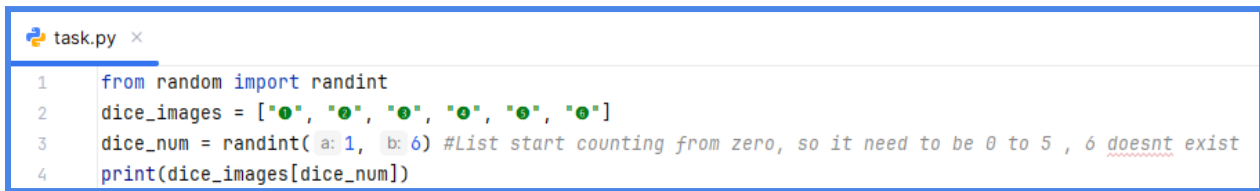


# DAY 13: Debugging, How to find and fix errors in your code

**4/02/2025:** So in this lesson, I's about some techniques and tips for how to find bugs and how to get rid of them from your code.

## Reproduce the Bug

This section explains how to reproduce and fix a common bug caused by improper list indexing. The issue occurs when selecting a random dice image from a list using `randint(1,6)`. Since Python lists use **zero-based indexing**, valid indices for a list of six items range from **0 to 5**, not **1 to 6**.

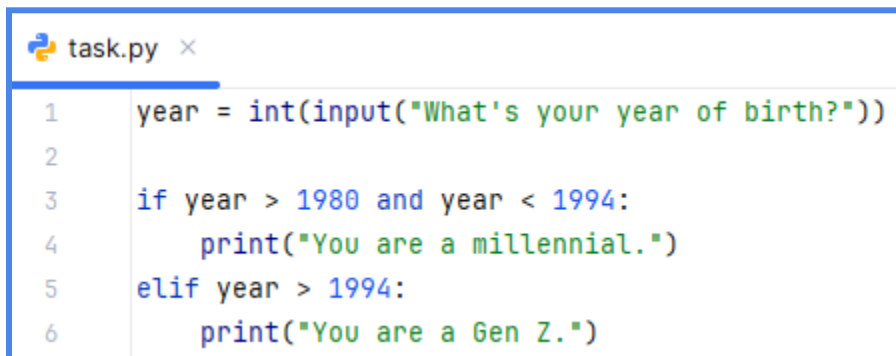


```
task.py x
1 from random import randint
2 dice_images = ["0", "0", "0", "0", "0", "0"]
3 dice_num = randint(a=1, b=6) #List start counting from zero, so it need to be 0 to 5 , 6 doesnt exist
4 print(dice_images[dice_num])
```

The bug leads to an **IndexError: list index out of range** when the randomly generated number is **6**, as there is no corresponding index in the list. To fix this, the code should generate a number between **0 and 5** using `randint(0,5)`. This ensures all dice images are accessible and prevents the error from occurring.

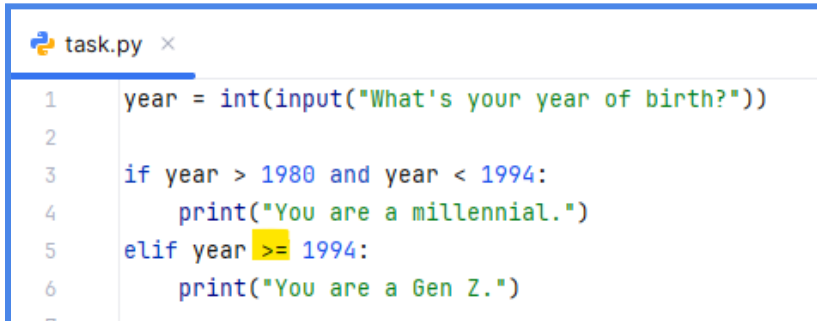
## Play Computer and Evaluate Each Line

This section emphasizes the importance of "**playing computer**"—manually going through code line by line to understand its logic, which is especially useful for debugging.



```
task.py x
1 year = int(input("What's your year of birth?"))
2
3 if year > 1980 and year < 1994:
4     print("You are a millennial.")
5 elif year > 1994:
6     print("You are a Gen Z.")
```

The example code takes a **year of birth** as input and classifies the user as either **Millennial** (born between 1980-1994) or **Gen Z** (born after 1994). However, when the input is **1994**, nothing is printed due to a logical error in the conditions.



```
task.py x
1 year = int(input("What's your year of birth?"))
2
3 if year > 1980 and year < 1994:
4     print("You are a millennial.")
5 elif year >= 1994:
6     print("You are a Gen Z.")
```

To include **1994** in one of the categories, the condition should use **greater than or equal to** (**>=**) or **less than or equal to** (**<=**) in one of the checks. Updating the condition ensures 1994 is properly classified as a **Millennial or Gen Z**, preventing the bug.

## Fixing Errors and Watching for Red Underlines

### 1. Fix Errors Before Proceeding

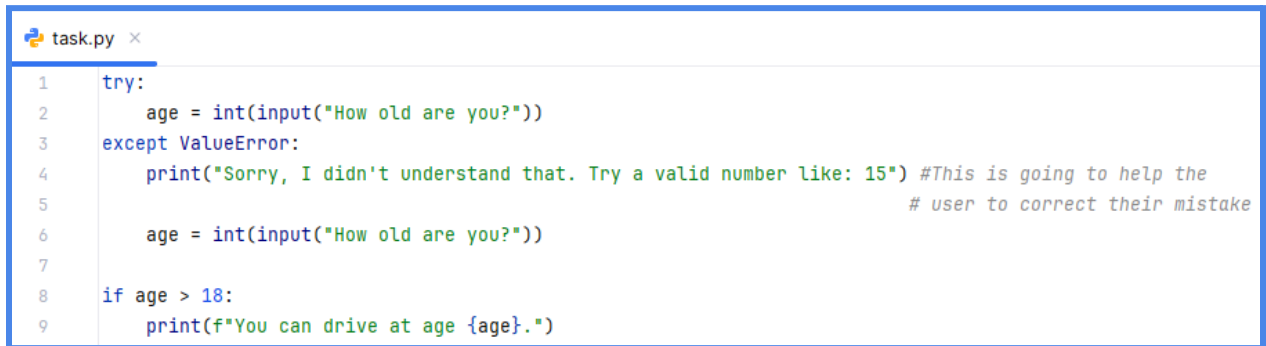
- Errors in the editor are easier to fix since they highlight the problematic line.
- Errors in the console appear only when running the code and depend on user input.

### 2. Using Google to Solve Errors

- Copy the relevant part of the error message (excluding specific code references).
- Search for the error along with "Python" to find explanations and solutions.

### 3. Handling User Input Errors with Try-Except

- Errors like `ValueError` occur when invalid input (e.g., "twelve" instead of 12) is converted into an integer.
- Use a `try` block to wrap the risky code and an `except` block to catch the error and provide user feedback.
- Prompt the user to enter a valid numerical input instead of crashing the program.



```
task.py x
1 try:
2     age = int(input("How old are you?"))
3 except ValueError:
4     print("Sorry, I didn't understand that. Try a valid number like: 15") #This is going to help the
5                                     # user to correct their mistake
6     age = int(input("How old are you?"))
7
8 if age > 18:
9     print(f"You can drive at age {age}.")
```

### 4. Debugging Logical Errors (No Error Messages)

- Some bugs don't cause errors but make the program behave unexpectedly.
- Example: Displaying a variable name instead of its value—solved using f-strings.
- These issues require problem-solving skills and experience to fix.

### 5. Improving Debugging Skills

- Solve more bugs to improve your debugging ability.
- Help others on Stack Overflow or Discord to gain experience.

## Squash bugs with a print() Statement

Use `print()` to check variable values and debug errors.

```
pages = int(input("Enter number of pages: "))
words_per_page == int(input("Enter words per page: ")) # Bug here!
total_words = pages * words_per_page
print(total_words)
```

The output is 0

```
print(f"Pages: {pages}")
print(f"Words per page: {words_per_page}")
```

`pages` prints correctly, but `words_per_page` is 0, revealing the issue.

```
words_per_page = int(input("Enter words per page: ")) # Fixed!
```

The error was using `==` (comparison) instead of `=` (assignment). With `print()`, debugging becomes faster and easier.

## Bringing out the BIG Gun: Using a Debugger

`print()` is great, but **debuggers** are more powerful for complex issues.

A debugger lets you **pause, inspect, and step through** code execution.

```
def mutate(a_list):
    b_list = []
    new_item = 0
    for item in a_list:
        new_item = item * 2
        new_item += random.randint(1, 3)
        new_item = add(new_item, item)
    b_list.append(new_item) # Indentation issue!
    print(b_list)
```

1. **Breakpoint** - You can set a breakpoint by clicking on a line in the gutter of the code (where the line numbers are). This line will be where the program pauses during debug run.
2. **Step Over** - This button will go through the execution of your code line by line and allow you to view the intermediate values of your variables.
3. **Step Into** - This will enter into any other modules that your code references. e.g. If you use a function from the random module it will show you the original code for that function so you can better understand its functionality and how it relates to your problems.
4. **Step Into My Code** - This does the same thing as Step Into, but it limits the scope to your own project code and ignores library code such as random.

## Final Debugging Tips

### 1. Take a Break 🧘

- If you're stuck, **step away** from the screen.
- A fresh perspective after a break can make debugging easier.

### 2. Ask a Friend 👥

- A second pair of eyes can spot **assumptions and mistakes** you might have missed.
- Helping others also strengthens **your own debugging skills**.

### 3. Run Code Often 🏃

- Test small changes **frequently** instead of waiting until the end.
- Prevents piling up **multiple bugs at once**.

### 4. Fix One Bug at a Time 🔍

- If you see multiple issues, tackle them **one by one**.
- Jumping between problems can make debugging **more confusing**.

### 5. Use Stack Overflow Wisely 🌐

- **Search first**—chances are, someone already had the same issue.
- Only post questions when you've **exhausted** all other debugging methods.

### 6. Bugs Are Part of the Process 🐛➡️🦋

- Every bug you fix makes you **stronger as a programmer**.
- Debugging is like **lifting weights**—the more reps, the better you get!

## Debugging Odd or Even

exercise.py

```
1 def odd_or_even(number):
2     if number % 2 == 0:
3         return "This is an even number."
4     else:
5         return "This is an odd number."
6 print(odd_or_even(5))
7
```

## Debugging Leap Year

exercise.py

```
1 def is_leap(year):
2     if year % 4 == 0:
3         if year % 100 == 0:
4             if year % 400 == 0:
5                 return True
6             else:
7                 return False
8         else:
9             return True
10    else:
11        return False
12
13 print(is_leap(2000))
14
```

## Debugging FizzBuzz

exercise.py

```
1 # Target is the number up to which we count
2 def fizz_buzz(target):
3     for number in range(1, target+1):
4         if number % 3 == 0 and number % 5 == 0:
5             print("FizzBuzz")
6         elif number % 3 == 0:
7             print("Fizz")
8         elif number % 5 == 0:
9             print("Buzz")
10        else:
11            print(number)
12 fizz_buzz(15)
13
```

