2025 Summer Coding Camp

Google API and Pygame Basics

Introduction

Welcome to the 2025 Summer Coding Camp! We're super excited to work with you this year and have some really fun projects planned! To start off we're going to be learning about how we can use common API keys like Google API in order to access various models. This is just one example of APIs and models we can use, but the knowledge is often transferable! We will also explore some lower-level features of Pygame and programming data structures.

Length: 2 - 3 hours
Concepts: generative models, Google API, keyboard input, if-statements, for-loops, two-dimensional arrays / lists, classes

Project Setup

There are few things we need to do before we get started with our project. First, we suggest you download the PyCharm editor for this project, as that is what we use in this lab. There are many other options, however, so choose whatever you find easiest to work with, if you prefer! Just know that we can only help debug PyCharm-specific editor problems, not other editors.

If you are a student, many colleges and universities provide the full version of PyCharm for free, but the Community Edition is completely fine, also! In this project we will also be using Google Colab to get our API key and use our generative model, since it's the easiest way to access it. It doesn't take too long to set up and we'll be showing you how, so no need to worry there!

Setup Instructions: Link
Project Repository: Link

So what's the goal?

Basically, this is a warm-up project, where we learn how to use APIs, with a game development element to it to help us become more familiar with Pygame. Overall, the actual AI stuff is really easy! We'll mostly spend our time programming the game.

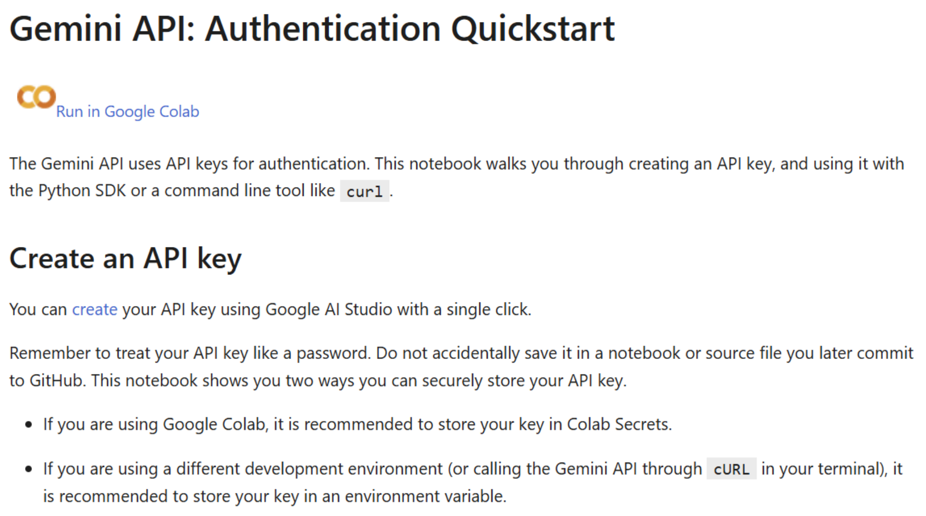
Basically, our goal is to learn to generate and use API keys in Google Colab, we'll generate some fun images of characters using the Gemini generative model, and then use those in a character select screen! The most important knowledge here is what API keys are and how we can access them, how to use them once we have them, and the basics of Pygame!

Starting with Gemini

First, let's start with using Gemini to create our little character images. You will need a Gmail account in order to access Google API and Google Colab, so make sure to create an account if you somehow don't already have one!

Then, we will continue with the following steps:

1. Log into our Gmail account, if needed!
2. We want to get an API key so we can use Gemini in our Python code.



Gemini API: Authentication Quickstart

Run in Google Colab

The Gemini API uses API keys for authentication. This notebook walks you through creating an API key, and using it with the Python SDK or a command line tool like `curl`.

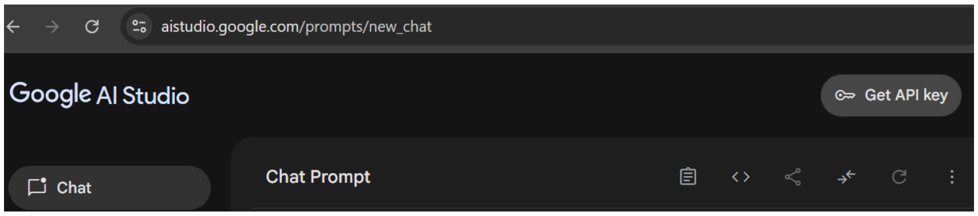
Create an API key

You can create your API key using Google AI Studio with a single click.

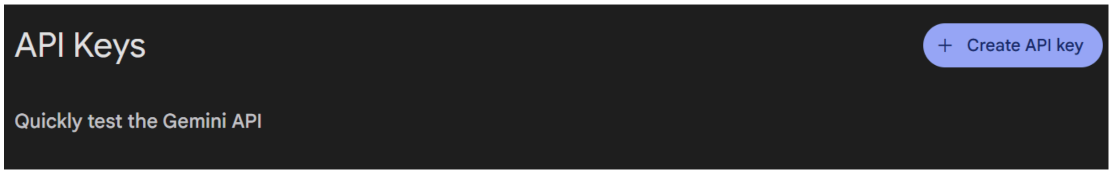
Remember to treat your API key like a password. Do not accidentally save it in a notebook or source file you later commit to GitHub. This notebook shows you two ways you can securely store your API key.

- If you are using Google Colab, it is recommended to store your key in Colab Secrets.
- If you are using a different development environment (or calling the Gemini API through `cURL` in your terminal), it is recommended to store your key in an environment variable.

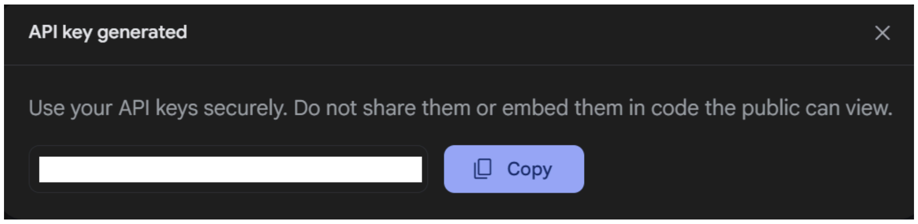
3. To do this, we will go to the Google AI Studio website.
4. Once here, we should see an option that says **Get API Key**, click it!



5. Then click **Create API Key**.

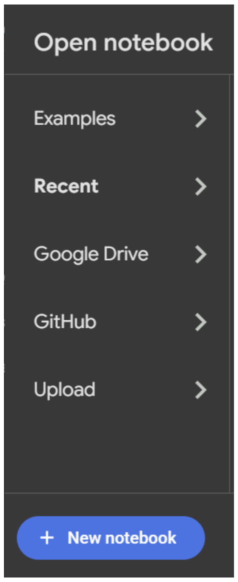


6. At this time, you should have your API key generated for you and displayed in a dialog box. We hid our key, and **it is important to keep your key protected and treat it as a password**. It is unique only to you, so don't lose it and don't send it to anyone!

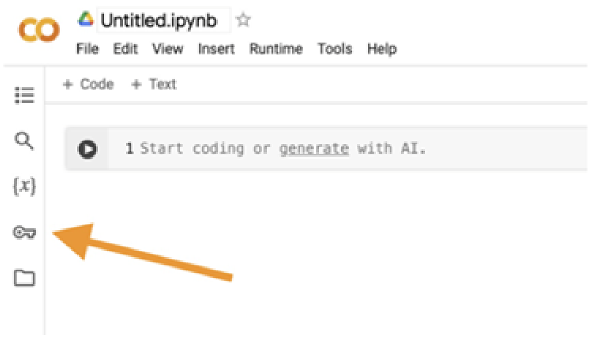

API key generated

Use your API keys securely. Do not share them or embed them in code the public can view.

7. Click **Copy** and now go to Google Colab!
8. Open a new notebook by clicking on **+ New notebook**. We can rename this to something like "Character Select Generation" or anything else you want. What matters is

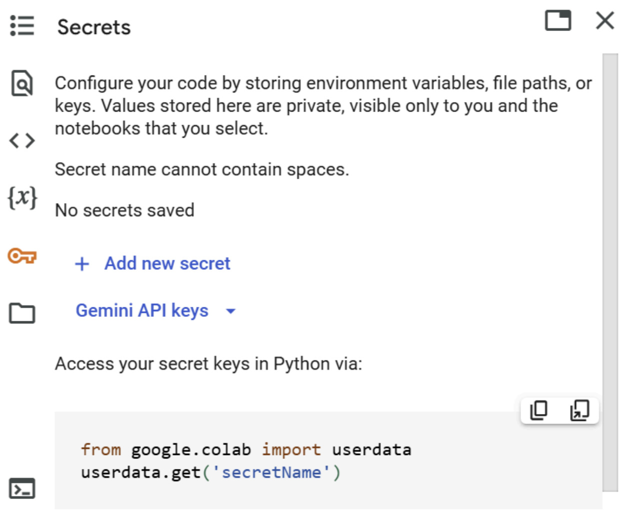


9. Next, we need to save our API key as a "secret" so we can use it in our Python code without displaying it. To do this, click on the button with the **key** icon. This is our Secrets button!



```
1 Start coding or generate with AI.
```

10. Then, we will click **+ Add new secret**!



Configure your code by storing environment variables, file paths, or keys. Values stored here are private, visible only to you and the notebooks that you select.

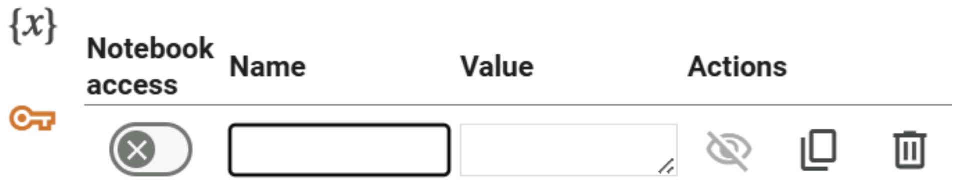
Secret name cannot contain spaces.

No secrets saved

Access your secret keys in Python via:

```
from google.colab import userdata
userdata.get('secretName')
```

11. Paste the **API key** under **Value**. Then, enter **GEMINI_API_KEY** under **Name**. Finally, click the toggle button for **Notebook access**. Now we can use the API key in our code easily!



Notebook access	Name	Value	Actions

12. Now we can access the API key by creating a variable and accessing the user data in order to get the secret!

```
from google.colab import userdata
# Fetch API key.
gemini_api_key = userdata.get("GEMINI_API_KEY")
```

13. To test that our API key works with Gemini, we can ask any simple question that you might ask it usually! For example, something about video games. You can ask it anything you want, we just want to make sure it works. :]

```python
from google import genai
from google.colab import userdata
# Fetch API key.
gemini_api_key = userdata.get("GEMINI_API_KEY")
client = genai.Client(api_key = gemini_api_key)
# Send a request to the Gemini model.
response = client.models.generate_content(
      model = "gemini-2.0-flash",
      contents = ["What are the 10 most popular video games?"]
)
# Output the results!
print(response.text)
```

14. Alternatively, we can also test if the generative model can access links like it should! This is really useful if you ever start digging into large-language models (LLMs) in the future!

```python
from google import genai
from google.colab import userdata
from google.genai import types
# Fetch API key.
gemini_api_key = userdata.get("GEMINI_API_KEY")
client = genai.Client(api_key = gemini_api_key)
# Send a request to the Gemini model.
response = client.models.generate_content(
      model = "gemini-2.0-flash",
      contents = types.Content(
            parts = [
                  types.Part(text = 'Can you summarize this video?
Then create a quiz with answer key based on the information in the
video'),
                  types.Part(file_data = types.FileData(file_uri =
'https://www.youtube.com/watch?v=MFhxShGxHWc'))
            ]
      )
)
# Output the results!
print(response.text)
```

15. Lastly, we'll generate a test image using Gemini. This is just an example prompt, so do whatever you want! Once again, we just want to ensure the model is working as intended and get a feel for how it works.

```
from google import genai
from google.colab import userdata
from PIL import Image
from google.genai import types
from io import BytesIO
# Fetch API key.
gemini_api_key = userdata.get("GEMINI_API_KEY")
client = genai.Client(api_key = gemini_api_key)
# Create a prompt for your image! Have fun with it. :]
contents = ('Can you draw a 3d rendered image of a cat '
            'with wings and a top hat flying over a happy '
            'futuristic sci-fi city with lots of greenery?')
# Send a request to the Gemini model.
response = client.models.generate_content(
      model = "gemini-2.0-flash-exp-image-generation",
      contents = contents,
      config = types.GenerateContentConfig(response_modalities =
['Text', 'Image'])
)
# Then, we will get the responses.
for part in response.candidates[0].content.parts:
      # Sometimes, Gemini will respond with some text that we may
      # want to read!
      if part.text is not None:
            print(part.text)
      # If there's inline data, that means it's image data!
      elif part.inline_data is not None:
            image = Image.open(BytesIO((part.inline_data.data)))
            # Save the image to a file so we can look at it.
            image.save('gemini-native-image.png')
```

Generating Our Character Images

Now that we have a basic understanding of how to generate and save an image using Gemini and API keys, we have our next step: we have to actually make our images we'll use in our character selection demo!

Here are the steps we need to take:

1. Create four prompts for our character images. Generally we want to include whether they're half-body or headshot portraits, the art-style you're looking for, and so on. Sometimes this takes some tweaking to get what you're looking for. Put these prompts into a list.
2. For each prompt, generate and save the image. Print out the text response if you want to see it too!
3. Check and see if we like the images. Repeat steps 1 and 2 as needed until you're satisfied with the result! Just keep in mind that on the free API key plan, we can make up to 500 requests a day, so there is *technically* a limit. However, it's pretty unlikely that you'll hit it!
4. Download the images from Google Colab, and then we can move on to making our actual little demo project!!

Getting the Project Files

The files needed for this project are included in the project repository, which can be found here.

Task 1: Complete the `Screen` Class

The first thing we need to do is create an enumerator for the current screen we're on. This includes the start screen and the character select screen. Of course, we could create individual variables for each of the screen states, but that can take up a lot of space and be hard to find as you expand your game! Instead, we'll use the Enum class, and create our own enumerator for the different screens of the game!

Complete the screen class by creating values for `START` and `SELECT`. These can have any values, but we suggest using 0 and 1.

Task 2: Initialize the GameManager Class

The GameManager class is the main portion of our project! Think of it as a controller for all the upper level stuff of our game, such as changing screens and running the main loop. In a larger project, it might control the pausing of a game, saving / loading, and changing scenes! In our project though, we'll use it to control the game loop (whether the game is running or not), changing between screens, and drawing the screens. We will start by writing the __init__ method, which is called when the GameManager is created!

1. Declare a boolean variable that represents whether the game is running or not. When we start the program, should this be True or False?
2. Initialize the current_screen of the game to be a default value. Make sure to use the Screen class we created in Task 1! What default value is appropriate for current_screen?
3. Initialize the character_image to be the image of our first character, using Pygame's pygame.image.load method. What constants can we use to make it so we don't have to write out the name of the image manually? What number is the first of our images?
    a. Then, scale character image using:
    ```
    character_image = pygame.transform.scale(character_image,
        (WIDTH / 2, HEIGHT / 2))
    ```
4. After we'v
5. e initialized our variables, we can start the main game loop. In most games this is done with a while-loop. Create a while-loop that checks if the game is running. If it is, call the GameManager's update method!
6. At the end of the __init__ function, tell Pygame to quit the game. If the while-loop terminates, it means we should exit the program!
7. Lastly, we'll create our GameManager object in the main code of our file.

Task 3: Create the Start Screen

Reviewing the `update` Method

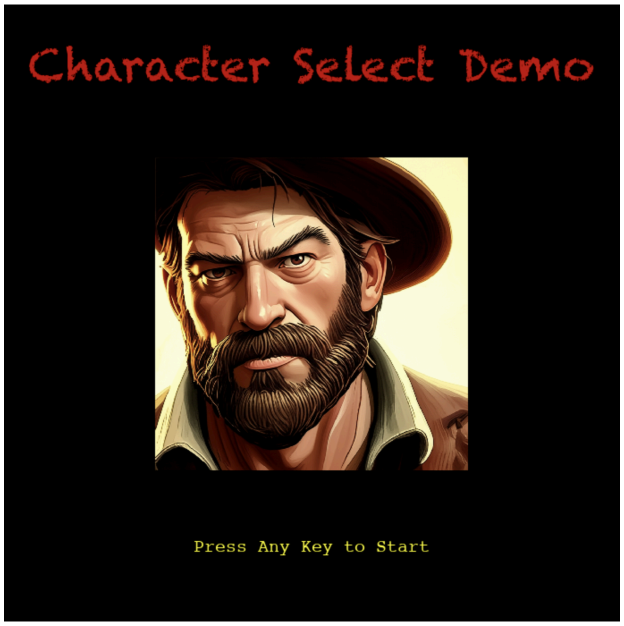
Before we get into drawing our screens, we need to discuss how this method is actually being called. Remember that in our `__init__` method, we have a while-loop that continues running as long as `running` is set to `True`. In that loop we call the `update` method, which is essential to how our game is able to run every frame, calling the `draw` and `get_input` methods! This method is also where we update the clock, allowing our program to run consistently at 60 frames per second.

```python
def update(self):
    """Updates the program each frame."""

    # Get the user's input and draw the screen.
    self.get_input()
    self.draw()
    # Runs at 60 FPS.
    self.clock.tick(60)
```

Complete the `draw` Method

Now, we can begin writing the code to draw the start screen! The `draw` method is where we'll write anything that has to do with rendering. We will use it to make our start screen look something like the image below (with your own images, of course):



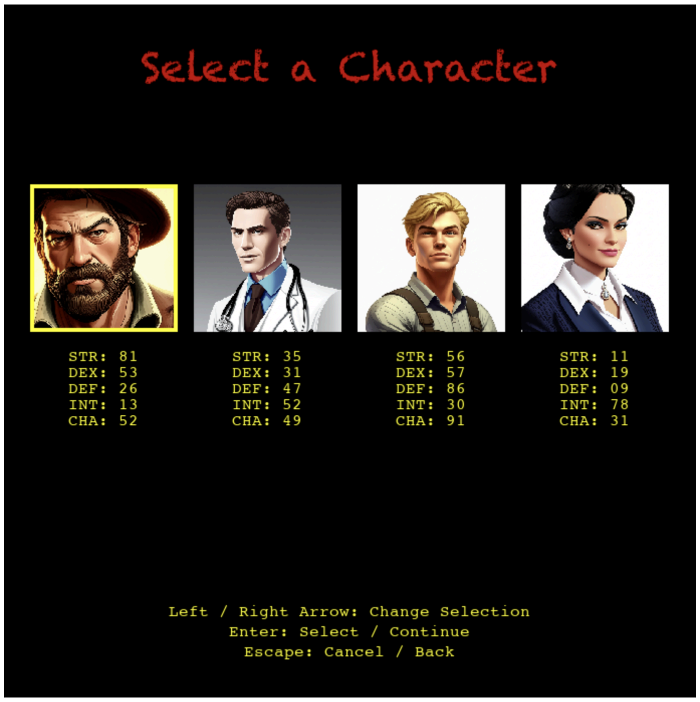
What information is used here? The text is always the same, but the image we want to display is stored in `character_image`. We use this variable because, later, we will make that start screen display a random character each time we visit it! For now, let's consider what we have to do to display this screen:

1. First, check that the current screen is the start screen. If it is, do the following steps.
2. Fill the screen with black to begin with, or whatever background color you want on the start screen!
3. Draw the title of the start screen:
    a. Create the font of the title text using the `pygame.font.SysFont` class. This can be any system font on your machine (eg. `pygame.font.SysFont("chalkduster", 50)`). Assign it a reasonable size for the title.
    b. Create a variable to hold the actual text of the title. This way if you want to change it later on in an easy to find place, you can!
    c. Render the title text using the `render` method of the `SysFont` object you created earlier. It takes three parameters, the text you want to render, a boolean representing if you want antialiasing, and the color of the font!
    d. Create a variable to hold the position of the title text, which we'll use to display it. It should be displayed at the top center of the screen! It should be written as `(x, y)`, a tuple of floats.
    e. Finally, use the self.screen object and call the `screen.blit` method, passing the rendered title text and position as parameters.
4. Draw the prompt of the start screen, following the same steps used to draw the title, just with the prompt's information instead! Make sure to use new variables for this, not the title's variables!
5. Draw the `character_image` on the screen:
    a. Create a variable to hold the position of the image, which should be centered on the screen on both axes.
    b. Use the `screen.blit` method to draw the `character_image` on the screen, similar to how you did with the rendered text.
6. Make sure the `pygame.display.flip` method is called at the end of your draw method, not inside any if-statement you created for this task!

When you play your game, you should see something similar to the example above! Tweak it as much as you like before continuing. You can change the fonts, colors, and placement however you see fit!

Task 4: "Loading" the Character Data

We can now see the start screen, which is great, but we want to show a character select screen that displays the images and stats of each of our characters. Later we'll even make it so we can move a selection box around on the screen. The goal is to have the character select screen look something similar to this:


Select a Character

STR: 81	STR: 35	STR: 56	STR: 11
DEX: 53	DEX: 31	DEX: 57	DEX: 19
DEF: 26	DEF: 47	DEF: 86	DEF: 09
INT: 13	INT: 52	INT: 30	INT: 78
CHA: 52	CHA: 49	CHA: 91	CHA: 31

Left / Right Arrow: Change Selection
Enter: Select / Continue
Escape: Cancel / Back

In a real project, we would probably declare the data of each character using a spreadsheet, saved as JSON or a CSV so it could be loaded up into the game and be adjusted as needed. However, in our case, we'll be using random values for each character's stats instead, just to simulate that experience but not have to write all the stats manually or the code needed to load up the data from the file!

We will be using the global variable `character_data` to store the characters' images and each of their stats, so it can be referenced from anywhere in our code if needed. `character_data` will be a two-dimensional list, meaning each value in it is a row representing the data of one character. For example, the first row of the `character_data` list based on the image above would be the following:

```
["gemini-character-0.png", 81, 53, 26, 13, 52]
```

and the entire list would be:

```
[["gemini-character-0.png", 81, 53, 26, 13, 52],
 ["gemini-character-1.png", 35, 31, 47, 52, 49],
 ["gemini-character-2.png", 56, 57, 86, 30, 91],
 ["gemini-character-3.png", 11, 19,  9, 78, 31]]
```

In our GameManager class, we will create a method called load_data, which will store this generated information into character_data. This method can be completed with the following steps:

1. Create a for-loop that starts at 0 and continues for the number of characters we have.
2. Create a variable representing the row that is being added to character_data. It should initially be an empty list.
3. Create a variable to hold the loaded image. Reference the code you wrote earlier in Task 2.3. The filepath of the image should be the following:

   ```
   IMAGE_NAME + [the number of the image you want] + IMAGE_EXT
   ```

   What value should replace the brackets? What variable keeps track of the number of the character we're currently loading data for?

4. Scale the image as you did in Task 2.3.
5. Append the newly loaded image to the row variable you created in Step 2 of this task.
6. Now, you need to randomly generate the stats. You can do this easily with a for-loop. For each stat, generate a random number between 1 - 100 and append it to the row.
7. Once you are done creating the row, append the row to character_data.

This will continue for each of the characters, initializing the character_data variable assuming your code is correct. In the __init__ method, call the load_data method and try printing character_data and make sure all four of your characters and their data have been added to the list. If any are missing, odds are you made a mistake in Step 1 of this task when declaring your for-loop!

Task 5: Write the set_screen Method

As mentioned in Task 3.5, we want to make the character_image random whenever we enter the start screen, to show off all of our characters. Instead of writing this every single time we want to change the screen, which becomes quite often when we start adding inputs that allow us to switch between the screens, we will be writing a method to change the screen and handle randomly setting the character_image variable when we need to.

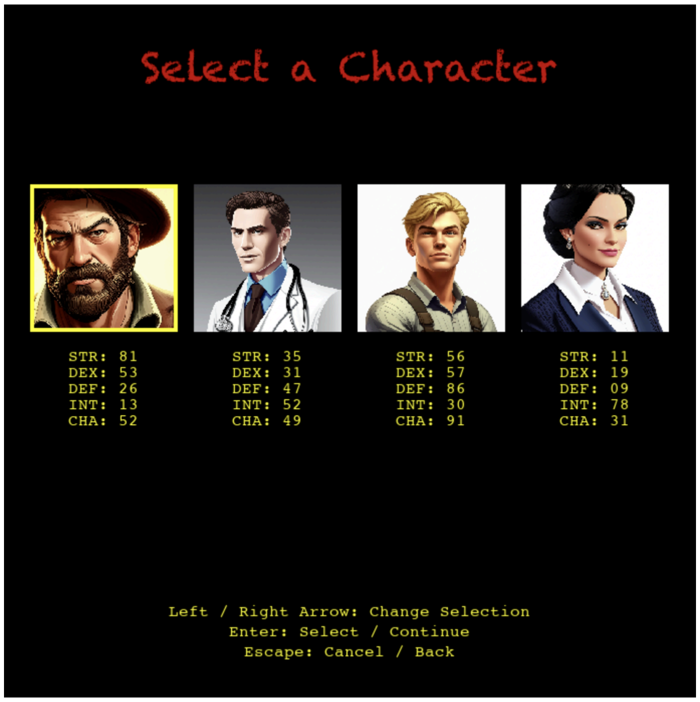
Write the set_screen method with the following steps:

1. Set the GameManager's current_screen to the new_screen, make sure to use the self keyword!
2. Check if we are changing to the start screen, if we are, do the next two steps.

3. Get a random index of one of the character images. What variable represents the number of characters that we have?
4. Set `character_image` to the picture of the random characters index. You will need to use the `character_data` 2D list for this. Keep in mind that a character's image is at index 0 of their data list!
5. Then, in the `__init__` method, add a call to `set_screen` after you initialize the `current_screen` variable.

Task 6: Create the Character Select Screen

Now we have the character data loaded and our set_screen method written, we can begin to make the character select screen. Start by changing the set_screen call in `__init__` to set the screen to `SELECT` instead of `START`, so we can test this code as we implement it. As shown before, we want to display all the character images and their stats.



```
Select a Character

STR: 81          STR: 35          STR: 56          STR: 11
DEX: 53          DEX: 31          DEX: 57          DEX: 19
DEF: 26          DEF: 47          DEF: 86          DEF: 09
INT: 13          INT: 52          INT: 30          INT: 78
CHA: 52          CHA: 49          CHA: 91          CHA: 31

          Left / Right Arrow: Change Selection
               Enter: Select / Continue
                 Escape: Cancel / Back
```

We also want to show what character is currently being "hovered" over and be able to move back and forth with the arrow keys, but that will be implemented in a later task. For now, we're just going to display the fixed texts, the characters, and their stats. We will be writing this in the draw method, using an elif-statement to check if we're on the character select screen. Here's what we'll need to do to display this screen:

1. First, check that the current screen is the character select screen. If it is, do the following steps.

2. Fill the screen with black to begin with, or whatever background color you want on the character select screen!
3. Loop through each character using an incremented for-loop, and do the following:

Task 6.1: Determine the Size of the Image

We need to get the size of each image so that they can fit on the screen with the `MARGIN` on either side of the images and `PADDING` between each of them. The algorithm to do this would be the following:

1. If we are placing the images starting at the left of the edge of the screen with no margin or padding between the images, the size of each image would be the width of the screen divided by the number of images.

```
WIDTH: 400 pixels, NUM_IMAGES: 4
WIDTH / NUM_IMAGES = 400 / 4 = 100 Pixels per Image
```

2. If we are to account for the margin on the left and right sides of the screen, we need to subtract that from the width of the screen before dividing.

```
WIDTH: 400 pixels, MARGIN: 20 Pixels
WIDTH - (MARGIN * 2) = 400 - 40 = 360 Pixels
```

3. Then we need to account for the padding between each image, which also needs to be subtracted from the width of the screen before dividing as well! Since there is only padding between consecutive images, we will only apply it to the number of images minus one.

```
WIDTH: 400 Pixels, MARGIN: 20 Pixels, PADDING: 10 Pixels,
NUM_IMAGES: 4
WIDTH - (MARGIN * 2) - (PADDING * (NUM_IMAGES - 1)) = 400 - (20
* 2) - (10 * (4 - 1)) = 400 - 40 - 30 = 330 Pixels
```

4. Now that we have the adjusted width, we can divide that by the number of images to get the size of each image!

```
ADJUSTED_WIDTH: 330 Pixels, NUM_IMAGES: 4
ADJUSTED_WIDTH / NUM_IMAGES = 360 / 4 = 82.5 Pixels (Round
Down) = 82 Pixels
```

5. Finally, we will scale the image down to the determined size using the `pygame.scale.transform` method!

Task 6.2: Determine the Position of the Image

Next, we need to determine the position of the image, which can be calculated by using the size of the image, the margin of the screen, and the padding between each image, as well as the number of the current image we are positioning (our for-loop is useful for this!). An important fact to remember while positioning is that images are positioned by their top-left corner, not their center!

Since we've already scaled the images so that they will fit perfectly between the margins and with padding between them, positioning the images is actually quite easy at this point! The first image will be positioned at the margin of the screen. Every image after that will be moved to the right by the width of the image plus the padding!

```
IMAGE_SIZE: 82 Pixels, MARGIN: 20 Pixels, PADDING: 10 Pixels
X Position = MARGIN + (image_index * (IMAGE_SIZE + PADDING)) = 20 +
(image_index * (82 + 10)) = 20 + (image_index * 92)
```

Then, we can draw the image on the screen using the `screen.blit` method, passing the scaled image and its determined position as the parameters!

Task 6.3: Drawing the Stats Below Each Image

Now, still in our for-loop, we need to draw the stats of the character under the current image. You could do this manually, but we'll be doing it with a nested for-loop in this document. Feel free to use whichever approach you find most appropriate as long as the results are comparable!

1. Start by declaring a list of strings with the abbreviated names of each stat. We suggest using abbreviations that have the same number of letters, so that all of the stats line up evenly whenever you draw them!

   ```
   ["STR", "DEX", "DEF", "INT", "CHA"]
   ```

2. Create a font for the stats using the `pygame.font.SysFont` class as we have done before. We suggest using a monospaced font, once again so all the stats line up evenly when they're drawn on the screen! In the official solution, we use the Consolas font with a size of 16.

3. Determine the starting Y position of the first stat label, which will be the Y position of the image which you determined earlier, the size of the image, and the padding added together. The padding is added in order to put a slight gap between the bottom of the image and the top of the first stat. Save this value in a variable for later.

```
Start Y Position = IMAGE_POSITION_Y + IMAGE_SIZE + PADDING
```

4. Then, loop through each of the stat names using an incremented for-loop and do the following steps:
    a. The label for each stat will be "`Stat Label: Stat Value`". For example "`STR: 81`" or "`DEF: 09`". To add the padded zeros on values with one digit, we will use the Python `string.zfill` method. How do we get the `Label` and `Value` in the formula below in our code?

    ```
    Text = Label + ": " + str(Value).zfill(2)
    ```

    b. Render this text using the font you created earlier and the color you prefer, and store in a variable.
    c. Determine the position of this stat label:
        i. The Y position will be the starting Y position you determined in the previous step of this task with the padding added for each line of text we're on.

        ```
        Y Position = START_Y + (stat_index * PADDING)
        ```

        ii. The X position will be centered under the current image, which would be the width of the text subtracted from the width of the image, divided by two added to the position of the image.

        ```
        X Position = IMAGE_POSITION_X + ((IMAGE_SIZE -
        TEXT_WIDTH) / 2)
        ```

    d. Use the `screen.blit` method to draw the text on the screen using the position you calculated.

Task 6.4: Drawing the Other Text

The rest of the character select screen is basically steps that we've performed before at this point. We want to display the title text and then the instructions. The title will be done similarly to the title text on the start screen and the instructions can be done in a way similar to what we did with the stats.

1. Display the title text in the same way we displayed the title before for the start screen, declare a text, a font, and a position variable before using the `screen.blit` method.
2. For the instructions at the bottom of the screen, we need a list of strings that contain each line of text.

```
["Left / Right Arrow: Change Selection",
 "Enter: Select / Continue",
 "Escape: Cancel / Back"]
```

3. Similar to displaying the stats, declare a starting Y position (600).
4. Loop through each instruction text with an incremented for-loop and draw each instruction separated by the padding. The X position should be the text centered on the screen.

```
X Position = (WIDTH - TEXT_WIDTH) / 2
```

Make sure everything is how you want it! Feel free to place the images and text manually, or with code to dynamically place them. Experiment with the fonts and colors as much as you'd like as well! When you play the game, you should start out on the character select screen since you changed the `set_screen` call in the `__init__` method, allowing you to quickly modify and see your changes.

Once you're done testing, change the `set_screen` call back to the start screen before continuing to the next task!

Task 7: Switching Between Screens

So, now we have our two individual screens, but no way to go between them. What we need is to take the user's input! There are many ways to design the logic of your input method, this is only one way we've designed it. If you scroll to your `get_input` method, you should see already one input has been completed for you:

```
for event in pygame.event.get():
      if event.type == pygame.QUIT:
            self.running = False
```

This allows us to quit the game when we press the exit button on the window, and is essential! However, now we're going to add a few new inputs:

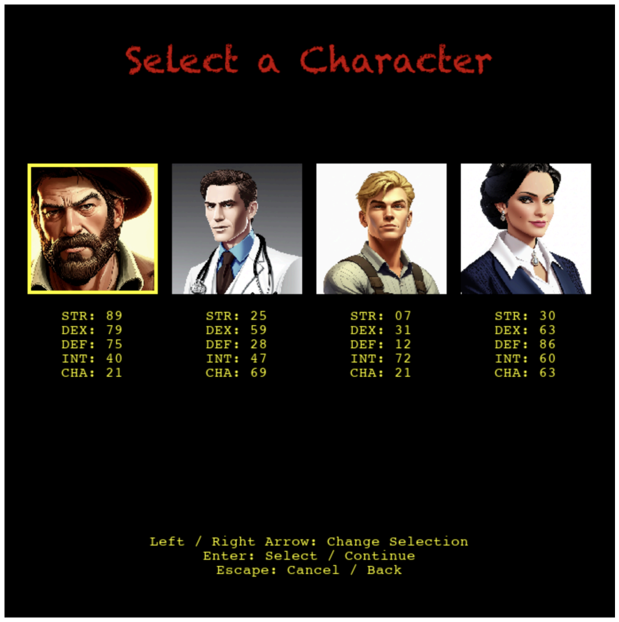
1. If the input is a `pygame.KEYDOWN` event, handle the following situations.
2. If on the start screen:
    a. If we press escape, quit the game.
    b. If we press any other key, change to the character select screen.
3. If on the character select screen:
    a. If we press the escape key, go back to the start screen.

We will add more inputs for our character select screen as we go along adding the ability to select and deselect characters, but for now this is enough to allow us to swap between the start and character select screens!

Task 8: Adding the Selection Box

We need to add the selection box to the character select screen. We'll add the ability to move it around in the next task, but we first have to declare variables and draw the box on the screen according to those variables! Then handling the inputs will be as simple as changing those variables accordingly.

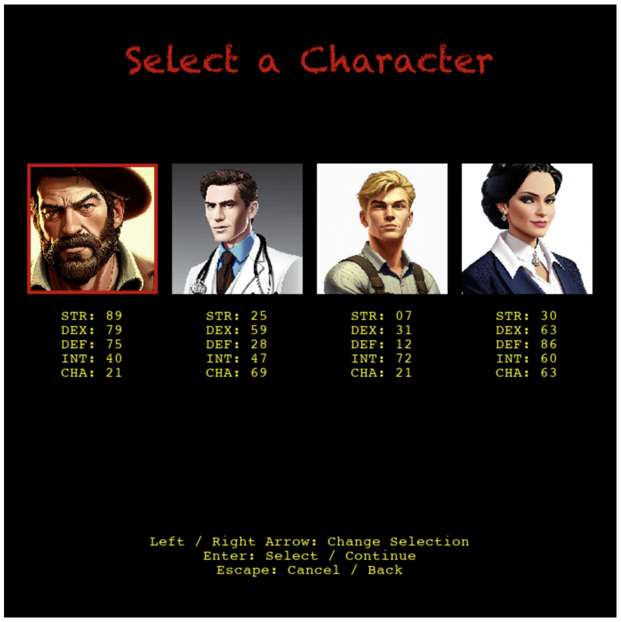
Our selection box has two states, yellow for when a character has not been selected, and red if it has, which you can see below.



Select a Character

```
STR: 89     STR: 25     STR: 07     STR: 30
DEX: 79     DEX: 59     DEX: 31     DEX: 63
DEF: 75     DEF: 28     DEF: 12     DEF: 86
INT: 40     INT: 47     INT: 72     INT: 60
CHA: 21     CHA: 69     CHA: 21     CHA: 63

      Left / Right Arrow: Change Selection
            Enter: Select / Continue
              Escape: Cancel / Back
```



Select a Character

```
STR: 89     STR: 25     STR: 07     STR: 30
DEX: 79     DEX: 59     DEX: 31     DEX: 63
DEF: 75     DEF: 28     DEF: 12     DEF: 86
INT: 40     INT: 47     INT: 72     INT: 60
CHA: 21     CHA: 69     CHA: 21     CHA: 63

      Left / Right Arrow: Change Selection
            Enter: Select / Continue
              Escape: Cancel / Back
```

So what variables do we need? We need one variable that tells us the index of the character we are currently hovering over with the selection box, and another that tells us if the character has been selected (meaning the box should be red).

1. In the `__init__` method, declare a variable that represents the index of the character currently being hovered over with the selection box. By default this should be 0.
2. Declare another variable that represents if the character has been selected. By default it should be `False`.

Now, we need to draw the selection box. Return to the `draw` method and find where you draw the image of each character.

3. If the image index is equal to the selected index variable you created earlier, create a new `rect` using the `pygame.Rect` class. This class has two initializing variables, the position of the rect and the size. Both of the values should match that of the image!

4. If the character is not selected, we want to draw a yellow box. Use the `pygame.draw.rect` method, passing the `screen`, the color `YELLOW`, the rect you created in the last step, and the variable `LINE_WIDTH` as parameters. This tells Pygame you want to draw a yellow box on the screen that is empty with a particular line width.
5. Else (the character IS selected), do the same as the previous step except with the color `RED`.

Now, try changing your selected index variable to another number in the `__init__` method and change your selected flag to `True`. Run your game to verify that the selection box is changing position and color when you change these variables appropriately! Once you're done, return their values to their defaults as described previously.

Task 9: Moving and Selecting Characters

Now we need to be able to move our selection box with our arrow keys. This is pretty simple to do, actually, we'll just modify our `get_input` method on the character select screen to handle additional situations. Make sure to test after each step to make sure your input is working as expected before continuing to the next step!

1. If a character is NOT selected and we press escape, return to the start screen (you'll need to modify your original if-statement to handle the escape key on the character select screen!).
2. If a character is NOT selected and we press enter, set it to True (select a character).
3. If the character IS selected and we press escape, set it back to False (deselect the character).
4. If a character is NOT selected and we press the left arrow key, move the selection box index to the left (by -1). However, do not let it go below 0!
5. If a character is NOT selected and we press the right arrow key, move the selection box index to the right (by +1). Once again, do not let it go above the last character's index!
6. Make sure to not allow the selection box to move if a character is selected!

Summary

And that's it for this portion of the character select project. Feel free to expand this project, adding more screens to select maps or enter the game, allow two players to select characters, or whatever else you can think of!

This base is perfect for expanding upon on your own. To add screens, simply add new values to your enumerator class, sections to your `draw` method, and sections to your `get_input` method! It may be easier to separate these methods into smaller sub-methods, such as `draw_start` or `get_input_select`, as your project become more complex