3.1 Introduction to Flow of Control

In any program, instructions by default run in sequence: first line, next line, next, and so on. This is **sequential execution**. But real problems require deviations: you need to decide between alternatives, repeat tasks until some condition holds, or skip parts of logic. Flow-of-control mechanisms (also called control structures) let you redirect program execution beyond pure linear order.

Flow of control is central to algorithm design. It lets code respond differently based on data, repeat operations, and handle branching. Without it, programs would be inflexible.

There are three broad categories:

1. **Selection / Decision** (branching) — choose among multiple paths (e.g. if / elif / else, match)
2. **Repetition / iteration** — repeat a block multiple times (e.g. for, while)
3. **Jump / control statements** — inside loops, one might break out, skip iteration, or place placeholders (e.g. break, continue, pass)

In Python's official tutorial, control flow is a named section covering if statements, loops, comprehension, exception handling, return, etc.

Why flow control matters

- **Dynamic decision-making**: For example, a banking app must check if a user's balance is sufficient before debiting.
- **Repetition**: e.g. processing every order in a list, reading lines from a file until EOF, polling a sensor until a threshold is reached.
- **Efficiency / early exit**: Sometimes you find what you want early—no need to continue scanning.
- **Code modularity and clarity**: Avoid huge monolithic scripts; break logic into branches and loops for maintainability.

Example skeleton (pseudocode)

```
# sequential portion
x = get_input()

# selection
if x < 0:
   handle_negative()
elif x == 0:
   handle_zero()
else:
   handle_positive()

# repetition
for i in range(10):
   process(i)

# inside repetition, you may break or continue
```

Real-life scenario: Web request handler

Consider a web API endpoint receiving a JSON payload. The handler might:

- Check if the JSON has required fields (selection)

- For each item in a list inside the payload, run some operation (repetition)
- If some item is invalid, skip processing that (continue)
- If a critical error is detected, abort processing further (break or exception)

Flow-of-control structures allow that logic to be expressed cleanly instead of manual “goto” hacks.

Points summary (expanded)

- Flow-of-control determines the **order** of execution, not just the content.
- Sequential is the default; to deviate, use control structures.
- Selection handles conditional branching, repetition handles loops, jump statements give fine grain control within loops.
- Mastering flow control is essential before moving to functions, recursion, data structures, etc.
- All higher-level logic in real software (validation, data processing, event loops) depends heavily on flow control.

3.2 Selection

Selection (or branching) means choosing which block(s) to run depending on conditions. It’s the “if this, else that” logic. Python offers if, elif, nested if, and from Python 3.10 onward, match (pattern matching).

If statement

Syntax:

```
if condition:
   # code_block
```

- condition is a Boolean expression (True/False).
- If condition is True, the indented block runs.
- If False, it is skipped entirely.

Example:

```
temp = 15
if temp < 20:
   print("It's cold today")
```

Real-world use: checking authentication—if user token is valid, proceed; otherwise reject.

Elif statement (else-if)

When multiple conditions might apply, elif lets you test them in series.

Syntax:

```
if cond1:
   ...
elif cond2:
   ...
elif cond3:
   ...
else:
   ...
```

- The first true branch runs, and subsequent checks are skipped.
- else is optional, used when none of above conditions hold.

Example:

```
score = 82
if score >= 90:
    grade = "A"
elif score >= 75:
    grade = "B"
elif score >= 60:
    grade = "C"
else:
    grade = "F"
print("Grade:", grade)
```

Real-world: tiered pricing or discounts—if order>1000, apply 20% discount, elif >500, 10%, else none.

Nested if statements

You can place an if inside another. Useful when second-level decisions depend on first being true.

Example:

```
x = 12
if x > 0:
    if x % 2 == 0:
        print("Positive even")
    else:
        print("Positive odd")
else:
    print("Non-positive (zero or negative)")
```

Real-world: login logic—if user exists, then check password; if password correct, check role, etc.

Caution: deep nesting reduces readability. Use elif, functions, or early returns to flatten.

Match statement (pattern matching; Python 3.10+)

The match statement is more powerful than a traditional switch/case. It can destructure data, match types, constants, sequences, and use "guards".

Syntax:

```
match subject:
    case pattern1:
        ...
    case pattern2:
        ...
    case _:
        ...
```

- subject is matched against each pattern.
- The first pattern that matches executes.
- _ is a catch-all default.

Example:

```
command = ("move", 10, 20)
match command:
    case ("move", x, y):
        print(f"Move to {x}, {y}")
    case ("stop",):
        print("Stop command")
    case _:
        print("Unknown command")
```

Real-world: parsing structured input or commands (e.g. in CLI apps, network protocols, AST processing). match replaces many if/elif constructs when data is structured.

Extended features (from docs)

- Patterns can be nested, include wildcards, OR (|), value binding, etc.
- Guards: you can add if conditions to a case.
- Helps to keep code clean when handling complex variant types or messages.

Summary of selection

- if handles basic one-branch decision.
- elif supports multiple mutually exclusive conditions.
- nested if handles hierarchical decisions but may reduce readability if overused.
- match is expressive and suited for pattern-based branching (introduced in Python 3.10).
- Real-world usage spans validation, user input, protocol parsing, branching logic in business rules.

3.3 Repetition

Repetition (iteration) allows executing a block multiple times. Rather than writing code repeatedly, loops help you generalize over data. Python supports for, while, and nested loops.

For loop

Syntax:

```
for variable in sequence:
    # block
```

- sequence can be a list, tuple, string, range, or other iterable.
- Each iteration, variable takes next value from sequence.
- Loop ends when sequence is exhausted.

Example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print("I like", fruit)
```

Or:

```
for i in range(5):
    print(i)  # prints 0,1,2,3,4
```

Real-world: iterating over database records, processing each user, computing over list elements.

While loop

Syntax:

```
while condition:
    # block
    # ensure eventually condition becomes False
```

- condition is a Boolean expression checked before each iteration.
- Block runs as long as condition is True.

- Must include logic to break or change condition, or else infinite loop.

Example:

```
count = 0
while count < 3:
    print("Attempt", count + 1)
    count += 1
```

Real-world: reading inputs until valid, looping until a sensor reading crosses threshold, menu loop in CLI until user exits.

Nested loops

You can place loops inside loops. The inner loop completes fully for each iteration of the outer loop.

Syntax:

```
for i in range(3):
    for j in range(2):
        print(i, j)
```

Output:

```
0 0
0 1
1 0
1 1
2 0
2 1
```

Real-world: processing 2D grids, matrices, tables, nested structure traversal (e.g. list of lists), generating combinations (cartesian product).

Caveats and complexity

- **Time complexity multiplies**: nested loops of sizes m and n lead to ~m × n operations.
- **Careful with while loops**: risk of infinite loop if condition never falsifies.
- **Break/continue inside loops** can affect inner or outer loops—need careful control.

Example: combining loops and selection

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
for row in matrix:
    for val in row:
        if val % 2 == 0:
            continue  # skip even
        print(val)
```

This prints only odd numbers from the 2D structure.

Extended feature: else on loops

Python supports else clauses with loops: the else block runs if the loop finishes normally (no break). (

Example:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n // x}")
            break
    else:
        # reaches here if no break — meaning prime
        print(f"{n} is prime")
```

Summary of repetition

- for is used when you know or have an iterable or finite set.
- while is used when continuation depends on dynamic condition.
- nested loops allow multidimensional or combinatorial processing.
- loops + selection + jump statements produce powerful behavior.
- use else on loops carefully when you want logic upon no break.

3.4 break, continue, and pass Statements

Inside loops, sometimes the default flow (through each iteration fully) is not what we want. Python provides three control statements to tweak loop behavior: break, continue, pass. has a dedicated article on this.

break

break causes immediate exit from the **nearest enclosing loop** (for or while). Control moves to the first statement after that loop.

Syntax:

```
break
```

Example (for):

```
for i in range(10):
    if i == 5:
        break
    print(i)
# prints 0,1,2,3,4 then exits
```

Example (while):

```
i = 0
while i < 10:
    if i == 3:
        break
    print(i)
    i += 1
# prints 0,1,2
```

Real-world: search through a list of user reco0rds—once you find the matching user, break; you don’t continue scanning.

continue

continue skips the rest of the current iteration and jumps to the next iteration, re-evaluating loop condition.

Syntax:

```
continue
```

Example:

```
for i in range(5):
    if i == 2:
        continue
    print(i)
# prints 0,1,3,4 (skipping i = 2)
```

In a while context:

```
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    print(i)
```

Real-world: processing a batch of transactions, skipping invalid ones (continue), but not stopping entire loop.

example: skipping letters 'e' or 's' in a loop over a string.

pass

pass is a null operation. It does nothing at runtime, but syntactically fills a place where a statement is required.

Syntax:

```
pass
```

Example:

```
for i in range(3):
    if i == 1:
        pass  # placeholder
    print(i)
# prints 0,1,2
```

You may use pass in empty function bodies, class stubs, or placeholders for future logic.

Real-world: scaffolding code structure—you define function shells and fill later.

Interactions, nested loops, and advanced scenarios

- break only exits innermost loop. To break out of multiple nested loops, you often need a **flag** variable or wrap loops inside a function and return.
- continue only affects the current loop level.
- If break is used, a loop's else block is skipped. If loop ends normally, else executes.
- Using exceptions to break nested loops is possible but not idiomatic—exception paths are meant for exceptional conditions, not routine control flow.
- break, continue, pass must appear literally inside loop; they cannot be triggered inside called functions to affect outer loops.

Combined example with all three

```
def find_first_positive_even(nums):
    for x in nums:
        if x < 0:
            continue            # skip negatives
        if x == 0:
            pass               # placeholder logic for zero (we ignore)
        if x % 2 == 0:
            print("Found:", x)
            break              # exit loop once first positive even is found
    else:
```

```
    print("No positive even found")

# test
find_first_positive_even([-3, -2, 1, 0, 5, 8, 10])
# output: Found: 8
```

Here:

- negative numbers are skipped (continue)
- for zero we do nothing special (pass)
- once we hit first positive even (8), we break out
- else is not executed (because break happened)

Summary points

- break ends loops early when a condition is met
- continue skips the rest of this iteration, moves to next
- pass is placeholder, does nothing
- Use else on loops to run code if loop completed without break
- Multi-level break requires special handling (flags or function returns)
- These are critical tools for real-world loop control (data filtering, search, validation, pagination)