ch3 part 2

The Study Bible for Python Control Statements: A Mechatronics Perspective

Introduction: The Brain of the Machine - Directing the Flow of Logic

In the world of programming, and especially in the domain of mechatronics and robotics, a program's default behavior is remarkably simple. Statements are executed one after the other, in the exact order they are written. This is known as **sequential execution**.[1] Imagine following a recipe to bake a cake or a pilot running through a pre-flight checklist; each step is performed in a strict, unvarying sequence. This linear progression is the most basic form of logic, but it is fundamentally insufficient for any system that must interact with and respond to a dynamic environment.

To imbue a machine with what we perceive as "intelligence," we must be able to deviate from this rigid sequence. This is achieved through a concept called **transfer of control**, which allows us to specify that the next statement to be executed is something other than the next one in line.[1] This capability is the very foundation of adaptive and autonomous behavior. It is the difference between a simple clockwork automaton that repeats a fixed pattern and a modern industrial robot that can adjust its grip based on the object it is holding. The ability to transfer control allows a system to skip an unnecessary step, choose an alternative action based on sensor data, or repeat a process until a goal is achieved.

Pioneering research in computer science demonstrated that all complex algorithmic behavior could be constructed from just three fundamental forms of control: the default **sequential execution**, **selection statements** for making decisions, and **repetition statements** for performing loops.[1] These control statements are not merely tools for organizing code; in mechatronics, they are the architectural blueprints for a system's behavior. They are the mechanisms that translate raw sensor inputs—like a proximity reading or a motor encoder value—into physical actuator outputs, such as stopping a conveyor belt or rotating a robotic

arm.[2] The way these control statements are structured defines the system's core reactive and autonomous capabilities. Therefore, mastering control flow is not just an academic exercise in programming; it is an absolute prerequisite for designing any functional mechatronic system, from a simple automated guided vehicle to a complex surgical robot.

Chapter 1: The Bedrock of Predictability - Structured Programming

In the early history of programming, a powerful but dangerous statement existed in many languages: the goto statement. This command allowed for an unconditional transfer of control to virtually any other line of code in a program, identified by a label.[1] While flexible, this led to a phenomenon often called "spaghetti code," where the logical flow of a program was tangled and convoluted, jumping from one section to another in an unpredictable manner. Following the logic of such a program was like trying to trace a single noodle in a bowl of spaghetti. This made the code nearly impossible to read, debug, maintain, or formally verify, a critical failure point when dealing with complex systems.[3] Imagine a factory floor where any worker could, at any moment, abandon their station and instantly appear at any other station; the result would be chaos and a complete breakdown of the production line.

The solution to this chaos came in the 1e960s with a groundbreaking paper by Corrado Böhm and Giuseppe Jacopini. They provided a mathematical proof that any algorithm, regardless of its complexity, could be implemented using only three fundamental control structures: sequential execution, selection (decision-making), and repetition (looping).[1] This insight became the foundation of

structured programming, a paradigm that advocates for building programs from these simple, predictable blocks. A key characteristic of these blocks is that they have a single entry point and a single exit point. This discipline eliminates the chaotic jumps of the goto statement, ensuring a clear, hierarchical, and predictable flow of control. Python's design embraces this philosophy completely; its lack of a goto statement is not a limitation but a deliberate choice to enforce a more robust and maintainable programming style.[1]

This principle of structured programming is not merely about writing cleaner code; for a mechatronics engineer, it is a cornerstone of building safe and reliable systems. In safety-critical fields such as automotive engineering, aerospace, and medical devices, software failures can have catastrophic consequences. To manage this risk, industry standards like **MISRA C** (Motor Industry Software Reliability Association C) were developed to provide strict guidelines for writing code in these domains.[4] A central tenet of these standards is the enforcement of structured programming principles. For instance, MISRA C rules often

mandate that the body of every

if, else, while, or for statement must be enclosed in braces. This rule eliminates ambiguity about which statements belong to the control structure, making the code's logic explicit and easily verifiable by both human reviewers and automated static analysis tools.[5]

Ultimately, adopting structured programming is a powerful risk mitigation strategy. The unpredictable nature of goto statements makes it extraordinarily difficult to prove that a program will always behave as expected under all conditions, a requirement for certification under standards like ISO 26262 for automotive functional safety or IEC 62304 for medical device software.[4] By restricting control flow to single-entry, single-exit blocks, structured programming systematically reduces the number of possible execution paths a program can take. This makes the system more deterministic and its behavior more predictable, enabling the rigorous analysis necessary to ensure safety and reliability.[8] For the mechatronics engineer, writing structured code is not a matter of style; it is a fundamental professional discipline that directly impacts the safety and certifiability of the final product.

Chapter 2: Blueprinting Logic - The Art of the Flowchart

Before writing a single line of code, it is essential to have a clear and unambiguous plan for the program's logic. A **flowchart** is a powerful tool for this purpose, providing a graphical representation of an algorithm or a segment of one.[1] By using a standardized set of symbols, flowcharts allow engineers to visualize and communicate the flow of control in a way that is independent of any specific programming language.

The primary symbols used in a flowchart are distinct and intuitive [1]:

- **Rounded Rectangles** signify the start and end points of the algorithm, typically containing the words "Begin" and "End."
- **Rectangles** represent actions or processes. These are the "doing" parts of the algorithm, such as "calculate velocity," "activate gripper," or "read sensor value."
- **Diamonds** are decision symbols. They contain a question that can be answered with a "yes" or "no" (or "true" or "false"). This is where selection takes place, representing the logic of an if statement. Each diamond has at least two arrows, or **flowlines**, exiting from it, one for each possible answer.
- **Arrows**, known as flowlines, connect the symbols and indicate the direction of program execution, showing the path of control.
- **Small Circles**, or connector symbols, are used to link different parts of a flowchart,

especially in large diagrams that might span multiple pages or sections.

To illustrate this in a mechatronics context, consider the logic for a simple robotic "pick-and-place" operation, a foundational task in industrial automation.[9] A flowchart for this task would verbally trace as follows:

1. The process starts at a **rounded rectangle** labeled "Begin."
2. An arrow leads to a **rectangle** containing the action "Move to home position."
3. The flowline then points to a **diamond** with the question, "Is an object detected?".[11]
4. If the answer is "No," an arrow loops back to a point just before this decision, indicating the robot should wait or continue scanning.
5. If the answer is "Yes," an arrow proceeds to a sequence of **rectangles**: first, "Move gripper to object," followed by "Close gripper," then "Move to drop-off location," and finally "Open gripper".[12]
6. After the object is placed, a flowline directs the logic back to the "Move to home position" action, creating a loop to ready the robot for the next part.

This visual trace clearly separates actions from decisions and shows the overall program loop before any code is written, cementing the flowchart's role as an indispensable design and documentation tool.[13]

In the interdisciplinary field of mechatronics, where mechanical, electrical, and software engineers must collaborate seamlessly, the flowchart serves as a universal language.[2] A mechanical engineer can define the required physical sequence of operations for a machine, and a software engineer can then use that exact logic to implement the control program. The flowchart acts as the bridge between these domains, allowing the entire team to validate the system's intended behavior on a whiteboard. Discovering a logical flaw at this design stage is infinitely cheaper and faster than discovering it after a physical prototype has been built and programmed. Therefore, learning to think in flowcharts is not just a programming skill; it is a crucial competency in systems engineering, enabling clear communication and robust collaborative design in any complex, multidisciplinary project.

Chapter 3: The Crossroads of Logic - Selection Statements

Selection statements are the primary tools for implementing decision-making in a program. They allow a system to evaluate a **condition**—an expression that resolves to either True or False—and execute a specific block of code based on that outcome. Python provides three types of selection statements: the if statement, the if-else statement, and the if-elif-else

statement.[1]

3.1 The if Statement: The Conditional Gate

The if statement is the simplest form of decision-making, often called a **single-selection statement**.[1] It functions like a conditional gate. The program evaluates a condition; if that condition is

True, an associated block of code is executed. If the condition is False, the block is simply skipped, and the program continues with the next statement after the if block.[15] It is a structure for "do this, or do nothing."

- **Mechatronics Example 1: Robot Arm Safety Protocol.** Consider a collaborative robot arm designed to work alongside human operators. The arm is equipped with force sensors to prevent injury. The control logic would include a critical safety check. Verbally, this logic is: "The program checks the force sensor reading. if the force reading exceeds the pre-defined safety threshold, then immediately halt all motor movement." In this scenario, an action is only taken in the exceptional case of excessive force. If the force is within normal limits, the if block is skipped, and the robot continues its task.[17]
- **Mechatronics Example 2: Autonomous Vehicle Obstacle Avoidance.** An autonomous mobile robot navigating a warehouse uses an ultrasonic sensor to detect obstacles in its path. Its navigation loop contains the following logic: "if the distance reported by the front sensor is less than 25 centimeters, then command the drive motors to stop." If the path is clear and the distance is greater than 25 centimeters, the condition is false, the stop command is ignored, and the robot continues with its default "move forward" instruction that follows the if statement.[19]

3.2 The if-else Statement: The Fork in the Road

The if-else statement, or **double-selection statement**, presents a mandatory choice between two different paths of execution.[1] If the condition is

True, the first block of code is executed. If the condition is False, a second block of code, following the else keyword, is executed instead. With an if-else structure, one of the two blocks is guaranteed to run; the program cannot skip the decision entirely.[16]

- **Mechatronics Example 1: Line-Following Robot Correction.** A simple line-following

robot uses an infrared sensor on its right side to track a black line on a white surface. To stay on the line, it must constantly make corrections. The logic is: "if the right sensor detects the black line (meaning the robot has veered too far left), then execute a sharp right turn to get back on track. else (if the sensor is over the white surface), execute a slight left turn to continue searching for the line." The robot is always performing an active corrective maneuver, ensuring it aggressively tracks the line.[21]

- **Mechatronics Example 2: Industrial Sorting Conveyor.** In a logistics facility, a conveyor system sorts packages based on weight. As a package arrives at a checkpoint, it is weighed. The control logic is: "if the package weight is greater than 5 kilograms, then activate a pneumatic piston to divert the package onto the 'heavy items' conveyor. else, do nothing and allow the package to continue along the main 'light items' conveyor." A decisive action is taken for every single package based on this binary condition.[23]

3.3 The if-elif-else Statement: The Decision Ladder

The if-elif-else statement is a **multiple-selection statement** that allows the program to choose one action from a list of several possibilities.[1] It works like a ladder of decisions. The program checks the first

if condition. If it is True, its code block is executed, and the rest of the ladder is skipped. If it is False, the program moves to the first elif (short for "else if") and checks its condition. This process continues down the ladder. The final, optional else block acts as a default case, which is executed only if all preceding if and elif conditions were False.[16]

- **Mechatronics Example 1: Drone Flight Controller Logic.** A drone's flight controller must continuously manage its state by fusing data from multiple sensors.[27] Its main control loop might contain a decision ladder to prioritize actions: "
if the downward-facing lidar sensor reports a distance to the ground of less than 10 centimeters AND the landing command has been issued, then execute the final landing sequence by cutting motor power. elif the battery voltage sensor reports a level below 15 percent, then override all other commands and initiate the 'return to home' procedure. elif a new GPS waypoint has been received from the ground station, then calculate the new heading and proceed to that waypoint. else, maintain a stable hover." This hierarchy of checks ensures that critical safety conditions (like low battery) are handled with the highest priority.[29]
- **Mechatronics Example 2: Embedded System User Interface.** Consider an industrial machine controlled by a single button. The system needs to respond differently based on how the button is used. The logic for interpreting a button press could be: "if the button press duration is greater than 2 seconds, then initiate the system shutdown procedure. elif the time between two presses is less than 500 milliseconds (a double-click), then

cycle through the machine's diagnostic modes. else (for a single, short press), toggle the main hydraulic pump on or off." This structure allows for complex user interactions to be programmed in a clear and orderly fashion.[31]

These selection statements are more than just tools for branching logic; they are the fundamental building blocks of **Finite State Machines (FSMs)**, a core concept in the design of embedded and mechatronic systems.[32] A system, such as a robot, is always in a particular

state (for example, "IDLE," "NAVIGATING," "ERROR"). The if-elif-else ladder is the very mechanism that evaluates inputs (sensor readings, user commands) to determine when and how to transition from one state to the next.[33] The drone example perfectly illustrates this: each condition is a trigger for a state transition. Understanding this connection elevates a programmer's perspective to that of a systems architect. You are not just writing a decision; you are defining the logical, predictable, and safe behavior of a machine, one state transition at a time.

Chapter 4: The Engine of Repetition - Looping Statements

Repetition statements, or loops, are the workhorses of automation. They allow a program to execute a block of code multiple times without having to write it out repeatedly. This is essential for any task that requires persistence, monitoring, or processing a collection of items. Python provides two primary looping structures: the while statement and the for statement.[1]

4.1 The while Statement: The Conditional Loop

The while statement repeats a block of code *as long as* a specified condition remains True.[1] Before each potential execution of the loop's body, the program re-evaluates the condition. If the condition is still

True, the body is executed again. If the condition becomes False, the loop terminates, and the program continues with the statement following the loop. It is crucial to remember that if the condition is False to begin with, the loop's body will not execute even once.

- **Mechatronics Application: The "Heartbeat" of an Autonomous System.** In robotics

and autonomous systems, the most critical and common application of the while loop is the creation of an infinite main loop, often written as while True:. This structure serves as the system's continuous "heartbeat" or operational cycle.[34] Inside this perpetual loop, the robot endlessly performs its core functions in a cycle:

1. **Read Sensors:** It gathers data from all its sensors—cameras, encoders, gyroscopes, etc.
2. **Process Data:** It uses this data to update its understanding of its own state and the surrounding environment.
3. **Make Decisions:** It employs selection statements (if-elif-else) to decide on the next action.
4. **Command Actuators:** It sends commands to its motors, grippers, and other physical components.
5. Repeat: The loop immediately repeats, ensuring the robot is constantly aware and reactive.
   A line-following robot, for example, uses a while True: loop to continuously check its IR sensors and adjust its motor speeds to stay on the line, never stopping its cycle of sensing and reacting.21

- **Mechatronics Example: Process Control.** In an industrial automation setting, a control system might manage the temperature of a chemical vat. The logic could be: "while the temperature reading from the thermal probe is below 150 degrees Celsius, then keep the heating element energized." As soon as the temperature reaches or exceeds 150 degrees, the condition becomes False, the loop terminates, and the heating element is turned off. This entire while loop would itself be nested inside the main while True: control loop of the Programmable Logic Controller (PLC).

4.2 The for Statement: The Sequential Loop

The for statement is designed to iterate over a sequence of items, such as the elements in a list, or to execute a set number of times.[1] This is the preferred loop when the number of repetitions is known beforehand. It provides a clean and readable way to process each item in a collection, one by one, in order.

- **Mechatronics Example 1: CNC Machining.** A Computer Numerical Control (CNC) mill operates by executing a sequence of instructions from a file known as a G-code file.[37] A Python-based controller for such a machine would first read this file and store each line as an element in a list of commands. The core execution logic would then be a for loop. Verbally: "for each command in the list of G-code commands, then send that command to the motor control board and wait for the machine to confirm the action is complete." This structure guarantees that every machining operation, from moving the tool to changing its speed, is executed in the precise order required to manufacture the

part correctly.[38]

- **Mechatronics Example 2: Robotic Arm Path Planning.** An industrial robot arm is tasked with moving a component from a starting point to a destination, following a carefully planned, collision-free path. This path is often defined as a list of intermediate waypoints (specific coordinates and orientations). The robot's control logic would use a for loop to execute this motion.[40] The logic would be: "
  for each waypoint in the path list, then calculate the required joint angles (inverse kinematics) and command the arm's motors to move to that waypoint." This ensures the arm follows the prescribed path sequentially and accurately.[42]

The choice between a while loop and a for loop reflects two distinct philosophies of automation. A while loop is used for **state-driven** or **event-driven** tasks. Its logic says, "Keep performing this action until the state of the world changes." It is inherently reactive and is used when the number of iterations is unknown, such as waiting for a sensor to trigger or following a line of arbitrary length.[34] In contrast, a

for loop is used for **sequence-driven** or **data-driven** tasks. Its logic says, "Perform this exact process for every single item in this known collection." It is procedural and is used when the number of iterations is finite and known in advance, such as processing all 12 parts in a tray or visiting all four predefined corners of a calibration pattern.[41] Choosing the correct loop is a critical design decision. For systems that must react to an unpredictable environment, the

while loop is the primary tool. For systems that must execute a predefined, repeatable plan, the for loop is the more logical, efficient, and safer choice.

Conclusion: Stacking and Nesting - Building Complexity from Simple Blocks

The true power of the control structures discussed—sequential execution, the three forms of selection, and the two forms of repetition—is realized when they are combined. Every complex behavior exhibited by a sophisticated mechatronic system is built by combining these six fundamental forms of control in just two ways: **control-statement stacking** and **control-statement nesting**.[1] Stacking simply means placing control structures one after another in sequence. Nesting means placing one control structure inside the body of another. This "building block" approach is the essence of structured programming's power and simplicity.

To synthesize these concepts, consider the high-level logic for an autonomous robot in a smart warehouse. Its programming would be a masterclass in stacking and nesting control

statements.

The robot's entire operational life is governed by a main "heartbeat" loop, which is a **while loop** that is set to run forever. Verbally, the pseudo-code would be:

"Begin the main program loop: while True:
Inside this loop, the first stacked action is to check for new tasks. This is a stacked if statement: if a new shipping order has been received:
If an order exists, the robot must process a list of items. This calls for a nested for loop: for each item in the list of items from the order:
For each item, the robot must navigate to its location. This requires another level of nesting, a deeply nested for loop: for each waypoint in the calculated path to the item: then command the robot to move to that waypoint.
Once at the item's location, the robot must decide how to pick it up. This is a nested if-else statement: if the item's weight from the database is greater than 10 kilograms, then activate the heavy-duty grasping procedure, else, activate the standard grasping procedure.
This entire sequence of nested loops and decisions for fetching an item is contained within the first if statement. After that block, another stacked if statement might check the robot's battery status, and so on.
This example clearly demonstrates how all the fundamental control statements are combined to create a sophisticated and autonomous system. The outermost while loop ensures persistent operation. Stacked if statements handle different events like new orders or low battery. Nested for loops execute sequential tasks like following a path or processing a list of items. And nested if-else statements handle decisions that must be made within those tasks.

The principle that any program can be constructed from just six types of control statements combined in only two ways is profound. This simplicity is precisely what makes it possible to build systems of immense complexity with confidence.[1] If the foundational logic relied on unpredictable tools, the complexity would quickly become unmanageable, untestable, and unsafe. A deep, intuitive mastery of these simple, verifiable control statements is therefore not a remedial step in learning to program. It is the single most important skill for unlocking the ability to design, build, and debug the complex, reliable, and safe robotic and automated systems that define the future of mechatronics.

Works cited

1. ch3 part 2.docx
2. The Integration of Advanced Mechatronic Systems into Industry 4.0 for Smart Manufacturing, accessed September 26, 2025, https://www.mdpi.com/2071-1050/16/19/8504
3. The virtues of using goto - rubber duck typing, accessed September 26, 2025, https://rubber-duck-typing.com/posts/2017-04-26-goto-the-marvelous.html
4. MISRA C - Guidelines for the use of the C language in critical systems | arc42 Quality Model, accessed September 26, 2025,

https://quality.arc42.org/standards/misra-c
5. MISRA C & MISRA C++ | Coding Standards For Compliance | Perforce, accessed September 26, 2025, https://www.perforce.com/resources/qac/misra-c-cpp
6. MISRA C - Wikipedia, accessed September 26, 2025, https://en.wikipedia.org/wiki/MISRA_C
7. Programming Languages in Safety-Critical Applications - adesso SE, accessed September 26, 2025, https://www.adesso.de/en/news/blog/programming-languages-in-safety-critical-applications.jsp
8. Heterogeneous Models Integration for Safety Critical Mechatronic Systems and Related Digital Twin Definition: Application to a Collaborative Workplace for Aircraft Assembly - MDPI, accessed September 26, 2025, https://www.mdpi.com/2076-3417/12/6/2787
9. Pick-and-Place Workflow Using Stateflow for MATLAB - MathWorks, accessed September 26, 2025, https://www.mathworks.com/help/robotics/ug/pick-and-place-workflow-using-stateflow.html
10. Simple pick and place program — Pickit 2.4 documentation, accessed September 26, 2025, https://docs.pickit3d.com/en/2.4/robot-integrations/robot-independent/pick-and-place-simple.html
11. Flow chart of the pick and place application system - ResearchGate, accessed September 26, 2025, https://www.researchgate.net/figure/Flow-chart-of-the-pick-and-place-application-system_fig1_381564253
12. Flowchart for Robotic arm operation | Download Scientific Diagram - ResearchGate, accessed September 26, 2025, https://www.researchgate.net/figure/Flowchart-for-Robotic-arm-operation_fig3_320174036
13. Robot Flow Chart Mechatronics [classic] - Creately, accessed September 26, 2025, https://creately.com/diagram/example/i8yufmmv/robot-flow-chart-mechatronics-classic
14. Academics - Oakland University, accessed September 26, 2025, https://www.oakland.edu/academics/
15. Lesson: Conditional Statements - CS-STEM Network, accessed September 26, 2025, https://www.cs2n.org/u/mp/badge_pages/741
16. If Else Statement in Python: Syntax and Examples Explained - Simplilearn.com, accessed September 26, 2025, https://www.simplilearn.com/tutorials/python-tutorial/python-if-else-statement
17. How to Use Conditional Statements in Python for Robotics, accessed September 26, 2025, https://www.awerobotics.com/home/where-to-begin-with-robotics/learn-these-programming-languages-for-robotics/free-beginner-guide-on-python-for-robotics/conditional-statements-in-python-for-robotics/

18. Programming If Else Statements ROBOT C PLTW AUTOMATION & ROBOTICS - YouTube, accessed September 26, 2025, https://www.youtube.com/watch?v=HvHLAUOYjI8
19. Reference - ROBOTC, accessed September 26, 2025, https://www.robotc.net/files/pdf/vex-natural-language/hp_if_else.pdf
20. Python If Else Statements - Conditional Statements - GeeksforGeeks, accessed September 26, 2025, https://www.geeksforgeeks.org/python/python-if-else/
21. Line Follower Robot - Full - Python - Example Project, accessed September 26, 2025, https://ai.thestempedia.com/example/line-follower-robot-full-python/
22. Build a line-following robot - Code Club Projects - Raspberry Pi Foundation, accessed September 26, 2025, https://projects.raspberrypi.org/en/projects/rpi-python-line-following
23. Conveyor programming question : r/PLC - Reddit, accessed September 26, 2025, https://www.reddit.com/r/PLC/comments/1hv6wjs/conveyor_programming_question/
24. Sorting multiple boxes on the conveyor - Advices? | PLCtalk - Interactive Q & A, accessed September 26, 2025, https://www.plctalk.net/forums/threads/sorting-multiple-boxes-on-the-conveyor-advices.92490/
25. Python if, if...else Statement (With Examples) - Programiz, accessed September 26, 2025, https://www.programiz.com/python-programming/if-elif-else
26. if-else if-else statement (Glossary Entry) – Embedded Systems - Blogging with Plymouth University, accessed September 26, 2025, https://blogs.plymouth.ac.uk/embedded-systems/glossary-2/if-else-if-else-statement-glossary-entry/
27. Sensor Fusion Explained: The Future of Embedded Systems - DISTek Integration, accessed September 26, 2025, https://distek.com/blog/sensor-fusion-explained-the-future-of-embedded-systems/
28. Sensor fusion techniques for service robotic positioning and flight in GNSS denied environments using UWB technology - Webthesis, accessed September 26, 2025, https://webthesis.biblio.polito.it/21123/1/tesi.pdf
29. 1.6: Conditionals – Robolink Basecamp, accessed September 26, 2025, https://learn.robolink.com/lesson/1-6-conditionals-cde/
30. Autonomous flight using Python MAVLink library - ArduPilot Discourse, accessed September 26, 2025, https://discuss.ardupilot.org/t/autonomous-flight-using-python-mavlink-library/126457
31. If/else/else if function - Programming - Arduino Forum, accessed September 26, 2025, https://forum.arduino.cc/t/if-else-else-if-function/970925
32. Why state machines? : r/embedded - Reddit, accessed September 26, 2025, https://www.reddit.com/r/embedded/comments/18ko1i5/why_state_machines/
33. Simple state machines. When does switch() beat if - else if - Stack Overflow, accessed September 26, 2025, https://stackoverflow.com/questions/15574989/simple-state-machines-when-doe

s-switch-beat-if-else-if-else
34. Reference - ROBOTC, accessed September 26, 2025, https://www.robotc.net/files/pdf/vex-natural-language/hp_while.pdf
35. While Loop - Carnegie Mellon Robotics Academy, accessed September 26, 2025, http://cmra.rec.ri.cmu.edu/products/teaching_robotc_vex/reference/hp_while.html
36. While Loops in RobotC - YouTube, accessed September 26, 2025, https://www.youtube.com/watch?v=SExk1-R9CFw
37. Learn how to program CNC machines with G-Code - GCodeTutors, accessed September 26, 2025, https://gcodetutor.com/cnc-machine-training/cnc-g-codes.html
38. Loop and condition by g-code command - Stack Overflow, accessed September 26, 2025, https://stackoverflow.com/questions/41471908/loop-and-condition-by-g-code-command
39. CNC G-Code Macro Conditions & Looping - CNC Cookbook, accessed September 26, 2025, https://www.cnccookbook.com/cnc-g-code-macro-conditions-looping/
40. Path Planning for Industrial Robot arms - A Parallel Randomized Approach*, accessed September 26, 2025, https://publikationen.bibliothek.kit.edu/85296/763806
41. Control of industrial robots Motion planning - Paolo Rocco - Politecnico di Milano, accessed September 26, 2025, https://rocco.faculty.polimi.it/cir/Motion%20planning.pdf
42. Multi-Objective Optimal Trajectory Planning for Robotic Arms Using Deep Reinforcement Learning - MDPI, accessed September 26, 2025, https://www.mdpi.com/1424-8220/23/13/5974
43. Programming a Line Follower Robot - EmbedJournal, accessed September 26, 2025, https://embedjournal.com/programming-line-follower-robot/