

Behavioural Experiments as Bedrock for Learning to Program

Matthew J. Crossley

School of Psychological Sciences, Macquarie University, Sydney, Australia

Performance and Expertise Research Centre, Macquarie University, Sydney, Australia

Macquarie Minds and Intelligences Initiative, Macquarie University, Sydney, Australia

`matthew.crossley@mq.edu.au`

Abstract

...

Introduction

Instruction in programming and technical skills commonly emphasizes beginning with simple problems and progressively introducing complexity as new concepts become necessary. Incremental learning, scaffolding, and the gradual construction of more complex systems from simpler components are widely recognized pedagogical strategies across disciplines, including programming education. Within this approach, abstract concepts are introduced through concrete tasks, allowing learners to develop understanding through application rather than through isolated technical instruction.

Behavioural experiment construction provides a natural context in which this form of learning can be applied. Behavioural researchers routinely need to create and modify experimental tasks in order to test new hypotheses, adapt existing paradigms, or implement task designs that differ from standard examples. Contemporary experiment-building tools have substantially reduced the need to program experiments from scratch, and many paradigms can now be implemented through graphical interfaces or automatically generated code using artificial intelligence. Nevertheless, learning to read and write code remains an important skill for behavioural researchers. It enables researchers to understand code written by others, including code produced by artificial intelligence, diagnose and debug unexpected behaviour, and inspect implementations to verify that an experiment is operating as intended. These abilities are particularly important when experimental behaviour depends on timing, conditional branching, or adaptive task structure, where unintended implementation details can alter experimental behaviour in ways that are not immediately visible through graphical interfaces alone.

Behavioural experiment construction is especially well suited to incremental learning because experimental design already requires decomposition of tasks into phases, specification of the conditions under which behaviour changes, and definition of alternative outcomes based on participant

actions or elapsed time. Designing an experiment requires specifying when behaviour changes, what conditions trigger those changes, and how alternative outcomes are handled while the experiment is running. The experiment must repeatedly evaluate timing, monitor input, and determine whether conditions have been satisfied for behaviour to change. This structure closely mirrors fundamental programming concepts. Decomposing an experiment into task phases corresponds to organizing behaviour into distinct modes of operation, while specifying the conditions under which a trial progresses corresponds to conditional logic evaluated within a running program. Measurement of response time, handling of early or late responses, adaptive feedback, and branching task structure all arise from the same underlying requirement to specify how behaviour changes under different conditions. As a result, learners encounter core programming ideas—repetition, conditional execution, and state-dependent behaviour—not as abstract constructs, but as direct consequences of experimental requirements.

Behavioural experiment construction therefore provides a natural domain for learning to program. Many students in psychology and cognitive science encounter programming with limited prior technical experience, and learning programming through experimentally meaningful problems provides a concrete context in which problem decomposition and logical reasoning can be developed alongside technical skills. Programming concepts emerge as solutions to experimental requirements rather than as independent technical topics. Learners develop an understanding of loops, conditions, and control structure while simultaneously learning how experimental behaviour is specified, implemented, and modified in practice.

The present work describes a pedagogical approach in which behavioural experiment programming is introduced through a structured progression from simple interactive programs to complete experimental paradigms. Learners construct experiments incrementally while retaining a stable program structure, with timing, response measurement, and experimental control introduced as extensions of earlier components, allowing increasingly complex experimental behaviour to emerge from a common underlying structure. The accompanying repository provides a sequence of Python examples designed to support this progression. The contribution of the present work is to argue that established incremental teaching principles are particularly well served in psychology and cognitive science by learning to program behavioural experiments, because programming concepts and experimental design principles can be acquired together through the same progression.

Progressive Construction as a Teaching Approach

The preceding section argued that behavioural experiment construction provides a natural context in which programming concepts arise from the requirements of experimental design. The present section describes how this alignment can be used as a teaching strategy. The central idea is to introduce programming through the progressive construction of experiments, allowing increasingly complex experimental behaviour to emerge from extensions of a stable underlying program structure rather than from the introduction of independent technical topics.

The approach is guided by three principles. First, the overall structure of the program remains stable throughout the progression. Learners begin with a continuously running interactive program and retain this structure as new capabilities are introduced. Second, new programming concepts are introduced only when they are required to solve a concrete experimental problem, such as controlling stimulus duration, measuring response time, or handling alternative outcomes. Third, experimental behaviour is constructed incrementally, so that each stage builds directly on earlier components rather than replacing them. This allows learners to understand increasing complexity as the result of extending an existing system rather than learning new programming patterns in isolation.

The progression begins with stimulus presentation within a continuous execution loop. At this stage learners implement basic visual behaviour while observing that experimental programs must continuously update display and input rather than executing once from beginning to end. Timing is then introduced as a measurable quantity, allowing stimulus duration and event timing to be controlled explicitly. Subsequent stages introduce measurement of participant actions, first through discrete responses such as keypresses and later through continuously evaluated input such as mouse movement. These additions establish continuous execution, measurement of time, and detection of participant behaviour as the core components required for behavioural experiments.

As experimental requirements become more complex, learners encounter situations in which behaviour depends on multiple possible outcomes. At this stage experimental control is made explicit by organizing behaviour into task phases together with conditions governing transitions between phases. That is, the program is structured around a set of states that define distinct modes of operation, with conditions that specify when behaviour changes and how alternative outcomes are handled. This structure allows increasingly complex behaviour to be implemented as extensions of a common underlying program structure, with new states and transition conditions added as needed to implement new experimental requirements.

The final stage of the progression demonstrates how complete experimental paradigms emerge from this structure. Reaction time tasks, category learning paradigms, and movement-based experiments differ primarily in their state definitions and transition conditions while preserving the same underlying execution model. Learners therefore encounter different experimental designs as variations within a shared program structure rather than as entirely new implementations.

The accompanying repository provides a sequence of Python examples designed to support this progression. Each example introduces a small number of new concepts while retaining the structure established in earlier stages, allowing programming concepts and experimental reasoning to develop together through practice. In this way, programming is learned not as a separate technical skill but as a natural extension of experimental design.

Progressive Construction of Behavioural Experiments

The preceding sections described behavioural experiment construction as a progression in which programming concepts emerge from experimental requirements. The present section addresses how this progression is realized in practice. The goal is not to introduce a software framework, but to show how complete experimental behaviour can be constructed from a small set of implementation components that are extended incrementally. Examples are presented in pseudocode together with corresponding Python implementations drawn from the accompanying repository.

Concrete examples in this section are presented using Python together with the Pygame library for window management and visual stimulus presentation. Python was selected because it is widely used in behavioural research, has a low barrier to entry for learners with limited programming experience, and supports straightforward expression of experimental logic without extensive boilerplate code. Pygame provides direct and transparent control over display updating, input handling, and timing, making the underlying execution structure visible to learners rather than abstracted behind higher-level interfaces. The specific choice of language and library is not essential to the approach described here. Equivalent implementations could be constructed using experiment-focused toolkits such as PsychoPy, graphical experiment builders, or web-based environments using JavaScript. The examples presented here are therefore intended to illustrate general implementation principles rather than to prescribe a particular software environment.

Window Management and Visual Stimulus Presentation

All behavioural experiments require a mechanism for presenting stimuli and updating the display while the program is running. This aspect of implementation is necessarily dependent on the underlying system or library used for presentation. Different environments provide different interfaces for opening windows, drawing stimuli, and synchronizing display updates. The details therefore vary across toolkits such as PsychoPy, Pygame, or web-based frameworks. What remains constant across environments, however, is the underlying requirement that visual behaviour must be updated continuously while the experiment is active.

A common misconception among beginners is that stimuli are drawn once and remain on screen until replaced. In practice, modern display systems operate by repeatedly drawing frames. Each frame represents a complete image that is presented to the display for a single refresh interval. Experimental programs therefore construct visual output repeatedly, producing a sequence of frames that together create the appearance of continuous visual behaviour. Stimuli remain visible not because they persist automatically, but because they are redrawn on every update cycle.

This process is typically implemented through a “draw–update” or “draw–flip” cycle. During each iteration of the program loop, the display is first cleared, new visual content is drawn to an off-screen buffer, and the completed frame is then presented to the screen in a single operation. The final step, often called a *flip* or buffer swap, replaces the currently visible frame with the newly constructed one. Presenting the frame in a single operation prevents partially drawn images from

becoming visible and allows display updates to be synchronized with the monitor refresh cycle. Although the specific function names differ across libraries, this conceptual structure is shared across most stimulus presentation systems.

Introducing learners to this update cycle serves two purposes. Conceptually, it establishes that experimental programs operate as continuously running systems rather than as sequences of commands executed once from beginning to end. Pedagogically, it provides an immediate and engaging entry point. Visual changes on screen occur directly in response to changes in code, allowing learners to observe cause and effect without requiring prior familiarity with abstract programming concepts.

Continuous Display Update (Pseudocode)

```
initialize window

repeat while experiment is running:

    clear display
    draw stimulus
    update display
```

In Python/Pygame, the same structure appears as:

Continuous Display Update (Python)

```
running = True

while running:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    screen.fill(background_color)
    draw_stimulus(screen)
    pygame.display.flip()
```

Here, `pygame.display.flip()` performs the buffer swap that makes the newly drawn frame visible. Everything drawn prior to this call becomes visible simultaneously on the next display update. This loop forms the stable foundation upon which subsequent experimental functionality is added. Timing, input handling, and experimental control are later introduced as extensions of this same continuously running structure.

Timing and Measurement

Once continuous execution has been established, the next requirement that arises in experiment construction is control over duration. Experiments require stimuli to be shown for limited intervals,

responses to be accepted within defined windows, and delays to occur between task events. Timing therefore emerges naturally as soon as learners attempt to control how long experimental behaviour should persist.

In the progression used here, duration is initially introduced without reference to clocks. Instead, behaviour is changed after a fixed number of iterations through the main execution loop. This is not presented as a practical timing solution, but as a pedagogical step that allows learners to encounter conditional control flow before introducing library-specific timing mechanisms. At this stage, the goal is to establish that behaviour changes when conditions become true, rather than to achieve accurate temporal control.

Duration Control by Iteration (Pseudocode)

```
iteration_count = 0

repeat while experiment is running:

    draw stimulus
    update display

    iteration_count += 1

    if iteration_count > limit:
        change behaviour
```

A corresponding implementation in Python/Pygame appears as:

Duration Control by Iteration (Python)

```
iteration_count = 0

while running:

    screen.fill(background_color)
    draw_stimulus(screen)
    pygame.display.flip()

    iteration_count += 1

    if iteration_count > 120:
        show_stimulus = False
```

This stage serves two instructional purposes. First, learners encounter conditional logic in a concrete setting seeing that behaviour changes when a condition is satisfied. Second, it reinforces that experimental behaviour emerges from repeated evaluation within the running loop. The limitations of this approach then become immediately apparent. Because loop speed depends on hardware and rendering load, iteration counts do not correspond to reliable durations. This motivates the introduction of explicit time measurement.

The next step is therefore to introduce clocks and elapsed time. Instead of counting iterations, the program records a reference time and compares the current time against it during each update cycle. Duration is now defined in terms of measured elapsed time rather than the speed of execution.

Timing with Clocks (Pseudocode)

```
start_time = current_time()

repeat while experiment is running:

    draw stimulus
    update display

    if current_time() - start_time > duration:
        change behaviour
```

In Python/Pygame, timing is typically implemented using the library's clock functions:

Timing with Clocks (Python)

```
start_time = pygame.time.get_ticks()

while running:

    screen.fill(background_color)
    draw_stimulus(screen)
    pygame.display.flip()

    elapsed = pygame.time.get_ticks() - start_time

    if elapsed > stimulus_duration:
        show_stimulus = False
```

As with display updating, timing mechanisms are inherently library-dependent. Different environments provide different ways of accessing clocks or synchronizing execution with display refresh. PsychoPy provides clock objects tied to stimulus presentation, while web-based environments rely on browser timing functions. The specific implementation therefore varies, but the conceptual structure remains the same: elapsed time becomes another quantity evaluated within the running loop that determines when behaviour changes.

Introducing timing in this way preserves continuity with the earlier stages of the progression. The program does not pause while waiting for time to pass. Instead, time is continuously measured while display updates and input monitoring continue. This establishes a model of execution that supports later introduction of response measurement, deadlines, and state-dependent experimental control.

Input and Response Measurement

Behavioural experiments require measurement of participant actions. Once learners can control visual behaviour and duration, the next requirement arises naturally when the experiment must respond to participant behaviour. Input handling introduces conditional execution in a concrete form. Program behaviour now depends on whether an event has occurred while the experiment is running.

Keyboard input provides a clear first example because it represents discrete events that occur at identifiable moments in time. Learners encounter the idea that input must be checked repeatedly within the execution loop rather than requested once. Responses are detected as the program runs and recorded when conditions are satisfied. This reinforces the earlier principle that experimental programs do not stop while waiting for events. Instead, input is monitored continuously alongside display updating and timing.

Input Handling (Pseudocode)

```
repeat while experiment is running:  
    observe input events  
    if keypress detected:  
        record response
```

In Python using Pygame, input handling appears as part of the same continuously running loop introduced earlier:

Keyboard Input (Python)

```
running = True  
  
while running:  
  
    for event in pygame.event.get():  
  
        if event.type == pygame.QUIT:  
            running = False  
  
        if event.type == pygame.KEYDOWN:  
            response_key = event.key  
            response_time = pygame.time.get_ticks()  
  
    screen.fill(background_color)  
    draw_stimulus(screen)  
    pygame.display.flip()
```

At this stage learners encounter several ideas that later become central to experiment construction. Responses occur at unpredictable times. Detection of input must therefore be integrated into

the ongoing execution loop. Response measurement becomes a comparison between the time of an event and a previously recorded reference time such as stimulus onset.

After discrete input has been introduced, continuously evaluated input can be added using the same structure. Mouse position provides a useful example because it changes continuously while the program runs. Rather than detecting a single event, the program repeatedly evaluates whether a spatial condition has been satisfied.

Continuous Input (Pseudocode)

```
repeat while experiment is running:  
    cursor_position = get_cursor_position()  
    if cursor inside target region:  
        record outcome
```

A corresponding Pygame implementation again makes the running loop explicit:

Mouse Input (Python)

```
running = True  
  
while running:  
  
    for event in pygame.event.get():  
        if event.type == pygame.QUIT:  
            running = False  
  
    cursor_x, cursor_y = pygame.mouse.get_pos()  
  
    if target_rect.collidepoint(cursor_x, cursor_y):  
        reached_target = True  
  
    screen.fill(background_color)  
    draw_stimulus(screen)  
    pygame.display.flip()
```

Input handling is also dependent on the underlying library. Different environments provide different mechanisms for accessing keyboard, mouse, or other input devices. The conceptual role remains unchanged. Participant behaviour is treated as another condition evaluated within the running program. Experimental behaviour changes when those conditions are satisfied.

By this stage learners have implemented the essential components shared by most behavioural experiments. The program runs continuously. Time is measured explicitly. Participant behaviour is detected and recorded as it occurs. These components provide the foundation upon which explicit experimental control is introduced in the following section.

Explicit Experimental Control

As experimental requirements expand, behaviour must depend on multiple possible outcomes. A stimulus may remain visible until a response occurs, until a deadline is reached, or until another condition becomes true. Feedback may depend on performance, and trials may terminate early under specific conditions. At this stage learners encounter situations in which the program must behave differently depending on the current phase of the task. Experimental control is therefore made explicit by introducing task states and the conditions that determine transitions between them.

In the progression used in the repository, the idea of state is introduced in its simplest possible form. The program alternates between two modes of operation. Each mode produces a different visible behaviour, and a simple condition causes a transition between the two. This provides a concrete first example in which the meaning of a state is directly visible on screen.

Two State Toggle (Pseudocode)

```
state = A

repeat while program is running:

    if state == A:
        draw behaviour A
        if transition condition is true:
            state = B

    if state == B:
        draw behaviour B
        if transition condition is true:
            state = A
```

In Python using Pygame, this appears within the same running loop used throughout earlier stages:

Two State Toggle (Python)

```
state = "A"

running = True

while running:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        if event.type == pygame.KEYDOWN:
            if state == "A":
                state = "B"
            else:
                state = "A"

    screen.fill(background_color)

    if state == "A":
        draw_state_A(screen)

    if state == "B":
        draw_state_B(screen)

    pygame.display.flip()
```

This two state example establishes the core idea that experimental behaviour can be organised into distinct modes of operation. The program continues to update the display continuously, but what is drawn and what input is relevant depends on the current state. The transition condition can be changed without changing the overall structure. In later steps, the same toggle behaviour can be made time driven by changing the transition rule from a keypress to elapsed time. This reinforces that states provide an organisational structure, while transition conditions define when behaviour changes.

Once the idea is established with two states, the same structure generalises directly to tasks with multiple phases. Experiments differ in the number of states they require and in the conditions that govern transitions between them. The execution loop remains the same, but state specific behaviour and transition rules scale to any number of task phases.

Generic N State Control (Pseudocode)

```
state = STATE_1

repeat while program is running:

    observe input events
    update timing variables

    if state == STATE_1:
        execute actions for STATE_1
        if transition condition T_1 is true:
            state = next state

    elif state == STATE_2:
        execute actions for STATE_2
        if transition condition T_2 is true:
            state = next state

    ...
    elif state == STATE_N:
        execute actions for STATE_N
        if transition condition T_N is true:
            state = next state

    update display
```

In this form, the program is structured around two stable components. The execution loop provides continuous updating and access to timing and input. The state logic specifies what behaviour is active and the conditions under which that behaviour changes. Early responses, timeouts, adaptive structure, and trial dependent feedback are implemented by adding new states or adding alternative transition conditions within a state. This preserves continuity with the earlier progression while making experimental control explicit and inspectable.

Data Recording and File I/O

The final component introduced in the progression is data recording. File input and output are often presented as separate programming topics, but in experiment construction they arise directly from the need to preserve measurements produced during task execution. Responses, timing variables, and trial outcomes are generated as the experiment runs and must be stored so that behaviour can be analysed after the experiment has finished.

Within the structure developed in earlier sections, data recording is introduced as another explicit state of the experiment. Rather than writing data at arbitrary points in the program, recording occurs as a consequence of a state transition. This makes clear that data are produced by experimental events and that recording those events is part of experimental control itself. A dedicated write-to-file state also makes it explicit when data are finalized for a trial before the

experiment proceeds.

Data Recording State (Pseudocode)

```
state = STIMULUS

repeat while experiment is running:

    if state == STIMULUS:
        draw stimulus

        if response detected:
            store response variables
            state = WRITE_DATA

    if state == WRITE_DATA:
        write stored variables to file
        state = NEXT_TRIAL
```

A simplified Python example using the same structure appears below. The program continues to run within the same execution loop used throughout earlier stages, while data recording is handled explicitly as part of the state logic.

Data Recording State (Python)

```
state = "STIMULUS"

running = True

while running:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        if event.type == pygame.KEYDOWN and state == "STIMULUS":
            response_key = event.key
            reaction_time = current_time - stimulus_onset
            state = "WRITE_DATA"

    if state == "STIMULUS":
        draw_stimulus(screen)

    elif state == "WRITE_DATA":
        data_file.write(
            f"{trial},{response_key},{reaction_time}\n"
        )
        state = "NEXT_TRIAL"

    pygame.display.flip()
```

As with display updating and timing, the specific mechanisms for file handling depend on the programming environment. Different libraries provide different interfaces for writing structured data or managing files. The conceptual structure remains stable. Data recording captures the outcome of an experimental event and is performed as part of the transition between task phases.

Introducing file output in this way reinforces the overall structure developed throughout the progression. Experiments are continuously running systems that present stimuli, measure behaviour, and transition between states. Recording data becomes another explicit component of this structure, rather than an independent programming concern.

From Components to Experiments

At this point in the progression, complete experimental tasks no longer require new programming mechanisms. Instead, experiments emerge from the combination of components already introduced. Continuous execution provides the running structure of the program. Timing allows behaviour to be controlled over duration. Input handling allows behaviour to depend on participant actions. State-based control organizes behaviour into task phases. Data recording preserves the outcomes of those phases for later analysis. Together these elements form a minimal implementation structure from which a wide range of behavioural paradigms can be constructed.

This has an important instructional consequence. New experiments are not introduced as new kinds of programs. Reaction time tasks, learning paradigms, and movement-based experiments differ primarily in their state definitions and transition conditions while preserving the same underlying structure. Learners therefore encounter increasing experimental complexity as an extension of familiar program behaviour rather than as a sequence of unrelated implementations.

The following section demonstrates this directly by showing how different experimental paradigms arise from variations within this shared structure. The emphasis shifts from individual components to complete experimental tasks, illustrating how experimental design decisions map onto the same underlying implementation framework.

Experimental Paradigms Within a Shared Structure

The preceding sections described how behavioural experiments can be constructed through the progressive introduction of continuous execution, timing, input handling, explicit control structure, and data recording. Once these components are established, complete experimental paradigms emerge from variations within the same implementation structure. This section demonstrates how commonly used behavioural tasks differ primarily in their state definitions and transition conditions, while sharing the same overall program organisation.

The examples below include short Python and Pygame fragments to make the mapping from pseudocode to implementation concrete. Full runnable scripts, including complete drawing, trial management, and data logging, are provided in the accompanying repository.

Reaction Time Tasks

A reaction time task can be expressed as a small number of states such as fixation, stimulus, feedback, and trial advance. Transitions are driven by elapsed time and response detection. Reaction time arises from the interval between stimulus onset and the response event that causes a transition out of the stimulus state.

Reaction Time Task (Pseudocode)

```
state = FIXATION

repeat while experiment is running:

    if state == FIXATION:
        draw fixation
        if fixation time elapsed:
            stimulus_onset = current_time()
            state = STIMULUS

    if state == STIMULUS:
        draw stimulus
        if response detected:
            reaction_time = current_time() - stimulus_onset
            state = FEEDBACK
```

A simplified Python and Pygame fragment illustrates the same logic. Here, timing is implemented with timestamps, and keypress detection is handled through event polling.

Reaction Time Task (Python/Pygame Fragment)

```
state = "FIXATION"
state_start = pygame.time.get_ticks()
stimulus_onset = None
response_key = None

while running:

    response_key = None
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        if event.type == pygame.KEYDOWN:
            response_key = event.key

    now = pygame.time.get_ticks()
    screen.fill(background_color)

    if state == "FIXATION":
        draw_fixation(screen)
        if now - state_start > fixation_duration:
            state = "STIMULUS"
            stimulus_onset = now
            state_start = now

    elif state == "STIMULUS":
        draw_stimulus(screen)
        if response_key is not None:
            rt = now - stimulus_onset
            state = "FEEDBACK"
            state_start = now

    pygame.display.flip()
```

Response deadlines and anticipation errors can be added by introducing additional transition conditions within the stimulus or fixation states. The loop structure remains unchanged.

Category Learning Tasks

Category learning tasks use the same execution structure but introduce trial variables and transitions that depend on response correctness. The experiment presents a stimulus, collects a response, evaluates it relative to a category rule, and presents feedback based on the outcome.

Category Learning Task (Pseudocode)

```
state = STIMULUS

repeat while experiment is running:

    if state == STIMULUS:
        draw stimulus
        if response detected:
            correctness = evaluate_response()
            state = FEEDBACK

    if state == FEEDBACK:
        draw feedback(correctness)
        if feedback time elapsed:
            state = NEXT_TRIAL
```

A short Python and Pygame fragment shows how correctness is treated as a trial variable that is set at the transition from stimulus to feedback.

Category Learning (Python/Pygame Fragment)

```
if state == "STIMULUS":
    draw_stimulus(screen)

    if response_key is not None:
        correct = (response_key == correct_key_for_trial)
        state = "FEEDBACK"
        state_start = now

elif state == "FEEDBACK":
    draw_feedback(screen, correct)

    if now - state_start > feedback_duration:
        state = "NEXT_TRIAL"
```

The task differs from a reaction time task in its transition logic and trial variables, not in its underlying execution structure.

Movement-Based Experiments

Movement-based experiments provide an example in which transitions depend on continuously evaluated input. In a reaching task, cursor position is monitored throughout execution, and transitions occur when spatial conditions are satisfied.

Reaching Task (Pseudocode)

```
state = START

repeat while experiment is running:

    if state == START:
        draw start region
        if cursor in start region:
            state = MOVE

    if state == MOVE:
        draw target
        if cursor in target region:
            state = FEEDBACK
```

A Python and Pygame fragment illustrates continuous input evaluation using mouse position.

Reaching Task (Python/Pygame Fragment)

```
cursor_x, cursor_y = pygame.mouse.get_pos()

if state == "START":
    draw_start_region(screen)
    if start_rect.collidepoint(cursor_x, cursor_y):
        state = "MOVE"
        state_start = now

elif state == "MOVE":
    draw_target(screen)
    if target_rect.collidepoint(cursor_x, cursor_y):
        state = "FEEDBACK"
        state_start = now
```

Continuous movement changes the variables that govern transitions, but the program structure remains the same. The experiment still runs within the same loop that updates display, measures time, and evaluates conditions that determine when behaviour changes.

Across these paradigms, differences in experimental design correspond to differences in state definitions and transition conditions rather than differences in execution structure. This supports the instructional goal of the progression. Once learners understand the shared structure, new experimental designs can be implemented by modifying states and transitions rather than by adopting new programming patterns.

Discussion

The approach described in this paper is grounded in well-established educational principles. Incremental learning, scaffolding, and the progressive introduction of complexity are widely recognized as

effective strategies in both general education and programming instruction. From this perspective, presenting behavioural experiment construction as a progression from simple interactive programs to complete experimental paradigms may be viewed as a domain-specific application of existing pedagogical ideas rather than a fundamentally new teaching method. Many forms of programming instruction similarly emphasize building systems from simple components and introducing new concepts only as they become necessary for solving increasingly complex problems.

The contribution of the present work lies instead in identifying behavioural experiment construction as a domain in which programming concepts and experimental reasoning are naturally aligned. Experimental design already requires explicit specification of task phases, conditions governing behavioural change, and alternative outcomes arising from participant actions or timing constraints. Making this structure explicit during instruction allows programming concepts such as loops, conditional logic, and state-dependent behaviour to emerge directly from experimental requirements. In this way, learners acquire programming skills while simultaneously developing a clearer understanding of experimental control, rather than learning programming and experimental design as separate activities.

This alignment also distinguishes the present approach from instruction centered primarily on software tools or templates. Graphical experiment builders and automated code generation have substantially lowered barriers to implementing standard paradigms and remain appropriate for many purposes. However, understanding how experimental behaviour is specified and controlled remains important for modifying existing tasks, interpreting implementations written by others, and verifying that experimental behaviour matches intended design. The progression described here treats programming not as an end in itself, but as a means of making experimental logic explicit and inspectable.

An additional implication of the approach concerns the role of programming in contemporary research practice. The increasing availability of graphical experiment builders and artificial intelligence systems capable of generating code has reduced the need for researchers to implement experiments entirely from first principles. These developments do not, however, remove the need to understand program behaviour. Experiments remain executable systems whose behaviour depends on timing, conditional logic, and interaction between components. The ability to read, inspect, and reason about code remains necessary for diagnosing unexpected behaviour, verifying that an implementation matches experimental intent, and making targeted modifications when experimental requirements change. Learning to program through experiment construction therefore supports not only implementation but also critical evaluation of implementations produced by others or generated automatically.

The approach also emphasizes separation between experimental logic and software-specific implementation. The examples presented here use Python and Pygame because they make execution structure visible and accessible to learners. The underlying principles are not tied to a particular language or toolkit. The same control structure appears in specialised experiment software, graphical environments, and web-based frameworks. By focusing instruction on the relationship between

task structure and program behaviour, learners can transfer their understanding across tools rather than associating experimental design with a single software environment.

Several limitations should be acknowledged. The progression described here is conceptual and pedagogical rather than an empirically evaluated instructional intervention. The paper does not present quantitative evidence regarding learning outcomes, and future work may examine how the approach influences learners' ability to modify or extend experimental paradigms or to transfer experimental control concepts across programming environments. In addition, the approach assumes an instructional setting in which learners have sufficient time to engage with implementation details. Courses that prioritize rapid deployment of standard paradigms may reasonably adopt alternative approaches based on templates or graphical interfaces.

The examples presented focus on interactive behavioural experiments with clearly defined task phases. Other forms of experimental design, including asynchronous or distributed experimental systems, may require extensions to the framework described here. Future work may also explore how the progression can be adapted for web-based experimentation or integrated with existing experiment-building tools while retaining explicit representation of experimental control.

In summary, the present work argues that behavioural experiment construction provides a particularly effective context in which programming and experimental reasoning can develop together. By introducing programming concepts through the progressive construction of experimental behaviour, learners encounter core ideas of program control as direct consequences of experimental requirements. This alignment allows experiment construction to function both as a practical research skill and as a structured pathway for learning how complex interactive systems are specified, implemented, and understood.