DOI: 10.66069/ojspub.1137260810

AI Intelligent Programming Teaching Practice for Liberal Arts Students—Taking the Law Major as an Example

Weiwei Li[1], Yu Sheng[2]

[1,2]School of Digital and Intelligent Law (School of Intellectual Property), Shanghai University of Political Science and Law, Shanghai, China

[1]weiweili_817@163.com, [2]shengyu@shupl.edu.cn

Abstract: ***The rapid development of artificial intelligence technology has placed higher demands on the information literacy of humanities students, and how to effectively enhance their AI application skills has become a key issue in the development of New Liberal Arts Majors. Taking the law major as an example, this paper draws on teaching practices and pedagogical research to construct a three-pronged AI programming teaching framework: "AI-Empowered Practice—Practical Task-Driven Learning—Introduction of professional scene" The research is carried out from three dimensions: reconstruction of teaching content, innovation of teaching methods and design of experimental cases. A modular teaching scheme with professional embeddedness is developed, and the effectiveness of the model is verified through teaching practice, aiming to provide reference for the cultivation of liberal arts talents in the new era.***

Keywords: AI intelligent programming, Curriculum reform, Program design.

1. Introduction

With the rapid development of artificial intelligence technology and the accelerated pace of societal informatization, enhancing information literacy has become a critical educational goal. Both the Ministry of Education and the Shanghai Municipal Education Commission have been vigorously advocating for the development of the "New Liberal Arts" encouraging the exploration of deep integration between new technologies and disciplinary instruction, advancing classroom teaching reform, and fostering talent that transcends existing professional and disciplinary boundaries. This initiative aims to revitalize traditional liberal arts majors, curricula, and talent development models, thereby cultivating new liberal arts professionals with high professional competence, refined academic abilities, strong comprehensive capabilities, a creative outlook, and proficiency in interdisciplinary collaboration.

For law students in higher education, legal professionals in the new era must not only possess a noble "spirit of the law" but also be equipped with efficient "legal technology" AI programming instruction driven by legal scenarios can significantly reduce technical anxiety among liberal arts students, enhance their computational thinking and ability to formally express legal problems, and provide a replicable methodological framework for interdisciplinary talent development.

2. The Dilemma of Legal Education

Faced with the surging wave of artificial intelligence technology, legal education has exhibited two extreme responses: first, "technological nihilism" which clings to traditional doctrinal paradigms and views technology as a mere instrumental appendage; second, "technical instrumentalism" which offers standalone computer science courses that, however, devolve into mere formal training due to their detachment from the professional context. Neither of these approaches addresses the core contradiction: legal education aims to cultivate not software engineers, but "technical legal persons" equipped with technical understanding and the ability to formalize problems [1].

A deeper issue lies in the fact that law, as a quintessential humanities discipline, has a knowledge system centered on normative analysis, value judgments, and interpretive reasoning—which fundamentally differs in cognitive paradigm from the logical deduction, algorithmic thinking, and formalized expression upon which programming education relies. The "math anxiety" and "technical fear" commonly found among liberal arts students further exacerbate the barriers to accepting programming instruction. Therefore, how to construct an AI programming teaching system tailored to the cognitive characteristics of liberal arts students has become a key issue in legal education reform [2].

3. Theoretical basis: Analysis of the adaptability Between Law Students' Cognitive Characteristics and Programming Learning

3.1 Cognitive Structural Characteristics of Law Students

Based on research findings in cognitive psychology and subject-specific pedagogy, the cognitive structure of law students tends to organize knowledge primarily through semantic networks rather than formal symbols. The "concept–norm–case" triadic cognitive structure cultivated through long-term legal education training makes students more adept at processing natural language information laden with meaning, while formal operations involving abstract symbols impose a natural cognitive burden. Concepts in programming languages — such as variables, functions, and loops — are easily perceived as meaningless symbolic operations when lacking semantic anchoring. Law students' thinking places greater emphasis on "analyzing specific problems in specific

contexts" and legal reasoning is always embedded in specific social and value contexts. This highly contextualized thinking habit contrasts with the decontextualized, generalized abstract thinking required by programming. When faced with programming tasks, liberal arts students often find it difficult to abstract specific legal problems into computable formal models [3].

3.2 Theoretical Foundations of Adaptive Instruction

In traditional computer programming language curricula, teaching focuses on imparting knowledge of syntax and developing proficiency in hand-coding as core educational objectives. Course content typically explains syntax rules line by line at the statement level, relying on a large number of homogeneous, repetitive coding exercises to solidify students' foundational skills; assessments generally take the form of closed-book written exams without the support of intelligent tools, with the quality of students' independently handwritten code serving as the basis for evaluation. The overall teaching system emphasizes reinforcing learners' memorization of syntax rules and their practical proficiency in purely manual coding.

With the advent of an education era empowered by large AI models, the philosophy of programming language instruction has undergone fundamental shifts across three dimensions: instructional objectives, curriculum design, and the cultivation of comprehensive competencies. In this new era, teaching places computational thinking and programming logic at the core. The classroom curriculum now includes instruction on the standardized use of AI programming tools, guiding students to leverage these tools to handle mechanical and repetitive programming tasks such as basic code generation and program debugging; Consequently, the focus of instruction shifts to training in higher-order skills, such as breaking down real-world problems, designing business logic architectures, verifying generated code, and optimizing performance. Through this process, students gain an understanding of the language's fundamental syntax rules. Assessment is conducted through project-based evaluations, emphasizing learners' practical ability to collaborate with AI to solve engineering problems, as well as their capacity to discern and control the appropriate and compliant use of AI programming tools.

For law students, AI programming instruction must place greater emphasis on the fact that knowledge is generated within specific contexts, and learning should be embedded in real-world practical scenarios. Integrating programming instruction into legal contexts allows students to leverage their existing professional knowledge as "cognitive scaffolding" thereby reducing the difficulty of understanding abstract concepts. In instructional design, unnecessary syntactic details should be minimized, while guidance on problem-solving thought processes should be increased. For law students, this means shielding them from the complexity of underlying technologies and focusing on training in higher-order thinking. At the same time, designing "low-barrier, high-achievement" learning tasks to gradually build law students' technical self-efficacy is key to overcoming tech anxiety.

4. Developing a "Thinking"-Oriented AI Programming Course for Law Students

Taking into account the professional characteristics of law students, this paper explores course content tailored to their needs, guided by "thinking" and utilizing human-machine collaborative programming as its research method. It constructs a three-dimensional instructional framework model — "AI-Empowered Practice, Actual Combat Task Traction, Professional Scene Embedding", to achieve effective transformation from technical knowledge to professional competence and ultimately to thinking-oriented skills, thereby better assisting law students in adapting to the developmental demands of the AI era.

4.1 Human-Machine Collaborative Thinking

Against the backdrop of generative AI reshaping the technological ecosystem, the teaching of computer programming languages to liberal arts students urgently requires a paradigm shift: the focus should shift from the traditional imparting of syntactic skills to a new teaching framework centered on cultivating higher-order thinking skills. Specifically, instructional objectives should focus on guiding students to construct a cognitive framework for "human-machine collaborative programming" systematically master the methodological framework for code generation, debugging, parsing, and iterative optimization using large language models, and, with the support of AI tools, independently complete end-to-end development practices encompassing requirements analysis, solution design, and code implementation.

At the instructional design level, a three-dimensional, interconnected practical mechanism — "AI-Empowered, Task-Driven, Scenario-Embedded" — should be established: AI tools empower programming practice, real-world tasks drive the learning process, and professional scenarios anchor the application context. Within this framework, students can solidify their foundational knowledge of programming syntax and computational theory while addressing real-world problems, thereby achieving a deep transfer of learning from the tool-operational level to the cognitive-thinking level.

In the teaching implementation phase, large language models can be used to construct tailored teaching cases. These cases are broken down and explained in layers, embedding the teaching of basic programming language syntax within code modification, debugging, and program iteration. Through the hands-on process of troubleshooting and correcting errors, students can progressively absorb and internalize key syntactic concepts, which are then consolidated and deepened through practical training in program functionality optimization, this helps students master the operational model of human-computer collaborative programming and develop a standardized mindset for collaborative development.

4.2 Teaching Practice

Next, using the "Flappy Bird" arcade game as an example, we will provide a detailed explanation of the specific implementation of the teaching approach.

Experiment Objectives:

- Master AI prompt engineering methods for visualization projects. Be able to construct structured, precise AI prompts to effectively guide generative AI in producing visualization solutions, code frameworks, and optimization recommendations that align with design intent.

- Master core Python syntax and program control structures. Understand the basic syntax conventions of the Python language and master the fundamental application of functions and loop-based control structures.

- Develop a mindset for AI-assisted programming and independently complete creative visualization projects. Understand the working paradigm of human-machine collaborative programming and possess the ability to use AI for code generation, debugging, analysis, and iterative optimization; be able to independently complete the entire process—from requirements analysis and solution design to code implementation—with AI assistance.

Procedure:

1) Basic function description of the game

a) Control the bird's jumps using the spacebar or mouse clicks; use gravity physics simulation to make the bird rise and fall; avoid the continuously moving green pipes; and accumulate points by successfully passing through the pipes.

b) The program includes customizable difficulty settings and supports adjustments to window size, bird size, gravity acceleration, pipe spacing, and pipe movement speed to accommodate different play difficulties.

c) The game features a dual-victory mechanism: players can either clear the level by passing through 10 pipes or achieve a survival victory by staying alive for 30 seconds. The score, survival time, and high score are displayed in real time.

2) Generate Prompt

Using the introduction to the game features described above, guide students to organize and create a prompt framework, and use a large language model to generate a preliminary version of the game program.

3) Through parameter debugging, you can intuitively observe how parameter changes affect gameplay and strengthen your understanding of the logic behind the code's execution.

Task 1: Based on your own proficiency and preferences for playing the game, independently debug, adapt, and optimize the parameter settings to intuitively observe how parameter changes affect the program's performance, thereby deepening your understanding of the logic behind code parameters. Part of the parameter code is shown below.

```
# -------- Bird Settings --------
 BIRD_R = 15       # Bird radius (reference range: 8–25; the larger the value, the more likely collisions occur)
 # -------- Physics Parameters --------
 GRAVITY = 0.5     # Acceleration due to gravity (Reference range: 0.2–1.5; the higher the value, the faster the bird falls)
 JUMP_POWER = -10     # Jump power (Reference range: -5 to -18; the larger the absolute value, the higher the jump)
 # -------- Pipe Settings --------
 PIPE_W = 60          # Pipe width (Reference range: 40–100)
 PIPE_GAP = 200       # Pipe gap (Reference range: 100–250; the smaller the value, the harder the level)
 PIPE_SPEED = 4          # Pipe movement speed (Reference range: 1–6; the faster it is, the harder the game)
 PIPE_INTERVAL = 120       # Pipe generation interval (recommended range: 60–200; the lower the value, the more pipes appear)
# ==================
# Victory Conditions
#==================
 WIN_SCORE = 10             # Winning Score: Pass through 10 pipes to win (range: 5–30)
 WIN_TIME = 30              # Survival time: Survive for 30 seconds to win (range: 10–60)
 SHOW_TIMER = True       # Whether to display the timer (True = Show, False = Hide)
```

Task 2: Compare the syntactic differences between "while" and "for" loops, consider which structure is more appropriate here, adjust the number of flowers and clouds, as well as the size and position of the clouds, and observe how changes in these parameters affect the game's background.

- Modify the number of flowers:

For the "for" loop, change the number inside the "range()"brackets to specify the number of flowers to draw;

```
for i in range(10):
   pass
# To draw 10 flowers, change the number to range(10)
```

For the "while" loop, simply modify the termination condition of the counter;

```
while count < 10:
   pass
# To draw 10 flowers, change the condition to count < 10
```

- Modifying the number, coordinates, and size of clouds:

Clouds are stored in the "cloud_positions" list. The total number of clouds equals the number of tuples in the list. You can directly modify the tuple data in the "cloud_positions" list, or add or remove tuples to achieve the desired cloud display effect.

For example:

To modify the data for one cloud: Simply select a tuple (x-coordinate, y-coordinate, scale) from the list and modify it.

```
cloud_positions = [ (50, 80, 1), (350, 100, 0.8), (100, 160, 0.9),
```

(211, 150, 5)]

4) Detailed Explanation of Syntax Concepts

a) while Loop Structure:

Game Scenario: Draw 6 decorative flowers using random coordinates generated within a given range.

```
count = 0          # Initialize the counter
while count < 6:    # Condition: count is less than 6 flowers
    x = random.randint(6, W-6)      # Generate a random x-coordinate for the flower; W ensures the coordinate falls within the window width to prevent it from extending beyond the interface
    y = random.randint(H-70, H-15)    # Generate a random y-coordinate for the flower; H is fixed above the game ground to fit the screen layout
    self.draw_flower(x, y)    # Call the custom drawing method to draw a flower pattern at the random coordinates
    count += 1    # Increment the counter by 1 to track the total number of draws, gradually approaching the loop termination condition
```

b) For loop structure:

Game Scene: Draw 6 decorative flowers using random coordinates generated within a specified range.

```
for i in range(6):
    x = random.randint(6, W-6)
    y = random.randint(H-70, H-15)
    self.draw_flower(x, y)
```

Game Scene: Use a "for" loop to iterate through the cloud data and draw clouds at 5 different positions. The data for each position consists of the x-coordinate, y-coordinate, and cloud size.

```
cloud_positions = [ (50, 80, 1), (200, 40, 1.2), (350, 100, 0.8), (100, 160, 0.9), (300, 200, 1.1) ]
for x, y, size in cloud_positions:
    self.draw_cloud(x, y, size)
```

Task 3: In what other parts of the game are loop structures used?

Collision detection loop: Iterates through all pipes, checking one by one whether the bird has come into contact with a pipe. As soon as the bird collides with the body or bottom of a pipe, the game is immediately declared over, ensuring timely and accurate collision feedback.

Score Detection Loop: Iterates through all pipes to check whether the bird has successfully passed through the gaps between them. Each pipe is scored only once; when the bird flies past the right side of a pipe, the score increases, accompanied by a score animation.

Task 4: Create your own prompt to appropriately optimize the game's interface and functionality, and add a visual parameter tuning panel, as shown in Figure 1.

Interface Optimization Goals:

- Enhanced Animation: Have clouds move slowly from side to side to bring the background to life and enhance the sense of movement;
- Multiple Skin Options: Customize the color schemes for the bird, pipes, and flowers, and add a button to switch themes;
- Progress Bar: Display the progress toward completing the level;
- Dynamic Updates: Display the current score, high score, number of plays, and game timer in real time.

Functional Optimization Directions:

- Key Bindings Expansion: Add ESC to pause the game and P to pause/resume, expanding control options;
- Mode Selection: Add a pop-up window at the start of the game, offering a choice between Time-Limited Survival Mode and Score-Based Clear Mode;
- Difficulty Selection: Adjustable difficulty levels: Easy, Normal, Hard, and Hell;
- Visual Parameter Panel: Add sliders to adjust parameters such as gravity, gaps, and pipe speed in real time.

Figure 1: Added parameter adjustment interface

Task 5: Create your own prompt to further iterate and optimize the game's interface and functionality; Add a legal knowledge base challenge panel, as shown in Figure 2.

Legal Challenge Mechanism:

- Before the game begins, players must answer questions (from a four-level question bank: Beginner, Intermediate, Advanced, and Expert).
- Answering 2 consecutive questions correctly grants 1 game play session.

- An incorrect answer resets the consecutive correct answer count, incentivizing players to continue learning legal knowledge.

Figure 2: Added Legal Knowledge Quiz Control Interface

Task 6: Consider what new looping scenarios have been introduced after adding the legal knowledge challenge panel.

4.3 Experimental Case Analysis

This experiment uses the Flappy Bird visualization mini-game as a teaching tool, leveraging large AI models to conduct human-machine collaborative programming practice. It focuses on the core syntax of for and while loops in Python, achieving a deep integration of syntactic knowledge and practical application, aligning with the educational goal of cultivating advanced programming thinking in the AI era. The entire experiment follows the teaching process of "AI generation, parameter tuning, syntax study, and iterative optimization".

In the initial stage of the experiment, a large AI model is used to generate the complete Flappy Bird game code with a single click. The program features full functionality, including gravity simulation, moving pipes, score tracking, and a dual-clear mechanism, while also providing customizable game parameters. This offers students a mature, visual experimental platform, eliminating the inefficient learning process of repeatedly writing basic code. Students then proceed to independently fine-tune parameters. By adjusting the bird's attributes, physical parameters, pipe configurations, and level-clear conditions, they gain an intuitive understanding of how parameter changes affect game performance and begin to develop a logical understanding of code parameters.

The core instructional component of this experiment focuses on comparing and learning two types of loop structures. Using the drawing of flowers and clouds in a game-themed scene as a starting point, students break down the syntax, logic, counting methods, and termination rules of while loops and for loops. By adjusting the number of iterations, as well as the coordinates and size parameters of the clouds, students gain hands-on experience comparing the core differences between the two loop types. They clarify the boundaries of applicability—where "while" loops are suitable for an unspecified number of iterations and "for" loops are suitable for a fixed number of iterations and data iteration scenarios—and experience, understand, and learn the application of loop structures through debugging exercises and gameplay.

Finally, students use their newly acquired knowledge to iteratively optimize the game, implementing interface enhancements such as upgraded animations, theme skin switching, and level completion progress displays, as well as functional upgrades including button customization, difficulty levels, and a visual parameter adjustment panel. Building on this foundation and incorporating the unique characteristics of the law program, a legal quiz knowledge base module is added, allowing students to gain practical experience with the syntax and logic of loop structures through these functional expansions. The entire experiment leverages AI to lower the barrier to entry for basic coding, integrating syntax learning into the entire process of project debugging and optimization. This effectively cultivates students' human-machine collaborative programming skills, as well as higher-order thinking skills such as problem decomposition and code optimization.

4.4 Teaching Method Analysis

Amid the educational transformation driven by large AI models, the core of programming instruction for law students has shifted from the teaching of syntactic knowledge to the cultivation of advanced comprehensive thinking: On one hand, instructors guide students to become familiar with human-machine collaborative programming models and master practical skills in using AI tools for code generation, debugging, code analysis, and iterative optimization; on the other hand, by leveraging AI to support the entire software development process, instructors hone students' abilities in requirements analysis and solution design, enabling them to learn fundamental syntax rules within real-world project scenarios. Ultimately, this forms a practical course implementation system powered by AI technology, centered on projects, and underpinned by professional theory.

1) "Scaffolding" Teaching Process for Human-Machine Collaboration

The experiment establishes a four-stage progressive pathway: "AI Generation, Parameter Debugging, Syntax Study, Iterative Optimization" In the initial stage, AI generates complete game code with a single click, eliminating the inefficient process of repeatedly writing basic code found in traditional programming instruction and enabling students to quickly obtain a runnable experimental platform. This design aligns with Vygotsky's "Zone of Proximal Development" theory, in which AI serves as a "more capable guider" providing cognitive scaffolding that allows students to focus their limited cognitive resources on higher-order thinking activities—such as understanding parameter logic and conducting comparative syntactic analysis—rather than expending them on lower-order tasks like syntax correction.

2) A Comparative Inquiry Approach to In-Depth Syntax Learning

The experiment uses game background rendering as a starting point and centers on structured comparative learning between "while" loops and "for" loops. By modifying variables such as the number of iterations and coordinate parameters, students independently discovered the fundamental differences

between the two loop types in terms of counting methods, termination rules, and applicable scenarios through a "debug, observe, induce" inquiry cycle. This "comparative insight" strategy goes beyond mere syntax memorization, helping students construct "conditional knowledge"—not only knowing how to use loops, but also clearly understanding when to use which type of loop—thereby achieving a transition from procedural knowledge to strategic knowledge.

3) Cognitive Progression Design from Concrete to Abstract

Instruction follows the cognitive development path of "Perception → Understanding → Application → Creation": Students first gain concrete experience by adjusting parameters such as the bird's gravity and pipe spacing, establishing a cognitive connection between the code and the game's visual outcomes; they then delve deeper into dissecting the syntax of loops to construct their conceptual understanding; finally, based on what they have learned, they expand functionality by upgrading interface animations, implementing difficulty levels, and creating a visual parameter adjustment panel. This progressive scaffolding design gives abstract loop syntax "vitality" within real-world problem contexts, effectively promoting the transfer and application of knowledge.

5. Summary

Targeting law students, this paper explores innovative approaches to curriculum reform in the context of artificial intelligence from multiple dimensions, including the optimization of teaching content, the innovation of teaching methods, and the design of experimental case studies. It shifts the teaching objective from "writing code" to "AI-assisted programming" with a focus on cultivating students' higher-order thinking skills, such as problem decomposition, parameter tuning, and code optimization. Through interaction with AI, students learn to articulate precise requirements, understand generated code, and debug, iterate, and optimize—which are precisely the core competencies of programming education in the AI era: not competing with AI in coding speed, but rather the ability to harness AI tools to solve problems. Through a restructured three-dimensional interactive mechanism of "AI Empowerment, Task-Driven, Scenario Integration" students are guided to develop proactive problem-solving habits and cultivate rigorous computational thinking. This further enhances law students' scientific thinking and digital literacy, helping to cultivate multidisciplinary legal professionals equipped to meet the demands of digital and intelligent rule-of-law development.

References

[1] Fang Liang, Yu Muyun. Reform and Persistence: Legal Education in the Era of Generative Artificial Intelligence [J]. Journal of Anhui University of Technology (Social Science Edition), 2024, 41(4): 77-79.

[2] Liu Yanhong. "Three Questions of the Era" in AI Jurisprudence [J]. Dongfang Jurisprudence, 2021, (05): 4-9

[3] Zhou Aoying, Jin Cheqing, Wang Guoren, et al. Introduction to Computational Thinking [M]. Higher Education Press, 2020:45-62

[4] Papadakis S. Evaluating a game-development approach to teach introductory programming concepts in secondary education[J]. International Journal of Technology Enhanced Learning, 2020, 12(2): 127.

[5] Ding Shiqiang, Wang Pingsheng, et al. Research on Project-based Teaching for the Development of Computational-Thinking Competency[J]. Modern Educational Technology, 2020(9): 49-55.