Eric Roberts and Jed Rembold
CS 151

Handout #17
November 4, 2022

Project #4: Enigma

Due: Friday, November 18, 5:00P.M.

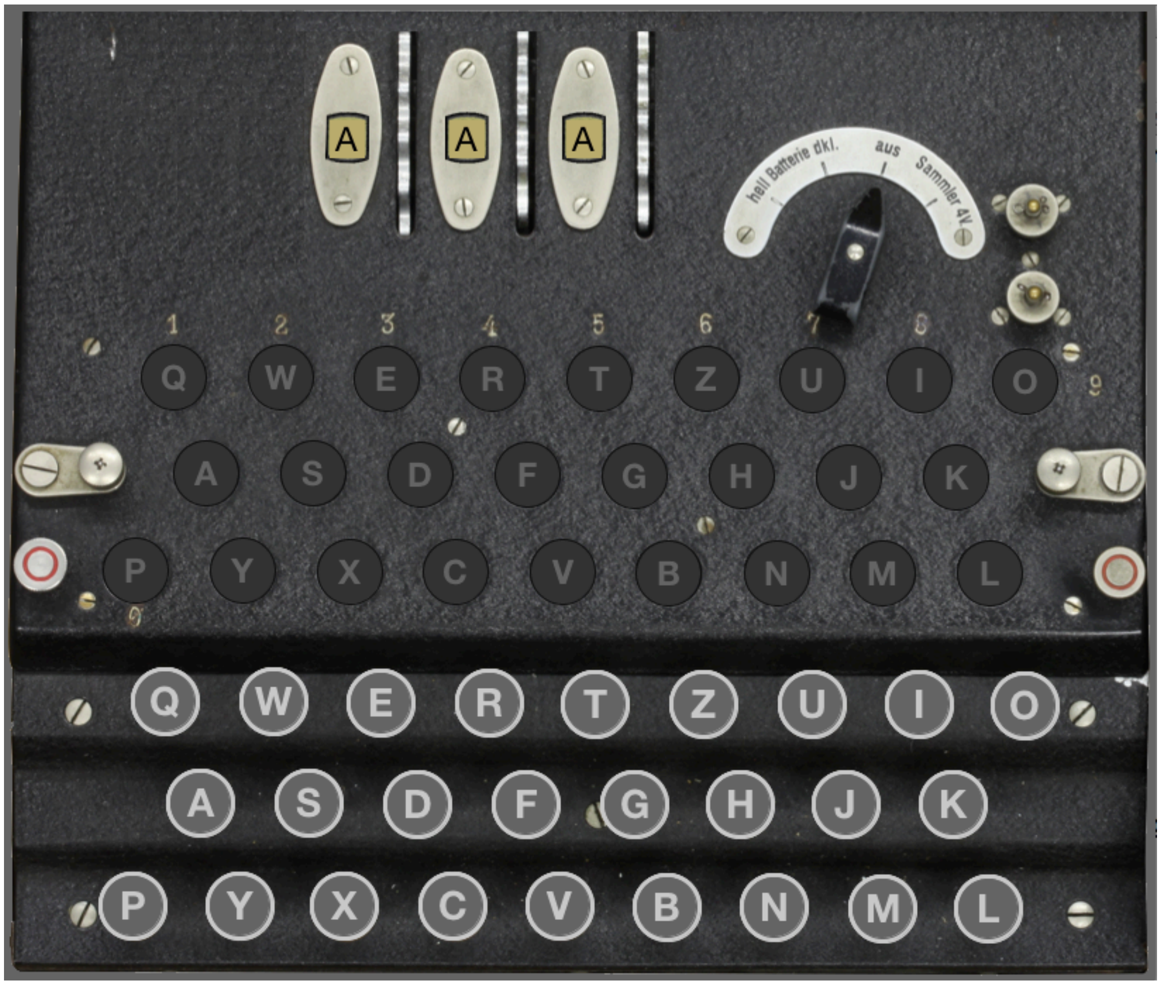
Your task in this assignment is to write a program that simulates the operation of the German Enigma machine from World War II. This handout describes what you have to do to complete the assignment. A companion handout describes the operation of the machine.

The starter files

When you download the **Project4** folder from the web, the folder includes three source files—`EnigmaModel.py`, `EnigmaView.py`, and `EnigmaConstants.py`—along with an `images` folder. When you run the `EnigmaModel.py` program as it appears in the starter project, it produces the screen display shown in Figure 1.

Sadly, even though the screen display shows the entire Enigma machine, that doesn't mean you're finished with the project. The display is simply an image that shows what the Enigma machine looks like when viewed from the top. Your job is to make it interactive.

Figure 1. The output produced by the starter version of Enigma.py



The Model-View-Controller pattern

Before looking in more detail at the starter code for the Enigma project, it is useful to describe the overall programming strategy it uses. As programs have become more sophisticated, software developers have discovered that libraries that define commonly used classes and functions code, while certainly essential to effective programming, are not sufficient in themselves. Software developers also need a collection of strategies that turn out to be useful in solving a variety of problems. Such strategies are called ***design patterns.*** Design patterns were first described in the landmark 1994 book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, commonly referred to as the "Gang of Four."

One of the most widely used design patterns is the ***Model-View-Controller pattern,*** often referred to by its initials as ***MVC.*** This pattern arose from the early work on graphical user interfaces at the Xerox Palo Alto Research Center (Xerox PARC) in the 1970s, where developers created the styles of user interaction that found their way into the Apple Macintosh and that have become ubiquitous today. The key idea behind the MVC pattern is creating a separation between a ***model*** that maintains the state of an application without thinking at all about how it is presented to the user, a ***view*** that uses the model to present a graphical representation of the current state to the user, and a ***controller*** that processes user commands and then updates the model in response. The MVC pattern is illustrated graphically in Figure 2.

In the Enigma project, you are responsible only for model component, which is defined in the file `Enigma.py`, which contains the main program and an incomplete version of the `EnigmaModel` class. The view and controller components are implemented for you in a Python module that exports the `EnigmaView` class, which acts as both view and controller.

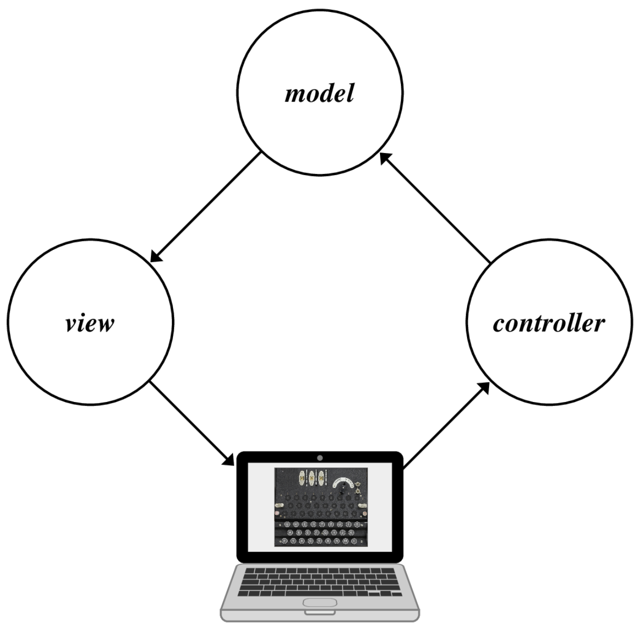
Figure 2. The Model-View-Controller pattern

The ***model*** component keeps track of the internal state but contains no graphics code.

Whenever the state changes, the model sends a message to the ***view*** component asking it to update the image on the user's display.

When the user takes an action such as pressing a key or clicking the mouse, that action is forwarded to the ***controller*** component, which in turn sends a message to the model asking it to update its state.

In many applications, including the Enigma simulator, it makes sense to combine the view and controller in a single Python module.



The `EnigmaModel.py` starter file

The starter code for the `EnigmaModel.py` file appears in Figure 3. The next several sections walk you through the steps you need to complete the implementation.

Figure 3. The supplied starter code for EnigmaModel.py

```python
# File: EnigmaModel.py

"""
This is the starter file for the Enigma project.  This file defines the
EnigmaModel class and the function enigma, which runs the application.
"""

from EnigmaView import EnigmaView

class EnigmaModel:

    def __init__(self):
        """Creates a new EnigmaModel with no views."""
        self._views = [ ]

    def add_view(self, view):
        """Adds a view to this model."""
        self._views.append(view)

    def update(self):
        """Sends an update request to all the views."""
        for view in self._views:
            view.update()

    def key_pressed(self, letter):
        self.update()       # Add code to respond when the key is pressed

    def key_released(self, letter):
        self.update()       # Add code to respond when the key is released

    def is_key_down(self, letter):
        return False        # Add code to return the state of the key

    def is_lamp_on(self, letter):
        return False        # Add code to return the state of the lamp

    def rotor_clicked(self, index):
        self.update()       # Add code to respond when the rotor is clicked

    def get_rotor_letter(self, index):
        return "A"          # In the stub version, all rotors are set to "A"

def enigma():
    """Runs the Enigma simulator."""
    model = EnigmaModel()
    view = EnigmaView(model)
    model.add_view(view)

# Startup code

if __name__ == "__main__":
    enigma()
```

Milestone #1: Activate the keyboard

The first step on your way to implementing an interactive Enigma machine is to make the keyboard operational, which requires creating a data structure to record which keys have been pressed and then completing the definitions of the methods `key_pressed`, `key_released`, and `is_key_down`. The `key_pressed` and `key_released` methods are called by the `EnigmaView.py` module whenever the user presses the mouse down or releases it over one of the keys on the display, passing the letter as an argument. For these operations, the `EnigmaView.py` module is acting as a controller. The `is_key_down` method is called when the `EnigmaView.py` module is operating as a view. As it repaints the display, it asks the model about the state of each key by calling the `is_key_down` method, which returns `True` if the key is down.

As the implementer, you have several choices in terms of how you represent the state of the keys inside the model. Given that the methods in this class take single-character strings as arguments, the most straightforward approach is to store the state in a dictionary, using the letter as the key and a Boolean value, which is `False` if the key is up and `True` if the key is down. If you use this strategy, each of the methods you have to write is just one line long. Alternatively, you could store the key state in a 26-element list, where the index is the position of the letter in the alphabet (**A** = 0, **B** = 1, and so on). A third strategy, which is likely to be overly complex for this aspect of the program but which will come in handy later, is to define a new `EnigmaKey` class that holds the state, and then store those values in a dictionary or a list.

The important point to observe here is that the relationship between the model and the view is independent of the underlying implementation. The `EnigmaView` class is acting as a client of `EnigmaModel`. When it makes a call to a method like `is_key_down`, the view cares about the answer it gets back but is unconcerned about how the `EnigmaModel` class stores that information internally.

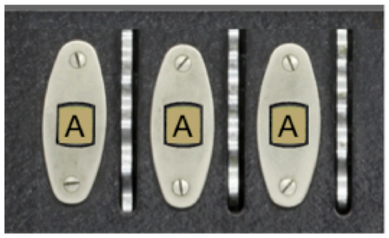
Once you have defined the internal data structure to record whether keys are pressed and completed the implementations of `key_pressed`, `key_released`, and `is_key_down`, you should be able to press the mouse down on the key images and have the letter change color to red. Releasing the mouse button restores the original key color.

Milestone #2: Connect the keys directly to the lamps

The next milestone—which is even easier than the first—is to apply the steps from Milestone #1 to the lamp panel. Once again, you need to set up an internal data structure to record the state of the lamps and change the definition of `is_lamp_on` to record it. The `EnigmaModel` class, however, has no precise counterparts to the `key_pressed` and `key_released` methods because the lamp panel is not directly called by the controller. What you need to do instead is change the definitions of `key_pressed` and `key_released` so that they also change the state of the lamp, even though this change is only for this milestone and will eventually be replaced by code that encrypts the letter before lighting the lamp. If you get this milestone working, you will know that your lamp code is working before moving on to the more challenging problem of implementing the rotors.

Milestone #3: Design and implement the rotor data structure

The encryption rotors for the Enigma machine are positioned in the top center of the window, just above the lamps. The only part of a rotor you can see is the letter that appears in the window at the top of the Enigma image. For example, in the diagram shown in Figure 1, you see the letters



at the top of the window. The letters visible through the windows are printed along the side of the rotor and mounted underneath the panel so that only one letter is visible at a time for each rotor. Whenever a rotor turns one notch, the next letter in the alphabet appears in the window. The act of moving a rotor into its next position is called ***advancing*** the rotor.

For this milestone, your job is to design a data structure that maintains the state of the rotors. There are three rotors in the standard Enigma machine. As described in the companion handout on the Enigma machine, the rotor on the right advances on each keystroke and is called the ***fast rotor.*** Each of the other rotors advances only when the rotor to its right has made a complete revolution. The middle rotor therefore advances only once every 26 keystrokes and is called the ***medium rotor.*** The rotor on the left is the ***slow rotor*** and advances only once in 676 (26×26) keystrokes.

The fact that there are three rotors in a linear arrangement suggests that an appropriate data structure would be a list in which the rotor number serves as an index. But what are the values of each of those list elements? One possible strategy is simply to store the letter that appears in the rotor window, because that is the value that changes as the rotors turn. Although it would be possible to implement the Enigma simulator using that approach, doing so fails to take advantage of the power of object-oriented programming.

The Enigma rotor has several characteristics that suggest that each rotor should be modeled as an object. For one thing, rotors have multiple components in their state. In addition to the letter that appears in the window, each rotor encrypts letters using a different permutation. Moreover, as you will discover in Milestone #5, each rotor in fact implements two permutations, one for when the signal is moving across the rotors from right to left and one for when the signal is moving from left to right. Since these permutations are part of each rotor's state, it makes sense to include them as part of the object. In addition to this state information, rotors exhibit behavior. Rotors *advance.* Using an object to represent a rotor makes it possible to encapsulate that behavior together with the data.

Your job in Milestone #3 is therefore to define an `EnigmaRotor` class that combines the information about a rotor into a single object. Each rotor stores its permutation and an indication of how far it has advanced from its initial position in which the letter **A** appears in the rotor window. This value is easiest to store as an integer, which is called the rotor's ***offset.*** The definition of `EnigmaRotor` therefore needs to export the following methods:

- A constructor that takes the rotor permutation, represented as a 26-character string. The constructor should initialize the rotor offset to 0.
- A `get_offset` method that returns the current value of the offset.
- A `get_permutation` method that returns the permutation string.
- An `advance` method that adds one to the offset, cycling back to 0 when the rotor has made a full revolution. For Milestone #6, you will have to make a small adjustment to this method to handle the process of propagating the advance operation to the next rotor after a complete revolution, but you can ignore that detail for now.

Once you have defined the `EnigmaRotor` class, you need to create the rotors when you initialize the `EnigmaModel` class. The permutations for each of the three rotors—which in fact correspond to rotors in the real World War II Enigma machine—are supplied in the `EnigmaConstants.py` file in the form of the following array:

```
ROTOR_PERMUTATIONS = [
  "EKMFLGDQVZNTOWYHXUSPAIBRCJ",
  "AJDKSIRUXBLHWTMCQGZNPYFVOE",
  "BDFHJLCPRTXVZNYEIWGAKMUSQO"
]
```

The permutation for the slow rotor, for example, is `"EKMFLGDQVZNTOWYHXUSPAIBRCJ"`, which corresponds to the following character mapping:

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
E	**K**	**M**	**F**	**L**	**G**	**D**	**Q**	**V**	**Z**	**N**	**T**	**O**	**W**	**Y**	**H**	**X**	**U**	**S**	**P**	**A**	**I**	**B**	**R**	**C**	**J**

At the moment, you don't have to implement the encryption; you will have a chance to do that starting with Milestone #4. All you need to do is store the permutation for each rotor as part of the Python object that defines the rotor. For example, if you're creating the slow rotor, you have to store the string `"EKMFLGDQVZNTOWYHXUSPAIBRCJ"` as a new attribute in the `EnigmaRotor` object.

To complete this milestone, you need to make two additional changes:

1. Change the definition of `get_rotor_letter` so that it determines the letter in the rotor window by asking the `EnigmaRotor` object for its offset and then translating that offset to one of the 26 letters in the alphabet.
2. Update the definition of `rotor_clicked` so that it calls the advance method of the specified rotor. You don't need to think about implementing the process of carrying to the next rotor position until Milestone #6.

Milestone #4: Implement one stage in the encryption

So far in this assignment, you've set up the initial machine state and implemented the `EnigmaRotor` class, but haven't as yet done any encryption. Since Milestone #2, your program has responded to pressing the letter **Q** by lighting the lamp for the letter **Q**. Your job in Milestone #4 is to implement the first stage in that encryption.

The Enigma machine encrypts a letter by passing it through the rotors, each of which implements a simple letter-substitution cipher. Instead of trying to get the entire encryption path working at once, it makes much more sense to trace the current through just one step of the Enigma encryption. For example, suppose that the rotor setting is **AAA** and the Enigma operator presses the letter **Q**. Current flows from right to left across the fast rotor, as shown in Figure 4. The wiring inside the fast rotor maps that current to the letter **I**, which you can determine immediately from the permutation for the fast rotor, which looks like this:

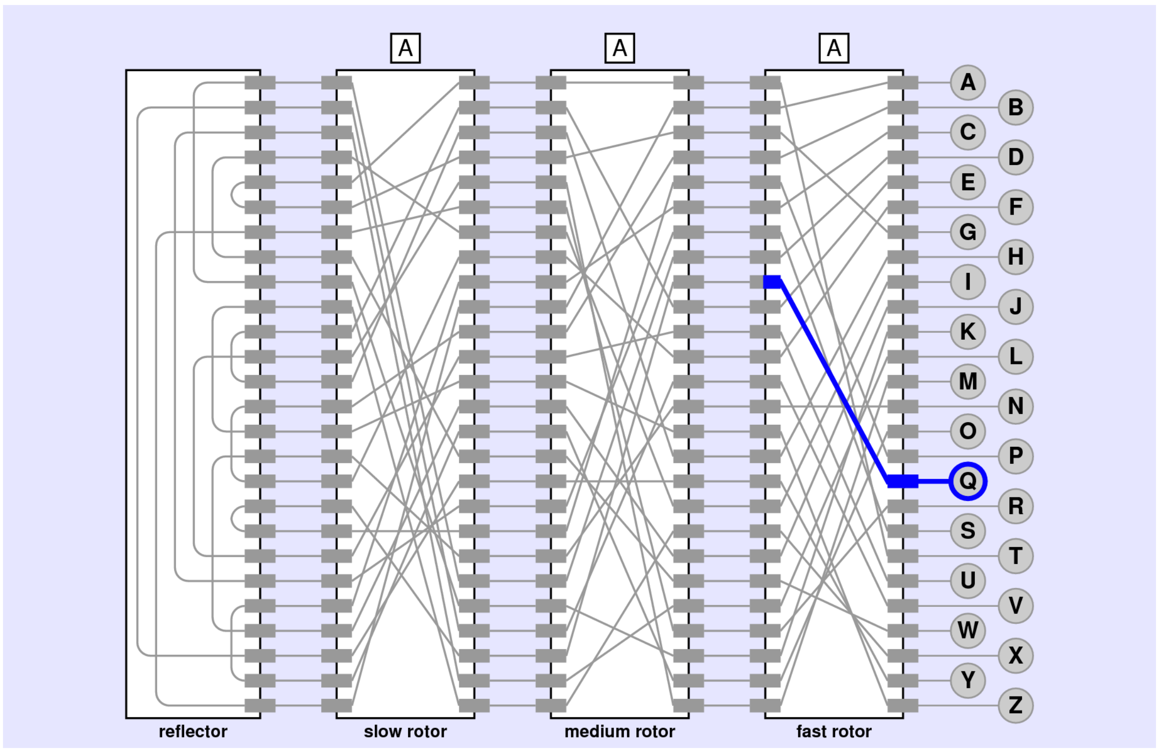
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
B	D	F	H	J	L	C	P	R	T	X	V	Z	N	Y	E	I	W	G	A	K	M	U	S	Q	O

So far, so good. The process of translating a letter, however, is not quite so simple because the rotor may not be in its initial position. What happens if the offset is not 0?

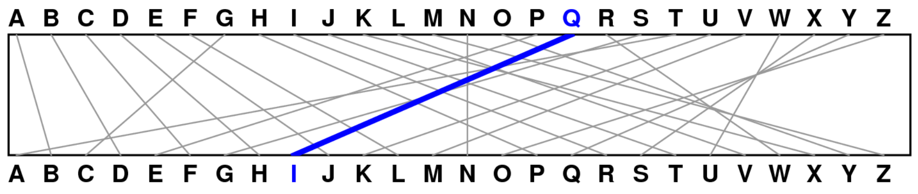
One of the best ways to think about the Enigma encryption is to view it as a combination of a letter-substitution cipher and a Caesar cipher, because the offset rotates the permutation in a cyclical fashion similar to the process that a Caesar cipher uses. If the offset of the fast rotor were 1 instead of 0, the encryption would use the wiring for the next letter in the alphabet. Thus, instead of using the straightforward translation of **Q** to **I**, the program would need somehow to apply the transformation implied by the next position in the permutation, which shows that **R** translates to **W**.

Focusing on the letters, however, is likely to get you confused. The problem is that the positions of the letters on each end of the rotor chain are fixed. After all, the lamps and

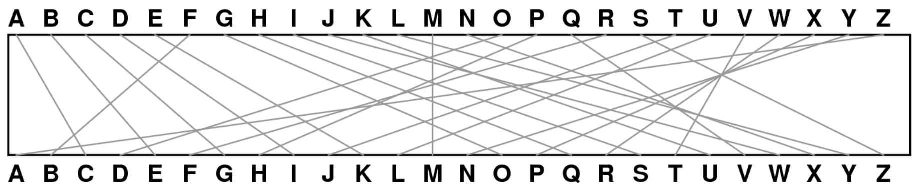
Figure 4. First step on the encryption path when the operator presses Q



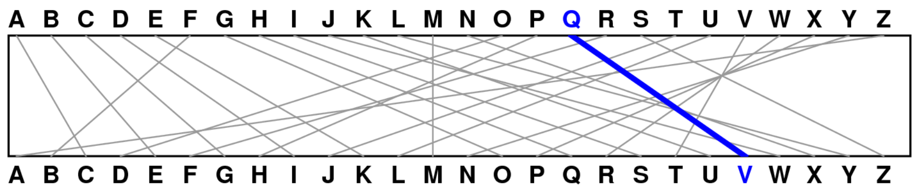
keys don't move. What happens instead is that the rotor connections move underneath the letters. It is therefore more accurate to view the transformation implemented by the fast rotor in the following form:


A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

When the rotor advances, the letters maintain their positions, but every wire rotates one position to the left with respect to the top and bottom connections. After advancing one notch, the rotor looks like this:


A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The change in the wiring is easiest to see in the straight line that connects **N** to **N** in the original state. After advancing, this wire connects **M** to **M**. If you press the **Q** key again in this new configuration, it shows up on the other side of the rotor as **V**:


A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

What's happening in this example is that the translation after advancing the rotor is using the wire that connects **R** to **W** in the initial rotor position. After advancing, that wire connects **Q** and **V**, each of which appears one letter earlier in the alphabet. If the rotor has advanced k times, the translation will use the wire k steps ahead of the letter you're translating. Similarly, the letter in that position of the permutation string shows the letter k steps ahead of the letter you want. You therefore have to add the offset of the rotor to the index of the letter before calculating the permutation and, after doing so, subtract the offset from the index of the character you get. In each case, you need to remember that the process of addition and subtraction may wrap around the end of the alphabet. You therefore need to account for this possibility in the code in much the same way that the Caesar cipher does. The remainder operator will come in very handy here.

The strategy expressed in the preceding paragraph can be translated into the following pseudocode method, which is easiest to implement as a top-level function in the `EnigmaRotor.py` module:

```
def apply_permutation(index, permutation, offset):
    Compute the index of the letter after shifting it by the offset, wrapping around if necessary.
    Look up the character at that index in the permutation string.
    Return the index of the new character after subtracting the offset, wrapping if necessary.
```

If you implement this function and call it in the `key_pressed` and `key_released` methods in `EnigmaModel`, your simulation should implement this first stage. Pressing the **Q** key when the rotor setting is **AAA** should light up the lamp for the letter **I**. If you click on the fast rotor to advance the rotor setting to **AAB**, clicking **Q** again should light the lamp for the letter **V**.

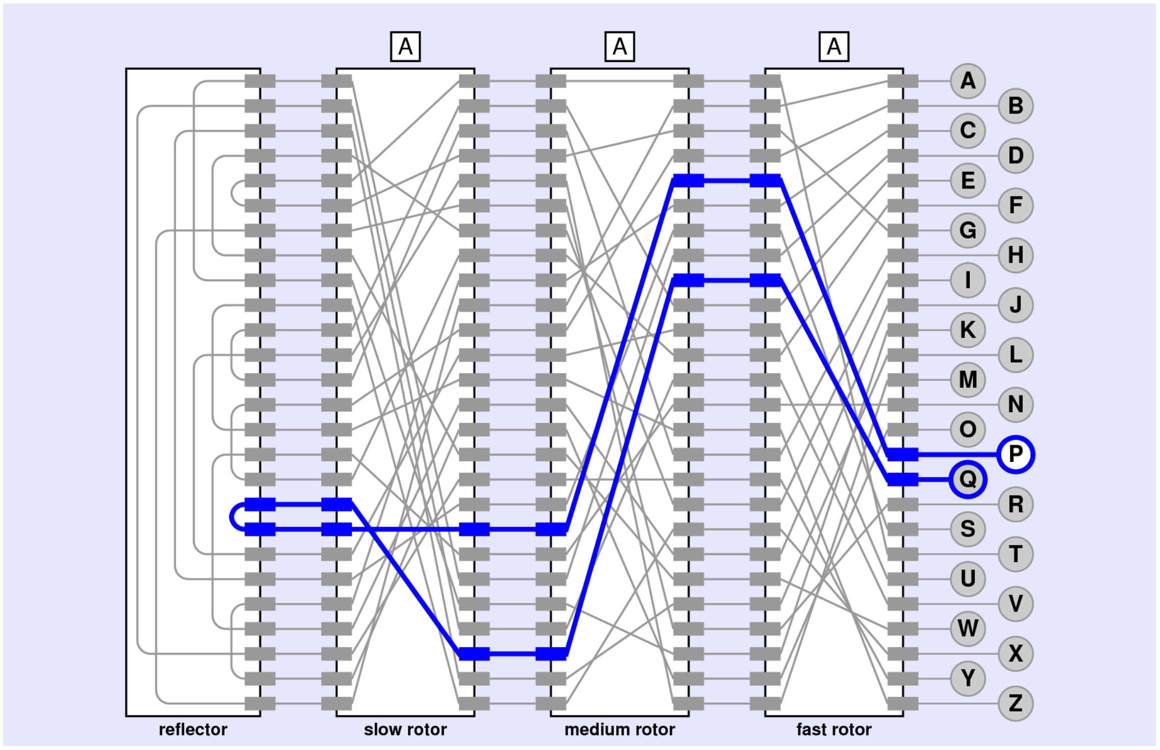
Milestone #5: Implement the full encryption path

Once you have completed Milestone #4 (and, in particular, once you have implemented a working version of `apply_permutation`), you're ready to tackle the complete Enigma encryption path. Pressing a key on the Enigma keyboard sends a current from right to left through the fast rotor, through the middle rotor, and through the slow rotor. From there, the current enters the reflector, which implements the following fixed permutation for which the offset is always 0:

```
REFLECTOR_PERMUTATION = "IXUHFEZDAOMTKQJWNSRLCYPBVG"
```

When the current leaves the reflector, it makes its way backward from left to right, starting with the slow rotor, moving on to the medium rotor, and finally passing through the fast rotor to arrive at one of the lamps. The complete path that arises from pressing **Q** is illustrated in Figure 5. As you can see from the diagram, the current flows through the fast rotor from right to left to arrive at **I** (index 8), through the medium rotor to arrive at **X** (index 23), and through the slow rotor to arrive at **R** (index 17). It then makes a quick loop through the reflector and emerges at **S** (index 18). From there, the current makes its way backward through the slow rotor straight across to **S** (index 18), through the medium rotor to **E** (index 4) and finally landing at **P** (index 15).

Figure 5. Complete trace of the encryption path when the operator presses Q



If everything is working from your previous milestones, making your way through the rotors and the reflector should not be too difficult. The challenge comes when you need to move backward through the rotors. The permutation strings show how the letters are transformed when you move through a rotor from right to left. When the signal is running from left to right, however, you can't use the permutation strings directly because the translation has to happen "backwards." What you need is the ***inverse*** of the original permutation.

The idea of inverting a key is most easily illustrated by example. Suppose, that the encryption key is `"QWERTYUIOPASDFGHJKLZXCVBNM"`, which is the example I used in my first cryptography lecture. That key represents the following translation table:

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Q	**W**	**E**	**R**	**T**	**Y**	**U**	**I**	**O**	**P**	**A**	**S**	**D**	**F**	**G**	**H**	**J**	**K**	**L**	**Z**	**X**	**C**	**V**	**B**	**N**	**M**

The translation table shows that **A** maps into **Q**, **B** maps into **W**, **C** maps into **E**, and so on. To turn the encryption process around, you have to read the translation table from bottom to top, looking to see what plaintext letter gives rise to each possible letter in the ciphertext. For example, if you look for the letter **A** in the ciphertext, you discover that the corresponding letter in the plaintext must have been **K**. Similarly, the only way to get a **B** in the ciphertext is to start with an **X** in the original message. The first two entries in the inverted translation table therefore look like this:

A	**B**
↓	↓
K	**X**

If you continue this process by finding each letter of the alphabet on the bottom of the original translation table and then looking to see what letter appears on top, you will eventually complete the inverted table, as follows:

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
K	**X**	**V**	**M**	**C**	**N**	**O**	**P**	**H**	**Q**	**R**	**S**	**Z**	**Y**	**I**	**J**	**A**	**D**	**L**	**E**	**G**	**W**	**B**	**U**	**F**	**T**

The inverted key is simply the 26-letter string on the bottom row, which in this case is `"KXVMCNOPHQRSZYIJADLEGWBUFT"`.

To complete this milestone, you first task is to implement the function `invert_key`. Once you have done so, you need to change the constructor for `EnigmaRotor` so that each rotor contains both a right-to-left and a left-to-right version of the permutation. The offset applies to each of these permutations in exactly the same way.

Milestone #6: Implement the rotor advance on pressing a key

The final step in the creation of the Enigma simulator is to implement the feature that advances the fast rotor every time a key is pressed. Although some sources (and most of the movies) suggest that the fast rotor advances *after* the key is pressed, watching the Enigma machine work makes it clear that it is the force of the key press that advances the rotor. Thus, the fast rotor advances *before* the translation occurs. When the fast rotor has made a complete revolution, the medium rotor advances. Similarly, when the medium rotor makes a complete revolution, the slow rotor advances.

Given that you already have a `key_pressed` method that is called when the user presses an Enigma key and a function to advance a rotor, the only new capability you need to implement is the "carry" from one rotor to the next. A useful technique to get this process working is to change the definition of `advance` so that it returns a Boolean: `False` in the usual case when no carry occurs, and `True` when the offset for that rotor wraps back to 0. The function that calls `advance` can check this result to determine whether it needs to propagate the carry to the next rotor.

When you complete this milestone, you're done!

Strategy and tactics

As with all projects in this course, the most important advice is to start early. You have two weeks to complete this assignment, which means that you can complete one milestone every two days and still finish early. Beyond that, most of the milestones require less than ten lines of new code. For those, you can almost certainly complete two or even three milestones in a day. What almost certainly won't work is if you start the day before it is due and try to finish all six milestones in a day. It is also important to complete each milestone before moving on to the next. Most of the later milestones depend on the earlier ones, and you need to know that the earlier code is working before you can debug the code you've added on top of the existing base.

The following tips will also probably help you do well on this assignment:

- *Try to get into the spirit of the history.* Although this project is an assignment in 2022, it may help to remember that the outcome of World War II once depended on people solving this problem using tools that were far more primitive that the ones you have at your disposal. Computing sometimes matters a great deal, and there will probably be situations in your lifetimes when the consequences of solving some programming challenge will be just as important. The fate of the world may someday lie in your hands, just as it did for the cryptographers at Bletchley Park.
- *Draw lots of diagrams.* Understanding how the Enigma machine works is an exercise in visualization. A picture is often worth a thousand words here.
- *Debug your program by seeing what values appear in the variables.* When you are debugging a program, it is far more useful to figure out what your program *is* doing than trying to determine why it *isn't* doing what you want. Every part of this assignment works with strings, and you can get an enormous amount of information about what your program is doing by using `print` to display the value of the strings you're using at interesting points.
- *Check your answers against the demonstration programs.* The web site includes a demo program that implements each milestone in the context of an interactive web page. Your code should generate the same results that the demo programs do.

Possible extensions

There are many things you can implement to make this assignment more challenging. Here are a couple of ideas:

- *Implement other features of the Enigma machine.* The German wartime Enigma was more complicated than the model presented here. In particular, the wartime machines had a stock of five rotors of which the operators could use any three in any order. The Germans also added a plugboard (***Steckerbrett*** in German) that swapped pairs of letters before they were fed into the rotors and after they came out.
- *Simulate the actions of the Bombe decryption machine.* This assignment has you build a simulator for the German Enigma machine. Look at the extended documentation on Enigma decryption on the CS 151 web site and consider implementing some of the British decoding strategies.