ChatGPT

Python Control Flow and Data Fundamentals

Conditional Statements

Conditional statements allow a program to execute different code blocks based on boolean conditions [1]. In Python, these use the keywords `if`, `elif`, and `else`. The basic syntax is:

```
if condition:
    # runs if condition is True
elif another_condition:
    # runs if first condition was False and this is True
else:
    # runs if all above conditions were False
```

Python evaluates each condition in order, executes the first block whose condition is true, and skips the rest [2] [3]. For example, in an `if-elif-else` chain for grading, once a true `if` or `elif` is found, subsequent checks are not performed [3].

Flowchart of a simple `if` *statement: The condition is evaluated, and if true one branch is taken, otherwise it skips to the next step.* As shown above, the flow proceeds by evaluating the condition in a diamond; if it's true the program follows one path, and if it's false it bypasses that block. You must use a colon (`:`) at the end of each condition line, and indent (typically 4 spaces) the code block beneath it [4] [2].

Flowchart illustrating an `if-else` *statement: either the* `if` *block executes (if true) or the* `else` *block runs (if false).* The `else` block provides a fallback that executes when the `if` condition is false [5]. Comparison operators like `==`, `!=`, `<`, `>`, `<=`, `>=` are commonly used in conditions [6]. Logical operators (`and`, `or`, `not`) combine conditions; for example, `if age >= 18 and has_id:` checks that **both** sub-conditions hold [7] [8].

Flowchart of an `if-elif-else` *ladder: multiple conditions are checked in sequence.* The keyword `elif` stands for "else if" and avoids deeply nested `if` blocks [9]. For instance, a grading system might use `elif` to check ranges of scores: once one `elif` branch is taken, Python skips the remaining `elif` and `else` clauses [3]. If none of the conditions match, the final `else` block executes. Nested conditionals are also possible (an `if` inside another `if`), allowing finer-grained checks.

Comparison and logical operators: Conditions usually involve comparison operators, e.g., `x == 5`, `x != y`, `x > 0`, etc. [6]. Logical operators can combine these, such as `if (x > 0) and (y < 10):`. The `not` operator negates a condition (e.g., `if not done:`).

Common mistakes: Beginners often forget the colon `:` after an `if` or `elif` line, use `=` instead of `==` for comparisons, or mis-indent the blocks. Python does **not** use braces `{}` for blocks; indentation is mandatory. Also avoid writing multiple separate `if` statements where an `elif` chain is intended, because separate `if`s will all be evaluated, potentially causing logic errors.

Examples: Real-world scenarios abound. For instance, a login check might use:

```
if username == "admin" and password == "1234":
    print("Login successful")
else:
    print("Access denied")
```

Or a weather check:

```
if temp > 30:
    print("It's hot outside")
elif temp >= 20:
    print("Weather is pleasant")
else:
    print("It's cold")
```

These are typical `if-elif-else` patterns.

DSA-style problems: Conditionals often appear in algorithms. For example, checking even/odd is simply `if n % 2 == 0:`, finding primes uses a loop with an `if n % i == 0` check, and FizzBuzz uses: if divisible by 3 and 5 print "FizzBuzz", elif by 3 print "Fizz", etc. These patterns rely on sequential condition checks [3] .

Loops and Iteration

Loops repeat a block of code multiple times, typically to process items in a sequence or until a condition changes. Python provides two main loop constructs: `for` **loops** and `while` **loops**. A `for` loop iterates over the items of any iterable (like lists, strings, or ranges) [10] . A `while` loop repeats as long as a specified condition remains true [11] .

Flowchart showing a simple `for` *loop: it initializes* `i = 0`*, checks the loop condition, executes the body, increments* `i`*, and repeats until the end.* The above diagram describes a loop from `0` to `9` printing “looping”. In Python, you often use `for i in range(10): ...`, which iterates `i` from 0 up to 9 [12] . The loop terminates when the sequence is exhausted. Importantly, Python’s `for` loop does not require a separate index variable; it directly fetches each element from the iterable [10] .

Break and continue statements modify loop flow:
- `break` immediately exits the closest enclosing loop [13] .
- `continue` skips the rest of the current iteration and moves to the next loop cycle [14] .
For example, `for x in items: if x < 0: break` stops the loop on a negative number [13] , whereas `if x == skip: continue` would skip processing a specific element [14] .

Flowchart of a `while` loop: the program repeatedly asks for input (or increments a counter) until the condition becomes false. The above flowchart is an example where the loop continues until a positive number is entered. A `while` loop in Python has the form:

```
while condition:
    # code
```

It requires an explicit update to eventually make the condition false, or else it will loop indefinitely [15]. For example:

```
i = 1
while i < 6:
    print(i)
    i += 1
```

prints 1 through 5, then stops once `i` is no longer less than 6 [11].

Python also supports an `else` clause on loops: this block executes once if the loop **completes normally** (no `break` encountered). For instance, a `for-else` structure runs the `else` after the loop if no `break` was triggered [16]. Similarly, a `while-else` runs its `else` when the condition becomes false (again, only if not broken out) [17].

Nested loops: You can place a loop inside another (nested loops). In that case, the inner loop runs fully on each iteration of the outer loop [18]. For example, iterating over rows and columns of a matrix requires nested loops (an inner loop inside an outer loop) [18].

Iterator tools: Python makes iteration easier with helpers: `range(start, stop, step)` to generate sequences of numbers [12]; `enumerate(seq)` to loop with indices; `zip(a, b)` to loop over pairs; and `reversed(seq)` to go backwards. List comprehensions provide a concise way to build lists from iterables with optional filtering [19]. For example: `[x for x in nums if x % 2 == 0]` collects all even numbers. Generator expressions (using parentheses) similarly create iterators lazily, saving memory since they yield one item at a time [20].

Variables and Expressions

A **variable** is a name that refers to a value [21]. For example, `x = 10` assigns the integer `10` to the variable `x`, and `name = "Alice"` assigns a string to `name`. Assignment uses the single equals sign `=`, which links the name on the left to the value on the right; this is different from the comparison operator `==` [22]. Variables are dynamically typed in Python: you do not declare a type, and a variable can be reassigned to objects of different types (though that is usually discouraged for clarity).

Expressions and operators: An expression combines values and operators to produce a new value. Common arithmetic operators include `+`, `-`, `*`, `/` (division), `//` (floor division), `%` (modulo), and `**` (exponent) [6]. For instance, `a + b`, `a * b`, and `(a + b) * c` are arithmetic expressions; Python respects standard precedence (multiplication before addition, etc.) unless parentheses override it. Logical, comparison, identity, and membership operators (`and`, `or`, `not`, `==`, `!=`, `<`, `>`, `is`, `in`, etc.) produce boolean results and can be combined in expressions. For example, `age >= 18 and not is_minor` might combine comparisons and logical operators.

Shorthand assignment: You can update variables concisely: `n += 5` is shorthand for `n = n + 5`, and similarly `n *= 2`, etc. Multiple assignment is supported: `x, y, z = 1, 2, 3` assigns three values at once, and swapping two variables can be done as `x, y = y, x`.

Type conversion: To convert types explicitly, Python provides functions like `int()`, `float()`, `str()`, etc. For example, `int("123")` yields the integer `123`, and `float("3.14")` gives `3.14` [23]. Invalid conversions raise `ValueError`, so you can use `try/except` to handle that safely.

Mutability: Some objects are *immutable* (unchangeable): for example, `int`, `float`, `str`, and `tuple` types. Others are *mutable*: e.g., `list`, `dict`, and `set`. This means you cannot alter an immutable object's value once created (e.g., you cannot change a character of a string), whereas you can modify a list or dict in place. Understanding this distinction is important: reassigning a variable always makes it reference a new object, but mutating changes the object itself (only possible with mutable types).

Storing in a database (example): You can store Python variables in databases by serialization. For instance, converting a Python dictionary to a JSON string with `json.dumps()` and saving it to a TEXT field in SQLite. When reading back, you parse the JSON to reconstruct the dictionary. Numeric and string values can be stored directly, but complex objects (lists, dicts) typically need serialization. The code snippet in the content shows using SQLite: creating a table, inserting values as text, and later fetching and converting them back (e.g., using `int()`, `float()`, or `json.loads()`) [22] [24].

Local vs Global Variables

Scope: Variables defined at the top level of a module are *global*: they exist for the program's entire runtime and can be read (and, with care, modified) inside any function. Variables defined inside a function are *local* to that function: they exist only during the function's execution and cannot be accessed outside it [25] [26]. For example, `x = 10` outside any function is global, whereas `x = 5` inside `def func():` is local to `func`.

Lifetime: A global variable is created when the program starts and lasts until it ends. A local variable is created anew each time its function is called and is destroyed when the function returns [27].

Modifying globals: By default, a function reading a global variable can do so without any declaration. However, to reassign a global variable inside a function, you must declare it `global` in that function; otherwise, assignment will create a new local variable of the same name [28] [29]. In the provided code example, using `global_list.append(...)` works without `global` because it mutates a global list (mutating a mutable global does not require the `global` keyword), but reassigning a global name would require the keyword [27].

Shadowing: If a local and global variable share the same name, the local one *shadows* the global inside the function. The global remains unchanged unless explicitly declared global. For instance, in the GfG example, a function `g()` defines its own `a = 2`, so the global `a` is unaffected [30].

Performance: Accessing local variables is typically faster than globals, due to how Python handles namespaces; lookups go local → global → builtins [27]. However, this is a minor point for most applications.

Best practices: Prefer using local variables for temporary values in functions, which makes code modular and avoids unintended interactions. Use global variables sparingly (often only for constants or configuration) to reduce bugs. The LEGB rule (Local → Enclosing → Global → Builtins) governs variable lookup.

Each of these topics—conditional statements, loops, variables, scope—is fundamental to writing effective Python code. Practice using them, and consider flowcharts or pseudocode to plan logic. The visual diagrams above illustrate control flow, but in code you verify correctness with test examples (such as the provided assertions in the original modules).

Sources: Authoritative Python tutorials and references provide detailed explanations of these concepts [1] [9] [10] [13] [21] [25]. The embedded flowcharts are illustrative diagrams from Python learning resources.

---

[1] [2] [3] [4] [5] [6] [7] [8] Python if, else, and elif Conditional Statements: A Complete Guide
https://inventivehq.com/blog/python-if-else-and-else-if-conditional-statements

[9] 4. More Control Flow Tools — Python 3.14.0 documentation
https://docs.python.org/3/tutorial/controlflow.html

[10] [12] [13] [14] [16] [18] Python For Loops
https://www.w3schools.com/python/python_for_loops.asp

[11] [15] [17] Python While Loops
https://www.w3schools.com/python/python_while_loops.asp

[19] [23] Python - List Comprehension
https://www.w3schools.com/python/python_lists_comprehension.asp

[20] Python | Generator Expressions - GeeksforGeeks
https://www.geeksforgeeks.org/python/generator-expressions/

[21] [22] [24] 2.7. Variables — Foundations of Python Programming
https://runestone.academy/ns/books/published/fopp/SimplePythonData/Variables.html

[25] [26] [27] [28] [29] [30] Global and Local Variables in Python - GeeksforGeeks
https://www.geeksforgeeks.org/python/global-local-variables-python/