the Python 2 differences - especially when they're as easy to convert as the formatting operator is.

If you want to skip this video, and come back to it if you have to work with Python 2 code, then that's fine.

Summary

- In this section we've looked at running a simple Hello World Python program, before moving on to talk about variables.
- We then mentioned the different types of variables that are built in to Python, before discussing the number data types, int and float, and one of the sequence data types, str (for string).
- We've seen the basic operations that can be performed on numbers and strings, and looked at operator precedence.

Summary

- Indexing a sequence is incredibly useful, and we used the string type to learn how to do that.
- Next we saw how to use slices to extract sub-strings, and the string operators +, * and in.
- Finally, we looked at various ways to format strings, and how to print strings and numbers together.

Code Blocks

Like many programming languages, Python works in blocks of code.

One of the design principles of Python, though, was that it should be uncluttered and easy to read.

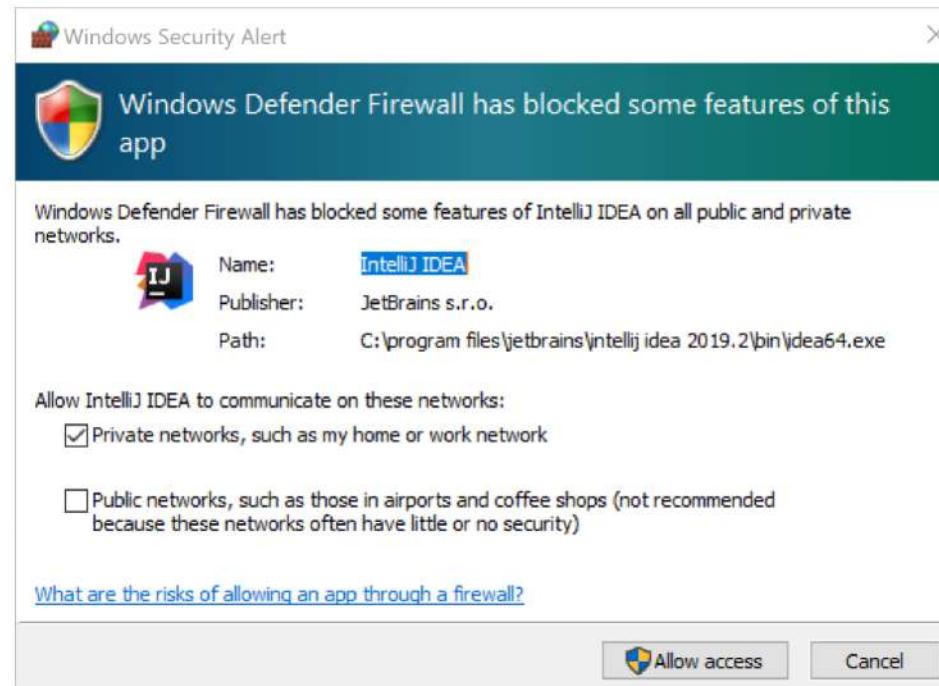
Because of this, Python doesn't use delimiters around blocks of code.

Code Blocks

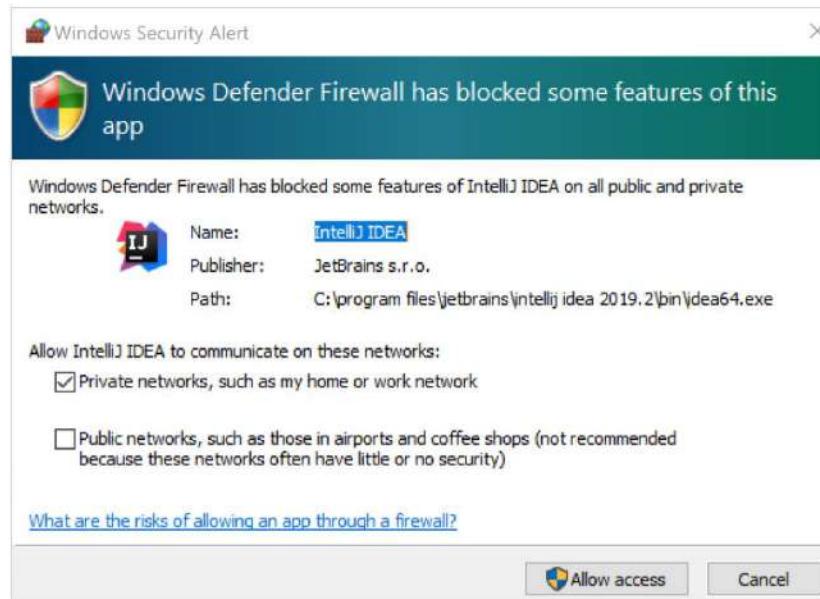
If you've used other languages, you may be used to seeing braces {} or BEGIN and END around code.

Python doesn't use anything like that, but instead uses indentation to indicate when a new block starts.

Using a Debugger in IntelliJ or Pycharm



Using a Debugger in IntelliJ or Pycharm



Make sure that box is ticked, then click Allow access.

Using a Debugger in IntelliJ or Pycharm

On a Mac, you may notice the little message that appears, at the bottom right. It suggests installing a Cython extension.

If you're using a different implementation of Python, you won't see that, but if are on a Mac and see that popup you can install it if you want.

if / elif / else

```
if guess < answer:  
    print("Please guess higher")  
elif guess > answer:  
    print("Please guess lower")  
else:  
    print("You got it first time")
```

An **if** statement begins with the keyword **if**, followed by a condition.

The **if** condition will always be evaluated.

if / elif / else

```
if guess < answer:  
    print("Please guess higher")  
elif guess > answer:  
    print("Please guess lower")  
else:  
    print("You got it first time")
```

Next, you can have one or more **elif** blocks.

You don't have to include **elif** - and we'll be seeing code that doesn't.

But if you have any **elif** lines then they come after the **if**.

elif also has to come before **else** - if there is an **else**.

if / elif / else

```
if guess < answer:  
    print("Please guess higher")  
elif guess > answer:  
    print("Please guess lower")  
else:  
    print("You got it first time")
```

Finally, you may have an **else** block.

You don't have to use **else**, but if you do, it must come after the **if**.

It must also come after any **elifs** - if there are any.

if Blocks

- A block can be quite complex, including further if and else blocks (and much more) contained within it.
- When testing for equality, we can't use a single `=` symbol. `=` is used for assigning values to variables, so when testing for equality we have to use `==`
- An if statement can include many **elif** parts, but there can be only one **else**. **elif** is short for "else if".
- The else, if there is one, must come after all the **elif** blocks.
- Duplicating code is generally a bad idea - there's almost always a better way.

Conditional Operators

When testing conditions, we can use any of the value comparison operators.

There are other types of comparisons we can perform, but we'll focus on these 6 for now.

Condition	Symbol
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	==
Not equal to	!=

Conditional Operators

The 6 value comparison operators are:

Condition	Symbol
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	==
Not equal to	!=

Once again, remember that we test for equality using 2 = symbols. That's something you'll forget at first - but you'll get a syntax error if you use a single =, which should remind you.

Comments

Comments are ignored by the Python interpreter, and are there to document your code. In this case, the comments prevent the code from being executed.

Generally, you should use them to explain why you've done something in a certain way, or as a reminder of certain conditions or variable states that may not be obvious.

```
# if guess < answer:  
#     print("Please guess higher")  
#     guess = int(input())  
#     if guess == answer:  
#         print("Well done, you guessed it")  
#     else:  
#         print("Sorry, you have not guessed correctly")  
# elif guess > answer:  
#     print("Please guess lower")  
#     guess = int(input())  
#     if guess == answer:  
#         print("Well done, you guessed it")  
#     else:  
#         print("Sorry, you have not guessed correctly")  
# else:  
#     print("You got it first time")
```

Comments

Remember that you may come back to modify code, weeks or months after you wrote it.

Documentation can be extremely helpful when you work on another programmer's code for the first time, or come back to your own code after a gap.

```
# if guess < answer:  
#     print("Please guess higher")  
#     guess = int(input())  
#     if guess == answer:  
#         print("Well done, you guessed it")  
#     else:  
#         print("Sorry, you have not guessed correctly")  
# elif guess > answer:  
#     print("Please guess lower")  
#     guess = int(input())  
#     if guess == answer:  
#         print("Well done, you guessed it")  
#     else:  
#         print("Sorry, you have not guessed correctly")  
# else:  
#     print("You got it first time")
```

Comments

In Python, comments start with a `#` symbol, and can appear on a line of their own, or at the end of a line of code.

You can't place them in the middle of a line, though, and `#` inside a string is treated just like any other character, and doesn't indicate a comment.

If you want a comment to span several lines, then start each line with a `#`.

```
# if guess < answer:  
#     print("Please guess higher")  
#     guess = int(input())  
#     if guess == answer:  
#         print("Well done, you guessed it")  
#     else:  
#         print("Sorry, you have not guessed correctly")  
# elif guess > answer:  
#     print("Please guess lower")  
#     guess = int(input())  
#     if guess == answer:  
#         print("Well done, you guessed it")  
#     else:  
#         print("Sorry, you have not guessed correctly")  
# else:  
#     print("You got it first time")
```

Using and

Let's say you were asked this question:

Would you like an uncomfortable seat **and** an ice cream?

If you answer yes to that question, you may enjoy the ice cream, but you won't be very comfortable.

You've agreed to **both** parts of the condition.

Using or

On the other hand, the question may have been phrased as:

Would you like an uncomfortable seat **or** an ice cream?

That allows you to choose either one of the options.

This analogy falls down slightly, because in English there's an implication that you can only have one or the other.

In programming languages, **both** conditions could be True.

Let's see the possible results of evaluating expressions using **and** compared to using **or**.

and Truth Table

and Truth Table		
	True	False
True	True	False
False	False	False

To find out the result of using **and** with 2 conditions, read the value at the intersection of the 2 values that are being anded together.

The next 4 slides show the results for each of the combinations of True and False.

and Truth Table

and Truth Table		
	True	False
True	True	False
False	False	False

True and True = True

and Truth Table

and Truth Table		
	True	False
True	True	False
False	False	False

True and False = False

and Truth Table

and Truth Table		
	True	False
True	True	False
False	False	False

False and True = False

and Truth Table

and Truth Table		
	True	False
True	True	False
False	False	False

False and False = False

or Truth Table

or Truth Table		
	True	False
True	True	True
False	True	False

The **or** truth table works in the same way, but the results are different of course.

To find the result of using **or** with 2 conditions, read the value at the intersection of the 2 values that are being ored together.

or Truth Table

or Truth Table		
	True	False
True	True	True
False	True	False

True or True = True

or Truth Table

or Truth Table		
	True	False
True	True	True
False	True	False

True or False = True

or Truth Table

or Truth Table		
	True	False
True	True	True
False	True	False

False or True = True

or Truth Table

or Truth Table		
	True	False
True	True	True
False	True	False

False or False = False

Simplify Chained Comparisons

When comparing conditions using **and**, Python will stop checking as soon as it finds a condition that is **False**.

When comparing conditions using **or**, it will stop as soon as it finds one that is **True**.

Boolean Values

A **Boolean** value is a value that can either be true or false.

Those are the only 2 values that a Boolean can have.

True and False

Python defines 2 constants:

- True
- False

Note that they're written with a capital letter. All names, in Python, are case sensitive.

True has a capital **T**, and **False** has a capital **F**.

Boolean Expressions

A Boolean expression is an expression that evaluates to either **True** or **False**.

We've seen a few Boolean expressions - things like

`age >= 16`

for example.

Boolean expressions aren't restricted to just testing numbers. We could also test a string value:

`name == "Tim"`

is a Boolean expression that will evaluate to True if the value of **name** is the string Tim. It will evaluate to False if **name** has any other value.

Boolean Expressions

Boolean expressions can be very complex.

For example, you might have a condition such as:

```
if day == "Saturday" and temperature > 27 and not raining:  
    print("Go swimming")
```

In that example, **raining** is a Boolean variable. It has the value True or False.

not is used to reverse the value.

not

We can use **not** to reverse a Boolean value.

not True is False.

not False is True.

Looping

Python provides several ways to repeat a block of code - things like:

- **for** loops
- **while** loops
- list comprehensions and generator expressions

In the next few videos, we'll look at **for** loops.

We'll cover **while** loops after that, but we'll be leaving **list comprehensions** and **generator expressions** for a later section.

for Loop

We've seen a **for** loop in earlier examples.

Now it's time to look at them in more detail, and explain what was happening in that code.

A **for** loop works by iterating over some set of values.

It assigns each of the values, one by one, to one or more variables.

It then executes a block of code once for each value.

for Loop

The set of values comes from a **sequence**, or some other **iterable** object.

We've seen one **sequence** type - the **string** type. I'll use that for our first examples.

An **iterable** object is anything that can be iterated over. That means a **sequence** is also an **iterable**.

In simple terms, if you can use it with a **for** loop, then it's **iterable**.

I'll be introducing one **iterable** type - the **range** - in some of the following examples.

Stepping through a for loop

In that case, we should examine each character in the **number** string, to check if it's a digit or not. As we saw in the previous video, a **for** loop can be used to do that.

We're going to iterate over **numbers**, and append anything that isn't a digit to the **separators** string.

Once we've seen the code working, I'll step through it in the debugger, so we can see what it's doing.

Stepping through a for loop

The first step is to initialise our **separators** variable.

This is something that causes a lot of confusion, and I'm going to spend a minute discussing it, once we've seen the code working.

Iterating over a range of values

In those languages, you provide a starting value and an ending value, and increment a variable each time round the loop.

Python's different approach to **for** loops makes them incredibly powerful and also very flexible.

We can get the same effect as C **for** loops, by iterating over a **range** of values.

Iterating over a range of values

Remember how our string slices returned the characters up to **but not including** the end number?

Ranges work in the same way - the last value specified is **not** included.

range produces a range of numbers - from the starting value, up to **but not including** the end value.

Nested for loops

Nesting a for loop within another for loop is a powerful way to process data. In fact, it's something that's done a lot.

It sounds like something special, but it's just one block of code inside another block.

Let's see an example, using for loops to generate times tables. These were used to teach children multiplication, and normally went up to 12.

Nested for loops

There are another couple of things we need to know about **for** loops, and we'll look at them in the next few videos.

We've seen the basics of how a **for** loop works, to iterate over some sequence.

The sequence can be a string or a range, but Python also has other sequences that we can iterate over.

continue

Sometimes you may need to interrupt the normal flow of a loop, to either jump out of it completely, or stop the current iteration and move on to the next one.

In the next 2 videos, we're going to look at the **continue** and **break** statements that we can use with loops.

continue

We need some examples to show how these statements can be useful, so I'm going to start by introducing **lists**.

We'll be looking at them in more detail in the next section. This is a very quick introduction to what a list is.

A **list**, in Python, is an ordered sequence of values, enclosed in square brackets.

For example, we might have a shopping list.

while loop

Python provides two main ways to loop round a block of code; **for** loops and **while** loops.

We've seen how a **for** loop can be used to iterate through an iterable. In our examples, we used the items in a sequence and the numbers in a range.

Sometimes we need to keep looping as long as some condition is True, and stop when it becomes False.

We do this using a **while** loop.

while loop

The basic form is:

```
while <condition>:  
    execute block of code
```

The condition can be anything that can evaluate to True or False.

As long as the condition is True, the code in the loop will be executed.

When the condition becomes False, the loop terminates.

More on while loops

One of the important features of **while** loops is that they can be used when you can't determine, in advance, how many times you will need to loop.

A **for** loop will repeat for each item in a pre-determined sequence, whereas with a **while** loop you don't need to know how many times the loop will execute.

One particular application of this is reading data from a file, or an internet stream.

It's difficult (not impossible, but difficult) to know in advance how much data there is to read.

A **while** loop can be used to keep reading, until there's no more data left.

We'll see this soon when we look at file I/O (input and output) in Python.

The random module and import

There are a number of problems with this game.

One problem is that it only allows two guesses.

If we'd wanted to allow more than two guesses, we'd have to nest more and more if tests, every time the player guessed incorrectly.

The other problem is that **5** is always the correct number, which makes the game a bit boring (not that it was terribly engaging to begin with).

The random module and import

The first problem can be solved using a **while** loop, which is what you'll be doing in the challenge, next.

Before we do that, though, let's digress slightly and address the second problem, so you can produce a more interesting game.

To fix the problem of the correct number always being 5, we need to get the computer to generate a random number in the chosen range.

That's 1 to 10 in the example, but we can extend that.

The random module and import

Python has a lot of features built in, such as the ability to do arithmetic; lists, ranges and tuples; and more.

There's enough built in, in fact, for us to be able to write a random number generator if we wanted.

However, one has already been written for us, and comes included with Python, in a module called **random**.

The random module and import

We'll be looking at creating modules and packages, much later in the course. But for now, we're just going to use the one that comes with Python.

Because it's in an external module, we need to import it into our program, using an **import** statement.

When you want to use objects from the Python Standard Library, you import them - usually at the start of your program.

Importing a module allows you to use all the objects from that module, in your own code.

Challenge Solution

You may have written the code differently - there's often several ways to get the same results.

And if you struggled to complete the challenge, don't worry - the important thing was to have a go.

You'll find my solution makes more sense, if you tried to solve the challenge yourself, first.

You'll also understand **why** what you were trying didn't work.

Binary Search

Trying to guess a number between 1 and 1000 is difficult - the guess will be wrong 999 times out of a thousand.

What if we could narrow that range down, so we're only guessing a number from 500 numbers?

That improves our chances, now we'd only be wrong 499 times.

If we reduce that further, to 250 numbers, we'd only be wrong 249 times.

Binary Search

That's how a binary search works.

When you have an ordered set of data to search through, you can split the data in half each time.

Guessing a number is very much like searching for a value in an ordered set of data.

Of course, we know that the value we're guessing does exist.

That simplifies things slightly, but the basic steps are the same.

Binary Search

500	500	Guess 1
250	250	Guess 2
125	125	Guess 3
62.5	62	Guess 4
31.250	31	Guess 5
15.625	15	Guess 6
7.8125	7	Guess 7
3.90625	3	Guess 8
1.953125	1	Guess 9

Our first guess is 500, which is half of 1000.

Binary Search

500	500	Guess 1
250	250	Guess 2
125	125	Guess 3
62.5	62	Guess 4
31.250	31	Guess 5
15.625	15	Guess 6
7.8125	7	Guess 7
3.90625	3	Guess 8
1.953125	1	Guess 9

Our first guess is 500, which is half of 1000.

If we have to guess lower, we halve 500 and guess 250.

Binary Search

500	500	Guess 1
250	250	Guess 2
125	125	Guess 3
62.5	62	Guess 4
31.250	31	Guess 5
15.625	15	Guess 6
7.8125	7	Guess 7
3.90625	3	Guess 8
1.953125	1	Guess 9

Our first guess is 500, which is half of 1000.

If we have to guess lower, we halve 500 and guess 250.

For the third guess, we halve 250 and guess 125.

Binary Search

500	500	Guess 1
250	250	Guess 2
125	125	Guess 3
62.5	62	Guess 4
31.250	31	Guess 5
15.625	15	Guess 6
7.8125	7	Guess 7
3.90625	3	Guess 8
1.953125	1	Guess 9

Our first guess is 500, which is half of 1000.

If we have to guess lower, we halve 500 and guess 250.

For the third guess, we halve 250 and guess 125.

Halving 125 gives a floating point value, 62.5.

Binary Search

500	500	Guess 1
250	250	Guess 2
125	125	Guess 3
62.5	62	Guess 4
31.250	31	Guess 5
15.625	15	Guess 6
7.8125	7	Guess 7
3.90625	3	Guess 8
1.953125	1	Guess 9

We're only guessing whole numbers, and we've seen that Python's integer division rounds down.

The middle column shows the integer values we'd use, instead of the floating point values that we get when dividing by 2.

Binary Search

500	500	Guess 1
250	250	Guess 2
125	125	Guess 3
62.5	62	Guess 4
31.250	31	Guess 5
15.625	15	Guess 6
7.8125	7	Guess 7
3.90625	3	Guess 8
1.953125	1	Guess 9

By continuing to half the range of values that we have to guess, we can guess any number from 1 to 1000 in a maximum of 10 guesses.

Binary Search

500	500	Guess 1
250	250	Guess 2
125	125	Guess 3
62.5	62	Guess 4
31.250	31	Guess 5
15.625	15	Guess 6
7.8125	7	Guess 7
3.90625	3	Guess 8
1.953125	1	Guess 9

The table only shows 9 guesses.

It's showing what happens when we have to guess lower each time, to show how the range of values halves with each guess.

Binary Search

500	500	Guess 1
250	250	Guess 2
125	125	Guess 3
62.5	62	Guess 4
31.250	31	Guess 5
15.625	15	Guess 6
7.8125	7	Guess 7
3.90625	3	Guess 8
1.953125	1	Guess 9

The table only shows 9 guesses.

It's showing what happens when we have to guess lower each time, to show how the range of values halves with each guess.

If the answer was 2, we'd have to have another higher guess, after we'd guessed 1.

That would give 10 guesses, which is the most we need, in order to guess a number from 1 to 1000.

We may need fewer guesses, but shouldn't need more than 10.

Binary Search

500	500	Guess 1
750	750	Guess 2
875	875	Guess 3
938.3755	938	Guess 4
969.6875	969	Guess 5
985.34375	985	Guess 6
993.171875	993	Guess 7
997.0859375	997	Guess 8
999.04296	999	Guess 9
1000.02148	1000	Guess 10

Of course, sometimes we'll have to guess higher, not lower, but the principle's the same.

When guessing higher, we use the upper half of the range, instead of the lower half.

Binary Search

500	500	Guess 1
750	750	Guess 2
875	875	Guess 3
938.3755	938	Guess 4
969.6875	969	Guess 5
985.34375	985	Guess 6
993.171875	993	Guess 7
997.0859375	997	Guess 8
999.04296	999	Guess 9
1000.02148	1000	Guess 10

Our first guess would still be 500.

For the second guess, we'd split the range from 500 to 1000, giving 750.

Binary Search

500	500	Guess 1
750	750	Guess 2
875	875	Guess 3
938.3755	938	Guess 4
969.6875	969	Guess 5
985.34375	985	Guess 6
993.171875	993	Guess 7
997.0859375	997	Guess 8
999.04296	999	Guess 9
1000.02148	1000	Guess 10

Our first guess would still be 500.

For the second guess, we'd split the range from 500 to 1000, giving 750.

If we have to guess higher again, the third guess splits the range from 750 to 1000, to give 875.

With each guess, the range of numbers we have to guess is halved.

Binary Search

500	500	Guess 1
750	750	Guess 2
875	875	Guess 3
938.3755	938	Guess 4
969.6875	969	Guess 5
985.34375	985	Guess 6
993.171875	993	Guess 7
997.0859375	997	Guess 8
999.04296	999	Guess 9
1000.02148	1000	Guess 10

In reality, we'll use a combination of higher and lower guesses.

For example, let's say the answer was 998.

After our 9th guess (999) we'd have to guess lower.

The range of numbers to guess is between 997 (our 8th guess) and 999. So we'd guess 998, and get it right with the 10th guess.

Binary Search

Let's work through a real example.

To keep the slides manageable, we'll be guessing a number between 1 and 10.

We should be able to do that within 4 guesses.

Let's say the answer is 7.

Binary Search

Low										High
1	2	3	4	5	6	7	8	9	10	

We start off by guessing the mid-point between the low and high values.

The formula we use is:

`low + (high - low) // 2`

Binary Search

Low				Mid					High
1	2	3	4	5	6	7	8	9	10

Putting our low and high values into the formula, we get:

$$1 + (10 - 1) // 2$$

Which is 1 plus 4, giving 5. Remember that integer division rounds down, and $9 // 2$ is 4
5 is our first guess.

Binary Search

Low				Mid	Low		Mid		High
1	2	3	4	5	6	7	8	9	10

We're told to guess higher. That means the answer must lie somewhere between 6 and 10.

We know it's greater than 5, so the lowest it can be is 6. That's our new low value for the range.

The mid-point now becomes $6 + (10 - 6) // 2$ which is 8.

Our next guess will be 8.

Binary Search

Low				Mid	Low	High	Mid		High
1	2	3	4	5	6	7	8	9	10

We're told to guess lower. That means the answer must lie somewhere between 6 and 7.

We know the answer's less than 8, so the largest it can be is 7. That's our new high value for the range.

The mid-point now becomes $6 + (7 - 6) // 2$, which is 6 (integer division rounds down).

Our third guess will be 6.

Binary Search

Low				Mid	Low	L / H	Mid		High
1	2	3	4	5	6	7	8	9	10

7 is greater than 6, so we're told to guess higher.

Our low value now becomes 7, the same as the high value. At this point, a human would guess 7 right away. The computer has to do the calculation first:

The mid-point now becomes $7 + (7 - 7) // 2$ which is 7.

Our fourth guess will be 7, and we get the correct answer.

Binary Search

Low				Mid					High
1	2	3	4	5	6	7	8	9	10

Low		Mid		High
6	7	8	9	10

Low	High
6	7

L / H
7

Each incorrect guess halves the range of numbers that we have to guess.

That's why this technique is known as a "binary search" or "binary chop".

Binary Search

The binary search algorithm is the most efficient way of finding an item in an ordered list.

It forms the basis for the data structures that database programs, such as SQLite and Oracle, use for storing the database keys.

That allows very fast searching of the data.

In the next video, we'll write the code to guess a number, using a binary chop.

Algorithm

An **algorithm** is a set of steps that are followed, to perform a task or calculate a result.

Hi Lo game

It will be useful to count the number of guesses. We know that the computer should be able to get the correct answer with 10 guesses, at most.

That's because we're halving the range of numbers each time, and 2^{10} is 1024.

That's slightly more than 1000. In fact, the computer can guess any number from 1 to 1023 within 10 guesses.

But 1023 is a strange number to show to the user, so we'll stick with 1000.

Testing the Hi Lo game

This is quite a hard program to test.

A simple typing error could result in the program never guessing correctly, and we may not detect that unless we choose a number that causes the problem.

Before I would be completely happy with this code, I'd probably test it for all possible numbers - so I'd think of each number from 1 to 1000 in turn.

So the next 35 hours of video will be me pressing **h** and **l** over and over again. 😊

Of course not - that's not a very good use of our time, especially when we have computers to perform repetitive tasks for us.

Testing the Hi Lo game

Later in the course, we'll get the computer to test this code for us. But we need to understand about functions first, so we'll leave it till later.

One thing you might want to do, is run the program in the debugger. I won't do that, but go ahead and do so if you want.

The reason I won't do it, is because there's only 2 values that we'd be interested in, when stepping through the code.

The loop already pauses each time round, because it waits for our input. Printing out the values of low and high is much easier than using the debugger, in this case.

Augmented Assignment

IntelliJ and PyCharm highlight errors in our code. They also show warnings sometimes, to indicate possible problems with our code, even if it compiles successfully.

The third things it can flag up are suggestions. These are ways that we can improve our code, and you should generally pay attention to them.

That doesn't mean you have to accept them - it's your code, after all - but most of the time, the suggestions will improve your code.

Augmented Assignment

This one says **Assignment can be replaced with augmented assignment.**

I touched on that briefly, in the previous section. I mentioned that I'd explain it more later, so let's see what it's all about.

Augmented Assignment

Augmented assignment sounds really fancy, but it's just a shorthand way of assigning values to a variable.

The Python documentation describes them as:

"the combination, in a single statement, of a binary operation and an assignment statement",

which is true but not terribly understandable.

Augmented Assignment

27  guesses = guesses + 1|

On line 27, we perform an addition.

We take the value of **guesses** and add **1** to it.

We then bind the variable **guesses** to the result.

guesses becomes 1 greater than it was.

The **addition** is the binary operation (binary because it takes 2 operands to work on) and then there is an assignment.

Augmented Assignment

If you were programming in Java or C++, IntelliJ wouldn't have suggested this change.

That's because, in those languages, both statements are identical. Their use is just a matter of preference, or style.

In Python, however, the augmented assignment form only evaluates the assignee (**guesses** in our example) once.

In the other form, where **guesses** appeared twice, the variable is evaluated once each time it appears. That's less efficient.

Augmented Assignment

Now, I'll admit that the evaluation isn't very complicated here.

After all, it's not like we're doing some complex calculation. Something like **12 * 850 + 13 million / 18** is obviously going to take a bit of time to work out.

Evaluating a Variable

What do I mean by evaluating the variable?

Evaluating a variable basically consists of looking up its value. That's pretty fast, and probably takes a small fraction of a millisecond.

But doing it twice, when you could do it just once, is obviously less efficient.

If you do that in a loop, going round a million times, you could be wasting a second or more.

Efficient code

That may not sound bad, unless your code's trying to land an aeroplane.

A jumbo jet can travel at over 600 miles per hour. At that speed, 1 second represents a sixth of a mile, or about 268 metres.

Of course, if your code's trying to land an aeroplane without slowing it down first, then you've got more serious problems than missing an augmented assignment 😊.

But you get the point.

Augmented Assignment

There's another reason why augmented assignment is more efficient.

```
27      # guesses = guesses + 1
28      guesses += 1
```

With our first form, on line 27, Python creates a new variable.

It binds that new variable to the result of performing the calculation, then destroys the original variable.

Augmented Assignment

Using an augmented assignment, it can perform the operation in-place where possible, modifying the original variable.

This isn't always possible, depending on the type of object (variable) in question, but will be done in place whenever possible.

Python Enhancement Proposal

A **PEP** is a Python Enhancement Proposal.

All changes, and new features, to the Python language start with a proposal.

The proposal is reviewed, discussed, improved if necessary, then either accepted or rejected.

Until 2018, Guido van Rossum made the final decision about whether to implement a PEP or not.

Since then, the decision is taken by a group of people.

You can see who they are by looking at **PEP 0013**, and I'll open the link to <https://www.python.org/dev/peps/pep-0013/> in a new tab.

Refactoring Code

It wouldn't be too hard to change them in this short program, but in a larger program they could be appearing all over the place.

If you forgot to change one of them, or made a typing error, you'd cause an error in the code. That's not good, and is why you shouldn't refactor code just to conform to a style guide.

But I'm going to break that rule here, because I want to show you how your IDE can make that a lot easier - and safer.

We're going to **refactor** the code, to make it comply with PEP 8.

Refactoring

Refactoring code means changing its structure, without changing its behaviour.

Refactoring might change **how** the code does things, but doesn't change **what** it does.

Refactoring

Most modern IDEs will have a feature that allows you to rename things in your code, and that's much safer than doing it manually.

As I said, if you miss one or make a typing error, you'll break the code.

Make sure you test your code after refactoring it. Run the program to make sure it still works.

Refactoring

The code in this course will follow the PEP 8 guidelines - most of the time.

I'm re-recording the videos for this course, and some of the early examples were written a long time ago.

You may find that some of them don't follow PEP 8.

You'll also find code in the Python libraries (and elsewhere) that doesn't follow those guidelines either.

The code that you write should follow the guidelines, and I'll refer you back to them, as we cover various new Python features.

Binary Guess

In the binary search slides, earlier, we worked through what happens when we try to guess the number 7.

We were guessing in the range 1 to 10. I'll quickly review those earlier slides.

COMPLETE PYTHON MASTERCLASS
else in the Hi Lo Game



PROGRAM FLOW CONTROL IN PYTHON
PM-39-1

Binary Guess

Low				Mid						High
1	2	3	4	5	6	7	8	9	10	

We start off by guessing the mid-point between the low and high values.

Our first guess is 5.

Binary Guess

Low				Mid	Low		Mid		High
1	2	3	4	5	6	7	8	9	10

We're told to guess higher. That means the answer must lie somewhere between 6 and 10.

The new mid-point is 8, which becomes our second guess.

Binary Guess

Low				Mid	Low	High	Mid		High
1	2	3	4	5	6	7	8	9	10

We're told to guess lower. That means the answer must lie somewhere between 6 and 7.

Our third guess will be 6.

Where things get interesting, is in the next slide.

Binary Guess

Low				Mid	Low	L / H	Mid		High
1	2	3	4	5	6	7	8	9	10

7 is greater than 6, so we're told to guess higher.

The low value becomes equal to the high value.

Binary Guess

Low				Mid	Low	L / H	Mid		High
1	2	3	4	5	6	7	8	9	10

When we went through this example earlier, I said that at this point, a human would guess 7 right away.

I also said that the computer has to do the calculation first, but that's not completely true.

Rather than use our formula to calculate the new mid-point, the computer could test to see if `low == high`.

Binary Guess

Low				Mid	Low	L / H	Mid		High
1	2	3	4	5	6	7	8	9	10

If low does equal high, the computer doesn't have to ask us if the guess is correct.

When low and high are equal, it must have the correct number.

Binary Guess

Low				Mid	Low	L / H	Mid		High
1	2	3	4	5	6	7	8	9	10

If low does equal high, the computer doesn't have to ask us if the guess is correct.

When low and high are equal, it must have the correct number.

Note that this is only the case because all numbers are present.

In a normal binary search, when what we're looking for may not be present, this doesn't apply.

Binary Guess

Low				Mid	Low	L / H	Mid		High
1	2	3	4	5	6	7	8	9	10

We can make our program a bit more impressive, by not asking if the value is correct.

We know it must be, when `low == high`, and we can tell the player that we've guessed their number.

The changes to our code are quite simple, so let's get started.

else in the Hi Lo Game

That's a bit more impressive. It would be really good if we could get the computer to do that for any guess, but we can't.

It only knows that it has the correct answer when **high** and **low** converge to the same value.

With a high value that's close to an exact power of 2, that happens about half the time.

1000 is close to 1024, and the computer can tell it has the correct answer for 489 numbers that we think of.

else in the Hi Lo Game

If you'd like to run this code in the debugger, to help understand what's happening, you'll find it a bit frustrating.

It can be annoying debugging code that waits for input.

Setting breakpoints can help a bit, but there's another technique you can use.

Conditional Debugging

Running our high-low game in the debugger can be a bit slow.

The loop is only going round 10 times, but if we had a much larger range of numbers, you'd soon get bored clicking the Step Over button.

When I ran the program in the last video, things only got interesting when the computer guessed 99.

It would be really good if we could get the debugger to stop at that point, when guess is 99.

Conditional Debugging

Luckily, we can do just that.

I'll show you how to do that with the IntelliJ and PyCharm debuggers.

But if you're using a different IDE, your debugger should have a similar feature.

Unfortunately, the IDLE debugger doesn't have conditional breakpoints.

If you're using IDLE, you'll have to step over your code in the usual way.

Summary

In this section, we looked at blocks in Python, and saw how the level of indentation is crucial in determining which block a line of code belongs to.

We then used **if** and **else** to test conditions, and only execute blocks of code if those conditions were either **True** or **False**.

We created more complex conditions, and saw how to use **elif** to check several conditions in turn.

Summary

Next, we moved on to execute blocks of code repeatedly, using loops.

The first type of loop we used was the **for** statement.

It's used to iterate through a range of values.

for is used when you know, in advance, how many times you want to go round the loop.

Summary

Next, we looked at **while** loops.

They're used to loop as long as a condition is True.

while loops are generally used when you don't know, in advance, how many times the loop should execute.

Summary

We covered **continue** and **break** to interrupt the normal flow of a for loop, and **else** to define a block that always executes if the loop finishes without break.

This is a different use of the **else** keyword to how it's used in an **if** statement - which is unfortunate and can be confusing, but that's the keyword that was chosen.

Summary

A couple of videos weren't related to Python loops.

We digressed to look at **augmented assignment**, and why you should use it in preference to a normal assignment.

We also discussed **PEP8** and the style of your Python code.

I'll finish this section with a short challenge, to give you a chance to practice what you've learnt in the course so far.

You'll have a chance to make use of the string formatting that we covered in previous videos, as well as reading in input that a user types at the keyboard.

You'll also get chance to practice using if to execute code, depending on a condition.

Thinking like a Programmer

One thing you'll find, when you learn to program, is that the way you think changes.

COMPLETE PYTHON MASTERCLASS
Optional Extra Solution



PROGRAM FLOW CONTROL IN PYTHON
PM-44-1

Thinking like a Programmer

One thing you'll find, when you learn to program, is that the way you think changes. It happens slowly, over time, as you work with more and more code.

Thinking like a Programmer

One thing you'll find, when you learn to program, is that the way you think changes.

It happens slowly, over time, as you work with more and more code.

You'll automatically start looking at things in different ways, and considering different options to solving a problem.

Sequences

A **sequence** is an ordered collection of items.

The word **ordered** there is important.

If a sequence wasn't ordered, you couldn't refer to individual items by their index position.

Sequences

In an earlier example, we indexed the string "Norwegian Blue".

That's an ordered sequence of characters. The letter **B** is at index position 10, and

`"Norwegian Blue"[10]`

will always return **B**.

Sequences

When you iterate over a sequence - using a **for** loop, for example - you'll always get the items in the same order.

That may seem obvious, so I won't labor the point.

Iterables

In Python, anything that you can iterate over is an **iterable**.

That means that if you can use it in a **for** loop, then it's **iterable**.

That's not a very "scientific" description, but it'll help with the videos in the rest of this section.

Iterables

The accurate definition of an iterable is that it's an object that contains either an `__iter__` method
or an `__getitem__` method

But that won't mean much at the moment.

Indexing must also start from zero.

Iterables

All sequence types can be iterated over.

That means all sequence types - strings, lists, etc - are also iterable types.

Not all iterables are sequences.

For example, you can use a dictionary in a **for** loop, but it's not a sequence.

Iterables

Don't worry too much about this.

I mention it because the documentation talks about sequences and iterables.

At the moment, the only iterable types we're looking at are sequences.

But be aware that other types can also be used, when the documentation mentions **iterable** rather than **sequence**.

Lists

OK, that's all been revision so far. It's not surprising that indexing and slicing works the same with a list as it did with a string, because they're both **sequence** types.

But there's one big difference between strings and lists: strings are **immutable**, which means they can't be changed.

Lists, on the other hand, are **mutable**. You can change the contents of a list.

Immutable objects

When an object is described as **immutable**, that means it can't be changed.

The following **immutable** types are built into Python:

- int
- float
- bool (True and False) : a subclass of int
- str (string)
- tuple
- frozenset
- bytes

Immutable objects

It should be obvious that an int like **5** can't be changed.

5 is always **5**. You can add another value to it. For example, we can add **1** to **5** to get a new number: **6**.

But we can't change **5**.

Immutable objects

The same's true of floats. 3.14159 will always be 3.14159.

Once again, we can perform arithmetic with a float, and get a new float value.

But we can't change the value of **3.14159**.

Stay with me on this, because there might be an obvious question that you're asking.

I'll explain that next, but at the moment, trust me that these values are immutable.

Immutable objects

bool values are also immutable, and so are strings.

The question you're asking is probably becoming disbelief now. So let's see some code that proves this.

Immutable objects

The documentation doesn't state that it should be the object's address, just that it must be **"guaranteed to be unique and constant for this object during its lifetime."**

The CPython implementation **does** return the object's memory address, but not all Pythons will do that.

The ID for an object may be different each time you run the program, but while your program's running, the object will have the same id.

Of course, if Python has to destroy the object and re-create it, then its ID will change.

That gives us a good way to tell if an object is changed, or if Python has to create a new object.

Mutable objects

A **mutable** object is one whose value can be changed.

Python has the following **mutable** objects built in:

- list
- dict
- set
- Bytearray

We'll be looking at dictionaries and sets in the next section.

We can change the value of mutable objects, without the object being destroyed and re-created.

Mutable objects

Now that you've seen the same thing done with a **mutable** object, and have seen the different behavior, the previous video should make more sense.

Strings are **immutable**. When we tried to change a string, Python created a new object - a new string - and re-attached the name to it.

Lists are **mutable** - they **can** be changed. When we appended a new item, Python was able to change the contents of the list, without creating a new one.

Mutable objects

So that's the difference between mutable and immutable objects.

An **immutable** object can't be changed. You can create a new object, and use the same variable name for it, but you **can't** change the value of an **immutable** object.

Mutable objects, such as lists, **can** be changed.

Methods and Functions

I used the term **method** there.

A method is the same as a function, except that it's bound to an object.

That means we need an object, in order to call the **method**.

Methods and Functions

We used a few **functions** in the last video: we used **min** and **max**, and the **len** function.

When you call a function, you just type its name, and provide any arguments in parentheses.

For example **len(even)** gave us the number of items in the list called **even**.

Methods and dot notation

When we call a **method**, we tell it which object it's called on.

In other words, which object it should be using when it performs its function.

For example, the entry we're looking at, in the table, is

s.append(x)

dot notation

`s.append(x)`

means that we'll be appending `x` to a list called `s`.

We pass `x` as an argument, so that the method knows what to append.

But we don't have to use an argument to tell it which list to append to, because that comes at the start, before the **dot**.

dot notation

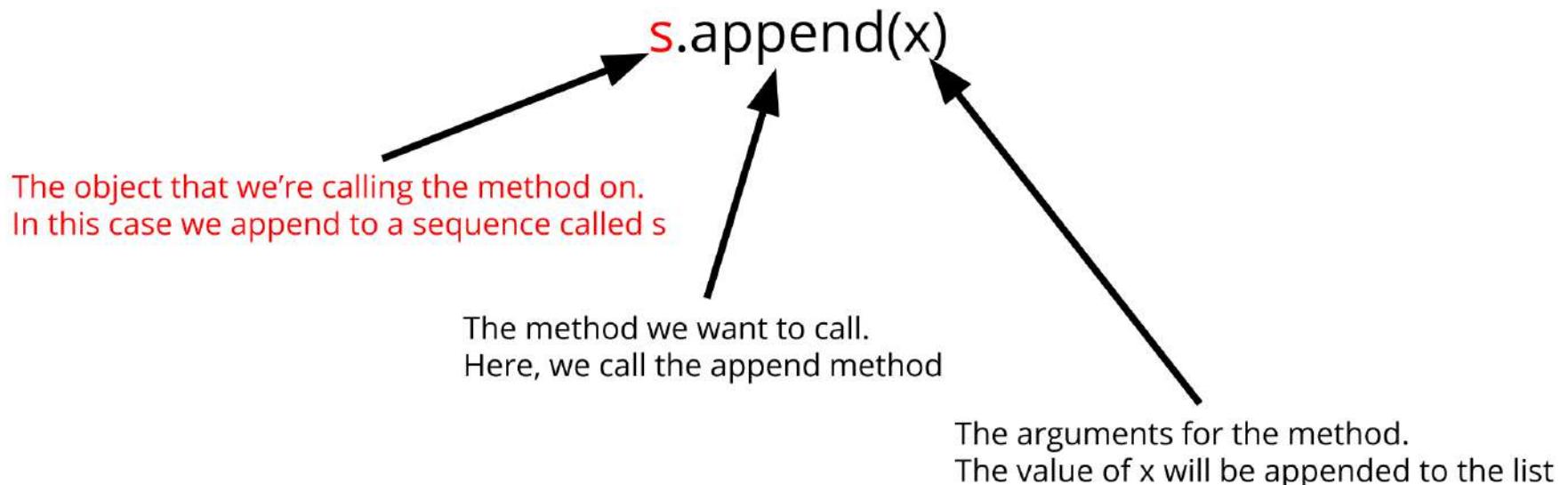
Dot notation's used a lot in Object Oriented Programming, and we'll learn more about it in the OOP section.

Before we learn how to write our own methods, we'll be using a lot of methods that Python's built-in objects have.

The syntax is simple. You start with the object you're using, then a dot, then the name of the method.

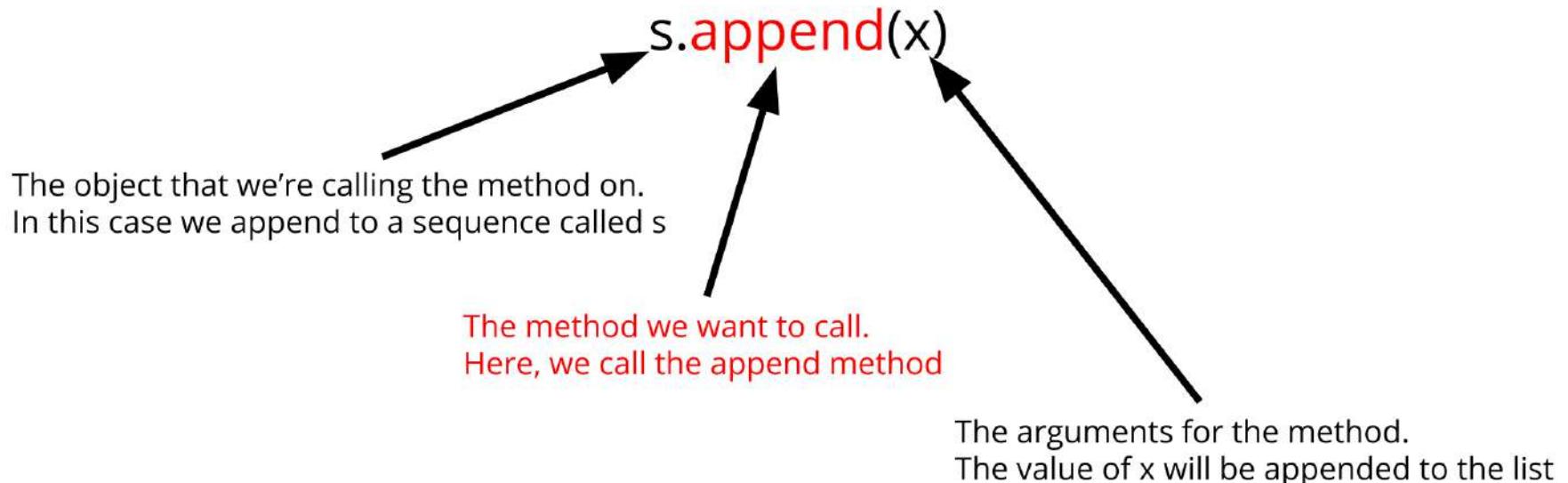
If the method needs arguments, you put them in parentheses after the method name.

dot notation



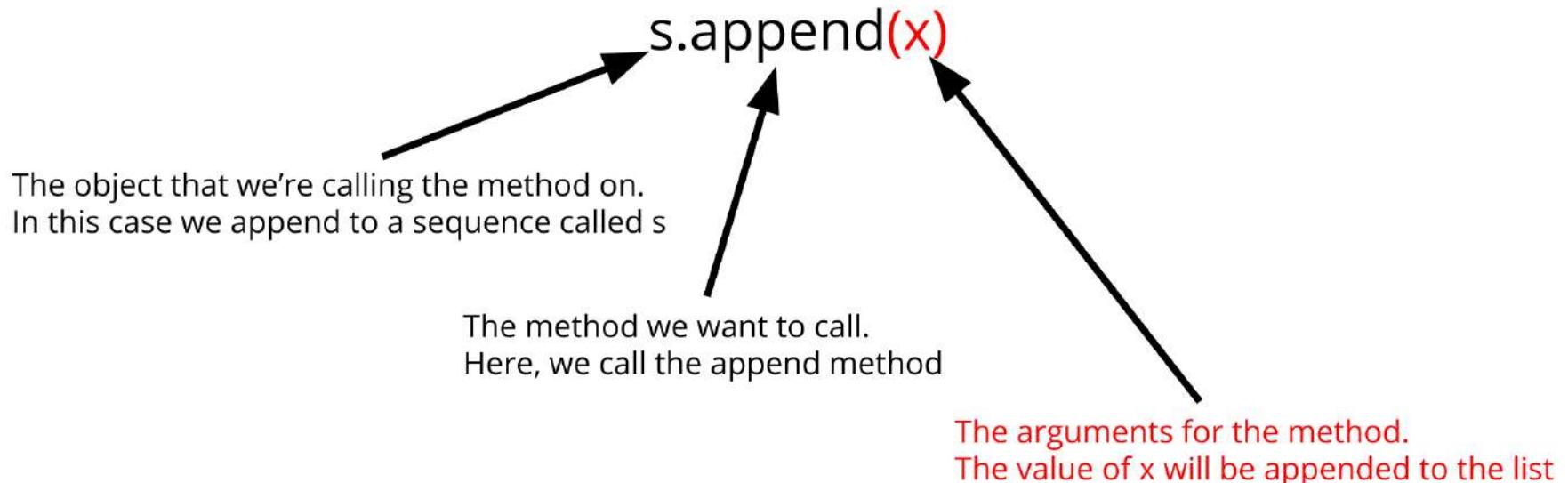
We start with the name of an object. The documentation uses `s`, because it's talking about **sequence** types.

dot notation



After the dot, we have the name of the method that we want to call.

dot notation



If the method needs any arguments, they appear in parentheses after the method name.

The parentheses are still needed, even if there are no arguments. That's the same as when you call a function.

Enumerate

If the list was sorted, we could use a binary search.

But our list isn't in order.

Python has to check each item, to see if it equals the value of part.

The first part will be found straight away.

Enumerate

The second part will take a bit longer.

Python will check if **monitor** equals **computer**, then go on to check the second item in the list.

That matches, so it can return 1 as the index value.

If there are hundreds or thousands of items, finding the index positions will take a while.

This is a very common thing to have to do, and Python provides a much more efficient way of doing it.

That's the **enumerate** function.

enumerate returns each item, with its index position.

Test, Test and Test. Then Test Again!

In the last couple of videos, we wrote some code that stripped outlying values from an ordered data set.

It seemed to work, but we've only tried it with one set of data.

We can't release this code for use, until we've tested it more thoroughly.

Thinking of How to Break your Code

When you test your code, you're trying to think of ways to break it.

I know! You've spent hours, sweating over your code. The last thing you want to do is break it. You don't want it to break!

But if **you** don't break it, **someone else** will. If you've got a test team, that's great. They'll break your code for you 😊

It's their job to find bugs in our code, but that doesn't mean we should throw poor quality code at them.

The very least we should do, is test our code with different data, to be reasonably confident that it works.

Edge Case

How do you choose the data to test with?

You'll come across the term **edge case** in software testing.

Special Values

There are also special values that you should consider.

When your code performs division, for example, you should make sure that it can cope if it's asked to divide by zero.

In our code, there's a special value that we should consider - similar to zero, when dividing.

How will our code behave, if we give it an empty list?

It should work, but we won't know for sure until we test it.

Corner Cases

There are also **corner cases**.

A **corner case** is like an edge case, but when there's more than one parameter involved.

For example, how should you handle dividing, when **both** values are zero?

Anything divided by zero is undefined, but anything divided by itself is **one**.

Corner Cases

If you're dealing with ratios, and both values are zero, then you've got the same amount of each.

You may want to handle that differently, rather than returning "undefined".

It's very important to consider things like that, and make sure your code copes correctly.

At the very least, it shouldn't crash or produce unexpected results.

Test Cases

So, what are the cases we need to consider?

We should test our code with data that meets the following criteria:

- Outlying values at both the low and high ends.
- Outlying values at the low end only.
- Outlying values at the high end only.
- No outlying values.
- Only outlying values (no valid ones).
- An empty data set.

More to Test

We've already tested the first case - when we have outliers at both ends of the data. That leaves five more cases still to test.

Removing Rogue Values

Removing rogue values from an unsorted list can be done with less code than our previous example.

We've seen that we can't remove items from a list, while iterating **forwards** over it. If the size of the list changes, things go wrong.

When an item's removed from the list, all the later items are shuffled down, to fill in the gap.

That messes up the index numbers, as we work **forwards** through the list.

Removing Items from a List Backwards

But what would happen if we worked **backwards** through it, instead?

If we remove an item at position 10, all the items above that would shuffle down. But the indexes from 9 down to 0 wouldn't be affected.

In an earlier video, I said that you should be very careful, when changing a list that you're iterating over. One way to be careful, is to iterate **backwards**.

Iterating backwards is a valuable technique, and allows the size of your sequence to be changed without causing a problem.

Algorithms Performance

We've had to use two loops, where our other programs did it with only one.

This code also only works with sorted data. The other loops could delete values from unsorted lists, as well.

Obviously, there must be an advantage. There is; this code is significantly faster.

The difference is so great, that I want to demonstrate it.

Algorithms Performance

To do that, I'll have to use a load of stuff that we haven't covered yet. One of the easiest ways to time code, is to put it into a function.

And we haven't covered functions yet.

We also haven't covered how to time code. We need to cover **imports** first.

There's a lot we need to learn, before this next bit of code will make sense.

Algorithms Performance

Our first algorithm is more complicated. But it takes advantage of the fact that the lists are sorted, and only performs two deletions.

I created that first example, to show you **how** to delete a slice from a sequence.

This performance test has demonstrated **why** that can be useful.

Deleting a range of values, in one go, can be a lot more efficient than deleting them one-by-one.

Algorithms Performance

Ok, I'll finish this video with a warning. If you want to run this code on your computer, make sure you've got enough RAM.

If you've got less than 16 Gigabytes of ram, **don't** put one hundred million items in your lists.

You'll see the difference, even with smaller lists. Just be careful not to run out of memory!

If you run out of memory on Linux, the system will kill the process for you.

If you do that on Windows, the program will crash with a memory error.

Your computer will probably become very unresponsive, for a while.

Summary

In this section we introduced the **list** sequence type.

Summary

Lists are **mutable**, unlike strings, which is another sequence type. The contents of a list can be changed, without creating new references to the same object.

We saw the difference between immutable and mutable objects, using slides.

We looked at **indexing** and **slicing** in lists, and also **common sequence operations** such as min, max, index, count and len.

These operations can be used with both mutable and immutable sequence types.

append is a method that can only be used with a mutable sequence type - such as a list - to add new values.

Summary

A **for** loop, that iterates over the items in a list, is a convenient way to process the data in your lists.

If you also want to use the index positions of items, the **enumerate** function is an efficient way of getting the index positions, when iterating.

We looked at updating items in a list, and removing items.

Summary

The **sort method** for lists can be used to sort the contents, in place, without creating a copy of the list.

The **sorted** function sorts a sequence, and returns a list.

A **keyword argument** can be added to specify how items in a list should be compared, such as making them case-insensitive, for example.

Summary

There are various ways to create lists, which include using a **list literal**, **slice**, **concatenation** and **list comprehensions**.

Also, the **sorted function** will sort any iterable or literal object and return a new list, leaving the original list unchanged.

The **list** function can be used to create a list from any iterable, with the items from the string appearing in the same order. It's also a way to copy a list.

Something to be careful with, when iterating over an object, is changing the size of it.

The example we used was removing the outlying numbers from an ordered list of experimental data.

Summary

Testing your program thoroughly is vital. You need to test for all the scenarios in your code, to make sure there are no bugs.

Make sure you test **edge cases** and **corner cases**: An **edge case** is "a problem or situation that occurs at an extreme operation parameter", and a **corner case** is similar, but when there's more than one parameter involved.

We then moved on to removing items from sorted and unsorted lists, by iterating backwards.

The **reversed** function is one of the ways of doing this, with the advantage that we can use **enumerate**.

Summary

The final thing we've looked at was **performance**.

We compared three methods for removing items from a list, and saw that deleting a slice is more efficient than deleting the individual items.

More on Code Style

I said **bracket** there, because I was using it in its generic sense. This formatting applies whether your construct uses parentheses, square brackets or curly braces.

You'll see curly braces when we look at sets and dictionaries, later.

Whenever we introduce a new structure in the course, check back in this Style Guide, to see what it has to say about code style for that construct.

I often include the first item just after the opening bracket, on the same line. You'll find that done in this style guide, for function arguments.

It's not done in these two examples, but it is acceptable.

More on Code Style

Note, though, that subsequent items should line up with the first one. It's not acceptable to include a value immediately after the opening bracket, then indent subsequent lines by only four spaces.

I'll show an example of what I mean by that, in a moment.

Google always put the first item on its own line. But I have a limited amount of space on the video, and starting on the same line as the opening bracket lets me show one extra line of code on screen.

Function Signatures

The term **function signature** means the definition of a function.

That includes the function's name, and the parameters that it defines.

Function Signatures

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

The name of the function is **print**.

The name's followed by parentheses, with the parameters inside the parentheses.

Not all functions have parameters, but the parentheses are still needed when defining and calling a function.

Function Signatures

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

The first parameter for the **print** function is defined in a strange way. **objects** has an asterisk before it.

That means you can provide zero or more values. We've usually only provided one value - or none, when we wanted to print a blank line.

I'll demonstrate **print** being used with several values, when we get back to our code.

Keyword Arguments

```
print(*objects, sep='', end='\n', file=sys.stdout, flush=False)
```

Next, the **print** function defines the first of several **keyword** arguments.

They're also called **named arguments**, and you'll hear me use that term as well.

We've used keyword arguments, when we reversed the sort order, and again when we provided a key to the **sorted** function.

print defines 4 of these in total. The first one is **sep**.

keyword arguments are useful, because you can give them a default value.

If you don't provide a value for **sep**, it defaults to a space.

Keyword Arguments

```
print(*objects, sep=' ', end='\\n', file=sys.stdout, flush=False)
```

The **end** keyword argument defaults to the newline character **\\n**.

We haven't been providing an argument for **end**, which is why we get a newline each time we call `print`.

Keyword Arguments

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

I won't discuss the other two keyword arguments just yet. They'll make sense when we look at writing data to files on disk.

Summary

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

So that's what **function signature** means.

It's the name of the function, and the parameters that it defines.

We'll look at all this in more detail, when we come to write our own functions.

You'll learn more about keyword arguments, and you'll also learn exactly what that asterisk is doing.

For now, let's go back to the documentation, to see what those parameters let us do.

Generator Expressions

I'm not going to cover them yet - they're covered in a later section. You've got enough new things to learn, at the moment, without worrying about more advanced Python features.

That can be frustrating for experienced programmers, who are taking this course to add Python to their portfolio of languages. So I'm going to quickly demonstrate a generator expression.

If you're new to programming, don't worry about understanding this code. It will all make sense when you've worked through the videos, and get to the section on comprehensions and generator expressions.

This is another of those few times when you don't have to type in the code. Once again, sit back, relax, and marvel at the power of Python 😊.

Tuples

A **tuple** is a mathematical name for an ordered set of data.

If you remember, from the introduction to this section, being **ordered** is a requirement for a Python sequence.

In Python, tuples are another sequence type; along with strings, lists and ranges, that we've already seen.

Tuples differ from lists because tuples are **immutable**. That means they can't be changed after they're created - just like strings.

Tuples

Python 3.8 added **Assignment Expressions** to the language - in PEP 572. They use what's informally referred to as the **walrus operator**.

We'll cover them later, but be aware that you'll get strange results, if you use a tuple after the **walrus** operator, and don't enclose it in parentheses. If the code compiles, it probably won't do what you expect.

The Python style guide, PEP 8, doesn't have much to say about parentheses around tuples.

The only guidance is when defining a tuple with only one item, which we'll see soon, in another video.

But as I said, you may find it easier to always enclose your tuples in parentheses, to avoid having to remember when they **are** necessary.

Tuples

Alright, that's a quick introduction to the syntax of tuples.

It hasn't been very exciting, and you may be wondering what's the point of them.

They seem to be just like a list, but with parentheses instead of square brackets.

We will find out why they are useful in upcoming videos.

Tuples are Immutable

Alright, that's the main difference between a **tuple** and a **list**. **Tuples** are immutable.

Because they don't have the overhead of the methods needed to change them, tuples use less memory than lists. That's one reason why you might want to use a tuple.

If your program's dealing with millions of these things, you can save memory by using a tuple.

COMPLETE PYTHON MASTERCLASS
More on Tuples



LISTS, TUPLES AND RANGES
PM-38-1

Tuples are Immutable

There are another couple of reasons for preferring a tuple over a list. Both reasons are due to tuples being immutable.

The first reason is to protect the integrity of your data.

We've just seen that attempting to change the title of our Metallica album gave an error, because we can't change tuples.

Let's try to do that with a list.

COMPLETE PYTHON MASTERCLASS
More on Tuples



LISTS, TUPLES AND RANGES
PM-38-2

Advantages of Tuples: Number 1

Using a tuple, for things that shouldn't change, can help to prevent bugs in your programs.

When we tried to alter the tuple a few minutes ago, the program crashed. Obviously that's not good, but it's something that we'd spot when testing the code.

When we used a list, the program didn't crash, and we ended up with incorrect data. That could be far worse than the program crashing.

I know this example is only the year that an album was released, but what if we were displaying the medicines that could be used to treat a certain disease?

A doctor, relying on our program, might prescribe something totally unsuitable - with possibly fatal consequences.

Advantages of Tuples: Number 1

So that's one reason why you might prefer a tuple over a list.

If your code attempts to change one of the fields in a tuple, your code will fail. That will highlight something that your code probably shouldn't be doing.

Advantages of Tuples: Number 2

The second advantage of tuples, is that you can always unpack them successfully.

Because a tuple can't be changed, you always know how many items there are to unpack.

That's why this is described as **unpacking a tuple**, even though you can unpack any sequence type.

More Unpacking

First, we're hoping that this extra bit of video will really help you to remember about unpacking tuples.

Also remember that you **can** unpack values from a list, as long as you're sure that the size of the list won't be changed.

In fact, Aatif's data was in lists. It's lists that are being unpacked in this code.

His example code used literal values, but the real data was coming from a disk file.

A CSV file will always contain the same number of items on each row, so it's fine to unpack the lists.

More Unpacking

The second reason is, that's it's useful to see techniques being used in real code.

We're not sure what the purpose of that code is, but it looks like Aatif is processing coordinates in a 3 dimensional space.

It's a real project, anyway, not just a contrived example.

A lot of that code probably won't make much sense, because we haven't covered dictionaries and comprehensions yet, but it does show unpacking a list in a real application.

More Unpacking

And finally, I've used this example because of that strange syntax in that for loop, on line 2.

```
2     for key, (x, y, z) in newDict.items():
```

Notice that x, y and z are enclosed in a set of parentheses.

That leads nicely into dealing with nested data structures - such as a tuple stored inside a list, or inside another tuple.

Google is your Friend

Google for those 2 terms, **homogeneous** and **heterogeneous**, if you're still not sure what they mean.

Deciding What To Use

We sometimes get asked how you can decide whether to use a list or a tuple, and that's one way to decide.

Another criteria to help you decide which one to use, is whether you want to add new items to it, or not.

You can't append to a tuple because tuples are immutable. If you're going to be adding new items, then a list is more suitable than a tuple.

It's quite likely that we'd want to add a new album to our **albums** list. Our program could be reading the albums from a disk file, or asking a user to type them in.

It's far less likely that we'd want to add another item to each of those tuples, while the program's executing. Which is a good thing, because we can't.

Deciding What To Use

Sometimes, your initial design might be wrong. In this case, I've forgotten to add the songs to our album tuples.

When that happens, it's fine to modify your tuples in your code. You can't do it while the program's executing, of course, but you can change your program if it doesn't do exactly what you want.

Nesting Further

We might want to store a list of the songs on each album, too.

If we do that, each item in the list would become a tuple, holding the name, artist, year, and list of songs.

We might end up with a tuple, containing a list that also contain tuples.

Or maybe, a list containing tuples, that also contain lists.

Our basic data will be something like this next slide:

Our Albums Data

I haven't included all the songs for each album, but we've got enough to see how we can represent the data.

Album	Artist	Year	Songs
Welcome to my Nightmare	Alice Cooper	1975	1: Welcome to my Nightmare 2: Devil's Food 3: The Black Widow 4: Some Folks 5: Only Women Bleed
Bad Company	Bad Company	1974	1: Can't Get Enough 2: Rock Steady 3: Ready for Love 4: Don't Let Me Down 5: Bad Company 6: The Way I Choose 7: Movin' On 8: Seagull
Nightflight	Budgie	1981	1: I Turned to Stone 2: Keeping a Rendezvous 3: Reaper of the Glory 4: She Used Me Up
More Mayhem	Imelda May	2011	1: Pulling the Rug 2: Psycho 3: Mayhem 4: Kentish Town Waltz

Our Albums Data

You need to decide how you're going to store the songs. Would you use a list, or would you use a tuple?

Have a think about that, pause the video and come back when you've made a decision.

Album	Artist	Year	Songs
Welcome to my Nightmare	Alice Cooper	1975	1: Welcome to my Nightmare 2: Devil's Food 3: The Black Widow 4: Some Folks 5: Only Women Bleed
Bad Company	Bad Company	1974	1: Can't Get Enough 2: Rock Steady 3: Ready for Love 4: Don't Let Me Down 5: Bad Company 6: The Way I Choose 7: Movin' On 8: Seagull
Nightflight	Budgie	1981	1: I Turned to Stone 2: Keeping a Rendezvous 3: Reaper of the Glory 4: She Used Me Up
More Mayhem	Imelda May	2011	1: Pulling the Rug 2: Psycho 3: Mayhem 4: Kentish Town Waltz

Architecture Choices

Alright, have you decided whether to use a list or a tuple, to hold the songs for each album?

Which one you chose isn't as important as the reasons **why** you chose it.

Let's examine what we know - and what we don't know.

Architecture Choices

We know that once an album is released, the tracks on it don't change.

You may get a later release that contains bonus tracks, but that's not the same album.

If the number of tracks aren't going to change, we don't need to add or remove items from whichever structure we've chosen.

A tuple will work fine here.

Architecture Choices

You might want to store two values for each song: the name of the song, and its position on the album.

That means you'd be storing heterogeneous data - an **int** and a **string**.

We could use a list for that, but heterogeneous data is a good indication that a tuple might be a more suitable choice.

We'd end up with either a **list** of tuples, or a **tuple** of tuples.

Architecture Choices

What I mean is, we could have either

List of tuples

```
("More Mayhem", "Imelda May", 2011,  
 [ # a list of tuples  
   (1, "Pulling the Rug"),  
   (2, "Psycho"),  
   (3, "Mayhem"),  
   (4, "Kentish Town Waltz"),  
 ]  
)
```

OR

Tuple of tuples

```
("More Mayhem", "Imelda May", 2011,  
 ( # a tuple of tuples  
   (1, "Pulling the Rug"),  
   (2, "Psycho"),  
   (3, "Mayhem"),  
   (4, "Kentish Town Waltz"),  
 )  
)
```

Note that the first option uses a list to hold the four tuples for the songs, and the second option uses a tuple to hold the four song tuples.

Architecture Choices

Tuple containing a list of tuples

```
("More Mayhem", "Imelda May", 2011,  
[ # a list of tuples  
  (1, "Pulling the Rug"),  
  (2, "Psycho"),  
  (3, "Mayhem"),  
  (4, "Kentish Town Waltz"),  
]  
)
```

OR

Tuple containing a tuple of tuples

```
("More Mayhem", "Imelda May", 2011,  
( # a tuple of tuples  
  (1, "Pulling the Rug"),  
  (2, "Psycho"),  
  (3, "Mayhem"),  
  (4, "Kentish Town Waltz"),  
)  
)
```

We've got three levels of nesting here:

Either a tuple, containing a list, containing tuples

Or a tuple, containing a tuple, containing tuples.

Architecture Choices

One thing we don't know, is where the data will be coming from.

It's possible that the user interface will ask the user for the album details first, then ask them to type in the songs one after the other.

That could cause problems with a tuple, because we'd want to add each song when the user finishes typing it. We can't add new songs to a tuple.

In that case, we have to use a list.

Conclusion

As you can see, choices like this aren't always straightforward. Sometimes, you may decide to use one type of data structure, then find that you can't perform an important operation.

Fortunately, converting lists to tuples (and vice versa) is trivial in Python, so you can always change your code, and convert any data that your users have entered.

Quite often, when you're learning, the best approach is to pick one and see what happens.

The good news is that it's often apparent when you've made the wrong choice. You don't always know if you've got it right, but you'll soon find out if you haven't 😊.

Simple Jukebox Demonstration

In this video, we're going to produce the menu for a jukebox, using our **albums** data structure. This will be a more realistic application of indexing into a nested structure.

The previous exercises were fine for getting used to how nested indexing works, but don't really reflect how you'd do it in practice.

Often, you'd use variables for the indexes and calculate them in your code.

I'll start by demonstrating the program that we're going to create. While it's running, refer to the **albums** list in your code, to see where the options are coming from.

Constant

A constant is a fixed value that doesn't change.

Mathematics and the physical sciences use lots of constants.

For example

- PI: The ratio of a circle's circumference to its diameter. 3.141592653589793
- e: The base of natural logarithms. 2.718281828
- Avogadro's constant: The number of particles in one mole. 6.02214076E23
- c: The speed of light in a vacuum. very fast

Constant

If your code is navigating an aeroplane, and it changes the value of PI, your plane won't get to its correct destination.

Constants must not be changed.

Start off Slowly

Students often worry that they understand code examples that someone else has written, but don't feel able to write their own programs from scratch.

That's perfectly normal.

There's a big jump, from understanding someone else's code, to writing your own.

At this point in the course, I wouldn't expect a new programmer to be able to write a program from scratch. It takes time.

Start off Slowly

A great way to practice, is to modify existing code. Just like we did in this challenge.

Go back through the earlier examples, and change the code to behave differently. You don't have to do anything major - even small changes will improve your programming skills.

You might want to add some more items to the **buy_computer** program, in this section.

Now that you know about tuples, you could change the **available_parts** list to store tuples, rather than just strings.

Start off Slowly

Each tuple would contain the name of the part, and its price.

You could then display the prices, so that the user knows how much they're spending.

When they've chosen all their parts, print out the total price.

Have a look through the other examples, including those from previous sections, and play around with that code.

The more you practice, the sooner you'll be writing your own code.

Summary

- In the second half of this section, we showed how you can create nested lists - a list containing lists.
- Nested for loops are useful when processing a nested list.
- We used an example that printed each list, with the contents of each inner list.
- We looked at code style in **PEP 8 - Style Guide for Python Code**, with regard to lists.
- The function signature enables us to understand the parameters of a function. The **print** function is an interesting example, because it uses **keyword arguments** - also called named arguments.

Summary

- For the benefit of experienced programmers, we took a quick look at the generator expression, to produce output without a trailing comma.
- We then went on to look at more string methods, starting with the **join** method, which joins all the items in an iterable, providing the iterable only contains strings.
- The **split** method does the opposite to **join**, and returns a list from a sequence. This method has the named arguments **self** and **sep**.
- A **tuple** is an ordered set of data, and is another Python **sequence** type. Tuples, unlike lists, are **immutable** - they can't be changed after they've been created.
- You can append new items to a list, but not to a tuple.

Summary

- Some advantages of tuples are that they use less memory, protect against unintended changes to your data and help prevent bugs in your program.
- Tuples can be unpacked reliably, because they always contain the expected number of items. Because they're immutable, items can't be added or removed while your code's running.
- Unpacking a tuple can make code more readable and easier to work with.
- We saw how tuples and lists can be nested. We used data containing lists of albums, and demonstrated why you need to use parentheses around tuples.
- You also had a challenge, to fix the errors in the program by unpacking the tuples.

Summary

- Using our album list as an example, we talked about further nesting.
- We also saw some guidelines about when to use lists or tuples to store data.
- Nested data structures may seem complicated at first, so we gave you the chance to experiment indexing into different lists and tuples, e.g. albums and songs.
- We then showed you a more concise way to get details from our albums list, and gave you some exercises to practise the art of indexing into a nested structure.

Summary

In the next section, we're going to have a quick look at **functions**.

There will be more to learn about functions - the next section is an introduction to them.

We're covering functions now, so that we can start using them in our code examples.

That will make our examples better reflect the way real code is written.

Functions

We've used quite a few functions, so far.

We've also talked briefly about **arguments** and **parameters**. We saw a definition of those two terms, back in Section three.

One function we've used a lot, is the **print** function. We also looked at its definition in the documentation.

You should have bookmarked the link to Python's **built-in** functions. If you haven't, a google search will find them.

Python includes a lot of functions as standard, and we can also import others.

If there isn't a function that does what you want, you can write your own.

Functions

In this section, we're going to look at functions in more detail.

We'll see how to write our own functions, which is a great way to avoid duplicating code.

Putting code in a function allows you to use it over and over again, without re-writing it.

Functions

We'll also learn about returning values from functions.

Not all functions return a useful value. Some functions, like **print**, and the **.sort** method of lists, perform an action.

print prints something out. When you use **.sort** it sorts the list. These functions don't return anything useful.

Other functions calculate a value, and return it for your main code to use.

We'll look at the difference, in this section, and see how to write both types of functions.

Methods

A function that's bound to an instance of a class is called a **method**.

You define them in the same way as a function, and you'll learn more about them in the OOP section of this course.

print is a function, but **.sort** in **my_list.sort()** is a method. We've also seen some of the string methods, such as **str.casefold**.

You use methods in the same way as you use functions, but specify the object that they will act on, before the dot.

Why this is just an Introduction

One thing I'd like to make clear, is that this section won't cover everything about functions.

Before we remastered this course, we'd left the discussion about functions until later.

That allowed me to talk about everything you need to know about functions, in one section.

You need to understand dictionaries, for example, to understand some things about functions. So we left functions until after dictionaries.

But doing it that way meant that we couldn't use functions in our examples.

Why this is just an Introduction

So this is a brief introduction to functions, and we'll be talking more about them later.

This section will let you write your own functions, and understand why they're useful.

Just be aware that there is more to functions than we'll be covering in this section, and we'll come back to them later.

Function Definition

```
def multiply(parameters):
```

A function definition starts with the keyword def

Next, we have the name of the function

If the function will take parameters, they're declared in parentheses.

The parentheses are required, even when there are no parameters.

Function Definition

```
def multiply(parameters):
```

A function definition starts with the keyword def

Next, we have the name of the function

If the function will take parameters, they're declared in parentheses.

The parentheses are required, even when there are no parameters.

Function Definition

def multiply(parameters):

A function definition starts with the keyword def

Next, we have the name of the function

If the function will take parameters, they're declared in parentheses.

The parentheses are required, even when there are no parameters.

Program Flow when Calling a Function

Alright, that demonstrated how the flow of your program changes, when you call a function. We saw how execution jumps into the function, and executes its code.

When the function terminates, execution resumes at the point after the function call.

If you're assigning the return value of the function, as we are doing here, the debugger will stop on the same line as the function call.

That's because the assignment is still waiting to happen.

Not all functions return a useful value, and we'll write some that don't, later.

Parameters and Arguments

Our multiply function isn't very useful. It multiplies 10.5 by 4 and returns the answer 42, but we don't need a function to do that.

It would be a bit more useful if it could multiply different numbers.

It still won't be very useful, and you wouldn't really use a function for such a simple operation, but we're learning how functions work at the moment.

I want to keep the function itself simple, so that we don't get confused by a load of complex code.

Parameters

Parameters are like placeholders for the real values that you'll pass to your function.

They're just variables, but they're given a value when you call the function.

You may also see them referred to as **formal parameters**.

You might also hear programmers refer to them as arguments. That's not strictly accurate, but people often use less precise terms when talking.

Arguments

Arguments are the values that will be used by **formal parameters**, when we call a function.

Each parameter must be given a value, by providing an **argument** in the function call.

Providing values as arguments is called **passing** the arguments.

If a function defines two parameters, we **pass** two arguments to it when we call it.

Note that some function parameters can have **default** values. We'll look at them soon.

Debugging with Parameters

For those students who **have** programmed before, Python arguments are **passed by assignment**.

The *behaviour* is similar to **pass by reference**, when passing a mutable object. For immutable objects, the behaviour is closer to **pass by value**.

But note that neither of those terms really describe how arguments are passed, in Python.

Pass by reference and **pass by value** don't have any sensible meaning, in Python.

If you understand what that means, you won't miss anything by skipping through this video.

Positional Arguments

Positional arguments are assigned to the parameters in the order they appear.

In fact, the arguments in this example are really called **positional-or-keyword** arguments.

But as **positional-only** arguments didn't formally appear in the language until Python 3.8, you'll often find them referred to as **positional arguments**.

Functions

Now that we've seen what a function is, and how to write them, let's practice writing a few more.

I'm going to continue adding the functions to our **functions.py** file, but if you want to create a new file then that's fine.

I'm using our existing file, because I want to make it clear that you can have many functions in the same program.

In fact, you normally will have. Functions let you split up a problem into simple steps, with a separate function for each step.

Functions

It's much easier to debug small pieces of code - and it's also much easier to write code - if you break it down into small pieces.

Back in the **Stepping into the World of Python** section, we saw how to use a slice to reverse a string. We can use that technique to check if a word is a palindrome.

Palindrome

A palindrome is a word that reads the same backwards as forwards.

Palindromes are normally created for fun. There's nothing wrong with having a bit of fun, but they do have practical applications.

One application is in genetics. Work on DNA sequencing generates vast quantities of data, but standard compression algorithms don't compress DNA sequences well - they often end up larger, after compression.

In the last 20 years, algorithms have been developed to provide more efficient compression of the data, using palindromes in DNA sequences.

Palindrome

I'll put some links in the resources for this video, if you'd like to learn a bit more about that application.

<https://pubmed.ncbi.nlm.nih.gov/11700586/>

https://www.arpapress.com/volumes/vol9issue3/ijrras_9_3_14.pdf

<https://www.sciencedirect.com/science/article/pii/S0304397508008852>

At the moment, we're not really interested in the practical application of palindromes - we're just using them to learn about functions.

So let's write a function to detect if something is a palindrome or not.

Palindromes Challenge Solution

Our function only works with words - we can't use it to check if a sentence is a palindrome. The rules for a sentence are a bit more complicated; only letters count. We have to ignore spaces and punctuation.

With what we've covered so far, the easiest way you'll find to do that, is to build up a new string. The new string should only contain the characters that are alpha-numeric.

You've seen how to do something like that in the **strings2.py** program. That was back in the **Program Flow Control in Python** section.

I said **you** there, because that's going to be your next challenge.

Palindrome Sentences

Some examples of palindrome sentences are:

- Was it a car, or a cat, I saw?
- Do geese see god?
- Desnes not far, Rafton sensed.

Googling will find more examples.

Remember to also test with sentences that aren't palindromes.

What Options Do We Have?

Well, for this particular application, we could return zero. The main code uses zero to terminate the program. If the user types rubbish instead of a number, the program would terminate.

That is an option here. It probably won't be suitable for all applications - zero is a valid number, after all.

Another option is to keep asking, until the user **does** enter something valid.

If they enter something that isn't a number, the function can loop and let them provide input again.

That's the approach I'll take here.

Not All Functions **return** Something Useful

It's often useful for a function to return a value, but sometimes they don't have anything to return.

Some functions, as we've seen, return something back to the calling code.

You'll also write functions that perform some action, rather than calculating a value.

One example of that was the **sort** method, that we used to sort a list. That was back in the section on lists.

The **sort** function didn't return anything useful, but it did perform a useful function - it sorted the list.

Summary

- When you define a function to have **positional** arguments, the arguments are assigned to the parameters in their corresponding position.
- If you want a function to return a value, use the **return** keyword to specify the value that should be returned.
- Not all functions return something useful - some functions perform an action, rather than producing a value.

Summary

- If you don't explicitly return a value, Python will automatically return **None**.
- It is valid to explicitly return None from your functions. You might do that to indicate something wasn't found, for example. The dictionary **get** method does this.
- Functions that perform an action, rather than returning a value, used to be called **procedures**; but that distinction is no longer made, in modern programming languages.

Default Parameter Values

In the last video, we added another parameter to our **banner text** function. That allowed different widths to be used, making our function more flexible.

But you may have noticed that we broke all our existing code. We got an error on every line, from line 14 onwards.

If a function defines two positional parameters, then you must provide two arguments when calling it.

UNLESS the parameters have default values, that is. If you provide a default value for a parameter, then it's not necessary to provide a corresponding argument for it.

Brief Summary

Ok, there was a lot to take in, in the last few videos.

We've seen how to define **parameters**, and pass the corresponding **arguments** when calling our functions.

We've also learnt about **positional** arguments - where the arguments are used to provide a value for the parameter in the same position.

You can also use the parameters as **keyword** parameters, by specifying the parameter name when you pass the argument.

Brief Summary

It might not have been obvious, but that means you can pass the arguments in a different order, to the order that the parameters are declared. We'll see an example of that.

The default type of parameters, in Python, are **positional-or-keyword** parameters. You can either pass the arguments by their position, or you can provide the parameter name as the keyword.

You can even mix the two - but there are some rules about that, as we'll see.

Reasons to Document your Code

- Documenting your functions (and classes) makes it much easier for other people to use them.

Without documentation, other programmers will have to guess what your functions do, and how to call them.

If guessing is too much work, they won't use your functions. If they have to waste time writing the same functions again, you'll get fired and your children will starve.

Reasons to Document your Code

- That "other programmer" might be you, months or years later.

Remember that your code could be in use for years.

Documentation will help you to understand your code, when you come to modify it later.

Reasons to Document your Code

- If you write the documentation for your functions, before you write the code, you'll have a clear idea of what the function's going to do.

You'll have described any parameters, and the return values (if there are any), and what the function is supposed to do.

This is the reason that might not have been obvious.

If you start writing code with a clear idea of what that code has to achieve, you're far more likely to produce something that works.

Reasons to Document your Code

If you just start coding, without a clear idea of what the code is supposed to do, how will you know if it does it?

Documentation is important.

Docstrings

Notice that the function Docstrings are **inside** the function definition.

The convention, in C++ and Java, is to put the Docstrings **before** the function declaration.

In Python, the Docstrings go inside the function. There's a good reason for that - they become an attribute of the function.

Remember that everything, in Python, is an object. That means functions can have attributes - and I'll demonstrate that shortly.

Docstrings

If you haven't programmed before, don't worry about that - it probably won't make much sense.

The important thing is, the Docstrings go **inside** the function definition, just before the code.

Unpleasant Truths

Truth 1: Many programmers (ourselves included) don't document the functions that they write.

Truth 2: Many programmers (ourselves included) one day regret not documenting the functions that they wrote!

What is a Module?

From the Python documentation:

“A module is a file containing Python definitions and statements.

The file name is the module name with the suffix **.py** appended.”

Each **.py** file that you create becomes a new Python **module**.

Modules can be imported into other modules, or executed.

We've imported functions from the **random** module, and we've executed all the modules that we've created.

Docstrings

There's not a lot you need to know, when creating your Docstrings. Let IntelliJ, or your IDE, generate the stub for you, and use backticks around any Python names that you refer to.

The system takes care of everything else.

I mentioned that Python Docstrings appear **inside** the function. They become an **attribute** of the function.

Don't worry too much about what an **attribute** is - that will become clear in the **OOP** section of this course.

What I will do, here, is demonstrate why that's a good thing.

Fibonacci Numbers

There is still something I want to talk about, in this section about functions, and that's **function annotations and type hints**.

Before we do that, we'll write a few more functions, for practice.

The best way to remember something is by doing it, so we're going to "do" writing functions, for a while.

That will also give us some functions to add type hints to - we're killing two birds with one stone. Or "feeding two birds with one scone", if you're a vegan.

Fibonacci Numbers

The first function we'll write, will be one to calculate Fibonacci numbers.

That's going to cause some groans 😊. But we've deliberately used non-mathematical functions so far, and it's only fair to include a couple of maths ones.

If you think you're not good at maths, don't worry. If you can add up two numbers and count up to two, then you'll be fine.

Fibonacci Numbers

The Fibonacci series is quite fascinating. It's a theoretical mathematics thing, but it crops up quite often in nature.

Mathematicians came up with a way of calculating something that pineapples, artichokes and pine cones just got on with. In fact, many plants use Fibonacci numbers - without knowing about computers.

If they can do it, you can too 😊.

Function Annotations

And that's **function annotations**. They make it clearer what kinds of values your functions can accept, and what they return. We'll be annotating most of the functions that we create, in the remainder of this course.

We'll also look at how to annotate a generic type. That's a list of integers, for example, to indicate that a list of strings isn't suitable. We'll see how to do that, when we write a function that accepts a list.

Type Hints

We're not going to be using **type hints** in general, in this course. When we get to a suitable example, I'll include some **type hints**. It's useful to see them, so that you know what they are when you come across them.

They can be useful in very large programs, but would be far too much in our examples. They'd clutter the code and provide no useful benefit.

But please note that I'm only referring to the code in this course. I'm certainly **not** saying that you shouldn't use type hints. If you're working for a company that **does** use them, then you should too.

A History Lesson

We're going to leave maths for this next function. Instead, we'll see how to print text with different effects - different colours, bold and so on.

It will also give me an opportunity to demonstrate something about testing your code.

We've been running our code in an IDE, which is a slightly artificial environment. When you distribute your programs to your users, they won't be running in an IDE. They'll run your code from a command prompt on Windows, or a Terminal session on Linux and Mac.

Sometimes that can lead to different behaviour, and our next example will demonstrate that.

A History Lesson

Most terminals will respond to certain character sequences, to perform actions rather than printing.

We've seen two such sequences already; **backslash n** causes the terminal to start a new line, rather than printing characters. **backslash t** advances to the next tab stop.

It's time for a history lesson. So sit back and relax again, while we learn some history that explains why things work the way they do, today.

And in case anyone suspects I'm recording this on April Fools day, this is all true. This really is how things were.

A History Lesson

Terminals needed both a carriage return and a line feed, to start a new line.

Carriage return took the print head back to the left hand side, and line feed moved the paper up a line. Obviously, that slowed the typist down on this particular terminal - they had to press two keys to start each new line.

Device drivers for Unix started including the carriage return automatically, which meant software only had to send a line feed.

That's why you sometimes don't get correct line breaks, if you open a Linux file on a Windows computer. Windows uses carriage return **and** line feed, to indicate a new line.

A History Lesson

It also explains something that has confused a few students. When they compare two files, they notice that the contents appear identical, but the Windows file has a larger size. That's because of the extra **carriage return** character, at the end of each line.

You can probably imagine my joy, when I got access to a terminal with a screen.

Because it was using a screen, rather than printing on paper, it became possible to print to different parts of the screen.

A History Lesson

You could maintain a counter in the top, right hand corner, for example. Or show useful statistics in a table on the left of the screen, while the user typed their instructions in the remaining 80 columns.

At first, there wasn't a standard way to control the cursor position. Eventually, a standard did emerge - **ANSI** sequences. ANSI is the **American National Standards Institute**.

We're going to use ANSI escape sequences, to change the colour of the text that we print. And the first thing you'll notice, is that the sequences are horrible.

Printing in Colour

The escape character, by the way, is ASCII character 27. That's 1B in hex, and it's easier to include the characters using the hexadecimal unicode character.

This is slightly confusing, because the backslash character is also called the **escape** character, in Python - and a few other languages.

It was called the **escape** character because it changes the meaning of what comes next. When a **TTY** terminal received an escape character, it knew that it should interpret the next characters differently.

Printing in Colour

The backslash, in Python strings, does the same thing. It tells Python that it should interpret what comes next in a special way.

So **backslash n** is a newline, and **backslash backslash** represents a backslash - you have to escape your escape character, if you want it to be recognised literally.

Note for Windows Users

If you're using Windows, and are using a different IDE, you may not see the colour change. If you're running your code in a Windows command prompt, then you probably also won't get red text.

That might change, as Microsoft replace the command prompt with **Windows Terminal**, but at the time I'm recording this, Windows doesn't recognise ANSI escape sequences by default.

Don't worry, I'll be showing you how to make this work, in the next couple of videos. Keep watching and typing the code, and it'll all magically work, soon.

Opening a Command Prompt or Terminal Session

On Windows, type **cmd** into the search bar.

On a Mac or Linux computer, open the terminal in the usual way.

Linux users generally know how to do that: the shortcut **Ctrl-Alt-T** works on Ubuntu and some other distros.

On a Mac, use the **launchpad** and type in **terminal**.

Colorama Package for Windows

This video is for students who want to test their code on Windows.

If you're not using Windows, and don't intend to test your code on Windows, then you might want to skip this video.

Before you do, it does cover how to install a package that you've downloaded. If you're working off-line, and have to use another computer to download packages, then you should find this information useful.

Pre-release Colorama Package

The current version of the **colorama** package doesn't support underlining and reverse video. It also has an issue with bright colours, if your command prompt text is already bright.

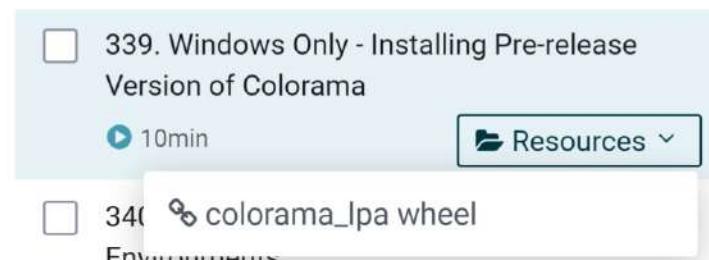
colorama is an open source package, and we've modified it to fix those problems. We've submitted a pull request, on github, so that the authors can include our fixes in the main package.

But that takes time - especially with open source projects. The authors don't get paid, and are usually fitting in their open source work around their day job. There are a few updates for them to add, before they get to ours.

Pre-release Colorama Package

In this video, we're going to install the Ipa (learnProgramming.academy) version of the package. At some point, we expect these changes to get into the release version. When that happens, we'll remove this video from the course.

Start by downloading the **colorama_Ipa** wheel from the resources for this video. A **wheel** is the name given to a file containing a Python package.



Colorama Module and Virtual Environments

If you read through the Wikipedia article on ANSI escape sequences, you'll have seen a reference to a Python package that allowed ANSI codes to work on the Windows command prompt.

That package is called **colorama**, and it's still being actively maintained. The latest version, at the time I'm recording this, was released in December 2019.

It's important to check that, if you're going to use external packages. If something isn't being maintained, it could contain bugs that aren't documented anywhere.

If you include a buggy package in your code, then your program could be buggy too.

Colorama Module and Virtual Environments

I won't provide a link to the **colorama** documentation. Use your favourite search engine to search for **python colorama**.

You'll find it either on **PyPI** (the Python Package index) or on **github** - either source will be fine.

Do read through the documentation. You should check the documentation for all packages that you install.

Review of Package Installation

The **colorama** package is now installed in our **Python virtual environment** - or **virtualenv**, or even **venv**.

I've had you creating a virtualenv, and now installing a package into it, so it's probably time to find out what a **virtualenv** is.

Python Virtual Environment

A Python virtual environment is an isolated installation of Python, separate from your main installation.

If you had to update the **pip** and **setuptools** packages, you'll have seen that packages come in different versions. Python packages get updated, just like anything else.

If you have a program that works with a particular version of a package, it might break when you upgrade that package.

Python Virtual Environment

That wouldn't be good, and as developers, you'll often try out new versions of things.

If you install new versions into your main Python installation, you could break a program that you rely on.

That could be very bad indeed ...

Mac and Linux rely on Python

Mac and Linux computers rely on Python for lots of things. Many components of a Linux GUI use Python scripts to function.

If you break the system's installation of Python, you could lose access to your operating system's desktop environment.

Windows users have it slightly easier - you can just uninstall Python and install it again.

But that's still a hassle, and using a virtual environment saves the hassle.

Virtual Environment to the Rescue

A Python virtual environment contains a copy of your main Python installation.

If you ticked the box to **Use Site Packages**, when you created your virtualenv, the environment will also have access to any packages in your main installation.

In addition, you can install packages into the virtualenv, without affecting your main installation.

If you install something that doesn't work, or upgrade something to a later version that doesn't work, you won't break important programs that rely on your main Python installation.

At worst, you can delete the virtualenv and create a new one.

Always use some form of Isolated Environment

The title of this slide quotes *Brett Cannon*, one of the members of the Python Steering Council.

Until 2015, he used to work for Google. He now works for Microsoft, so it's safe to assume that he's pretty good 😊.

The isolated environments we're using are **virtualenvs**. They're built into Python, and the JetBrains IDEs have good support for them.

There are other ways to isolate Python - Conda environments, for example. But we'll use Python virtual environments.

Always use some form of Isolated Environment

Ok, that's a brief description of virtual environments. I generally refer to them as **virtualenv**, and you'll come across the even shorter name, **venv**.

They're all the same thing.

Activating the Virtual Environment

On Windows, the command we'll use is something like

```
c:\Users\tim\venv\Python38\Scripts\activate
```

We run **activate dot bat** from the virtualenv directory. Obviously, you'd use your own path there.

On Linux or Mac computers, the command will be something like

```
source /home/timbuchalka/venv/python38/bin/activate
```

We use the **source** command, to load the functions file **activate** from the venv directory.

Testing the Environment

Note that this **isn't** how your users will run your programs.

What we've done here, is tested the environment that your code will be running in.

When your users run your programs, they'll use a desktop shortcut, or a simple command line instruction.

But we need to create working software, before we concern ourselves with installing it on users' computers.

A Function to Test our HiLo Game

We've digressed a bit in the last few videos, but there was useful stuff in there.

When you start experimenting with different Python packages, you can install them into their own virtual environment, and avoid messing anything up.

You've also got a way to make your output more interesting, using colour. Just like IntelliJ does with errors and links, in the Run pane.

In this video, we're going to put the code from the **HiLo** game into a function.

A Function to Test our HiLo Game

Why might we want to do that?

When we were testing the game, in an earlier section of this course, I joked that the next 35 hours of video would be me pressing **h** and **l**, over and over again.

That would be one way to test every single number, but it's not a good use of our time. We've got a computer to do that for us.

If we put the code into a function, we can call it in a loop and test every possible value.

Testing and Debugging

Testing is the process of finding out if there are bugs in your code.

Debugging is the process of working out what the bugs are, and fixing them.

Testing and debugging go together.

You have to **test** the code first, to see if it has any bugs.

Once you've discovered that the code doesn't work, you then move on to identifying the causes of the errors, and fixing them.

Then you repeat the cycle - test the code again, to see if you've removed all the bugs.

Testing and Debugging

Our code might contain more than one bug. Before I add another one, you might like this quote:

I know there are bugs in my code ... I put them there!

I haven't been able to find the source of that quote, but it's something that's stuck with me.

Assume that all complex programs will contain bugs.

The bugs are often discovered, when testing the code. But sometimes, the bugs can go unnoticed for years.

Heartbleed Bug

OpenSSL is an open-source library that deals with the TLS and SSL security protocols.

They're used, for example, whenever you visit a site using HTTPS - which these days, is most sites on the internet.

In March 2012, OpenSSL contained a serious bug. The bug went undiscovered until 2014, and a fixed version was released in April of that year.

Because of that bug, a great part of the internet was insecure for two years.

We don't want Bugs, but we put them there

There is evidence that the Heartbleed bug was exploited.

Bugs may cause our code to crash, but they can also leave our programs open to attack.

Thorough testing can help to eliminate bugs.

And modern IDEs also help to prevent them.

The buffer overrun bug that affected Heartbleed is a common cause of security problems.
Modern IDEs now warn about buffer overruns, when they can detect them.

That's why it's important to pay attention to the warnings that your IDE gives you.

Counting Correct Guesses

When we created this game, I mentioned that the computer could tell it had the correct number, about half the time.

I said that it could correctly guess 489 out of our thousand numbers.

We'd like to thank Andre Stevens, a student on this course, for a reminder to explain how we got the figure 489.

We got that figure by getting Python to count the number of times it guessed correctly.

If you remember, that happens when the values for **low** and **high** converge.

That takes ten guesses, so we can count the number of times our function returns ten.

Counting Correct Guesses

You'll probably want to experiment with different HIGH values. Using specific values - such as 10, in this case - will make that harder.

What I'll do instead, is calculate the maximum number of guesses, as we go round the loop.

We'll also count how many times that maximum number is returned.

Parameter Types

There are 5 different kinds of parameters that you can use. As I said, we won't be looking at positional-only parameters in this course.

The description's blanked out in the slide, to keep things simple.

parameter

A named entity in a [function](#) (or [method](#)) definition that specifies an [argument](#) (or in some cases, [arguments](#)) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either [positionally](#) or as a [keyword argument](#). This is the default kind of parameter, for example `foo` and `bar` in the following:

```
def func(foo, bar=None): ...
```

- *positional-only*:
- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example `kw_only1` and `kw_only2` in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example `args` in the following:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example `kwargs` in the example above.

© Copyright 2001-2020, Python Software Foundation.

The Rules

When we use the different parameter types in our functions, we have to use them in the order that they appear below:

- Any **positional-or-keyword** arguments that we define, **MUST** come **first** in the parameter list.
- If we have a **var-positional** parameter - that's a parameter that starts with ***** - then it must come **after** any positional-or-keyword arguments.

There's a good reason for that, as we'll see soon.

parameter

A named entity in a **function** (or **method**) definition that specifies an **argument** (or in some cases, **arguments**) that the function can accept. There are five kinds of parameter:

- **positional-or-keyword**: specifies an argument that can be passed either **positionally** or as a **keyword argument**. This is the default kind of parameter, for example **foo** and **bar** in the following:

```
def func(foo, bar=None): ...
```

- **positional-only**:

- **keyword-only**: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare ***** in the parameter list of the function definition before them, for example **kw_only1** and **kw_only2** in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- **var-positional**: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with *****, for example **args** in the following:

```
def func(*args, **kwargs): ...
```

- **var-keyword**: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with ******, for example **kwargs** in the example above.

© Copyright 2001-2020, Python Software Foundation.

The Rules

- Any parameters defined **after** a var-positional parameter must be **keyword-only** arguments (which includes **var-keyword** arguments).
- Finally, any **var-keyword** arguments appear last. We'll be looking at how to define **var-keyword** arguments later. We need to learn about *dictionaries* first

parameter

A named entity in a [function](#) (or [method](#)) definition that specifies an [argument](#) (or in some cases, [arguments](#)) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either [positionally](#) or as a [keyword argument](#). This is the default kind of parameter, for example `foo` and `bar` in the following:

```
def func(foo, bar=None): ...
```

- *positional-only*:

- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example `kw_only1` and `kw_only2` in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

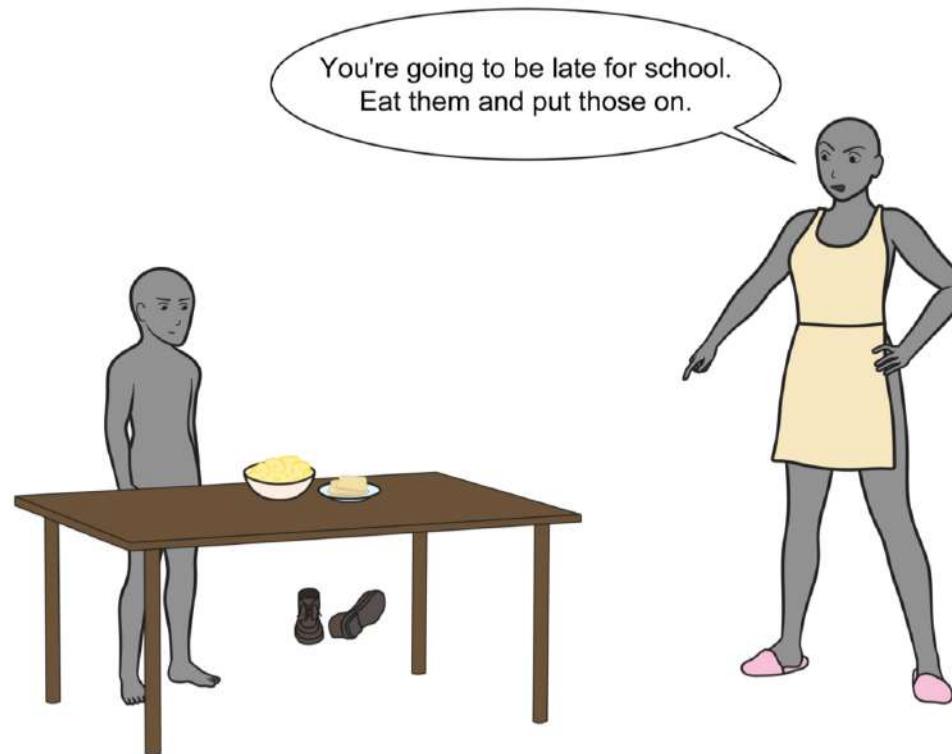
- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example `args` in the following:

```
def func(*args, **kwargs): ...
```

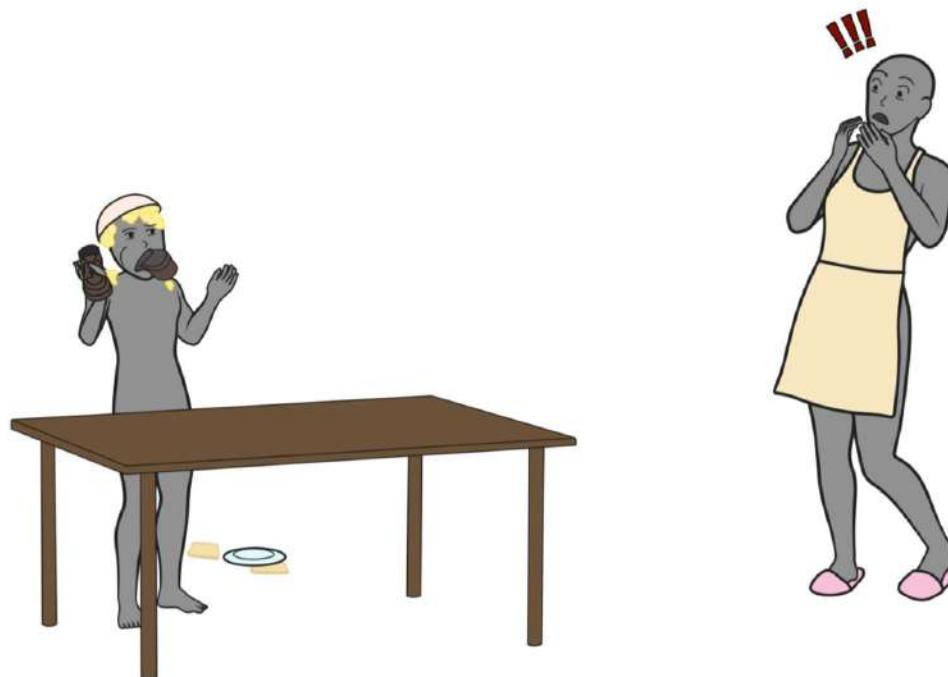
- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example `kwargs` in the example above.

© Copyright 2001-2020, Python Software Foundation.

Rules for Variable Number of Arguments



Rules for Variable Number of Arguments



Rules for Variable Number of Arguments



The small alien had been told to hurry, and did the best that it could, with those instructions.

If we write ambiguous instructions in our code, the best we can hope for is an error.

The worst we can hope for, is that our shoes get eaten!

In the next video, we'll see how our code would be ambiguous, if we don't define our parameters in the order I described.

Section Summary

In this section, you've learnt a lot about functions. Not everything yet, but enough to let you write our own functions.

You now understand what **parameters** and **arguments** are, and how the arguments provide the values for your formal parameters.

You define and name your parameters when you define the function. You then pass arguments when you call the function.

You've seen how the flow of execution jumps into the function body, when a function is called, and returns to the calling code, when the function terminates.

Section Summary

We covered **scope**, and saw that our parameter variables only exist inside the function. They're not available in the code outside the function.

We still have a bit more to learn about **scope**, but we know all we need to write our own functions.

Procedures

We looked at functions that perform actions, and functions that return a result.

It is possible for a function to do both, but we haven't seen any examples of that. We deliberately didn't write functions that way, because you should generally avoid doing it.

Functions should be self contained blocks of code, that either perform some action, or return a result. Doing both in the same function can result in code that's difficult to maintain.

Functions that perform an action used to be called **procedures**, but modern programming languages no longer make that distinction.

Instead, they return **None**, in Python. Languages like C and Java use **void** as the return type, to indicate that nothing is returned.

Default Parameter Values

We've seen how to provide **default values** for our parameters. Default values make using the function easier, because you don't have to provide arguments for commonly used parameter values.

We used the parameter name, when specifying an argument, in the **banner_text** function. Keyword arguments let you pass arguments in a different order than the parameters are defined.

More importantly, they allow arguments to be provided, when a parameter with a default value appears earlier in the list.

We saw one example of that, when we wanted to change the **screen_width** without providing any text.

Docstrings

Docstrings are an important feature of Python functions. They're used to document what the function does, and how to use it.

You've had a lot of practice writing Docstrings, and will get more as you write more functions in later sections.

Function annotations also help with documenting your functions. They describe the type of the arguments, and the return value.

Type hints perform a similar job, but are used for variables in your main program.

Testing

A very important part of programming is **testing**.

We learned how to activate a virtual environment, so that we could run our code like our users will run it.

We also used a function to test the HiLo game. Putting the code into a function allowed us to test every possible value. That would be very tedious to do manually.

We haven't looked at automated testing yet, but that example demonstrated how useful it can be. If we can get the computer to test our code, we can perform more thorough tests, and save ourselves a lot of time.

Parameter Types

We finished the section by looking at the five different types of parameters.

We haven't discussed **positional-only** parameters, because you're unlikely to use them.

var-keyword parameters will be discussed later, after you've learnt about Python dictionaries.

We've seen examples of the other three parameter types, and used them in our functions.

Introduction

Earlier in the course, we looked at two of the data structures that are built into Python. We saw how to use lists and tuples, and discussed some of the differences between them. In this section, we're going to learn about two more built-in data structures: dictionaries and sets.

Lists and tuples are sequence types. That means we can use indexing to access individual items.

Dictionaries and sets **aren't** sequences. We can't access the items using index positions.

Introduction

When dealing with items in a dictionary, we use a **key** to get individual values from the dictionary. Dictionaries store key/value pairs.

You'll also find a dictionary referred to as a **mapping**. If you know Java, then a Python **dict** is equivalent to a Java **HashMap**.

A set is an unordered collection of *things*. There's no way to retrieve a specific item from a set - which can be confusing.

But sets can be very useful, as we'll see.

What is a dictionary?

A dictionary is a collection of **values**, that are stored using a **key**.

As an example, employees in a company might be given an *employee* ID. That would form their key, and the value would be their personnel records.

When a car triggers a speed camera, the car's registration number - or license number - is recorded.

That becomes the **key** that's used to find the owner of the vehicle, so that a speeding fine can be issued.

What is a dictionary?

If the camera only recorded that it was a blue car, the system wouldn't function very well. The **key** is needed to find the records for that particular blue car, in the collection of all cars in the country.

That's basically what a dictionary is. Each value that's stored in the dictionary has a key. We use the key to get the values from the dictionary.

You'll find dictionaries described as storing *key/value* pairs. Each value has a unique key, that's used to refer to it.

Indexing Versus the get Method

If the key doesn't exist, **dot get** will return **None**, whereas indexing will crash with a **KeyError**.

The **get** method is useful when you're not sure if the key exists, or not.

Indexing Versus the get Method

Now you may be wondering "why not just use the get method all the time? Why bother with indexing, if it can crash?".

One answer to that is, indexing is faster.

If you know that the key will be present, then use indexing (with square brackets).

If there's a possibility that the key won't be present, then call the **get** method.

We'll see how we can be sure that the key **is** present, in the next few videos.

Indexing Versus the get Method

Another reason is that sometimes you want to get an error.

We'll talk more about that when we look at handling exceptions.

But sometimes it's better for your code to crash, rather than for it to process invalid data.

Experiment by printing out different values from the **vehicles** dictionary, using both the **dot get** method and indexing, and I'll see you in the next video.

.items() in Python 2

You might come across that inefficient loop if you find yourself modifying or converting old Python 2 code.

.items() in Python 2 created a list. That means it needed a copy of the data, and used more memory.

For that reason, programs might have used the simple for loop, to avoid the extra memory overhead of using **.items()**.

.iteritems() in Python 2.2

Python 2.2 added a **.iteritems()** method, which works in a similar way to Python 3's **.items()** method.

Python 3 now uses something called a generator, and doesn't copy the data from the dictionary.

With Python 3, remember to use **enumerate** when iterating over sequences, and **.items()** when iterating over dictionaries.

Changes to Python Dictionaries

If you installed Python from the [python.org](https://www.python.org) website, the implementation of Python that you installed from that site is called **CPython**.

That's because most of the implementation is written in C.

There are other implementations of Python; for example **IronPython** (or IPython) and **Jython**.

Changes to Python Dictionaries

CPython is the reference implementation of Python - the version written by Guido van Rossum.

Version 3.6 of **C_Python** changed the way that dictionaries worked.

In earlier versions, the keys of a dictionary were unordered. That means you could get the keys printed in a different order, each time you run the program.

If you're using **Python 3.5**, and the dictionary prints in a different order each time, now you know why that is.

Change in Python 3.6

Python 3.6 preserved the insertion order of dictionary keys. That means you'll always iterate over them in the same order.

That was an implementation detail, which meant you couldn't rely on that behaviour in other implementations of Python 3.6.

With **Python 3.7**, that behaviour became part of the language.

When you iterate over a dictionary with **Python 3.7** and above, the keys will appear in the order they were added to the dictionary.

Change in Python 3.6

That's something you need to be aware of, because you can get a different order in earlier versions of Python.

Python 3.5 goes out of support in September 2020 - but that doesn't mean people will stop using it.

But it does mean it's acceptable for you to specify that your code requires **Python 3.6** or higher, if you rely on features introduced in that version.

Adding items to a dictionary

That's what the documentation means, when it says that dictionaries **preserve the insertion order**.

As I've mentioned, if you're using Python version 3.5 or earlier, you'll get the entries appearing in random order. Prior to Python 3.6, dictionaries were unordered. You couldn't rely on the order of the keys, when iterating over the dictionary.

Cython 3.6 preserved the insertion order, and that became a language feature with Python 3.7.

In a later video, we'll see how to iterate over the keys in order. That can be useful, because inserting and deleting keys can result in a sorted dictionary becoming unsorted.

But first, we'll look at changing the value associated with a key, and deleting keys.

Removing items from a dictionary

In this video, we'll look at a couple of ways to remove items from a dictionary, and discuss when you might use each one.

That will give us the basic operations we can perform with a dictionary, and we can move on to using them for more interesting things.

We've seen how to delete an item from a list, using the **del** keyword. We first used that in the lecture **Replacing a slice**, back in the **Lists and Tuples** section. Review those lectures, if you've forgotten how to delete from a list.

Deleting from a dictionary is very similar, but we use the key rather than an index position.

Groan! Not Another Menu!

Note that we're not learning how to write menus here.

We're looking at how to access and add items to Python data structures.

We've already seen how to do that with a list - now we'll do the same thing with a dictionary.

But Which One Should I Use?

That's a question new programmers often ask. There are so many options, and it can be confusing working out what you should use, and when.

We're going to write the same program, using a dictionary instead of a list.

We can then talk about the advantages that a dictionary gives us, and the disadvantages.

When you understand the pros and cons of each, you'll be in a position to decide which one to use, for your particular application.

Understanding the different capabilities of these data structures is the key here.

Once you understand the different ways that a dictionary works, compared to a list, then choosing between them becomes much easier.

in with a dictionary

in with a dictionary

```
1 available_parts = {"1": "computer",
2                         "2": "monitor",
3                         "3": "keyboard",
4                         "4": "mouse",
5                         "5": "hdmi cable",
6                         "6": "dvd drive",
7                         }
8
9 print("mouse" in available_parts) # False
10 print("4" in available_parts) # True
```

in with a list

```
1 available_parts = ["computer",
2                     "monitor",
3                     "keyboard",
4                     "mouse",
5                     "hdmi cable",
6                     "dvd drive",
7                     ]
8
9 print("mouse" in available_parts) # True
10 print("4" in available_parts) # False
```

On the left, we're using a dictionary. `in` is checking the keys, and "mouse" **isn't** a key in our dictionary. So **line 9** is **False**.

The key "4" **is** in the dictionary, so we get **True** on **line 10**.

in with a dictionary

in with a dictionary

```
1 available_parts = {"1": "computer",
2                         "2": "monitor",
3                         "3": "keyboard",
4                         "4": "mouse",
5                         "5": "hdmi cable",
6                         "6": "dvd drive",
7                         }
8
9 print("mouse" in available_parts) # False
10 print("4" in available_parts) # True
```

in with a list

```
1 available_parts = ["computer",
2                     "monitor",
3                     "keyboard",
4                     "mouse",
5                     "hdmi cable",
6                     "dvd drive",
7                     ]
8
9 print("mouse" in available_parts) # True
10 print("4" in available_parts) # False
```

On the right, we're working with a list. The value "mouse" **is** in the list, so we get **True** on **line 9**.

The value "4" **isn't** in the list, and **line 10** prints **False**.

in with a dictionary

in with a dictionary

```
1 available_parts = {"1": "computer",
2                         "2": "monitor",
3                         "3": "keyboard",
4                         "4": "mouse",
5                         "5": "hdmi cable",
6                         "6": "dvd drive",
7                         }
8
9 print("mouse" in available_parts) # False
10 print("4" in available_parts) # True
```

in with a list

```
1 available_parts = ["computer",
2                     "monitor",
3                     "keyboard",
4                     "mouse",
5                     "hdmi cable",
6                     "dvd drive",
7                     ]
8
9 print("mouse" in available_parts) # True
10 print("4" in available_parts) # False
```

Remember, that when you use `in` with a list, it checks the items in a list. When you use `in` with a dictionary, it checks the keys in the dictionary - **not** the values.

Shopping Cart Menu

If you were writing a shopping cart for an on-line store, you wouldn't have the items typed into your code.

The items available for sale might come from a database, or a file on disk.

If the items appear in this format:

```
1, computer
2, monitor
3, keyboard
4, mouse
5, hdmi cable
6, dvd drive
```

then it would be quite easy to create a dictionary by reading in that data.

Shopping Cart Menu

But what if the items were presented to you without the numbers? You may read in a series of items, like this:

```
computer
monitor
keyboard
mouse
hdmi cable
dvd drive
```

In this case, you'd have to generate the numerical characters that the user would type. If you've got to include that code anyway, then it may be easier to leave the items in a list.

There's another reason why you might prefer to use a list. Python can sort lists easily.

I'll finish this video by demonstrating that, with the original **buy_computer** program.

COMPLETE PYTHON MASTERCLASS
Dictionary menu challenge solution



DICTIONARIES AND SETS
PM-8-2

Dictionary menu challenge solution

So we've seen how to use a list and a dictionary to present our menu.

Which one is "better" depends on the nature of your original data, and exactly what you want to do with the data.

You've seen some examples of why a dictionary may be more suitable here - shorter code, for instance.

But you've also seen that a list may be more appropriate, if you have to sort the values.

In the next video, we'll return to our new program, and add items that the user chooses.

Before that, try this challenge.

Groan! Not another menu

Once again, we're not learning how to write a menu.

We're learning how to transform data.

We display it in a menu, because that lets us see what we've done to it.

Just printing it out would serve the same purpose, but it's more interesting if there's a use for the transformed data.

So how do we want to transform the data?

Transforming our recipe dictionary

What we want, is something like

```
Please choose your recipe
-----
1 - Butter chicken
2 - Chicken and chips
3 - Pizza
4 - Egg sandwich
5 - Beans on toast
6 - Spam a la tin
:
```

But what we have is:

```
recipes = {
    "Butter chicken": [ ... ],
    "Chicken and chips": [ ... ],
    "Pizza": [ ... ],
    "Egg sandwich": [ ... ],
    "Beans on toast": [ ... ],
    "Spam a la tin": [ ... ],
}
```

The dictionary values have been replaced with an ellipsis, on the right hand side, just so everything fits on the slide.

But you can see that the right hand side represents our **recipes** dictionary.

Transforming our recipe dictionary

What we want, is something like

```
Please choose your recipe
-----
1 - Butter chicken
2 - Chicken and chips
3 - Pizza
4 - Egg sandwich
5 - Beans on toast
6 - Spam a la tin
:
```

But what we have is:

```
recipes = {
    "Butter chicken": [ ... ],
    "Chicken and chips": [ ... ],
    "Pizza": [ ... ],
    "Egg sandwich": [ ... ],
    "Beans on toast": [ ... ],
    "Spam a la tin": [ ... ],
}
```

We need to transform that, to what we can see on the left hand side.

In Python, that's easy. In fact, we can do it with 1 line of code.

Because that would use something we'll be covering later, we're going to use 3 lines of code instead.

What's for tea?

We have to learn to walk before we can run. Obviously, you can't learn about dictionary comprehensions **before** you've learnt about dictionaries.

But that leaves me with a quandry. I have to show you code that makes sense, with what we've covered so far. But I'm also aware that there's a more efficient and Pythonic way to write it.

To help solve that quandary, I've presented the more efficient code as a comment.

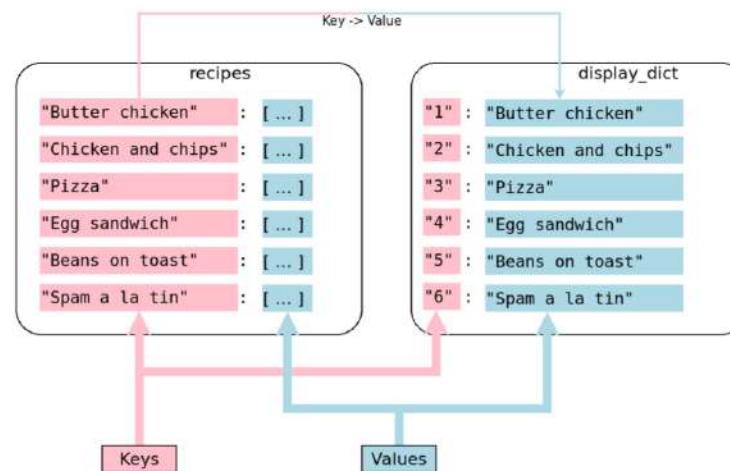
When you come to review these lectures and the code examples, that comment will remind you that you should use a comprehension here.

What's for tea?

It **will** make perfect sense, once you've watched the section on comprehensions.
Until then, think of these comments as a reminder to your future self.

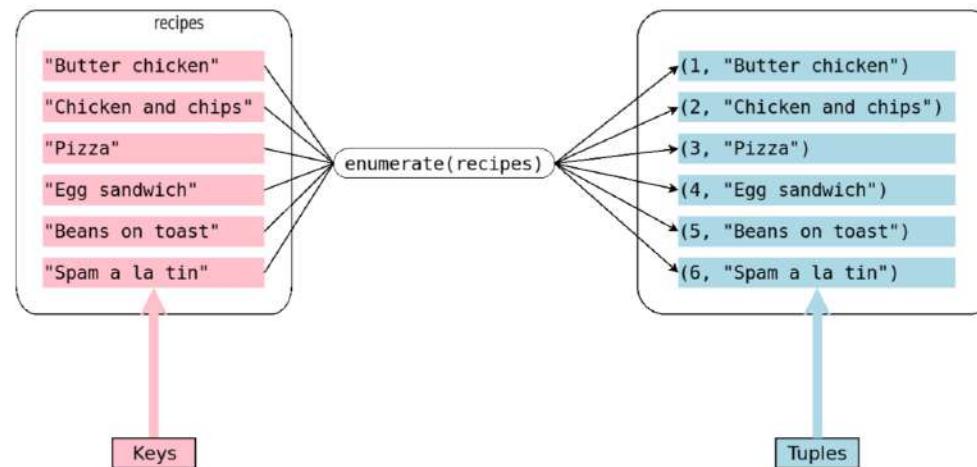
Transforming our data

We've transformed the data, so that the keys from `recipes` become the values in `display_dict`.



The **keys** in our `recipes` dictionary, on the left, have become the **values** for the `display_dict` dictionary, on the right.

Enumerating over dictionary keys

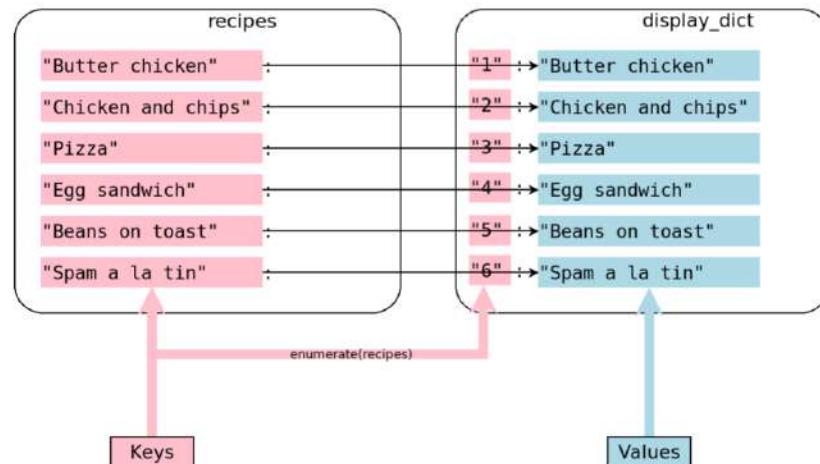


The `enumerate` function takes each of the keys from the dictionary, generates the next number for each one, and emits both values as a tuple.

Our code then takes the first item in each tuple - which is the number - and converts it to a string. It uses that string as the key in the `display_dict` dictionary.

It uses the second item in the tuple - the recipe name - as the value for that key.

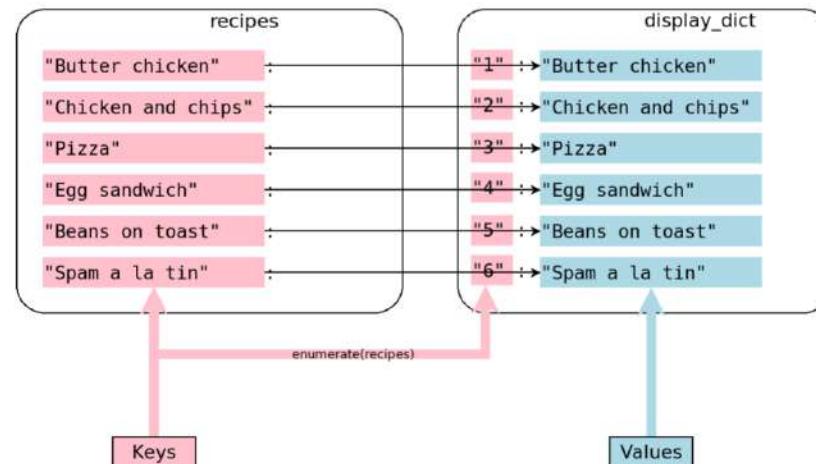
Generating a new dictionary



We end up with a new dictionary - the `display_dict` dictionary shown on the right.

All the **values** on the right are valid **keys** in the `recipes` dictionary on the left.

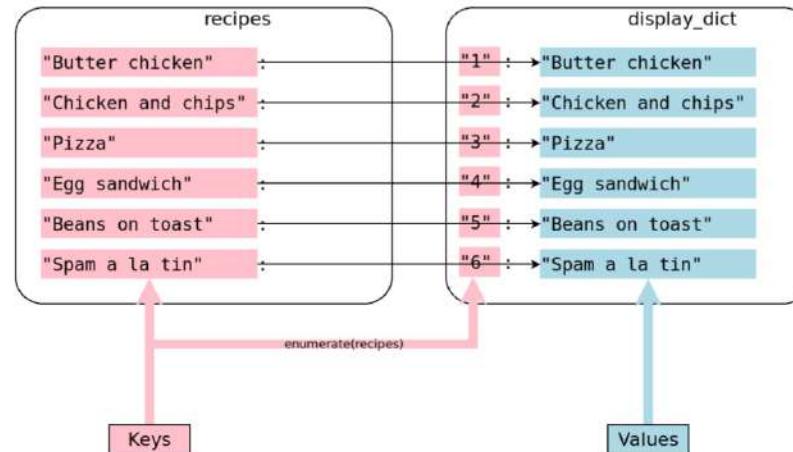
Generating a new dictionary



That means we can use those values to access items in the `recipes` dictionary.

If the user chooses option 3 for a Pizza, we can look up "Pizza" in the `recipes`, to get the list of ingredients for a pizza.

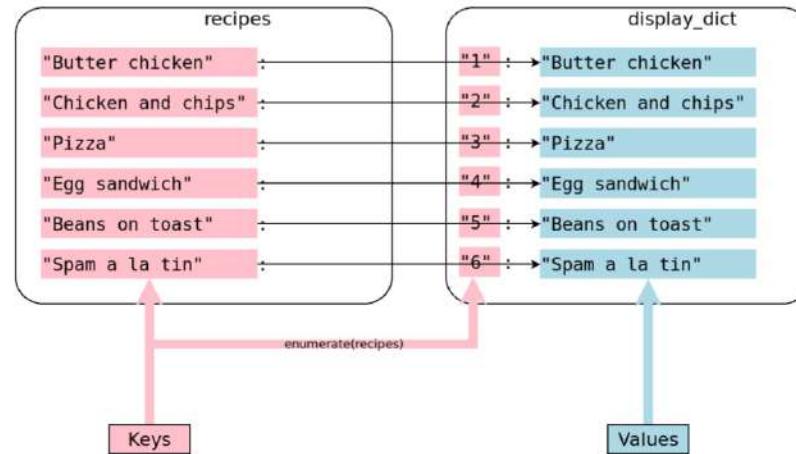
Create one dictionary from another



We saw this in the last video.

We use `enumerate` to take the keys from the `recipes` dictionary, and produce a new dictionary called `display_dict`.

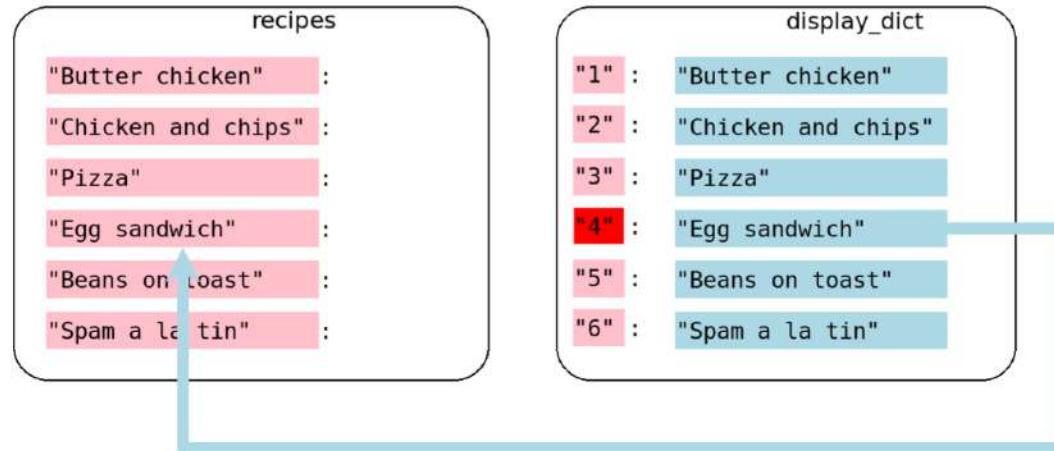
Create one dictionary from another



That lets us display the keys, with an option number that the user can use to select a recipe.

Now lets see what happens when the user makes a selection.

Linking the dictionaries



I've used option 4 for this example, because that's the option I used when I ran the program.

Option 4 is the **Egg sandwich**, and we retrieve that value from the **display_dict** dictionary.

Linking the dictionaries

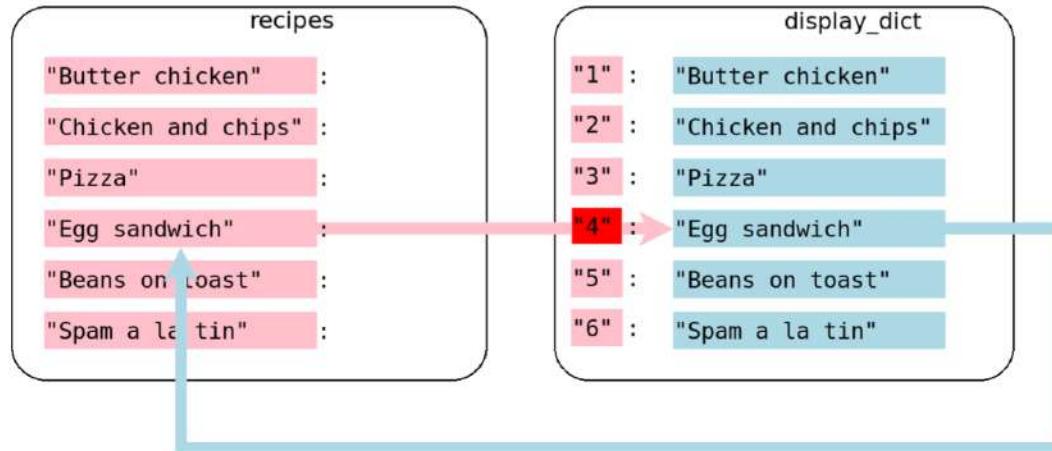


We can then use that value, to retrieve the ingredients list from `recipes`.

That's the blue arrow on the slide.

When we have a value from the `display_dict` dictionary, we can use it to index into the `recipes` dictionary.

Linking the dictionaries



Because all the values on the right were produced from the keys on the left, we know they'll work as indexes into the `recipes` dictionary.

The pale red arrow shows where "Egg sandwich" came from.

We didn't re-type those strings, so we don't have to worry about things like typing errors, missing spaces or incorrect capitalisation.

How to choose?

Ok, how do we decide which of these data structures will be more appropriate for our program?

This can have new programmers staring at the screen for hours, unable to proceed. They know they should use one of them, but can't decide which.

If you find yourself in that position, just pick one. Toss a coin, if you like.

It's much better to start doing something, than to sit there in a state of paralysed indecision.

Trust me - if it matters, and you chose the wrong one, you'll soon find out.

How to choose?

Please note that I'm not suggesting you design your programs by tossing a coin.

Think carefully about the requirements, and what you'll need to do with your data structures.

But if you can't decide between different alternatives, then picking one and getting on with it is much better than doing nothing.

If you do nothing, that's what you'll learn - nothing!

How to choose?

If you make a mistake and make the wrong choice, you'll have learnt that it's the wrong choice. And hopefully, you'll also have learnt why.

Here, it doesn't really matter.

I'll go back to our code, and we'll see what getting the item and quantity looks like, in each case.

Checking quantities - the code

The output is also more accurate. If you remember, "Butter chicken" was one of the recipes that our program told us we could make.

It checked that we had each ingredient, but didn't check **how much** we had.

It's much more useful now - it makes sure we have enough of each ingredient.

I'm not sure that we should be printing the ingredients that are OK. They clutter the output, and detract from the ingredients we need to buy.

But that's more of an observation than a definite objection. Including the OK items allows our cook to check that they really do have those ingredients.

Checking quantities - the code

If someone came down for a midnight snack and ate the yogurt, our program might not know about it.

If you want, you could use the `colorama` module, to print the output in different colours.

Pay attention to what you display to your users. Sometimes, too much information can make your user miss important detail.

Alright, we're now starting to get a useful program.

But it only produces output on the screen. Our cook will either have to write things down, or print the output.

Checking quantities - the code

That's just not good enough, these days. Users expect more from their technology.

Our program could order the items from an online food store, or send the details to a mobile phone. That would be what modern users expect.

We're not going to do all of that straight away, but you will do the first step.

Whatever we decide to get the program to do, it's going to need the data in a more usable form. So it's time for a challenge.

Solution: Create a shopping list challenge

How did you get on?

Your solution may be different to mine - there's no right or wrong here.

The important thing is that you get something you can iterate over, to get the ingredients and quantities.

We could use a dictionary here, or a list of tuples.

This section is about dictionaries, and a dictionary would be a good choice.

But often, you'll need to mix different data structures. I don't want to give the impression that you can only use dictionaries with dictionaries.

Solution: Create a shopping list challenge

So I'm going to create a list of tuples in my solution.

And we're going to find out that it wasn't the best choice 😊.

I've said, a few times, that you'll soon find out if you make the "wrong" choice.

I've also said not to worry too much about getting it wrong. But that doesn't necessarily stop students from worrying.

This is a good chance to see what happens if you do get it wrong, and why it doesn't have to be that big a deal.

Thinking Ahead

When you're starting to program, it's normal to focus on the specific bit of code that you're writing.

It's a bit like playing chess.

At first, you concentrate on your current move.

As you get better, you start to look ahead. You consider the move that your opponent will make, as a result of what you've done.

The better you get, the further ahead you look. A master chess player will be considering the state of the board, 10 to 15 moves in the future.

Thinking Ahead

It's the same situation with programming.

At first, you're trying to get one line of code right.

Then, you start to focus on the whole loop, or a complete function.

After a while, you're considering more complex blocks of code, like nested loops. You understand what each of the nested loops will be doing, within the outer loop.

When you've been programming for a while, and are starting to master programming, you'll be aware of how your code fits into the overall program.

You'll be writing something like our list of tuples, and you'll be aware of how it's going to be used, within the main program.

Thinking Ahead

Once you've learnt how the chess pieces move, the way to get better at chess is to

Play Chess!

When programming, once you've learnt the syntax of things like for loops, the way to get better at programming is to

Write Programs!

COMPLETE PYTHON MASTERCLASS

Solution: Create a shopping list challenge



DICTIONARIES AND SETS

PM-17-5

Wrong decisions don't have to be fatal

We decided to use a list of tuples, and searching through the list is going to be a bit tricky.

We saw how to use the `index` method of a list, to find an item, but we don't know what item to search for.

Items in the list are in two parts; an ingredient and a quantity.

We know the ingredient part that we want, but we don't know what quantity it'll have, if it's already in the list.

Wrong decisions don't have to be fatal

That means we can't search for a specific tuple. We'd have to iterate over the list, and compare the first item in each tuple.

Choosing a list of tuples, instead of a dictionary, wasn't the best decision here.

But we produced a working program, and it doesn't do a bad job.

A couple of videos ago, I suggested that you should just pick one, if you can't decide which option is best.

Here, I picked the wrong option - a list of tuples isn't as suitable as a dictionary, in this case.

But it's not the end of the world. Our program isn't as easy to modify as it could be, but we produced something that works.

Wrong decisions don't have to be fatal

In the process, we've learnt something. A dictionary is more appropriate here.

Always strive to produce the best code - that goes without saying - but don't let indecision hold you back.

Brief Summary

More importantly, we didn't have to make a lot of changes to the program.

In fact, the only changes we made to the main program were because of my bad decision.

But how bad a decision was it, really?

We had to make a few changes to the code, to use a dictionary instead of a list of tuples.

But we haven't had to completely rewrite everything. In fact, the changes we needed to make were minor.

Hopefully, that's convinced you to just go ahead and start coding, whenever you can't choose between more than one option.

Brief Summary

It didn't take long to realize that my decision wasn't the best one. And it was easy to fix.

When you've gained more experience, and are working on much larger programs, then changes will be more costly.

But don't let indecision hold you back, while you're learning.

Have a go and make mistakes. You'll learn a lot from your mistakes. You learn nothing from doing nothing.

setdefault vs collections.defaultdict

If you google for **python setdefault**, you'll find lots of posts talking about a **defaultdict**.

We're not going to look at **defaultdict** just yet. It's one of the objects in the **collections** module, and it works very like the dictionaries we've been looking at.

The difference is that it can automatically return a default value, and add a key to the dictionary, if a key doesn't exist.

setdefault vs collections.defaultdict

Often, using a `defaultdict` can be a good alternative to calling `setdefault` on a normal dictionary.

The documentation for `defaultdict` talks about ***subclassing***, and ***overriding*** methods. Those terms won't make sense until after the OOP section of this course.

We'll look at the `collections defaultdict` class later, once you've learnt about classes in Python.

The setdefault method

Alright, there are still a few dictionary operations that we haven't looked at, but we've covered the most common things that you'll want to do with dictionaries.

We'll check out the documentation, and look at the remaining dictionary methods, after the next video.

Before we do that, I'll demonstrate why our shopping list is more useful than just printing out the values.

If you remember, we modified the program to create a shopping list dictionary, rather than just printing it.

APIs and a mobile phone demo

At the end of the last video, I mentioned that we could do something more useful with our shopping list.

If we just print it out, our cooks will have to write the list down before they go shopping.

If we store it in some kind of data structure, as we've done, we can do something more exciting with it.

So in this video, I'm going to demonstrate something more exciting.

I'm not going to show the code for this, nor will I make it available in the resources for this video.

APIs and a mobile phone demo

This is purely a demonstration of what's possible. At this point in the course, you haven't learnt enough to be messing with your Google accounts.

It's possible, if you attempt too many logins too quickly, or get the authentication wrong, that Google will lock you out of your account.

Of course, if you're using an iPhone then that won't bother you as much 😊.

But if your main phone is an Android device, then being locked out will be inconvenient, to say the least.

Another reason I won't be making this code available, is that it uses an unofficial API. We'll see the demo, then I'll talk a bit about APIs.

APIs and a mobile phone demo

I have to run this demo from a terminal, rather than from IntelliJ's Run pane. That's because we're using the Python `getpass` module.

The `getpass` function is similar to `input`, but it doesn't echo the input to the console. If I used the `input` function here, my username and password would be visible on the video, which obviously isn't a good thing.

If you use `getpass` in your code, be aware that it doesn't work in the IntelliJ Run pane. It does work in the IntelliJ terminal, as well as a normal terminal and Windows command prompt.

API

API is an acronym for Application Programming Interface.

It defines the objects and functions that are made available, and how to use them.

We've already used a lot of APIs in this course. For example, we used the `list` API, when we worked with lists.

The documentation described the `list` object, and the methods we can use to manipulate lists.

Shortly, we'll look at the documentation for Python's `dict` API.

You won't normally find programmers referring to these as APIs, but that's what they are.

API

If you create a system that other programmers will find useful, you may decide to create an API for that system.

For example, Google make all sorts of data available. Raw data can be useful, but providing a way to interact with that data is even better.

Google provide a YouTube API that lets you query YouTube, to get statistics on your YouTube channels.

Those statistics are available when you log into the YouTube website, but you might want to write your own program to analyse the statistics. Their Channels API lets you do that, from your Python code.

API

Students sometimes ask how they can allow a user to log in, with a password, before being granted access to a program. Our advice is don't.

Authenticating users is extremely difficult to get right, and comes with all sorts of problems. You may be aware of websites getting hacked, and all their customers' passwords being stolen.

If you need to perform authentication, let someone else handle it. Google, Apple and Facebook, to name a few, spend a lot of money on their authentication systems. They have the resources to make them secure, and the money to spend on development and testing.

API

I'll switch to a browser, to show Google's documentation for their OAuth API that you can use from Python:

<https://github.com/googleapis/google-api-python-client/blob/master/docs/oauth-installed.md>

Unofficial API

An official API is supported by whoever created it. More importantly, there's an implicit guarantee that the API won't be changed in a way that breaks client code.

If you write code that uses a Google API, for example, then Google will make sure that any changes they make won't cause your code to break.

The Google Keep API that we've used is an unofficial one. A programmer called Kai created a module to use that API, and has made it available. But if Google change the way Keep works, any code using Kai's API might break.

Unofficial API

If you're an experienced programmer, you'll have no trouble finding Kai's API and working out how to use it.

If you're new to programming, it's a bit early in the course to be experimenting with your live Google or Microsoft accounts. By the end of the course, you'll be able to make sense of published APIs, and use them in your programs.

tuples as keys

But it does mention some restrictions. We know that tuples are immutable, so they could be used as a dictionary key.

But if a tuple contains a list, then it's no longer suitable as a key.

So a tuple can be used as a dictionary key, as long as the items in the tuple are also immutable.

tuples as keys

Suitable as a key (hashable)	Unsuitable as a key (not hashable)
(1, "Nightflight")	(1, ["a", "b", "c"])
97	
"Tim"	

The left hand column shows some immutable objects that we can use as dictionary keys. The first item is a tuple that only contains other immutable items.

The tuple on the right hand side **isn't** suitable. It contains a list, and the list could be mutated. Our code could add items to it, or delete them. Because of that, we can't use this tuple as a dictionary key.

Python won't attempt to create a hash for a tuple that contains things like lists. You'll get a **TypeError: unhashable type** if you attempt to hash the tuple on the right.

LIFO: Last In First Out

That's what **LIFO** means - it's an acronym meaning **Last In, First Out**.

Imagine someone wearing bangles on their wrist. If they put on a red bangle, a blue bangle and a green bangle, then they'll take them off in the opposite order.

The green bangle has to come off first - last on, first off.

Another example would be a stack of coins. The first coin you'd take off the stack would be the last one you put on it.

In fact, a stack is a very common data structure, in programming.

FIFO: First In First Out

The opposite of **LIFO** is **FIFO** - **F**irst **I**n, **F**irst **O**ut.

That's like a queue at a bus stop. The first person to arrive at the bus stop will be the first person to leave, when they get on the bus.

We're assuming civilized behaviour here, of course.

FIFO: First In First Out

Another example of a **FIFO** queue would be putting items away in your fridge.

If you bought more milk, you'd put the new bottle behind any that were already in the fridge.

When you take a bottle of milk out of the fridge, you'd be taking the first bottle that was put in.

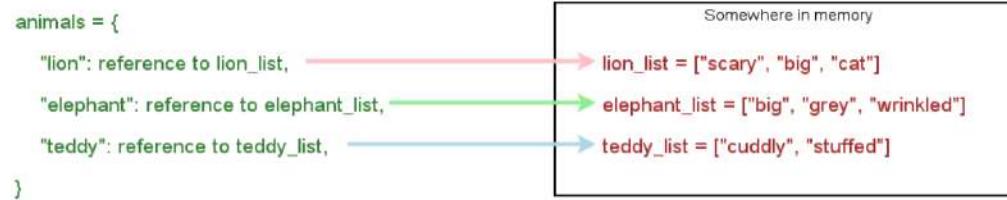
If you don't do that, you could end up with a bottle of milk that has been in the fridge for months. Yeuch!

Shallow copy step-by-step

In this video, we'll go through the shallow copy process, step by step.

I'll use slides to show what's happening, when you perform a shallow copy of a container that contains mutable objects.

Shallow Copy of a dict Containing Lists

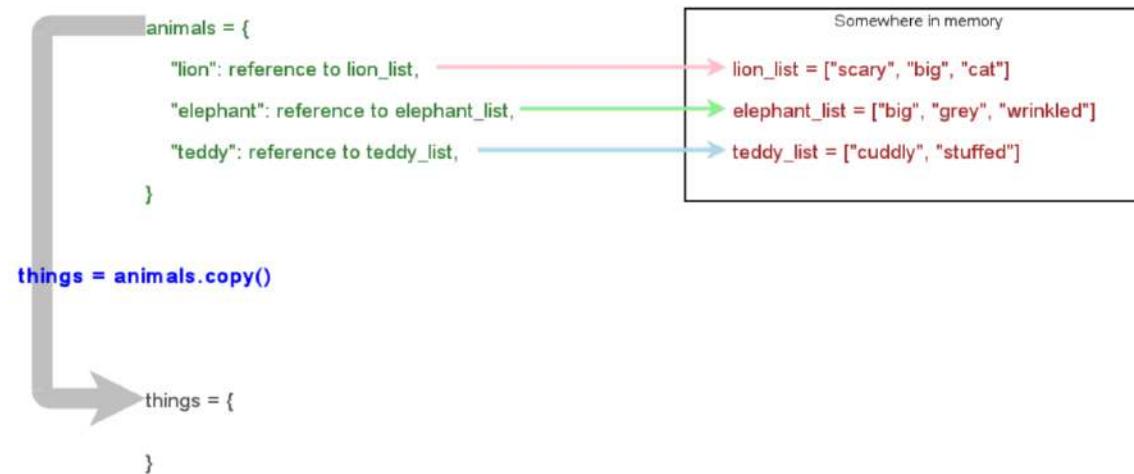


We start off with a dictionary whose values are lists. This is the same dictionary that we had in the previous video.

Note that the values that are stored, inside the dictionary, aren't lists. They're references to lists.

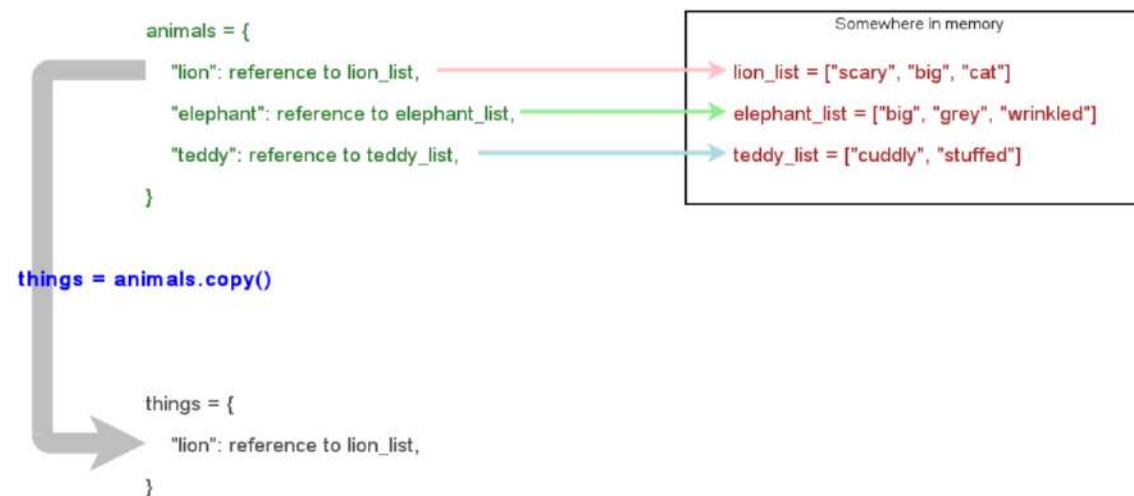
The value for `"lion"` is a reference to a list, and the list exists somewhere else in memory.

Shallow Copy of a dict Containing Lists



When we use the **copy** method to create a copy of the dictionary, Python starts by creating a new dictionary. We called the copy **things**.

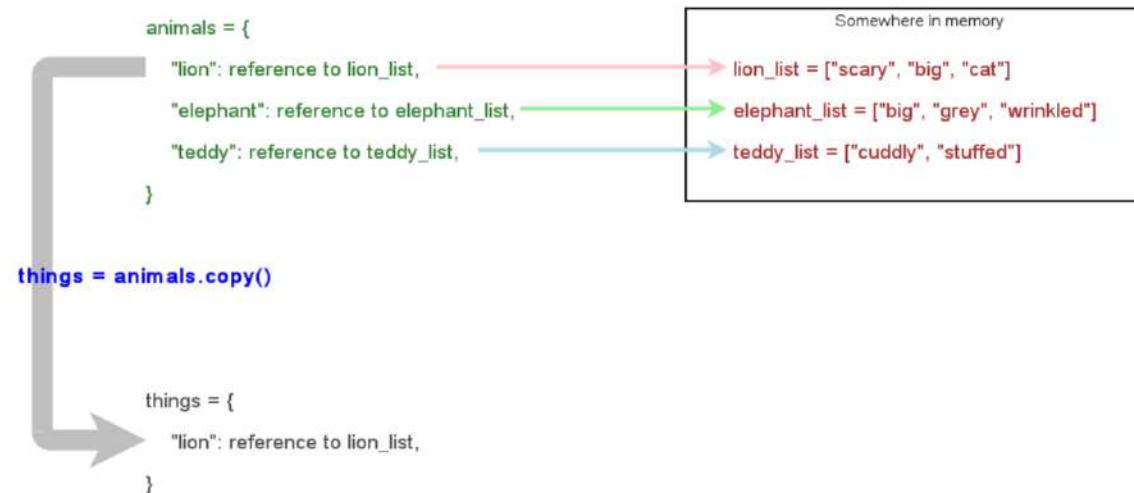
Shallow Copy of a dict Containing Lists



Python then goes through each of the dictionary keys. It copies each key, and its value, into the new dictionary.

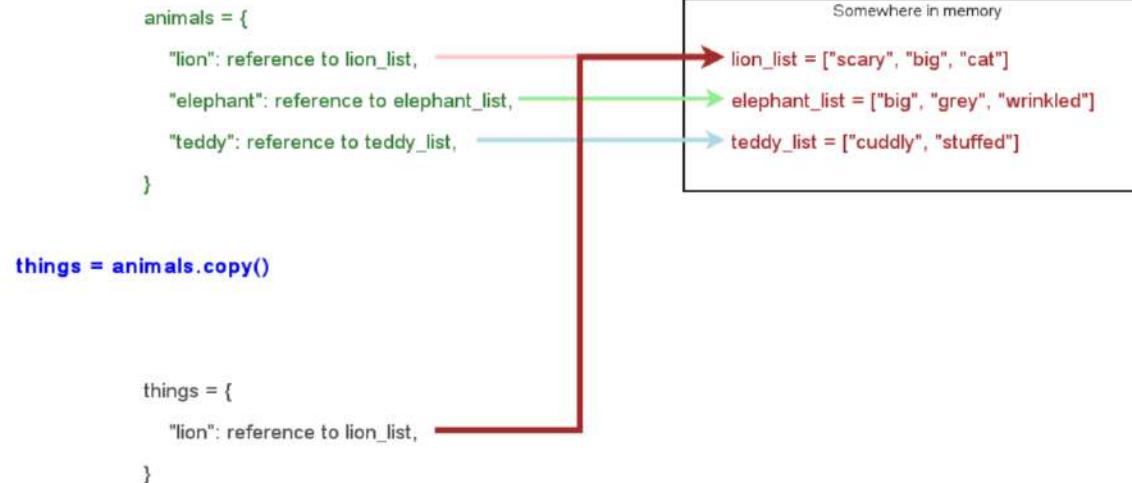
Here, we can see the key "lion" has been copied into the `things` dictionary.

Shallow Copy of a dict Containing Lists



The value is also copied. That means the value for "lion" is a reference to the same list that's being referred to in the `animals` dictionary.

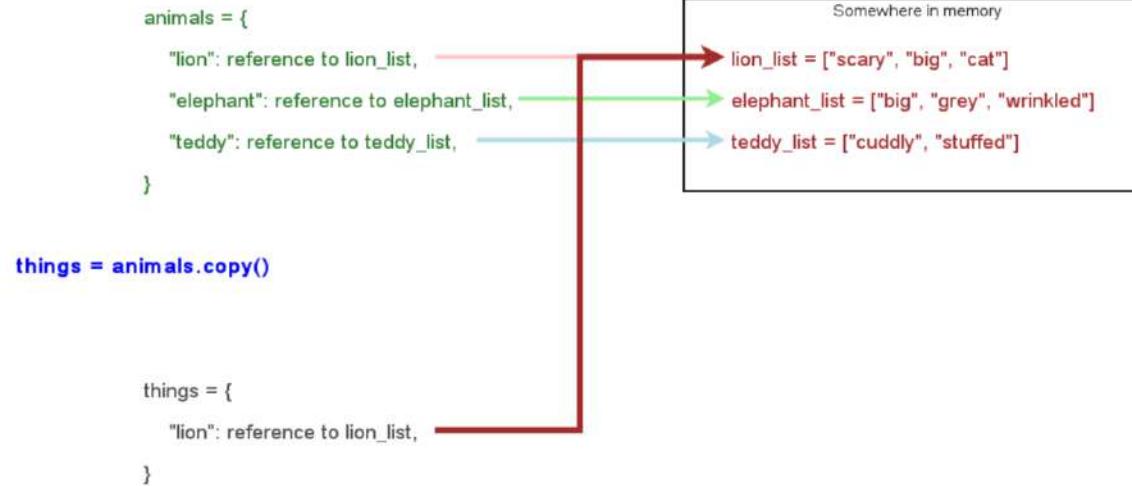
Shallow Copy of a dict Containing Lists



The value of the "lion" key in `animals`, and the value of the "lion" key in `things`, both refer to the same list.

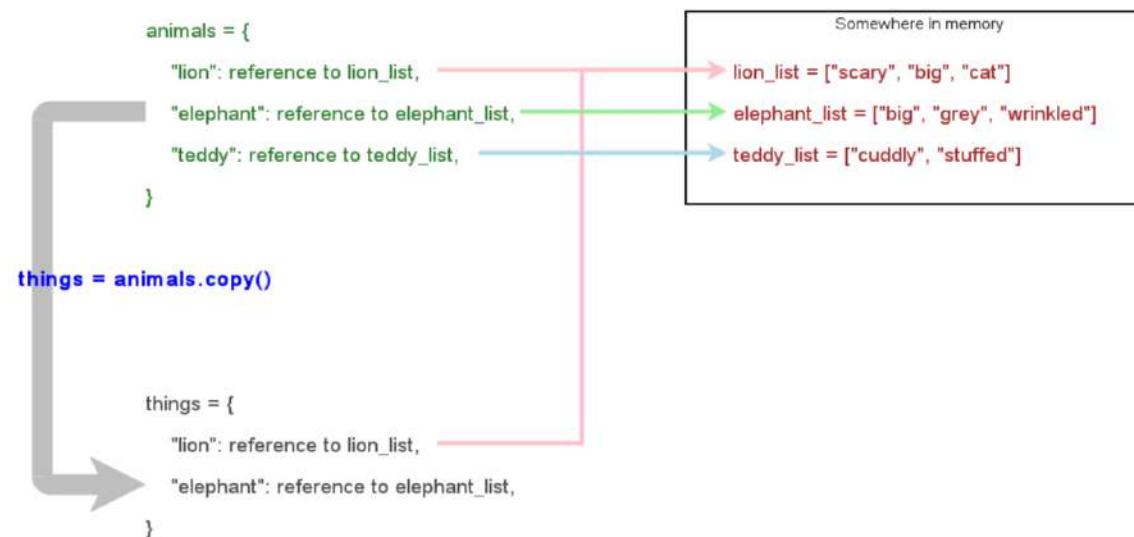
We're performing a **shallow copy** here. The list **isn't** copied.

Shallow Copy of a dict Containing Lists



A shallow copy copies references. It **doesn't** make a copy of the things that are being referred to.

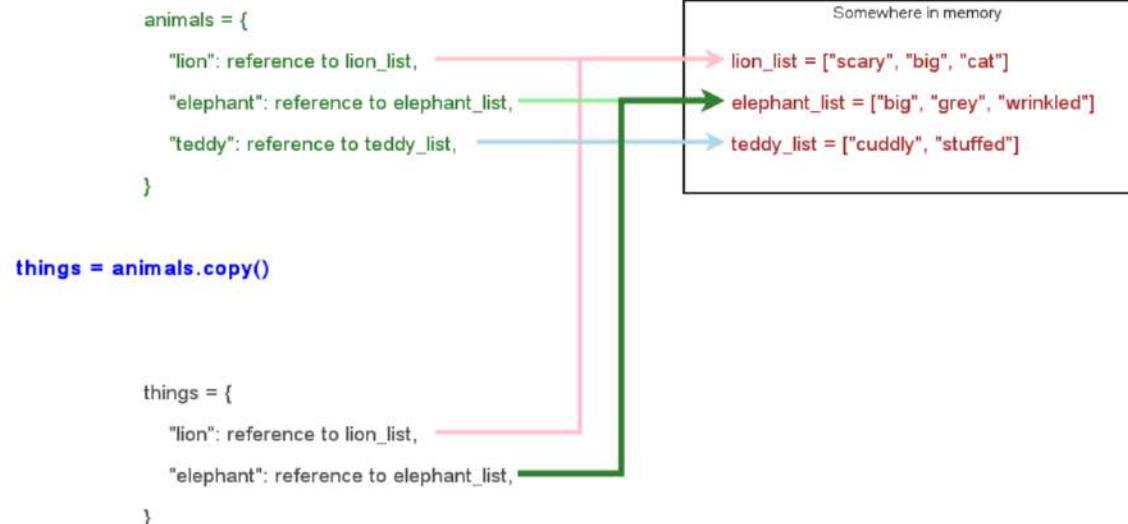
Shallow Copy of a dict Containing Lists



The next key is copied in exactly the same way.

Here, Python copies the "elephant" key into the `things` dictionary.

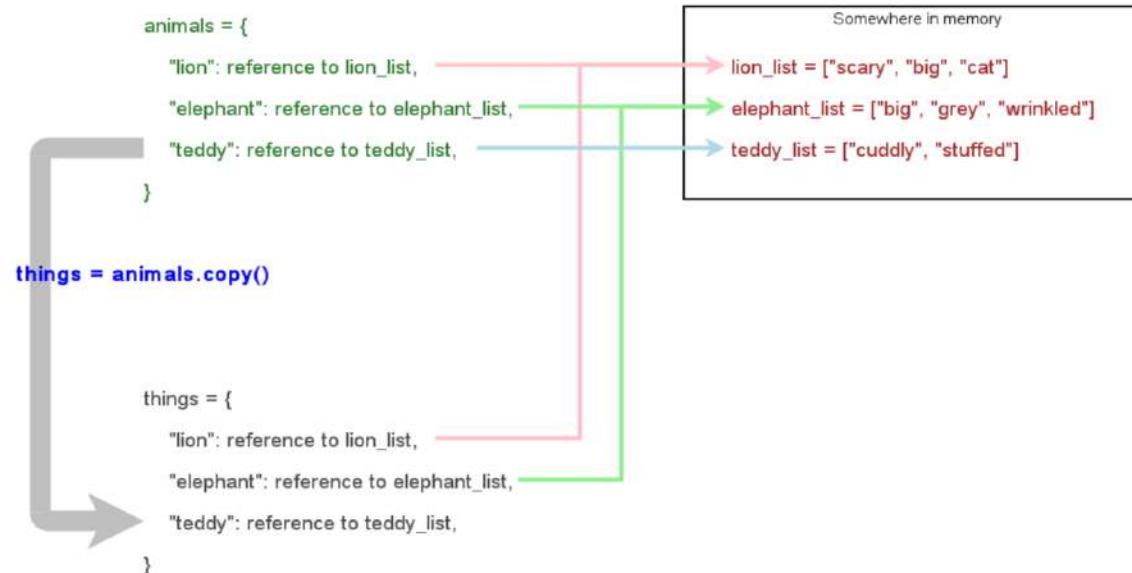
Shallow Copy of a dict Containing Lists



Once again, it's the **reference** to the list that gets copied, **not** the list itself.

There's only one list for "elephant", and the values in both dictionaries are referring to it.

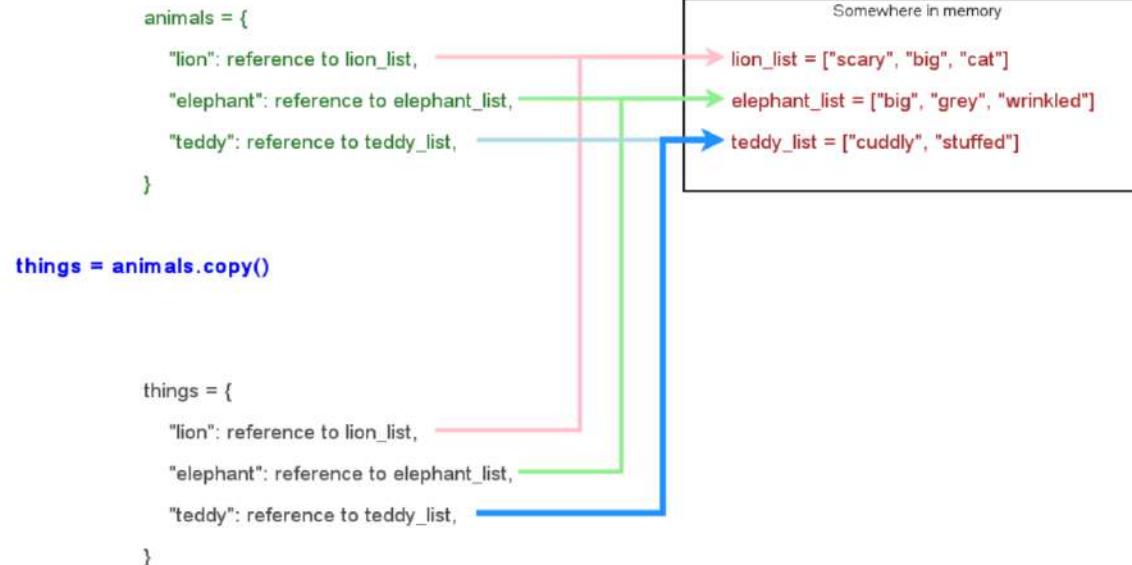
Shallow Copy of a dict Containing Lists



Our original dictionary only has 3 keys - which is a good thing, or these slides would go on for a long time!

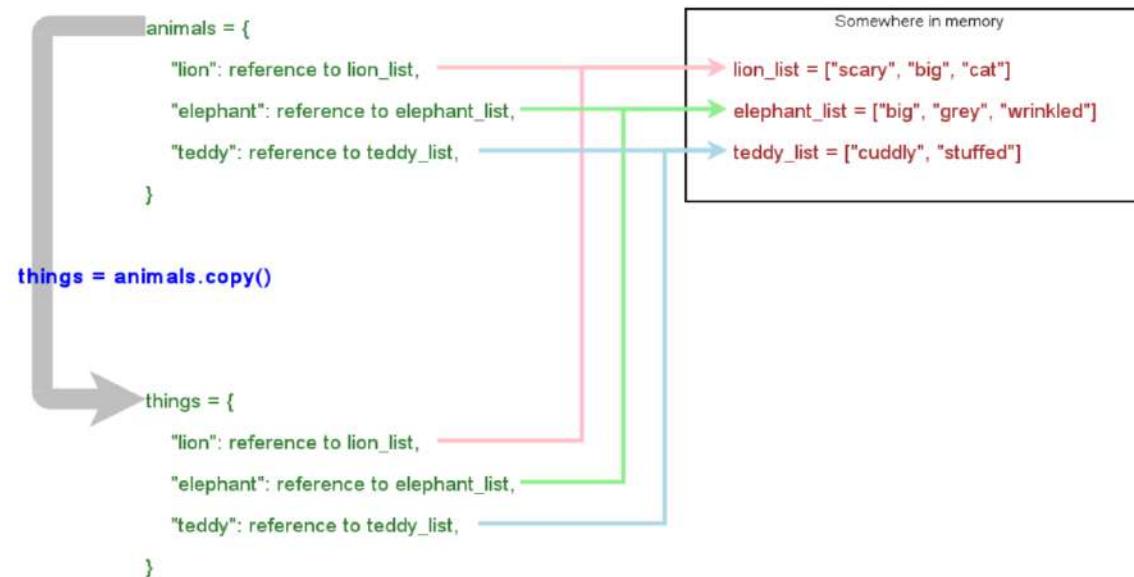
Once again, the key is copied from `animals` into `things`.

Shallow Copy of a dict Containing Lists



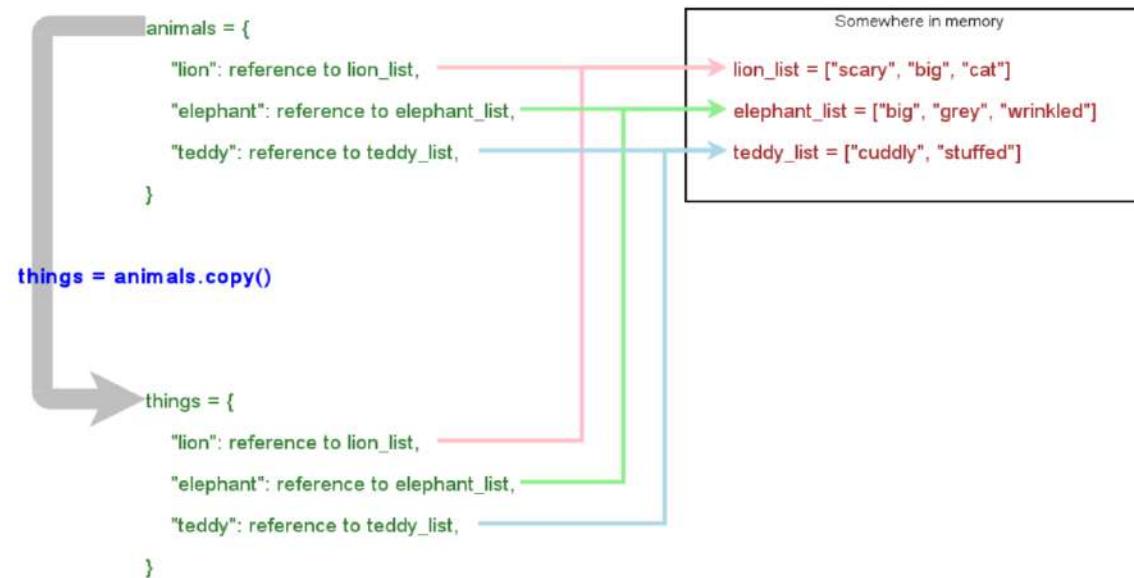
And once again, it's the **reference** to the `teddy_list` that's copied.

Shallow Copy of a dict Containing Lists



We end up with a copy of the original dictionary, containing the same keys and values. That means the values will continue to refer to the same objects in the copy.

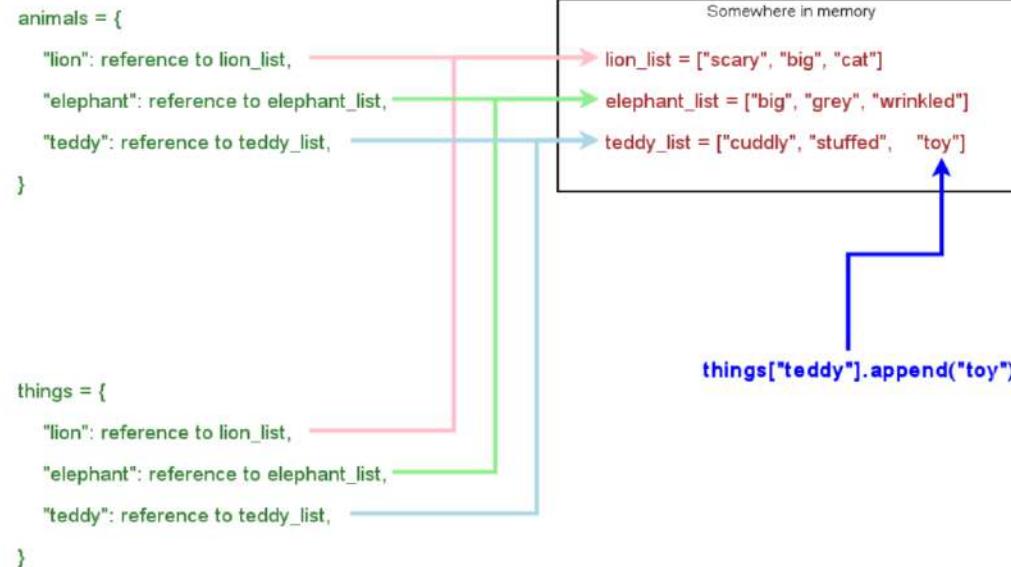
Shallow Copy of a dict Containing Lists



As you can see from these slides, there's only one `lion_list`, and both dictionaries refer to it.

The same for `elephant_list` and `teddy_list`.

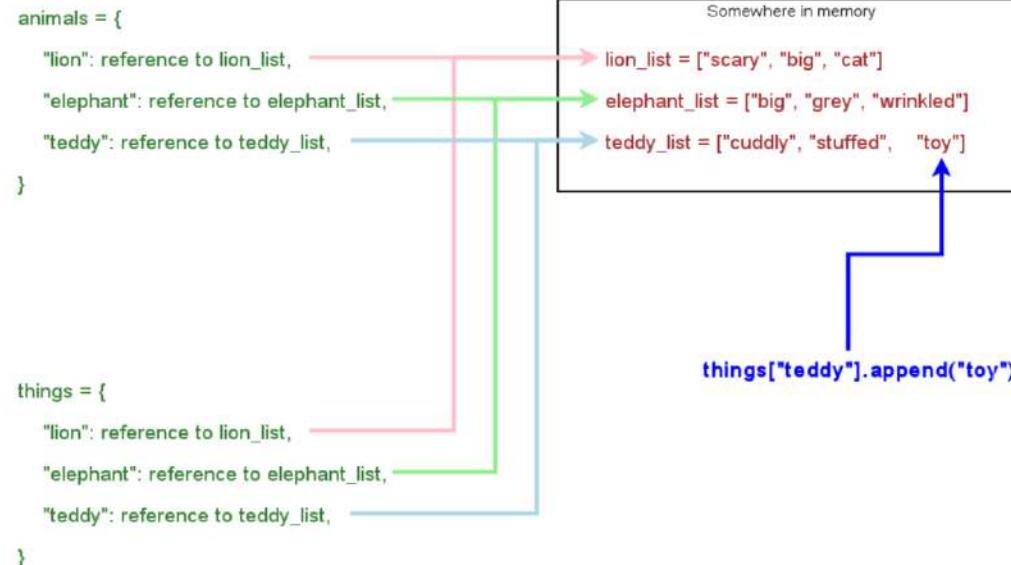
Shallow Copy of a dict Containing Lists



If the values are immutable, such as strings or ints, then we don't really care about references being copied.

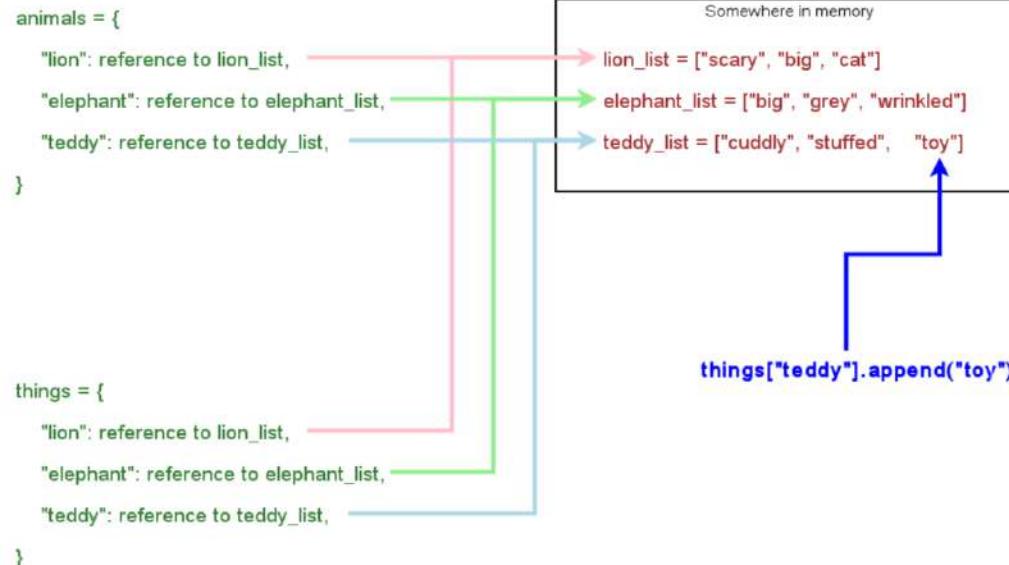
If you can't mutate a value, then none of this really matters.

Shallow Copy of a dict Containing Lists



Where it becomes important to understand all this, is when the values refer to **mutable** objects.

Shallow Copy of a dict Containing Lists



As we saw, adding "toy" to the "teddy" value in **things**, resulted in "toy" appearing in the "teddy" value for **animals** too.

Now that you know how a shallow copy works, that shouldn't be surprising any longer.

Shallow Copy vs Deep Copy

If you need to create a **deep** copy of an object, Python has a **copy** module that you can use.

The **copy** module provides a **deepcopy** function.

Performing a deep copy **will** create copies of lists, dictionaries and other mutable objects, that are contained in whatever you're copying.

If you want to experiment with a deep copy, check out the documentation at
<https://docs.python.org/3/library/copy.html>

Using the function is easy. The hard part is understanding the difference between a deep and shallow copy. And you understand that now 😊.

In the next video, we'll perform a **deep** copy to demonstrate the difference.

Simple deep copy solution

The **deepcopy** function uses a technique called **recursion**. When it finds an object that can contain other objects, it calls itself again to copy those containers.

If you wanted to write a complete **deep copy** function, you'd need to use recursion. We'll cover recursion a bit later in the course, when we revisit **functions**.

Alright, that's the end of the practical use of dictionaries. In the next few lectures, we're going to look at the theory.

A dictionary can contain millions of keys, but retrieving a value for a key is very fast. We'll look at the theory of how that works, using something called **hashes**. We saw them mentioned in the documentation, earlier. Now it's time to find out what they are.

Hashes can be Used Like Indexes

That's a very important point.

If you can calculate a hash, and use it to go directly to an entry in a hash table, then accessing the data becomes very fast.

If we assume a perfect hash function - one that produces no collisions - then we can retrieve values from a table containing millions of entries, in the same time as a table containing only a handful of entries.

It doesn't matter how many items are in the hash table. We calculate the hash, and can retrieve the value with a single operation.

Dictionaries are Fast

That's why retrieving a value from a dictionary, using its key, is so fast.

The speed doesn't depend on how many items are in the dictionary, it depends on how fast the hashing function can calculate the hash.

Accessing an item in a dictionary will be slightly slower than indexing a list, because of the time taken to calculate the hash. But it's largely independant of the number of items you've stored in the dictionary.

The hash functions, that most dictionaries use, have been designed to execute quickly.

Hash functions

That's a good tip when reading any documentation - if it looks like complete nonsense, move on.

If a section of documentation is going to be useful to you, then you'll know everything you need to understand it.

If you can't make sense of it, then it's no use to you. Sitting there, staring at it and panicking, won't achieve anything.

If I was asked to write a hashing function, using algebraic coding, I'd have to spend a week or so brushing up on the maths behind it.

It would be more efficient for me to ask someone else to do it. I'd ask J-P, but he's already told me that none of this **Algebraic coding** section makes any sense to him, either 😊.

Hash functions

So don't panic when you see stuff like this. You'd have to be programming for years, before anyone would consider asking you to write a hashing function. Even then, it would be a job more suited to a mathematician than a programmer.

Skip the bits you don't understand, focus on the bits you do, and you should come away from this article with the following points:

Hashing Functions

- A hash function produces fixed-size hash values from its input. The hashes are usually integers, but don't have to be.
- There are many different ways to implement a hash function. There's no single way to write one, and different algorithms have their own strengths, weaknesses and applications.
- A hash function produces values that can be used to index a fixed-sized data structure, called a **hash table**. If the hash table can hold 500 items, then the hash function should produce 500 distinct hashes.
- A hash (or hash code) doesn't have to be unique. For example, two different strings can have the same hash. That's known as a **collision**.

Hashing Functions

- There are several different strategies for handling collisions. One of the simplest, is to dump all keys with the same hash into the same "bucket". You then compare each item in the bucket with the original key, to see if it exists.
- Because handling collisions is slower than indexing directly into the table, it's important that a hash function produces as few collisions as possible.
- The best case is that every key has a unique hash.
- The worst case is that every key has the same hash - if that happens, the hashing function isn't suitable for that particular application.

A really bad hashing function

We've now got an idea of what a **hashing function** is, but it's all been very theoretical.

In this video, we'll write our own hashing function, and see how it can be used in practice.

The title of this video should have made the point that this is going to be "a really bad hashing function".

We're going to look at a very simple implementation, and you wouldn't use anything this simple in your code. But as I've already mentioned, you'll probably never have to write your own hashing function.

A really bad hashing function

Python comes with one built in, and there are modules in the standard library, that you can also use.

So just to be doubly clear, this video is purely for educational purposes. It's intended to help you understand how hashes are used. It's very definitely **not** code that you'd use.

A really bad hashing function

Every character is represented by a number, when it's stored in the computer. In the very early days of computing, that used to be the ASCII value. ASCII stands for **American Standard Code for Information Interchange**.

Google for **ASCII** if you want to know a bit more about it - it's useful to understand some of the history of computers. But that's not essential for what we're doing here - all we need to know is, that every character is represented by a unique number.

These days, we don't use ASCII, we use **unicode**. ASCII can only represent 127 different characters. That's not enough to handle all the languages and character sets, that are used throughout the world.

Bullet Points from the Overview Section

In the Wikipedia Overview about hash functions, there were three bullet points.

Let's see how our function satisfies those three points.

1. Convert variable length keys into fixed length (usually machine word length or less) values, by folding them by words or other units using a parity-preserving operator like ADD or XOR.

Our **simple_hash** function maps any string into a fixed sized value: an integer in the range 0 to 9.

We only use the first character in the string, but a hash function would normally use all characters.

Bullet Points from the Overview Section

We could add the ordinal values of each character, and use the total of all the ordinal values. But we're deliberately keeping this simple, and want a small integer value to be produced.

2. Scramble the bits of the key so that the resulting values are uniformly distributed over the key space.

Our "scrambling" is very basic. We take the remainder after dividing by 10. More serious strategies include stripping off all but the low 64 bits of the result (if a 64 bit hash is needed).

Bullet Points from the Overview Section

3. Map the key values into ones less than or equal to the size of the table.

We do that by only using the last digit. That gives a value in the range 0 to 9. The length of our "table" will be 10.

Notes about Python's hash Function

First, Python's hash function randomizes the hashes that it produces. You'll get the same hash for any particular string, while your program's running. But each time you run the program, the hash codes will be different.

That was introduced in Python 3.3, to prevent Denial of Service (DoS) attacks on web servers using Python dictionaries.

So don't be confused if you get different hashes for the strings, each time you run the program.

Notes about Python's hash Function

Second, integers that have their high bit set are interpreted as negative. That's why some of the hashes print out as negative.

Google for **twos complement** if you want to learn more about how negative numbers are stored, in a computer.

Notes about Python's hash Function

Finally, you can see why I've written that very simple hash function.

If we try working with tables that can contain trillions of items, we're going to get in a right mess. Our hashing function is very simple, but it allows us to work with a very small table - only 10 items - so that we can see what's going on.

Really finally - following on from the last point - don't be worried that Python creates hash tables with trillions of items, for its dictionaries. It uses techniques such as **sparse arrays**, so that only the non-empty values get stored.

Simple Dictionary Implementation

In this video, we're going to implement our own dictionary, using a hash table.

Before I continue, you should know that your computer doesn't understand lists, dictionaries, tuples or any other kind of data structure.

All your computer knows about is its memory.

The closest data structure to your computer's memory is a one-dimensional array.

The memory addresses are similar to indexes into an array.

Each memory address can access one memory location.

Simple Dictionary Implementation

Anything more complicated than that is implemented by your programming language. That's Python, in our case.

Python provides implementations for lists, tuples, sets and dictionaries.

Those are the data structures that are built into Python.

If the only data structure that a computer understands is a linear array of memory locations, it should be obvious that any data structure can be implemented, using an array.

At a low level, that's all the computer has to work with.

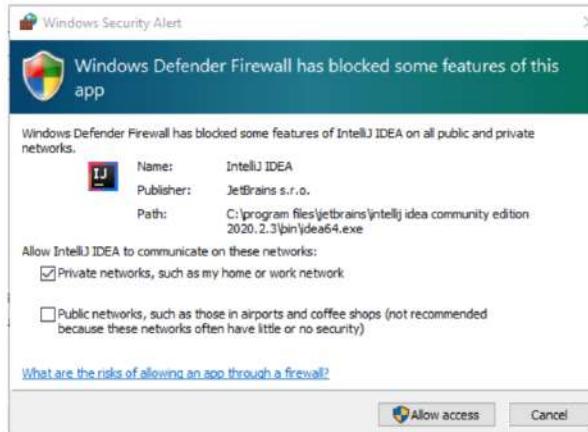
Simple Dictionary Implementation

We're going to use fixed sized lists, to create a basic dictionary implementation, using a hash table.

A Python list is quite similar to an array, and a fixed-size list is almost identical.

Windows Defender

If you get a dialogue like this one, on Windows, make sure you tick the box to **Allow access on Private networks**.



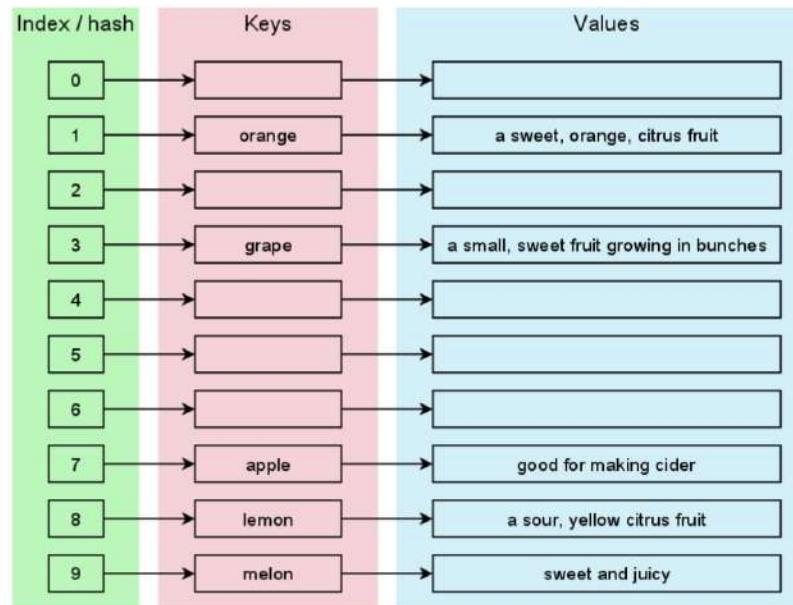
Click the **Allow Access** button after changing the selection to **Private networks**.

If you don't, you'll have to configure the Windows Firewall yourself, and that's a bit tedious. Tick the box, to allow the Firewall to be configured automatically.

Keys and Values Hash Table

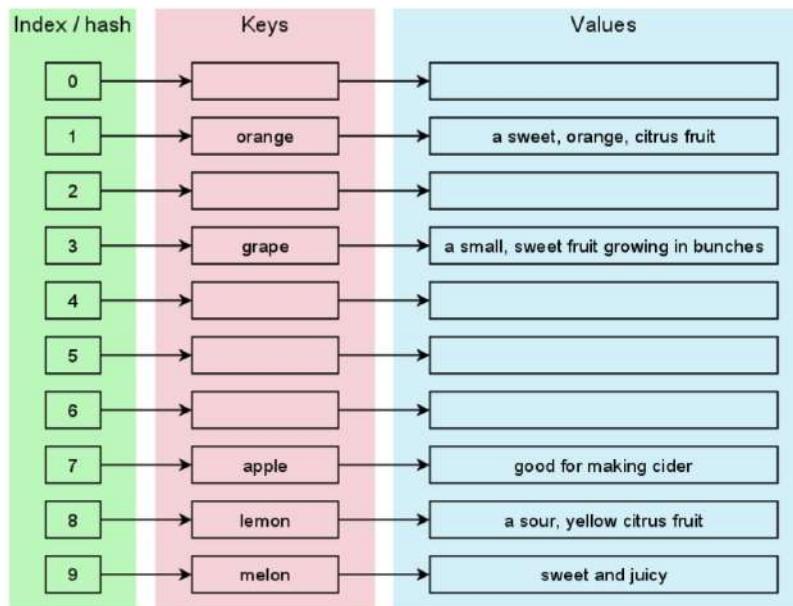
We saw that our hashing function produced the hash **1** for "orange".

That means the key "orange" goes into the keys table at index position 1.



Keys and Values Hash Table

Its value goes into the values table at the same index position. The same for "grape". The string "grape" hashed to the value **3**, so the key and value go into the tables at index position **3**.



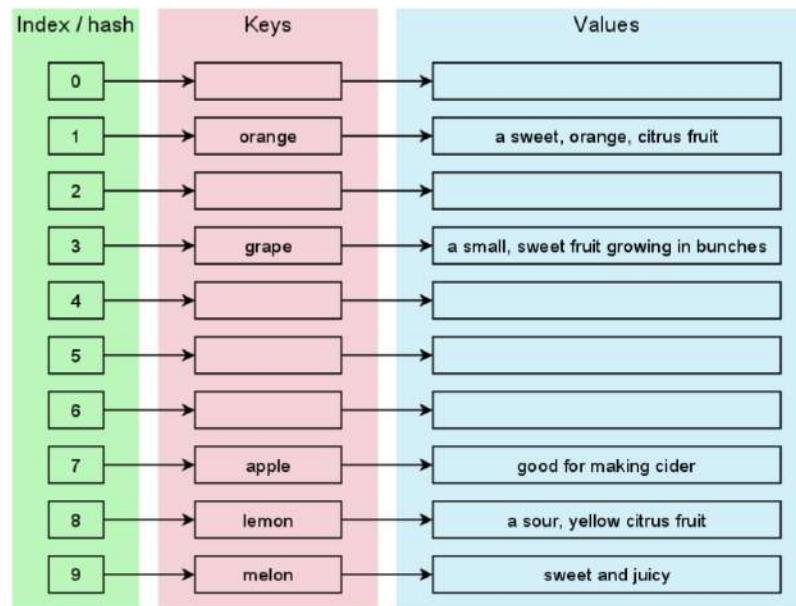
Keys and Values Hash Table

I've included the index numbers in the leftmost column of the slide, to make it easier to see what's happening. We don't store the hashes - we'll calculate them as they're needed.

Index / hash	Keys	Values
0		
1	orange	a sweet, orange, citrus fruit
2		
3	grape	a small, sweet fruit growing in bunches
4		
5		
6		
7	apple	good for making cider
8	lemon	a sour, yellow citrus fruit
9	melon	sweet and juicy

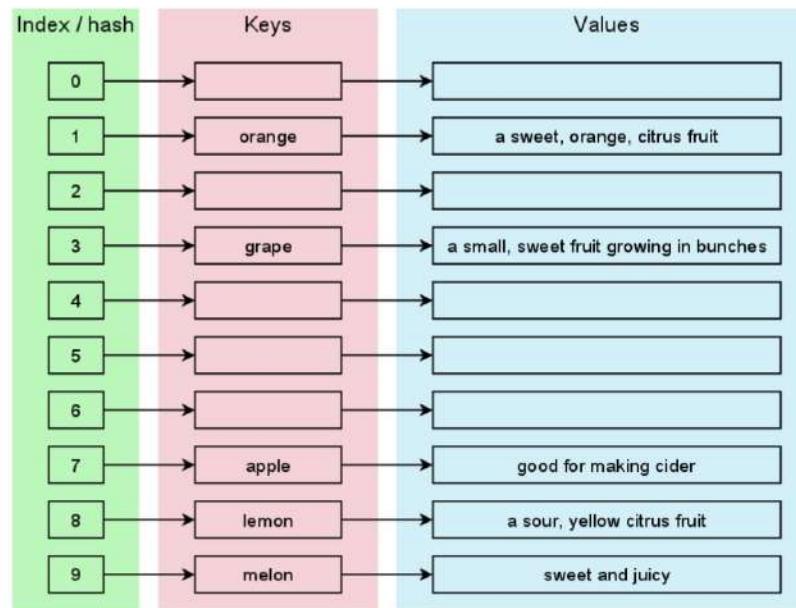
Simple Dictionary Implementation

This is a simplified description of how Python dictionaries are implemented, behind the scenes.



Simple Dictionary Implementation

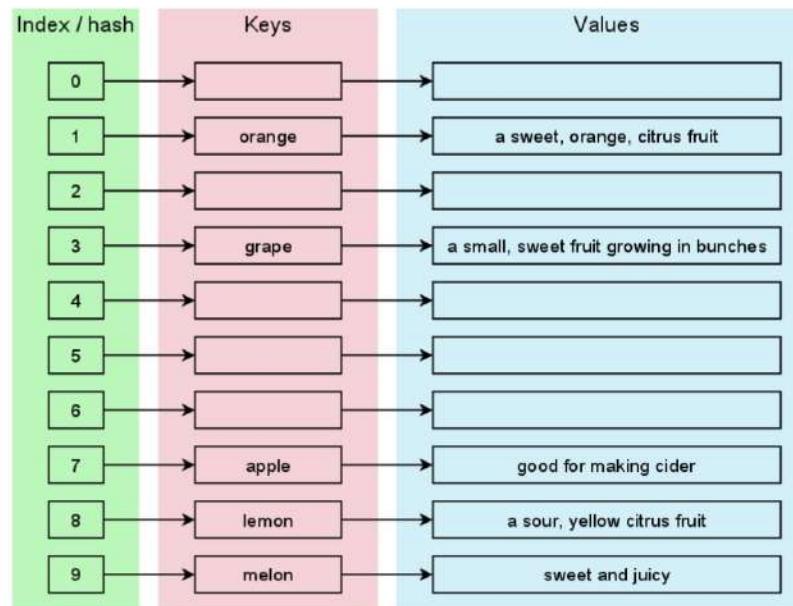
Python's implementation is a bit more complicated than this. It doesn't store all those empty strings, for one thing.



Simple Dictionary Implementation

But the basic principle is the same.

The hash of a key is used to retrieve the value from the hash table.



Completing our simple dictionary implementation

We haven't written a complete implementation of a dictionary, but this code does demonstrate how Python's dictionaries work.

The real implementation works very similar to this.

It uses a hash table, to allow keys to be accessed very quickly.

Hashes of the keys are used as the indexes into the hash table.

Completing our simple dictionary implementation

Unfortunately, our implementation is broken.

We saw that hashes can have collisions - that's when 2 different keys have the same hash.

With a serious hash function, collisions will be rare and the Python **dict** contains code to handle collisions.

Our dictionary doesn't handle collisions.

It also uses a really bad hashing function.

That results in over half of the keys producing the same hash.

Completing our simple dictionary implementation

Retrieving a value for a key is very fast.

Importantly, it's fast no matter how many items are in the dictionary.

We calculate an index position, and can go directly to that position to retrieve a value.

Compare that to finding an item in an unsorted list.

You'd check the first value, to see if it's the one you want.

You'd then do the same with the next value, and so on, until you either find what you're looking for, or you reach the end of the list.

Completing our simple dictionary implementation

As the list gets larger, the number of checks you'd need also gets larger.

On average, you need to check more and more values to find the one you want.

Getting a value from a dictionary takes the same amount of time, no matter how many items are in the dictionary.

That's called **constant time**, or **O(1)**.

We'll see what that means, later in the course, when we learn **about Big O notation**.

Hash Functions and Security

Ok, we're going to finish off our discussion of hash functions with a quick look at another use for them. They're used a lot in security.

In case you don't already know this, you should never, **never, NEVER** store passwords as plain text.

Hash Functions and Security

There have been many cases where a company's password database has been stolen, providing the criminals with large numbers of email addresses, and the corresponding passwords.

The very least that should be done, is to store a hash of the password.

When the user supplies their logon credentials, the hash of the password they type is compared to the hash in the database.

Importantly, **the original password is never stored.**

Some Sites do Store Passwords

If a website is able to tell you what your password is when you forget it, then they're storing your password as plain text.

In that case, it's only a matter of time before someone steals your password.

Be very careful on websites like that. Don't use the same password anywhere else, and definitely don't let the site store your payment card details.

Hashes are Not Reversible

Hash functions are one-way.

We saw that "lemon" and "banana" hashed to the same value, when we wrote our simple hash function.

If different keys can produce the same hash, then it should be obvious that you can't get the key back from its hash.

Our `simple_hash` function was very simple, but the same applies to real hash functions. You can't reverse the process to find out a value from its hash.

Cracking a Hashed Password

Although you can't reverse a hash, there's no way to retrieve the key from the hash itself - that doesn't mean hashes are totally secure.

Attackers can use a **brute force** approach.

That involves generating millions of keys and hashing them. When a hash matches a hash in the stolen database, they've found out what the password was.

Probably. They might not know the exact password if there's a collision, but a good hash function should minimise collisions.

Cracking a Hashed Password

A brute force attack can take a long time.

But if the criminals set their computer away, trying to crack a database with millions of entries, for a week or more...

If it cracks only 10 percent of them after a week, that's still a lot of passwords.

Even worse, they then share their cracked databases on-line.

Avoid Performing your Own Authentication

The obvious thing to do, in a video about **Hash functions and security**, would be to show you how to hash passwords.

I'm not going to do that.

I don't want to encourage you to store passwords, in any form - even hashed.

Avoid Performing your Own Authentication

If you really want to do that, then you'll want to research computer security thoroughly.

It's quite easy to produce something that works. It's much, much harder to produce something that's secure.

So I don't want students going away from this course, thinking they know how to store passwords safely, then getting fired because thousands of customers' login credentials were stolen.

Tim's just Paranoid

Just in case you think I'm being over cautious, have a look at this Wikipedia entry, at
https://en.wikipedia.org/wiki/List_of_data_breaches

The numbers of users' records that have been stolen is staggering.

There are some big names in that list, including Facebook, Google and the US Ministry of Defence.

If your Twitter password is the same as your GMail password, then an attacker can use the details they steal from Twitter to access your email.

And once they've got into your email, they can reset all your other passwords - using the "Forgotten password" link that most websites have.

Python's hashlib Module

So we're not going to show you how to hash passwords. You'll need to learn a lot about computer security, before you can safely attempt authentication code.

What we will do, is see how to detect if a file, or email message, has been changed - including being tampered with.

hashes in Version Control Systems (VCS)

Another use is in version control systems.

You upload your Python code files to a version control system, such as github.

After you've worked on your code, you'll then upload the latest versions.

The VCS software calculates the hash of all your files, and only uploads ones where the new hash is different to the hash of the old version.

That avoids uploading files unnecessarily, and also means that the timestamp of files doesn't get changed, if the file wasn't changed.

Dictionaries and hashes

Alright, that was a quick look at secure hashes. We didn't want you to think that hashes are only used by dictionaries, which is why we've included this video.

It's been a bit artificial, because the string we hashed was stored in our code – the `python_program` string.

In the next section, we'll read the data from a file instead.

So if you're struggling to appreciate why all this is useful, don't worry. It will all make sense when we use a more realistic example.

Introduction to Android-Tim

Before we resume coding in the next video, I wanted to bring something to your attention.

I've been struggling with a constant coughing problem since October 2019. It's made the recording of videos in this, and other courses, much more difficult, since I constantly have to edit out coughs. The coughing gets worse, the more time I spend recording.

This has slowed down the rate of production by a big factor.

I have had all tests under the sun (including a test for Covid, which thankfully came back negative) and in short, it's a condition that has stumped the doctors, who have more or less said I have to live with it. In the grand scheme of things, coughing is not really a problem, and under normal circumstances, it's more of a problem for those in the same room hearing me cough. But from the point of view of recording videos, it's a problem.

Introduction to Android-Tim

A solution had to be found, and we've found a way to get the computer to assist with the audio, leaving me to record the videos.

This video, along with the last video, used audio generated by the computer, and not actually spoken by me. The voice sounds like me, it's just generated by a computer. I dare say I could have not said anything, and few people would have noticed.

But in the interests of full disclosure, I wanted to be upfront and say that the audio of this video, and the previous one, was generated by a computer. It's my intention, moving forward, for all videos to be generated the same way.

Introduction to Android-Tim

Let's do a test though.

I'm going to record myself speaking for a while, and then put the computer-generated audio immediately after, so you can compare them.

Ready?

Introduction to Android-Tim

Right, this is Tim's version of talking. You should be pretty familiar with my voice, if you have got this far into the course.

This is the first time you've heard my real voice in this video. Everything previously has been computer generated.

Now I'll swap over to the computer-generated audio.

Introduction to Android-Tim

This is the computer-generated audio of Tim talking. It should sound the same. Isn't technology wonderful !

The computer-generated audio will continue now for the rest of the video.

Introduction to Android-Tim

What does this mean for you ?

Nothing's really changed. The actual video content is still being created by us, and the videos are still being recorded by Tim. We're just getting help with the audio, to help out Tim, who due to his coughing is unable to record at anywhere near the volume and quality of videos he has done in the past.

If you have any comments about what we are doing here, please post in the "Q-and-A" section of this course. Alternatively, send a private message to Tim or J-P, or go to Tim's website and leave him a comment at Tim Buchalka's Blog - timbuchalka.com.

Right, I'm going to end the video here; lets get back into programming in the next one. I'll see you, and Android Tim will talk to you, in the next video.

Introduction to sets

We'll finish this section by looking at the last of Python's built-in data structures – **sets**.

A set is an unordered collection with no duplicate entries. Python sets work the same way as they do in **set theory**.

If you don't know about set theory, don't worry. We're not going to go into the mathematics of set theory here.

Set theory is an interesting, and important, branch of mathematics. But we're going to use Python sets to solve some common programming problems.

I won't be getting all mathematical in these videos.

Introduction to sets

If you **are** familiar with **set theory**, then you'll find that Python sets behave exactly how you'd expect. You can perform all the usual set operations on them.

We'll be looking at those operations in a non-mathematical context, but you'll have no problem applying them to mathematical uses of sets, if that's something you need to do.

I'll be using slides to explain things like set **union** and **intersection**. Fast forward those parts of the videos, if you're already familiar with those operations.

Introduction to sets

The keys of a dictionary are very similar to a set. The main difference, since Python 3.7, is that dictionary keys are ordered.

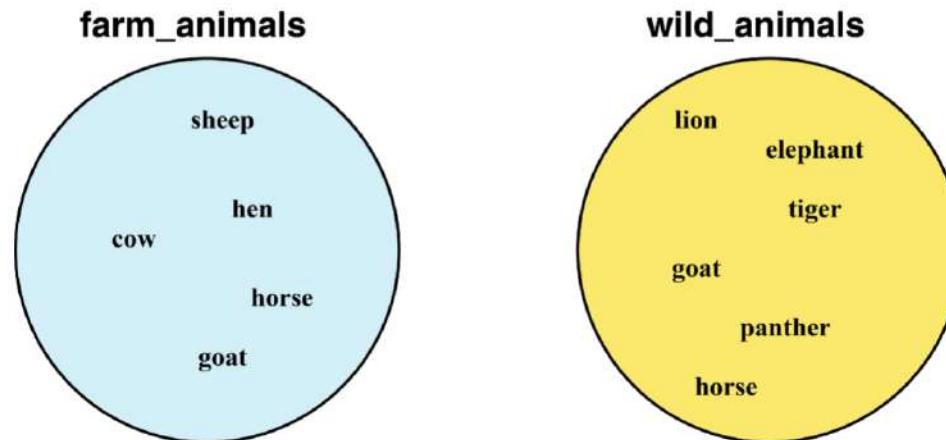
Sets have no ordering.

That's an important point, and explains why you can't do certain things with sets. For example, there's no way to access individual elements of a set.

We'll see that sets are unordered, when we iterate over some sets in the next video.

Before that, I'll introduce the basic set operations.

set examples

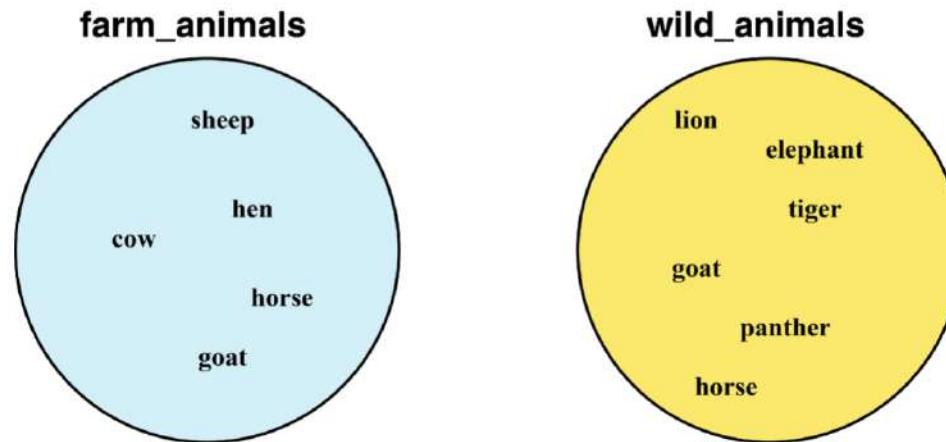


Here, we've got two sets. They represent a collection of farm animals, on the left, and wild animals, on the right.

The sets are over-simplified, to keep them easier to understand. You'd struggle to find wild cows these days, but there are wild sheep in many parts of the world.

We're keeping things simple: **sheep** is only in the `farm_animals` set.

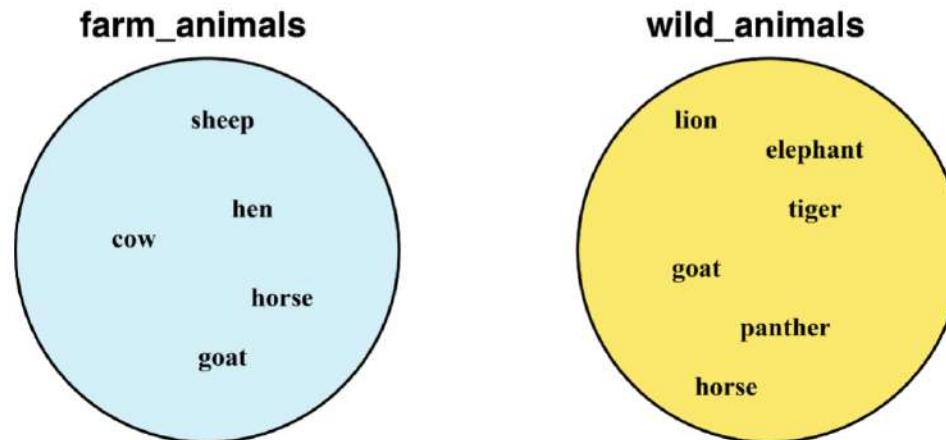
set examples



When we represent sets with circles, like these, it's easier to see that they are **unordered**.

A set is just a collection of items. Our animals could wander about inside their circles, but the set would still contain the same items.

set membership

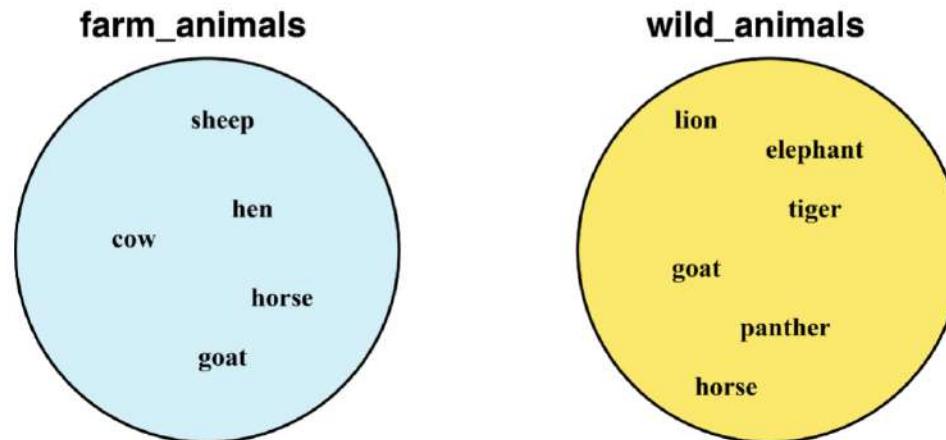


The most common operation we can perform on a set, is to test for **membership**.

Looking at the `farm_animals` set, we can see that **cow** is a member of `farm_animals`.

hen, sheep, goat, and horse are also members of the `farm_animals` set.

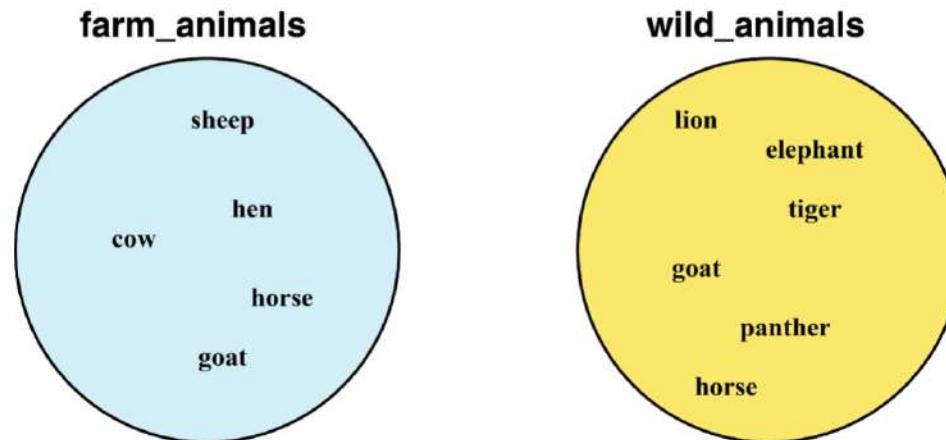
set membership



In Python, we test for membership using the `in` keyword. We've seen that with lists, tuples and dictionaries, and it works the same with sets.

tiger is a member of the `wild_animals` set, so the condition `'tiger' in wild_animals` will be True.

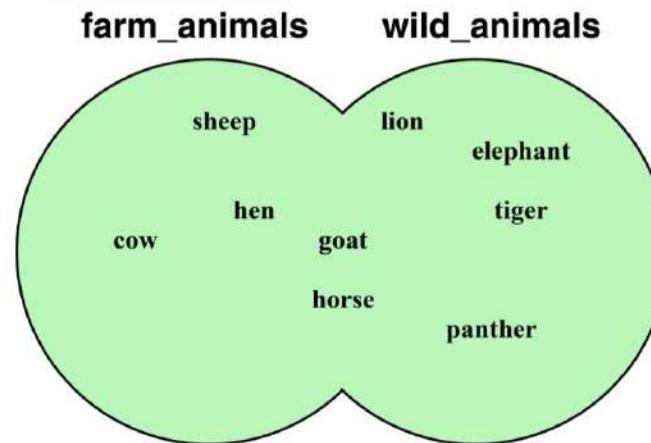
set membership



Testing if something is in a set can make our code more efficient. It's also a good way to fix a bug that we had, in our very early menu programs.

We'll have a look at that, in the next video.

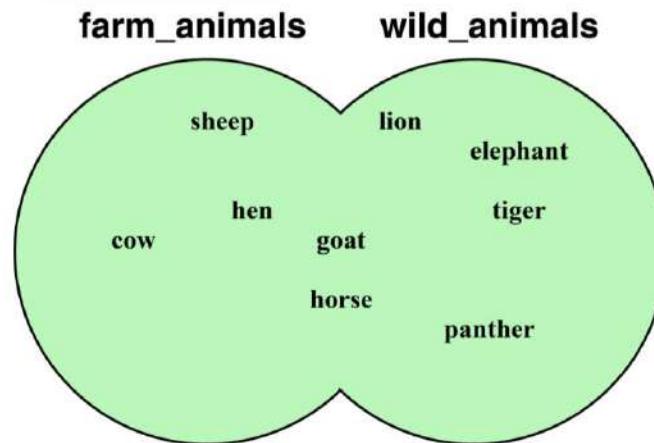
set union



The **union** of two or more sets, is the set of all items that exist in all the sets.

Here, we have the **union** of the `farm_animals` and `wild_animals` sets.

set union



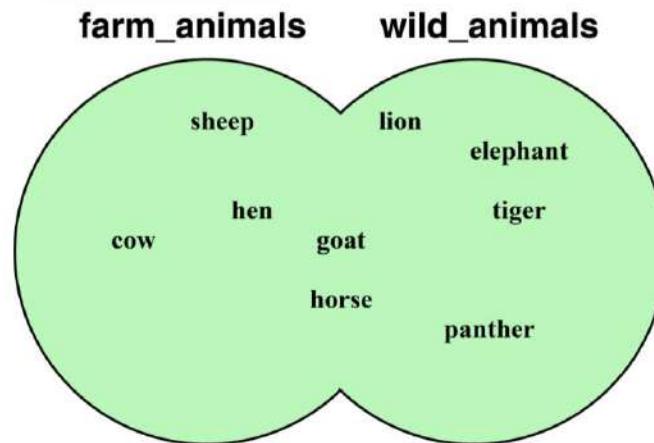
In Python, the union of these two sets is written:

```
farm_animals.union(wild_animals)
```

or

```
farm_animals | wild_animals
```

set union

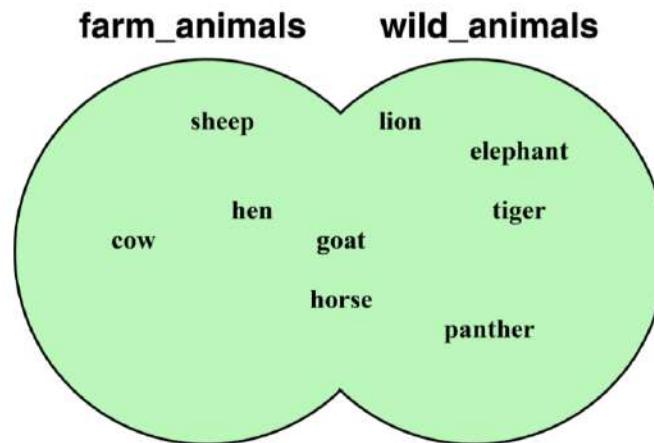


One important point to notice, looking at our union set, is that items only appear once.

The elements of a set are **unique**. That's part of the definition of a set.

goat and **horse** appear in both sets, but they only appear once in the **union** set.

set union

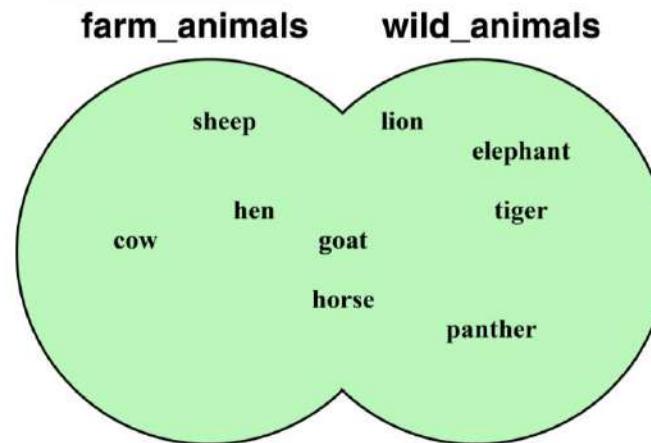


It doesn't make sense for something to be in a set more than once.

That property of sets – that the items are unique (or distinct, in mathematical terms) – can be useful in our code.

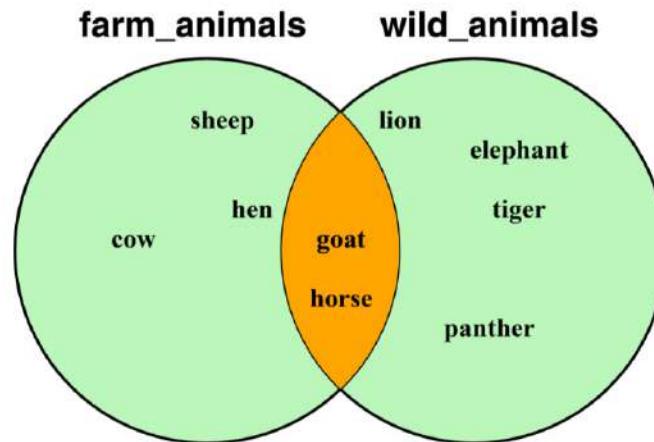
Converting a list to a set, for example, will automatically remove any duplicate values.

set union



Once again, we'll see some examples of this, later. At the moment, we're just learning the basic terminology that applies to sets.

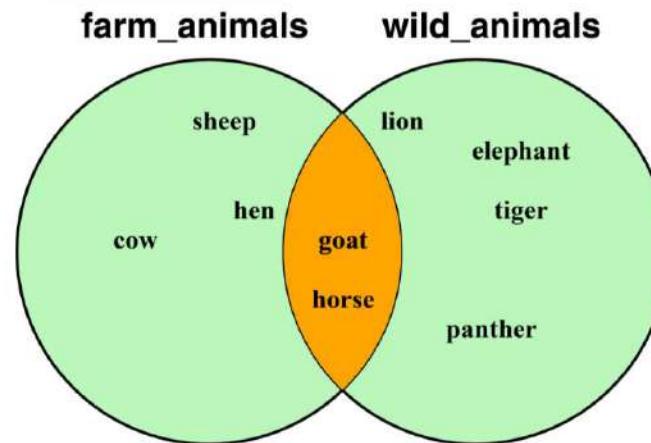
set intersection



The **intersection** of two or more sets, is the set of items that appear in **all** sets.

That's quite easy to remember – it's the items that appear in the area where our circles **intersect**. We can see that **horse** and **goat** are in both sets, and appear in the orange area.

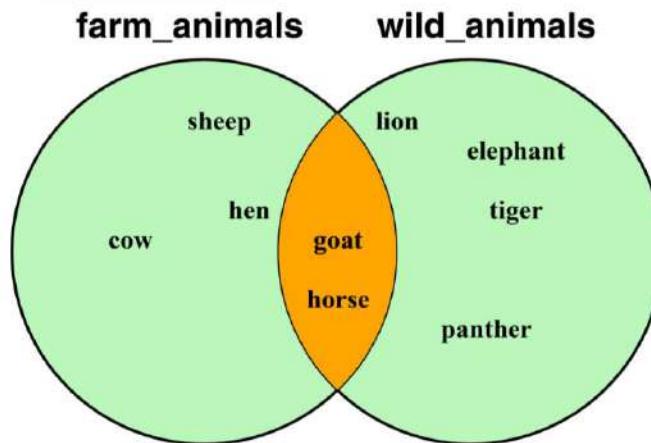
set intersection



If we had a third set, of animals that can be ridden, for example, then **horse** would probably be the only animal in the intersection of all 3 sets.

It may be possible to ride a goat, but it's certainly not very common 😊.

set intersection



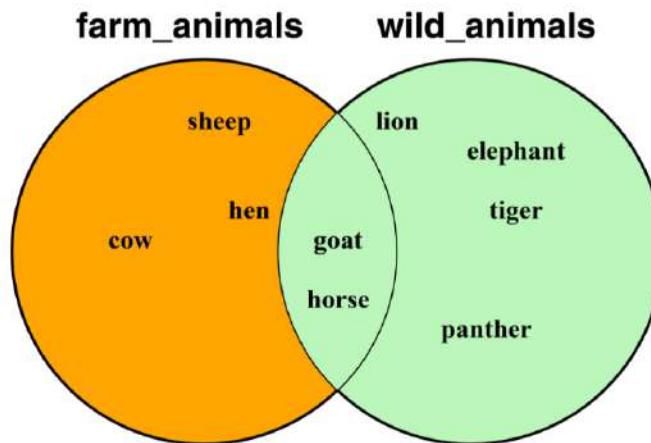
The Python code representing this intersection would be:

`farm_animals.intersection(wild_animals)`

or

`farm_animals & wild_animals`

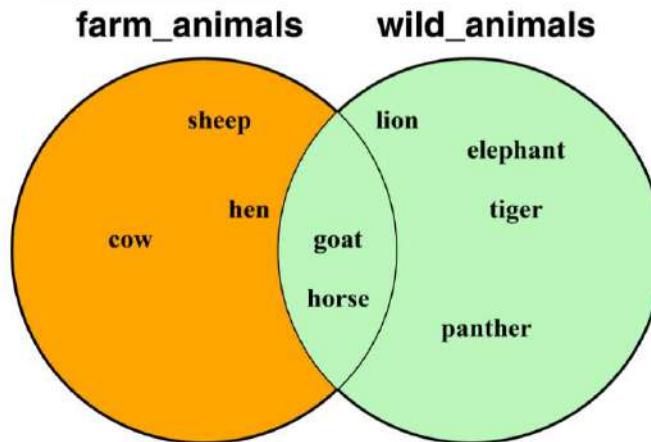
set difference



You can also subtract one set from another. Here, we've subtracted `wild_animals` from `farm_animals`.

That leaves **cow**, **sheep** and **hen**. Any items from `wild_animals` are removed, when we subtract the two sets.

set difference



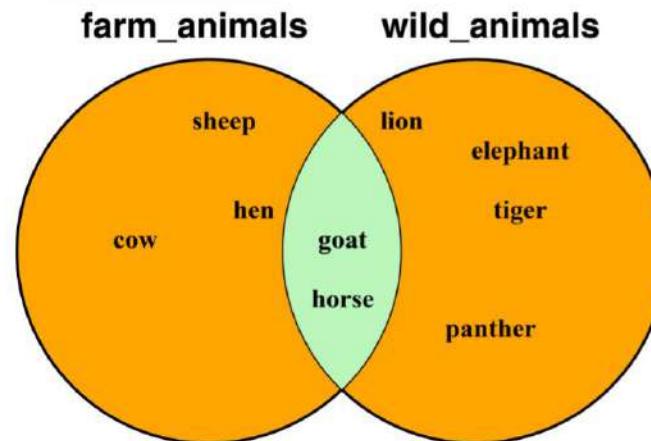
In Python, we'd write that as either

`farm_animals - wild_animals`

or

`farm_animals.difference(wild_animals)`

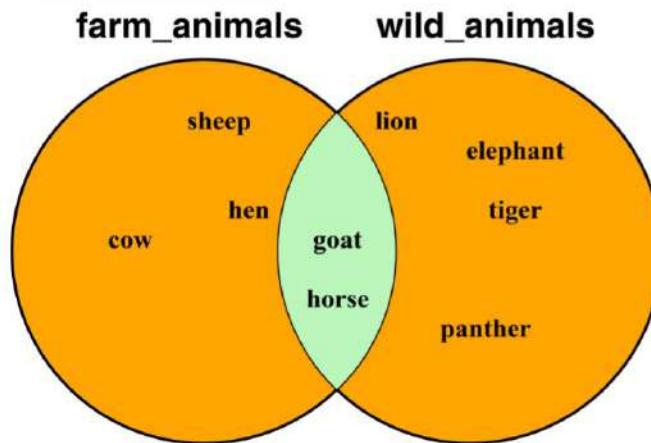
symmetric difference



Set theory also defines a **symmetric difference**.

The **symmetric difference** is the set of items that are in one set or the other, but **not both**.

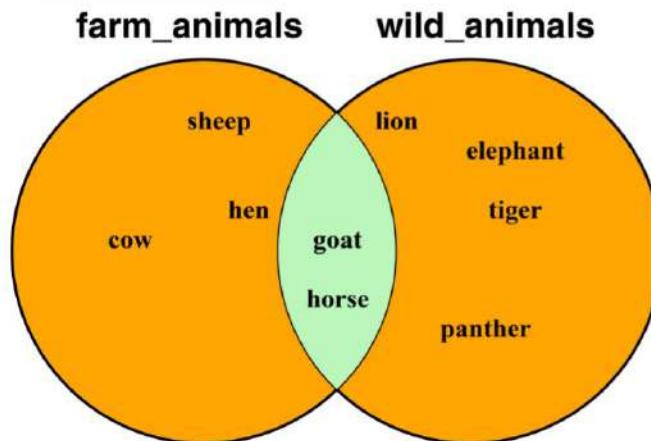
symmetric difference



In this example, the symmetric difference **doesn't** include goat and horse. All the animals in the orange areas **are** included in the symmetric difference.

It's the opposite of the **intersection** of 2 or more sets.

symmetric difference



In Python, we'd write that as:

```
farm_animals.symmetric_difference(wild_animals)
```

or

```
farm_animals ^ wild_animals
```

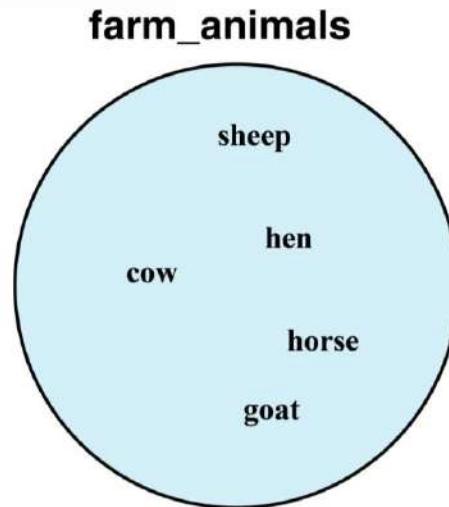
subsets and supersets

One set can also be a subset of another set.

We use the normal comparison operators; `<`, `<=`, `>` and `>=`, to check for subsets.

But that's enough slides, and theory, for now. We'll leave subsets and supersets until we've had some practice with using sets.

sets are unordered

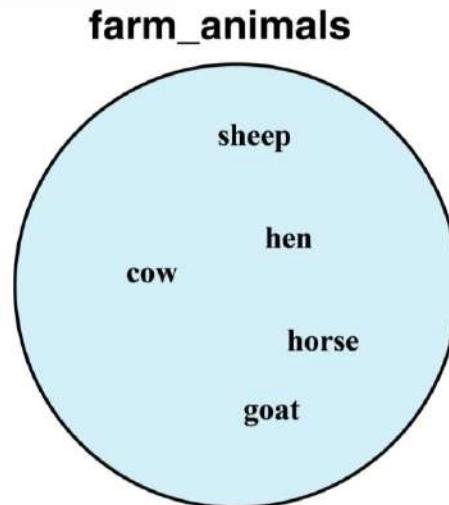


It would be more accurate to represent the set of farm_animals like this. That's just a collection — or group — of things.

That's more obviously **unordered**.

If it was ordered, what order would you use?

sets are unordered



Would you go from left to right, and get cow, sheep, goat, hen, horse?

Or would you go from top to bottom, and get sheep, hen, cow, horse, goat?

Sets use hashes

Because sets use hashes to store the items, anything that you want to put into a set **must** be **hashable**.

That's exactly the same requirement that we had for dictionary keys.

If you try to put a list into a set, for example, you'll get an error — the same "unhashable type" error that we talked about, in **the dict documentation** video, earlier in this section.

Although that's obvious, you may not have thought about it.

Warning – pep talk!

That code is very basic, but you probably struggled a bit with the challenge, at the time.

Now you can understand it easily.

You've come a long way since we wrote that code, a few sections ago.

When you find yourself struggling and are getting frustrated, or thinking that you'll never get the hang of this, take a break.

Have a look at some of the earlier examples that we wrote.

Reviewing the earlier code, which you probably also struggled with at the time, will show that you **are** learning a lot.

Warning – pep talk!

It's easy to lose sight of that, when you're stuck.

Focus on how far you've come, and you'll get the confidence to continue.

Alright, pep talk over 😊.

set membership and hash codes

In this video, we'll see why using `in` with a set is faster than when you use a list.

The reason is because sets use hash codes – just like keys in a dictionary.

We saw how dictionary keys are stored using their hash codes, when we created our simple dictionary implementation.

Review the **Hash tables** video, if you want to remind yourself about how we used hash codes to store keys.

set membership and hash codes



When we check if something is in a list, Python has to check each item in the list, until it finds the one we want.

If we're checking for the string '1', Python starts by comparing the 1st item in the list.

'4' doesn't equal '1' so it moves on to check the next value.

'5' also doesn't equal '1', and Python has to continue searching.

It has to perform 4 comparisons, before it knows that '1' is in the list.

set membership and hash codes



Of course, if we're checking for the string '**4**', it will find it on the first go.

But if we're testing for '**7**', the entire list would have to be checked, before Python could tell us that it **isn't** in the list.

That's called a **linear search**. You start at the beginning, and check each item until you either find the one you want, or reach the end of the list.

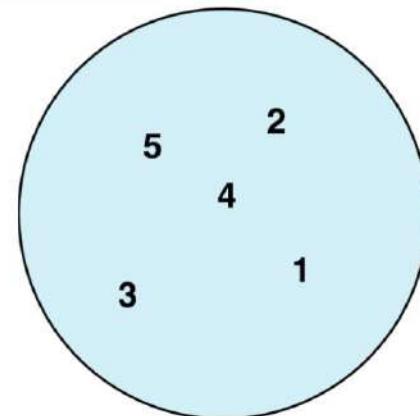
set membership and hash codes



Note that this is the case even with a sorted list.

Python has no way to tell if a list is sorted, and has to perform a linear search, when checking if something is in a list.

set membership and hash codes

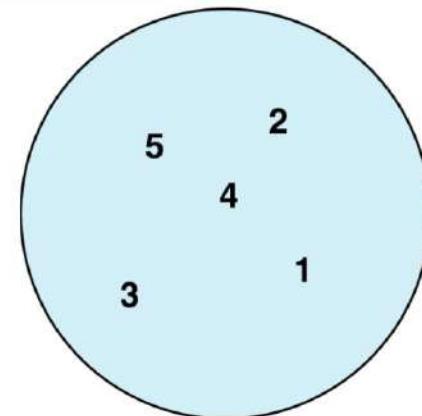


When checking for membership of a set, Python uses the hash code to find out where the item should be.

If it's there, then the item is `in` the set, otherwise it's not.

Hash codes are used in the same way as we saw for dictionary keys.

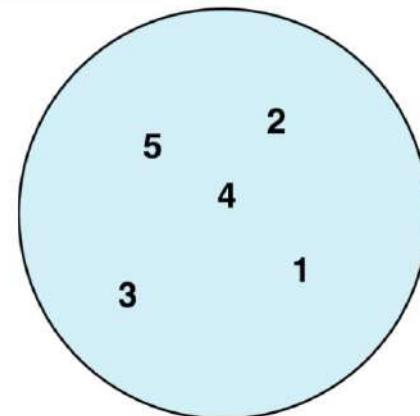
set membership and hash codes



A hash code lets us go directly to the item in the hash table.

There's a small overhead while the hash code is calculated, but once that's done, access is very fast.

set membership and hash codes



You can check if a value is in a set of one billion items, just as quickly as in a set of five items.

The size of the set has no effect on the time taken, to check if something's in it.

set membership and hash codes

So that's the advantage of using a set, rather than a list, when testing for membership.

If you're working with large data, checking for membership will be a lot faster with a set, compared to a list.

COMPLETE PYTHON MASTERCLASS
Testing set membership is fast



DICTIONARIES AND SETS
PM-41-8

Keep things in perspective

Does that mean you should replace sets with lists, whenever you're testing if something is in the list?

I've just explained why checking an item in a set is a lot faster than checking it in a list. So you might be tempted to convert all your lists to sets.

But that's **not** what I'm suggesting.

Things are rarely that simple, and there aren't rules that you can apply everywhere. Programming isn't like that.

Understand how things work, rather than trying to remember a set of rules.

Keep things in perspective

If we could produce a set of rules to tell you how to write code, then we could get computers to write the code for us.

Computers are very good at following rules, after all.

COMPLETE PYTHON MASTERCLASS
Testing set membership is fast



DICTIONARIES AND SETS
PM-41-10

Keep things in perspective

Treat each case individually.

Understand what's happening in the code, and understand the implications of any decisions you make.

That's not as hard as it sounds, once you've got used to the various objects that are available.

But students are often still unsure of which object they should use, so I'm going to digress slightly and give you some pointers.

I'll use the **summarychallenge** code we've just been looking at, and discuss a couple of things that might cause you to worry whether you're using the right object.

Examining options

```
choice = "-" # initialise choice to something invalid
while choice != "0":
    if choice in "12345":
        print("You chose {}".format(choice))
```

Our original code checked for the choice being in a string. That's the string "12345".

That had a bug, as we saw – so a string isn't really suitable here. We need to use a list, a tuple or a set.

Examining options

```
choice = "-" # initialise choice to something invalid
while choice != "0":
    if choice in list("12345"):
        print("You chose {}".format(choice))
```

To fix the bug, we used a `list`. A `tuple` would also have worked.

That fixed the bug, and is an acceptable solution to the problem.

Even though checking for membership of a `list` or a `tuple` is quite slow, we need to keep things in perspective.

"quite slow" is relative

```
choice = "-" # initialise choice to something invalid
while choice != "0":
    if choice in set("12345"):
        print("You chose {}".format(choice))
```

We've only got 5 items in the list.

A set would be faster, but the difference is a few microseconds. That's millionths of a second.

Of course, we had to test that statement before recording this video. On an Intel Core i7 2.2 GHz Quad-Core CPU, we got the following timings.

"quite slow" is relative

We used a list and a set with one million items, and tested checking for an item that **was** in the data, and an item that **wasn't**.

	item present	item not present
list	0.000003749	0.011801786
set	0.000001768	0.000001679

The performance using a set was better than using a list, as we'd expect.

Not surprisingly, using a list was a lot slower when the value **wasn't** present. In that case, the entire list has to be scanned before Python can know that the value doesn't exist.

Even then, in the worst case, the test for membership of a list only took one hundredth of a second.

optimise when it's important

Our **summarychallenge** program spends most of its time waiting for the user to type their choice.

It makes very little sense to worry about optimising the code to save a few hundredths of a second, or less.

Would you notice a delay of a few milliseconds when typing?

But if we were testing membership inside a tight loop, where performance was important, there is a performance improvement we can make.

optimise when it's important

```
choice = "-" # initialise choice to something invalid
while choice != "0":
    if choice in set("12345"):
        print("You chose {}".format(choice))
```

Using a set, rather than a list, is a good start – as we've seen.

But how we create that set is also important.

In this example, we call the set function. We pass a sequence containing the characters we need, and Python will create a set for us.

That means the set has to be created, each time round the loop.

That's quite expensive in terms of CPU time, and can be improved.

optimise when it's important

```
choice = "-" # initialise choice to something invalid
while choice != "0":
    if choice in {"1", "2", "3", "4", "5"}:
        print("You chose {}".format(choice))
```

Using a set literal, instead of the set function, is much faster.

After this simple change, the code performed twice as fast as using the set function.

That's definitely worth knowing, when you find that performance is an issue, and you want to speed things up.

The disadvantage is that it takes a bit longer to type 😊. That's not a huge problem, but does explain why you'll sometimes see a string being passed to the set function, in cases like this.

optimise when it's important

```
choice = "-" # initialise choice to something invalid
valid_choices = {"1", "2", "3", "4", "5"}
while choice != "0":
    if choice in valid_choices:
        print("You chose {}".format(choice))
```

You might also decide to create the set once, outside the loop.

Inside the loop, the `valid_choices` variable is used in the condition.

We found very little performance difference between this code, and the code in the last slide.

optimise when it's important

Once you start thinking in terms of what Python has to do, when executing your code, it gets easier to write more efficient code.

But as I've said, keep it in perspective. Optimising code that's mainly spent waiting for user input, isn't usually worth the extra typing, or the risk of making your code less readable.

If your loop is doing a lot of computation, and has to run quickly, then it **is** worth optimising it.

What is a tight loop

A **tight loop** is one that iterates many times, and has a significant impact on the total execution time.

You'll find other definitions on line, but that's the one we use.

All definitions will talk about the loop iterating many times – hundreds, thousands or maybe millions of times.

Those are the kinds of loops that can benefit from more efficient ways of writing the code.

Testing set membership is fast

That's the end of this little digression. We've looked at why you might want to use a set, when you need to test for membership. And we've seen how a set is faster than a list or tuple.

In the remaining videos in this section, we'll look at the other things we can do with sets.

See you in the next video.

discard vs remove

As the Zen of Python says, "There should be one – and preferably only one – obvious way to do it".

So why do we have both `discard` and `remove`?

We've just seen the difference in their behaviour, and it **is** an important difference.

discard vs remove

There are two different things we might want to do, and having two different methods reflects that:

1. Ensure that, when we're done, something no longer exists. In this case, we don't care if it was already there or not – we just want it gone.
2. Remove something if it exists, but provide some sort of notification if it doesn't.

These are different use cases, and Python provides an obvious way to do each one; `discard` for the first case, and `remove` for the second.

Ensure that something no longer exists

Airports restrict the items that you can carry in hand luggage.

If you're very well organised, you might run your packing list through a Python program, before packing. You might have the following set of things that you can't take:

```
restricted_items = {"gun", "catapult", "blade", "scissors"}
```

That's not the full list, but those will do for our example.

So your Python program will iterate over that set, and remove anything it finds from your set of items to pack.

Ensure that something no longer exists

The code might look something like this:

```
for item in restricted_items:  
    packing_set.discard(item)
```

We loop through the items in the set of restricted items, and discard any restricted items from our packing set.

Using `discard` there is fine. If you haven't included the item in your packing set, `discard` won't raise an exception. The code just moves on to the next item.

Ensure that something no longer exists

The code might look something like this:

```
for item in restricted_items:  
    packing_set.discard(item)
```

Using `remove` wouldn't be a good idea. When it failed to find "catapult" in the list of things you intend to pack, the code would crash.

This is an example of when you don't care if something is in the set or not – you just want to make sure it's not there, after the code has executed.

Remove something if it exists

Lets assume we've created a list of sets. The sets contain the medications that patients are taking, and our intention is to replace the blood thinning drug **warfarin** with **edoxaban**.

Both drugs thin the blood to prevent blood clots, but **edoxaban** has some advantages over **warfarin**.

I'm going to assume that we've already identified those patients who would benefit from changing their medication.

And yes, this is a bit contrived 😊. But it also does represent a real situation that could arise.

So our code has to remove **warfarin** from the set of medications that each patient is taking.

Remove something if it exists

The aim here, is to replace one anti-coagulant drug with another. That means we can't just use **discard**.

When we come to remove **warfarin**, it's important that the patient was actually taking it.

Remove something if it exists

In this case, it's better that our code crashes. Otherwise, we could end up prescribing the anti-coagulant drug, **edoxaban**, to someone who didn't need an anti-coagulant.

`remove` is a better option here.

If we attempt to remove **warfarin**, and the patient wasn't taking it, that's a situation we need to be alerted to.

When we use `remove`, we'll get an error, if **warfarin** isn't in the set of medications that the patient was taking.

Remove something if it exists

Now I'm not suggesting that you write code that you know will crash. That's not the point of all this.

The point is that `remove` will let us know, if we try to delete something that doesn't exist. `discard` won't.

Remove something if it exists

Later, we'll see how to handle the exception that `remove` causes, when you try to remove something that doesn't exist.

At the moment, we're looking at why there are two different methods to delete an item from a set.

So that's the reason why.

In the next two videos, we'll look at the code for both these methods. Examples will make the difference clear, and give an idea of when you should use each method.

set.pop

We've seen pop used with a dictionary, and it can also be used with a list.

When popping items from a set, there's a slight difference. Sets aren't indexable, so the set pop method doesn't take any arguments.

It pops an arbitrary item from the set, and returns that item.

set.pop

What do I mean by "arbitrary" item?

I mean that you can't reliably predict which item will be popped from the set.

You might get consistent behaviour, when popping integer values from a set, with some versions of Python.

But if you run the same code on another Python version, you might get different values.

That's a bit useless, isn't it?

You may think that getting "random" items from your set isn't much use.

How can you write reliable code, if you don't know which value you'll be dealing with?

In fact, you'll find that you often don't care.

set.pop

Imagine you're processing a set of tasks, and it doesn't matter what order they're performed.

Popping them from a set would be fine, in that case.

The `pip` module works like that. You can see a reference to it, in Martijn Pieter's answer to this stack overflow question.

I'll switch to it in my browser. Read through it at your leisure. The link will be in the resources for this video.

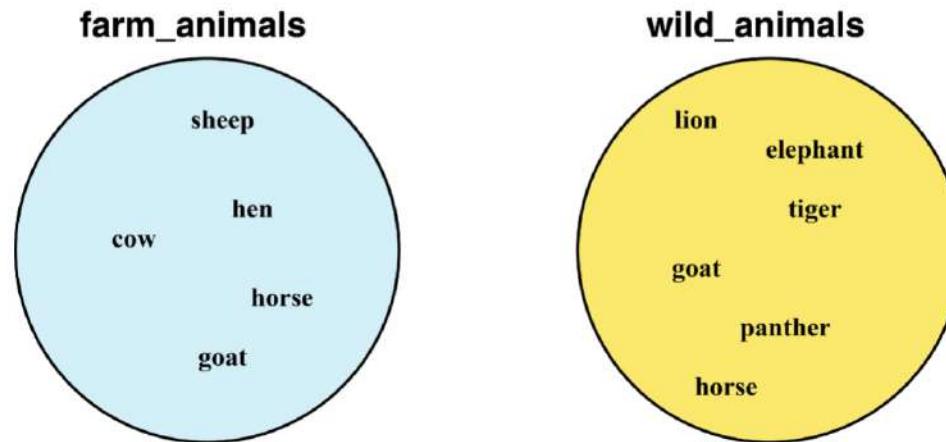
set.pop

We're going to do something similar, in our example.

We'll process each of the patients in our trial – from the previous video.

We don't care what order we process them; as long as we process them all, and each one only gets processed once.

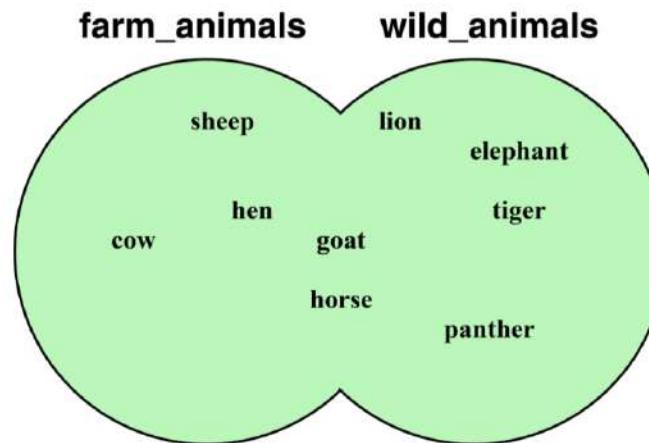
set union



Back in the **Introduction to Python sets** video, we had two sets; a set of farm animals, and a set of wild animals.

We'll use these sets to see how we can create the union of two sets.

set union



The union of two (or more), sets is the set of elements from all the sets.

As the diagram shows, the union of `farm_animals` and `wild_animals` contains all the animals.

Let's see how to create a union, in Python.

set union is commutative

The union of two (or more) sets is commutative. That's a posh way of saying that it doesn't matter which way round you do it.

It's the same as addition and multiplication, in arithmetic.

a + b is the same as **b + a**.

2 * 4 is the same as **4 * 2**.

Application of set union

We've seen the theory of set union, and we've seen how to create a union, in Python.

In this video, we'll have a look at a practical application for all this.

Open up **prescription_data.py**, and have a look at the `adverse_interactions` list, on **lines 17 to 25**.

We've got a list. Each item in the list is a set, containing drugs that shouldn't be taken together.

Application of set union

```
adverse_interactions = [  
    {metformin, amlodipine},  
    {simvastatin, erythromycin},  
    {citalopram, buspirone},  
    {warfarin, citalopram},  
    {warfarin, edoxaban},  
    {warfarin, erythromycin},  
    {warfarin, amlodipine},  
]
```

Now, imagine you're a dispensing chemist, and you've got a queue of people at the counter.

You're about to hand over some Carbimazole.

How quickly can you check that list, to see if Carbimazole has an adverse reaction with anything?

Pause the video, and see how long it takes.

Application of set union

That wasn't very easy, and there are only eight drugs in our list. Imagine if it contained **all** drugs that have reactions with other drugs.

To help our dispensing chemists, we'll create an alphabetical list of these drugs.

An alphabetical list is much quicker for a human to work with.

Another example

Another example might be the trial we looked at, a few videos ago.

When evaluating the performance of a new drug, you might have 3 sets of people. In fact, that's the case with lots of research, not just clinical trials.

One group would receive the new drug – or whatever it was that was being researched.

A second group would be given a placebo – a pill that does nothing.

And in some cases, you'd have a third group who didn't even know they were part of the research.

Another example

Often, you get their data by using historical data, from the general population. In that case, the names would be anonymised.

That gives three sets, and you could use a set union to produce a set containing all the data in the trial.

In the next video, we'll improve this code slightly. We'll look at another set operation, **union update**.

exercises

In the next video, I'll discuss the advantage that the set operation methods have, over the operators.

Before you start watching the next video, create some sets of your own, and practise with the union and update operations.

exercises

For example, you might create these sets:

```
scorpions = {"emperor", "red claw", "arizona", "forest", "fat tail"}  
snakes = {"python", "cobra", "viper", "anaconda", "mamba"}  
spiders = {"tarantula", "black widow", "wolf spider", "crab spider"}  
vespas = {"yellowjacket", "hornet", "paper wasp"}
```

and create a set of things that bite, or things that sting. Spiders bite, scorpions and vespas sting.

Spiders and scorpions are arachnids, so you could get a set of arachnids by forming the union of those two sets.

Experiment, and I'll see you in the next video.

COMPLETE PYTHON MASTERCLASS
Union update

 LearnProgramming
Academy

DICTIONARIES AND SETS
PM-50-2

Patient trial

It's quite possible that there will be more than one trial being conducted, at the same time.

Giving different experimental drugs, to the same person, probably isn't a good idea.

In terms of the trials, it would be difficult to know which drug was responsible for any observed effects.

It's also not good for the patient's health.

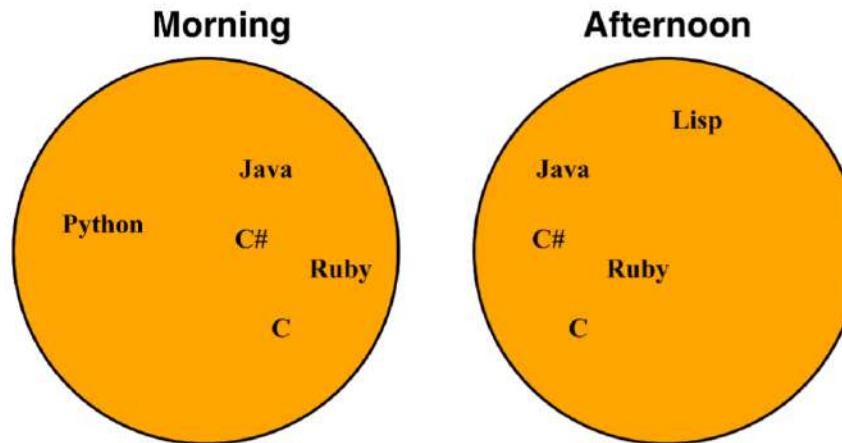
So we want to identify any patients who might have been included in more than one trial.

Patient trial

Set intersection is ideal for that – it can tell us who's in more than one set.

The intersection of 2 sets is the set of elements that are present in both sets. That's what we want here.

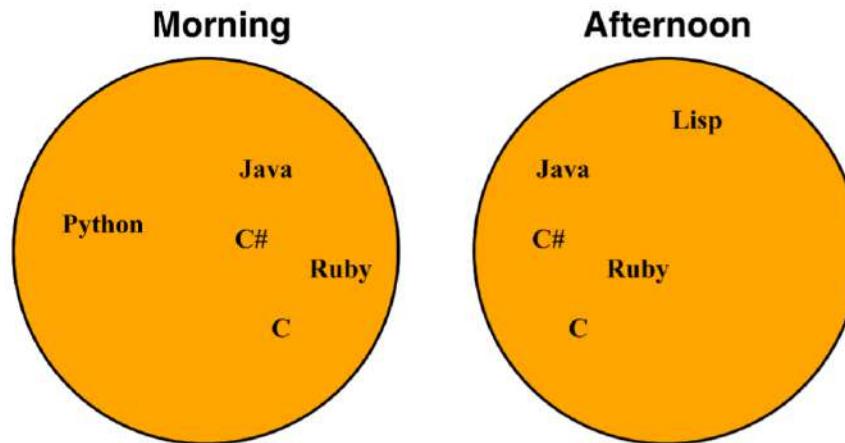
symmetric difference



For this example, we've got two sets. They represent programming courses at a training establishment.

We'd like to learn more than one language, and we're determined not to miss any lessons.

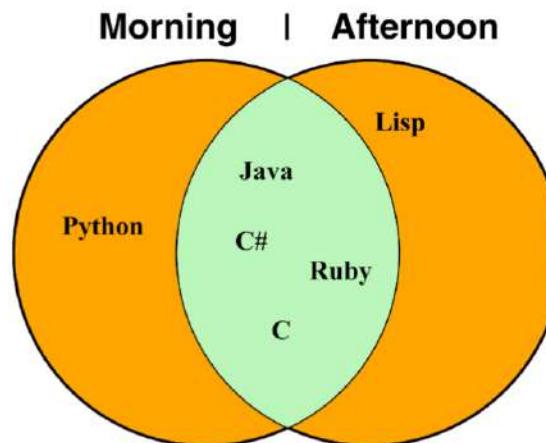
symmetric difference



That means we'll have to take one course in the mornings, and another in the afternoons.

Which means we can't take any course that has lessons in both the morning and the afternoon.

set intersection

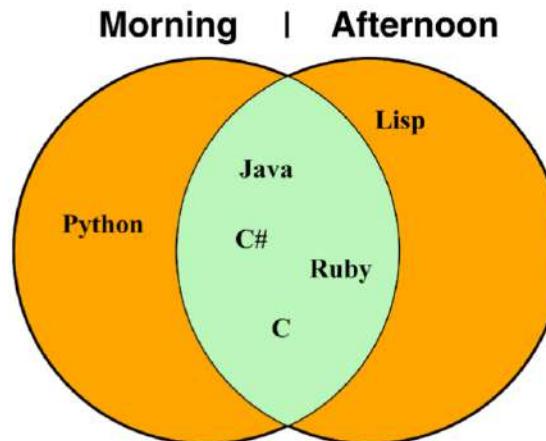


We've seen the set intersection before.

We can use that to see which courses have lessons in the morning **and** afternoon.

The green area represents the intersection of the two sets, and shows us which courses we **can't** take.

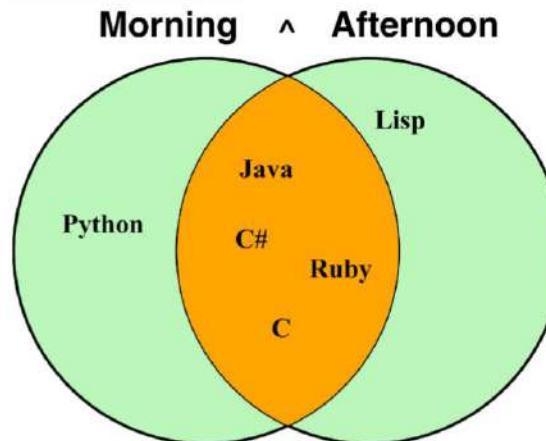
set intersection



There are only 6 courses in this example, so it's not hard to see which courses we **can** take. Bear with me on this, and imagine sets that could contain hundreds (or thousands) of items.

To get the set of courses we can take, we want the opposite of the intersection.

symmetric difference



The symmetric difference is the opposite of intersection.

It produces the set of items that are in one set or the other, but **not in** both.

Java, C, C# and Ruby are in both sets, so they're excluded from the symmetric difference.

That leaves Python and Lisp, which is exactly what we wanted.

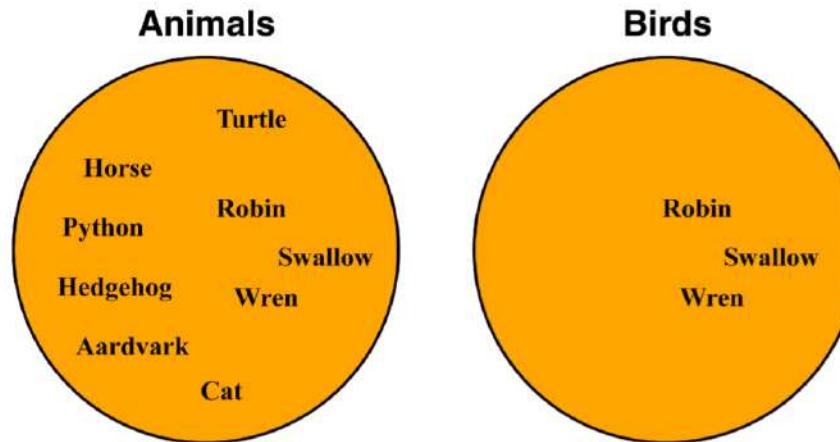
subsets and supersets

In the **Introduction to sets** video, we looked at what sets are, and discussed things like set membership, and the various operations that we can perform on sets.

I finished that video by mentioning subsets and supersets. I didn't go into detail at the time, because we'd had enough theory at that point.

In the next few videos, we'll see what subsets and supersets are, and how we can test for them in Python.

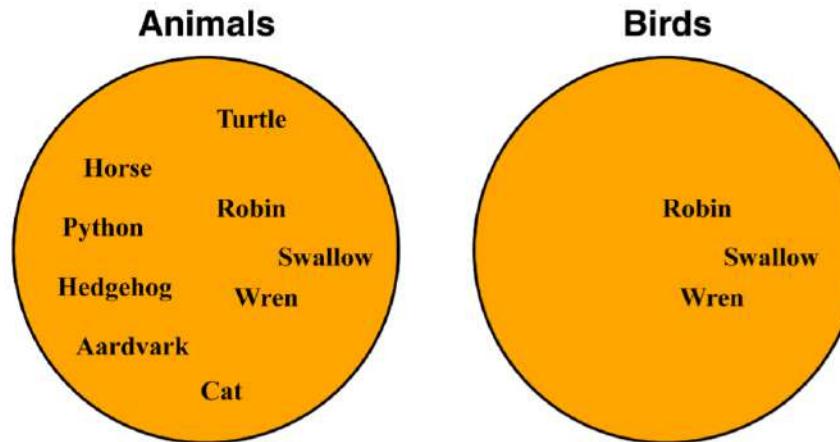
subsets and supersets



This diagram represents two sets: a set of animals, and a set of birds.

Birds are also animals, and all of our birds are also members of the **Animals** set.

subsets and supersets

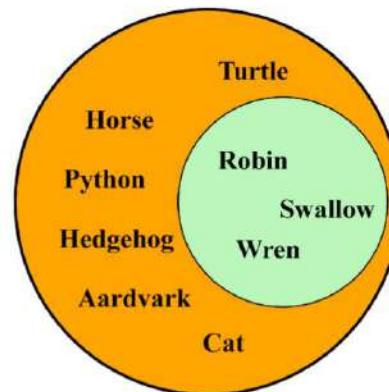


When **all** the items of one set are also in another set, the contained set is called a **subset** of the containing set.

Here, **Birds** is a subset of **Animals** – as we can see on the next slide.

subset

Birds is a subset of Animals



In set theory, that would be written as:

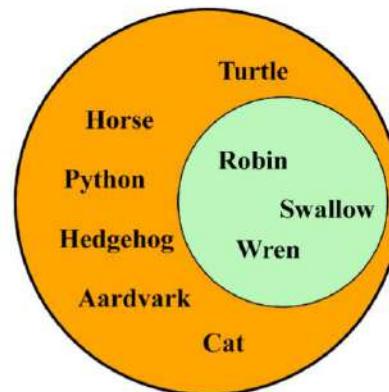
$\text{Birds} \subseteq \text{Animals}$

In Python, we use the less than or equal to symbol:

`Birds <= Animals`

proper subset

Birds is a subset of Animals

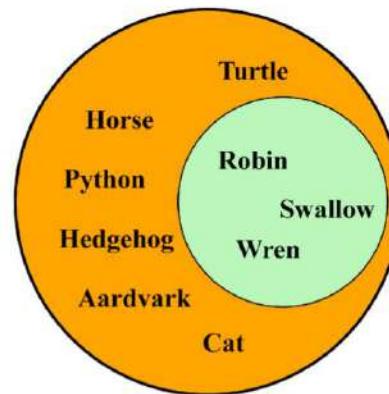


Birds is also a **proper** subset of Animals.

A **proper** subset is a subset that also isn't equal to the set that contains it. The distinction is important, in set theory.

proper subset

Birds is a subset of Animals



In set theory, a **proper subset** would be written as:

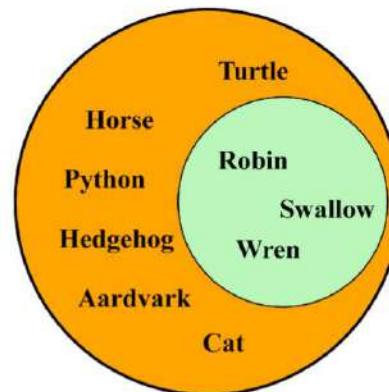
$\text{Birds} \subset \text{Animals}$

In Python, we use the less than symbol:

`Birds < Animals`

superset

Animals is a **superset** of Birds



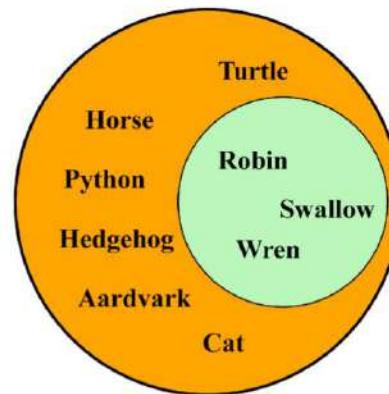
Birds is a subset of Animals, and we also say that Animals is a **superset** of Birds.

That means all the items of Birds are contained in Animals.

The two terms are complementary. If set A is a **subset** of B, then set B is a **superset** of A.

superset

Animals is a superset of Birds



In set theory, that's written:

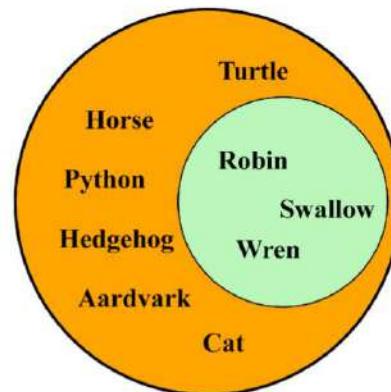
$\text{Animals} \supseteq \text{Birds}$

In Python, we use the greater than or equal to symbol:

`Animals >= Birds`

proper superset

Animals is a superset of Birds

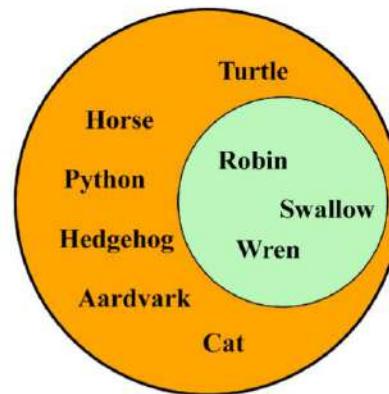


Animals is also a **proper superset** of Birds.

A proper superset is a set that contains another set, but isn't equal to it.

proper superset

Animals is a superset of Birds



In set theory, that's written:

$\text{Animals} \supset \text{Birds}$

In Python, we use the greater than symbol:

`Animals > Birds`

subsets and supersets

With small sets, when we can draw diagrams, it's fairly easy to work out if one set is a superset or subset of another.

When we're working with large sets, or have many sets to check, Python can make this much easier.

Make sure you understand what a subset and a superset is, and that you understand what **proper** subsets and supersets are.

A **proper** subset must contain fewer elements than its superset, and a **proper** superset must contain more elements than the subset.

In the next video, we'll write some Python code to check if one set is a subset of another set, then we'll look at a practical application.

Big O

You may have come across the term **Big O**, when looking at different algorithms. In this section, I'll explain what it is and why it's useful.

I'll also be explaining what it's **not**. In particular, it doesn't guarantee that one algorithm will execute faster than another.

It's basically a way to talk about the **complexity** of an algorithm, or the complexity of operations on a data structure.

Big O

What I certainly **won't** be doing, is providing a rigorous, mathematical discussion of **Big O**.

There's a good reason why I won't be doing that — the maths gets horrible.

When we looked at the Wikipedia entry for **Hash functions**, back in the **Dictionaries and sets** section, there was some pretty scary maths in there.

I encouraged you to skim through the article, because there were some interesting points that you could get from it, without worrying about the maths.

Big O notation

Big O notation is used to describe how the running time (or space requirements) of an algorithm grows, as you give it larger data sets to process.

Programmers commonly use **big O notation**, rather than the other notations that are also available. Big O describes the upper bound.

Some other notations are **big omega notation**, which uses the Greek letter Ω , and **big theta notation**, which uses the Greek letter Θ . They're used to describe a lower bound, and both a lower and upper bound, respectively.

But this is starting to sound a bit mathematical again 😊. So we'll stick with big O.

In the tables that follow, I've listed the most common **big O** notations that you'll come across, with a brief description of what they mean.

Big O notation

The notations appear in the order of how well they scale. As we get further into the slides, the algorithms will take increasingly larger amounts of time, as `n` increases.

If we're applying them to memory usage, then the algorithms will use more and more memory, as `n` increases.

The letter `n` refers to the number of items being processed. The size of the data, in other words.

Big O notation in practice

It's important to understand that **big O** doesn't tell you how fast an algorithm is. It's an indication of how the algorithm **scales**.

By that, we mean how it performs when processing larger and larger data sets.

Accessing an item in a list, using its index position, is **O(1)** or constant time. No matter how many items are in the list, we can go directly to any item by using its index position.

Accessing the 900th item takes the same time, whether the list contains a thousand, a million, or several billion items.

Big O notation in practice

On the other hand, finding a value in an unsorted list is **O(n)** or linear time.

If we have a list of 10 items, and we want to check if it contains "Python", then we might have to test 10 different values.

If the list contains 1000 items, then we might have to check all one thousand entries.

If the list is sorted, we can perform a binary search. That will execute in **O(log n)**. We discussed the binary search in the **Program flow control in Python** section.

Big O notation in practice

Of course, the item we want might be the first item in the list. That means we'll find it straight away.

Similarly, with a binary search, the item might be at the midpoint of the sorted list. If so, we'll also find it straight away.

Which leads nicely onto the next slide: What does big O measure?

What does big O measure?

Big O is used to measure the **worst case** of an algorithm.

In the worst case, **when the item we want is at the end of the list**, then searching an unsorted list will involve checking every item in the list.

With a list of 10 items, we'll find our item on the 10th try. With 1000 items, we'll compare 1000 values before we find the one we want.

O(n) doesn't tell us that we'll need **n** operations. It tells us that, **in the worst case**, the time taken to find something will increase as the size of the list increases.

Similarly, **O(log n)** implies that the algorithm's time increases as the log of **n** increases.

What does big O measure?

Here are some of the most common **big O** notations that you'll come across, with a brief description of what they mean.

The notations appear in the order of how well they scale. As we get further into the slides, the algorithms will take increasingly larger amounts of time, as `n` increases.

O(1)

O notation	Common name	Description and examples
O(1)	constant time	<p>The algorithm takes the same amount of time, no matter how many items it has to process.</p> <ul style="list-style-type: none">• Indexing into a list or array.• Retrieving a value from a dictionary by its key.

O(1) means constant time.

You can't get better than a constant time algorithm. It takes the same amount of time, no matter how large `n` becomes. Examples include retrieving a value from a list, using its index position; and getting a value from a dictionary, using its key.

$O(\log n)$

O notation	Common name	Description and examples
$O(\log n)$	logarithmic time	<p>The number of operations increases by the log of n. If n is 2, $\log_2(n)$ is 1. For n = 1024, $\log_2(n)$ is 10. This indicates a very efficient algorithm. We've increased the number of items from 2 to 1024, but the complexity has only increased by 9.</p> <ul style="list-style-type: none">• Binary search.

If you can find an algorithm that scales in the order of $\log n$, **$O(\log n)$** , that's pretty good. Our binary search example was an $O(\log n)$ algorithm. We could get the correct value out of 10 numbers, with at most 4 guesses. When we increased the range of values to 1000, we only needed 10 guesses.

O(n)

O notation	Common name	Description and examples
O(n)	linear time	<p>The complexity increases as n increases.</p> <ul style="list-style-type: none">• Finding an item in a list.• Checking if a value exists in a dictionary (rather than checking for a key).• A for loop.

The time taken by a linear algorithm increases directly as 'n' increases. For example, if we're processing 2 items, we might perform 2 operations. When we increase the number of items to 16, that will take 16 operations. 128 items takes 128 operations.

$O(n \log n)$

O notation	Common name	Description and examples
$O(n \log n)$	$n \log n$	<p>The complexity increases as $n * \log(n)$ increases.</p> <ul style="list-style-type: none">• Various sorting functions, such as heapsort or merge sort.

$n \log n$ increases a bit faster than n , but not by a huge amount. We've seen that $\log(n)$ increases quite slowly, so n isn't being multiplied by large values. Using the previous example, processing 2 items might take 2 operations ($\log_2(2)$ is 1). 16 items increases from 16 operations to 64 (because $\log_2(16)$ is 4). With 128 items, the complexity increases to $128 * 7$, which is 896.

I know that looks like a huge increase over $O(n)$, but wait till you see the next two entries!

$O(n^2)$

O notation	Common name	Description and examples
$O(n^2)$	quadratic	<p>The complexity increases as `n` squared.</p> <ul style="list-style-type: none">• Simple sorting algorithms like bubble sort.• A nested for loop.

Things are now getting complex.

That doesn't mean $O(n^2)$ algorithms are bad, some tasks do require that much processing. But they slow down quite significantly, as you have larger data sets to process.

Using our previous figures, we might have 4 operations when processing 2 items; 256 operations with 16 items, and 16,384 operations when processing 128 items. As we increase the value of n , the computer has more and more work to do.

$O(n!)$

O notation	Common name	Description and examples
$O(n!)$	factorial time	<p>Factorial time algorithms take much more time to run, as the size of your data increases.</p> <ul style="list-style-type: none">• Complex algorithms, such as the "travelling salesman problem". Google it, if you want to find out what it involves.• Finding all permutations of an array (or list).

We wrote a factorial function, in the Functions section, and saw that the factorial of 16 is over 20 trillion. That's a lot of operations! Some tasks really are that complicated, and can take days to run, even on a supercomputer.

[Note: the travelling salesman problem can be reduced to $O(n^2 * 2^n)$, using some clever optimisations. That's still very slow, but an improvement over $O(n!)$]

Big O notation

Those were some examples of Big O notations, describing how they increase as the data size increases.

We also saw some examples of algorithms, for the various Big O notations.

In the next video, we'll look at some graphs, to see how the complexities increase.

Big O notation

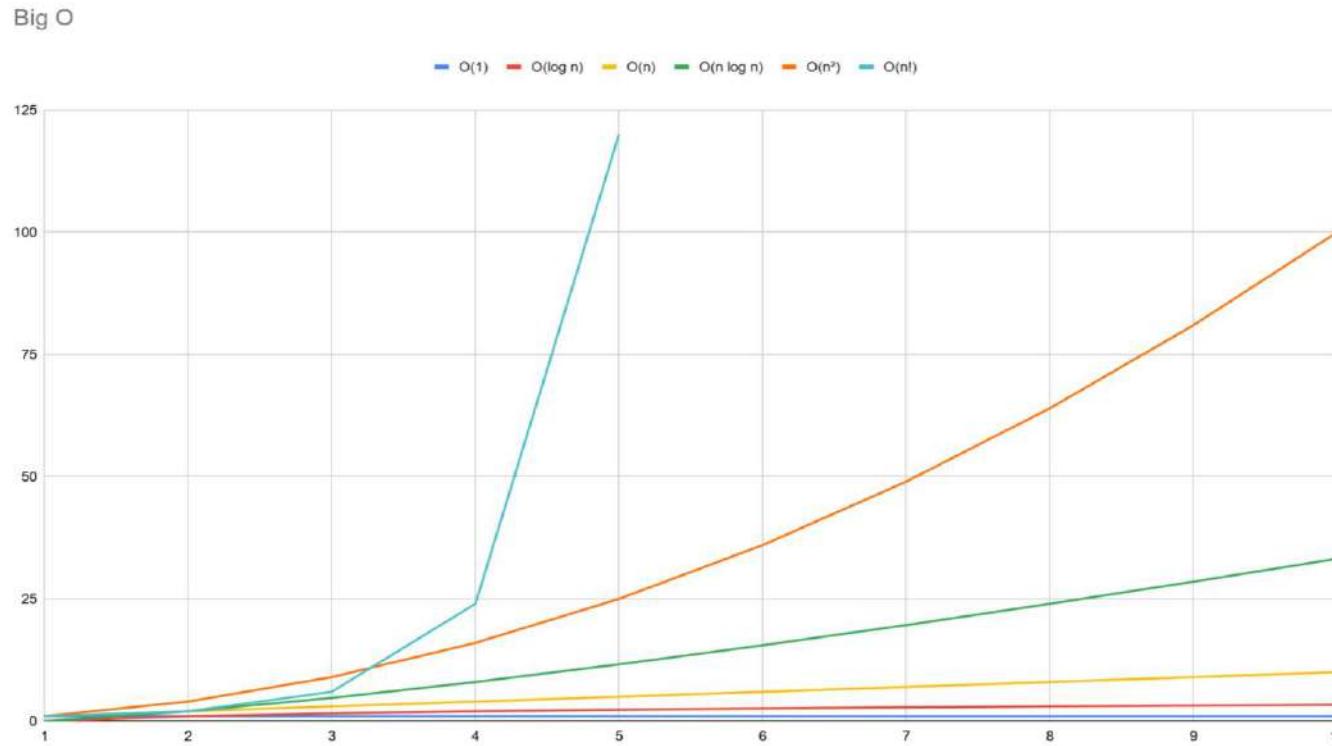
Before we look at some graphs, to see how the complexities increase, there are a few points to be aware of.

- Big O measures the complexity of an algorithm as its input grows. It doesn't tell you how long the algorithm will take to complete, it indicates how its time increases with larger inputs.
- If two functions are both $O(n)$, that doesn't mean they take the same time to execute. It means that, however long each one takes, it will take about ten times as long to process 1,000 items as it does to process 100.

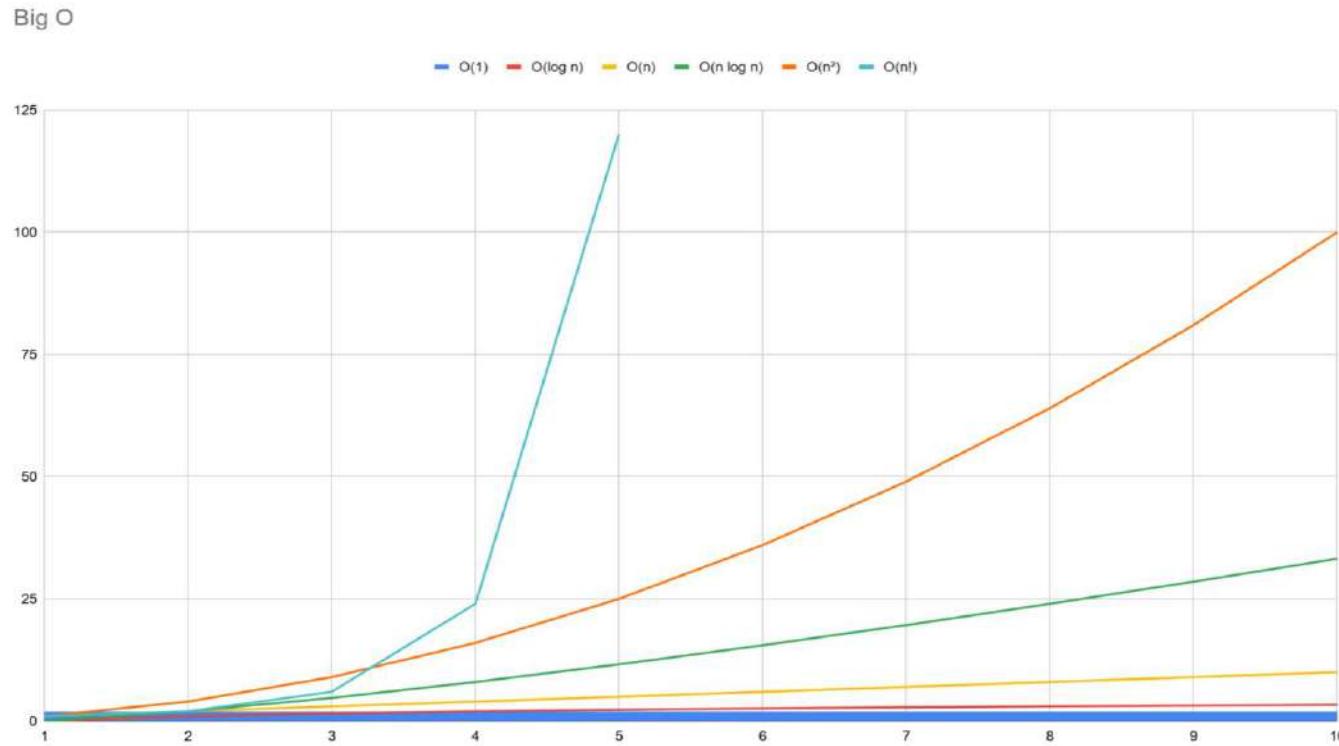
Big O notation

- Some calculations really are very complex. If something is $O(n^2)$, then it will take a long time with large `n`. Unless you can come up with a more efficient algorithm that does the same job, you're stuck with that. But understanding the complexity allows you to make decisions, such as doing the processing over night. Executing code that runs in *factorial* time, while a customer's waiting on the telephone, is something you might want to reconsider.
- Don't worry if you don't know what logarithms are. All you really need to understand, is how quickly things grow.

Big O notation



Big O notation



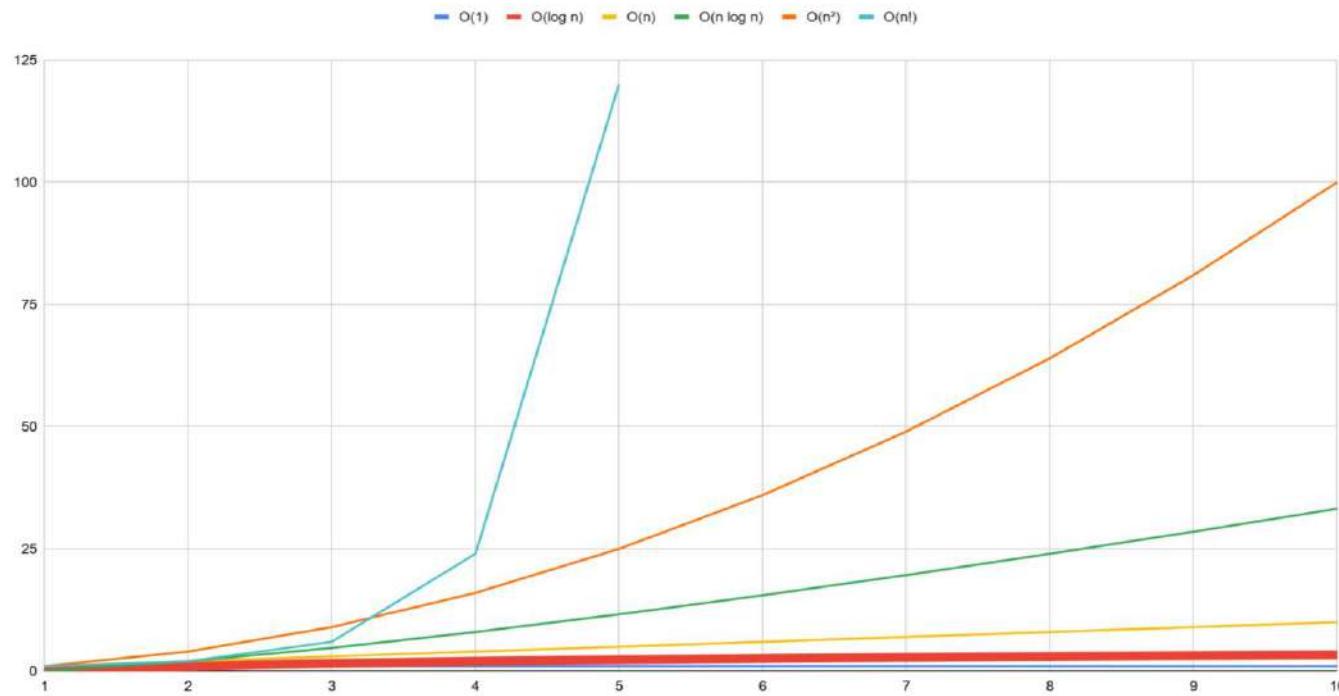
On this graph, we can see how each of the big O types increase, as `n` gets larger.

The constant time line, $O(1)$, is the blue, horizontal line along the bottom of the chart.

The time remains constant, no matter how many items we process.

Big O notation

Big O

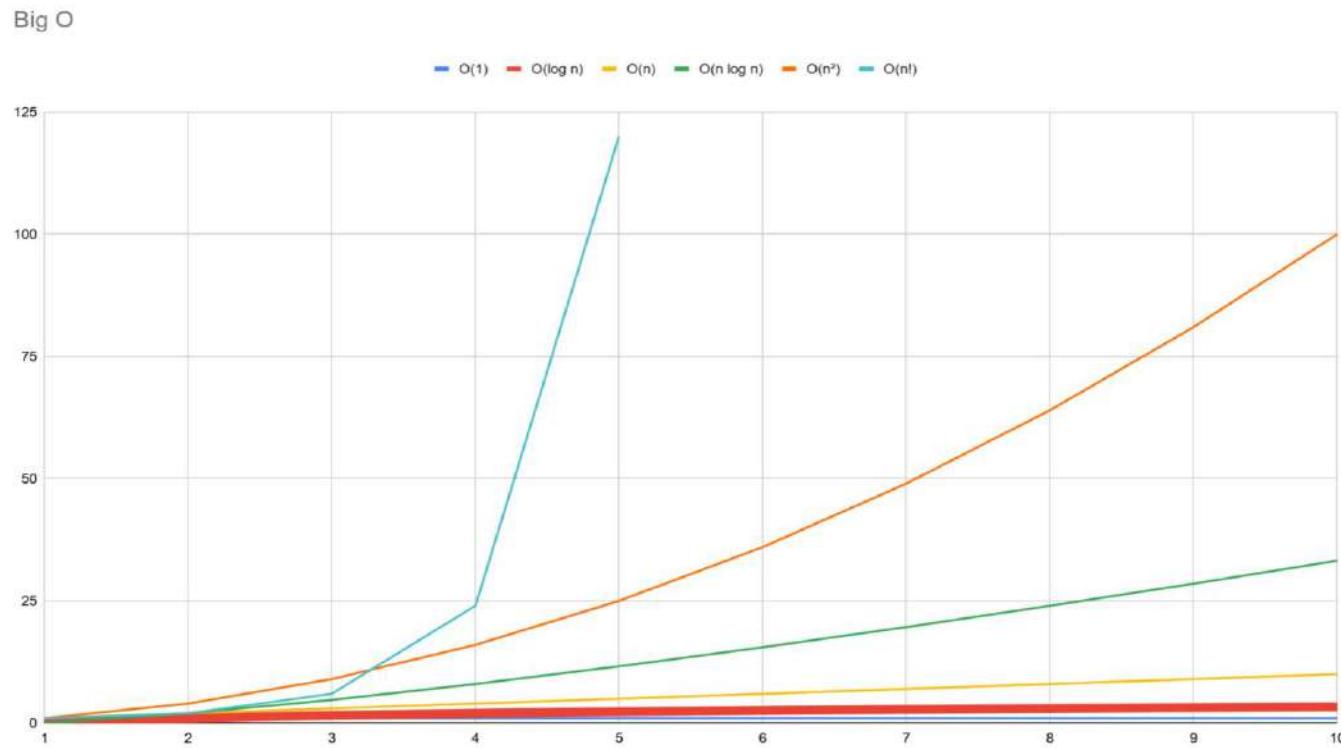


The red line represents $O(\log n)$.

That increases quite slowly. We've seen that log to base 2 of 1000 is about 10.

Using logs to base 10, $\log(1000)$ is 3.

Big O notation



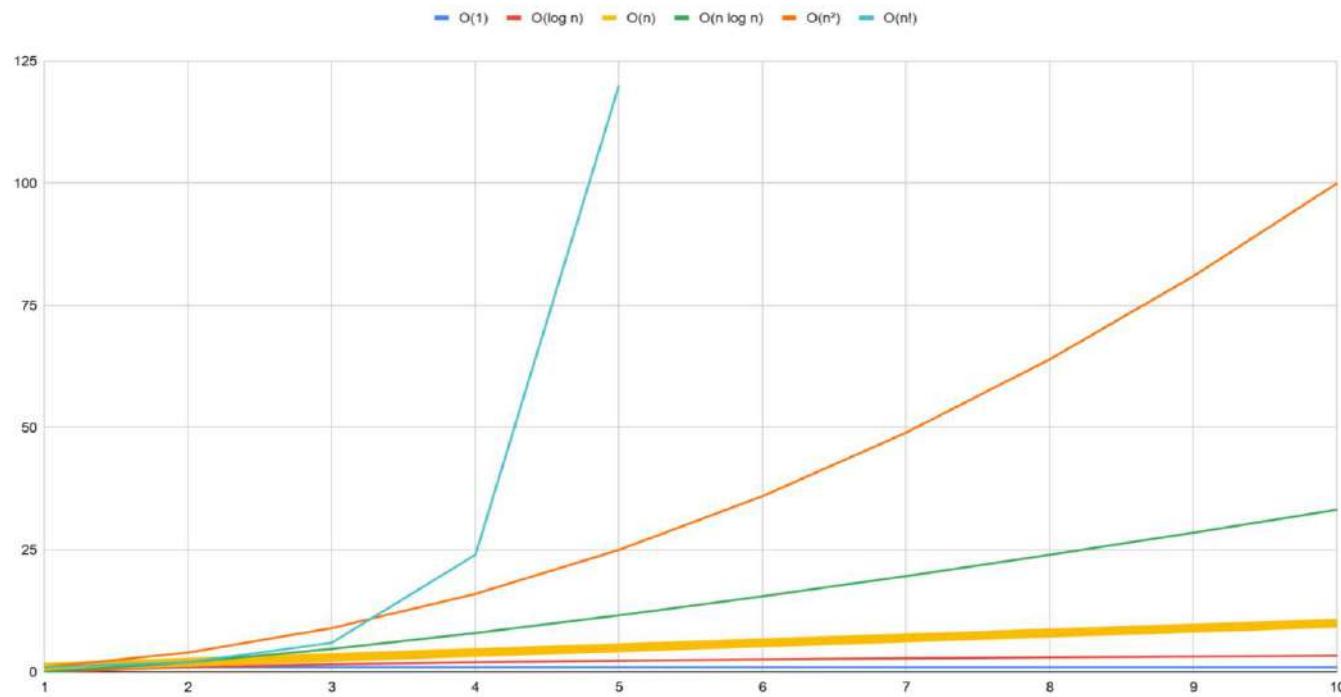
The red line represents $O(\log n)$.

The big O notation doesn't use any particular base for $\log n$.

As programmers, we tend to think of base 2 (binary), but $O(\log n)$ increases slowly whichever base is used.

Big O notation

Big O

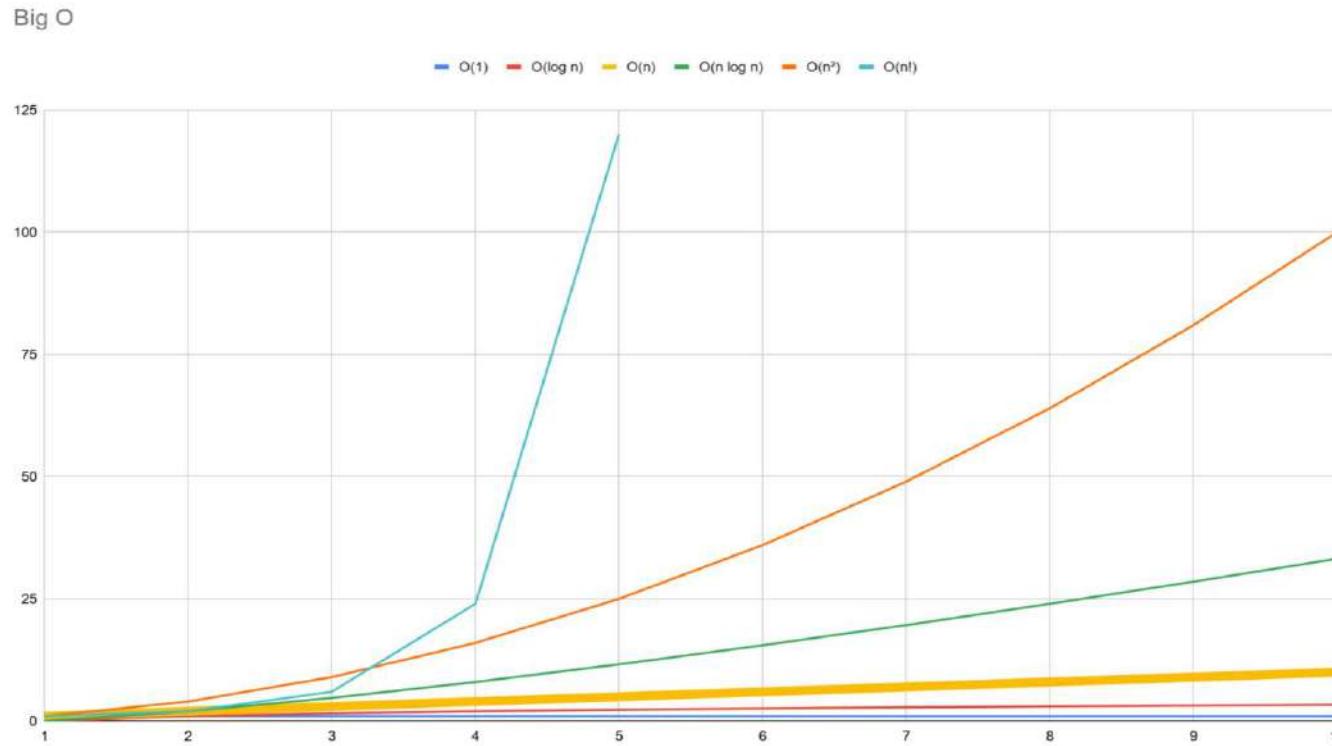


$O(n)$ is represented by the yellow line.

That's growing a lot faster than the first 2 lines.

The scale of the graph doesn't really represent how much quicker $O(n)$ is increasing.

Big O notation

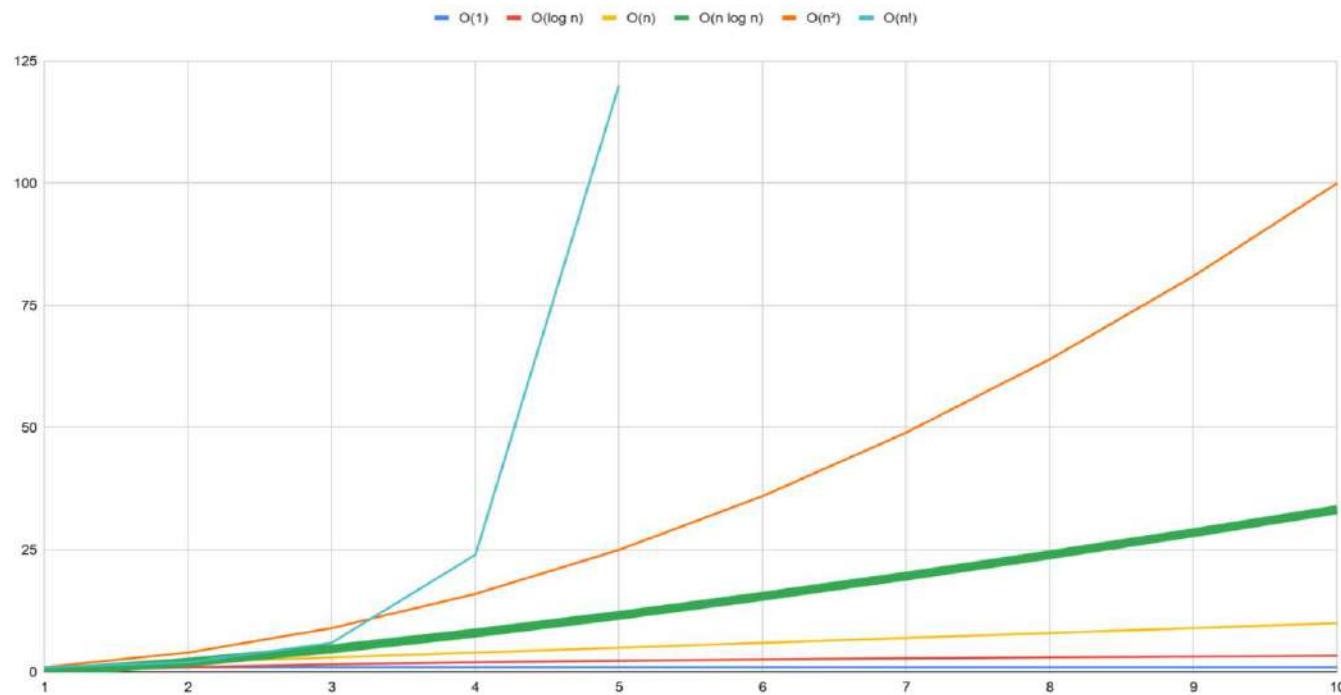


The scale of the graph doesn't really represent how much quicker $O(n)$ is increasing.

If we extended the graph right, to show the values for 125, $O(n)$ would be at the top right of the chart. $O(\log_{10} n)$ would be about 2.

Big O notation

Big O

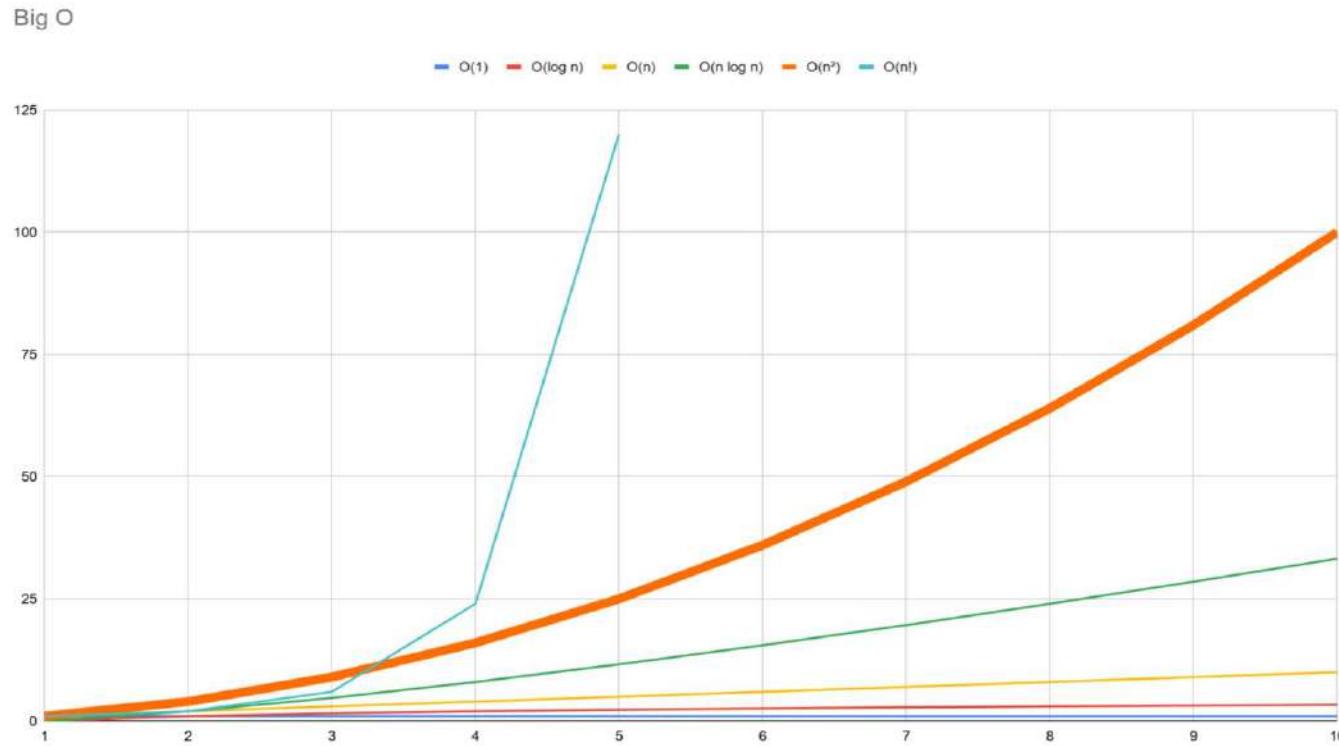


$O(n \log n)$ is represented by the green line.

Even with this limited range for 'n', from 1 to 10, you can see that it increases faster than the first 3 lines.

Many sorting functions run in $n \log n$ time. Sorting is quite a slow operation.

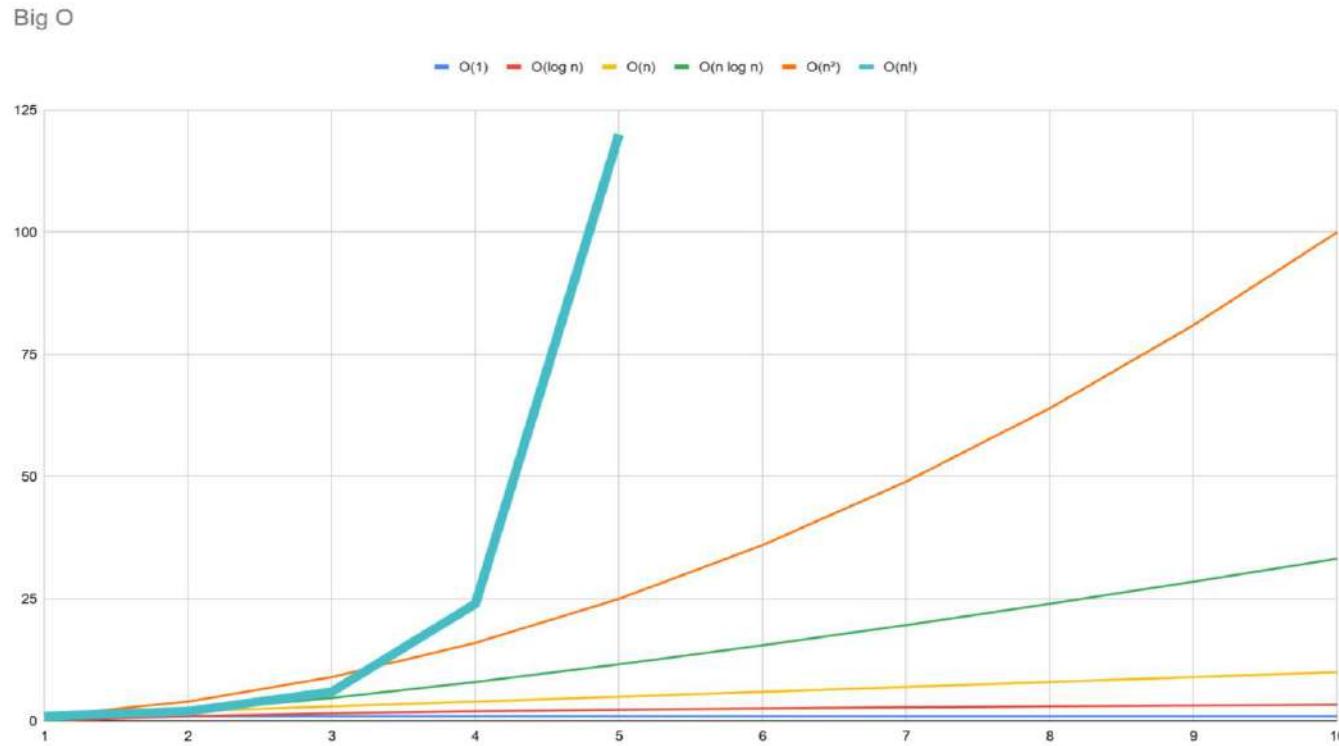
Big O notation



Now we get to algorithms that are really costly. They'll take a long time to complete, as the amount of data they're working with increases.

$O(n^2)$ is increasing quickly. With 1000 items to process, n^2 will be 1 million.

Big O notation



If you create an algorithm that runs in $O(n!)$ time, then you're trying to calculate something that's very complex.

As you can see, $O(n!)$ grows so quickly that we can only fit the first 5 values on the chart.

$6!$ is 720, which is way off the top.

Big O notation

You might have found some of that a bit scary!

Don't worry if you don't understand it all. The main thing you're interested in, is how the complexity affects the performance (or space requirements) of your code.

For example, an $O(n)$ algorithm will slow down more than an $O(\log n)$ algorithm, as ' n ' gets larger. As you process more and more data, in other words.

Just in case the charts were confusing, on the next slide we can see values for each of the orders we discussed.

Big O notation

The table shows how the various orders increase, as we process 10 times as much data, on each row.

If you come across an order that we haven't discussed, use your spreadsheet to calculate the values of the expression.

n	O(1)	O(log n)	O(n)	O(n log n)	O(n ²)	O(n!)	O(n ² * 2 ⁿ)
1	1	0	1	0	1	1	2
10	1	3	10	33	100	3628800	102400
100	1	7	100	664	10000	9E+157	1E+34
1000	1	10	1000	9,966	1000000	4E+2567	1E+307
10000	1	13	10000	132,877	100000000	2E+35659	1E+3018
100000	1	17	100000	1,660,964	100000000000	2E+456573	1E+30113
1000000	1	20	1000000	19,931,569	100000000000000		1E+301042
10000000	1	23	10000000	232,534,967	1000000000000000		
100000000	1	27	100000000	2,657,542,476	1E+16		
1000000000	1	30	1000000000	29,897,352,853	1E+18		

Big O notation

For example, I mentioned that there was an optimisation that reduces the *travelling salesman* problem to $O(n^2 * 2^n)$.

In the last column, I've calculated the values of $n^2 * 2^n$ for each value of n .

n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n!)$	$O(n^2 * 2^n)$
1	1	0	1	0	1	1	2
10	1	3	10	33	100	3628800	102400
100	1	7	100	664	10000	9E+157	1E+34
1000	1	10	1000	9,966	1000000	4E+2567	1E+307
10000	1	13	10000	132,877	100000000	2E+35659	1E+3018
100000	1	17	100000	1,660,964	100000000000	2E+456573	1E+30113
1000000	1	20	1000000	19,931,569	100000000000000		1E+301042
10000000	1	23	10000000	232,534,967	1000000000000000		
100000000	1	27	100000000	2,657,542,476	1E+16		
1000000000	1	30	1000000000	29,897,352,853	1E+18		

Big O notation

As you can see, it grows a lot slower than a factorial time ($n!$) algorithm.

When n is 100, $n!$ is 9 followed by 157 zeros. That's what $9E+157$ means — and it's a huge number.

n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n!)$	$O(n^2 * 2^n)$
1	1	0	1	0	1	1	2
10	1	3	10	33	100	3628800	102400
100	1	7	100	664	10000	9E+157	1E+34
1000	1	10	1000	9,966	1000000	4E+2567	1E+307
10000	1	13	10000	132,877	100000000	2E+35659	1E+3018
100000	1	17	100000	1,660,964	100000000000	2E+456573	1E+30113
1000000	1	20	1000000	19,931,569	1000000000000000		1E+301042
10000000	1	23	10000000	232,534,967	10000000000000000000		
100000000	1	27	100000000	2,657,542,476	1E+16		
1000000000	1	30	1000000000	29,897,352,853	1E+18		

Big O notation

$n^2 * 2^n$ is 1E+34, which is a 1 followed by 34 zeros. That's still huge, but a lot smaller than $n!$

The blank cells, by the way, are because my spreadsheet couldn't handle numbers that large.

n	O(1)	O(log n)	O(n)	O(n log n)	O(n ²)	O(n!)	O(n ² * 2 ⁿ)
1	1	0	1	0	1	1	2
10	1	3	10	33	100	3628800	102400
100	1	7	100	664	10000	9E+157	1E+34
1000	1	10	1000	9,966	1000000	4E+2567	1E+307
10000	1	13	10000	132,877	100000000	2E+35659	1E+3018
100000	1	17	100000	1,660,964	100000000000	2E+456573	1E+30113
1000000	1	20	1000000	19,931,569	100000000000000		1E+301042
10000000	1	23	10000000	232,534,967	1000000000000000		
100000000	1	27	100000000	2,657,542,476	1E+16		
1000000000	1	30	1000000000	29,897,352,853	1E+18		

Big O notation

It should be obvious that **Big O** doesn't tell you exactly how long an algorithm will take to execute, nor how many operations it will perform.

$\log(1)$ is zero, and we can't calculate anything in zero time.

The values just give an indication of how an algorithm will slow down, as n increases.

n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n!)$	$O(n^2 * 2^n)$
1	1	0	1	0	1	1	2
10	1	3	10	33	100	3628800	102400
100	1	7	100	664	10000	9E+157	1E+34
1000	1	10	1000	9,966	1000000	4E+2567	1E+307
10000	1	13	10000	132,877	100000000	2E+35659	1E+3018
100000	1	17	100000	1,660,964	1000000000000	2E+456573	1E+30113
1000000	1	20	1000000	19,931,569	1000000000000000		1E+301042
10000000	1	23	10000000	232,534,967	10000000000000000		
100000000	1	27	100000000	2,657,542,476	1E+16		
1000000000	1	30	1000000000	29,897,352,853	1E+18		

Bubble Sort

Bubble sort is the simplest sorting algorithm available. It's very basic, and isn't used in real applications.

It is a very useful algorithm for educational purposes though. Because it's so simple, we can concentrate on its performance, without getting bogged down in the details of how it's implemented.

The basic idea is that successive pairs of values are compared, and swapped if they're not in order.

Bubble Sort

As a result, large values bubble up to the end of the list.

I'm going to start with a very naive implementation of the **Bubble sort** algorithm. There are a couple of optimisations we can add to improve performance, and we'll do that once we've seen it working.

Big O gives an indication, not an exact value

Technically, **Bubble sort** is $O((n - 1)^2)$

Each loop goes round $n - 1$ times.

If you're happy with maths, $(n - 1)^2$ is the same as $n^2 - 2n + 1$.

But when specifying Big O, we're only interested in the term that increases the most.

$2n$ doesn't increase as fast as n^2 . And 1 doesn't increase at all.

So those terms are ignored, when specifying **Big O**.

Big O specifies an **asymptotic upper bound**, it's not an exact value.

Big O gives an indication, not an exact value

If you're not happy with maths, all you need to know is that you'd drop things like -1 , and the 2 n , in this case, when specifying **Big O** for an algorithm.

Big O is an indication of how an algorithm performs, with increasing data sizes.

As we're about to see, we can reduce the number of comparisons. But **Bubble sort** is still $O(n^2)$.

What is the Big O of our improved Bubble sort?

Have a think about that question, and I'll tell you the answer in the next video.

What is the Big O of our improved Bubble sort?

In the last video, we optimised the **Bubble sort** algorithm.

With a list of 7 items, we reduced it from 36 comparisons, to only 21.

So is the Big O still **n squared** ?

And if so, why?

Summing all numbers from 1 to n

The answer is, it's still $O(n^2)$.

To answer **Why?**, we need to learn a simple arithmetic trick.

How quickly can you add up all the numbers from 1 to 20 (inclusive)?

You might start adding $1 + 2 + 3 + 4 + 5$ and so on.

But there's a much quicker way. You can find the sum of all numbers from 1 to n with the formula

$$n(n + 1) / 2$$

Summing all numbers from 1 to n

When $n = 20$, that's easy to calculate.

$n / 2$ is 10, so we get **10 * 21**, which equals 210.

Perform all the additions, if you don't believe that works.

How does that help?

The sum of all numbers from 1 to 6 is 21. That's how many comparisons our sort performed.

On the first pass, the inner loop goes round 6 times.

On the second pass, it goes round 5 times.

The next pass goes round 4 times.

On the last pass, the inner loop only executes once.

The total number of comparisons is the sum of all the numbers between 1 and 6, inclusive.

How does that help?

So the formula for the number of comparisons is

$$n(n + 1) / 2$$

If you want to replace n with $n - 1$, that's fine. We'd get

$$(n - 1)(n - 1 + 1) / 2$$

or

$$n(n - 1) / 2$$

if you prefer.

I still don't see how that helps

To see why the optimised algorithm is still $O(n^2)$, we expand $n(n + 1) / 2$

It becomes

$$(n^2 + n) / 2$$

Or, if you prefer,

$$n^2 / 2 + n / 2$$

The term that grows the fastest there, is **n squared**. Our optimised Bubble sort is still $O(n^2)$.

I still don't see how that helps

I may be labouring the point here, but people often get confused when an $O(n^2)$ algorithm takes less than **n squared** operations.

It doesn't matter if you divide n squared by some value, or subtract something from it, the **order** is still **n squared**.

We're measuring the rate of increase

What $O(n^2)$ is saying, is that the algorithm's time (or space, if that's what's being measured) grows as **n squared** grows.

Another way to look at that, is to say that if you **double** n, the algorithm will take about **4** times as long.

If you multiply **n** by 10, the algorithm's time will increase in the order of 10 squared, or **100** times.

We may have had **n squared / 2** in our formula, but multiplying n by 10 will cause the complexity to increase by 100 times, not 100 divided by 2 (i.e. 50) times.

That's easy to demonstrate, by creating a bigger list for our program to sort.

Big O

Big O is used to describe the complexity of an algorithm, as its input increases.

Complexity can refer to execution time, or space requirements. For example, Merge sort has a worst case time complexity of $O(n \log n)$ and a space complexity of $O(n)$.

If memory is an issue, then you may prefer Heap sort.

Big O

Heap sort has a worst case time complexity of $O(n \log n)$, the same as Merge sort. But its space complexity is $O(1)$.

What that tells us, is that Merge sort will use more and more memory, as the size of the data increases.

Heap sort doesn't need more memory. $O(1)$ is constant, and doesn't change, when you have more data to process.

Big O

Big O doesn't describe how long an algorithm will take to run.

It describes how the time will increase, as the number of inputs (the size of the data) increases.

Similarly, it doesn't describe how much memory will be used.

It describes how the memory requirement will increase, as n gets larger.

Big O

In this section, you've seen how various Big O notations describe the complexity of an algorithm.

You've also seen how to implement a simple sorting algorithm — Bubble sort.

You probably wouldn't use bubble sort in a real program, but it does give a good idea of what a sorting algorithm does.

You also saw the steps that you can take, when optimising your code.

Optimisation

There are no hard and fast rules for how to optimise your code.

There are a lot of techniques available — in the **Dictionaries and sets** section, you saw that using a set can give a huge performance increase, rather than using a list.

The two optimisations, that we made to the bubble sort code, demonstrated the steps that you can follow.

We started out by understanding exactly what the code was doing.

Optimisation

Once you understand that, you can look for patterns and short-cuts.

Reducing the number of iterations, of our inner loop, had a significant impact on the performance.

We also stored the length of the list, in a variable, to save calling the `len` function repeatedly.

Optimisation

Writing efficient code comes with experience.

When you're starting to learn programming, you'll be pleased just to produce code that works.

Be pleased — and proud. Your early code won't be perfect, and it doesn't have to be.

The more code you write, the better your code will become.

Practise, write lots of code, and you'll find that each program will be better than the previous one.

Python Masterclass Remaster Slides

Challenge Slides.

COMPLETE PYTHON MASTERCLASS
Python Masterclass Remaster Slides



Mini Challenge

Add some code to the program, so that it prints out **we win**.

Each character should appear on a separate line.

The program should get the characters from the parrot string, using indexing.

The w is already printed out, you just need to print the remaining 5 characters.

With the text that is already being printed, the final output from the program should be:

Norwegian Blue

w
e

w
i
n

Challenge

Using the **letters** string from the video, add some code to create the following slices.

- Create a slice that produces the characters **qpo**.
- Slice the string to produce **edcba**.
- Slice the string to produce the last **8** characters, in reverse order.

Challenge

Using the **letters** string from the video, add some code to create the following slices.

- Create a slice that produces the characters **qpo**.
- Slice the string to produce **edcba**.
- Slice the string to produce the last **8** characters, in reverse order.

Challenge

Change the condition on line 6 to

if guess == answer:

then change the program to give the correct results.

```
5
6 if guess != answer:
7     if guess < answer:
8         print("Please guess higher")
9     else: # guess must be greater than answer
10        print("Please guess lower")
11    guess = int(input())
12    if guess == answer:
```

Challenge

Change the condition on line 6 to

if guess == answer:

then change the program to give the correct results.

```
5
6  if guess != answer:
7  if guess < answer:
8      print("Please guess higher")
9  else: # guess must be greater than answer
10     print("Please guess lower")
11     guess = int(input())
12     if guess == answer:
```

Challenge

Write a small program to ask for a name and an age.

When both values have been entered, check if the person is the right age to go on an 18-30 holiday (they must be over 18 and under 31).

If they are, welcome them to the holiday, otherwise print a (polite) message refusing them entry.

Our programs expect valid input. We'll see how to handle invalid numbers, later in the course.

Challenge

Remember that human languages aren't always very precise.

Someone counts as "over 18" from 1 second after midnight on their birthday.

Their actual age **equals** 18, but they're counted as being **over** 18 when considering things like voting, or getting a credit card.

Mini Challenge

What happens if we:

1. Indent the `print("-----")` line another 4 spaces?
2. Remove the indent completely for that line?

Work out what you'd expect to happen, before you try it.

Challenge

Modify the program to use a **while** loop, to allow the player to keep guessing.

The program should let the player know whether to guess higher or lower, and should print a message when the guess is correct.

It already does that, but I suggest you don't worry about printing a different message, if they get the correct answer on their first guess.

We'll look at counting the number of guesses later.

A correct guess will terminate the program.

As an optional extra, allow the player to quit by entering 0 (zero) for their guess.

Challenge

Write a program to print a number of options, and allow the user to select an option from the list.

The options should be numbered from 1 to 9 - although you can use less than 9 options if you want.

Make sure there are at least 4 options.

Challenge

As an example, your program might display:

```
Please choose your option from the list below:  
1. Learn Python  
2. Learn Java  
3. Go swimming  
4. Have dinner  
5. Go to bed  
0. Exit
```

Challenge

The program should continue looping, allowing the user to choose one of the options each time round.

The loop should only terminate when the user chooses 0.

If the user makes a valid choice, the program should print a short message.

The message should include the value that they typed.

Challenge

Don't print a different message for each choice - the only thing that should change is the number they typed.

Although you could print out the description, I want to show you a much better way of doing that, later.

We're keeping this simple, because there are still lots of things we haven't covered yet.

If their choice isn't one of the options in the menu, nothing should be printed (although you will see their input on the screen).

Challenge

As an optional extra, modify the program so that the menu is printed again, if they choose an invalid option.

Be careful with that one: you may start off by duplicating the code to print the menu, but it's possible to write the program without duplicating the print lines.

Mini Challenge

Change the program to have another option, option **6** for an **hdmi** cable.

Remember that you'll need to make a change in **3** places.

Nested List Challenge

Write code to print out all the meals in the menu, but with **spam** removed.

You can choose which approach you want to use:

- Remove spam from each list, then print the list.
- Print out the items in each list, as long as it's not **spam**.

You may want to write two sets of code, using both approaches.

Challenge

It's up to you how you format the output. What's important is that you produce eight meals, all without spam.

The output should contain something like:

```
['egg', 'bacon']
['egg', 'sausage', 'bacon']
['egg']
['egg', 'bacon']
['egg', 'bacon', 'sausage']
['bacon', 'sausage']
['sausage', 'bacon', 'tomato']
['egg', 'bacon']
```

Challenge

We're practising processing a nested list here. Don't worry too much about formatting your output.

You'll struggle to print several items on the same line, when coding the second approach. That's fine - print each one on a separate line.

After showing my solution, I'll use this example to explain how to print several things on the same line, from inside a **for** loop.

Challenge

This challenge is quite simple. We want to change the behaviour of the program.

At the moment, if you enter an invalid choice for a song, the program terminates.

That's fine when the albums are being displayed, but we want slightly different behaviour when the user's choosing a song.

Instead of exiting the program, an invalid song choice should display the list of albums again.

That will allow the user to go back to the albums, without choosing a song to play.

Palindrome Function Challenge

Fix our **is_palindrome** function so that it ignores the case of the letters, when checking if the two strings are equal.

Remember to test the function with words that are palindromes, using a mix of upper and lower case letters.

Also test that it correctly identifies words that **aren't** palindromes.

Palindrome Sentence Challenge

Create a new function called **palindrome_sentence**.

The function should return True if the sentence is a palindrome - False, otherwise.

Remember that we ignore spaces, punctuation and things like tabs and line feeds. We're only interested in alphanumeric characters.

Check out the **String Methods** in the documentation, to find one that's suitable for identifying if a character is alphanumeric.

There are two methods you could use, depending on whether you want to allow numbers or not.

Palindrome Sentence Challenge

Once you've excluded invalid characters from the string, the remainder of the function will be very similar to what we did for the **is palindrome** function.

Test your code thoroughly.

get_integer Challenge

Modify the **get_integer** function so that it prints a message, if the user enters an invalid number.

Challenge

Modify the **banner_text** function so that it takes another argument, the width for the banner.

You'll need to provide that argument in all the lines that call the function.

Docstrings Challenge

Create Docstrings for the three functions that we wrote, in the **functions.py** module.

Check your Docstrings by using **Ctrl-Q (Ctrl-J on a Mac)**, to make sure they're formatted correctly, and provide all the information someone would need, to use the functions.

Note: make sure you test your functions, after adding the Docstrings, to make sure you haven't broken anything.

Challenge

We don't have a Docstring for this function, so your challenge is to add one.

You should document the exception that can be raised.

Search on-line for something like **python exception in docstring** to see how to do that.

I won't record a video with our idea of what it should contain. You can find our attempt in the document that appears next.

Hint: The return value shouldn't be documented - this function doesn't return anything useful.

Add your Docstring, and compare it against our suggestion.

Challenge: Playing Fizz Buzz

For this challenge, you'll use the function that you wrote in the last exercise.

You'll do this in your IDE, because the on-line Python interpreter can't accept input from the keyboard.

The Python interpreter for Udemy's code exercises is running on a remote server. That means you're not sitting at its keyboard, and have no way to type in your input.

Challenge Details

For this challenge, replace that loop (the one I showed) with a loop that will play the game.

The computer will start with the value **1**.

The player and the computer will then take turns.

The player will type in their response, and the computer will print out its next value.

And so on.

The game ends when the player makes a mistake, or gets to 100.

If the player gets it wrong, they lose.

Challenge Details

This challenge doesn't involve writing another function - you're using a function that you've already written, in the previous exercise.

My solution will be in the next video. In the remainder of this video, I'll plan the program, so that we know what we're trying to do.

You may want to have a go at that yourself. Write the steps in whichever language you think in. In my case, that's English - or Australian 😊.

When planning a program, write in your native language. That way, you'll produce clear steps describing what the program is going to do.

Challenge Details

If you dive straight in and start writing code, you'll find it very difficult. Write the steps in the language that you think in.

A good rule is, **reach for a pen and paper before you reach for your keyboard.**

Pause the video now, and have a go, or continue watching if you want to see my English solution first.

Pseudocode

Pseudocode is code that you write in a simplified language - usually a simple version of your native language.

It's a simplified version, because computer languages are much simpler than human languages.

You may find that hard to believe. Python probably seems quite difficult to write at the moment; but programming languages have very few words, and very few grammatical structures.

Pseudocode

It's very easy to say, in English, that "the computer and the player take turns". But phrasing that with the limited vocabulary, and grammar, of a programming language needs a bit of thought.

So here's our pseudocode for this challenge ...

Pseudocode

The basic steps can be described as:

Calculate the computer's "number" (which may be a number, or may be "fizz", "buzz" or "fizz buzz").

Print the computer's response.

Calculate the player's correct answer.

Get input from the player.

Pseudocode

Compare the correct answer to the player's input.

Repeat, until the player makes a mistake, or we reach 100.

Print a suitable message: "Congratulations" or "Wrong".

Pseudocode

You might then refine that slightly, to produce something like

Call `fizz_buzz()` and print the result

(We'll need a variable to pass to `fizz_buzz`).

Calculate the player's correct answer

(Increment the variable that was previously passed to `fizz_buzz`, and use it again here).

Pseudocode

Get input from the player.

Check the player's input against the correct answer.

If it's not correct, break out of the loop. Print a "Wrong" message.

Repeat until we reach 100.

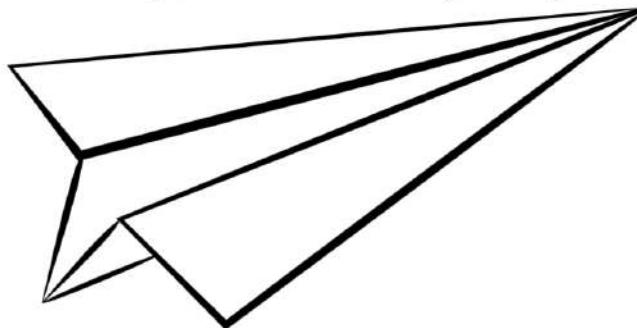
Print a suitable congratulations message.

We've now got the basic steps that we need to perform, and are ready to convert it to valid Python code.

I'll go over my solution in the next video.

Challenge

Time to come back to earth - the only plane I'm likely to fly is one of these:



Add the code to insert the value "**glider**" to our dictionary, with the key "**toy**".

Pause the video, and come back when you've made that change.

Challenge

Add an **else** block to our code, and print the menu when the user's choice isn't in the dictionary.

If you're not sure how to do that, review the video **Iterating over a dictionary**. The loop in your **else** block will be very similar to the code we used to iterate over the dictionary's keys and values.

Challenge

It's very common to use both lists and dictionaries in the same program.

Have a go at adding and removing items from a `computer_parts` list, and I'll add my code as a document, after this video.

Remember to use the **dict_list.py** file, that we copied earlier.

The new code that you add will be very similar to the code that we've already got, in the original **buy_computer.py** program.

Shopping List Challenge

At the moment, our program prints out the items that we need to buy.

Just printing to the screen isn't very useful - unless you want to write it down, or send it to a printer, and take bits of paper when you go shopping.

That's so "last century" 😊.

Modify the program, so that it puts the items we need to buy into some sort of data structure.

It's entirely up to you what type of structure you use.

You may decide to create some sort of list, or a dictionary.

Shopping List Challenge

The important thing is, that you must produce something that can be iterated over, to retrieve the ingredient and quantity.

Hint: We did something similar with the **buy_computer_dict** program, earlier in this section.

Another hint: Whatever data structure you choose, use a **function** to add the items to it.

Simple Deep Copy Challenge

Write a function that takes a dictionary as an argument, and returns a deep copy of the dictionary.

You're going to write your own function, to do a similar job to the `deepcopy` function that we've just used. But you'll do it **without** using the `copy` module.

Your function will be a lot simpler than `deepcopy`. It only has to cope with 1 level of contained objects. It should be able to successfully copy dictionaries like our `animals` or `recipes` dictionaries.

It doesn't have to handle dictionaries that contain objects that also contain objects. That's much too difficult at this stage.

Simple Deep Copy Challenge

The basic approach will be to create a new, empty dictionary.

Next, iterate over the keys and values of the dictionary that's being copied.

For each key, copy the value, then add the copy of the value to the new dictionary.

Your function only has to handle values that are dictionaries or lists. Both of those objects have a **copy** method.

Simple Deep Copy Challenge

The code to test your function might be something like this:

```
1  from contents import recipes
2
3
4  def my_deepcopy(d: dict) -> dict:
5      """Copy a dictionary, creating copies of the `list` or `dict` values.
6
7      The function will crash with an AttributeError if the values aren't
8      lists or dictionaries.
9
10     :param d: The dictionary to copy.
11     :return: A copy of `d`, with the values being copies of the original values.
12     """
13
14     # Your code goes here.
15
16
17     recipes_copy = my_deepcopy(recipes)
18     recipes_copy["Butter chicken"]["ginger"] = 300
19     print(recipes_copy["Butter chicken"]["ginger"])
20     print(recipes["Butter chicken"]["ginger"])
```

Simple Deep Copy Challenge

The output you should get will be:

300

3