intro_palm_api

October 9, 2023

```
[ ]: # Copyright 2023 Google LLC
     #
     # Licensed under the Apache License, Version 2.0 (the "License");
     # you may not use this file except in compliance with the License.
     # You may obtain a copy of the License at
     #
     #     https://www.apache.org/licenses/LICENSE-2.0
     #
     # Unless required by applicable law or agreed to in writing, software
     # distributed under the License is distributed on an "AS IS" BASIS,
     # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     # See the License for the specific language governing permissions and
     # limitations under the License.
```

1 Getting Started with the Vertex AI PaLM API & Python SDK

Run in Colab

View on GitHub

Open in Vertex AI Workbench

1.1 Overview

1.1.1 What are LLMs?

Large language models (LLMs) are deep learning models trained on massive datasets of text. LLMs can translate language, summarize text, generate creative writing, generate code, power chatbots and virtual assistants, and complement search engines and recommendation systems.

1.1.2 PaLM

Following its predecessor, PaLM, PaLM 2 is Google's next generation large language model that builds on Google's legacy of breakthrough research in machine learning and responsible AI. PaLM 2 excels at tasks like advanced reasoning, translation, and code generation because of how it was built.

PaLM 2 excels at advanced reasoning tasks, including code and math, classification and question answering, translation and multilingual proficiency, and natural language generation better than our previous state-of-the-art LLMs, including PaLM. It can accomplish these tasks because of the

way it was built – bringing together compute-optimal scaling, an improved dataset mixture, and model architecture improvements.

PaLM 2 is grounded in Google's approach to building and deploying AI responsibly. It was evaluated rigorously for its potential harms and biases, capabilities and downstream uses in research and in-product applications. It's being used in other state-of-the-art models, like Med-PaLM 2 and Sec-PaLM, and is powering generative AI features and tools at Google, like Bard and the PaLM API.

PaLM is pre-trained on a wide range of text data using an unsupervised learning approach, without any specific task. During this pre-training process, PaLM learns to predict the next word in a sentence, given the preceding words. This enables the model to generate coherent, fluent text resembling human writing. This large size enables it to learn complex patterns and relationships in language and generate high-quality text for various applications. This is why models like PaLM are referred to as "foundational models."

Creating an LLM requires massive amounts of data, significant compute resources, and specialized skills. Because LLMs require a big investment to create, they target broad rather than specific use cases. On Vertex AI, you can customize a foundation model for more specific tasks or knowledge domains by using prompt design and model tuning.

1.1.3 Vertex AI PaLM API

The Vertex AI PaLM API, released on May 10, 2023, is powered by PaLM 2.

1.1.4 Using Vertex AI PaLM API

You can interact with the Vertex AI PaLM API using the following methods:

- Use the Generative AI Studio for quick testing and command generation.
- Use cURL commands in Cloud Shell.
- Use the Python SDK in a Jupyter notebook

This notebook focuses on using the Python SDK to call the Vertex AI PaLM API. For more information on using Generative AI Studio without writing code, you can explore Getting Started with the UI instructions

For more information, check out the documentation on generative AI support for Vertex AI.

1.1.5 Objectives

In this tutorial, you will learn how to use PaLM API with the Python SDK and explore its various parameters.

By the end of the notebook, you should be able to understand various nuances of generative model parameters like `temperature`, `top_k`, `top_p`, and how each parameter affects the results.

The steps performed include:

- Installing the Python SDK
- Using Vertex AI PaLM API
  - Text generation model with `text-bison@001`

    - Understanding model parameters (`temperature`, `max_output_token`, `top_k`, `top_p`)
  - Chat model with `chat-bison@001`
  - Embeddings model with `textembedding-gecko@001`

1.1.6 Costs

This tutorial uses billable components of Google Cloud:

- Vertex AI Generative AI Studio

Learn about Vertex AI pricing, and use the Pricing Calculator to generate a cost estimate based on your projected usage.

1.1.7 Data governance and security

For more information, see the documentation on Data Governance and Generative AI on Google Cloud.

1.1.8 Responsible AI

Large language models (LLMs) can translate language, summarize text, generate creative writing, generate code, power chatbots and virtual assistants, and complement search engines and recommendation systems. At the same time, as an early-stage technology, its evolving capabilities and uses create potential for misapplication, misuse, and unintended or unforeseen consequences. Large language models can generate output that you don't expect, including text that's offensive, insensitive, or factually incorrect.

What's more, the incredible versatility of LLMs is also what makes it difficult to predict exactly what kinds of unintended or unforeseen outputs they might produce. Given these risks and complexities, the PaLM API is designed with Google's AI Principles in mind. However, it is important for developers to understand and test their models to deploy safely and responsibly. To aid developers, the Generative AI Studio has built-in content filtering, and the PaLM API has safety attribute scoring to help customers test Google's safety filters and define confidence thresholds that are right for their use case and business. Please refer to the Safety filters and attributes section to learn more.

When the PaLM API is integrated into a customer's unique use case and context, additional responsible AI considerations and PaLM limitations may need to be considered. We encourage customers to leverage fairness, interpretability, privacy and security recommended practices.

1.2 Getting Started

1.2.1 Install Vertex AI SDK

[ ]:
```
!pip install google-cloud-aiplatform --upgrade --user
```

Colab only: Uncomment the following cell to restart the kernel or use the button to restart the kernel. For Vertex AI Workbench you can restart the terminal using the button on top.

```
[ ]: # # Automatically restart kernel after installs so that your environment can␣
      ↪access the new packages
     # import IPython

     # app = IPython.Application.instance()
     # app.kernel.do_shutdown(True)
```

1.2.2 Authenticating your notebook environment

- If you are using **Colab** to run this notebook, uncomment the cell below and continue.
- If you are using **Vertex AI Workbench**, check out the setup instructions here.

```
[ ]: # from google.colab import auth
     # auth.authenticate_user()
```

1.3 Vertex AI PaLM API models

The Vertex AI PaLM API enables you to test, customize, and deploy instances of Google's large language models (LLM) called as PaLM, so that you can leverage the capabilities of PaLM in your applications.

1.3.1 Model naming scheme

Foundation model names have three components: use case, model size, and version number. The naming convention is in the format:
`<use case>-<model size>@<version number>`

For example, text-bison@001 represents the Bison text model, version 001.

The model sizes are as follows: - **Bison**: The best value in terms of capability and cost. - **Gecko**: The smallest and cheapest model for simple tasks.

1.3.2 Available models

The Vertex AI PaLM API currently supports five models:

- `text-bison@001` : Fine-tuned to follow natural language instructions and is suitable for a variety of language tasks.
- `chat-bison@001` : Fine-tuned for multi-turn conversation use cases like building a chatbot.
- `textembedding-gecko@001` : Returns model embeddings for text inputs.
- `code-bison@001`: A model fine-tuned to generate code based on a natural language description of the desired code. For example, it can generate a unit test for a function.
- `code-gecko@001`: A model fine-tuned to suggest code completion based on the context in code that's written.
- `codechat-bison@001`: A model fine-tuned for chatbot conversations that help with code-related questions.

You can find more information about the properties of these foundational models in the Generative AI Studio documentation.

1.3.3 Import libraries

Colab only: Uncomment the following cell to initialize the Vertex AI SDK. For Vertex AI Workbench, you don't need to run this.

[ ]:
```
# import vertexai

# PROJECT_ID = ""  # @param {type:"string"}
# vertexai.init(project=PROJECT_ID, location="us-central1")
```

[ ]:
```
import pandas as pd
import seaborn as sns
from IPython.display import Markdown, display
from sklearn.metrics.pairwise import cosine_similarity
from vertexai.language_models import TextGenerationModel, \
                                     TextEmbeddingModel, \
                                     ChatModel, \
                                     InputOutputTextPair, \
                                     CodeGenerationModel, \
                                     CodeChatModel
```

1.4 Text generation with `text-bison@001`

The text generation model from PaLM API that you will use in this notebook is `text-bison@001`. It is fine-tuned to follow natural language instructions and is suitable for a variety of language tasks, such as:

- Classification
- Sentiment analysis
- Entity extraction
- Extractive question-answering
- Summarization
- Re-writing text in a different style
- Ad copy generation
- Concept ideation
- Concept simplification

Load model

[ ]:
```
generation_model = TextGenerationModel.from_pretrained("text-bison@001")
```

Prompt design Prompt design is the process of creating prompts that elicit the desired response from a language model. Prompt design is an important part of using language models because it allows non-specialists to control the output of the model with minimal overhead. By carefully crafting the prompts, you can nudge the model to generate a desired result. Prompt design can be an efficient way to experiment with adapting an LLM for a specific use case. The iterative process

of repeatedly updating prompts and assessing the model's responses is sometimes called prompt engineering.

Hello PaLM Create your first prompt and send it to the text generation model.

```
[ ]: prompt = "What is a large language model?"

response = generation_model.predict(prompt=prompt)

print(response.text)
```

Try out your own prompt

- What are the biggest challenges facing the healthcare industry?
- What are the latest developments in the automotive industry?
- What are the biggest opportunities in the retail industry?
- (Try your own prompts!)

```
[ ]: prompt = """Create a numbered list of 10 items. Each item in the list should be␣
     ↪a trend in the tech industry.

Each trend should be less than 5 words.""" # try your own prompt

response = generation_model.predict(prompt=prompt)

print(response.text)
```

Prompt templates Prompt templates are useful if you have found a good way to structure your prompt that you can re-use. This can be also be helpful in limiting the open-endedness of freeform prompts. There are many ways to implement prompt templates, and below is just one example using f-strings.

```
[ ]: my_industry = "tech" # try changing this to a different industry

response = generation_model.predict(
    prompt=f"""Create a numbered list of 10 items. Each item in the list should
    be a trend in the {my_industry} industry.

    Each trend should be less than 5 words."""
)

print(response.text)
```

1.4.1 Model parameters for `text-bison@001`

You can customize how the PaLM API behaves in response to your prompt by using the following parameters for `text-bison@001`:

- `temperature`: higher means more "creative" responses

- `max_output_tokens`: sets the max number of tokens in the output
- `top_p`: higher means it will pull from more possible next tokens, based on cumulative probability
- `top_k`: higher means it will sample from more possible next tokens

The section below covers each parameter and how to use them.

The `temperature` parameter (range: 0.0 - 1.0, default 0)

What is *temperature*? The temperature is used for sampling during the response generation, which occurs when top_p and top_k are applied. Temperature controls the degree of randomness in token selection.

How does *temperature* affect the response? Lower temperatures are good for prompts that require a more deterministic and less open-ended response. In comparison, higher temperatures can lead to more "creative" or diverse results. A temperature of `0` is deterministic: the highest probability response is always selected. For most use cases, try starting with a temperature of `0.2`.

A higher temperature value will result in a more exploratative output, with a higher likelihood of generating rare or unusual words or phrases. Conversely, a lower temperature value will result in a more conservative output, with a higher likelihood of generating common or expected words or phrases.

Example: For example,

`temperature = 0.0`:

- *The cat sat on the couch, watching the birds outside.*
- *The cat sat on the windowsill, basking in the sun.*

`temperature = 0.9`:

- *The cat sat on the moon, meowing at the stars.*
- *The cat sat on the cheeseburger, purring with delight.*

Note: It's important to note that while the temperature parameter can help generate more diverse and interesting text, it can also increase the likelihood of generating nonsensical or inappropriate text (i.e. hallucinations). Therefore, it's important to use it carefully and with consideration for the desired outcome.

For more information on the `temperature` parameter for text models, please refer to the documentation on model parameters.

If you run the following cell multiple times, it should always return the same response, as `temperature=0` is deterministic.

[ ]:
```
temp_val = 0.0
prompt_temperature = "Complete the sentence: As I prepared the picture frame, I␣
 ↪reached into my toolkit to fetch my:"

response = generation_model.predict(
    prompt=prompt_temperature,
```

```
    temperature=temp_val,
)

print(f"[temperature = {temp_val}]")
print(response.text)
```

If you run the following cell multiple times, it may return different responses, as higher temperature values can lead to more diverse results, even though the prompt is the same as the above cell.

[ ]:
```
temp_val = 1.0

response = generation_model.predict(
    prompt=prompt_temperature,
    temperature=temp_val,
)

print(f"[temperature = {temp_val}]")
print(response.text)
```

The `max_output_tokens` parameter (range: 1 - 1024, default 128)

Tokens A single token may be smaller than a word. For example, a token is approximately four characters. So 100 tokens correspond to roughly 60-80 words. It's essential to be aware of the token sizes as models have a limit on input and output tokens.

What is *max_output_tokens*? `max_output_tokens` is the maximum number of tokens that can be generated in the response.

How does *max_output_tokens* affect the response? Specify a lower value for shorter responses and a higher value for longer responses. A token may be smaller than a word. A token is approximately four characters. 100 tokens correspond to roughly 60-80 words.

For more information on the `max_output_tokens` parameter for text models, please refer to the documentation on model parameters.

[ ]:
```
max_output_tokens_val = 5

response = generation_model.predict(
    prompt="List ten ways that generative AI can help improve the online␣
 ↪shopping experience for users",
    max_output_tokens=max_output_tokens_val,
)

print(f"[max_output_tokens = {max_output_tokens_val}]")
print(response.text)
```

[ ]:
```
max_output_tokens_val = 500
```

```
response = generation_model.predict(
    prompt="List ten ways that generative AI can help improve the online␣
 ↪shopping experience for users",
    max_output_tokens=max_output_tokens_val,
)

print(f"[max_output_tokens = {max_output_tokens_val}]")
print(response.text)
```

For easier reading, you can also render Markdown in Jupyter:

[ ]:
```
display(Markdown(response.text))
```

The `top_p` parameter (range: 0.0 - 1.0, default 0.95)

What is *top_p*? `top_p` controls how the model selects tokens for output by adjusting the probability distribution of the next word in the generated text based on a cumulative probability cutoff. Specifically, it selects the smallest set of tokens whose cumulative probability exceeds the given cutoff probability p, and samples from this set uniformly.

For example, suppose tokens A, B, and C have a probability of 0.3, 0.2, and 0.1, and the `top_p` value is 0.5. In that case, the model will select either A or B as the next token (using temperature) and not consider C, because the cumulative probability of top_p is <= 0.5. Specify a lower value for less random responses and a higher value for more random responses.

How does *top_p* affect the response? The `top_p` parameter is used to control the diversity of the generated text. A higher `top_p` parameter value results in more "diverse" and "interesting" outputs, with the model being allowed to sample from a larger pool of possibilities. In contrast, a lower `top_p` parameter value resulted in more predictable outputs, with the model being constrained to a smaller set of possible tokens.

Example: `top_p = 0.1`:

- The cat sat on the mat.
- The cat sat on the floor.

`top_p = 0.9`:

- The cat sat on the windowsill, soaking up the sun's rays.
- The cat sat on the edge of the bed, watching the birds outside.

For more information on the `top_p` parameter for text models, please refer to the documentation on model parameters.

[ ]:
```
top_p_val = 0.0
prompt_top_p_example = (
    "Create a marketing campaign for jackets that involves blue elephants and␣
 ↪avocados."
)
```

```
response = generation_model.predict(
    prompt=prompt_top_p_example, temperature=0.9, top_p=top_p_val
)

print(f"[top_p = {top_p_val}]")
print(response.text)
```

[ ]:
```
top_p_val = 1.0

response = generation_model.predict(
    prompt=prompt_top_p_example, temperature=0.9, top_p=top_p_val
)

print(f"[top_p = {top_p_val}]")
print(response.text)
```

The `top_k` parameter (range: 0.0 - 40, default 40)

What is *top_k*? `top_k` changes how the model selects tokens for output. A `top_k` of 1 means the selected token is the most probable among all tokens in the model's vocabulary (also called greedy decoding). In contrast, a `top_k` of 3 means that the next token is selected from the top 3 most probable tokens (using temperature). For each token selection step, the `top_k` tokens with the highest probabilities are sampled. Then tokens are further filtered based on `top_p` with the final token selected using temperature sampling.

How does *top_k* affect the response? Specify a lower value for less random responses and a higher value for more random responses.

For more information on the `top_k` parameter for text models, please refer to the documentation on model parameters.

[ ]:
```
prompt_top_k_example = "Write a 2-day itinerary for France."
top_k_val = 1

response = generation_model.predict(
    prompt=prompt_top_k_example, max_output_tokens=300, temperature=0.9,␣
 ↪top_k=top_k_val
)

print(f"[top_k = {top_k_val}]")
print(response.text)
```

[ ]:
```
top_k_val = 40

response = generation_model.predict(
    prompt=prompt_top_k_example,
    max_output_tokens=300,
```

```
    temperature=0.9,
    top_k=top_k_val,
)

print(f"[top_k = {top_k_val}]")
print(response.text)
```

1.5 Chat model with `chat-bison@001`

The `chat-bison@001` model lets you have a freeform conversation across multiple turns. The application tracks what was previously said in the conversation. As such, if you expect to use conversations in your application, use the `chat-bison@001` model because it has been fine-tuned for multi-turn conversation use cases.

[ ]:
```
chat_model = ChatModel.from_pretrained("chat-bison@001")

chat = chat_model.start_chat()

print(
    chat.send_message(
        """
Hello! Can you write a 300 word abstract for a research paper I need to write␣
 ↪about the impact of AI on society?
"""
    )
)
```

As shown below, the model should respond based on what was previously said in the conversation:

[ ]:
```
print(
    chat.send_message(
        """
Could you give me a catchy title for the paper?
"""
    )
)
```

1.5.1 Advanced Chat model with the SDK

You can also provide a `context` and `examples` to the model. The model will then respond based on the provided context and examples. You can also use `temperature`, `max_output_tokens`, `top_p`, and `top_k`. These parameters should be used when you start your chat with `chat_model.start_chat()`.

For more information on chat models, please refer to the documentation on chat model parameters.

[ ]:
```
chat = chat_model.start_chat(
    context="My name is Ned. You are my personal assistant. My favorite movies␣
 ↪are Lord of the Rings and Hobbit.",
```

```
    examples=[
        InputOutputTextPair(
            input_text="Who do you work for?",
            output_text="I work for Ned.",
        ),
        InputOutputTextPair(
            input_text="What do I like?",
            output_text="Ned likes watching movies.",
        ),
    ],
    temperature=0.3,
    max_output_tokens=200,
    top_p=0.8,
    top_k=40,
)
print(chat.send_message("Are my favorite movies based on a book series?"))
```

[ ]:
```
print(chat.send_message("When where these books published?"))
```

1.6 Embedding model with `textembedding-gecko@001`

Text embeddings are a dense, often low-dimensional, vector representation of a piece of content such that, if two pieces of content are semantically similar, their respective embeddings are located near each other in the embedding vector space. This representation can be used to solve common NLP tasks, such as:

- **Semantic search**: Search text ranked by semantic similarity.
- **Recommendation**: Return items with text attributes similar to the given text.
- **Classification**: Return the class of items whose text attributes are similar to the given text.
- **Clustering**: Cluster items whose text attributes are similar to the given text.
- **Outlier Detection**: Return items where text attributes are least related to the given text.

Please refer to the text embedding model documentation for more information.

[ ]:
```
embedding_model = TextEmbeddingModel.from_pretrained("textembedding-gecko@001")

embeddings = embedding_model.get_embeddings(["What is life?"])

for embedding in embeddings:
    vector = embedding.values
    print(f"Length = {len(vector)}")
    print(vector)
```

Embeddings and Pandas DataFrames If your text is stored in a column of a DataFrame, you can create a new column with the embeddings with the example below.

[ ]:
```
text = [
    "i really enjoyed the movie last night",
```

```
    "so many amazing cinematic scenes yesterday",
    "had a great time writing my Python scripts a few days ago",
    "huge sense of relief when my .py script finally ran without error",
    "O Romeo, Romeo, wherefore art thou Romeo?",
]

df = pd.DataFrame(text, columns=["text"])
df
```

Create a new column, `embeddings`, using the apply function in pandas with the embeddings model.

[ ]:
```
df["embeddings"] = [
    emb.values for emb in embedding_model.get_embeddings(df.text.values)
]
df
```

Comparing similarity of text examples using cosine similarity By converting text into embeddings, you can compute similarity scores. There are many ways to compute similarity scores, and one common technique is using cosine similarity.

In the example from above, two of the sentences in the `text` column relate to enjoying a *movie*, and the other two relates to enjoying *coding*. Cosine similarity scores should be higher (closer to 1.0) when doing pairwise comparisons between semantically-related sentences, and scores should be lower between semantically-different sentences.

The DataFrame output below shows the resulting cosine similarity scores between the embeddings:

[ ]:
```
cos_sim_array = cosine_similarity(list(df.embeddings.values))

# display as DataFrame
df = pd.DataFrame(cos_sim_array, index=text, columns=text)
df
```

To make this easier to understand, you can use a heatmap. Naturally, text is most similar when they are identical (score of 1.0). The next highest scores are when sentences are semantically similar. The lowest scores are when sentences are quite different in meaning.

[ ]:
```
ax = sns.heatmap(df, annot=True, cmap="crest")
ax.xaxis.tick_top()
ax.set_xticklabels(text, rotation=90)
```

1.7 Code generation with code-bison@001

The code generation model (Codey) from PaLM API that you will use in this notebook is code-bison@001. It is fine-tuned to follow natural language instructions to generate required code and is suitable for a variety of coding tasks, such as:

- writing functions
- writing classes
- web-apges

- unit tests
- docstrings
- code translations, and many more use-cases.

Currently it supports the following languages: - C++ - C# - Go - GoogleSQL - Java - JavaScript - Kotlin - PHP - Python - Ruby - Rust - Scala - Swift - TypeScript

You can find our more details here.

1.7.1 Load model

[ ]:
```
code_generation_model = CodeGenerationModel.from_pretrained("code-bison@001")
```

1.7.2 Model parameters for `code-bison@001`

You can customize how the PaLM API code generation behaves in response to your prompt by using the following parameters for `code-bison@001`:

- `prefix`: it represents the beginning of a piece of meaningful programming code or a natural language prompt that describes code to be generated.
- `temperature`: higher means more "creative" code responses. range: (0.0 - 1.0, default 0).
- `max_output_tokens`: sets the max number of tokens in the output. range: (1 - 2048, default 2048)

1.7.3 Hello Codey

[ ]:
```
prefix = "write a python function to do binary search"

response = code_generation_model.predict(prefix=prefix)

print(response.text)
```

1.7.4 Try out your own prompt

Some examples: * write Go program to extract ip addresses from the text file * write Java program that can extract pincodes from addresses * write a standard SQL function that strips all non-alphabet characters from the string and encodes it to utf-8

[ ]:
```
prefix = """write a python function named as "calculate_cosine_similairty" and␣
 ↪three unit \
           tests where it takes two arguments "vector1" and "vector2". \
           It then uses numpy dot function to calculate the dot product of the␣
 ↪two vectors. \n
         """

response = code_generation_model.predict(prefix=prefix, max_output_tokens=1024)

print(response.text)
```

1.7.5 Prompt templates

Prompt templates are useful if you have found a good way to structure your prompt that you can re-use. This can be also be helpful in limiting the open-endedness of freeform prompts. There are many ways to implement prompt templates, and below is just one example using f-strings. This way you can structure the prompts as per the expected funcationality of the code.

[ ]:
```
language = "C++ function"
file_format = "json"
extract_info = "names"
requirments = """
              - the name should be start with capital letters.
              - There should be no duplicate names in the final list.
              """

prefix = f"""Create a {language} to parse {file_format} and extract␣
 ↪{extract_info} with the following requirements: {requirments}.
              """

response = code_generation_model.predict(prefix=prefix, max_output_tokens=1024)

print(response.text)
```

1.8 Code completion with `code-gecko@001`

Code completion uses the code-gecko foundation model to generate and complete code based on code being written. `code-gecko` completes code that was recently typed by a user.

To learn more about creating prompts for code completion, see Create prompts for code completion.

Code completion API has few more parameters than code generation.

- prefix: *required* : For code models, prefix represents the beginning of a piece of meaningful programming code or a natural language prompt that describes code to be generated.
- suffix: *optional* : For code completion, suffix represents the end of a piece of meaningful programming code. The model attempts to fill in the code in between the prefix and suffix.
- temperature: *required* : Temperature controls the degree of randomness in token selection. Same as for other models. range: (0.0 - 1.0, default 0)
- maxOutputTokens: *required* : Maximum number of tokens that can be generated in the response. **range: (1 - 64, default 64)**
- stopSequences: *optional* : Specifies a list of strings that tells the model to stop generating text if one of the strings is encountered in the response. The strings are case-sensitive.

[ ]:
```
code_completion_model = CodeGenerationModel.from_pretrained("code-gecko@001")
```

[ ]:
```
prefix = """
          def find_x_in_string(string_s, x):
         """
```

```
response = code_completion_model.predict(prefix=prefix,
                                         max_output_tokens=64)

print(response.text)
```

[ ]:
```
prefix = """
         def reverse_string(s):
            return s[::-1]
         def test_empty_input_string()
         """

response = code_completion_model.predict(prefix=prefix,
                                         max_output_tokens=64)

print(response.text)
```

1.9 Code chat with `codechat-bison@001`

The `codechat-bison@001` model lets you have a freeform conversation across multiple turns from a code context. The application tracks what was previously said in the conversation. As such, if you expect to use conversations in your application for code generation, use the `codechat-bison@001` model because it has been fine-tuned for multi-turn conversation use cases.

[ ]:
```
code_chat_model = CodeChatModel.from_pretrained("codechat-bison@001")

code_chat = code_chat_model.start_chat()

print(code_chat.send_message(
        "Please help write a function to calculate the min of two numbers",
    )
)
```

As shown below, the model should respond based on what was previously asked in the conversation:

[ ]:
```
print(code_chat.send_message(
        "can you explain the code line by line in bullets?",
    )
)
```

You can take another example and ask the model to give more general code suggestion for a specific problem that you are working on.

[ ]:
```
code_chat = code_chat_model.start_chat()

print(code_chat.send_message(
        "what is the most scalable way to traverse a list in python?",
    )
)
```

You can continue to ask follow-up questions to the origianl query.

[ ]:
```
print(code_chat.send_message(
        "how would i measure the iteration per second for the following code?",

    )
)
```