In lesson one, we'll be covering models, prompts, and parsers. So models refers to the language models underpinning a lot of it. Prompts refers to the style of creating inputs to pass into the models. And then parsers is on the opposite end. It involves taking the output of these models and parsing it into a more structured format so that you can do things downstream with it. Yep, so when you build an application using LLM, there'll often be reusable models. We repeatedly prompt a model, parses outputs, and so LangChain gives an easy set of abstractions to do this type of operation. So with that, let's jump in and take a look at models, prompts, and parsers. So to get started, here's a little bit of starter code. I'm going to import OS, import OpenAI, and load my OpenAI secret key. The OpenAI library is already installed in my Jupyter Notebook environment. If you're running this locally, and you don't have OpenAI installed yet, you might need to run that. Bang, pip install OpenAI, but I'm not going to do that here. And then here's a helper function. This is actually very similar to the helper function that you might have seen in the ChaiGPT prompt engineering for developers course that I offered together with OpenAI's Iza Fulford. And so, with this helper function, you can say get completion on what is 1 plus 1, and this will call ChatGPT, or technically the model, GPT 3.5 Turbo, to give you an answer back like this.  Now, to motivate the line chain abstractions for model prompts and parsers, let's say you get an email from a customer in a language other than English. In order to make sure this is accessible, the other language I'm going to use is the English pirate language, where the consists are, I'd be fuming that me blender lid flew off and splattered my kitchen walls with smoothie. And to make matters worse, the warranty don't cover the cost of cleaning up me kitchen. I need your help right now, matey. And so, what we will do is, ask this LLM to translate the text to American English in a calm and respectful tone. So I'm going to set style to American English in a calm and respectful tone. And so, in order to actually accomplish this, if you've seen a little bit of prompting before, I'm going to specify the prompt using an f-string with the instructions, translate the text that is delimited by triple backticks into style that is style, and then plug in these two styles. And so, this generates a prompt that says translate the text and so on. I encourage you to pause the video and run the code, and also try modifying the prompt to see if you can get a different output. You can then prompt the Large Language Model to get a response. Let's see what the response is. Says translated the English pirate's message into this very polite, I'm really frustrated that my blender lid flew off and made a mess of my kitchen walls with smoothie and so on. I could really use your help right now, my friend. That sounds very nice. So, if you have different customers writing reviews in different languages, not just English pirate, but French, German, Japanese, and so on, you can imagine having to generate a whole sequence of prompts to generate such translations. Let's look at how we can do this in a more convenient, way using LangChain. I'm going to import chat OpenAI. This is LangChain's abstraction for the chatGPT API endpoint.  And so, if I then set chat equals chat OpenAI and look at what chat is, it creates this object as follows that uses the chatGPT model, which is also called GPT 3.5 turbo.  When I'm building applications, one thing I will often do is set the temperature parameter to be equal to zero. So the default temperature is 0.7. But let me actually redo that with temperature equals 0.0, and now the temperature is set to 0 to make this output a little bit less random. And now, let me define the template string as follows. Translate the text delimited by triple vectors into style that is style, and then here's the text. And to repeatedly reuse this template, let's import LangChain's chat prompt template, and then, let me create a prompt template using that template string that we just wrote above. From the prompt template, you can actually extract the original prompt, and it realizes that this prompt has two input variables, the style and the text, which were shown here with the curly braces. And here is the original template as well that we had specified. In fact, if I print this out, it realizes it has two input variables, style and text. Now, let's specify the style. This is a style that I want the customer message to be translated to, so I'm going to call this customer style,  and here's my same customer email as before. And now, if I create customer messages, his will generate the prompt, and will pass this large language model in a minute to get a response. So if you want to look at the types, the customer message is actually a list, and if you look at the first element of the list, this is more or less that prompt that you would expect this to be creating. Lastly, let's pass this prompt to the LLM, so I'm going to call chat, which we had set earlier, as a reference to the OpenAI chatGPT endpoint, and, if we print out the customer responses content, then, it gives you back this text translated from English pirate to polite American English. And of course, you can imagine other use cases where the customer emails are in other languages and this too can be used to translate the messages for an English-speaking to understand and reply to. I encourage you to pause the video and run the code and also try modifying the prompt to see if you can get a different output. Now let's hope our customer service agent replies to the customer in their original language. So let's say, English-speaking customer service agent writes this and says, "Hey there customer, warranty does not cover cleaning expenses for your kitchen because it's your fault. You misused your blender by forgetting to put on the lid. Tough luck. See ya." Now they're polite message, but let's say this is what a customer service agent wants. We are going to specify that the service message is going to be translated to this pirate style. So we want it to be in a polite tone that speaks in English pirate. And because we previously created that prompt template, the cool thing is, we can now reuse that prompt template and specify that the output style we want is this service style pirate and the text is this service reply. And if we do that, that's the prompt. And if we prompt, ChaiGPT, this is the response it gives us back. "Ahoy there, matey! I must kindly inform you that the warranty be not covering the expenses or cleaning your galley." And so on. Aye, tough luck. Farewell, me hearty. So, you might be wondering, why are we using prompt templates instead of, you know, just an f-string? The answer is that as you build sophisticated applications, prompts can be quite long and detailed.  And so, prompt templates are a useful abstraction to help you reuse good prompts when you can. This is an example of a relatively long prompt to grade a student's submission for an online learning application. And a prompt like this can be quite long, in which you can ask the LLM to first solve the problem, and then have the output in a certain format, and the output in a certain format.  And wrapping this in a LangChain prompt makes it easier to reuse a prompt like this. Also, you see later that LangChain provides prompts for some common operations, such as summarization, or question answering, or connecting to SQL databases, or connecting to different APIs. And so by using some of LangChain's built-in prompts, you can quickly get an application working without needing to, engineer your own prompts. One other aspect of LangChain's prompt libraries is that it also supports output parsing, which we'll get to in a minute. But when you're building a complex application using an LLM, you often instruct the LLM to generate its output in a certain format, such as using specific keywords. This example on the left illustrates using an LLM to carry out something called chain of thought reasoning using a framework called the React framework. But don't worry about the technical details, but the keys of that is that the thought is what the LLM is thinking, because by giving an LLM space to think, it can often get to more accurate conclusions. Then action as a keyword to carry the specific action, and then observation to show what it learned from that action, and so on. And if you have a prompt that instructs the LLM to use these specific keywords, thought, action, and observation, then this prompt can be coupled with a parser to extract out the text that has been tagged with these specific keywords. And so that together gives a very nice abstraction to specify the input to an LLM, and then also have a parser correctly interpret the output that the LLM gives. And so with that, let's return to see an example of an output parser using LangChain. In this example, let's take a look at how you can have an LLM output JSON, and use LangChain to parse that output. And the running example that I'll use will be to extract information from a product review, and format that output in a JSON format. So here's an example of how you would like the output to be formatted.  Technically, this is a Python dictionary, where whether or not the product is a gift, maps to false, the number of days it took to deliver was five, and the price value was pretty affordable. So this is one example of a desired output, here is an example of customer review, as well as a template to try to get to that JSON output. So here's a customer review. It says, this lead blower is pretty amazing. It has four settings, candle blower, gender breeze, windy city, and tornado. It arrived in two days, just in time for my wife's anniversary present. I think my wife liked it so much she was speechless. So far, I've been the only one using it, and so on. And here's a review template. For the following text, extract the following information. Specify, was this a gift? So in this case, it would be yes, because this is a gift. And also, delivery days. How long did it take to deliver? It looks like in this case, it arrived in two days. And what's the price value? You know, slightly more expensive than the lead blowers, and so on. So, the review template asks the LLM to take as input a customer review and extract these three fields and then format the output as JSON with the following keys. All right. So here's how you can wrap this in LangChain. Let's import the chat prompt template. We'd actually imported this already earlier. So technically, this line is redundant, but I'll just import it again and then have the prompt templates created from the review template up on top. And so, here's the prompt template. And now, similar to our earlier usage of a prompt template, let's create the messages to pass to the OpenAI, endpoint. Create the OpenAI endpoint, call that endpoint, and then let's print out the response. I encourage you to pause the video and run the code. And there it is. It says, give us true, delivery day is 2, and the price value also looks pretty accurate. But note that if we check the type of the response, this is actually a string. So it looks like JSON and looks like it has key-value pairs, but it's actually not a dictionary. This is just one long string. So what I'd really like to do is go to the response content and get the value from the gift key which should be true, but I run this, this should generate an error because, well, this is actually a string. This is not a Python dictionary. So, let's see how we will use LangChain's parser in order to do this. I'm going to import response schema and structured output parser from LangChain. And I'm going to tell it what I wanted to parse by specifying these response schemas. So the gif schema is named gif, and here's the description. Was the item purchased as a gift for someone else? Answer true or yes, false if not or unknown, and so on. So, have a gift schema, delivery date schema, price value schema, and then let's put all three of them into a list as follows. Now that I've specified the schema for these LangChain can actually give you the prompt itself by having the output parser tell you what instructions it wants you to send to the LLM. So if I were to print format instructions. She has a pretty precise set of instructions for the LLM that will cause it to generate an output that the output parser can process. So here's the new review template, and the review template includes the format instructions that LangChain generated. And so we can create a prompt from the review template too, and then create the messages that will pass to the OpenAI endpoint. If you want, you can take a look at the actual prompt, which gives the instructions to extract the fields gift, delivery days, price value, here's the text, and then here are the formatting instructions. Finally, if we call the OpenAI endpoint, let's take a look at what response we got. It is now this, and now if we use the output parser that we created earlier, you can then parse this into an output dictionary, which if I print, looks like this. And notice that this is of type dictionary, not a string, which is why I can now extract the value associated with the key gift and get true, or the value associated with delivery days and get two, or you can also extract the value associated with price value. So this is a nifty way to take your LLM output and parse it into a Python dictionary, to make the output easier to use in downstream processing. I encourage you to pause the video and run the code. And so, that's it for models, prompts, and parsers. With these tools, hopefully you'll be able to reuse your own prompt templates easily, share prompt templates with others that you're collaborating with, even use LangChain's built-in prompt templates, which as you just saw, can often be coupled with an output parser, so that the input prompt to output in a specific format and then the parser, parses that output to store the data in a Python dictionary or some other data structure that makes it easy for downstream processing. I hope you find this useful in many of your applications. And with that, let's go into the next video where we'll see how LangChain can help you build better chatbots, or have an LLM have more effective chats by better managing what it remembers from the conversation you've had so far.