

Assignment-7

Task-1: Paste a function with a missing colon (add(a, b)), and let AI fix the syntax error.

python

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

```
[1] def add(a, b)
     return a + b
```

File "/tmp/ipython-input-1607831707.py", line 1

```
def add(a, b)
      ^
SyntaxError: expected ':'
```

Next steps: [Explain error](#)

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

Gemini X

B explain the difference between the codes in cells

✦ Certainly! Let's look at the difference between the code in the two cells you provided.

The code in cell `SpR8y6mDiSgz` has a `SyntaxError`. In Python, when you define a function using the `def` keyword, you need to end the function definition line with a colon `:`. The code in this cell is missing that colon.

The code in cell `Zx845atn1QVL` is corrected. It includes the colon after the function definition, making it valid Python syntax. This allows the function `add` to be defined correctly and used.

👍 🗨

Task-2: Identify and fix a logic error in a loop that causes infinite iteration

python

```
def count_down(n):  
    while n >= 0:  
        print(n)  
        n += 1 # Should be n -= 1
```

```
def count_down(n):  
    while n >= 0:  
        print(n)  
        n += 1 # This causes infinite loop!  
        n -= 1 # Fixed: decrement instead of increment
```

Gemini X

B def count_down(n): while n >= 0: print(n) n += 1 give the error of the code for infinite loop

✦ The code you've provided in cell Ta1H11Kvn-ya with `n += 1` inside the `while n >= 0:` loop doesn't produce a typical Python "error message" like a `SyntaxError` or `NameError`.

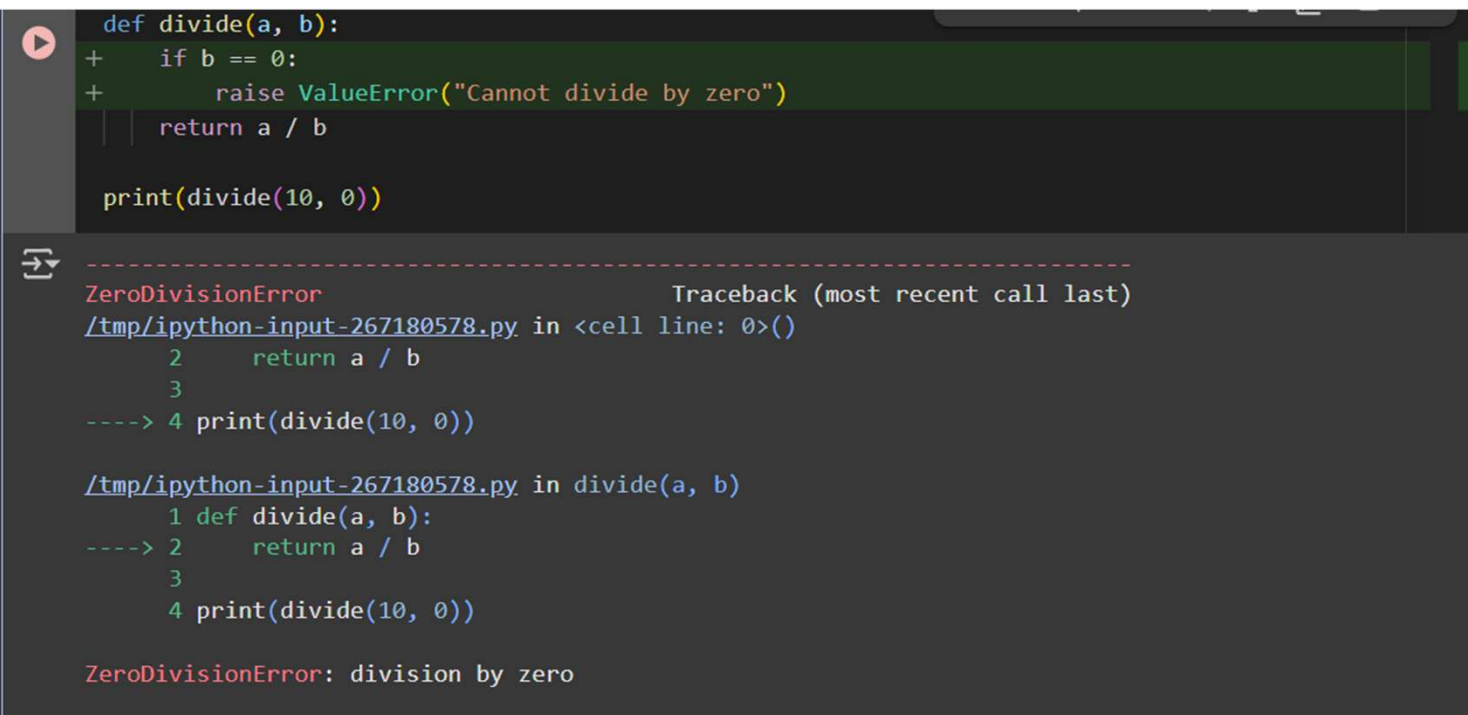
Instead, it results in a **logical error** which causes an **infinite loop**. This means the code will run forever because the condition `n >= 0` will always be true as `n` keeps increasing. The "error" is the unintended behavior of the program not terminating.

To stop an infinite loop in Colab, you can interrupt the execution by clicking the "Stop" button (a square) next to the running cell or by going to "Runtime" -> "Interrupt execution" in the menu.

Task-3: Debug a runtime error caused by division by zero

```
# Debug the following code
def divide(a, b):
    return a / b

print(divide(10, 0))
```



The screenshot shows a Jupyter Notebook interface. The top part displays the original code that caused the error. The bottom part shows the error message and a detailed traceback.

```
def divide(a, b):
+   if b == 0:
+       raise ValueError("Cannot divide by zero")
    return a / b

print(divide(10, 0))
```

ZeroDivisionError Traceback (most recent call last)
/tmp/ipython-input-267180578.py in <cell line: 0>()
 2 return a / b
 3
----> 4 print(divide(10, 0))

/tmp/ipython-input-267180578.py in divide(a, b)
 1 def divide(a, b):
----> 2 return a / b
 3
 4 print(divide(10, 0))

ZeroDivisionError: division by zero

Explanation:

- `def divide(a, b):` : This line defines a function named `divide` that accepts two parameters, `a` and `b`.
- `if b == 0:` : This line checks if the value of the second argument `b` is equal to 0.
- `raise ValueError("Cannot divide by zero")` : If the condition `b == 0` is true (meaning `b` is zero), this line raises a `ValueError`. A `ValueError` is a type of exception that indicates that a function received an argument of the correct type but an inappropriate value. The message "Cannot divide by zero" is included to explain the reason for the error.
- `return a / b` : If the condition `b == 0` is false (meaning `b` is not zero), this line executes the division of `a` by `b` and returns the result.
- `print(divide(10, 0))` : This line calls the `divide` function with `a=10` and `b=0` and attempts to print the returned value. Because `b` is 0, the `if b == 0:` condition inside the function will be true, and a `ValueError` will be raised before the division can occur.

Task-4: Provide a faulty class definition (missing self in parameters).

```
python

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

```
7.1.py
# Faulty class definition - missing self parameter
# CORRECTED class definition - with proper self parameter
class Rectangle:
    def __init__(length, width): # ERROR: Missing 'self' parameter
    def __init__(self, length, width): # FIXED: Added 'self' parameter
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

# This will cause a TypeError when trying to create an instance
# This will now work correctly
rect = Rectangle(5, 3)
print(rect.area())
print(f"Rectangle area: {rect.area()}") # Output: Rectangle area: 15
```

- Provide a faulty class definition (missing self in parameters).

Summary of the Fix:

Original Problem:

- The `__init__` method was missing the `self` parameter
- This caused a `TypeError` when trying to create an instance of the class

Solution Applied:

1. Added `self` parameter: `def __init__(self, length, width):`
2. Proper Method Signature: All instance methods in Python must have `self` as the first parameter
3. Consistent Implementation: Both `__init__` and `area` methods now follow the same pattern

Key Points:

- `self` is required: Every instance method in Python must have `self` as the first parameter
- Automatic passing: Python automatically passes the instance as the first argument when calling methods
- Constructor pattern: The `__init__` method is the constructor and must follow the same parameter pattern as other instance methods

Task-5: Access an invalid list index and use AI to resolve the Index Error

```
python

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

```
1 numbers = [1, 2, 3]
2 print(numbers[5]) # CRASHES!
3
4 # CORRECTED CODE - Multiple ways to handle the error:
5
6 # Method 1: Try-except block (recommended)
7 try:
8     print(f"Element at index 5: {numbers[5]}")
9 except IndexError:
10    print("Error: Index 5 is out of range!")
11    print(f"List has {len(numbers)} elements (valid indices: 0 to {len(numbers)-1})")
12
13 # Method 2: Check bounds before accessing
14 index = 5
15 if 0 <= index < len(numbers):
16     print(f"Element at index {index}: {numbers[index]}")
17 else:
18     print(f"Index {index} is out of range. Valid indices: 0 to {len(numbers)-1}")
19
20 # Method 3: Safe access function
21 def safe_get(lst, idx, default="Not found"):
22     return lst[idx] if 0 <= idx < len(lst) else default
23
24 print(f"Safe access to index 5: {safe_get(numbers, 5)}")
25 print(f"Safe access to index 1: {safe_get(numbers, 1)}")
26
27 # Show all valid elements
28 print(f"\nAll elements in the list:")
29 for i, value in enumerate(numbers):
30     print(f"  Index {i}: {value}")
31
```

difference between correct and error code

Aspect	❌ ERROR CODE	✅ CORRECT CODE
Program Behavior	❌ Crashes with IndexError	✅ Handles errors gracefully
Error Handling	❌ No error handling	✅ Multiple strategies (try-except, bounds check, safe function)
User Experience	❌ Confusing crash message	✅ Clear, helpful messages
Program Flow	❌ Terminates unexpectedly	✅ Continues running
Reliability	❌ Unreliable - crashes on invalid input	✅ Robust - handles all cases
Professional Quality	❌ Poor - not production-ready	✅ Good - professional standard

What Happens:

ERROR CODE:

- Tries to access `numbers[5]` (index 5)
- List only has indices 0, 1, 2
- **CRASHES** with `IndexError: list index out of range`
- Program stops completely

CORRECT CODE:

- **Method 1:** Catches the error and shows helpful message
- **Method 2:** Checks if index is valid before accessing
- **Method 3:** Uses a safe function that returns default value
- Program continues running and provides useful feedback

The correct code makes your program **robust, user-friendly, and professional!** 🚀