think2code Teacher Guide

Visual Programming for Python Education

Table of Contents

Introduction 4
What is think2code? 4
Educational Philosophy 4
Why Flowcharts for Programming Education? 5
What Makes think2code Different? 6
Educational Benefits 7
Teaching with think2code 9
Instructional Philosophy 9
Integration into Curriculum 10
Pacing Recommendations 10
Pedagogical Strategies 12
Demonstration Best Practices 15
Formative Assessment Techniques 16
Managing Common Classroom Scenarios 17
Vocabulary Development 18
Cross-Curricular Connections 19
Transitioning to Text-Based Python 20
Supporting Diverse Learners 22
Assessment Alignment 24
Pedagogical Approach 27
Constructivist Learning Framework 27
Cognitive Load Theory Application 28
Dual Coding Theory 29
Scaffolded Learning Progression 30
Metacognitive Development 31
Error as Learning Opportunity 32

Differentiation Through Universal Design 33
Assessment for Learning (AfL) 34
Social Constructivism & Collaborative Learning 35
Transfer of Learning 36
Equity and Inclusion 37
Building Computational Identity 38

Introduction

What is think2code?

think2code is a browser-based visual programming environment that seamlessly bridges the gap between flowchart-based algorithm design and executable Python code. Unlike traditional flowcharting tools that produce static diagrams, think2code creates executable flowcharts that run in real-time while simultaneously generating authentic Python code.

The Core Innovation: Students design algorithms using familiar flowchart symbols (rectangles, diamonds, parallelograms), connect them visually, and immediately see both:

1. The flowchart executing step-by-step with visual highlighting
2. The equivalent Python code being generated automatically

This dual representation allows students to think algorithmically before worrying about syntax, while still learning real programming constructs that translate directly to Python.

Educational Philosophy

think2code is built on the principle that computational thinking precedes coding syntax. Many students struggle with programming not because they can't think logically, but because they're simultaneously wrestling with:

Abstract logic (what should happen when?)

Concrete syntax (how do I write this in code?)

Debugging (where did I go wrong?)

think2code separates these concerns by letting students focus on logic first, syntax second:

1. Design Phase: Build the algorithm visually (low cognitive load)
2. Testing Phase: Watch it execute with visual feedback
3. Connection Phase: See the Python equivalent emerge automatically
4. Understanding Phase: Recognize patterns between visual and textual representations

Why Flowcharts for Programming Education?

Research-Based Benefits:

Concreteness Fading: Moving from concrete (visual blocks) to abstract (text code) is a proven learning progression. Flowcharts serve as the intermediate representation between real-world problems and abstract code.

Cognitive Load Management:

Visual programming reduces working memory demands by:

Eliminating syntax errors during the learning phase

Making control flow physically visible (arrows show execution paths)

Providing immediate visual feedback during execution

Error Prevention:

Many common programming errors simply cannot occur:

No indentation errors

No misspelled keywords

Connection logic prevents impossible control flows

Executive Function Support:

The visual-spatial nature supports students with:

Dyslexia (less reliance on text processing)

ADHD (tangible, manipulable objects maintain focus)

Language processing difficulties (visual symbols transcend language barriers)

What Makes think2code Different?

Compared to Block-Based Tools (Scratch, Blockly):

✅ Uses standard flowchart notation (transferable to CS exams and industry)

✅ Generates real Python code (not blocks-only)

✅ Professional visual language (prepares for technical documentation)

✅ Focused on algorithms over multimedia (academic rigor) Compared to Traditional Flowcharting Tools (draw.io, Lucidchart):

✅ Diagrams are executable (not just documentation)

✅ Real-time debugging with step-through execution

✅ Automatic code generation (students see translation immediately)

✅ Variable tracking during execution (debugger panel) Compared to Direct Python Learning (IDLE, Replit):

✅ Visual scaffolding reduces initial frustration

✅ Impossible to have syntax errors during design

✅ Logic errors are easier to spot visually

✅ Natural transition path (export Python code and continue)

Educational Benefits

- Visual Learning & Concrete Representation
- Flowcharts provide tangible, spatial representations of abstract programming concepts Students can literally “see” how data flows through their program
- Control structures (if/else, loops) have distinctive shapes that become visual anchors The spatial layout reinforces sequential thinking and logical dependencies
- Immediate, Multi-Modal Feedback
    - Visual: Green highlighting shows which node is currently executing
    - Textual: Python code updates in real-time as flowchart changes
    - Data: Variable values appear in debugger panel during execution
    - Output: Terminal shows program results immediately
- Reduced Cognitive Load During Learning
- Eliminates syntax memorization during initial concept acquisition Students think about “what” before “how” Drag-and-drop interface removes typing barriers Connection validation prevents impossible program structures
- Authentic Programming Practice
- Generates actual Python code, not pseudocode Code can be exported and run in any Python environment Students learn real control structures (if/elif/else, while, for) Variable scoping and data types work identically to Python
- Built-In Scaffolding
- Interface prevents many common errors (syntax, indentation, brackets) Visual connections make control flow explicit Debugger provides insight into program state Speed control allows granular observation of execution
- Self-Paced & Differentiated
- Students work at their own speed Multiple entry points (Examples menu provides templates) Progressive challenge system (30 levels from beginner to advanced)
- Visual medium supports diverse learning styles and abilities 7. Bridge to Text-Based Coding
- Students gradually shift attention from flowchart to Python panel Export feature allows seamless transition to Python IDEs Pattern recognition develops: “This flowchart shape means this Python syntax” Confidence built visually transfers to text-based environments

- Debugging as Visual Problem-Solving
    - Watch program execution step-by-step at controllable speed
    - Variable watch panel shows values changing in real-time
    - Trace logic errors by following the green highlight
    - Visual debugging reduces frustration and builds persistence

Teaching with think2code

Instructional Philosophy

think2code is most effective when used as a thinking tool rather than just a programming tool. The goal is not simply to create working programs, but to develop computational thinking skills that transfer to any programming environment.

Core Teaching Approach:

1. Think First, Code Later: Encourage students to plan their algorithm before touching the computer
2. Visual Reasoning: Use the flowchart to explain WHY the program works, not just WHAT it does
3. Pattern Recognition: Help students see recurring structures (accumulators, counters, guards)
4. Gradual Abstraction: Start concrete (specific numbers), move to abstract (variables)
5. Connection Building: Constantly link flowchart shapes to Python syntax

Integration into Curriculum

think2code as Introduction (Weeks 1-4)

Use think2code to teach core concepts without syntax burden

Students build confidence and understanding

Focus on algorithm design and logical thinking

Transition to text-based Python with familiar concepts

think2code as Supplement (Throughout Course)

Students prototype complex algorithms in think2code first

Translate to Python afterward

Use for visual debugging of text-based programs

Great for standardized test prep (flowchart questions)

think2code as Differentiation Tool

Advanced students: Export to Python and extend

Struggling students: Continue visual approach longer

Visual learners: Primary tool for concept acquisition

All students: Use for planning before coding tests

Pacing Recommendations

First-Time Users (No Prior Programming):

Week 1: Sequence and variables (3-4 class periods)

Week 2: Input/output and simple decisions (3-4 periods)

Week 3: Complex decisions and while loops (4-5 periods)

Week 4: For loops and lists (4-5 periods)

Week 5: Integration projects and transition prep (3-4 periods)

Students with Block-Based Experience (Scratch, etc.):

Week 1: Interface, sequence, variables, I/O (2-3 periods)

Week 2: Decisions and loops (3-4 periods)

Week 3: Lists and advanced patterns (3-4 periods)

Week 4: Projects and Python transition (2-3 periods)

As Supplementary Tool (Ongoing):

1-2 class periods per major new concept

30-45 minutes for algorithm planning before coding assignments

Use as needed for struggling students

Pedagogical Strategies

Strategy 1: Think-Pair-Program Structure:

1. Think (5 min): Students individually sketch flowchart on paper
2. Pair (5 min): Partners compare, discuss, merge ideas
3. Program (10 min): One person drives think2code, other navigates
4. Share (5 min): Pairs demonstrate to class or another pair

Benefits: Separates design from implementation, promotes communication, reduces errors

Best For: Complex problems (decisions, loops, multi-step algorithms)

Strategy 2: Predict-Observe-Explain Structure:

1. Predict: Before running, students write down expected output
2. Observe: Run program, watch execution with slow speed
3. Explain: If prediction was wrong, explain why

Variations:

Teacher runs, students predict each step Students predict variable values at specific points Predict generated Python code before revealing

Benefits: Develops mental models, catches misconceptions, forces active engagement

Best For: Debugging, loops (predicting iteration counts), variable tracking

Strategy 3: Code-to-Flowchart Translation Structure:

1. Provide students with Python code
2. Students recreate as flowchart in think2code
3. Verify by comparing generated Python to original

Example:

score = int(input("Enter score")) if score >= 90: print("A") elif score >= 80: print("B") else: print("C")

Benefits: Reinforces syntax understanding, builds bidirectional fluency, assessment opportunity

Best For: Review, assessment, preparation for AP/IB exams with flowchart questions

Strategy 4: Progressive Refinement Structure:

1. Start with working but inefficient flowchart
2. Students improve without changing functionality
3. Compare solutions for elegance

Example: Sum of 5 numbers

Version 1: Five separate input blocks, five separate addition operations Version 2: Loop with accumulator pattern

Benefits: Shows that multiple solutions exist, introduces efficiency concepts, teaches refactoring

Best For: After students grasp basic functionality, preparing for algorithm analysis

Strategy 5: Visual Debugging Protocol Structure:

1. Student runs program with SLOW speed
2. Teacher/peer watches alongside
3. Stop at each node, ask: “What should happen? What variables should change?”
4. Compare prediction to debugger panel
5. Identify where prediction diverges from actual

Benefits: Systematic debugging approach, reduces frustration, teaches troubleshooting

Best For: When students are stuck, teaching debugging as a skill, loops with errors

Strategy 6: Constrained Redesign Structure:

1. Give students a working flowchart
2. Add a constraint: “Must use a list” or “Must use a while loop instead of for”
3. Students modify to meet new requirement

Benefits: Deeper understanding of constructs, flexibility in thinking, comparative analysis

Best For: After mastering a concept, advanced students, test preparation

Demonstration Best Practices

When Introducing New Concepts:

✅ DO: Build flowchart slowly with student input Think aloud: “I need to store this, so I’ll use a Variable block” Make intentional mistakes and debug publicly Use slow execution speed initially Pause during execution to ask prediction questions Show Python code frequently: “See how this became that?”

❌ DON’T: Show finished flowchart without construction process Rush through connection-making Ignore the Python preview panel Use only fast execution speed Proceed if students look confused (check for understanding)

When Students Work Independently:

✅ DO: Circulate and observe screens Ask students to explain their logic verbally Encourage peer consultation before asking teacher Validate multiple approaches: “Different from mine, but does it work?” Use slow speed for complex sections Have students show you debugger values

❌ DON’T: Fix student work for them Give direct solutions without thought process Allow students to randomly try connections Skip past errors without understanding

Formative Assessment Techniques

Real-Time Checks:

1. Finger Signals

Show 1-5 fingers for confidence level Use before running program: “How confident are you?” Use after running: “Did it do what you expected?”

2. Exit Tickets

Last 5 minutes of class “Draw the flowchart shape for a decision” “Write a condition that checks if x is less than 10” “What does the accumulator pattern mean?”

3. Think-Alouds

Call on students to explain flowchart sections “Talk me through what this loop does” Look for: sequencing, variable tracking, condition understanding

4. Debugging Challenges

Provide broken flowchart Students identify error before running Validates understanding vs. trial-and-error

5. Whiteboard Sketches

Before computer work: sketch flowchart on paper/whiteboard Teacher circulates, catches misconceptions early Prevents “fumbling at keyboard” syndrome

6. Concept Mapping

“When would you use a Decision block vs. a Variable block?” “How is a for loop different from a while loop?” Checks conceptual understanding, not just procedure

Managing Common Classroom Scenarios

Scenario: Students Finish at Different Speeds

Solutions:

Prepare extension challenges (more complex versions) Have fast finishers help slower peers (teaching solidifies learning) Introduce “stretch goals”: Make it work with any number, add error checking Challenge: Export to Python and add features

Scenario: Student Says “It Doesn’t Work”

Debugging Protocol:

1. “Show me what you expected to happen”
2. “Let’s run it slowly together and watch”
3. “Point to where you think the problem is”
4. “What does the debugger show for that variable?”
5. “Let’s trace the path the yellow highlight takes”

Resist fixing immediately—teach debugging process.

Scenario: Whole Class Stuck on Same Concept

Intervention:

Stop independent work Reconvene as group Use projector to demonstrate Build example together step-by-step Check for understanding before releasing again

Vocabulary Development

Tier 1: Essential Terms (First Week)

Algorithm Sequence Execute Variable Input/Output

Tier 2: Control Structures (Second Week)

Condition Boolean Decision/Selection True/False Branch/Path

Tier 3: Iteration (Third Week)

Loop Iteration Accumulator Counter Increment/Decrement

Tier 4: Data Structures (Fourth Week)

List/Array Index Element Traverse

Vocabulary Strategy:

Word wall with flowchart shapes next to terms

Students create visual dictionary with screenshots

Use terms consistently in class discussion

Connect to Python equivalents explicitly

Cross-Curricular Connections

Mathematics:

Fibonacci sequence (recursive patterns)

Prime number checking (modulus operator, nested decisions)

Quadratic formula (formula translation to code)

Statistics (sum, average, maximum—accumulator patterns)

Science:

Data analysis from experiments (lists, averages)

Simulation models (loops for time steps)

Classification algorithms (decision trees)

Language Arts:

Text analysis (counting words, finding patterns)

Mad Libs (input substitution)

Story branching (interactive fiction with decisions)

Transitioning to Text-Based Python

Recommended Transition Process:

Phase 1: Dual View (1-2 weeks)

Students work in think2code but spend increasing time studying Python panel

Ask: “What Python keyword represents this Decision block?”

Point out patterns: if , else , while , for

Phase 2: Parallel Creation (1 week)

Design algorithm in think2code

Export Python code

Manually type the same code in Python IDE

Compare execution

Phase 3: think2code as Planning Tool (2+ weeks)

Design complex algorithms in think2code first

Use as pseudocode/planning tool

Implement directly in Python

Use think2code to debug logic (not syntax) issues

Phase 4: Independent Python (Ongoing)

Students primarily code in Python

Return to think2code for particularly complex algorithm planning

Use think2code for test prep (standardized tests often include flowcharts)

Success Indicators for Transition:

Student can look at flowchart and write Python

Student can read Python and sketch corresponding flowchart

Student uses programming vocabulary correctly

Student debugs logic errors independently

Supporting Diverse Learners

For Students with Dyslexia:

Visual-spatial nature reduces reading load

Use color coding consistently

Allow extra time for text input (typing prompts/variable names)

Pair with peer for double-checking text entry

Export capabilities allow assistive technology use on Python code

For English Language Learners:

Visual symbols are language-independent

Pre-teach vocabulary with visual connections

Allow native language comments/variable names initially

Flowchart shapes transcend language barriers

Partner with strong English speaker for explanation tasks

For Students with ADHD/Executive Function Challenges:

Tangible manipulation (drag/drop) maintains engagement
Visual feedback loop is immediate (reduces waiting frustration)
Breaking algorithms into blocks provides natural chunking
Slow execution shows cause-effect relationships clearly
Canvas organization reflects thinking organization

For Gifted/Advanced Students:

Export to Python for extended features
Challenge: Implement same algorithm three different ways
Study generated Python for optimization
Create comprehensive programs with nested structures
Serve as peer tutors (teaching reinforces learning)

Assessment Alignment

Think2Code supports:

Formative Assessment:

- Immediate visual feedback during construction
- Observable logic during execution
- Variable tracking for conceptual checking
- Easy peer review (visual algorithms are discussable)

Summative Assessment:

- Exportable artifacts (flowcharts, code, output)
- Authentic problem-solving
- Can assess both algorithm design AND code generation
- Prepares for standardized test flowchart questions

Standards Alignment:

GCSE Computer Science (AQA / OCR / Edexcel)

think2code supports the development of core GCSE Computer Science skills, including:

Algorithmic Thinking: designing algorithms using flowcharts and pseudocode representing sequence, selection, and iteration refining and improving solutions

Problem Decomposition: breaking problems into smaller logical steps identifying inputs, processes, and outputs

Programming Constructs: variables and assignment IF / ELSE decision making WHILE / FOR loops tracing execution using dry runs

Testing and Debugging: identifying and correcting logic errors stepping through program execution comparing outputs with expected results

Data in Programs: use of lists/arrays input and output of dataRelevant GCSE content areas include:

Algorithms Programming fundamentals

Designing, using, and refining flowcharts & pseudocode

Developing and testing programs

BTEC IT / Computing (Level 2 and Level 3)

think2code supports units where learners must design, develop, test, and evaluate programs, including:

Creating Solutions to Programming Problems

Computational Thinking and Problem Solving

Software or Website Development Projects

Learners develop the ability to:

Design: produce flowcharts and structured design diagrams identify program requirements and structure

Implement: translate designs into working code understand the link between design and execution

Test and Debug: produce test tables step through program execution explain and fix logic errors

Plan and Document: decompose problems into smaller tasks justify design choices using appropriate constructs produce assignment-ready design evidence

⭐ How think2code supports both GCSE and BTEC learners

Learners can:

design algorithms visually before coding

automatically generate executable Python code

see sequencing, selection, and iteration in action

view real-time variable values

Pedagogical Approach

Constructivist Learning Framework

think2code embodies constructivist principles where students build understanding through active creation rather than passive reception.

Key Constructivist Elements:

1. Hands-On Construction: Students physically manipulate objects (blocks) to build mental models
2. Immediate Feedback Loop: Actions (connecting blocks) yield instant results (code generation, execution)
3. Zone of Proximal Development: Visual scaffolding supports just beyond current ability
4. Social Learning: Pair programming and peer explanation reinforce concepts
5. Authentic Tasks: Programs solve real problems, not just syntax exercises

Cognitive Load Theory Application

think2code strategically manages cognitive load during learning:

Intrinsic Load (Inherent Difficulty):

- Kept constant: Algorithm logic complexity doesn't change
- Gradually increased: Start simple (sequence), build to complex (nested loops)

Extraneous Load (Irrelevant Processing):

- Eliminated: Syntax memorization, bracket matching, indentation rules
- Minimized: Type syntax errors, debugging syntax vs. logic
- Reduced: Visual connections prevent impossible control flows

Germane Load (Learning-Focused Processing):

- Maximized: All cognitive resources devoted to algorithmic thinking
- Supported: Visual patterns aid schema development
- Enhanced: Dual coding (visual + textual) strengthens understanding

Result:

Students think about WHAT to do, not HOW to type it, maximizing learning efficiency.

Dual Coding Theory

think2code leverages dual coding by presenting information in two formats simultaneously:

Visual Channel:

> Flowchart shapes and connections Spatial relationships Color coding Execution highlighting

Verbal Channel:

> Python code text Variable names Terminal output Condition expressions

Cognitive Benefit:

> Students build two mental representations that reinforce each other Visual learners access through diagrams Text learners access through code Connections between channels deepen understanding Retrieval cues multiplied (can recall via either channel)

Scaffolded Learning Progression

Fading Support Model: think2code provides maximum support initially, gradually reducing as competence develops.

Stage 1: Full Scaffolding (Weeks 1-2)

Focus entirely on flowchart Ignore Python panel Use Examples menu for templates Teacher demonstrates every step Students imitate, then modify slightly

Stage 2: Awareness (Weeks 2-3)

Continue building in flowchart Occasionally reference Python panel Teacher points out code patterns Students notice: “This shape makes that code” Begin recognizing syntax structures

Stage 3: Comparison (Weeks 3-4)

Build in flowchart first Study generated Python second Make explicit connections Students predict code from flowchart Forward translation strengthens

Stage 4: Reverse Engineering (Week 4-5)

Given Python code, create flowchart Backward translation challenges Bidirectional fluency developing Students explain in both languages

Stage 5: Independence (Week 5+)

think2code becomes planning tool Design algorithm visually Implement in Python directly Return to think2code only for complex logic Visul thinking informs text coding

Metacognitive Development

think2code naturally promotes metacognition (thinking about thinking):

Planning Phase:

"What blocks do I need?" "What order should they go in?" "Where do I need decisions vs. loops?"

Monitoring Phase:

"Is the yellow highlight going where I expect?" "Are the variables showing correct values?" "Did my prediction match the actual output?"

Evaluation Phase:

"Why didn't it work?" "What part of my logic was wrong?" "How can I improve this algorithm?"

Teaching Strategy:

Model metacognitive self-talk aloud during demonstrations

Ask students to explain their reasoning before running

Use "predict-observe-explain" protocol regularly

Encourage written reflections on problem-solving process

Error as Learning Opportunity

Traditional coding environments punish errors with cryptic messages. think2code reframes errors:

Syntax Errors: Impossible (blocks prevent them)

Logic Errors: Visible (watch execution diverge from expectation)

Runtime Errors: Rare and clear (input type mismatches shown explicitly)

Educational Benefit:

Reduces frustration and learned helplessness

Students experiment without fear

Failure becomes data, not judgment

Iterative refinement is normalized

Growth mindset reinforced

Teacher Move: Celebrate errors as discovery moments. “Great! We found where our thinking was different from the computer’s. Now we can fix our mental model.”

Differentiation Through Universal Design

think2code inherently supports diverse learners without requiring separate accommodations:

Multiple Means of Representation:

Visual (flowchart shapes and layout) Textual (Python code and terminal) Kinesthetic (drag-and-drop interaction) Procedural (step-by-step execution)

Multiple Means of Action/Expression:

Students can explain via flowchart OR code Visual demonstrations show understanding Exported artifacts provide tangible evidence Various complexity levels available

Multiple Means of Engagement:

Game-like interaction (drag blocks, see results) Creative projects (tell stories, design systems) Challenge progression (30 levels, self-paced) Real-world applications (simulations, tools)

Assessment for Learning (AfL)

think2code enables continuous, embedded assessment:

Observable Behaviors:

Does student reach for correct block type? Are connections logical and sequential? Does student predict before running? Can student explain their flowchart to peer?

Diagnostic Information:

Where does student pause or struggle? Which concepts require re-teaching? Is understanding superficial or deep? Can student transfer to new contexts?

Formative Feedback Loops:

Immediate: Program execution shows correctness Short-term: Teacher observation during work Medium-term: Peer review and discussion Long-term: Project complexity increases

Key Insight: Every run is a formative assessment. Students constantly test hypotheses and revise understanding.

Social Constructivism & Collaborative Learning

Pair Programming Protocol:

Driver: Controls mouse/keyboard, builds flowchart

Navigator: Thinks ahead, catches errors, asks questions

Switch roles regularly (every 10 minutes)

Benefits:

Articulating ideas clarifies thinking Peer catches errors immediately Two perspectives yield better solutions Communication skills develop Reduces isolation and frustration

Think-Pair-Share for Debugging:

1. Individual: Attempt problem alone (5 min)
2. Pair: Discuss approach with partner (5 min)
3. Share: One pair demonstrates to class (5 min)

Collaborative Problem-Solving:

Groups of 3-4 receive complex challenge One computer, multiple minds Must reach consensus before implementing Present solution with justification

Transfer of Learning

Near Transfer (Within think2code):

Concept learned with one scenario applies to another

Example: Accumulator pattern in sum → also works for counting

Far Transfer (Beyond think2code):

Algorithmic thinking applies to text-based Python Logic structures transfer to other languages (Java, JavaScript) Problem decomposition useful in math, science, writing

Promoting Transfer:

Explicit connections: “This is the same pattern we used before”

Varied practice: Same concept, different contexts

Abstract principles: “Any time you need a total, use accumulator”

Metacognitive reflection: “How is this like something you’ve done?”

Equity and Inclusion

think2code Reduces Barriers:

Language Barriers:

Visual symbols transcend language Less reading required than text-based coding ELL students access content through multiple modalities

Prior Knowledge Gaps:

No assumptions about typing speed or computer literacy No assumed programming background needed Interface is intuitive and discoverable

Stereotype Threat:

Visual-spatial approach appeals to diverse groups Game-like interaction reduces “who belongs in CS” messaging Creative applications (stories, art) welcome all interests

Accessible Design:

High contrast visuals Large click targets Keyboard alternatives for some actions Works on low-cost devices (Chromebooks)

Teacher Responsibility: Create classroom culture where all approaches are valued, multiple solutions celebrated, and help-seeking normalized.

Building Computational Identity

Beyond teaching programming, think2code helps students see themselves as computational thinkers:

Agency: “I can create programs that do what I want”

Competence: “I understand how computers execute instructions”

Belonging: “I’m the kind of person who can code”

Purpose: “Programming is a tool I can use to solve real problems”

Teacher Moves to Support Identity:

Highlight student creativity in solutions Display student work prominently Connect programming to student interests Share stories of diverse programmers Frame programming as thinking tool, not career gate

Feedback & Iteration This guide is a living document. Please provide feedback:

Contact: adamclement@exe-coll.ac.uk

Thank you for using think2code to empower your students with computational thinking skills. Your dedication to making programming accessible and engaging makes a lasting difference in students' lives and futures.