ISSN: 2710-0588

Functional Transformation of Introductory Programming Courses for Non-Computer Majors: From C/C++ to Python

Wei Gao, Mei Li

Department of Computer Science, North China Electric Power University (Baoding), Baoding, China

Abstract

With the widespread application of Python in data processing and machine learning, an increasing number of non-computer majors are shifting their introductory programming courses from C/C++ to Python. This transition is not merely a change of programming language, but a systematic restructuring of course objectives, teaching content, and instructional approaches. Based on teaching practice and from the perspective of instructors, this study analyzes the transformation of course function from training in program construction to the cultivation of computational tool application ability. It further examines how the course mainline shifts from program implementation mechanisms to data-processing-oriented tasks and application contexts, and how this shift leads to the reorganization of teaching content and the formation of a task-driven instructional model. In addition, the new requirements for teachers' knowledge structure and teaching competence under this transformation are discussed. The results show that the introductory programming course for non-computer majors is evolving from a discipline-oriented foundational course into a computational literacy course aimed at developing general-purpose computing ability. Its core focus is moving from language implementation mechanisms to data processing processes. This instructional reconstruction provides an implementable curriculum form and pedagogical pathway for the reform of introductory programming courses for non-computer majors.

Keywords

Introductory programming course; Python teaching; non-computer majors; reconstruction of teaching content; transformation of teaching methods.

1. Introduction

In the context of emerging engineering education and data-driven research paradigms, programming competence has become an essential foundational tool for science and engineering students [1]. Traditionally, introductory programming courses for non-computer majors have adopted C or C++ as the teaching language, with course objectives focusing on fundamental program design and an understanding of computer system principles. With the widespread use of Python in scientific computing, data analysis, and machine learning, an increasing number of disciplines are shifting their introductory programming courses from C/C++ to Python in order to enhance students' ability to solve domain-specific problems using computational methods [2],[3].

However, the change of programming language is not merely a syntactic replacement; rather, it entails a transformation of course function and instructional focus. C/C++-based courses are organized around program structures and language implementation mechanisms, whereas Python-based courses emphasize data processing and rapid modeling. This shift renders the traditional content organization centered on a language knowledge system no longer suitable,

ISSN: 2710-0588

and it also imposes new requirements on instructional approaches and teachers' competence structures.

Existing studies on Python-based teaching reform mainly examine student learning outcomes or curriculum system design [4], while systematic analyses of how teaching content and instructional methods are reconstructed from the perspective of front-line instructors remain relatively limited. To address this gap, this study draws on the teaching practice of introductory programming courses for non-computer majors and investigates the shifting trajectory of instructional focus during the transition from C/C++ to Python. Specifically, it analyzes the transformation of course function, the reconstruction of teaching content, and the change in instructional methods, with the aim of providing a reference for related curriculum reform.

2. Functional Transformation of the Course: From Program Construction to Computational Tool Application

In the C/C++-based instructional system, introductory programming courses are typically regarded as an extension of foundational computer science education. Centered on structured programming, the course is organized around three basic control structures—sequence, selection, and iteration—as well as the modular use of functions. Through stepwise refinement of algorithms, computational processes are implemented in the form of complete programs. In this process, data organization methods such as arrays and structures mainly serve as means of information representation to support program execution. Consequently, the mainline of the course is characterized by the construction of complete programs, and its primary objective is to enable students to transform real-world problems into executable computational procedures.

When the teaching language shifts to Python, its application scenarios change significantly. For non-computer majors, Python is no longer primarily used to construct complete programs from scratch; rather, it functions as a tool for processing experimental data, performing simple modeling, and invoking existing algorithm libraries to solve domain-specific problems. Accordingly, the core objective of the course changes from mastering program design methods to developing the ability to complete data processing and analysis tasks using computational tools. This change in course objectives is first reflected in the differences in typical classroom problems, as shown in Table 1.

Table 1. Comparison of Typical Classroom Problems in C/C++ and Python

Typical Problems in C/C++ Classrooms	Typical Problems in Python Classrooms
How to determine whether a number is prime?	How to use libraries to filter data?
How to use arrays to analyze score distributions?	How to read a CSV file and perform statistical analysis?
How to implement bubble sort using loops?	How to compute the average of experimental data and visualize the results?
How to implement string searching from scratch?	How to process text using built-in functions?
Focus on the execution process of algorithms	Focus on data-processing workflows

The problems in C/C++ classrooms are centered on algorithm implementation, aiming to train students' understanding of control structures and data organization through the construction of complete programs. In contrast, Python classroom problems are organized around data-processing tasks, where programs mainly serve as tools for implementing data-processing workflows. As a result, classroom activities shift from "constructing programs" to "completing tasks", which fundamentally changes the way course content is organized.

ISSN: 2710-0588

The transformation of course function directly reshapes the organization of teaching content. Instead of following the implementation path of complete programs, instruction is organized around the execution process of data-processing tasks, making data acquisition, representation, and analysis the mainline of the course. In this context, programs primarily function as tools for implementing data-processing workflows, and the instructional focus moves from the full implementation of algorithms to the decomposition of data-processing steps and the completion of specific tasks through function encapsulation and library calls. From the perspective of course function, the two approaches exhibit systematic differences in terms of the role of code, the focus of instructional analysis, task function, and competence development, as summarized in Table 2.

Table 2. Structural Differences in Course Function between C/C++ and Python

Dimension	C/C++	Python
Role of code	Object of study	Tool for implementation
Focus of instructional analysis	Program execution mechanisms	Data-processing workflows
Function of tasks	Introducing program construction	Organizing computational processes
Competence development	Ability to construct programs	Ability to use computational tools

The differences shown in Table 2 indicate not only adjustments in teaching content but also a transformation in the nature of the course. In C/C++ courses, code itself is the primary object of instructional analysis, and computational thinking is cultivated through the process of program construction. In Python courses, however, code recedes to the role of a tool for implementing data-processing tasks, and the focus of instructional analysis shifts to the understanding and organization of computational processes.

Therefore, the transition from C/C++ to Python is essentially not a simple replacement of programming languages, but a paradigm shift in introductory programming education—from program-construction-oriented training to computational-tool-oriented competence development.

3. Reconstruction of Teaching Content

In traditional introductory programming courses represented by C/C++, teaching content is typically organized along the implementation path of program construction. The course starts with basic data types and expressions, introduces sequence, selection, and iteration structures, implements typical algorithms using arrays, and achieves modular program design through functions. The connection among different knowledge units is based on the gradual increase in program complexity, and the instructional objective is to enable students to master the essential elements and implementation procedures required for constructing complete programs. In this system, teaching cases mainly focus on algorithm implementation, such as sorting, searching, and simple numerical computation, with emphasis on the complete representation of program execution processes.

When the teaching language shifts to Python, the organization of teaching content changes from being program-construction-oriented to being data-task-oriented. The starting point of the course is no longer the language elements required for building complete programs, but the data-processing requirements embedded in specific tasks. For example, experimental data are obtained through file reading, data are organized using lists or dictionaries, and data processing and result output are implemented through function encapsulation or library calls. In this

ISSN: 2710-0588

process, control structures and functions remain fundamental components of the course; however, their role shifts from constructing program execution flows to organizing data-processing steps.

Data representation becomes a central component of teaching content. In traditional courses, arrays mainly serve as storage structures for algorithm implementation, whereas in Python-based courses, data structures directly correspond to the form of data in tasks, and instructional focus shifts to the relationships among data and their processing methods. For instance, in a task of grade data analysis, student information is typically organized as key–value pairs in the form of "student ID–score". Instruction no longer emphasizes the location of data in memory, but rather the relationships among data: retrieving scores efficiently through student IDs, computing the average of all scores, and counting the number of students within different score ranges. Such tasks treat datasets as the basic processing objects, transforming teaching content from program implementation for individual values to workflow-oriented processing of entire datasets.

The teaching of algorithms also changes accordingly. In C/C++ courses, typical algorithms are usually implemented in full, and the instructional focus lies in understanding the execution steps of algorithms through coding and their correspondence with program structures. In Python courses, however, algorithms often appear in the form of functions or libraries, and the instructional focus shifts to understanding data-processing strategies and the effective use of existing computational resources. Table 3 presents a comparison of the implementation of a grade data-processing task in the two languages.

Table 3. Implementation of a Grade Data-Processing Task in C/C++ and Python

Operation	C/C++	Python

Data storage	struct student
	{ char id[10];
	int score; };
	student s[]={ "2023001", 86,"2023002",92,"2023003",78};
	n=sizeof(s)/sizeof(s[0]);
		scores = {
		"2023001": 86,
		"2023002": 92,
		"2023003": 78
		}

Fast retrieval	for(i=0;i<n;i++)
	if(s[i].id=="2023001")
	cout<<s[i].score;
		scores["2023002"]

Average score calculation	double aver=0;
	for(i=0;i<n;i++)
	aver+=s[i].score;
	aver/=n;
		avg = sum(scores.values()) / len(scores)

Score distribution by range	int result[10]={0};
	for(i=0;i<n;i++)
	if(s[i].score>=90)
	result[0]++;
	else if(s[i].score>=80)
	result[1]++;
	else
	result[2]++;
		result = {">=90": 0, "80-89": 0, "<80": 0}
		for score in scores.values():
		if score >= 90:
		result[">=90"] += 1
		elif score >= 80:
		result["80-89"] += 1
		else:
		result["<80"] += 1

The comparison shows that in C/C++ courses, statistical functions rely on complete program structures and loop control, and the instructional emphasis is placed on the unfolding of algorithm execution processes. In Python courses, by contrast, the same operations can be

ISSN: 2710-0588

completed through direct data organization and built-in functions, and instructional attention shifts from algorithm coding to the formation of data-processing strategies.

On this basis, teaching content further incorporates file operations for real data and result presentation, forming a complete workflow of "data acquisition–data processing–result output" [5] Instruction is no longer aimed at the completion of a single program, but is organized around complete data-processing tasks, and different knowledge units are connected according to the functional stages of data processing. Table 4 illustrates the typical classroom implementation of the three stages in the data-processing workflow and their corresponding instructional implications.

Table 4. Classroom Implementation of the Three Stages of the Data-Processing Workflow and Their Instructional Implications

Operation	Example	Instructional Implication
Data acquisition (file reading)	with open("score.txt") as f: data = f.readlines()	The focus is not on syntax, but on understanding where real data come from and how they are transformed into computable forms.
Data transformation (converting a line of text into a numeric list)	nums = list(map(float, line.split()))	Data must be structured before computation.
Data processing (batch aggregation)	avg = sum(scores) / len(scores)	The focus shifts from operating on a single value to performing computations on a dataset as a whole.
Data processing (conditional filtering and counting failed students)	count = sum(score < 60 for score in scores)	Processing a collection of data based on specified conditions.
Result presentation (formatted output)	print(f""Average score: {avg:.2f}")	Results should be interpretable and readable.
Result presentation (basic visualization: bar chart)	plt.bar(range(len(scores)), scores) plt.xlabel("Student Index") plt.ylabel("Score") plt.title("Scores") plt.show()	The goal of computation is data analysis rather than merely producing a runnable program.

Unlike the traditional organization of knowledge around program structures, teaching content in Python courses is structured around complete data-processing tasks. Data acquisition, data transformation, data computation, and result presentation form a continuous learning process, and the programming language is no longer the primary object of analysis but a tool for accomplishing computational tasks.

Therefore, the organization of teaching content shifts from a program-implementation-path orientation to a data-processing-workflow orientation. The connections among knowledge units are reflected in the progressive expansion of data-processing functions, and the role of the programming language in the course changes into a tool for accomplishing specific tasks. This transformation indicates that, under the functional transformation of the course, teaching content has undergone a structural reconstruction from an algorithm-implementation orientation to a data-task orientation.

ISSN: 2710-0588

4. Transformation of Instructional Methods

In traditional C/C++-based introductory programming courses, instruction is usually organized according to the sequence of the language knowledge system. Basic data types and sequential structures are introduced first, followed by selection and iteration structures and arrays, and finally the integrated application of knowledge is achieved through comprehensive program design. Classroom examples and laboratory practice correspond to different knowledge points, and during the comprehensive exercise stage students are required to write complete programs from scratch, so that the various knowledge units are connected in the final process of program construction. A key feature of this approach is the temporal separation between knowledge instruction and comprehensive application: each practice task is independent, and the instructional process proceeds in stages from knowledge explanation to program construction.

In data-task-oriented Python instruction, teaching activities are continuously organized around the same data-processing task rather than through separate comprehensive exercises. The task is accomplished through the progressive extension of a single program across multiple sessions. For example, in the teaching of grade data processing, students first implement file reading based on a given program framework so that the data are presented in the form of a list. They then add a function for calculating the average score, further extend the program to count the number of failing students, and finally incorporate result visualization into the same program. Each practice activity builds on the previous version of the program, and the learning process is thus characterized by the continuous refinement of a single program.

In this instructional process, the way knowledge is introduced also changes. Relevant syntax and functions are no longer explained systematically in advance; instead, they are introduced on demand during task implementation, so that the learning sequence of knowledge is aligned with the data-processing workflow. The focus of classroom explanation shifts from line-by-line analysis of program execution to a holistic explanation of functional modules. By clarifying the relationships among data acquisition, data processing, and result presentation, students are guided to understand the correspondence between program structure and task implementation.

The goal of practical activities changes accordingly, from "independently completing a specified program" to "extending the functionality of an existing program". Students' primary programming activity is no longer repeatedly constructing programs from scratch, but reading, understanding, and modifying existing code to meet new processing requirements, which makes the programming process closer to real-world data-processing practices. Consequently, the instructional approach shifts from the linear model of "knowledge instruction followed by comprehensive application" to a process-oriented model of gradual construction during task implementation. Classroom teaching and laboratory practice form a continuous learning process centered on a unified data-processing workflow. A comparison of instructional implementation in C/C++ and Python courses is presented in Table 5.

The differences shown in Table 5 indicate that in traditional C/C++ courses, teaching activities are organized around discrete knowledge points, and classroom instruction and practice separately contribute to the development of program-construction ability. In Python courses, however, teaching activities are organized around complete data-processing tasks, in which knowledge learning, program implementation, and practical operation are integrated into the same process. The instructional approach thus changes from a staged progression to a continuous construction process, and students' programming activities shift from program-construction training to program comprehension and functional extension.

ISSN: 2710-0588

Table 5. Comparison of Instructional Implementation in C/C++ and Python Courses

Dimension	C/C++ Introductory Programming Course	Python Introductory Programming Course
Organization of teaching content	Arranged as a comprehensive task after relevant knowledge units	Progressively developed as a data-processing task over sessions
Starting point of classroom instruction	Constructing complete programs from syntax rules and program structures	Introducing tasks from real data or an existing program framework
Program implementation	Writing complete programs from scratch	Incrementally extending an existing program
Mode of knowledge introduction	Each knowledge unit corresponds to an independent example program	New knowledge introduced through the extension of the same program
Focus of classroom explanation	Statement execution, loop control, and index changes	Decomposition of data-processing steps and composition of functional modules
Form of laboratory practice	Each lab requires the completion of an independent program	Multiple labs progressively refine the same program
Criterion for task completion	Program runs correctly and produces computational results	Data-processing workflow is complete and yields interpretable results
Characteristics of students' programming	Program-construction training	Program comprehension and functional extension

This transformation shows that, on the basis of the functional transformation of the course and the reconstruction of teaching content, classroom instruction moves from a linear model of "teaching knowledge first and then applying it comprehensively" to a process-oriented model in which knowledge is introduced and functions are progressively extended during task implementation. In this sense, instructional methods shift from a knowledge-system-oriented approach to a task-oriented approach.

5. New Requirements for the Teacher Competence Structure

With the shift of the course orientation from program construction to data-task-based learning, the organizational unit of teaching activities changes from discrete knowledge points to complete data-processing tasks, and the instructional process moves from staged progression to continuous construction. This transformation imposes new requirements on the competence structure of teachers [6]. In traditional C/C++ courses, teachers' primary responsibility is to organize teaching content according to the language knowledge system, and classroom instruction focuses on program structures and algorithm implementation. Their core competence is reflected in the analysis of program execution mechanisms and the explanation of typical algorithmic implementation paths.

In data-task-oriented Python instruction, the course is organized around data-processing tasks. Teachers are required to extract computable tasks from authentic problem contexts and arrange the sequence of knowledge introduction according to the task implementation process, so that syntax rules, data structures, and function calls correspond to the data-processing workflow. This requires teachers to reconstruct teaching content with tasks as the mainline, rather than simply following the existing knowledge system.

ISSN: 2710-0588

The form of example programs also changes accordingly. In traditional courses, example programs are usually verification-oriented code designed for individual knowledge points. In data-task-oriented courses, however, example programs become continuous programs that run across multiple sessions, and the complete task is accomplished through the progressive addition of functional modules. This requires teachers to design programs in a modular manner and extend their functionality while maintaining structural stability, so that the program can both support knowledge introduction and reflect the overall structure of the data-processing workflow.

In practical teaching, students' primary programming activity shifts from constructing programs from scratch to understanding and modifying existing programs. Correspondingly, teachers' classroom guidance moves from the coding process to program structure analysis and functional extension. Teachers therefore need the ability to guide students in reading code, locating functional modules, and analyzing the relationships among data-processing steps, making programming activities closer to real-world computational practices.

The introduction of real data establishes connections between course content and students' disciplinary backgrounds. Teaching tasks are no longer derived from abstract algorithmic problems but from concrete data-processing needs. Consequently, teachers need a certain level of data-processing experience in order to select representative task scenarios and design appropriate data organization methods and processing workflows, so that programming learning can serve the solution of domain-specific problems.

Therefore, teachers' instructional competence shifts from a language-knowledge-oriented structure to an integrated competence that combines task design, program organization, and data application. Its core lies in the holistic construction of teaching content and teaching activities with data-processing tasks as the mainline.

6. Conclusion

During the transition of introductory programming courses from C/C++ to Python, the course mainline shifts from program construction to data-processing tasks, and the course function changes from training in program design to the cultivation of computational tool application ability. Teaching content is reconstructed from a program-implementation-path orientation to a data-processing-workflow orientation, and instructional implementation changes from comprehensive program training to task-driven gradual construction. Students' programming activities move from constructing programs from scratch to understanding and extending existing programs, while the competence structure of teachers shifts toward an integration of task design, program organization, and data application. This transformation indicates that organizing instruction around data-processing tasks enables the reconstruction of the curriculum form of introductory programming courses for non-computer majors and provides an implementable pathway for their pedagogical reform.

Acknowledgments

Supported by 'the Fundamental Research Funds for the Central Universities (2023MS136)'

References

[1] Wing, J. M. (2006). Computational thinking. Communications of the ACM, 49(3), 33–35. https://doi.org/10.1145/1118178.1118215

[2] Guo, P. . (2014). Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities. Communications of the ACM, 57(10), 70–77. https://doi.org/10.1145/2632856

ISSN: 2710-0588

[3] Song, T., Huang, T., & Li, X. (2016). Python language: An ideal choice for teaching reform in programming courses. China University Teaching, (2), 42–47.

[4] DIBo & WANG Xiao-dan. (2014). Object-orientedprogrammingcourse teaching based on Python language. ComputerEngineering & Science, 36 (S1), 122-125.

[5] Finzer, W. (2013). The data science education dilemma. Technology Innovations in Statistics Education, 7(2). https://doi.org/10.5070/T572013891

[6] Shulman, L. S. (1986). Those who understand: Knowledge growth in teaching. Educational Researcher, 15(2), 4–14. https://doi.org/10.3102/0013189X01500200