Programming Foundations

Python Basics, Software Principles, and Lift Control

Martin Benning

University College London

ENGF0034 – Design and Professional Skills

Lecture Overview

Objectives

To explore the foundations of computer programming, the nature of algorithms, core Python concepts, essential software engineering principles, and a complex real-world computational challenge.

Algorithms and Computation

What is an Algorithm?

Definition: Algorithm

An algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation.

What is an Algorithm?

Definition: Algorithm

An algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation.

Key Properties (Donald Knuth)

- **Finiteness:** Must terminate after a finite number of steps.
- **Definiteness:** Each step must be precisely and unambiguously specified.
- **Effectiveness:** Operations must be sufficiently basic that they can be carried out exactly and in finite time.

What is an Algorithm?

Key Properties (Donald Knuth)

- **Finiteness:** Must terminate after a finite number of steps.
- **Definiteness:** Each step must be precisely and unambiguously specified.
- **Effectiveness:** Operations must be sufficiently basic that they can be carried out exactly and in finite time.

Analysis of Algorithms

We analyse algorithms primarily for their **correctness** (does it solve the problem?) and their **efficiency** (how do resource requirements scale? - Complexity Analysis).

Algorithm of the Week: Euclid's GCD

Motivation and History

- Euclid of Alexandria (c. 300 BC) developed one of the oldest known non-trivial algorithms still in use.

Algorithm of the Week: Euclid's GCD

Motivation and History

- Euclid of Alexandria (c. 300 BC) developed one of the oldest known non-trivial algorithms still in use.
- The Greatest Common Divisor (GCD) is essential for simplifying fractions.

Algorithm of the Week: Euclid's GCD

Motivation and History

- Euclid of Alexandria (c. 300 BC) developed one of the oldest known non-trivial algorithms still in use.
- The Greatest Common Divisor (GCD) is essential for simplifying fractions.
- Example:

$$\frac{390253}{228769} = \frac{29 \cdot 13457}{17 \cdot 13457} = \frac{29}{17}$$

since $\mathrm{GCD}(390253, 228769) = 13457$.

Algorithm of the Week: Euclid's GCD

Motivation and History

- Euclid of Alexandria (c. 300 BC) developed one of the oldest known non-trivial algorithms still in use.
- The Greatest Common Divisor (GCD) is essential for simplifying fractions.
- Example:

$$\frac{390253}{228769} = \frac{29 \cdot 13457}{17 \cdot 13457} = \frac{29}{17}$$

since $\mathrm{GCD}(390253, 228769) = 13457$.

The Computational Challenge

How do we compute the GCD efficiently? The naive approach, prime factorisation, is computationally hard (intractable for large numbers, which is the basis of RSA cryptography).

Euclid's Algorithm (Subtraction Method)

Euclid's Insight (The Invariant)

The GCD of two numbers does not change if the larger number is replaced by its difference with the smaller number:

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a - b, b) \quad (\text{if } a > b)$$

Euclid's Algorithm (Subtraction Method)

Euclid's Insight (The Invariant)

The GCD of two numbers does not change if the larger number is replaced by its difference with the smaller number:

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a - b, b) \quad (\text{if } a > b)$$

The Algorithm (in prose)

1. Start with two positive integers, a and b.

Euclid's Algorithm (Subtraction Method)

Euclid's Insight (The Invariant)

The GCD of two numbers does not change if the larger number is replaced by its difference with the smaller number:

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a - b, b) \quad (\text{if } a > b)$$

The Algorithm (in prose)

1. Start with two positive integers, a and b.
2. If $a = b$, stop. The GCD is a.

Euclid's Algorithm (Subtraction Method)

Euclid's Insight (The Invariant)

The GCD of two numbers does not change if the larger number is replaced by its difference with the smaller number:

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a - b, b) \quad (\text{if } a > b)$$

The Algorithm (in prose)

1. Start with two positive integers, a and b.
2. If $a = b$, stop. The GCD is a.
3. If $a > b$, replace a with $a - b$.

Euclid's Algorithm (Subtraction Method)

Euclid's Insight (The Invariant)

The GCD of two numbers does not change if the larger number is replaced by its difference with the smaller number:

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a - b, b) \quad (\text{if } a > b)$$

The Algorithm (in prose)

1. Start with two positive integers, a and b.
2. If $a = b$, stop. The GCD is a.
3. If $a > b$, replace a with $a - b$.
4. If $b > a$, replace b with $b - a$.

Euclid's Algorithm (Subtraction Method)

Euclid's Insight (The Invariant)

The GCD of two numbers does not change if the larger number is replaced by its difference with the smaller number:

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a - b, b) \quad (\text{if } a > b)$$

The Algorithm (in prose)

1. Start with two positive integers, a and b.
2. If $a = b$, stop. The GCD is a.
3. If $a > b$, replace a with $a - b$.
4. If $b > a$, replace b with $b - a$.
5. Go back to step 2.

Euclid's Algorithm (Subtraction Method)

The Algorithm (in prose)

1. Start with two positive integers, a and b.
2. If $a = b$, stop. The GCD is a.
3. If $a > b$, replace a with $a - b$.
4. If $b > a$, replace b with $b - a$.
5. Go back to step 2.

Example: GCD(42, 30)

$(42, 30) \rightarrow (12, 30) \rightarrow (12, 18) \rightarrow (12, 6) \rightarrow (6, 6)$. Result: 6.

Why Euclid's Algorithm Works: The Proof (I)

We must prove both termination and correctness.

Termination

In each step, we replace a number with a smaller positive integer. The sum $a + b$ strictly decreases in every iteration. Since the sum is bounded below by 0, the algorithm must terminate.

Why Euclid's Algorithm Works: The Proof (I)

We must prove both termination and correctness.

Termination

In each step, we replace a number with a smaller positive integer. The sum $a + b$ strictly decreases in every iteration. Since the sum is bounded below by 0, the algorithm must terminate.

Correctness (Setup)

We must show the GCD remains invariant. Let $g = \mathrm{GCD}(a, b)$. By definition we have

$$a = \alpha g$$
$$b = \beta g$$

where $\mathrm{GCD}(\alpha, \beta) = 1$ (they are coprime). Assuming $a > b$, the next value a' is $a' = a - b = (\alpha - \beta)g$.

Why Euclid's Algorithm Works: The Proof (II)

We see that g is a common divisor of a' and b. But is it the *greatest*? We must show $\mathrm{GCD}(a', b) = g$.

Why Euclid's Algorithm Works: The Proof (II)

We see that g is a common divisor of a' and b. But is it the *greatest*? We must show $\mathrm{GCD}(a', b) = g$.

Proof by Contradiction

1. Assume $\mathrm{GCD}(a', b) = g' > g$.

Why Euclid's Algorithm Works: The Proof (II)

We see that g is a common divisor of a' and b. But is it the *greatest*? We must show $\mathrm{GCD}(a', b) = g$.

Proof by Contradiction

1. Assume $\mathrm{GCD}(a', b) = g' > g$.
2. This implies that the coefficients $(\alpha - \beta)$ and β must share a common factor $c > 1$:

$$\mathrm{GCD}(\alpha - \beta, \beta) = c > 1$$

Why Euclid's Algorithm Works: The Proof (II)

We see that g is a common divisor of a' and b. But is it the *greatest*? We must show $\mathrm{GCD}(a', b) = g$.

Proof by Contradiction

1. Assume $\mathrm{GCD}(a', b) = g' > g$.
2. This implies that the coefficients $(\alpha - \beta)$ and β must share a common factor $c > 1$:

$$\mathrm{GCD}(\alpha - \beta, \beta) = c > 1$$

3. If c divides β AND c divides $(\alpha - \beta)$, then c must also divide their sum:

$$c \mid ((\alpha - \beta) + \beta) \implies c \mid \alpha$$

Why Euclid's Algorithm Works: The Proof (II)

We see that g is a common divisor of a' and b. But is it the *greatest*? We must show $\mathrm{GCD}(a', b) = g$.

Proof by Contradiction

1. Assume $\mathrm{GCD}(a', b) = g' > g$.
2. This implies that the coefficients $(\alpha - \beta)$ and β must share a common factor $c > 1$:

$$\mathrm{GCD}(\alpha - \beta, \beta) = c > 1$$

3. If c divides β AND c divides $(\alpha - \beta)$, then c must also divide their sum:

$$c \mid ((\alpha - \beta) + \beta) \implies c \mid \alpha$$

4. Therefore, $c > 1$ is a common divisor of both α and β.

Why Euclid's Algorithm Works: The Proof (II)

Proof by Contradiction

1. Assume $\mathrm{GCD}(a', b) = g' > g$.
2. This implies that the coefficients $(\alpha - \beta)$ and β must share a common factor $c > 1$:

$$\mathrm{GCD}(\alpha - \beta, \beta) = c > 1$$

3. If c divides β AND c divides $(\alpha - \beta)$, then c must also divide their sum:

$$c \mid ((\alpha - \beta) + \beta) \implies c \mid \alpha$$

4. Therefore, $c > 1$ is a common divisor of both α and β.
5. **Contradiction!** This contradicts our premise that $\mathrm{GCD}(\alpha, \beta) = 1$.

Why Euclid's Algorithm Works: The Proof (III)

Conclusion

Therefore, $\mathrm{GCD}(a', b) = g$. The algorithm preserves the GCD (an invariant) while reducing the magnitude of the numbers until they are equal to the GCD.

Python Foundations

From Algorithm to Program: Basic Concepts

Let's translate the steps into Python, understanding the underlying concepts.

Components

- **Statements:** Complete units of execution that perform an action (e.g., `a = 42`).
- **Expressions:** Fragments of code that evaluate to a value (e.g., `42`, `a - b`).
- **Assignments:** Statements that bind a name (LHS) to the value of an expression (RHS).

```python
# Initial assignments
a = 42
b = 30

# Example step: RHS evaluated first, then assigned to LHS
a = a - b  # a is now 12
```

From Algorithm to Program: Basic Concepts

```python
# Initial assignments
a = 42
b = 30

# Example step: RHS evaluated first, then assigned to LHS
a = a - b  # a is now 12
```

Python Idioms: Simultaneous Assignment

`a, b = 42, 30`. All expressions on the RHS are evaluated before any assignment occurs. This allows elegant swapping: `a, b = b, a`.

Understanding Types (I)

In Python, every value is an object, and every object has a type. The type defines the data and the permitted operations.

Python's Type System: Dynamic Typing

Type checking is performed at **runtime**, just before an operation is executed.

- Variables are just names (labels) pointing to objects. The variable itself does not have a fixed type.

```python
a = 42      # a refers to an int
a = "Hello" # a now refers to a str. This is fine.
```

Understanding Types (II)

In Python, every value is an object, and every object has a type. The type defines the data and the permitted operations.

Python's Type System: Strong Typing

The type of an object matters. Operations are strictly checked for compatibility. Python refuses implicit conversions between incompatible types.

```
>>> "The answer is " + 42
TypeError: can only concatenate str (not "int") to str
```

You must explicitly convert: `"The answer is " + str(42)`.

The Role of Type Systems

Contrast: Static Typing (e.g., C++, Java)

Type checking is performed at **compile time**.

- Pros: Catches errors early; better performance (types known ahead of time).
- Cons: More verbose (requires type annotations); sometimes less flexible.

The Role of Type Systems

Contrast: Static Typing (e.g., C++, Java)

Type checking is performed at **compile time**.

- Pros: Catches errors early; better performance (types known ahead of time).
- Cons: More verbose (requires type annotations); sometimes less flexible.

Types as Formal Methods

Type systems are the most popular lightweight formal methods that aid program correctness. They ensure only permissible operations are executed, eliminating a large class of trivial bugs. But they do not guarantee logical correctness.

Numeric Types: Integers (`int`)

Arbitrary Precision

A key feature of Python integers is that they have **arbitrary precision**.

- Unlike fixed-width integers (e.g., 64-bit) in many languages, Python integers automatically expand, limited only by available memory.

Numeric Types: Integers (`int`)

Arbitrary Precision

A key feature of Python integers is that they have **arbitrary precision**.

- Unlike fixed-width integers (e.g., 64-bit) in many languages, Python integers automatically expand, limited only by available memory.

Operations

- Arithmetic: `+`, `-`, `*`, `**` (power).

Numeric Types: Integers (`int`)

Arbitrary Precision

A key feature of Python integers is that they have **arbitrary precision**.

- Unlike fixed-width integers (e.g., 64-bit) in many languages, Python integers automatically expand, limited only by available memory.

Operations

- Arithmetic: `+`, `-`, `*`, `**` (power).
- Integer Division (Floor): `//`. E.g., `5 // 2 = 2`.

Numeric Types: Integers (`int`)

Arbitrary Precision

A key feature of Python integers is that they have **arbitrary precision**.

- Unlike fixed-width integers (e.g., 64-bit) in many languages, Python integers automatically expand, limited only by available memory.

Operations

- Arithmetic: `+`, `-`, `*`, `**` (power).
- Integer Division (Floor): `//`. E.g., `5 // 2 = 2`.
- Modulo (Remainder): `%`. E.g., `5 % 2 = 1`.

Numeric Types: Integers (`int`)

Arbitrary Precision

A key feature of Python integers is that they have **arbitrary precision**.

- Unlike fixed-width integers (e.g., 64-bit) in many languages, Python integers automatically expand, limited only by available memory.

Operations

- Arithmetic: `+`, `-`, `*`, `**` (power).
- Integer Division (Floor): `//`. E.g., `5 // 2 = 2`.
- Modulo (Remainder): `%`. E.g., `5 % 2 = 1`.
- True Division: `/`. E.g., `5 / 2 = 2.5`.

Numeric Types: Integers (`int`)

Arbitrary Precision

A key feature of Python integers is that they have **arbitrary precision**.

- Unlike fixed-width integers (e.g., 64-bit) in many languages, Python integers automatically expand, limited only by available memory.

Operations

- Arithmetic: `+`, `-`, `*`, `**` (power).
- Integer Division (Floor): `//`. E.g., `5 // 2 = 2`.
- Modulo (Remainder): `%`. E.g., `5 % 2 = 1`.
- True Division: `/`. E.g., `5 / 2 = 2.5`.

Note that `/` always returns a `float`, even if the operands are integers (e.g., `4 / 2 = 2.0`).

Numeric Types: Integers (`int`)

Arbitrary Precision

A key feature of Python integers is that they have **arbitrary precision**.

- Unlike fixed-width integers (e.g., 64-bit) in many languages, Python integers automatically expand, limited only by available memory.

Computational Note: GCD Optimisation

The subtraction-based GCD can be slow if $a \gg b$. The standard implementation uses the modulo operator: $\mathrm{GCD}(a, b) = \mathrm{GCD}(b, a \bmod b)$. This is much faster (logarithmic complexity).

Control Flow: Automating Execution

To implement algorithms, we need mechanisms to control the order of execution beyond sequential steps.

Control Flow: Automating Execution

To implement algorithms, we need mechanisms to control the order of execution beyond sequential steps.

1. **Conditional (Selection):** Execute a block of code only if a condition is met (`if`/`else`).

Control Flow: Automating Execution

To implement algorithms, we need mechanisms to control the order of execution beyond sequential steps.

1. **Conditional (Selection):** Execute a block of code only if a condition is met (`if`/`else`).
2. **Iterative (Repetition):** Execute a block of code repeatedly (`while`, `for`).

Control Flow: Automating Execution

To implement algorithms, we need mechanisms to control the order of execution beyond sequential steps.

1. **Conditional (Selection):** Execute a block of code only if a condition is met (`if`/`else`).
2. **Iterative (Repetition):** Execute a block of code repeatedly (`while`, `for`).

Indentation Defines Blocks

In Python, blocks of code are defined by their indentation level. This is crucial. Use 4 spaces consistently. Do not mix tabs and spaces.

Conditional Execution: The `if` Statement

Syntax

```python
if condition:
    # Block 1 (executed if condition is True)
elif another_condition:
    # Block 2
else:
    # Block 3 (executed otherwise)
```

Conditional Execution: The `if` Statement

Syntax

```
if condition:
    # Block 1 (executed if condition is True)
elif another_condition:
    # Block 2
else:
    # Block 3 (executed otherwise)
```

Conditions and `bool`

Conditions are expressions that evaluate to the `bool` type (`True` or `False`).

- Comparison Operators: ==, `!=`, <, >, etc.

Conditional Execution: The `if` Statement

Syntax

```
if condition:
    # Block 1 (executed if condition is True)
elif another_condition:
    # Block 2
else:
    # Block 3 (executed otherwise)
```

Conditions and `bool`

Conditions are expressions that evaluate to the `bool` type (`True` or `False`).

- Comparison Operators: `==`, `!=`, `<`, `>`, etc.
- Logical Operators: `and`, `or`, `not`.

Iteration: The `while` Loop

Syntax

Repeatedly executes a block of statements as long as a condition remains `True`.

```
while condition:
    # Loop body
    statement_1
```

Execution Flow

1. Evaluate the `condition`.

Iteration: The `while` Loop

Syntax

Repeatedly executes a block of statements as long as a condition remains `True`.

```
while condition:
    # Loop body
    statement_1
```

Execution Flow

1. Evaluate the `condition`.
2. If `False`, skip the loop body.

Iteration: The `while` Loop

Syntax

Repeatedly executes a block of statements as long as a condition remains `True`.

```
while condition:
    # Loop body
    statement_1
```

Execution Flow

1. Evaluate the `condition`.
2. If `False`, skip the loop body.
3. If `True`, execute the loop body.

Iteration: The `while` Loop

Syntax

Repeatedly executes a block of statements as long as a condition remains `True`.

```
while condition:
    # Loop body
    statement_1
```

Execution Flow

1. Evaluate the `condition`.
2. If `False`, skip the loop body.
3. If `True`, execute the loop body.
4. Go back to step 1.

Iteration: The `while` Loop

Syntax

Repeatedly executes a block of statements as long as a condition remains `True`.

```
while condition:
    # Loop body
    statement_1
```

Danger: Infinite Loops

You must ensure that the state changes within the loop such that the condition eventually becomes `False`. (Euclid's algorithm is proven to terminate).

Automating Euclid's Algorithm

Combine `while` and `if` to fully automate algorithm

```python
# GCD computation for 42 and 30
a, b = 42, 30
# Repeat until the numbers are equal
while a != b:
    # Check which number is larger
    if a > b:
        # Subtract smaller from larger (update a)
        a = a - b
    else:
        # Subtract smaller from larger (update b)
        b = b - a
# Now a == b, so the GCD is in a (or b).
print(a)
```

Strings in Depth

Strings (`str`)

Definition

A string in Python 3 is an immutable sequence of Unicode code points.

Unicode Representation

Python 3 uses Unicode (UTF-8 encoding by default for source files) to represent text, supporting characters from all languages and symbols.

```
>>> s = "Hello"
>>> len(s)
5  # Length is the number of Unicode characters
```

Strings (`str`)

Definition

A string in Python 3 is an immutable sequence of Unicode code points.

Unicode Representation

Python 3 uses Unicode (UTF-8 encoding by default for source files) to represent text, supporting characters from all languages and symbols.

```
>>> s = "Hello"
>>> len(s)
5  # Length is the number of Unicode characters
```

Strings (`str`)

Unicode Representation

Python 3 uses Unicode (UTF-8 encoding by default for source files) to represent text, supporting characters from all languages and symbols.

```
>>> s = "Hello"
>>> len(s)
5  # Length is the number of Unicode characters
```

String Literals

Created using single ('), double ("), or triple quotes ("""). Triple quotes allow multi-line strings.

Indexing and Slicing (I)

As sequences, strings support indexing and slicing.

Indexing (Zero-Based)

Access individual characters. Negative indices count from the end.

```
s = "Python"
# P y t h o n
# 0 1 2 3 4 5
#-6-5-4-3-2-1

>>> s[0]
'P'
>>> s[-1]
'n'
```

Indexing and Slicing (II)

Slicing

Extracting a substring: `s[start:stop:step]`.

- `start` is inclusive; `stop` is exclusive.
- Defaults: `start=0`, `stop=len(s)`, `step=1`.

```
>>> s[0:2]
'Py'
>>> s[2:]   # From index 2 to the end
'thon'
>>> s[::-1] # Reverse the string (step of -1)
'nohtyP'
```

String Immutability

Key Concept: Immutability

Strings in Python are immutable. Once created, they cannot be changed in place.

```
s = "Python"
>>> s[0] = "J"
TypeError: 'str' object does not support item assignment
```

String Immutability

Key Concept: Immutability

Strings in Python are immutable. Once created, they cannot be changed in place.

```
s = "Python"
>>> s[0] = "J"
TypeError: 'str' object does not support item assignment
```

Modifying Strings

To ”modify” a string, you must create a new string based on the old one.

```
s = "J" + s[1:] # Creates a new string 'Jython'
```

String Immutability

Key Concept: Immutability

Strings in Python are immutable. Once created, they cannot be changed in place.

```
s = "Python"
>>> s[0] = "J"
TypeError: 'str' object does not support item assignment
```

Computational Implication

Immutability makes strings safe (e.g., as dictionary keys). However, repeatedly concatenating long strings in a loop (`s += "..."`) is inefficient ($O(N^2)$) as it creates many intermediate objects. Use `.join()` for efficient concatenation ($O(N)$).

String Formatting (f-Strings)

Formatted String Literals (f-Strings)

Introduced in Python 3.6, f-strings provide a concise and readable way to embed expressions inside string literals. Prefix the string with `f`.

```
name = "Alice"
age = 30
# Expressions inside {} are evaluated at runtime
message = f"My name is {name} and I am {age} years old."
>>> print(message)
My name is Alice and I am 30 years old.
```

String Formatting (f-Strings)

Formatted String Literals (f-Strings)

Introduced in Python 3.6, f-strings provide a concise and readable way to embed expressions inside string literals. Prefix the string with `f`.

Advanced Formatting

f-strings support powerful formatting options (e.g., precision, padding).

```
import math
>>> print(f"Pi is approximately {math.pi:.3f}")
Pi is approximately 3.142
```

Structuring Code and Software Principles

Abstraction and Functions

The previous implementation of GCD works, but it is monolithic. To improve reusability and readability, we use functions.

Abstraction and Functions

The previous implementation of GCD works, but it is monolithic. To improve reusability and readability, we use functions.

Functions

Functions are named blocks of code that can be reused with different inputs. They are the fundamental unit of procedural abstraction.

Abstraction and Functions

The previous implementation of GCD works, but it is monolithic. To improve reusability and readability, we use functions.

Functions

Functions are named blocks of code that can be reused with different inputs. They are the fundamental unit of procedural abstraction.

Guiding Principles

- **Abstraction Principle (B. C. Pierce):** ”Each significant piece of functionality in a program should be implemented in just one place in the source code.”

Abstraction and Functions

The previous implementation of GCD works, but it is monolithic. To improve reusability and readability, we use functions.

Functions

Functions are named blocks of code that can be reused with different inputs. They are the fundamental unit of procedural abstraction.

Guiding Principles

- **Abstraction Principle (B. C. Pierce):** "Each significant piece of functionality in a program should be implemented in just one place in the source code."
- **DRY (Don't Repeat Yourself) Principle (Hunt & Thomas):** "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

Euclid as a Function

Refactoring

Encapsulating the logic within a function.

```python
def GCD(a, b):
    """Compute the GCD of two positive integers."""
    # The parameters a and b are local to this function
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    # Return the result
    return a
```

Euclid as a Function

Refactoring

Encapsulating the logic within a function.

Calling the Function

```python
result1 = GCD(42, 30)
print(result1) # Output: 6

result2 = GCD(100, 15)
print(result2) # Output: 5
```

Variable Scope (LEGB Rule)

What is Scope?

Scope defines the region of a program where a variable name is accessible. It ensures locality and prevents name clashes. In the GCD example, modifying `a` inside the function did not affect the variable `a` outside.

Variable Scope (LEGB Rule)

What is Scope?

Scope defines the region of a program where a variable name is accessible. It ensures locality and prevents name clashes. In the GCD example, modifying `a` inside the function did not affect the variable `a` outside.

The LEGB Rule

When Python encounters a name, it searches for its definition in this specific order:

Variable Scope (LEGB Rule)

What is Scope?

Scope defines the region of a program where a variable name is accessible. It ensures locality and prevents name clashes. In the GCD example, modifying `a` inside the function did not affect the variable `a` outside.

The LEGB Rule

When Python encounters a name, it searches for its definition in this specific order:

L (Local): Within the current function.

Variable Scope (LEGB Rule)

What is Scope?

Scope defines the region of a program where a variable name is accessible. It ensures locality and prevents name clashes. In the GCD example, modifying `a` inside the function did not affect the variable `a` outside.

The LEGB Rule

When Python encounters a name, it searches for its definition in this specific order:

L (Local): Within the current function.

E (Enclosing): Within any enclosing functions (from inner to outer).

Variable Scope (LEGB Rule)

What is Scope?

Scope defines the region of a program where a variable name is accessible. It ensures locality and prevents name clashes. In the GCD example, modifying `a` inside the function did not affect the variable `a` outside.

The LEGB Rule

When Python encounters a name, it searches for its definition in this specific order:

L (Local): Within the current function.

E (Enclosing): Within any enclosing functions (from inner to outer).

G (Global): At the top level of the module (file).

Variable Scope (LEGB Rule)

What is Scope?

Scope defines the region of a program where a variable name is accessible. It ensures locality and prevents name clashes. In the GCD example, modifying `a` inside the function did not affect the variable `a` outside.

The LEGB Rule

When Python encounters a name, it searches for its definition in this specific order:

- L (Local): Within the current function.
- E (Enclosing): Within any enclosing functions (from inner to outer).
- G (Global): At the top level of the module (file).
- B (Built-in): Pre-assigned in Python (e.g., `print`, `int`).

Variable Scope (LEGB Rule)

The LEGB Rule

When Python encounters a name, it searches for its definition in this specific order:

- L (Local): Within the current function.
- E (Enclosing): Within any enclosing functions (from inner to outer).
- G (Global): At the top level of the module (file).
- B (Built-in): Pre-assigned in Python (e.g., `print`, `int`).

Best Practice

Minimise the use of global variables. Functions should ideally communicate only through parameters and return values.

Ensuring Correctness: Testing (I)

The Role of Testing

Testing verifies that the code behaves as expected. While tests cannot prove the absence of bugs (this is generally undecidable), they increase confidence in correctness.

Ensuring Correctness: Testing (I)

The Role of Testing

Testing verifies that the code behaves as expected. While tests cannot prove the absence of bugs (this is generally undecidable), they increase confidence in correctness.

The `assert` Statement

A simple way to write tests. If the condition is `False`, it raises an `AssertionError`.

```
assert condition, "Optional error message"
```

Ensuring Correctness: Testing (II)

Testing Euclid

```python
def test_euclid():
    """Unit tests for the GCD function."""
    assert GCD(42, 30) == 6
    assert GCD(30, 42) == 6 # Test commutativity
    assert GCD(17, 5) == 1  # Boundary condition (coprime)
    assert GCD(1, 1) == 1   # Boundary condition

test_euclid()
```

What Makes a Good Test?

Advice from Kernighan and Pike (*The Practice of Programming*)

- **Coverage:** Ensure all lines and code paths (e.g., both branches of an `if`) are exercised.
- **Boundary Conditions:** Test edge cases (e.g., smallest/largest values, empty inputs).
- **Pre- and Post-conditions:** Verify assumptions about inputs and guarantees about outputs.
- **Error Paths:** Test how the code handles invalid inputs or exceptional circumstances.

What Makes a Good Test?

Advice from Kernighan and Pike (*The Practice of Programming*)

- **Coverage:** Ensure all lines and code paths (e.g., both branches of an `if`) are exercised.
- **Boundary Conditions:** Test edge cases (e.g., smallest/largest values, empty inputs).
- **Pre- and Post-conditions:** Verify assumptions about inputs and guarantees about outputs.
- **Error Paths:** Test how the code handles invalid inputs or exceptional circumstances.

Refactoring and Version Control

Comprehensive tests enable fearless **refactoring**—restructuring code without changing its external behaviour. **Version Control** (e.g., Git) allows tracking changes and reverting if refactoring introduces bugs.

Handling Errors: The Happy Path vs. Exceptions

Real-world code must be robust to exceptional circumstances (invalid inputs, system failures).

Principles of Error Handling

Handling Errors: The Happy Path vs. Exceptions

Real-world code must be robust to exceptional circumstances (invalid inputs, system failures).

Principles of Error Handling

- **Fail Fast, Fail Loudly:** Upon an error, the code should stop immediately and signal the error clearly. Never fail silently.

Handling Errors: The Happy Path vs. Exceptions

Real-world code must be robust to exceptional circumstances (invalid inputs, system failures).

Principles of Error Handling

- **Fail Fast, Fail Loudly:** Upon an error, the code should stop immediately and signal the error clearly. Never fail silently.
- **Clarity:** Error handling logic should not obscure the primary logic (the ”happy path”).

Handling Errors: The Happy Path vs. Exceptions

Real-world code must be robust to exceptional circumstances (invalid inputs, system failures).

Principles of Error Handling

- **Fail Fast, Fail Loudly:** Upon an error, the code should stop immediately and signal the error clearly. Never fail silently.
- **Clarity:** Error handling logic should not obscure the primary logic (the ”happy path”).

Exceptions in Python

Python uses exceptions (`raise`, `try`, `except`) rather than error return codes to handle runtime errors.

Raising Exceptions (Enforcing Preconditions)

Euclid's algorithm requires positive integers. What happens if we call `GCD(42, -5)`? (Infinite loop in our basic implementation!) We must enforce the precondition.

Raising Exceptions (Enforcing Preconditions)

Euclid's algorithm requires positive integers. What happens if we call `GCD(42, -5)`? (Infinite loop in our basic implementation!) We must enforce the precondition.

The `raise` Keyword

Interrupts the normal flow and signals an exception immediately.

```python
def GCD(a, b):
    """Compute the GCD of two positive integers."""
    if not (a > 0 and b > 0):
        # Raise an appropriate exception type (ValueError for
            invalid values)
        raise ValueError(f"Inputs {a}, {b} must be positive.")

    # To be continued...
```

Raising Exceptions (Enforcing Preconditions)

The `raise` Keyword

```python
def GCD(a, b):
    """Compute the GCD of two positive integers."""
    if not (a > 0 and b > 0):
        # Raise an appropriate exception type (ValueError for
            invalid values)
        raise ValueError(f"Inputs {a}, {b} must be positive.")
    # Algorithm logic (repeated here for completeness)
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Handling Exceptions: `try/except/finally`

We use `try/except` blocks to handle exceptions gracefully.

Syntax (Conceptual)

```
try:
    # Block of code that might raise an exception
    risky_operation()
except ExceptionType as e:
    # Block executed if ExceptionType occurs
    handle_the_error(e)
finally:
    # Block always executed (for cleanup, e.g., closing files)
    cleanup()
```

Handling Exceptions: `try/except/finally`

Strategy: Detect Low, Handle High

- Detect errors at a low level where the violation occurs (e.g., inside `GCD`).
- Handle (recover from) errors at a high level where the context is known (e.g., asking the user for new input).
- Only handle errors if you can meaningfully recover; otherwise, let the exception propagate up the call stack.

Computational Challenge: Lift Control Systems

The Lift (Elevator) Problem

We now apply algorithmic thinking to a complex, real-world engineering challenge: designing a lift control system.

The Challenge

To efficiently transport passengers within a building, minimising waiting times and travel times, while respecting physical constraints (capacity, speed, acceleration).

- This is a highly complex **stochastic optimisation problem**.
- Inputs (passenger arrivals) are uncertain and dynamic.
- Involves multiple agents (lifts) acting concurrently.
- It is NP-hard in its general form.

Performance Metrics (QoS)

How do we quantify the efficiency and quality of service (QoS)?

Key Metrics

Performance Metrics (QoS)

How do we quantify the efficiency and quality of service (QoS)?

Key Metrics

Average Waiting Time (AWT) Time from button press until the lift arrives. Critical for user satisfaction.

Performance Metrics (QoS)

How do we quantify the efficiency and quality of service (QoS)?

Key Metrics

Average Waiting Time (AWT) Time from button press until the lift arrives. Critical for user satisfaction.

Average Journey Time (AJT) Time spent inside the lift (includes travel time and intermediate stops).

Performance Metrics (QoS)

How do we quantify the efficiency and quality of service (QoS)?

Key Metrics

Average Waiting Time (AWT) Time from button press until the lift arrives. Critical for user satisfaction.

Average Journey Time (AJT) Time spent inside the lift (includes travel time and intermediate stops).

Throughput Passengers transported per unit time. Crucial during peak hours (e.g., morning rush).

Performance Metrics (QoS)

How do we quantify the efficiency and quality of service (QoS)?

Key Metrics

Average Waiting Time (AWT) Time from button press until the lift arrives. Critical for user satisfaction.

Average Journey Time (AJT) Time spent inside the lift (includes travel time and intermediate stops).

Throughput Passengers transported per unit time. Crucial during peak hours (e.g., morning rush).

Energy Consumption Minimising unnecessary movement.

Performance Metrics (QoS)

Key Metrics

Average Waiting Time (AWT) Time from button press until the lift arrives. Critical for user satisfaction.

Average Journey Time (AJT) Time spent inside the lift (includes travel time and intermediate stops).

Throughput Passengers transported per unit time. Crucial during peak hours (e.g., morning rush).

Energy Consumption Minimising unnecessary movement.

Trade-offs

Objectives often conflict. Minimising AWT might increase the AJT for passengers already on board. The control system must balance these trade-offs.

Basic Algorithms: FIFO and SSTF

First-In, First-Out (FIFO) / FCFS

Serve requests in the order they arrive.

Basic Algorithms: FIFO and SSTF

First-In, First-Out (FIFO) / FCFS

Serve requests in the order they arrive.

- **Pros:** Inherently fair.

Basic Algorithms: FIFO and SSTF

First-In, First-Out (FIFO) / FCFS

Serve requests in the order they arrive.

- **Pros:** Inherently fair.
- **Cons:** Highly inefficient. Excessive travel time (zig-zagging). Ignores opportunities to serve requests along the way.

Basic Algorithms: FIFO and SSTF

First-In, First-Out (FIFO) / FCFS

Serve requests in the order they arrive.

- **Pros:** Inherently fair.
- **Cons:** Highly inefficient. Excessive travel time (zig-zagging). Ignores opportunities to serve requests along the way.

Shortest Seek Time First (SSTF)

Always move to the nearest outstanding request. (Analogy: Disk I/O scheduling).

Basic Algorithms: FIFO and SSTF

First-In, First-Out (FIFO) / FCFS

Serve requests in the order they arrive.

- **Pros:** Inherently fair.
- **Cons:** Highly inefficient. Excessive travel time (zig-zagging). Ignores opportunities to serve requests along the way.

Shortest Seek Time First (SSTF)

Always move to the nearest outstanding request. (Analogy: Disk I/O scheduling).

- **Pros:** Reduces immediate travel time.

Basic Algorithms: FIFO and SSTF

First-In, First-Out (FIFO) / FCFS

Serve requests in the order they arrive.

- **Pros:** Inherently fair.
- **Cons:** Highly inefficient. Excessive travel time (zig-zagging). Ignores opportunities to serve requests along the way.

Shortest Seek Time First (SSTF)

Always move to the nearest outstanding request. (Analogy: Disk I/O scheduling).

- **Pros:** Reduces immediate travel time.
- **Cons:** Leads to **starvation**. Requests far from the current activity may wait indefinitely. Does not account for direction of travel.

The Standard ”Elevator Algorithm”: SCAN

The basis for most traditional lift systems, balancing efficiency and fairness.

SCAN Algorithm

The lift travels continuously in one direction (Up or Down). It only reverses direction at the physical ends of the shaft.

The Standard ”Elevator Algorithm”: SCAN

The basis for most traditional lift systems, balancing efficiency and fairness.

SCAN Algorithm

The lift travels continuously in one direction (Up or Down). It only reverses direction at the physical ends of the shaft.

- **While moving Up:** Stops at floors with Up hall calls or car calls in that direction.

The Standard ”Elevator Algorithm”: SCAN

The basis for most traditional lift systems, balancing efficiency and fairness.

SCAN Algorithm

The lift travels continuously in one direction (Up or Down). It only reverses direction at the physical ends of the shaft.

- **While moving Up:** Stops at floors with Up hall calls or car calls in that direction.
- **While moving Down:** Stops at floors with Down hall calls or car calls in that direction.

The Standard "Elevator Algorithm": SCAN

The basis for most traditional lift systems, balancing efficiency and fairness.

SCAN Algorithm

The lift travels continuously in one direction (Up or Down). It only reverses direction at the physical ends of the shaft.

- **While moving Up:** Stops at floors with Up hall calls or car calls in that direction.
- **While moving Down:** Stops at floors with Down hall calls or car calls in that direction.

Advantages and Disadvantages

- Pros: Bounded waiting times (fair). Efficient movement by batching requests.

The Standard ”Elevator Algorithm”: SCAN

The basis for most traditional lift systems, balancing efficiency and fairness.

SCAN Algorithm

The lift travels continuously in one direction (Up or Down). It only reverses direction at the physical ends of the shaft.

- **While moving Up:** Stops at floors with Up hall calls or car calls in that direction.
- **While moving Down:** Stops at floors with Down hall calls or car calls in that direction.

Advantages and Disadvantages

- Pros: Bounded waiting times (fair). Efficient movement by batching requests.
- Cons: Unnecessary travel to the absolute top/bottom floors.

Optimisations: LOOK and C-LOOK

LOOK Algorithm

The standard improvement over SCAN.

- The car only travels as far as the **furthest outstanding request** in the current direction.

Optimisations: LOOK and C-LOOK

LOOK Algorithm

The standard improvement over SCAN.

- The car only travels as far as the **furthest outstanding request** in the current direction.
- If there are no more requests ahead, it reverses direction.

Optimisations: LOOK and C-LOOK

LOOK Algorithm

The standard improvement over SCAN.

- The car only travels as far as the **furthest outstanding request** in the current direction.
- If there are no more requests ahead, it reverses direction.
- This is what is typically implemented in practice.

Optimisations: LOOK and C-LOOK

LOOK Algorithm

The standard improvement over SCAN.

- The car only travels as far as the **furthest outstanding request** in the current direction.
- If there are no more requests ahead, it reverses direction.
- This is what is typically implemented in practice.

C-LOOK (Circular LOOK)

- Serves requests only in *one* direction (e.g., always Up).

Optimisations: LOOK and C-LOOK

LOOK Algorithm

The standard improvement over SCAN.

- The car only travels as far as the **furthest outstanding request** in the current direction.
- If there are no more requests ahead, it reverses direction.
- This is what is typically implemented in practice.

C-LOOK (Circular LOOK)

- Serves requests only in *one* direction (e.g., always Up).
- When it reaches the furthest request, it quickly returns (express) to the lowest request.

Optimisations: LOOK and C-LOOK

LOOK Algorithm

The standard improvement over SCAN.

- The car only travels as far as the **furthest outstanding request** in the current direction.
- If there are no more requests ahead, it reverses direction.
- This is what is typically implemented in practice.

C-LOOK (Circular LOOK)

- Serves requests only in *one* direction (e.g., always Up).
- When it reaches the furthest request, it quickly returns (express) to the lowest request.
- Provides more uniform waiting times across all floors compared to LOOK, as the middle floors aren't visited twice in a cycle.

Handling Multiple Lifts: The Challenge

When multiple lifts operate in a group, coordination is essential.

The Problem of Bunching

Lifts tend to cluster together over time, moving as a platoon. This is a major source of inefficiency.

Handling Multiple Lifts: The Challenge

When multiple lifts operate in a group, coordination is essential.

The Problem of Bunching

Lifts tend to cluster together over time, moving as a platoon. This is a major source of inefficiency.

Handling Multiple Lifts: The Challenge

When multiple lifts operate in a group, coordination is essential.

The Problem of Bunching

Lifts tend to cluster together over time, moving as a platoon. This is a major source of inefficiency.

Why Bunching Occurs (System Instability)

1. A lift (Lift A) is delayed (e.g., high passenger load).

Handling Multiple Lifts: The Challenge

When multiple lifts operate in a group, coordination is essential.

The Problem of Bunching

Lifts tend to cluster together over time, moving as a platoon. This is a major source of inefficiency.

Why Bunching Occurs (System Instability)

1. A lift (Lift A) is delayed (e.g., high passenger load).
2. Gap to the preceding lift increases, causing more passengers to arrive ahead of Lift A.

Handling Multiple Lifts: The Challenge

When multiple lifts operate in a group, coordination is essential.

The Problem of Bunching

Lifts tend to cluster together over time, moving as a platoon. This is a major source of inefficiency.

Why Bunching Occurs (System Instability)

1. A lift (Lift A) is delayed (e.g., high passenger load).
2. Gap to the preceding lift increases, causing more passengers to arrive ahead of Lift A.
3. Lift A has to make more stops, further delaying it.

Handling Multiple Lifts: The Challenge

When multiple lifts operate in a group, coordination is essential.

The Problem of Bunching

Lifts tend to cluster together over time, moving as a platoon. This is a major source of inefficiency.

Why Bunching Occurs (System Instability)

1. A lift (Lift A) is delayed (e.g., high passenger load).
2. Gap to the preceding lift increases, causing more passengers to arrive ahead of Lift A.
3. Lift A has to make more stops, further delaying it.
4. The following lift (Lift B) faces fewer passengers (picked up by A), causing it to speed up and catch up to Lift A.

Handling Multiple Lifts: The Challenge

When multiple lifts operate in a group, coordination is essential.

Why Bunching Occurs (System Instability)

1. A lift (Lift A) is delayed (e.g., high passenger load).
2. Gap to the preceding lift increases, causing more passengers to arrive ahead of Lift A.
3. Lift A has to make more stops, further delaying it.
4. The following lift (Lift B) faces fewer passengers (picked up by A), causing it to speed up and catch up to Lift A.

Consequence

Long gaps in service (high AWT) followed by the arrival of multiple lifts simultaneously.

Multi-Lift Strategies: HCA

Zoning (Static Allocation)

Divide the building into vertical zones. Each lift serves only its zone. Simple but inflexible to changing traffic patterns.

Multi-Lift Strategies: HCA

Zoning (Static Allocation)

Divide the building into vertical zones. Each lift serves only its zone. Simple but inflexible to changing traffic patterns.

Dynamic Allocation / Hall Call Allocation (HCA)

Lifts are dynamically assigned to hall calls in real-time by a centralised controller.

Multi-Lift Strategies: HCA

Zoning (Static Allocation)

Divide the building into vertical zones. Each lift serves only its zone. Simple but inflexible to changing traffic patterns.

Dynamic Allocation / Hall Call Allocation (HCA)

Lifts are dynamically assigned to hall calls in real-time by a centralised controller.

The Core Problem (HCA)

When a new hall call is registered (e.g., Floor 5 Up), which lift should serve it?

Multi-Lift Strategies: HCA

Dynamic Allocation / Hall Call Allocation (HCA)

Lifts are dynamically assigned to hall calls in real-time by a centralised controller.

The Core Problem (HCA)

When a new hall call is registered (e.g., Floor 5 Up), which lift should serve it?

Optimisation Goal

Minimise a cost function (e.g., Estimated Time of Arrival (ETA), impact on existing passengers, energy use, etc.).

The Computational Complexity of HCA

Finding the optimal allocation is computationally hard.

The Optimisation Problem

Minimise the global cost function (e.g., total AWT + AJT) over a time horizon, considering all current and (predicted) future requests.

The Computational Complexity of HCA

Finding the optimal allocation is computationally hard.

The Optimisation Problem

Minimise the global cost function (e.g., total AWT + AJT) over a time horizon, considering all current and (predicted) future requests.

Complexity Analysis

- Analogous to the dynamic Vehicle Routing Problem (VRP) with capacity constraints.
- Known to be **NP-hard**.

The Computational Complexity of HCA

Finding the optimal allocation is computationally hard.

Complexity Analysis

- Analogous to the dynamic Vehicle Routing Problem (VRP) with capacity constraints.
- Known to be **NP-hard**.

Implications

- Finding the exact optimum is intractable in real-time.
- Practical systems must rely on **heuristics** (e.g., Nearest Car, Minimum Cost Insertion) and approximations.

The Information Deficit and DDS

The Information Deficit

Traditional systems (HCA) only know the passenger's starting floor and direction (Up/Down). They do not know the destination until the passenger is inside the car. This severely limits optimisation potential.

The Information Deficit and DDS

The Information Deficit

Traditional systems (HCA) only know the passenger's starting floor and direction (Up/Down). They do not know the destination until the passenger is inside the car. This severely limits optimisation potential.

A Paradigm Shift: Destination Dispatch Systems (DDS)

Modern systems change the input mechanism to gain crucial information earlier. Passengers enter their exact **destination floor** on a keypad in the lobby *before* entering the lift.

Traditional (HCA) $\xrightarrow{\text{More Information}}$ Destination Dispatch (DDS)

Input: (Origin, Direction) — Input: (Origin, Destination)

Advantages of DDS

Knowing the destination upfront significantly improves optimisation potential.

Advantages of DDS

Knowing the destination upfront significantly improves optimisation potential.

Key Benefits

- **Grouping:** The system can group passengers traveling to the same or nearby floors into the same lift.

Advantages of DDS

Knowing the destination upfront significantly improves optimisation potential.

Key Benefits

- **Grouping:** The system can group passengers traveling to the same or nearby floors into the same lift.
- **Reduced Stops:** Lifts make fewer intermediate stops, significantly reducing AJT.

Advantages of DDS

Knowing the destination upfront significantly improves optimisation potential.

Key Benefits

- **Grouping:** The system can group passengers traveling to the same or nearby floors into the same lift.
- **Reduced Stops:** Lifts make fewer intermediate stops, significantly reducing AJT.
- **Improved Throughput:** Overall handling capacity increases (often cited up to 30% improvement during peak hours).

Advantages of DDS

Key Benefits

- **Grouping:** The system can group passengers traveling to the same or nearby floors into the same lift.
- **Reduced Stops:** Lifts make fewer intermediate stops, significantly reducing AJT.
- **Improved Throughput:** Overall handling capacity increases (often cited up to 30% improvement during peak hours).

Computational Perspective

DDS reduces uncertainty. In HCA, the controller must guess the destination, leading to suboptimal commitments. DDS allows the optimiser to solve the (still NP-hard) problem more effectively using advanced techniques like Genetic Algorithms (GAs) or heuristic search (A*).

Modern Approaches: Machine Learning and RL

Applying modern computational techniques to the lift control problem.

Traffic Prediction

Using historical data and ML to predict future passenger arrival rates and Origin-Destination (OD) matrices.

- Allows proactive positioning of lifts (e.g., moving lifts to lobby before the morning rush).

Modern Approaches: Machine Learning and RL

Traffic Prediction

Using historical data and ML to predict future passenger arrival rates and Origin-Destination (OD) matrices.

- Allows proactive positioning of lifts (e.g., moving lifts to lobby before the morning rush).

Reinforcement Learning (RL)

Treating lift control as a sequential decision-making problem (Markov Decision Process).

- **Agents:** The lifts or a centralised controller.
- **State:** Positions of lifts, loads, pending calls.
- **Actions:** Move up, move down, stop.
- **Reward:** Minimised waiting/journey time.

Modern Approaches: Machine Learning and RL

Traffic Prediction

Using historical data and ML to predict future passenger arrival rates and Origin-Destination (OD) matrices.

Reinforcement Learning (RL)

Treating lift control as a sequential decision-making problem (Markov Decision Process).

- **Agents:** The lifts or a centralised controller.
- **State:** Positions of lifts, loads, pending calls.
- **Actions:** Move up, move down, stop.
- **Reward:** Minimised waiting/journey time.

RL agents can learn complex policies that adapt to dynamic traffic patterns and potentially outperform traditional heuristics.

Summary and Conclusion

We have explored the foundations of programming and applied them to a complex engineering challenge.

Programming Foundations

- **Algorithms:** Precise specifications; correctness relies on invariants (e.g., Euclid's GCD).

Summary and Conclusion

We have explored the foundations of programming and applied them to a complex engineering challenge.

Programming Foundations

- **Algorithms:** Precise specifications; correctness relies on invariants (e.g., Euclid's GCD).
- **Python Basics:** Types (Strong/Dynamic), Control Flow (`if`, `while`), Strings (Immutable sequences).

Summary and Conclusion

We have explored the foundations of programming and applied them to a complex engineering challenge.

Programming Foundations

- **Algorithms:** Precise specifications; correctness relies on invariants (e.g., Euclid's GCD).
- **Python Basics:** Types (Strong/Dynamic), Control Flow (`if`, `while`), Strings (Immutable sequences).
- **Structuring Code:** Functions, Scope (LEGB), Abstraction, DRY.

Summary and Conclusion

We have explored the foundations of programming and applied them to a complex engineering challenge.

Programming Foundations

- **Algorithms:** Precise specifications; correctness relies on invariants (e.g., Euclid's GCD).
- **Python Basics:** Types (Strong/Dynamic), Control Flow (`if`, `while`), Strings (Immutable sequences).
- **Structuring Code:** Functions, Scope (LEGB), Abstraction, DRY.
- **Robustness:** Testing (`assert`) and Error Handling (Exceptions).

Summary and Conclusion

We have explored the foundations of programming and applied them to a complex engineering challenge.

Computational Challenges (Lift Control)

- A complex stochastic optimisation problem (NP-hard).

Summary and Conclusion

We have explored the foundations of programming and applied them to a complex engineering challenge.

Computational Challenges (Lift Control)

- A complex stochastic optimisation problem (NP-hard).
- Balancing conflicting metrics (AWT vs. AJT).

Summary and Conclusion

We have explored the foundations of programming and applied them to a complex engineering challenge.

Computational Challenges (Lift Control)

- A complex stochastic optimisation problem (NP-hard).
- Balancing conflicting metrics (AWT vs. AJT).
- Information is key: DDS outperforms traditional systems (SCAN/LOOK) by reducing uncertainty.

Summary and Conclusion

We have explored the foundations of programming and applied them to a complex engineering challenge.

Computational Challenges (Lift Control)

- A complex stochastic optimisation problem (NP-hard).
- Balancing conflicting metrics (AWT vs. AJT).
- Information is key: DDS outperforms traditional systems (SCAN/LOOK) by reducing uncertainty.
- Modern techniques (ML/RL) offer potential for adaptive control.