ISSN (Online):3048-5088

International Journal of Advancement and Innovation in Technology and Research (IJAITR)
Volume 3, Issue 1, (Jan-Jun) 2025

A Ground-Up Pedagogical Framework for Teaching Python Problem Solving

Radha Bhonde
Dept. of CSE
SKITM, Indore
Indore, India
shinderadha25@gmail.com

Deepak Bhonde
Dept. of CSE
SKITM, Indore
Indore, India

Kiran Rai
Dept. of CSE
SKITM, Indore
Indore, India

***Abstract*—-Accomplishing capability in Python programming depends not as it were on getting a handle on its language structure but, more vitally, on developing compelling problem-solving abilities through dynamic hone. For apprentices, working on well-scoped small problems a which disconnect one or two center concepts and surrender quick, unquestionable output provides the foremost coordinate course to internalizing computational rationale. Such issues ordinarily span fundamental categories (e.g., number juggling operations and sort transformation, string control, conditionals, circles, information structures, and basic capacities), reflecting a characteristic learning movement from consecutive execution to decision-making, reiteration, information organization, and seclusion. By over and over deciphering hypothetical builds into concrete code, learners uncover crevices in understanding, lock in in basic investigating, and construct the muscle memory required for rectify sentence structure and coherent stream. Organized fittingly, these works out strengthen foundational concepts and get ready understudies to handle more complex challenges. Thus, a beginners educational programs ought to prioritize hands-on issue solving using brief, concept-focused exercises while treating hypothetical instruction as a springboard for prompt application and iterative ability refinement.**

***Index Terms*—Python exercises, Hands-on coding, Active learning in programming, Learning progression in Python.**

I. Introduction

Achieving expertise in Python programming goes far beyond syntax. Essentially, it requires strong analysis and connections of error resistance skills. For those starting with coding adventures, fighting the challenge of byte size provides the most efficient way to establish a solid understanding of understanding. This strategy translates abstract theory into capabilities and promotes an intuitive sense of algorithmic discussion. The repeated focus, training and newcomer issues in many real educational materials highlight the fundamental truth. Passive absorption of materials, such as textual overlap and consideration of demonstrations, is no longer a true understanding of programming. True commands arise from a dedicated and cyclical process of trying solutions, fixing in the face of errors, and increasing improvements in the approach. Such active participation is extremely important as it forces learners to transform conceptual ideas such as loop functionality and conditional logic into material, executed scripts. Of course, this transformation is misleading, requires debugging, and constructs a critical reflection for accurate syntax and coherent program flow. Ultimately, the educational path of beginners should be strongly emphasized in the hands of the question, thereby allowing theoretical concepts to act as practical, immediate stepping stones. This synergy ensures that core ideas are not only genuinely learned, but also can be thoroughly and easily used.

II. Literature Review:

A literature review on understanding ”mini challenges” in Python for novices explores how small, focused programming tasks can support learning in introductory computer science education. While the term ”mini challenges” isn’t widely standardized in academic literature, related research sheds light on their pedagogical value. Here's a synthesis of key insights:

1. Learning Challenges for Novices

Novice programmers often struggle with:

- Abstract thinking and algorithmic logic
- Syntax and semantics of programming languages
- Misconceptions about how code executes (e.g., variable assignment, loops)
- Lack of confidence and fear of failure3

2. Role of Mini Challenges

Mini challenges—short, targeted coding tasks—can:

- Reinforce specific concepts like loops, conditionals, or functions
- Provide immediate feedback and build confidence
- Encourage active learning and problem-solving
- Help instructors identify misconceptions early

3. Instructional Strategies

Effective use of mini challenges includes:

- Embedding them in larger projects or lessons
- Using them as formative assessments
- Encouraging peer discussion and reflection
- Aligning them with common novice errors3

4. Research-Based Approaches

Studies suggest that:

- Structured, bite-sized tasks reduce cognitive overload
- Addressing misconceptions explicitly improves conceptual understanding
- Action research and systematic literature reviews help identify best practices for teaching programming to beginners

I. Understanding “Mini Challenges” in Python for Novices

Understanding mini-working tasks in Python for beginners refers to limited complexity issues focusing on one or many basic programming ideas in the context of Python organization for beginners. These tasks are intentionally not complicated, concise, and can be resolved in a few minutes to an hour. Typically, it avoids relying on external modules and instead encourages the creation of custom logic, enhancing the core concept. It is important for beginner level questions to create clear and immediate editions to help learners check their solutions effectively and create trust. Classification of problems in this way helps learners determine areas that need improvement and promote a balanced acquisition of various Python skills. Education platforms often organize tasks at the beginner level in sections such as string manipulation, additional typical group-based features, and more. ”Fundamental syntax, includes operations, includes operations. Repeatability (e.g., loops) Let's explore a short example and its code snippet:

1. Basic Calculations & Variables

```
a, b = int(input(”A: ”)), int(input(”B: ”))
print(”Swapped:”, b, a)
c = float(input(”Temp °C: ”))
print(”In °F:”, (c * 9/5) + 32)
```

2. Text & String Operations

```
text = input(”Text: ”)
print(”Reversed:”, text[::-1])
print(”Vowels:”, sum(1 for ch in text if ch.lower() in ”aeiou”))
print(”Masked:”, text[0] + ””(len(text)-2) + text[-1] if len(text) ¿ 2 else text)
```

3. Decision Making

```
n = int(input(”Number: ”))
print(”Even” if n % 2 == 0 else ”Odd”)
s1, s2 = input(”Str1: ”), input(”Str2: ”)
print(”Equal” if s1 == s2 else ”Not equal”)
```

4. Repetition & Loops

```
print(”Sum 1 to n:”, sum(range(1, n+1)))

print(”Prime:”, all(n % i for i in range(2, int(n**0.5)+1)) if n ¿ 1 else False)
```

5. Functions

```
def gcd(a, b): return a if b == 0 else gcd(b, a % b)
print(”GCD:”, gcd(a, b))
print(”Radians:”, math.radians(float(input(”Degrees: ”))))
print(”Hello”, input(”Name: ”))
```

6. List & Array Manipulation

```
lst = list(map(int, input(”List: ”).split()))
lst.append(100)
print(”Sorted:”, sorted(lst))
print(”Evens:”, [x for x in lst if x % 2 == 0])
```

TABLE I: Core Python Problem Types for Novice Programmers

S.No.	Category	Overview	Sample Exercises	Key Concepts
1	Basic Calculations & Variables	Simple math operations and storing values in names	Swap two variables Temperature Converter basic Calculator	Variables, arithmetic operators, user input, conversion
2	Text & String Operations	Manipulating and analyzing text data	Reverse a string Count Vowels Mask sensitive info	String slicing, concatenation, methods, loops
3	Decision Making	Writing code that chooses different paths based on conditions	Even-odd checker Compare two strings Range tester	if/elif/else, Boolean logic, comparison ops
4	Repetition & Loops	Repeating actions until a condition is met	Compute factorial Sum numbers 1-N Prime tester	for loops, while loops, range(), iteration
5	Reusable Functions	Packaging logic into callable blocks	Degrees↔radians converter GCD Calculator Custom greeting	def, parameters, return values, modular design
6	List & Array Manipulation	Working with ordered collections	Sort a list Filter mixed-type list Find max value	3*Lists, indexing, appending, sorting, list methods
7	Type Conversion Exercises	Changing data from one form to another	Decimal → binary converter String → integer parser	int(), float(), str(), bin()

```
print(”Max:”, max(lst))
```

7. Type Conversion

```
d = int(input(”Decimal: ”))
print(”Binary:”, bin(d))
print(”Parsed Int:”, int(input(”Number (as string): ”)))
```

Python Starter Pack: Essential Coding Challenges:

This section presents a selection of small questions, each reinforcing the basic Python concept. These issues are chosen for their ability to clarity, management, and to promote active coding and logical thinking. The aim is to promote independent thinking and face many diverse challenges without revealing solutions to solve problems.

Let’s explore a short example and its code snippet:

In-place variable swapping using Python’s tuple unpacking

Input two values

```
a = input(”Enter value for A: ”)
b = input(”Enter value for B: ”)
print(f’Before Swapping: A = {a}, B = {b}”)
```

Swapping using tuple unpacking (no temp variable)

```
a, b = b, a

print(f’After Swapping: A = {a}, B = {b}”)
```

II. Concept practice: Variable assignment, Understanding Pythons tupelo Pack/editing.

Practised concepts: user input (input()), type conversion (int()), conditionalinstructions (if-else), modulo operator (%).

Let’s explore a short example and its code snippet:

1. Variable Assignment

```
age = 25
name = ”ABC”
print(f’Assigned directly: age = {age}, name = ’{name}’”)
```

```
2. User Input
user_name = input(”Enter your name: ”)
print(”Hello”, user_name)
3. Type Conversion
age = int(input(”Enter your age: ”))
4. Conditionals (Adult or Minor)
if age ¿= 18:
print(”You are an Adult.”)
else:
print(”You are a Minor.”)
5. Modulo Operator (Even/Odd check)
num = int(input(”Enter a number to check even or
odd: ”))
if num % 2 == 0:

  print(”Even”)
else:
print(”Odd”)
```

III. Concept practice: function definition, cutting edge of strings, implicit understanding of string immutability. The character ”Y” should not be counted as a vowel.

Practical concepts: function definition, string identification (loop), conditional instructions, string methods (.lower(), in). The function must perform the specified calculation and return the result.

Let’s explore a short example and its code snippet:

```
  Function Definition (from Table 4)
def count_vowels(text):
vowels = ”aeiou”
vowel_count = 0
for char in text: { Looping Through String }
if char.lower() in vowels: { Case-insensitive,
excluding ’y’ }
vowel_count += 1 { Strings are immutable, count
in a new variable }

  return vowel_count { Return final count }
Main Code
user_input = input(”Enter a string: ”)
result = count_vowels(user_input)
print(f”Total vowels (excluding ’Y/y’):”, result)
```

IV. Concept Practice: Functions, Parameters, Conditionals, and Arithmetic Operators in Python

Practical concepts: function definitions, some parameters, conditional statements (if-elif-else), arithmetic operators.

Let’s explore a short example and its code snippet:

```
  Function Definition with Parameters def
calculate(a, b):
{ Conditional Statements }
if a ¿ b:
print(”a is greater than b”)
elif a ¡ b:
print(”b is greater than a”)
else:
print(”a and b are equal”)
{ Arithmetic Operations }
print(”Sum:”, a + b)
print(”Difference:”, a - b)
print(”Product:”, a * b)
if b != 0:
print(”Division:”, a / b)
print(”Remainder (Modulo):”, a % b)

  else:
print(”Division and Modulo not possible (b is 0)”)
Main Code
x = int(input(”Enter first number (a): ”))
y = int(input(”Enter second number (b): ”))
calculate(x, y)
```

V. Concept Practice: Sorting with Control Flow and Iteration in Python

Practice concepts: loops (or in between), conditional statements (for basic cases such as 0), iteration or recursive logic. If the string is ”ASC”, the function must return a list sorted in ascending order. For ”desc”, it returns in descending order. And if ”none” you need to return the original list without modifying it.

Let’s explore a short example and its code snippet:

```
  def sort_list(order, input_list):
{ Conditional logic to decide sorting }
if order.lower() == ”asc”: {asc for ascending}
return sorted(input_list)
```

TABLE II: Practice Essentials – Variables, Input, Conversion, Conditionals, and Modulo in Python

S.No.	Concept	Description	Example Code Snippet	Output Example
2				
1 name = ”Radha” name is ”Radha”	Variable Assignment age is now 25	Assigning values to variables using =	age = 25	
2 input(”Enter name: ”) → user_name = ”Alex”	User Input If input is ”Alex”	Taking input from users using input()	user_name =	
3 age: ”)) → age = 18	Type Conversion If input is 18	Converting input from string to integer using int()	age = int(input(”Enter	

4
print(”Adult”)
else:
print(”Minor”)
based on input
	Conditionals
	Adult or Minor
		Making decisions using if, else	if age ¿= 18:	

5
print(”Even”)
else:
print(”Odd”)
→ Odd
	Modulo Operator (%)
	If input is 7
		Finding remainder to check divisibility or parity (even/odd)	if num % 2 == 0:	

```
elif order.lower() == ”desc”: { desc for descending }
return sorted(input_list, reverse=True)
elif order.lower() == ”none”: {none to preserve the original list}
return input_list
else:
return ”Invalid order type! Use ’asc’, ’desc’, or ’none’.”
Example usage

original = [5, 3, 9, 1, 7]
print(”Original List:”, original)
print(”Ascending Order:”, sort_list(”asc”, original))
print(”Descending Order:”, sort_list(”desc”, original))
print(”No Sorting:”, sort_list(”none”, original))
```

VI. Concept Practice : Temperature Conversion with Functions, Lists, and Conditional Statements in PythonPractical: concepts: function definition, list operations (.sort() or sorted()), conditional statements, string comparisons. The program must request the user a temperature of degrees Celsius.

Let’s explore a short example and its code snippet:

```
1. Function to count vowels in a string
def count_vowels(text):
vowel_count = 0
for char in text:
if char.lower() in ”aeiou”:
vowel_count += 1
return vowel_count
2. Function for basic calculator logic using conditionals and arithmetic
def calculate(a, b):
if a ¿ b:
return a - b
else:
return a + b, a * b, a / b, a % b
3. Function to sort a list based on a given order
```

TABLE III: Key Practice Areas – Python Functions and Conditional String Operations

S.No.	Concept	Description	Example Code Snippet	Key Takeaways

1	Function Definition	Define a reusable block of code that accepts input and returns a result.	`def count_vowels(text):`
			`# logic here`
				Functions improve modularity and readability.

2	Looping Through a String	Use a loop to iterate through each character in a string.	`for char in text:`	Enables character-level operations on strings.
3	String Immutability	Strings cannot be changed after creation; instead, new strings are built when modifying.	Modify using slicing or concatenation, not direct assignment.	Understanding immutability avoids runtime errors.
4	Conditional Instructions	Use if statements to check if a character is a vowel.	`if char in "aeiou":`	Logical checks help filter specific characters.
5	Excluding ’Y’ as Vowel	Ensure the function does not consider ’Y’ or ’y’ a vowel.	`if char in "aeiou":` (do not include ’y’ in vowel list)	Emphasizes precision in conditional logic.
6	String Methods	Use `.lower()` to normalize case and `in` to check character presence.	`if char.lower() in "aeiou":`	`.lower()` simplifies case-sensitive checks.
7	Return Value	The function should return the final count of vowels found.	`return vowel_count`	`return` sends results back to the caller.

TABLE IV: Practical Application – Python Functions and Logical Operations

S.No.	Concept	Explanation	Illustrative Code Example
1	Function Definition	Creating reusable code blocks that perform specific tasks.	`def calculate():`
2	Parameters	Inputs passed into functions to make them dynamic and flexible.	`def calculate(a, b):`

3	Conditional Statements	Using if, elif, and else to control logic flow based on conditions.	`if a > b:`
			`    return a`
			`else:`
			`    return b`

4	Arithmetic Operators	Performing calculations using +, -, *, /, %.	`return a + b, a * b,`
			`a / b, a - b, a % b`

def sort_list(order, input_list):
if order == ”asc”:
return sorted(input_list)

elif order == ”desc”:
return sorted(input_list, reverse=True)
else:
return input_list { return as-is if ”none” }

4. Function to collect temperatures, sort, and classify
def convert_temperature():
temperatures = []
count = int(input(”How many temperatures do you want to enter? ”))
for i in range(count):
temp = float(input(f”Enter temperature {i+1} in Celsius: ”))
temperatures.append(temp)

TABLE V: Practical Application – Python Functions and Logical Operations

S.No.	Concept	Explanation	Illustrative Code Example
1	Function Definition	Creating reusable code blocks that perform specific tasks.	`def calculate():`
2	Parameters	Inputs passed into functions to make them dynamic and flexible.	`def calculate(a, b):`

3	Conditional Statements	Using if, elif, and else to control logic flow based on conditions.	`if a > b:`
			`return a`
			`else:`
			`return b`

4	Arithmetic Operators	Performing calculations using +, -, *, /, %.	`return a + b, a * b, a / b, a - b, a % b`

```
order = input(”Sort order (asc/desc/none): ”).strip().lower()
if order == ”asc”:
temperatures.sort()
elif order == ”desc”:
temperatures.sort(reverse=True)
print(”Classification:”)
for temp in temperatures:
if temp ¡ 20:
print(f”{temp}°C - Cold”)
elif temp ¡ 30:
print(f”{temp}°C - Warm”)
else:
print(f”{temp}°C - Hot”)
def main():
while True:
print(”— Python Concept Programs —”)
print(”1. Count Vowels in a String”)
print(”2. Basic Calculator (Using Conditionals)”)
print(”3. Sort List (asc/desc/none)”)
print(”4. Temperature Handler (Sorting & Classification)”)
print(”5. Exit”)
choice = input(”Enter your choice (1-5): ”)
if choice == ”1”:
text = input(”Enter a string: ”)
print(”Number of vowels:”, count_vowels(text))
elif choice == ”2”:
a = float(input(”Enter first number: ”))
b = float(input(”Enter second number: ”))
result = calculate(a, b)
print(”Result:”, result)
elif choice == ”3”:
lst = input(”Enter list elements separated by space: ”).split()
lst = [int(x) for x in lst]
order = input(”Enter sort order (asc/desc/none): ”).strip().lower()
sorted_list = sort_list(order, lst
print(”Sorted List:”, sorted_list)
elif choice == ”4”:
convert_temperature()
elif choice == ”5”:
print(”Exiting Program. Goodbye!”)
break

   else:
print(”Invalid choice. Please try again.”)
if _name_ == ”_main_”:
main()
```

VII. Concept Practice : Applying Arithmetic and Formulas with User Input in Python

Practical concepts: user input, type conversion (float), arithmetic operations, formula applications. Table Form given below

Let's explore a short example and its code snippet: 1. User Input radius = float(input(”Enter the radius of the circle: ”)) { Type Conversion from string to float } 2. Arithmetic Operations and Formula Applications Calculate the area of the circle area = math.pi * radius * radius Display the result print(f”The area of the circle with radius {radius} is: {area}”) 3. User Input for temperature conversion fahrenheit

TABLE VI: Function-Based List Sorting – ASC, DESC, or None in Python

S.No.	Concept	Description	Example Techniques
1	Looping / Iteration	Repeating operations over elements in a sequence (list, string, etc.)	`for item in list:`, `while`, list comprehensions
2	Conditional Statements	Making decisions based on values such as ”asc”, ”desc”, or ”none”	`if`, `elif`, `else` structures
3	Sorting Mechanisms	Sorting lists based on conditions	`sorted(list)`, `sorted(list, reverse=True)`
4	Preserving Original List	Returning the list as-is when no sorting is required	`return original_list`
5	Function Definition	Wrapping the logic in a function that accepts input and returns output	`def sort_list(order, input_list):`
6	Parameter Usage	Accepting dynamic input values such as order type and list to be sorted	`order: str, input_list: list`
7	Optional: Recursion	An advanced method for iterating through structures or decision trees (if applicable)	(Not required for this example but could be explored in variations)

= float(input(”Enter temperature in Fahrenheit: ”)) { Type Conversion from string to float } 4. Formula Application for Celsius conversion celsius = (fahrenheit - 32) * 5 / 9 Display the result print(f”The temperature in Celsius is: {celsius}”)

VIII. Concept Practice – Filtering Integers from a List Using

Functions and String Operations in Python Practical- concepts: function definition, cutting edge of strings, string

quet, string iteration. This function must return a new list with only integers in its original order.

Let’s explore a short example and its code snippet:

def filter_integers(data):
2. String Checks
filtered_list = []
3. String Iteration and
4. List Construction
for item in data:
if isinstance(item, int): { Check if the item is an integer }
filtered_list.append(item) { Add integer to the new list }
return filtered_list { 5. Return Statement }
Example usage

input_data = [1, ’hello’, 3.5, 2, ’42’, 7, ’world’, 10] { Mixed list of integers and strings }
result = filter_integers(input_data) { Call the function }
print(”Filtered integers:”, result) { Output the result }
Practical concepts: function definition, list identification, type check (isinstance()), list attachments. Despite being small, each problem increases specific concepts and prepares learners with somewhat complicated variations. It is a repetition of the core problem type (e.g., string inversion, factor calculation), but shows different limitations or proposed approaches (e.g., using a compared to loop or recursive versus iterative logic).

This will give you a deeper understanding and expose you to the versatility of Python. It’s not just about solving problems, but also about solving many problems, integrating patterns, and building a mental library of solutions. Therefore, the practice regime of beginners should include reviewing concepts through various problem contexts and researching alternative solutions to the same problem.

Refining Your Problem-Solving Approach: Developing powerful programming knowledge means more than just fighting exercises. A systematic approach is required. Small, manageable component problems create arithmetic thinking habits that clearly implement solutions into pseudocode and rigorously test your work, turning your challenges

TABLE VII: Function-Based Temperature Handling – A Concept Practice in Python

S.No.	Concept	Description	Example Techniques
1	Function Definition	Encapsulating logic into a reusable code block for clarity and reuse	`def convert_temperature():`
2	List Operations	Using .sort() or sorted() to organize temperature values if needed	`temperatures.sort(), sorted(temperatures)`
3	Conditional Statements	Making decisions based on input values or ranges (e.g., cold, warm, hot)	`if temp < 20:,elif temp < 30:,else:`
4	String Comparisons	Handling input types like ”asc”, ”desc” for sorting or interpreting temperature conditions	`if order == "asc":`
5	User Input	Collecting temperature value from the user in degrees Celsius	`temp = int(input("Enter temperature in Celsius: "))`
6	Type Conversion	Converting input string to integer/float for processing	`int(),float()`

into trustworthy, elegant code.

Tackling Complex Problems One Step at a Time: The problem is a conscious process of breaking up large, complex problems into small, independent, manageable sub-problems. Each sub-problem can be solved individually and combined with the solutions to effectively tackle the larger original challenges. This approach is the basis of computer-aided thinking.The advantages of the problem are numerous. It sharpens clarity by better understanding the requirements of the problem and helping you recognize potential edge cases in the early stages. For beginners who find it difficult to juggle syntax, logic, dataflow and exceptions at once, decomposition serves as an important cognitive help. Concentrating on small insulated pieces simultaneously reduces the mental load and becomes overwhelming. This means that the learning process is managed and motivated by progressive victory. Supports time management by assessing and prioritizing tasks more effectively. Additionally, corruption of the problem increases the likelihood that marginal cases will be recognized and planned from the start, thus facilitating thorough troubleshooting.Simplification of complex problems allows small tasks to be distributed among team members, making moving of blocking tasks accessible and collaborative. Each board can be given detailed attention to improve the overall solution quality. For example, you can decompose the construction of simple computers in the implementation of parsing inputs, precedence operators, and individual arithmetic operations. Instead of expecting learners to record implicitly, they should be presented as a central metaskill that will ensure students are gradually moved to complex programming tasks.

III. Conclusion:

True mastery of Python emerges not merely from learning its syntax but from repeatedly applying those constructs to well-focused, bite-sized problems. By tackling exercises that isolate one or two core ideas—whether arithmetic, string manipulation, control flow, loops, data structures, or simple functions—beginners translate abstract concepts into concrete code, uncover gaps in their understanding, and develop the investigative mindset and muscle memory essential for writing correct, readable programs. A curriculum that places hands-on, concept-driven practice at its heart—using theory only as a launchpad for immediate application—will best equip students to progress confidently from fundamental skills to tackling more sophisticated challenges.

IV. References

Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment A thesis submitted to attain the degree of DOCTOR OF SCIENCES of ETH ZURICH (Dr. sc. ETH Zurich) presented by Tobias Kohn Dipl. Math, ETH Zurich born on January

31, 1982 citizen of Endingen (AG) accepted on the recommendation of Prof. Dr. Juraj Hromkovi˘c, examiner Prof. Dr. Bill Manaris, co-examiner Prof. Dr. Thomas R. Gross, co-examiner Prof. Dr. Jürg Gutknecht, co-examiner 2017 Identifying Learning Challenges faced by Novice/Beginner Computer Programming Students: An Action Research Approach Sarita Singh1 1 College of Professional Studies, Northeastern University, USA

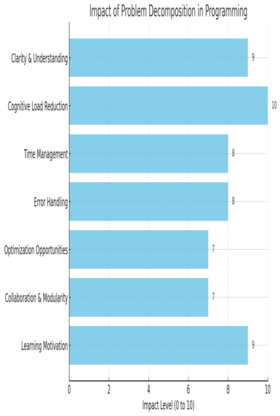

Fig. 1: impact of program decomposition in Programming

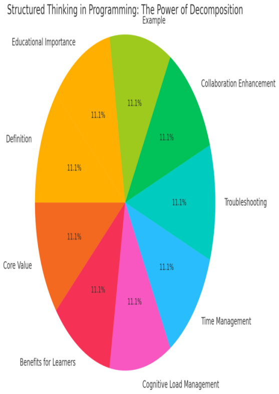

Fig. 2: Structured Thinking in Programming

References

[1] ACM & IEEE. (2021). *Computing Curricula 2020 CC2020: Paradigms for global Computing Education*. ACM & IEEE.

[2] Detienne, F. (1990). Expert Programming Knowledge: A Schema-based Approach. In Hoc, J., Green, T., Samurcay, R., and Gilmore, D. (Eds.), *Psychology of Programming* (pp. 206–222). Academic Press, London.

[3] Chernikova, O., Heitzmann, N., Stadler, M., Holzberger, D., Seidel, T., & Fischer, F. (2020). Simulation-based learning in higher education: A meta-analysis. *Review of Educational Research*, 90(4), 499–541. https://doi.org/10.3102/0034654320933544

[4] Aristeidou, M., Ferguson, R., Perryman, L. A., & Tegama, N. (2021). The roles and value of citizen science: Perceptions of professional educators enrolled on a postgraduate course. *Citizen Science: Theory and Practice*, 6(1), 1–14. https://doi.org/10.5334/CSTP.421

[5] Dhawan, S. (2020). Online learning: A panacea in the time of COVID-19 crisis. *Journal of Educational Technology Systems*, 49(1), 5–22. https://doi.org/10.1177/0047239520934018

[6] European Research Council. (2022, December 7). *Young people engaged for the planet* [Video]. YouTube. https://www.youtube.com/watch?v=QV3ZqvfZ0ow

[7] Kate, V., Bansal, C., Pancholi, C., Patidar, A., Patidar, G., Kitukale, D. Vyanjak: Innovative Video Intercom and Notification System for the Deaf Community.

[8] Goyal, K., & Kumar, S. (2021). Financial literacy: A systematic review and bibliometric analysis. *International Journal of Consumer Studies*, 45(1), 80–105. https://doi.org/10.1111/IJCS.12605