VAULTOFCODES
Assignment 2

1.Writeaprogramtocountwordfrequenciesinagiventext.

```
from collections import Counter
import re

def count_word_frequency(text):
    # Convert text to lowercase to ensure case-insensitivity
    text = text.lower()

    # Use regex to find words and ignore punctuation
    words = re.findall(r'\b\w+\b', text)

    # Use Counter to count the frequency of words
    word_counts = Counter(words)

    return word_counts

# Example usage
text = """
Your text goes here. It can be a paragraph, a page, or an entire book.
This script will count the frequency of each word in your text.
"""


word_frequency = count_word_frequency(text)

# Print the word frequencies
for word, count in word_frequency.items():
    print(f"{word}: {count})
```

OUTPUT:

```
your: 2
text: 2
goes: 1
here: 1
it: 1
can: 1
be: 1
a: 2
paragraph: 1
page: 1
or: 1
an: 1
entire: 1
book: 1
this: 1
script: 1
will: 1
count: 1
the: 1
frequency: 1
of: 1
each: 1
word: 1
in: 1


=== Code Execution Successful ===
```

2.PalindromeCheckerWriteaprogramthatchecksifagivenword is a palindrome.

Code:

```
   def is_palindrome(word):
       # Convert the word to lowercase to make the check case-
   insensitive
       word = word.lower()

     # Reverse the word and compare it to the original
       return word == word[::-1]
     # Input from the user
    word = input("Enter a word: ")
```

```
# Check if the word is a palindrome and print the result
if is_palindrome(word):
    print(f"'{word}' is a palindrome.")
else:
    print(f"'{word}' is not a palindrome.")
```

Output:

```
Enter a word: Hello
'hello' is not a
palindrome.
```

3. ListManipulation
Createalistofnumbers,thenwriteaprogramthatprintsthe square of each number in the list.

Program:

```
# Create a list of numbers
numbers = [1, 2, 3, 4, 5]

# Print the square of each number in the list
for number in numbers:
    square = number ** 2
    print(f"The square of {number} is {square}")
```

Output:

```
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
```

OOP'S IN PYTHON

SUDHAGONA SAI DEEKSHITH GOUD

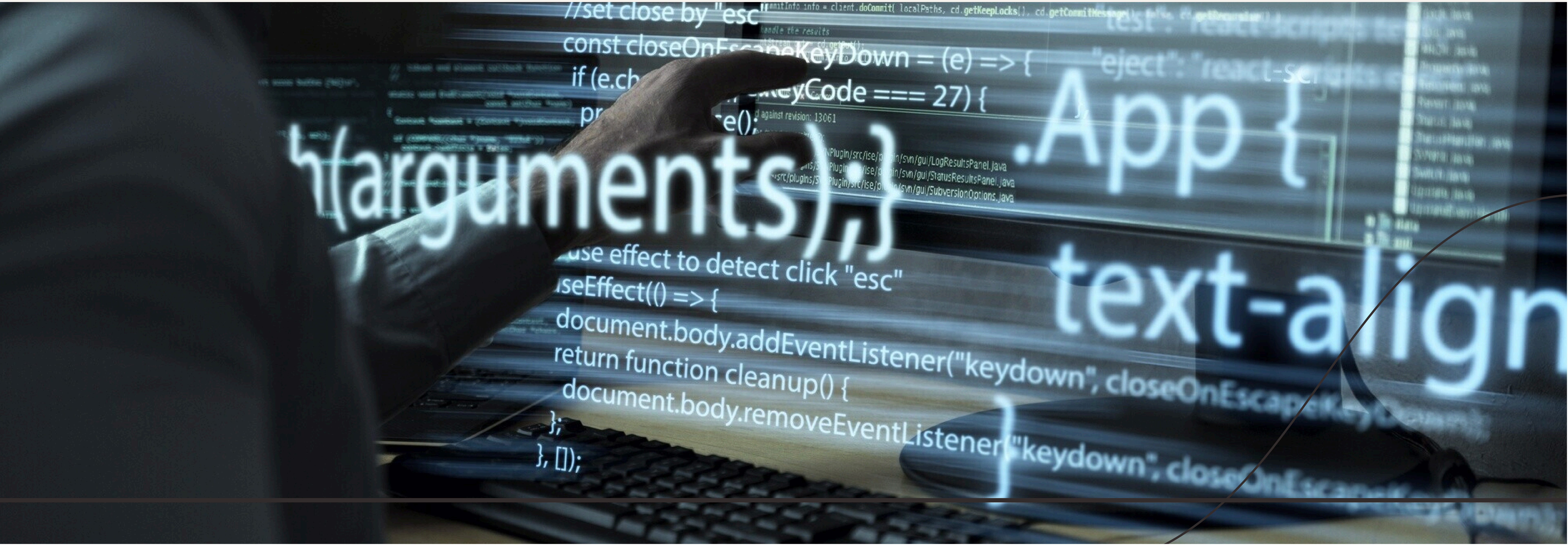
Introduction to OOP

```
ss";
mponent as ArrowIcon } from "../assets/icons/arrow.svg";
mponent as BoltIcon } from "../assets/icons/bolt.svg";
mponent as RightArrowIcon } from "../assets/icons/right-arrow.svg";

eState, useEffect, useRef } from "react";
sition } from "react-transition-group";
```

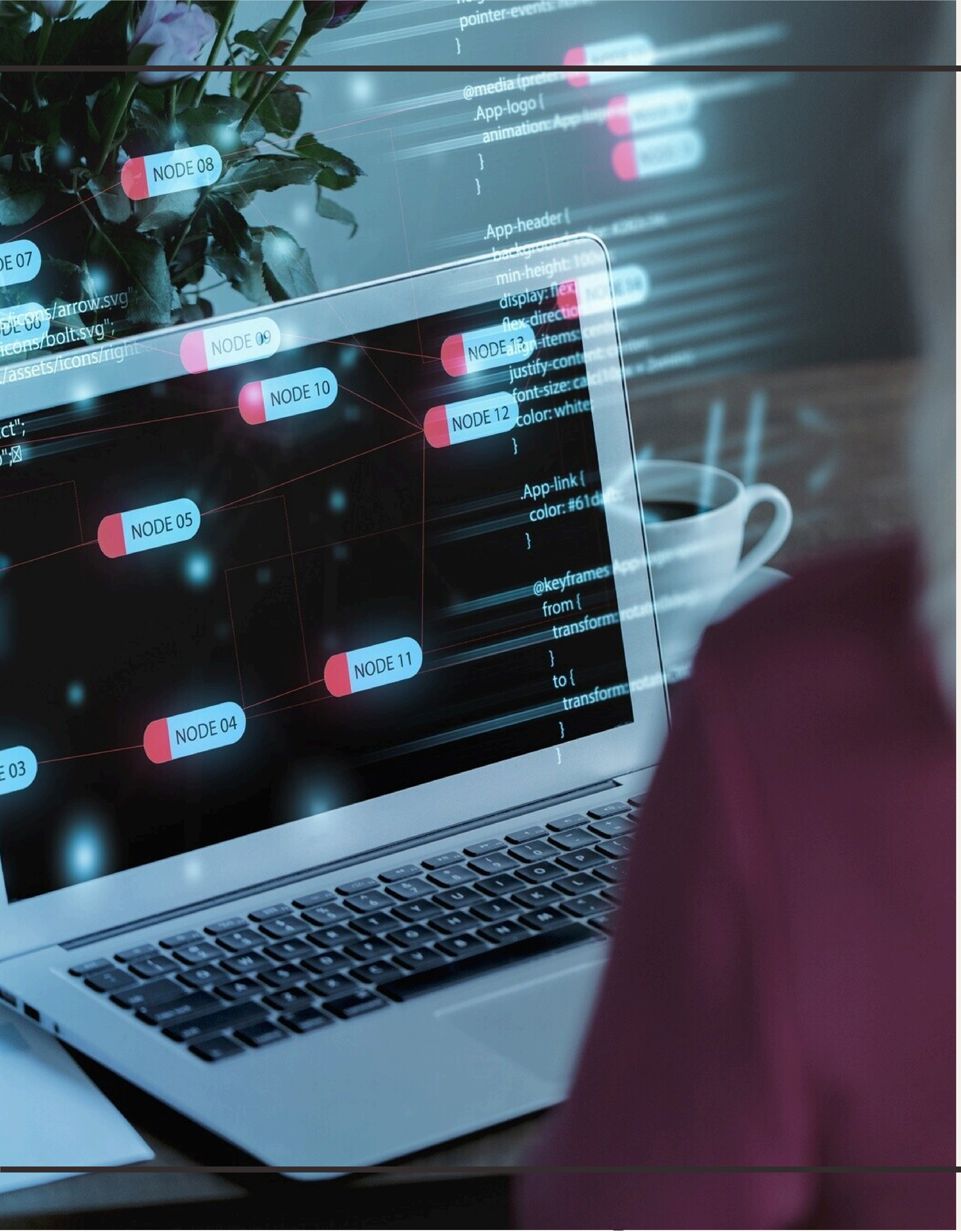
In this presentation, we will explore **Object-Oriented Programming (OOP)** in Python. We will discuss its **fundamental concepts**, common pitfalls that developers face, and best practices to enhance your coding skills. By the end, you will have a better understanding of how to effectively use OOP in your projects.

Classes and **Objects** are the building blocks of OOP in Python. A **class** defines a blueprint for creating objects, while an **object** is an instance of a class. Understanding these concepts is crucial for mastering OOP and designing efficient applications.



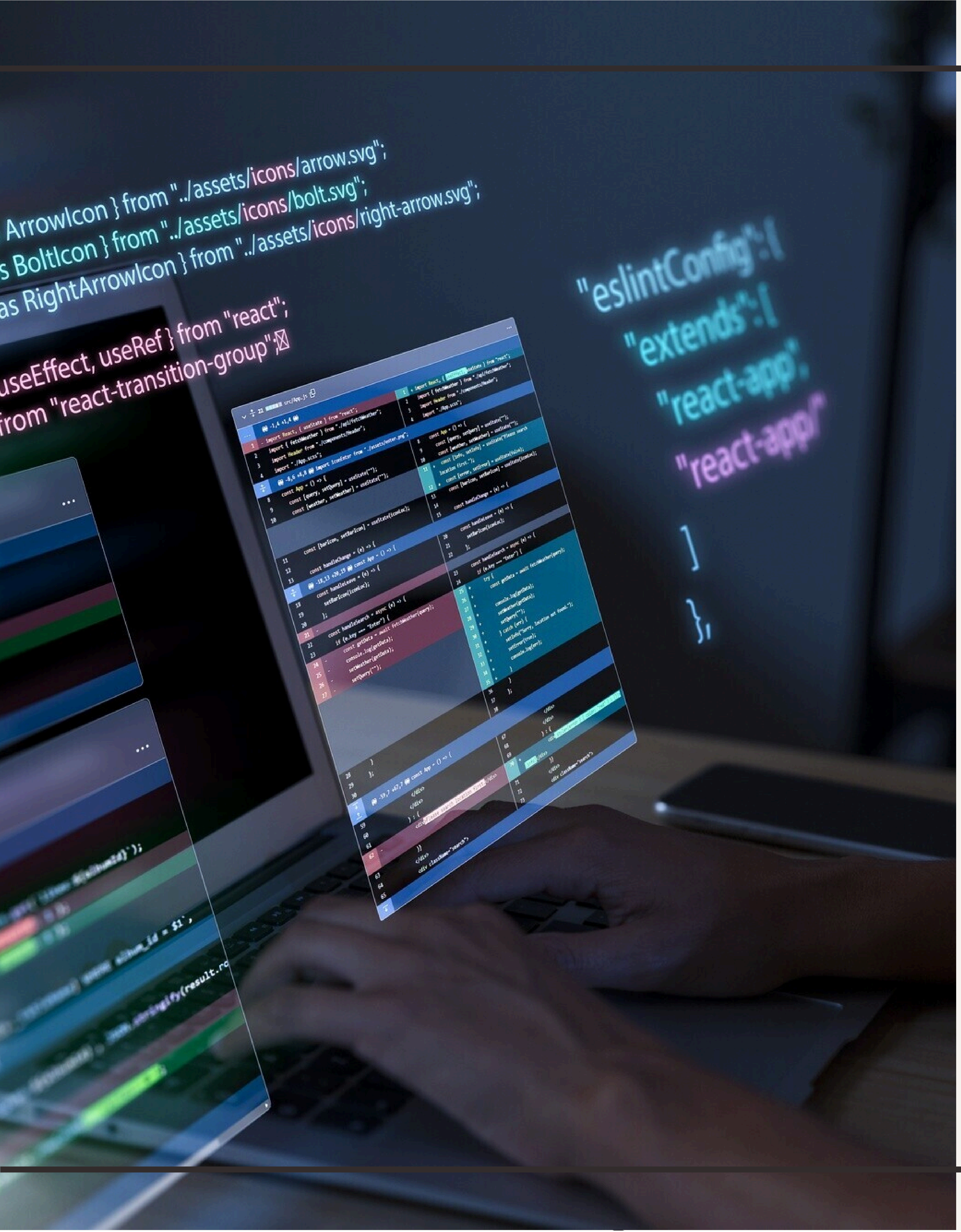
```
//set close by "esc"
const closeOnE      eKeyDown = (e) => {
if (e.ch      KeyCode === 27) {
pr      e();
(arguments),}
use effect to detect click "esc"
useEffect(() => {
document.body.addEventListener("keydown", closeOnEscapeKeyDown
return function cleanup() {
document.body.removeEventListener("keydown", closeOnEscape
};
}, []);
```

```
"test": "react-scripts te
"eject": "react-scripts
.App {
text-align
```



Encapsulation Explained

Encapsulation is the practice of bundling data and methods that operate on that data within a single unit, or **class**. It helps in **hiding** the internal state and only exposing a controlled interface. This promotes **data integrity** and reduces complexity.



Understanding Inheritance

Inheritance allows a class to inherit attributes and methods from another class, promoting **code reusability**. This concept helps in creating a **hierarchical relationship** between classes, making it easier to manage and extend code without redundancy.

Polymorphism in OOP

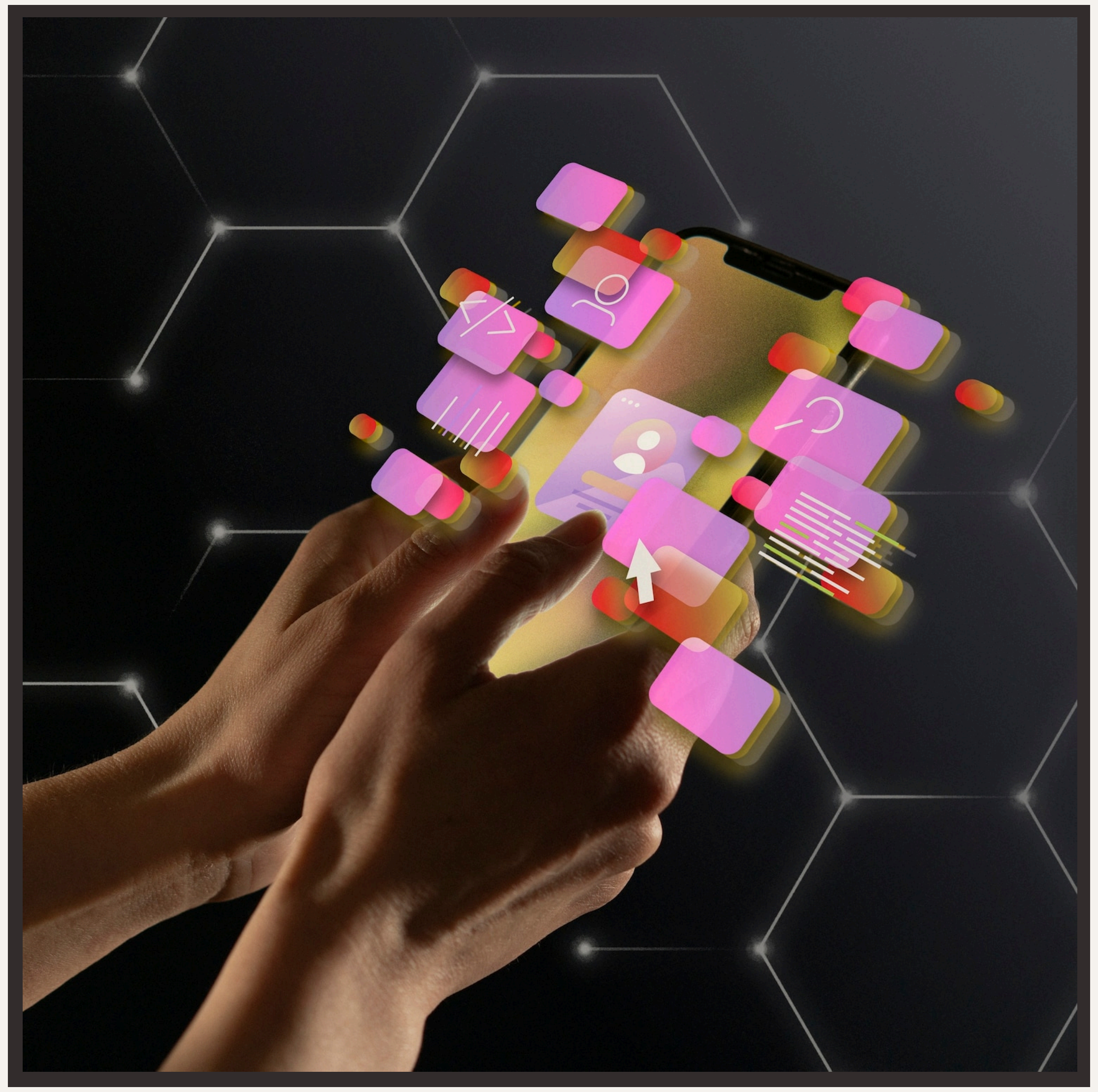
Polymorphism enables objects to be treated as instances of their parent class. This allows for **method overriding** and **overloading**, enhancing flexibility in code. Understanding polymorphism is essential for writing **generic** and **maintainable** code.

To master OOP in Python, follow best practices like **favoring composition over inheritance**, keeping classes focused, and adhering to the **Single Responsibility Principle**. These practices lead to **more modular**, **testable**, and **maintainable** code.

Testing OOP Code

Testing is crucial for ensuring the reliability of OOP code. Use **unit tests** to verify the functionality of individual classes and methods. This helps catch bugs early and ensures that changes do not introduce new issues.

Real-World Applications



OOP is widely used in software development, from **web applications** to **game development**. Understanding OOP principles allows developers to create scalable and efficient systems, making it an essential skill in the industry.

Conclusion:

Mastering OOP in Python involves understanding its core concepts, avoiding common pitfalls, and implementing best practices. By applying these principles, you can enhance your coding skills and develop robust applications that are easy to maintain and extend.

```
<html>
<head>
<script>
function show(shown, hidden) {
  document.getElementById(shown).style.display='block';
  document.getElementById(hidden).style.display='none';
  return false;
}
</script>
</head>
<body>

  <div id="Page1">
    Content of page 1
    <a href="#" onclick="return show('Page2','Page1');">Show page 2</a>
  </div>

  <div id="Page2" style="display:none">
    Content of page 2
    <a href="#" onclick="return show('Page1','Page2');">Show page 1</a>
  </div>

</body>
</html>
```