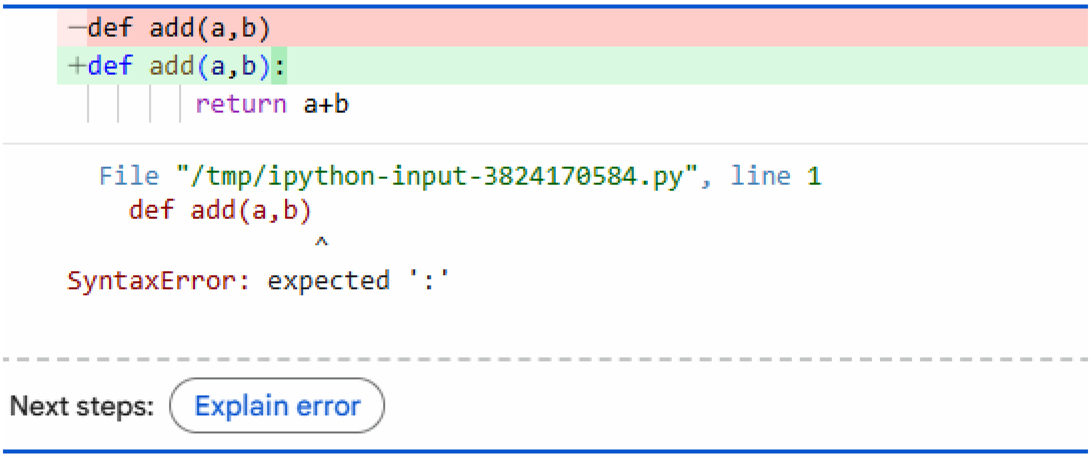
H.T.NO:2303A51836

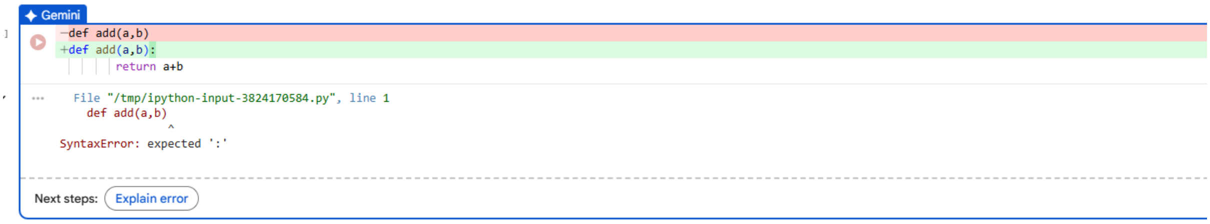
Batch 27

Task 1:Fixing Syntax Errors



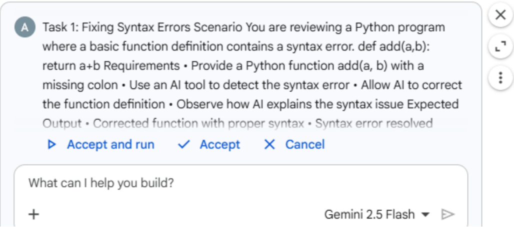
```
-def add(a,b)
+def add(a,b):
        return a+b
```

```
  File "/tmp/ipython-input-3824170584.py", line 1
    def add(a,b)
                ^
SyntaxError: expected ':'
```

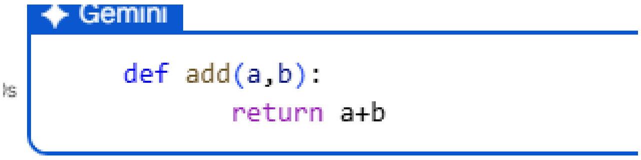


```
-def add(a,b)
+def add(a,b):
        return a+b
```

```
  File "/tmp/ipython-input-3824170584.py", line 1
    def add(a,b)
                ^
SyntaxError: expected ':'
```



Task 1: Fixing Syntax Errors Scenario You are reviewing a Python program where a basic function definition contains a syntax error. def add(a,b): return a+b Requirements • Provide a Python function add(a, b) with a missing colon • Use an AI tool to detect the syntax error • Allow AI to correct the function definition • Observe how AI explains the syntax issue Expected Output • Corrected function with proper syntax • Syntax error resolved

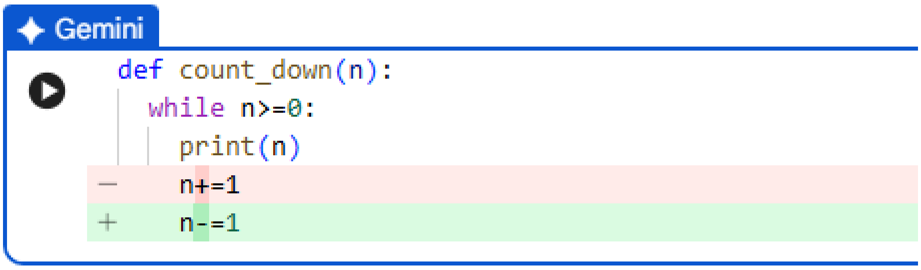


```
def add(a,b):
        return a+b
```

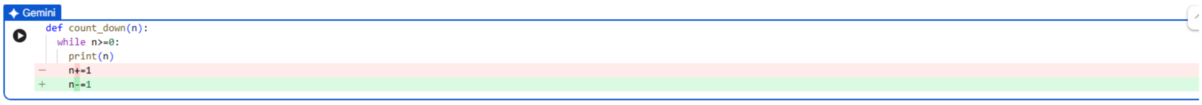
AI Generated Explanation of the fix:

Certainly! The original code had a SyntaxError: expected ':' on the line def add(a,b). In Python, when you define a function using the def keyword, the parameter list must always be followed by a colon (:). This colon signals the start of the function's code block. Without it, Python doesn't recognize the line as a complete function definition and expects to find that colon.

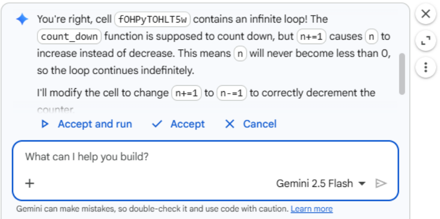
Task 2(Debugging Logic errors in loops):



```
def count_down(n):
  while n>=0:
    print(n)
−   n+=1
+   n-=1
```



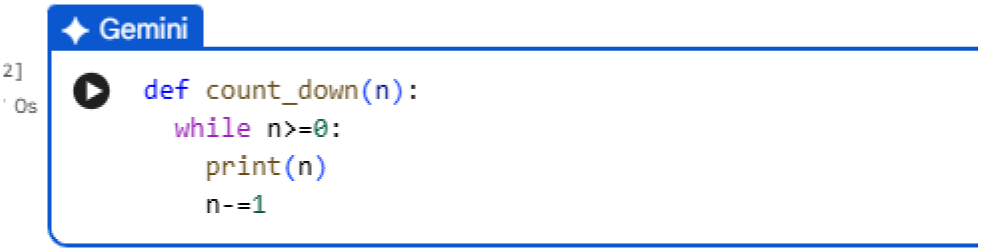
```
def count_down(n):
  while n>=0:
    print(n)
−   n+=1
+   n-=1
```



You're right, cell `fOHPyTOHLT5w` contains an infinite loop! The `count_down` function is supposed to count down, but `n+=1` causes `n` to increase instead of decrease. This means `n` will never become less than 0, so the loop continues indefinitely.

I'll modify the cell to change `n+=1` to `n-=1` to correctly decrement the counter.

Accept and run
Accept
Cancel
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes, so double-check it and use code with caution. Learn more



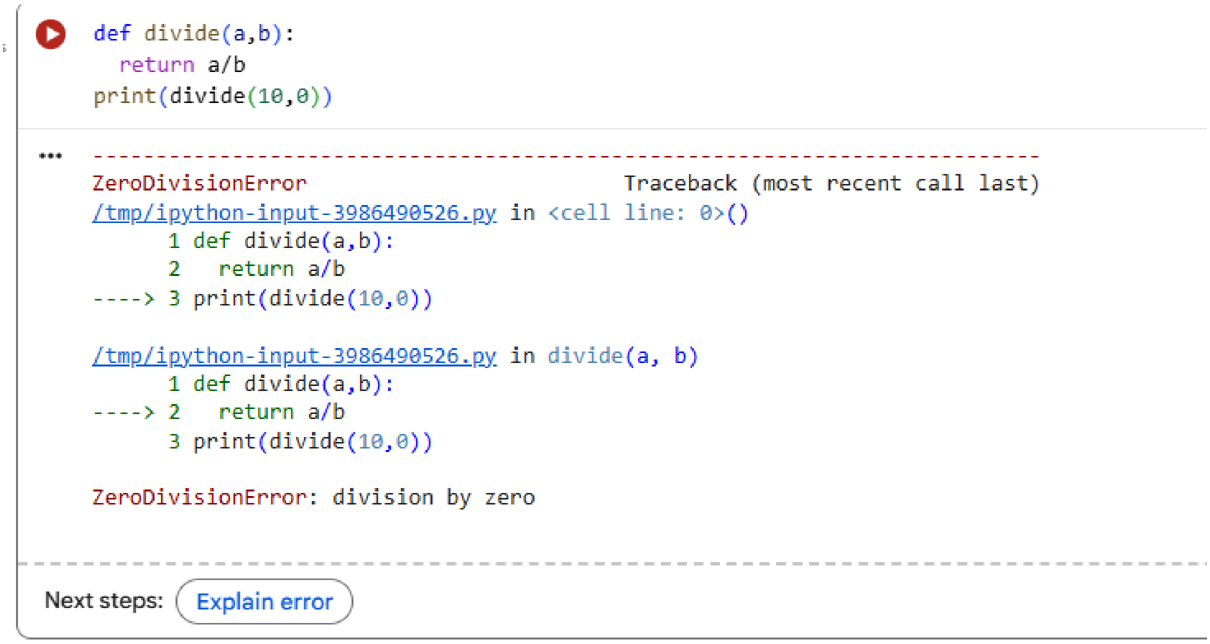
```
def count_down(n):
  while n>=0:
    print(n)
    n-=1
```

AI Explanation of the logic error:

The original logic error in the count_down function was within the line n+=1. This statement increments the value of n in each iteration of the while n>=0 loop. Since n was always increasing (or staying the same if it started at 0), the condition n>=0 would always remain true, leading to an infinite loop.

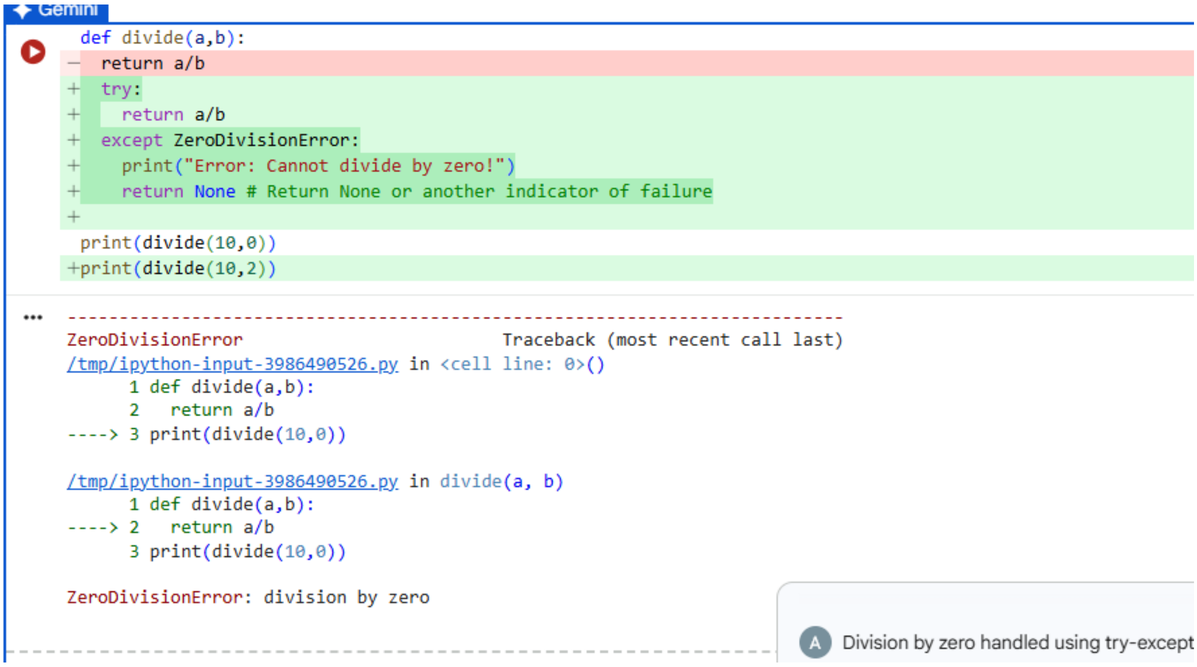
The fix involved changing n+=1 to n-=1, which correctly decrements n in each step, eventually making n less than 0 and terminating the loop as intended.

Task 3(Handling runtime errors(Division by zero)):



```
def divide(a,b):
  return a/b
print(divide(10,0))
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipython-input-3986490526.py in <cell line: 0>()
      1 def divide(a,b):
      2   return a/b
----> 3 print(divide(10,0))

/tmp/ipython-input-3986490526.py in divide(a, b)
      1 def divide(a,b):
----> 2   return a/b
      3 print(divide(10,0))

ZeroDivisionError: division by zero
```



```
 def divide(a,b):
-  return a/b
+  try:
+    return a/b
+  except ZeroDivisionError:
+    print("Error: Cannot divide by zero!")
+    return None # Return None or another indicator of failure
+
 print(divide(10,0))
+print(divide(10,2))
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipython-input-3986490526.py in <cell line: 0>()
      1 def divide(a,b):
      2   return a/b
----> 3 print(divide(10,0))

/tmp/ipython-input-3986490526.py in divide(a, b)
      1 def divide(a,b):
----> 2   return a/b
      3 print(divide(10,0))

ZeroDivisionError: division by zero
```

A Division by zero handled using try-except

AI Generated Explanation of Runtime Error handling:

Runtime errors, also known as exceptions, are problems that occur *while* your program is running, even after it has started execution. They are different from syntax errors (which prevent code from running) and logical errors (where code runs but produces incorrect results).

Think of them as unexpected events that disrupt the normal flow of your program. Common examples include:

- **ZeroDivisionError**: Trying to divide a number by zero.
- **FileNotFoundError**: When your program can't find a file it's trying to access.
- **TypeError**: An operation is attempted on an incompatible data type (e.g., trying to add a number to a string).

Why Handle Them?

If not handled, a runtime error will cause your program to crash. Error handling allows your program to:

1. **Prevent Crashes**: Instead of abruptly stopping, your program can respond gracefully.
2. **Provide Feedback**: Inform the user or log what went wrong in a user-friendly manner.
3. **Recover or Clean Up**: Your program can attempt to fix the problem, try an alternative, or at least clean up resources (like closing files) before exiting.

How to Handle Runtime Errors (Using try-except in Python)

The most common and effective way to handle runtime errors in Python is using a try-except block:

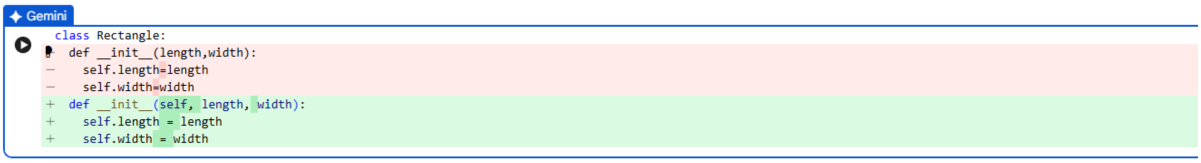
```
try:
  # Code that might cause an error
  numerator = 10
  denominator = 0
  result = numerator / denominator
  print(f"Result: {result}")
except ZeroDivisionError:
  # This block runs if a ZeroDivisionError occurs in the 'try' block
  print("Oops! You tried to divide by zero. Please use a non-zero denominator.")
  result = None
except Exception as e:
  # This is a more general catch-all for any other unexpected errors
  print(f"An unexpected error occurred: {e}")
  result = None
finally:
  # This block always runs, whether an error occurred or not
  print("Error handling process complete.")
```

- **try**: This is where you put the code that might raise an exception.

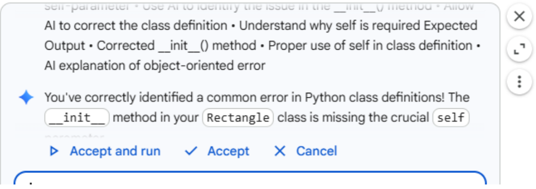
- **except**: If an error (of the specified type) occurs in the try block, the code in the corresponding except block is executed. You can have multiple except blocks for different error types.
- **finally**: This block is optional but very useful. Its code will *always* run, regardless of whether an exception was raised or handled. It's often used for cleanup tasks.

By implementing try-except blocks, you make your code more robust and user-friendly, allowing it to respond intelligently to unforeseen problems.

Task 4(Debugging Class Definition Errors):



```
 class Rectangle:
-  def __init__(length,width):
-    self.length=length
-    self.width=width
+  def __init__(self, length, width):
+    self.length = length
+    self.width = width
```



self-parameter • Use AI to identify the issue in the __init__() method • Allow AI to correct the class definition • Understand why self is required Expected Output • Corrected __init__() method • Proper use of self in class definition • AI explanation of object-oriented error

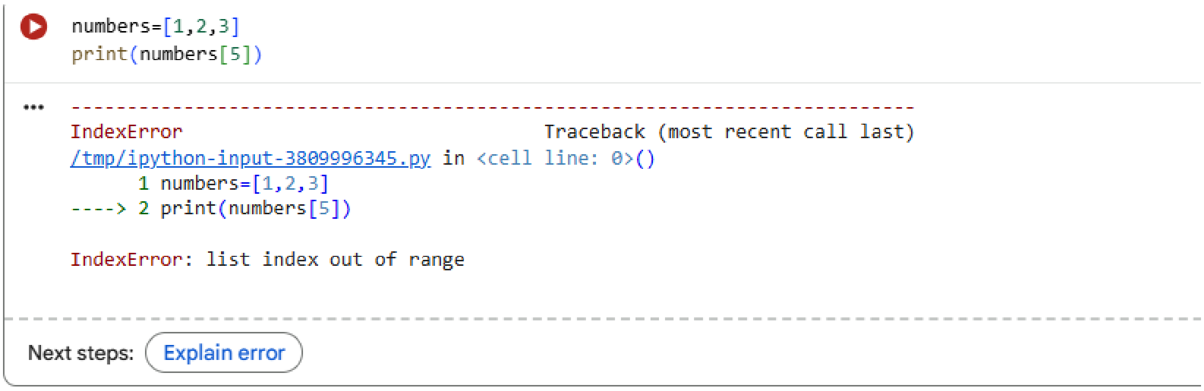
You've correctly identified a common error in Python class definitions! The `__init__` method in your `Rectangle` class is missing the crucial `self`

AI Explanation of the object-oriented error:

You've correctly identified a common error in Python class definitions!
The __init__ method in your Rectangle class is missing the crucial ‘self’ parameter.

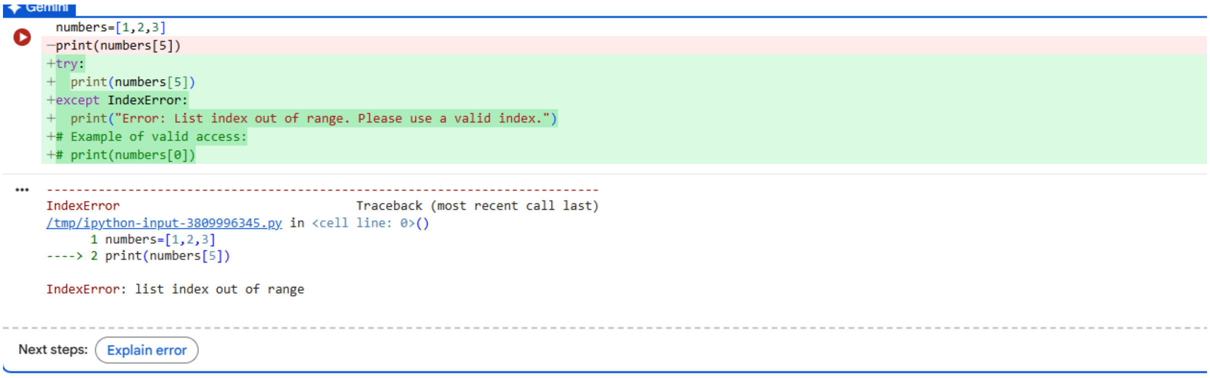
In Python, self is a reference to the instance of the class itself. It's the first parameter that every instance method must have. Without self, the method wouldn't know which specific Rectangle object's length and width attributes it should be setting.

Task 5(Resolving Index Errors in Lists):



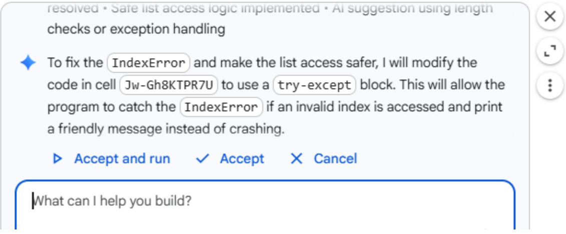
```
numbers=[1,2,3]
print(numbers[5])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipython-input-3809996345.py in <cell line: 0>()
      1 numbers=[1,2,3]
----> 2 print(numbers[5])

IndexError: list index out of range
```



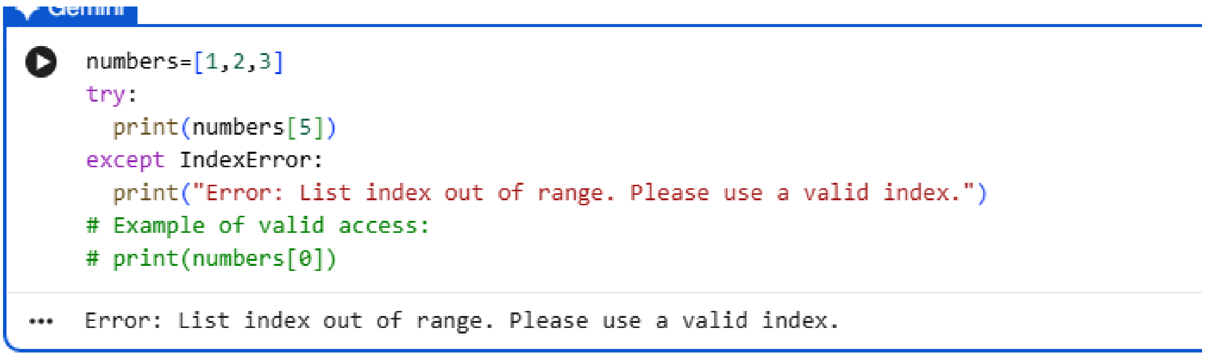
```
 numbers=[1,2,3]
-print(numbers[5])
+try:
+  print(numbers[5])
+except IndexError:
+  print("Error: List index out of range. Please use a valid index.")
+# Example of valid access:
+# print(numbers[0])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipython-input-3809996345.py in <cell line: 0>()
      1 numbers=[1,2,3]
----> 2 print(numbers[5])

IndexError: list index out of range
```



resolved • Safe list access logic implemented • AI suggestion using length checks or exception handling

To fix the `IndexError` and make the list access safer, I will modify the code in cell `Jw-Gh8KTPR7U` to use a `try-except` block. This will allow the program to catch the `IndexError` if an invalid index is accessed and print a friendly message instead of crashing.



```
numbers=[1,2,3]
try:
  print(numbers[5])
except IndexError:
  print("Error: List index out of range. Please use a valid index.")
# Example of valid access:
# print(numbers[0])
```

```
Error: List index out of range. Please use a valid index.
```

- **AI suggestion using exception handling**:
- The try-except mechanism was used to gracefully manage the error, as suggested.
- The try-except mechanism was suggested for handling the IndexError for several key reasons, especially when dealing with runtime errors that are hard to predict or prevent with simple conditional checks:
- **Graceful Error Handling**: Instead of the program crashing abruptly, try-except allows you to 'catch' the error and execute alternative code. This ensures your program can continue running or provide a user-friendly message, rather than a cryptic traceback.
- **Separation of Concerns**: It cleanly separates the 'normal' code logic from the 'error-handling' logic. The try block contains the code you expect to work, and the except block contains what to do if things go wrong. This makes your code cleaner and easier to read.

- **Handling Unpredictable Events**: While a simple length check (if index < len(my_list):) works well for preventing IndexError when you *know* you might be accessing an invalid index, try-except is more general. It can catch errors that might arise from more complex or unforeseen conditions, not just a simple bounds check.
- **Robustness and Reliability**: Using try-except makes your code more robust against unexpected inputs or states. It's a fundamental pattern in Python for building reliable applications that can gracefully recover from issues.
- In the case of IndexError, while a length check is also a valid approach, try-except provides a more general and idiomatic Python way to handle such exceptions, especially if the potential for an out-of-bounds access is rare or not easily predicted by a simple if statement.
-