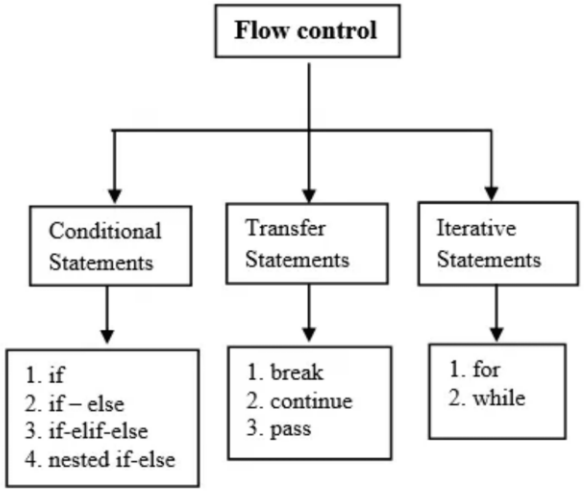
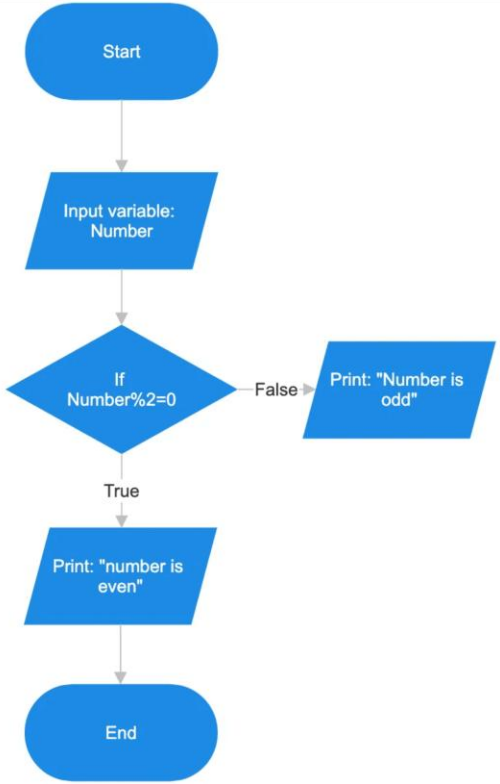
Python Control Flow: Thinking in Code

Mastering the Logic of Decision and Repetition





If-Else Logic: Directing the Program's Flow

Conditional Execution: Code blocks run only if a condition is met.

The `if` Statement: The entry point for decision-making.

The `elif` Statement: Allows for checking multiple, mutually exclusive conditions sequentially.

The `else` Statement: The fallback block that executes if all preceding conditions are false.

Indentation is Key: Python uses indentation (4 spaces) to define code blocks, not braces.

```
score = 85
 if score >= 90:
 print("Grade A")
 elif score >= 80:
 print("Grade B")
 else:
 print("Grade C")
```

Truthy vs. Falsy: The Implicit Boolean Check

Every Object Has a Boolean Value: In Python, any object can be evaluated in a boolean context (like an `if` statement).

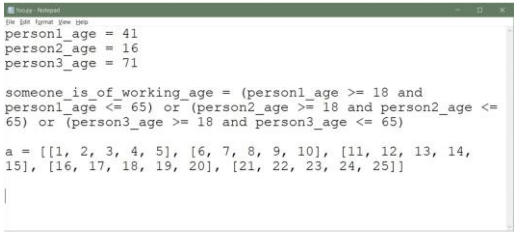
Falsy Values: Objects that evaluate to `False`. These include:

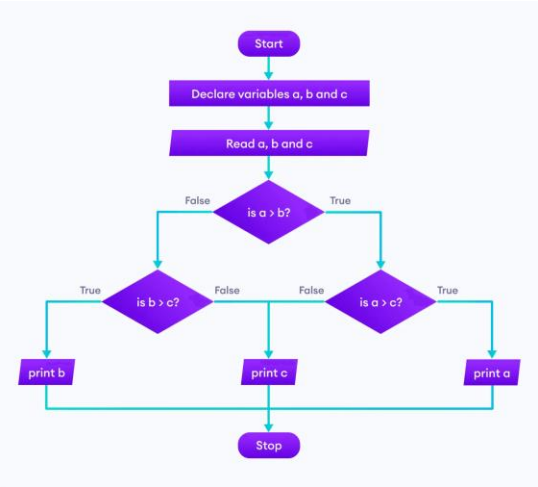
```
None, False, 0, 0.0, 0j
Empty sequences: '', [], (), {}
Empty mappings: {}
```

Truthy Values: All other values are considered `True`.

Best Practice: Use explicit comparisons (`if x is None:`) for clarity, but understand the implicit check for brevity.

```
my_list = []
if my_list:
print("Not empty")
else:
print("Empty (Falsy)")
```





Comparison vs. Logical: Defining and Combining Conditions

Comparison Operators: Used to compare two values and return a boolean (`True` or `False`).

> ==, !=, >, <, >=, <=

Logical Operators: Used to combine boolean results from comparisons.

- **`and`:** Returns `True` only if both operands are `True`.
- **`or`:** Returns `True` if at least one operand is `True`.
- **`not`:** Inverts the boolean value of an operand.

Operator Precedence: Comparison operators are evaluated before logical operators.

```
age = 25
is_student = True

if age > 18 and is_student:
print("Eligible for discount")
```

While Loops: Repeating Until a Condition Fails

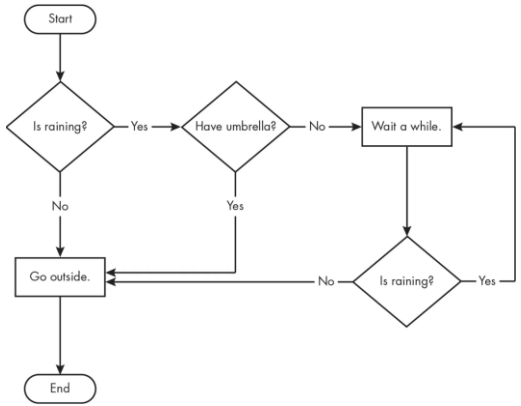
Condition-Based Repetition: The loop continues to execute as long as its condition remains `True`.

Primary Use Case: When the number of iterations is, **unknown** beforehand (e.g., waiting for user input, processing a stream).

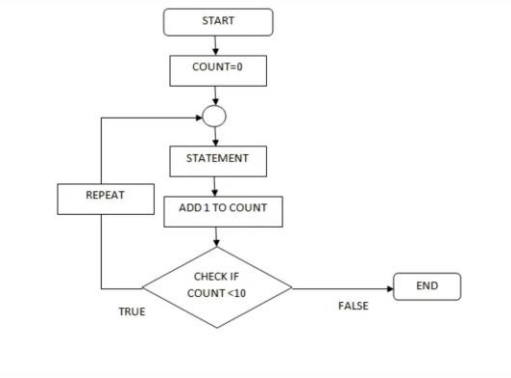
The Danger: If the condition never becomes `False`, you create an **infinite loop** , crashing the program.

When NOT to Use: When iterating over a fixed sequence (like a list or range of numbers)—use a `for` loop instead.

```
count = 0
while count < 3:
print(f"Count is {count}")
count += 1
```



For Loops Done Right: Iterating Over Sequences



Iteration, Not Condition: `for` loops iterate over the items of any sequence (list, tuple, string, range).

The `range()` Function: Generates a sequence of numbers, ideal for fixed-number iterations.

> range(stop): 0 up to (but not including) stop
> range(start, stop, step): Full control over the sequence

Iterating with Index: Use `enumerate()` to get both the index and the value simultaneously.

Iterating Dictionaries: Use `.items()` to iterate over key-value pairs efficiently.

```
names = ["Alice", "Bob", "Charlie"]
for index, name in enumerate(names):
print(f"{index}: {name}")
```

Control Statements: Fine-Tuning Loop Execution

break

Immediately terminates the current loop (both `for` and `while`) and moves execution to the statement following the loop.

Use Case: *Exiting a search early once a target is found.*

continue

Skips the rest of the code inside the current loop iteration and proceeds to the next iteration.

Use Case: *Skipping over invalid or unwanted data points.*

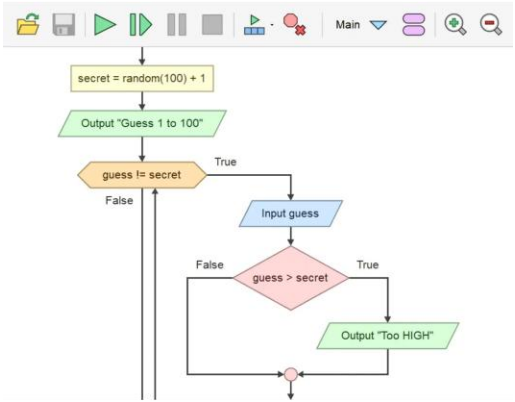
pass

A null operation. It does nothing but acts as a placeholder where a statement is syntactically required.

Use Case: *Defining empty function or class stubs for later implementation.*

Example: Using break, continue, and pass Together

```
for i in range(10):
 if i % 2 == 0:
 continue # Skip even numbers
 if i == 7:
 break &
```



Case Study: Building a Simple Input Validator

The Goal: Write a program that asks a user for a password and validates it against multiple rules.

Concepts Used:

- **`while` Loop:** To keep asking until a valid password is provided.
- **`if`/`elif`/`else`:** To check multiple validation rules.
- **Logical Operators:** To combine length and character checks.
- **`break`:** To exit the loop upon successful validation.

Program Logic:

Start an infinite `while True` loop.

Get password input from the user.

Check length (e.g., > 8 characters).

Check for required characters (e.g., uppercase, number).

If all checks pass, print success and `break`.

If any check fails, print the specific error and `continue` to the next iteration.

Implementation: Password Validation Script

```
whileTrue:
 password = input("Enter password (min 8 chars, 1 digit): ")

# 1. Check Length
iflen(password) < 8:
print("Error: Password must be at least 8 characters.")
continue

# 2. Check for Digit
 has_digit = False
for char in password:
if char.isdigit():
 has_digit = True
break

ifnot has_digit:
print("Error: Password must contain at least one digit.")
continue

# Success
print("Password accepted!")
break
```

Summary: Control Flow Fundamentals

Decisions: Use `if`, `elif`, and `else` to execute code conditionally. Remember that **indentation** defines the block.

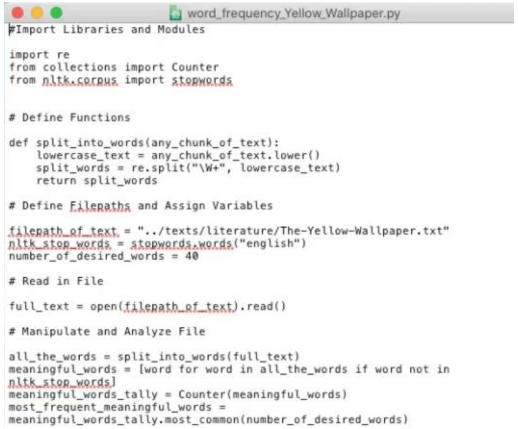
Iteration: Use `for` for known sequences and `while` for condition-based, **unknown** repetitions.

Boolean Logic: Be mindful of **Truthy** and **Falsy** values when writing conditions.

Control: Use `break` to exit a loop early and `continue` to skip an iteration.

Practice: The best way to master control flow is by writing small, logical programs.

Start building your own logical programs today!


word_frequency_Yellow_Wallpaper.py

```
#Import Libraries and Modules

import re
from collections import Counter
from nltk.corpus import stopwords


# Define Functions

def split_into_words(any_chunk_of_text):
    lowercase_text = any_chunk_of_text.lower()
    split_words = re.split("\W+", lowercase_text)
    return split_words

# Define Filepaths and Assign Variables

filepath_of_text = "../texts/literature/The-Yellow-Wallpaper.txt"
nltk_stop_words = stopwords.words("english")
number_of_desired_words = 40

# Read in File

full_text = open(filepath_of_text).read()

# Manipulate and Analyze File

all_the_words = split_into_words(full_text)
meaningful_words = [word for word in all_the_words if word not in
nltk_stop_words]
meaningful_words_tally = Counter(meaningful_words)
most_frequent_meaningful_words =
meaningful_words_tally.most_common(number_of_desired_words)
```