ch3 part 11 wrapup

A Study Bible for the Mechatronics Engineer: Python Control Structures

Introduction: The Language of Intelligent Machines

Welcome to this in-depth study guide, where we will explore the fundamental concepts of Python's control statements. The material summarized in your textbook is more than just programming syntax; it is the essential grammar we use to communicate with and command intelligent physical systems.[1] In the world of mechatronics and robotics, these control structures are what allow us to translate human intent into precise, automated, and intelligent machine action.

Throughout our exploration, we will frequently return to a core principle of algorithm design: the three phases of execution. Nearly every task a mechatronic system performs, from a simple motor movement to a complex manufacturing sequence, can be broken down into an **initialization** phase, where the system is prepared; a **processing** phase, where the main work is done; and a **termination** phase, where the task is concluded and the system is brought to a safe or ready state.[1] Understanding this universal structure provides a powerful framework for designing robust and predictable automated systems.

Part I: The Logic of Decision-Making – The Robot's Reflexes

Conditional Statements (if, if...else, if...elif...else)

At the heart of any intelligent system is the ability to make decisions. In Python, the if, if...else, and if...elif...else statements are the primary tools for creating this decision-making logic.[2] Think of these statements as the digital equivalent of a biological reflex arc. A sensor provides a stimulus—an input from the physical world—and the conditional statement evaluates this input to trigger a specific, pre-programmed action or response.

Let's begin with the simplest form: the if statement. This is used for a single condition that leads to a single action. Imagine a mobile robot equipped with a forward-facing ultrasonic sensor to avoid collisions. Its core logic, running inside a continuous loop, would be verbally expressed as: "Check the distance measured by the sensor. *If* that distance is less than 25 centimeters, then execute the command to stop the motors".[3] If the condition is not met—that is, if the distance is 25 centimeters or more—the program simply ignores the stop command and continues on. This is a one-to-one relationship between a condition and a potential action.

Now, let's consider a situation that requires a binary choice, which is where the if...else statement excels. Picture a line-following robot designed to navigate a black line on a white surface using a downward-facing infrared sensor. In every moment of its operation, the robot must make a choice: is it on the line or off the line? The logic is: "*If* the sensor detects the black line, then activate the motors to move straight forward. *Else*, if the sensor sees the white surface, activate the motors to turn to the left to find the line again".[2] This structure guarantees that one of two actions is always taken. The robot is never idle; it is either purposefully moving forward or actively correcting its course, which is a fundamental requirement for stable navigation.[5]

For more complex scenarios, we use the if...elif...else structure, which creates a multi-branch decision tree. Consider an automated sorting system on a manufacturing line. Parts move along a conveyor belt and pass under a color sensor. The system must sort them into different bins. The logic would be: "*If* the sensor reads the color 'red', then actuate the first pneumatic piston to push the part into bin one. *Else if* the sensor reads 'blue', actuate the second piston for bin two. *Else if* the sensor reads 'green', actuate the third piston for bin three. *Else*, for any other color or a sensor error, do nothing and let the part travel to a final rejection bin".[2] This allows a single program to handle several distinct possibilities in an organized and efficient manner.

These conditional statements are not just for simple, reactive behaviors. They are the fundamental building blocks of a more advanced concept in control systems engineering: the Finite State Machine. A robot or automated system is always in a particular *state*—for example, 'moving forward', 'turning', or 'stopped'. A sensor reading acts as an *event* that triggers a transition to a new state. The if...elif...else structure is precisely how we implement this state transition logic in software. For our line-following robot, the system transitions from the 'moving forward' state to the 'searching for line' state based on the input from the infrared sensor. This reframes a simple programming construct as a powerful tool for designing complex, predictable, and reliable system behaviors, which is an absolute necessity for advanced mechatronics.

Boolean Operators (and, or, not) – The Bedrock of Industrial Safety

Boolean operators—specifically and, or, and not—are used to combine simple conditions into more complex and nuanced logical expressions.[1] While they are used in all areas of programming, in industrial automation, they are the absolute bedrock of machine safety logic.[6]

The and operator is used when multiple conditions must be true simultaneously for an action to occur. Consider the safety system for a large industrial press. To prevent horrific accidents, the machine must only operate when the operator's hands are verifiably clear of the mechanism. The safety logic is: "The press will activate *if, and only if*, the left-hand activation button is pressed *and* the right-hand activation button is pressed at the same time." We can add even more layers of safety: "*and* the safety cage door sensor indicates the door is closed".[7] The

and operator creates a chain of mandatory conditions, an "interlock," where the failure of any single condition prevents the dangerous action from occurring.

The or operator, conversely, is used when any one of several conditions is sufficient to trigger an action. This is commonly used in emergency stop systems. Imagine a long conveyor belt in a factory with several red emergency stop buttons placed along its length for accessibility. The logic governing the motor is: "The conveyor motor will shut down *if* e-stop button one is pressed, *or* e-stop button two is pressed, *or* e-stop button three is pressed".[7] This ensures that activating any single emergency stop is enough to bring the entire system to a safe state.

The not operator is used to invert a condition, which can often make logic more intuitive. Let's say a green light is used to indicate when it is safe for a technician to enter a robotic work cell. The logic could be stated as: "The green 'safe to enter' light will turn on *if* the robot's motion controller is *not* active *and* the safety gate is confirmed to be closed".[6] Using

not allows us to check for the absence of a condition (motion) rather than the presence of a "stopped" signal, which can be a more robust way to design fail-safe systems.

The implementation of this Boolean logic in safety systems has profound implications that go beyond mere code. Functional safety standards in manufacturing are deeply concerned with mitigating risk, especially the risk of human error.[9] The safety systems we've described are designed to be deterministic and, in critical moments, to override human action. The two-hand safety press is a physical manifestation of a design philosophy that aims to eliminate a specific type of human error. The logic,

(left_button AND right_button), physically guarantees that the operator's hands cannot be in the danger zone when the press activates. This provides a direct, tangible link between the abstract mathematical concepts of Boolean algebra and the real-world engineering of safe, reliable human-machine interaction.

Part II: Mastering Repetition and Automation – The Engine of Industry

The while Loop – The Heartbeat of Continuous Control

The while statement is designed for what is known as **sentinel-controlled repetition**; it creates a loop that continues to execute as long as a specified condition remains true.[1] This behavior makes it the perfect software structure for implementing the continuous control loops that are the very essence of industrial automation.[10]

At its core, automation operates on a simple but powerful paradigm: measure, decide, and actuate. A controller measures some variable in a process (like temperature or position), decides what to do based on that measurement, and then uses an actuator (like a heater or a motor) to affect the process. It then immediately re-measures to see the effect of its action, repeating this cycle endlessly.[10] The

while loop is the software embodiment of this perpetual cycle.

Consider an industrial vat in a chemical processing plant that must be maintained at a precise temperature. A temperature sensor is submerged in the liquid, and a heating element is available to raise the temperature. The control logic, implemented with a while loop, would be: "*While* the reading from the temperature sensor is less than the target setpoint of, say, 80 degrees Celsius, keep the heating element turned on." Inside this loop, the program continuously re-reads the sensor value. As soon as the temperature reaches or exceeds 80 degrees, the loop's condition becomes false, the loop terminates, and the heating element is turned off. This is a classic example of a closed-loop feedback control system.[10]

Another common use for the while loop is in validating user input, as described in your textbook's exercises.[1] Imagine a human-machine interface (HMI) where an operator must choose between option 1 (start process) and option 2 (run diagnostics). To prevent errors, the program must ensure a valid entry. The logic would be: "

While the operator's input is not equal to 1 *and* it is also not equal to 2, display an error message and keep prompting them to enter a correct value." The loop will only terminate when a valid input—the "sentinel" value—is received.

The choice to use a while loop is fundamentally tied to the type of control system being designed. Control engineering distinguishes between open-loop and closed-loop systems.[12] An open-loop system, like a simple timer-based toaster, executes a task for a set duration without any feedback; it doesn't care if the toast is burning.[13] This kind of system might use a different loop structure. A closed-loop system, however, is defined by its use of feedback to adjust its actions. A smart thermostat, a cruise control system in a car, or an advanced motor controller all rely on feedback.[11] The

while loop is the natural and necessary way to implement this. The loop's condition—for instance, while temperature is below setpoint or while motor position error is not zero—*is* the software implementation of the feedback mechanism. Therefore, understanding the while loop is synonymous with understanding the foundation of modern, responsive control systems.

The for Loop – Executing Predefined Sequences

In contrast to the while loop, the for statement is used for **sequence-controlled iteration**.[1] This means it is used when you know in advance exactly how many times you need to repeat an action, or when you want to process each and every item in a known collection, such as a list of coordinates or a series of steps.

A classic mechatronics example is a robotic arm performing a "pick-and-place" operation. Imagine the arm needs to pick up electronic components from a tray that has 10 distinct slots and place them onto a circuit board. Before the program runs, we can define a list containing the precise X, Y, and Z coordinates for each of the 10 slots. The program logic would then be: "*For* each coordinate in our list of slot coordinates, command the robot to move to that position, then activate the vacuum gripper to pick up the component, then move to the predefined drop-off position on the circuit board, and finally deactivate the gripper." Because the number of operations is fixed and known—there are exactly 10 components to move—the for loop is the ideal tool.[14]

Another powerful application is in Computer Numerical Control (CNC) machining. Let's say a CNC milling machine needs to cut a deep groove into a block of aluminum. It cannot cut the entire depth in one go; instead, it must make several shallow passes. The logic, using a for loop combined with the range function, would be: "*For* each pass number, starting from 1 up to and including 5, lower the cutting tool by an additional 0.5 millimeters and then execute the G-code commands that define the path of the groove." This repeats a complex action a fixed number of times to achieve a precise, cumulative physical result.[16]

The distinction between for and while loops reveals a fundamental duality in automation design. The choice of which loop to use reflects the core nature of the task. Are you designing a system that must react continuously to a dynamic and unpredictable environment? If so, a while loop, which bases its continuation on real-time feedback, is the correct choice. A self-driving car navigating traffic would rely on while loops, for example, while no obstacle is detected, continue driving forward. On the other hand, are you designing a system that executes a fixed, deterministic, and repeatable plan? In that case, a for loop is the superior choice. A robot on an automotive assembly line that applies the same 50 spot welds to every car chassis would use a for loop to iterate through a predefined list of 50 weld coordinates. Many complex mechatronic systems use both: a main while true loop that runs for the entire time the machine is powered on, representing its continuous operational state, with nested for loops inside that are called to execute specific, predefined tasks when commanded. Mastering this duality is a key step in architecting sophisticated automation software.

The range() Function – The Engineer's Counter

The built-in range() function is the perfect companion to the for loop. It generates sequences of integers, giving us a simple and efficient way to control exactly how many times a loop iterates.[1] It can be used in three primary ways.

First is the one-argument version, range(stop). This generates a sequence of numbers starting from 0 and going up to, but not including, the stop value. This is ideal for when you simply need to perform a task a specific number of times. For instance, in a product testing lab, an engineer might need to test the durability of an electromechanical solenoid by activating it 10,000 times. The code would read: "*for a counter variable 'i' in the sequence generated by range(10000)*..." and inside the loop would be the commands to energize and then de-energize the solenoid. The loop will run exactly ten thousand times.[17]

Second is the two-argument version, range(start, stop). This allows you to specify both a starting and an ending point for the sequence. This is useful when you are working with items that have specific numerical identifiers. Imagine a machine that needs to perform a quality control check on a batch of products labeled with serial numbers from 101 through 150. The code could be written as: "*for the variable item_number in the sequence generated by range(101, 151)*..." Note that the stop value is 151, because range always excludes the final number. Inside the loop, the item_number variable can be used to log which specific product is currently being inspected.[19]

Third is the powerful three-argument version, range(start, stop, step). The third argument, step, defines the interval between the numbers in the sequence. This has many practical applications. For example, a quality control system on a high-speed production line might not have time to inspect every single item. Instead, it might be programmed to sample every 5th item. The logic would use: "*for item_index in range(0, 1000, 5)*..." This would cause the inspection code inside the loop to run for item 0, item 5, item 10, and so on.[21] The step argument can also be negative, allowing you to count backward. This is useful for processes that happen in reverse stages. For example, to safely depressurize a vessel from 100 PSI down to 0 PSI in controlled increments of 10, the logic would be: "

for pressure_level in range(100, 0, -10)..." Inside the loop, a valve would be opened until the pressure sensor reads the current pressure_level.[18]

Part III: Advanced Flow Control for a Real-World Environment

break and continue – Intervening in Automated Processes

The break and continue statements provide a crucial capability: altering the normal flow of a loop based on conditions that arise during its execution.[1] The

break statement terminates the loop entirely and immediately, while the continue statement skips the remainder of the current iteration and proceeds directly to the next one.[22]

The break statement is essential for handling critical failures. Imagine a batch-processing loop that is programmed to fill 100 bottles with a liquid. A level sensor continuously monitors the main supply tank. The logic would be: "*For each bottle from 1 to 100*, begin the filling sequence. Inside this loop, constantly check the supply tank sensor. *If the sensor reports a critically low level*, print an alert message to the operator's screen and then execute a break statement." This action immediately halts the entire batch process. The loop is exited, preventing the pump from running dry and ensuring no more bottles are partially filled. The program control then moves to the code immediately following the loop, which would typically contain a safe shutdown or maintenance routine.[23]

The continue statement, by contrast, is used for managing non-critical, recoverable errors. Consider a quality control station where a camera system inspects parts for cosmetic defects as they move along a conveyor belt. The logic would be: "*For each part that passes under the camera*, capture and analyze its image. *If a minor, non-critical defect like a scratch is detected*, log that part's serial number to a rejection file and then execute a continue statement." This action immediately stops any further processing for that specific defective part—such as packaging or sorting—and tells the loop to move on to the next part in the sequence. This allows the production line to remain fully operational while still effectively identifying and flagging individual faulty items.[23]

The choice between using break and continue reflects a fundamental design decision in automation engineering related to system resilience. The break statement is a tool for implementing **fail-safe** mechanisms. When a condition arises that makes continued operation impossible, unsafe, or damaging to the equipment, break provides an immediate and orderly exit from the operational loop to a controlled stop state. The continue statement is a tool for building **fault tolerance**. It allows a system to gracefully handle expected, non-catastrophic errors—like a misaligned part, a bad barcode scan, or a minor cosmetic flaw—without bringing the entire production process to a halt.[24] An engineer must analyze potential failures and decide if they are recoverable. An overheating motor controller is a critical error that should trigger a

break. A single unreadable shipping label on a package is a non-critical error that should trigger a continue. This decision directly impacts crucial performance metrics like system uptime, reliability, and overall safety.

Part IV: The Engineer's Toolkit for Precision and Analysis

The Decimal Type – The Mandate for High-Precision Mechanics

Your textbook introduces the Decimal type in the context of precise monetary calculations, but its importance in science and engineering is even more profound.[1] To understand its value, we must first understand the limitation of standard floating-point numbers used by default in Python and most other languages. These numbers are stored in a binary, base-2 format. This means that many common decimal numbers, like 0.1, cannot be represented with perfect accuracy. Instead, they are stored as a very close binary approximation.[26] While this tiny error is irrelevant for many applications, in high-precision mechatronics, these minuscule errors can accumulate through repeated calculations, leading to significant real-world inaccuracies.[26]

A prime example is the process of robotic calibration. A brand-new industrial robot is repeatable—it can return to the same taught point with high precision, often within 0.05 millimeters—but it is not necessarily accurate. Its internal kinematic model does not perfectly match its physical structure due to tiny manufacturing tolerances; a link might be a fraction of a millimeter shorter than specified in the design.[27] Calibration is the process of measuring these physical errors and creating a software compensation model to correct for them.[28]

This process involves commanding the robot to move to a series of precise locations while an external, high-accuracy measurement device, like a laser tracker, records its actual position. The difference between the commanded position and the actual position is the error. Complex kinematic equations, involving matrix multiplication and trigonometric functions, are then used to calculate adjustments to the robot's internal model. Now, consider the problem: "*If we use standard floating-point math for these calculations*, each step introduces a tiny, new rounding error. After performing hundreds of these calculations for dozens of calibration points, the accumulated computational error in the software model can become larger than the physical precision we are trying to achieve," which for many applications is less than 0.1 millimeters.[28]
The

Decimal type solves this problem. It allows us to perform calculations that are exact to a user-defined number of decimal places, such as 28 or more, effectively eliminating computational rounding as a source of physical inaccuracy in the final calibrated system.[32]

This reveals a critical principle in mechatronic design: there is a direct and unbreakable contract between the required physical precision of a machine and the numerical precision of its control software. As manufacturing and scientific applications push into sub-micrometric and even nanometer scales, the physical tolerances become extraordinarily tight.[29] The software that commands these machines

must be able to represent and calculate positions and movements with a level of precision that exceeds the mechanical system's capabilities. Standard floating-point math often breaks this contract. The Decimal type re-establishes it, providing a software tool with the guaranteed precision necessary to control the next generation of ultra-high-precision hardware.

F-Strings – Professional Data Logging and Reporting

F-strings, or formatted string literals, provide an exceptionally clear and concise way to embed variables and expressions directly inside strings.[1] In an industrial automation context, they are indispensable for creating professional, human-readable, and machine-parsable logs and reports.

Imagine the control system for a robotic manufacturing cell. For monitoring and diagnostics, it needs to generate a continuous stream of log data. Using f-strings, we can create highly structured and informative messages. At each cycle of the robot's operation, the program could log a message constructed like this: "an f-string that reads, 'Timestamp: followed by the current_time variable, then comma, Motor_1_Temp: followed by the temp_sensor_1 variable formatted to two decimal places, then the letter C, then comma, Position_Error: followed by the error_value variable formatted to four decimal places, then the letters mm, then comma, Status: followed by the system_status variable.'" This single line of code produces a log entry that is easy for an engineer to read at a glance and is also structured enough for an automated script to parse for data analysis.[34]

F-strings are also perfect for generating summary reports. At the end of a production shift, a manager needs a clear overview of performance. A multiline f-string can be used to build this report: "a multiline f-string that starts with a header, '--- Production Summary ---', followed by a new line. The next line reads, 'Total Parts Produced: followed by the total_parts variable'. The next line, 'Parts Rejected: followed by the rejected_parts variable'. Then, 'Uptime: followed by the uptime_percentage variable formatted to one decimal place, followed by a percent sign'. And finally, 'Average Cycle Time: followed by the avg_time variable formatted to two decimal places, followed by the letter s.'" This creates a clean, well-formatted report suitable for management and quality control personnel.[34]

In the context of Industry 4.0, this capability is more important than ever. Log files are no longer just for debugging by humans; they are a critical source of data for performance analytics, predictive maintenance, and machine learning models. A simple, unformatted log message is of little use to these systems. A well-formatted f-string, however, turns a log file into a structured, time-series dataset. The ability to precisely format numbers, align text, and embed multiple variables transforms a simple text stream into a rich source of operational data. This data can then be ingested by other systems to plot performance trends, predict machine failures—for example, by noticing a gradual rise in motor temperature over several weeks—and ultimately optimize the entire production process. It is worth noting that for very high-frequency logging, the standard logging module's method of passing arguments separately can be more performant, as it defers the string formatting operation until the message is actually written, whereas an f-string is always evaluated immediately.[35] However, for general reporting and status updates, the clarity of f-strings is unparalleled.

The statistics Module – The Key to Modern Quality Control

The Python Standard Library's statistics module provides straightforward functions for calculating fundamental mathematical statistics, including measures of central tendency like the mean, median, and mode.[1] In a modern manufacturing environment, these tools are the foundation of Statistical Process Control, or SPC.

Let's illustrate this with an example. A factory is producing high-precision steel bolts, where the diameter is a critical quality characteristic. A laser micrometer is set up to measure the diameter of every 10th bolt that comes off the line, and these measurements are collected into a list in our Python program.

After a batch of 100 measurements has been collected, the quality control software uses the statistics module to analyze the data. First, it calculates the mean, using the function statistics.mean(diameters). This tells us if, on average, the process is centered on the target specification. For example, if the target diameter is 10.00 mm, but the mean is 10.03 mm, it indicates the process has drifted. Next, it calculates the standard deviation, using statistics.stdev(diameters). This measures the variability or consistency of the process. A low standard deviation means the bolt diameters are all very close to the average, while a high standard deviation indicates inconsistency.[38]

The crucial point here is that this analysis allows engineers to detect a problem *before* any single bolt is actually out of tolerance. A gradual drift in the mean or a slow increase in the standard deviation can signal issues like cutting tool wear, a change in raw material quality, or a machine falling out of calibration. This allows for proactive, scheduled maintenance to correct the process, rather than reactively scrapping a large number of finished parts that fail a final inspection.[40]

This application represents a significant philosophical shift in manufacturing, from simple "Quality Control" to data-driven "Quality Assurance" and "Process Intelligence." Traditional quality control is binary: it inspects a finished part and asks, "Is it good or bad?" Statistical Process Control, as implemented with tools like Python's statistics module, monitors the health of the *manufacturing process itself* in real-time. By tracking statistical trends, the system can predict when a machine will require maintenance or recalibration *before* it begins producing defective products. This is the core concept behind predictive maintenance, a cornerstone of modern smart manufacturing and Industry 4.0.

Part V: The Engineering Mindset – From Concept to Code

Top-Down Design and Pseudocode – Taming Complexity

Top-down, stepwise refinement is a powerful design methodology where you begin with a single, high-level description of a problem and systematically break it down into smaller, more detailed, and more manageable sub-problems. **Pseudocode** is the language we use during this process—an informal, human-readable description of an algorithm's logic, written without adherence to the strict syntax of any particular programming language.[1]

In mechatronics, these practices are not just helpful; they are essential. Mechatronic systems are, by their very nature, complex integrations of mechanical hardware, electronic sensors and actuators, and sophisticated software.[41] Attempting to build such a system from the "bottom up"—writing small, isolated pieces of code without a master plan—is a recipe for failure at the integration stage, where all the pieces must come together and work in harmony.[43] The top-down approach forces the engineer to maintain a holistic view of the entire system from the very beginning.

Let's walk through the design of an automated warehouse robot using this methodology.

Level 1 (The "Top"): The highest-level, most abstract goal is simply: "The robot must retrieve a specified part from a warehouse shelf and deliver it to a human packing station".[44]

Level 2 (First Refinement): We break this single, complex task into a logical sequence of major behaviors:

1. Receive a command with the part's location (aisle and shelf number).
2. Navigate from the home base to the correct aisle and shelf.
3. Actuate the lift and gripper mechanism to retrieve the part bin.
4. Navigate from the shelf to the designated packing station.
5. Deposit the part bin at the station.
6. Navigate back to the home base to await the next command.

Level 3 (Second Refinement - Pseudocode): Now, we take just one of these sub-problems, "Navigate to the correct aisle and shelf," and refine it further using pseudocode. This allows us to think through the detailed logic. The pseudocode might look like this:

"Start a WHILE loop that continues as long as the robot's current location is NOT the target shelf location. Inside this loop, continuously read the floor-line sensors to follow the main pathway. Also, continuously read the RFID tags embedded in the floor to identify which aisle the robot is currently passing. Check IF the current aisle number from the RFID tag matches the target aisle number from the command. IF they match, then execute a turn into that aisle. Now, move forward WHILE using a sensor to count the shelf markers on the racks. Check IF the current shelf marker count matches the target shelf number. IF it does, then STOP the motors and execute a BREAK command to exit the main navigation loop, because the destination has been reached".[44]

This pseudocode clearly defines the required logic using the very control structures we have been studying—while and if—but does so in a way that is completely independent of Python's specific syntax. This process is repeated for every sub-problem from Level 2, breaking each down until every component of the system's behavior is described by a simple, unambiguous piece of logic that can be translated directly into final code.

This methodology is far more than just a good programming practice. In a field as interdisciplinary as mechatronics, top-down design and pseudocode are indispensable tools for systems engineering and project management. The true complexity of mechatronics arises from the intricate interactions *between* different domains.[41] A software decision, such as how frequently to poll a sensor, has direct consequences for the electrical design (e.g., the required data bus speed) and the mechanical performance (e.g., the system's physical reaction time). By forcing the design process to start from the top, engineers must define the high-level requirements and interfaces for each subsystem before implementation begins. The pseudocode then acts as a clear, written "contract" between the different engineering teams. It specifies exactly what the software will do in response to certain sensor inputs and precisely what commands it will issue to the actuators, ensuring that when the hardware and software are finally integrated, they work together seamlessly. This structured approach is the primary strategy for preventing the costly and time-consuming integration failures that so often plague complex engineering projects.

Works cited

1. ch3 part 11 wrapup.docx
2. How to Use Conditional Statements in Python for Robotics - Awe ..., 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/
3. if Statements with Natural Language - ROBOTC, accessed September 26, 2025, https://www.robotc.net/files/pdf/vex-natural-language/hp_if_else.pdf
4. if-else Statement - Carnegie Mellon Robotics Academy, accessed September 26, 2025, http://cmra.rec.ri.cmu.edu/products/teaching_robotc_vex/reference/hp_if_else.ht

ml
5. Lesson: Conditional Statements - CS-STEM Network, accessed September 26, 2025, https://www.cs2n.org/u/mp/badge_pages/741
6. Machine Safety - Turck Banner (Pty) Ltd, accessed September 26, 2025, https://www.turckbanner.co.za/en/machine-safety-37293.php
7. IAT LESSON 06 Logic Gates Boolean Logic in PLCs | PDF - Scribd, accessed September 26, 2025, https://www.scribd.com/document/853582006/IAT-LESSON-06-Logic-Gates-Boolean-Logic-in-PLCs
8. 10 BOOLEAN LOGIC CIRCUITS (CIE) - Computer Science Cafe, accessed September 26, 2025, https://www.computersciencecafe.com/10-boolean-logic-circuits-cie.html
9. Operators and Safety Instrumented Functions (SIF) - Risknowlogy, accessed September 26, 2025, https://risknowlogy.com/articles/detail/14077/
10. Control loop is automation essence - Control Engineering, accessed September 26, 2025, https://www.controleng.com/control-loop-is-automation-essence/
11. Control loop - Wikipedia, accessed September 26, 2025, https://en.wikipedia.org/wiki/Control_loop
12. Control Loops - Game Manual 0, accessed September 26, 2025, https://gm0.org/en/latest/docs/software/concepts/control-loops.html
13. Open loop vs. closed loop: which system is better? - Motion Automation Intelligence, accessed September 26, 2025, https://ai.motion.com/open-loop-vs-closed-loop-which-system-is-better/
14. How to use for loops in Robot Framework? | BrowserStack, accessed September 26, 2025, https://www.browserstack.com/guide/robot-framework-for-loop
15. Dorna Robotic Arm: Python API and Programming Tutorial - YouTube, accessed September 26, 2025, https://www.youtube.com/watch?v=5cwC2Vu4erc
16. WHILE DO LOOP IN A CNC MILL PROGRAM - YouTube, accessed September 26, 2025, https://www.youtube.com/watch?v=KVI4EJgy3JY
17. Python range(): Represent Numerical Ranges - Real Python, accessed September 26, 2025, https://realpython.com/python-range/
18. Python range() Function, accessed September 26, 2025, https://cs.stanford.edu/people/nick/py/python-range.html
19. Python range() function - GeeksforGeeks, accessed September 26, 2025, https://www.geeksforgeeks.org/python/python-range-function/
20. Python range() Function [Python Tutorial] - Mimo, accessed September 26, 2025, https://mimo.org/glossary/python/range-function
21. Python Range() Function Tutorial | DataCamp, accessed September 26, 2025, https://www.datacamp.com/tutorial/python-range-function
22. Loop Control Statements - Python - GeeksforGeeks, accessed September 26, 2025, https://www.geeksforgeeks.org/python/break-continue-and-pass-in-python/
23. Break and Continue Statements in Python - Saurabh Adhau's Blog, accessed September 26, 2025, https://devopsvoyager.hashnode.dev/break-and-continue-statements-in-python

24. Python - Try, Except, Finally, Continue, Break in Loop Control Flow ..., accessed September 26, 2025, https://www.devcuriosity.com/manual/details/python-try-except-finally-continue-break-loops/
25. Make python code continue after exception - Stack Overflow, accessed September 26, 2025, https://stackoverflow.com/questions/18994334/make-python-code-continue-after-exception
26. decimal — Decimal fixed-point and floating-point arithmetic ..., accessed September 26, 2025, https://docs.python.org/3/library/decimal.html
27. Calibration and Precision Manufacturing, accessed September 26, 2025, http://www-cdr.stanford.edu/~cutkosky/me319/cal.pdf
28. Absolute Accuracy Robot Calibration | Robotic Calibration Solutions, accessed September 26, 2025, https://dynalog-us.com/application/absolute-accuracy-robot-calibration/
29. (PDF) Ultra-high-precision industrial robots calibration - ResearchGate, accessed September 26, 2025, https://www.researchgate.net/publication/224252315_Ultra-high-precision_industrial_robots_calibration
30. How to achieve robot accuracy of less than 0.1 millimeters - KEBA, accessed September 26, 2025, https://www.keba.com/en/news/industrial-automation/robot-accuracy
31. Joint Calibration Method for Robot Measurement Systems - PMC, accessed September 26, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC10490635/
32. PEP 327 – Decimal Data Type | peps.python.org, accessed September 26, 2025, https://peps.python.org/pep-0327/
33. Precision Handling in Python - GeeksforGeeks, accessed September 26, 2025, https://www.geeksforgeeks.org/python/precision-handling-python/
34. Python f-string: A Complete Guide | DataCamp, accessed September 26, 2025, https://www.datacamp.com/tutorial/python-f-string
35. logging-f-string (G004) | Ruff - Astral Docs, accessed September 26, 2025, https://docs.astral.sh/ruff/rules/logging-f-string/
36. Python 3.7 logging: f-strings vs % [duplicate] - Stack Overflow, accessed September 26, 2025, https://stackoverflow.com/questions/54367975/python-3-7-logging-f-strings-vs
37. statistics — Mathematical statistics functions — Python 3.13.7 documentation, accessed September 26, 2025, https://docs.python.org/3/library/statistics.html
38. Mastering Python's Built-in Statistics Module: A Complete Guide to ..., accessed September 26, 2025, https://www.kdnuggets.com/mastering-pythons-built-in-statistics-module
39. Various Uses of Python Statistics Module & Its Functions - Analytics Vidhya, accessed September 26, 2025, https://www.analyticsvidhya.com/blog/2024/01/various-uses-of-python-statistics-module-its-functions/
40. Production Quality Control Analysis & EDA - Kaggle, accessed September 26,

2025,
https://www.kaggle.com/code/melissamonfared/production-quality-control-analysis-eda

41. Survey of Mechatronic Techniques in Modern Machine Design - ResearchGate, accessed September 26, 2025, https://www.researchgate.net/publication/233980711_Survey_of_Mechatronic_Techniques_in_Modern_Machine_Design
42. Design of mechatronics systems | DOCX | Physics | Science, accessed September 26, 2025, https://www.slideshare.net/slideshow/design-of-mechatronics-systems-179294409/179294409
43. Is is preferable to design top down or bottom up? - Software Engineering Stack Exchange, accessed September 26, 2025, https://softwareengineering.stackexchange.com/questions/134633/is-is-preferable-to-design-top-down-or-bottom-up
44. Teaching Pseudocode and Flowcharts - Carnegie Mellon University, accessed September 26, 2025, https://www.cmu.edu/roboticsacademy/roboticscurriculum/VEX%20Curriculum/files/pseudocode_flowcharts.pdf