6.4 - Generate code with Azure OpenAI Service

Overview

This module shows engineers how to use the Azure OpenAI Service to generate and improve code.

By the end of this module, you'll be able to:

- Use natural language prompts to write code
- Build unit tests and understand complex code with AI models
- Generate comments and documentation for existing code

Introduction

The Azure OpenAI service enables you to use language models to generate content, including programming code. This functionality allows developers to generate and improve existing programming

code in various languages to increase efficiency and understanding.

In this module, you'll learn how to use Azure OpenAI to generate code and help various development tasks.

Construct code from natural language

One of the capabilities of Azure OpenAI models is to generate code from natural language prompts. Tasks can range from a simple one line command to a full application. The AI models can also edit and update provided code or previous responses to complete the requested task.

AI models for code generation

In previous generations of `gpt` models, some were trained specifically for use with code (often called *codex* models). As new generations of models evolve, the base models drastically improve their performance and understanding of both language and code, which results in not needing specific code models. This improvement results in just a single model for more recent generations (such as `gpt-35-turbo` and `gpt-4`) that can be used for both natural language and code.

The examples in this module are using `gpt-35-turbo`.

Write functions

Azure OpenAI models can create functions and apps in several languages by just describing what you want. For example, say you need an implementation of binary search but can't remember how it's done. Given the prompt `write a function for binary search in python`, you likely receive a response with the function and an explanation of the code.

```
Sure, here's an example implementation of binary search algorithm in Python:

def binary_search(array, target):
    low = 0
    high = len(array) - 1

    while low <= high:
        mid = (low + high) // 2
        if array[mid] == target:
            return mid
        elif array[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1

This function takes in two arguments - an array of integers and a target integer that we
want to search for in the array. The function initializes two pointers - low and high -
to the first and last indices of the array, respectively.
```

```
Inside the while loop, the function calculates the middle index using integer division
(//). If the middle element is equal to the target, the function returns the index of
the middle element. If the middle element is less than the target, we update the low
pointer to mid + 1. Otherwise, we update the high pointer to mid - 1. We continue these
steps until we find the target or the pointers cross each other (i.e., low > high).

If the target is not found in the array, the function returns -1.

Note that the binary search algorithm assumes that the input array is sorted in
ascending order. If the array is not sorted, the algorithm may not return the correct
result.
```

The request can go further than functions, and request a series of functions or an entire app. The response provides a good starting point for the task you're trying to achieve.

Change coding language

If you have code in one language, but need it in another, Azure OpenAI can translate that for you. For example, say you have this `print_squares` function in Python.

```python
def print_squares(n):
    for i in range(1, n+1):
        print(i**2)
```

If you provide that code, along with the prompt `convert this code to C#`, you likely get something similar to the following function

```
public void PrintSquares(int n)
{
    for (int i = 1; i <= n; i++)
    {
        Console.WriteLine(i * i);
    }
}
```

Notice the model understood the differences between how `for` loops and functions are defined in the different languages (including access modifiers in C#), and changed the code appropriately.

Understand unknown code

Azure OpenAI models are helpful for understanding code that doesn't make sense, or may be in a language you aren't familiar with. For example, say you were given the following function (in a fictitious coding language!) and didn't know how to understand it.

```
fn qzplv(n: i32, m: i32) -> i32 {
    if n == 0 {
        return m + 1;
    } else if m == 0 {
        return qzplv(n - 1, 1);
    } else {
```

```
        return qzplv(n - 1, qzplv(n, m - 1));
    }
}
```

When provided to the model, along with the prompt `could you explain what this code is doing?`, you get a response similar to the following:

```
Sure, I'd be happy to explain what this code is doing!

This code is implementing the Ackermann function, which is a well-known mathematical
function that grows very quickly in terms of computation time as its inputs increase.
The function takes two integer arguments (n and m) and returns another integer.

Here's how the function works:
- If n is equal to 0, the function returns m + 1.
- If m is equal to 0, the function calls itself recursively with n - 1 and 1 as
arguments.
- Otherwise, the function calls itself recursively with n - 1 and the result of calling
itself with n and m - 1 as arguments.

The function is implemented in a way that is easy to understand and follow, but it has a
very high computational complexity. In fact, it grows so quickly that it can't be
computed for values of n and m greater than about 4 or 5.
```

Combining the explanation and translation into a language you're familiar with makes this functionality even more useful.

Complete code and assist the development process

Azure OpenAI can assist developers in common software development tasks such as writing unit tests, completing partial code, commenting code, and generating documentation. Using AI assistance can enable developers to spend more time on complex programming and problem solving tasks.

Complete partial code

Azure OpenAI models are capable of generating code based on comments, function names, and partially written code. The more context you can provide the model, the more accurate the response likely is.

For example, when given the following prompt:

```
complete the following function
"""
# calculate the average of the numbers in an array, but only if they're even
def
```

The model takes the comments and start of the function definition, and complete it from whatever context it has.

```
# calculate the average of the even numbers in an array
def calculate_even_average(numbers):
    total = 0
    count = 0
    for number in numbers:
        if number % 2 == 0:
            total += number
            count += 1
    if count == 0:
        return 0  # avoid division by zero
    else:
        return total / count
```

In this case, it was able to discern what we wanted due to a relatively simple and complete comment preceding the start of the function. With more complex functions or tasks, more context is required to get a useful response.

It's important to note that for some languages, like Python, the syntax is clear enough for the model to assume what language is desired. For syntactically similar languages like C# or Java, there's a higher likelihood that the model uses the wrong language. If that is the case, try specifying the language in the prompt or providing more code for context.

Similar functionality can be seen by providing partial code to the model, without any comments. Take the following prompt for example.

```
complete the following code
"""
def func1(n)
  if n==0:
```

The AI model does its best to complete the function with the most likely completion of the function.

```
def func1(n):
    if n == 0:
        return 1
    else:
        return n * func1(n-1)
```

The response likely also includes a natural language explanation of what that function is doing. In this case, it responded with a recursive implementation of the factorial function.

However, the prompt could have also been the start of a Fibonacci function, which illustrates a case where we need more context in the prompt to better indicate what we want from the model.

Tools such as GitHub Copilot utilize OpenAI models to suggest code and function completions in a similar way to the above examples. GitHub Copilot is an excellent example of real-world usage of this functionality that uses built-in prompts and the context of the code you're writing to act as your paired programmer.

Write unit tests

Azure OpenAI models can generate unit tests for functions you write to help make your code more robust. Take for example the binary search function.

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Provide that function along with the prompt `write three unit tests for this function`, and you get a response similar to the following.

```
def test_binary_search():
    # Test case 1: Target element is in the middle of the array
    arr1 = [1, 2, 3, 4, 5, 6]
    target1 = 4
    assert binary_search(arr1, target1) == 3

    # Test case 2: Target element is at the beginning of the array
    arr2 = [1, 2, 3, 4, 5, 6]
    target2 = 1
    assert binary_search(arr2, target2) == 0

    # Test case 3: Target element is not in the array
    arr3 = [1, 2, 3, 4, 5, 6]
    target3 = 7
    assert binary_search(arr3, target3) == -1
```

This functionality is even more useful if you specify the cases you want included in your prompt, which reduces the time it takes for you to write unit tests for your code.

Add comments and generate documentation

To further improve your code, AI models can add comments and documentation for you. Take the following function as an example, which can be a little hard to understand when first reading it without any code comments.

```
def permutations(lst):
    if len(lst) == 0:
        return []
    elif len(lst) == 1:
        return [lst]
    else:
        result = []
        for i in range(len(lst)):
            temp = lst[i]
            remaining = lst[:i] + lst[i+1:]
            for p in permutations(remaining):
                result.append([temp] + p)
        return result
```

Provide that function to the model, along with a prompt requesting that it add comments to the code, and you get a response similar to the following.

```
def permutations(lst):
    # Check if the input list is empty
    if len(lst) == 0:
        return []
    # Check if the input list has only one element
    elif len(lst) == 1:
        return [lst]
    else:
        # Initialize an empty list to store permutations
        result = []
        # Iterate through each element of the input list
        for i in range(len(lst)):
            # Remove the current element from the list and store it in a variable
            temp = lst[i]
            remaining = lst[:i] + lst[i+1:]
            # Recursively call the function on the remaining list
            for p in permutations(remaining):
                # Append the current element to the beginning of each permutation and
add it to the result list
                result.append([temp] + p)
        # Return the final list of permutations
        return result
```

Take it one step further, and request documentation for the function it just added comments to.

```
def permutations(lst):
    """
    Generates all possible permutations of a given list of integers using recursion.

    Args:
        lst (list): The list of integers to generate permutations for.

    Returns:
        list: A list of all possible permutations of the input list.
    """
    # Check if the input list is empty
    if len(lst) == 0:
        return []
    # Check if the input list has only one element
    elif len(lst) == 1:
        return [lst]
    else:
        # Initialize an empty list to store permutations
        result = []
        # Iterate through each element of the input list
        for i in range(len(lst)):
            # Remove the current element from the list and store it in a variable
            temp = lst[i]
            remaining = lst[:i] + lst[i+1:]
            # Recursively call the function on the remaining list
            for p in permutations(remaining):
                # Append the current element to the beginning of each permutation and
```

```
        add it to the result list
                    result.append([temp] + p)
        # Return the final list of permutations
        return result
```

Fix bugs and improve your code

Developers sometimes can write code that mostly works, but could be improved by fixing bugs, syntax, performance, or modularity. Azure OpenAI models can help identify ways to improve and provide suggestions on how to write better code.

Fix bugs in your code

Azure OpenAI models can help fix bugs in code by analyzing the code and suggesting changes that can potentially fix the issue. This can help developers identify and resolve bugs faster and more efficiently.

For example, say you have the following function that isn't working for you.

```
def calculate_average(numbers):
    total = 0
    for i in range(len(numbers)):
        number = numbers[i]
        total += number
    average = total
    rerun averave
```

Provide that function to the model, along with the prompt `Fix the bugs in this function`, and you get a response with the fixed code and an explanation of what was fixed.

```
def calculate_average(numbers):
    total = 0
    for i in range(len(numbers)):
        number = numbers[i]
        total += number
    average = total / len(numbers)
    return average
```

```
There were a few issues with the original code:

1. The variable "average" was assigned the value of "total" instead of the actual
average.
2. There was a typo in the last line - "rerun" should be "return".
3. The average was not actually calculated - the corrected code divides the total by the
length of the "numbers" list to get the average.
```

Improve performance

While code developers write may work, there might be a more efficient way to accomplish the task. Here's an example of a function that calculates the sum of the first n positive integers, possibly inefficiently:

```
def sum_of_n(n):
    result = 0
    for i in range(1, n+1):
        result += i
    return result
```

This version works correctly, but its time complexity is O(n). When provided to the model, here's the response:

```
We can improve this function it by using a mathematical formula instead.
Here's the more efficient version:
```

```
def sum_of_n(n):
    return n * (n + 1) // 2
```

This version still returns the correct result, but its time complexity is now O(1), which makes it much more efficient.

Refactor inefficient code

Better code is less prone to bugs and is easier to maintain, and the Azure OpenAI models can help guide developers on how to refactor their code.

Consider the following function.

```
def calculateTotalPrice(item, quantity):
    if item == 'apple':
        return quantity * 0.5
    elif item == 'banana':
        return quantity * 0.75
    elif item == 'orange':
        return quantity * 0.6
    else:
        return 0
```

This code calculates the total price of a particular item based on its name and quantity. However, the code isn't modular and can be difficult to maintain. When provided to the model with the request to refactor it, here's the response:

```
item_prices = {
    'apple': 0.5,
    'banana': 0.75,
    'orange': 0.6
}

def calculate_total_price(item, quantity):
    price = item_prices.get(item, 0)
    return quantity * price
```

Along with the code, the model provides an explanation of the refactoring.

```
In this refactored code, we've created an object called itemPrices that maps item names to their prices. We've also simplified the calculateTotalPrice function by using the itemPrices object to look up the price of the item, rather than using a series of if/else statements. This makes the code more modular and easier to maintain, and also reduces the risk of introducing bugs in the future.
```

Exercise - Generate and improve code with Azure OpenAI Service

The Azure OpenAI Service models can generate code for you using natural language prompts, fixing bugs in completed code, and providing code comments. These models can also explain and simplify existing code to help you understand what it does and how to improve it.

In scenario for this exercise, you will perform the role of a software developer exploring how to use generative AI to make coding tasks easier and more efficient. The techniques used in the exercise can be applied to other code files, programming languages, and use cases.

This exercise will take approximately **25** minutes.

Provision an Azure OpenAI resource

If you don't already have one, provision an Azure OpenAI resource in your Azure subscription.

1. Sign into the **Azure portal** at `https://portal.azure.com`.
2. Create an **Azure OpenAI** resource with the following settings:
   - **Subscription**: *Select an Azure subscription that has been approved for access to the Azure OpenAI service*
   - **Resource group**: *Choose or create a resource group*
   - **Region**: *Make a **random** choice from any of the following regions**
     - Australia East
     - Canada East
     - East US
     - East US 2
     - France Central
     - Japan East
     - North Central US
     - Sweden Central
     - Switzerland North
     - UK South
   - **Name**: *A unique name of your choice*
   - **Pricing tier**: Standard S0

> - Azure OpenAI resources are constrained by regional quotas. The listed regions include default quota for the model type(s) used in this exercise. Randomly choosing a region reduces the risk of a single region reaching its quota limit in scenarios where you are sharing a subscription with other users. In the event of a quota limit being reached later in the exercise, there's a possibility

> you may need to create another resource in a different region.need to create another resource in a different region.

3. Wait for deployment to complete. Then go to the deployed Azure OpenAI resource in the Azure portal.

Deploy a model

Azure provides a web-based portal named **Azure AI Foundry portal**, that you can use to deploy, manage, and explore models. You'll start your exploration of Azure OpenAI by using Azure AI Foundry portal to deploy a model.

> **Note**: As you use Azure AI Foundry portal, message boxes suggesting tasks for you to perform may be displayed. You can close these and follow the steps in this exercise.

1. In the Azure portal, on the **Overview** page for your Azure OpenAI resource, scroll down to the **Get Started** section and select the button to go to **AI Foundry portal** (previously AI Studio).
2. In Azure AI Foundry portal, in the pane on the left, select the **Deployments** page and view your existing model deployments. If you don't already have one, create a new deployment of the **gpt-35-turbo-16k** model with the following settings:
   - **Deployment name**: *A unique name of your choice*
   - **Model**: gpt-35-turbo-16k *(if the 16k model isn't available, choose gpt-35-turbo)*
   - **Model version**: *Use default version*
   - **Deployment type**: Standard
   - **Tokens per minute rate limit**: 5K*
   - **Content filter**: Default
   - **Enable dynamic quota**: Disabled

> - A rate limit of 5,000 tokens per minute is more than adequate to complete this exercise while leaving capacity for other people using the same subscription.

Generate code in chat playground

Before using in your app, examine how Azure OpenAI can generate and explain code in the chat playground.

1. In the **Playground** section, select the **Chat** page. The **Chat** playground page consists of a row of buttons and two main panels (which may be arranged right-to-left horizontally, or top-to-bottom vertically depending on your screen resolution):
   - **Configuration** - used to select your deployment, define system message, and set parameters for interacting with your deployment.
   - **Chat session** - used to submit chat messages and view responses.
2. Under **Deployments**, ensure that your model deployment is selected.
3. In the **System message** area, set the system message to `You are a programming assistant helping write code` and apply the changes.
4. In the **Chat session**, submit the following query:

```
Write a function in python that takes a character and a string as input, and
returns how many times the character appears in the string
```

The model will likely respond with a function, with some explanation of what the function does and how to call it.

5. Next, send the prompt `Do the same thing, but this time write it in C#`.
   The model likely responded very similarly as the first time, but this time coding in C#. You can ask it again for a different language of your choice, or a function to complete a different task such as reversing the input string.
6. Next, let's explore using AI to understand code. Submit the following prompt as the user message.

```
What does the following function do?
---
def multiply(a, b):
    result = 0
    negative = False
    if a < 0 and b > 0:
        a = -a
        negative = True
    elif a > 0 and b < 0:
        b = -b
        negative = True
    elif a < 0 and b < 0:
        a = -a
        b = -b
    while b > 0:
        result += a
        b -= 1
    if negative:
        return -result
    else:
        return result
```

   The model should describe what the function does, which is to multiply two numbers together by using a loop.
7. Submit the prompt `Can you simplify the function?`. The model should write a simpler version of the function.
8. Submit the prompt: `Add some comments to the function.` The model adds comments to the code.

Prepare to develop an app in Visual Studio Code

Now let's explore how you could build a custom app that uses Azure OpenAI service to generate code. You'll develop your app using Visual Studio Code. The code files for your app have been provided in a GitHub repo.

> **Tip**: If you have already cloned the **mslearn-openai** repo, open it in Visual Studio code. Otherwise, follow these steps to clone it to your development environment.

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLearning/mslearn-openai` repository to a local folder (it doesn't matter which folder).

3. When the repository has been cloned, open the folder in Visual Studio Code.

> **Note**: If Visual Studio Code shows you a pop-up message to prompt you to trust the code you are opening, click on **Yes, I trust the authors** option in the pop-up.

4. Wait while additional files are installed to support the C# code projects in the repo.

> **Note**: If you are prompted to add required assets to build and debug, select **Not Now**.

Configure your application

Applications for both C# and Python have been provided, as well as a sample text file you'll use to test the summarization. Both apps feature the same functionality. First, you'll complete some key parts of the application to enable using your Azure OpenAI resource.

1. In Visual Studio Code, in the **Explorer** pane, browse to the **Labfiles/04-code-generation** folder and expand the **CSharp** or **Python** folder depending on your language preference. Each folder contains the language-specific files for an app into which you're you're going to integrate Azure OpenAI functionality.
2. Right-click the **CSharp** or **Python** folder containing your code files and open an integrated terminal. Then install the Azure OpenAI SDK package by running the appropriate command for your language preference:

   C#:

   ```
   dotnet add package Azure.AI.OpenAI --version 1.0.0-beta.14
   ```

   Python:

   ```
   pip install openai==1.13.3
   ```

3. In the **Explorer** pane, in the **CSharp** or **Python** folder, open the configuration file for your preferred language
   - **C#**: appsettings.json
   - **Python**: .env
4. Update the configuration values to include:
   - The **endpoint** and a **key** from the Azure OpenAI resource you created (available on the **Keys and Endpoint** page for your Azure OpenAI resource in the Azure portal)
   - The **deployment name** you specified for your model deployment (available in the **Deployments** page in Azure AI Foundry portal).
5. Save the configuration file.

Add code to use your Azure OpenAI service model

Now you're ready to use the Azure OpenAI SDK to consume your deployed model.

1. In the **Explorer** pane, in the **CSharp** or **Python** folder, open the code file for your preferred language. In the function that calls the Azure OpenAI model, under the comment ***Format and send the request to the model***, add the code to format and send the request to the model.

C#: Program.cs

```
// Format and send the request to the model var chatCompletionsOptions = new ChatCompletionsOptions() { Messages = { new ChatRequestSystemMessage(systemPrompt), new ChatRequestUserMessage(userPrompt) }, Temperature = 0.7f, MaxTokens = 1000, DeploymentName = oaiDeploymentName }; // Get response from Azure OpenAI Response<ChatCompletions> response = await client.GetChatCompletionsAsync(chatCompletionsOptions); ChatCompletions completions = response.Value; string completion = completions.Choices[0].Message.Content;
```

Python: code-generation.py

```python
# Format and send the request to the model
messages = [
    {"role": "system", "content": system_message},
    {"role": "user", "content": user_message},
]

# Call the Azure OpenAI model
response = client.chat.completions.create(
    model=model,
    messages=messages,
    temperature=0.7,
    max_tokens=1000
)
```

2. Save the changes to the code file.

Run the application

Now that your app has been configured, run it to try generating code for each use case. The use case is numbered in the app, and can be run in any order.

> **Note**: Some users may experience rate limiting if calling the model too frequently. If you hit an error about a token rate limit, wait for a minute then try again.

1. In the **Explorer** pane, expand the **Labfiles/04-code-generation/sample-code** folder and review the function and the *go-fish* app for your language. These files will be used for the tasks in the app.
2. In the interactive terminal pane, ensure the folder context is the folder for your preferred language. Then enter the following command to run the application.
   - **C#**: `dotnet run`
   - **Python**: `python code-generation.py`

   > **Tip**: You can use the **Maximize panel size** (**^**) icon in the terminal toolbar to see more of the console text.

3. Choose option **1** to add comments to your code and enter the following prompt. Note, the response might take a few seconds for each of these tasks.

   ```
   Add comments to the following function. Return only the commented code.\n---\n
   ```

   The results will be put into **result/app.txt**. Open that file up, and compare it to the function file in **sample-code**.

4. Next, choose option **2** to write unit tests for that same function and enter the following prompt.
   `Write four unit tests for the following function.\n---\n`

   The results will replace what was in **result/app.txt**, and details four unit tests for that function.
5. Next, choose option **3** to fix bugs in an app for playing Go Fish. Enter the following prompt.
   `Fix the code below for an app to play Go Fish with the user. Return only the corrected code.\n---\n`

   The results will replace what was in **result/app.txt**, and should have very similar code with a few things corrected.

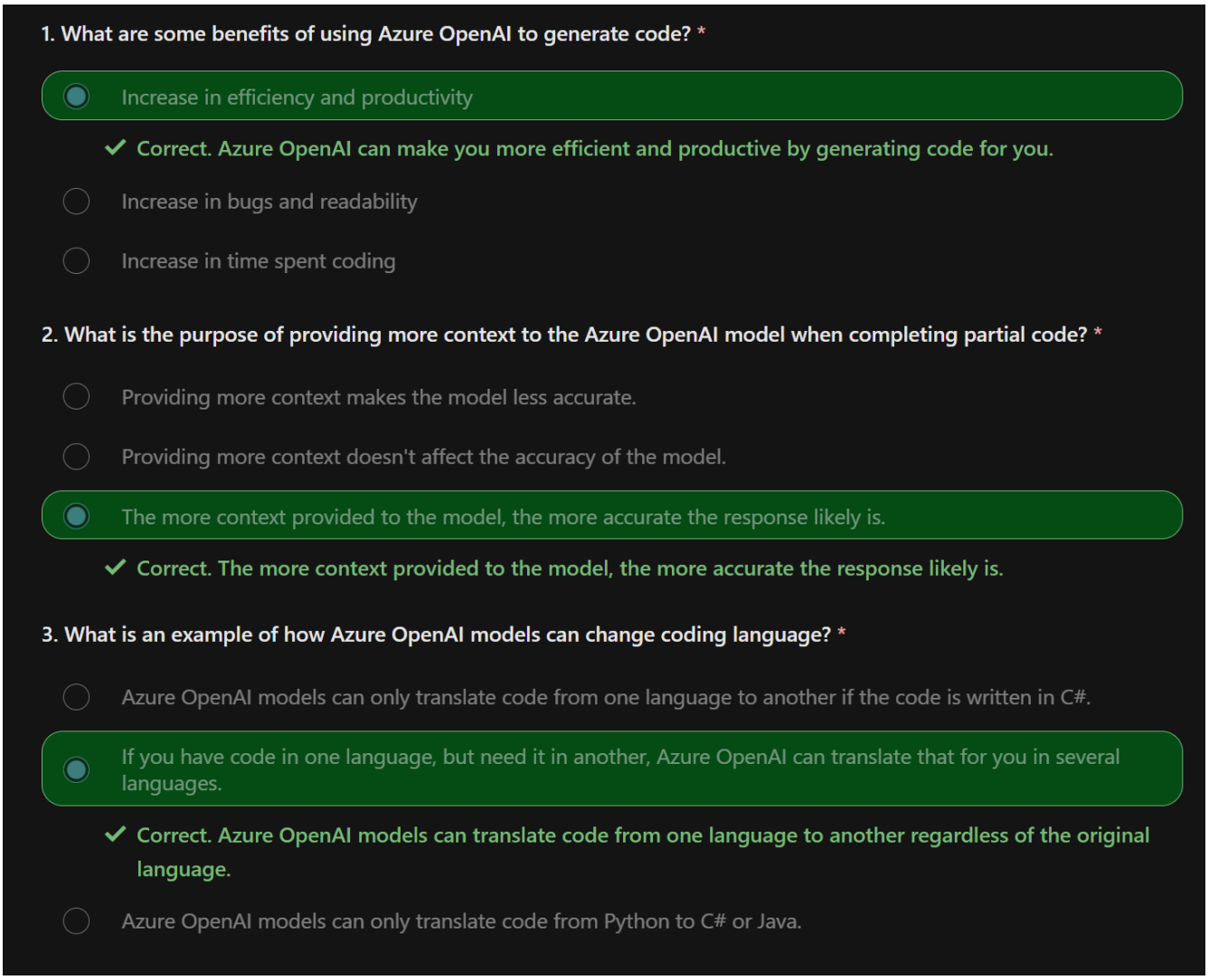
   - **C#**: Fixes are made on line 30 and 59
   - **Python**: Fixes are made on line 18 and 31

   The app for Go Fish in **sample-code** can be run if you replace the lines that contain bugs with the response from Azure OpenAI. If you run it without the fixes, it will not work correctly.

   > **Note**: It's important to note that even though the code for this Go Fish app was corrected for some syntax, it's not a strictly accurate representation of the game. If you look closely, there are issues with not checking if the deck is empty when drawing cards, not removing pairs from the players hand when they get a pair, and a few other bugs that require understanding of card games to realize. This is a great example of how useful generative AI models can be to assist with code generation, but can't be trusted as correct and need to be verified by the developer.

   If you would like to see the full response from Azure OpenAI, you can set the **printFullResponse** variable to `True`, and rerun the app.

Knowledge Check



Summary

The Azure OpenAI models can generate code from natural language prompts and existing code, ranging from a simple one-line command to a full application. Azure OpenAI models can translate code from one language to another, help developers understand unknown code, and assist developers in common software development tasks such as writing unit tests, completing partial code, commenting code, and generating documentation.

In this module, you learned how to:

- Use natural language prompts to write code
- Build unit tests and understand complex code with AI models
- Generate comments and documentation for existing code

Additional Reading:

- Azure OpenAI Service documentation
- Azure OpenAI models
- GitHub Copilot
- How to use Azure OpenAI to work with code

✍️ Compiled by Kenneth Leung (2025)