CS1010X: Programming Methodology I
Chapter 7: Tuples and Data Abstraction

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 7: Tuples and Data Abstraction

Learning Objectives

By the end of this chapter, students should be able to:

- **Explain** what an *abstract data type* (ADT) is, and how it separates the logical view of data from its concrete implementation.
- **Model** real-world entities (e.g. students, rational numbers) using Python tuples as simple compound data, choosing appropriate fields for each model.
- **Use** tuples in Python, including construction, nesting, indexing, concatenation, length, and slicing, and **state** the rough time complexity of common tuple operations.
- **Compare and contrast** tuples and strings as sequence types: recognise that they share similar syntax but serve different semantic roles (heterogeneous records vs. homogeneous text).
- **Reason about** programs using box-and-pointer diagrams and stack/heap diagrams, including how tuple elements are stored as references rather than raw values.
- **Distinguish** between *value equality* (==) and *object identity* (`is`), and **decide** which to use in a given situation, especially when reasoning about aliasing.
- **Describe** immutability and referential immutability for tuples, and **analyse** the effects of aliasing and rebinding on shared tuple structures.
- **Design** simple abstract data types using tuples, by specifying and implementing constructors, selectors, predicates, and operations (e.g. for rational numbers).
- **Apply** the *abstraction barrier* principle: write client code that uses only constructors and selectors, and **identify** code that wrongly depends on the internal tuple representation.
- **Use** Python's file I/O to achieve basic data persistence: open, read, process, and close text files safely, and **process** CSV files using the `csv` module.

7.1 Functional Abstraction to Data Abstraction

In the previous chapters we used *functional abstraction*:

- hide irrelevant details of *how* something is computed,
- expose only the interface (parameters and return value),
- reuse the function as a building block for bigger programs.

Since programming is about describing objects and their properties/attributes, and computing new properties from them, we want similar abstractions for *data*. Instead of working with individual numbers and strings, we want to model richer objects.

For example:

- In picture processing: objects are images; properties might be width, height, colour channels.
- In GCD: objects are numbers; property might be "largest common divisor".
- In taxi fare: the object is a trip; properties include distance and time; we compute *fare*.

To do this well, we need good ways to organise data.

7.1.1 Abstract Data Types (ADT)

An **Abstract Data Type** is:

- a *logical view* of some kind of data (e.g. "student record", "point in 2D"),
- together with a set of operations we are allowed to do on it,
- independent of *how* it is implemented internally.

7.2 Tuples

7.2.1 Modelling objects with tuples

A student has many attributes:

- an identification number (e.g. NRIC or matric number),
- a name,
- a collection of courses taken,
- maybe more (year, faculty, GPA, ...).

We would like to bundle these pieces into one object.

Python provides the **tuple** type: a finite ordered collection of values, written with parentheses `( ... )`. Note that unlike tuples in other languages (like C++'s `std::tuple`), the elements of the tuple need not have the same types.

```
x = (1, 'ab')
print(x) # (1, 'ab')
```

We can access elements by index (starting from 0):

```
print(x[0]) # 1
print(x[1]) # 'ab'
```

Tuples can store arbitrary types, even other tuples:

```
y = (3, 4)
z = (x, y)        # a tuple of tuples
print(z)          # ((1, 'ab'), (3, 4))
print(z[0][1])    # 'ab'
print(z[1][0])    # 3
```

Importantly, tuples are **immutable**. That is, you cannot change an existing element once the tuple is created.

```
x = (1, 'ab')
x[0] = 99 # TypeError: 'tuple' object does not support item assignment
```

We can *create* new tuples from old ones (e.g. via concatenation), but we can never modify one in place.

Looking Forward: Benefits of Immutability

We will see many benefits of immutability in CS2030/S. For now, the most important property of tuples due to their immutability is their ability to be hashed, as we will see in the chapter on dictionaries.

Immutability vs Referential Immutability

There are two related but slightly different ideas:

- **True / deep immutability.**
  An object is deeply immutable if *neither it nor anything it reaches can change*. Examples in this course so far:
  - integers, floats,
  - strings,
  - tuples that contain only immutable values (numbers, strings, and such tuples again).

  If you have

  ```
  s = (1, 'ab', (2,3))
  ```

  then no operation in Python can change any part of this structure in place. The only way to get a different value is to build a *new* tuple or string.

- **Referential (shallow) immutability** A container is *referentially immutable* if the set of *references* it stores cannot change, even though the objects it points to might be mutable.

  Recall that everything in Python is an `Object`. Therefore, all values in a tuple are **references**, or pointers to their actual objects. We will see this idea more clearly in the next section.

  Therefore, while the actual reference may not change (we cannot change the pointer to point at another object), the *underlying* object can change its internal state (we will see such objects later, e.g. lists or custom classes). In that case, looking through the tuple later may show different contents, even though the tuple itself never changed which objects it refers to.

Therefore, tuples are only **referentially immutable**, not truly immutable objects.

7.2.2 Box-and-Pointer Notation

To reason clearly about tuples (and later, more complex structures), we use a pictorial model: **box-and-pointer notation**.

There are some key rules.

- Each tuple variable refers to a *tuple object*.
- The tuple object is drawn as a rectangular box split into cells (one per element).
- Each cell stores a *pointer* (arrow) to the actual value (number, string, or even another tuple).

Python Tutor's Representations

Do note that Python Tutor represents tuples with primitive values within the boxes. This is not accepted for examinations.

Example: `x = (1, 'ab')`

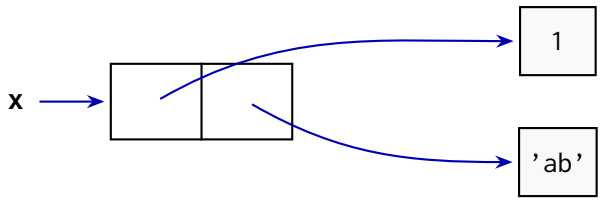

Figure 7.1: Box-and-pointer notation for `x = (1, 'ab')`. The tuple contains references (arrows) to the values, not the values themselves.

More detail: Stack and Heap

In more detail, we can concretely draw the stack and heap diagram for tuples. Note that indices only store the addresses to objects, not the value itself. This is why it is incorrect to write the value in the tuple boxes.

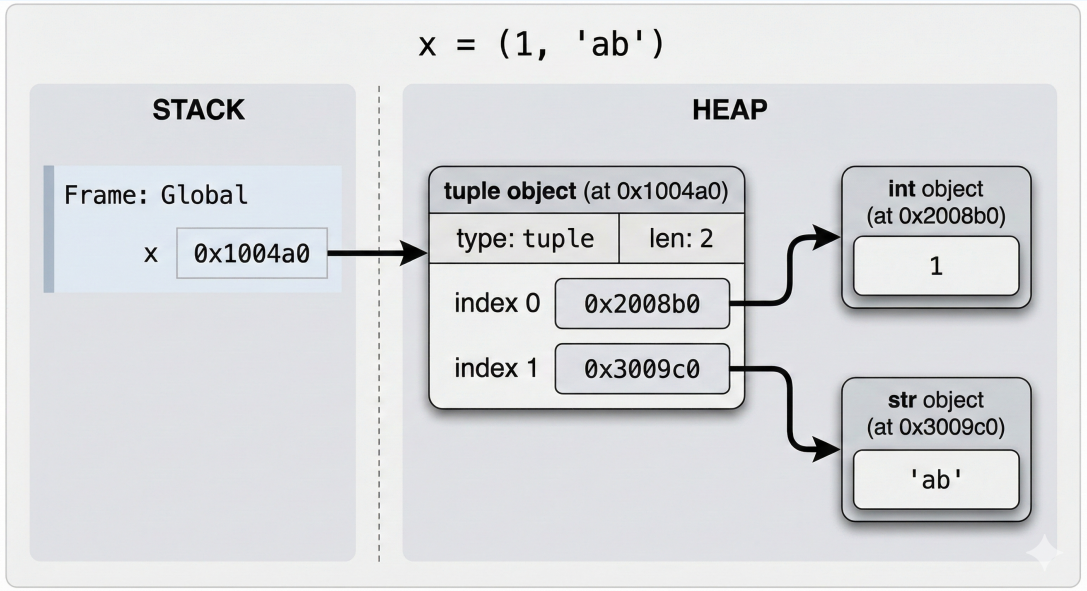

Figure 7.2: Addresses and indices for tuples.

7.2.3 Equality vs Identity: == and `is`

There are two ways to compare the equality of objects in Python:

- **Value Equality (==)** checks if values are equal.
- **Identity Equality (`is`)** checks if the references are equal.

For ==, it compares value (structural) equality,

- `a == b` compares if two objects have the same structure and values.
- **For numbers:** same numeric value (e.g. `3 == 3.0` is `True`).
- **For strings:** same sequence of characters.
- **For tuples:** same length and each corresponding element is equal.

For example:

```
a = (1, 2)
b = (1, 2)
c = (1, 3)

print(a == b)  # True  - same shape and same elements
print(a == c)  # False - last element differs
```

For `is`, it compares identity (object) equality.

- `a is b` determines if two variables referring to the exact same object in memory.
- This is the notion we used when talking about *aliasing* in the next section.

Continuing the example:

```
a = (1, 2)
b = (1, 2)
c = a

print(a == b)  # True  - same contents
print(a is b)  # Usually False - two separate tuple objects

print(a == c)  # True  - same contents
print(a is c)  # True  - c is just another name for a
```

Even though a and b *look* the same, they were created by two different tuple literals, so Python makes two separate tuple objects. Thus they are equal in value (==) but not identical in object identity (`is`).

IDLE vs. Interpreter

In the interactive IDLE shell, you will typically see:

```
a = (1, 2)
b = (1, 2)

print(a == b)  # True  - same contents
print(a is b)  # False - usually different objects in IDLE
```

However, if you put *exactly the same code* into a `.py` file and run it as a script, you may observe:

```
print(a == b)  # True
print(a is b)  # True  - often the same object in a script
```

Why the difference?

- In a **script** (`.py` file), Python compiles the whole file at once. The two literals `(1, 2)` are identical constants in the same code object, so CPython usually stores and reuses a single tuple object (this is termed *interning*).
- In the **IDLE shell** (and the plain REPL), each line (or block you run) is compiled separately. The two occurrences of `(1, 2)` live in different code objects, so Python typically creates two separate tuple objects, and `a is b` is `False`.

However, this is an *implementation detail*. You must **never rely** on `is` to compare ordinary values:

- Use == to compare *contents* (numbers, strings, tuples, etc.).
- Use `is` only for identity checks (e.g. `x is None`) or when explicitly reasoning about aliasing / object identity.

A warning about small numbers and strings

For performance, Python may *intern* some small integers and short strings, reusing the same object behind the scenes. That means you may sometimes see:

```
x = 10
y = 10
print(x is y)  # Often True (implementation detail!)

x = 1000
y = 1000
print(x is y)  # Often False
```

This behaviour is an implementation detail and must *never* be relied on. For comparing values, always use == rather than `is`.

In summary,

- Use == when you care about whether two values are *logically the same* (same number, same text, same tuple contents, . . .).

- Use `is` when you care about whether two names refer to the *same object*, usually for special singletons such as `None`:

```
if x is None:
    ...
```

- In this course, whenever you are reasoning about aliasing and box-and-pointer diagrams, `is` is the correct tool; whenever you are checking "does this value equal that one?" use `==`.

7.2.4 Aliasing and immutability

Since variables store *references to objects*. Two different variables can therefore refer to (or *alias*) the same underlying object. Consider:

```
x = (2, 7)
y = (x, 5)
```

The tuple `y` has two elements:

- `y[0]` is a *reference* to the same tuple object as `x`,
- `y[1]` is the integer `5`.

We can see the aliasing more clearly in the interpreter:

```
print(y[0] == x) # True
print(y[0] is x) # True, same underlying object
```

The following box-and-pointer diagram captures this situation:

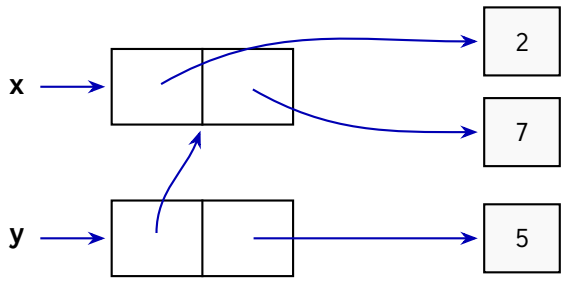

Figure 7.3: Aliasing: both `x` and `y[0]` refer to the same tuple $(2, 7)$.

There is still only *one* tuple object $(2, 7)$ in memory; it just happens to be reachable via two names: `x` and `y[0]`.

Immutability. Recall that tuples are immutable. You cannot change an element of `x` in place:

```
x[0] = 99 # TypeError: 'tuple' object does not support item assignment
```

Immutability means that once the tuple $(2, 7)$ is created, its two slots will always continue to point to the same objects. Every alias (`x`, `y[0]`, or any other reference) will always see $(2, 7)$ in that order.

Even though the *tuple* object cannot change, the *name* `x` can be rebound to refer to a different tuple:

```
x = (1, 4)
print(x)  # (1, 4)
print(y)  # ((2, 7), 5)
```

After this assignment:

- `x` now points to a completely new tuple $(1, 4)$,
- `y[0]` still points to the original tuple $(2, 7)$.

Python never "updates" `y` for you; assignment only affects the single name on the left-hand side, as seen below

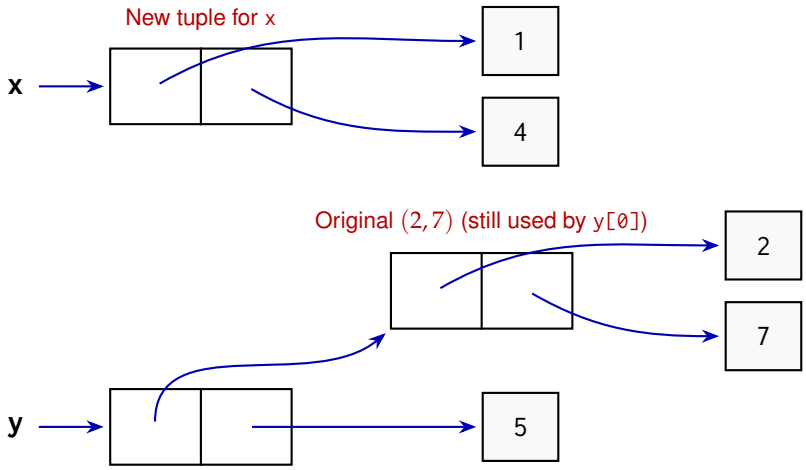

Figure 7.4: Rebinding `x`: the old tuple $(2, 7)$ remains unchanged and is still referenced by `y[0]`.

Aliasing, Equality, and Immutability

- Two names **alias** when they refer to the same object. In Python you can test this with `is`: `y[0] is x`.
- Tuples are immutable, so even if many names alias the same tuple, none of them can change its contents. Every alias will always see the same two elements.
- Assignment like `x = (1, 4)` never "updates" the tuple. It only changes which object the name `x` refers to. Other aliases (such as `y[0]`) are not affected.
- Later, when we work with mutable container types, aliasing becomes more subtle and can cause surprising behaviour. With tuples, aliasing is conceptually simple and safe: once a tuple is created, its shape and contents are fixed forever.

7.3 Tuple Operations in Python

7.3.1 Construction and Basic Operations

The most common tuple operations you will use in this module are summarised in Table 7.1. Here n denotes the length of the tuple and k the repetition factor.

7.3.2 Construction and Basic Operations

The most common tuple operations in this module are summarised in Table 7.1. Here n is the length of the tuple and k is the repetition factor.

Operation	Syntax / Example	Time	Notes
Constructors	`()`	$O(1)$	Create the unique empty tuple.
	`(1, 'ab'), (1, 2, 3)`	$O(n)$	Builds a new tuple of length n.
	`(42,)`	$O(1)$	Trailing comma is required; `(42)` is just the integer `42`.
Indexing	`triple[0], triple[2]`	$O(1)$	Random access by position; indices start at 0.
Length	`len(triple)`	$O(1)$	Length is stored with the tuple, so lookup is constant-time.
Concatenation	`a + b` `(1,2) + (3,4) # (1, 2, 3, 4)`	$O(\|a\| + \|b\|)$	Creates a *new* tuple containing all elements of `a` followed by all elements of `b`.
Repetition	`t * k` `('hi',) * 3 # ('hi', 'hi', 'hi')`	$O(n \cdot k)$	Returns a new tuple with the original contents repeated k times; the original tuple `t` is unchanged.

Table 7.1: Basic tuple operations: syntax, complexity, and behaviour.

7.3.3 Tuples and Strings: Conceptual Differences

Although tuples and strings share the same API for sequence operations, they serve fundamentally different roles in a program.

Syntactic Comparison

The following table demonstrates that syntax for retrieving information is identical.

Operation	Tuple Example	String Example
Empty value	`()`	`''`
Concatenation	`(1, 2) + (3, 4)` `# (1, 2, 3, 4)`	`'12' + '34'` `# '1234'`
Repetition	`(1, 2) * 2` `# (1, 2, 1, 2)`	`'No' * 2` `# 'NoNo'`
Length	`len((1, 2, 3))`	`len('abc')`
Indexing	`t = (10, 20, 30)` `t[1] # 20`	`s = 'abc'` `s[1] # 'b'`
Slicing	`t[1:]` `# (20, 30)`	`s[1:]` `# 'bc'`

Table 7.2: Side-by-side comparison of tuple and string operations.

Semantic Differences: Homogeneity vs. Heterogeneity

While Python enforces no rules on what you store, there is a strong convention regarding "logical" types:

- **Strings are Homogeneous (Flat):** A string is strictly a sequence of *characters*. It represents **textual data**. A string does not contain other types; it is atomic in that it represents a single piece of text.
- **Tuples are Heterogeneous (Structural):** A tuple is generally used to group *different* types of data that belong together. It represents a **record** or a structure.

Therefore, semantically, strings are a single unit of data, and should be treated as such.

7.4 Slicing Tuples

Since tuples are ordered sequences, they support the exact same slicing semantics as strings. The syntax `t[start:stop:step]` applies.

Recap: Sequence Slicing Syntax

The notation `sequence[start:stop:step]` works identically for both:

- **start**: Index of first element (inclusive). Defaults to 0.
- **stop**: Index *after* the last element (exclusive). Defaults to length.
- **step**: The spacing (stride). Defaults to 1.

Suppose we have

```
t = ('a', 'b', 'c', 'd', 'e', 'f')
```

The following table illustrates common slicing patterns applied to tuple `t`.

Pattern	Result	Explanation
Standard Range		
`t[0:2]`	`('a', 'b')`	Elements from index 0 up to (but not including) 2.
`t[2:5]`	`('c', 'd', 'e')`	Elements from index 2 up to 5.
Omitted Indices		
`t[:2]`	`('a', 'b')`	Start omitted → defaults to 0.
`t[2:]`	`('c', 'd', 'e', 'f')`	Stop omitted → defaults to end of tuple.
`t[:]`	`('a', ..., 'f')`	Returns a shallow copy of the entire tuple.
Strides (Step)		
`t[::2]`	`('a', 'c', 'e')`	Every 2nd element (0, 2, 4).
`t[1:5:3]`	`('b', 'e')`	Start at 1, take every 3rd, stop before 5.
Negative Step		
`t[::-1]`	`('f', 'e', ..., 'a')`	Returns the reversed tuple.
`t[-1::-2]`	`('f', 'd', 'b')`	Start at last element, step backwards by 2.

Table 7.3: Tuple slicing patterns and results.

Immutability Note

Slicing *never* modifies the original tuple.

- `t[1:3]` creates a **new tuple** object.
- The original `t` remains unchanged in memory.

7.5 Compound Data Abstractions

In previous sections, we learned how to use tuples to group data. However, working directly with raw tuples (e.g. `t[0]`, `t[1]`) can become messy and error-prone as programs grow.

To solve this, we use **data abstractions**. We will demonstrate this concept by building a software package to handle **rational numbers**.

7.5.1 Guidelines for Creating Compound Data

When designing a new data type, we should not just write ad-hoc code. We follow a formal "Contract" or "Interface" design pattern consisting of four key components:

1. **Constructors**: Functions that build the compound data from primitive pieces.
2. **Selectors (Accessors)**: Functions that retrieve specific parts of the compound data.
3. **Predicates**: Functions that ask True/False questions (e.g., "Is this a valid rational number?" or "Are these two equal?").
4. **Operations**: Functions that perform calculations (Arithmetic) using the data.

7.5.2 Rational Numbers

A rational number is a number that can be expressed in the form $\frac{n}{d}$, where n (the numerator) and d (the denominator) are integers. We create a dedicated package because

- Native `float` types (like `0.3333...`) are imprecise approximations.
- Rational numbers ($\frac{1}{3}$) maintain exact precision.
- They naturally require two integers to represent one conceptual value, making them a perfect candidate for compound data.

7.5.3 The Abstraction Layer

We begin by defining the "lowest level" of our package. This is the **only** place where we interact directly with the Python tuple.

```
# 1. CONSTRUCTOR
def make_rat(n, d):
    """Creates a rational number n/d."""
    return (n, d)  # We choose to implement it as a tuple

# 2. SELECTORS
def numer(r):
    """Returns the numerator."""
    return r[0]   # Direct index access

def denom(r):
    """Returns the denominator."""
    return r[1]   # Direct index access
```

7.5.4 Arithmetic Operations (The "Business Logic")

Now that we have `make_rat`, `numer`, and `denom`, we can write the logic for arithmetic.

Crucially, these functions should *not* know that the data is a tuple. They should only use the selectors.

Operation	Mathematical Formula	Python Implementation
Addition	$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$	`make_rat(n1*d2 + n2*d1, d1*d2)`
Subtraction	$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$	`make_rat(n1*d2 - n2*d1, d1*d2)`
Multiplication	$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$	`make_rat(n1*n2, d1*d2)`
Division	$\frac{n_1}{d_1} \div \frac{n_2}{d_2} = \frac{n_1 d_2}{d_1 n_2}$	`make_rat(n1*d2, d1*n2)`

Table 7.4: Implementing arithmetic using the Constructor/Selector interface.

Therefore, we can implement each function independent of the underlying representation of the rational number. If we were to change the underlying representation, this layer should **not** break.

```
def add_rat(x, y):
    # Retrieve components using SELECTORS (not indices!)
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)

    # Calculate new numerator and denominator
    new_numer = nx * dy + ny * dx
    new_denom = dx * dy

    # Return a new rational number using CONSTRUCTOR
    return make_rat(new_numer, new_denom)
```

7.5.5 The Abstraction Barrier

After implementing this package, the most important concept that you should take away is the concept of the **abstraction barrier**. It separates the *use* of data from its *representation*.

Notice that there are actually 4 layers of abstractions created, and we should be able to operate at any layer without knowing how deeper layers are implemented.

- **Level 1 (User Layer):** Programs that use the library (e.g. `total = add_rat(half, third)`).
- **Level 2 (Operations):** Functions like `add_rat`, `equal_rat`. These rely on selectors.
- **Level 3 (Representation):** `make_rat`, `numer`, `denom`. These rely on tuples.
- **Level 4 (Primitives):** Python Tuples, Integers.

Violation of the Barrier

Breaking the barrier occurs when a high-level function bypasses the selectors and touches the tuple directly.

Example: Creating a Printer Function

```
# BAD IMPLEMENTATION (Barrier Violation)
def print_rat_bad(r):
    # This code "knows" r is a tuple. If we change r to be a
    # dictionary later, this code breaks.
    print(str(r[0]) + "/" + str(r[1]))

# GOOD IMPLEMENTATION (Respects Abstraction)
def print_rat_good(r):
    # This code only uses the interface. It works regardless of
    # how 'make_rat' is implemented.
    print(str(numer(r)) + "/" + str(denom(r)))
```

7.5.6 Optimization: Reducing to Lowest Terms

A professional library should handle simplification. With our current code:

$$\frac{1}{3} + \frac{1}{3} \to \frac{6}{9}$$

Mathematically $\frac{6}{9}$ is correct, but we prefer $\frac{2}{3}$.

We can fix this by upgrading the **constructor**. We use the Greatest Common Divisor (GCD) to divide both terms before storing them.

```
from math import gcd

def make_rat(n, d):
    g = gcd(n, d)
    # The // operator ensures integer division
    return (n // g, d // g)
```

Notice that the functions that we have written before should still work! In fact, this is the key idea about abstraction: optimizing one layer or changing its representation should not affect layers above it.

7.6 Data Persistence: Working with Files

Real-world applications (like the NUS Registrar system) deal with records that must persist after the program ends. In reality, data is stored in persistent storage, such as files and databases. In particular, we will look at files in this section.

Working with a file always involves a three-step sequence: open → process → close.

7.6.1 Reading a File

Reading is done using the `open()` function with mode `'r'`. Since we often don't know the file size, we use a standard `while` loop pattern.

```
def read_file(filename):
    # 1. OPEN the file in read mode ('r')
    infile = open(filename, 'r')
    # Read the first line to start the process
    line = infile.readline()

    # 2. LOOP: Check for End of File (EOF)
    # An empty string '' means we hit the end.
    while line != '':
        clean_line = line.strip() # Remove leading and trailing
            whitespaces
        print("Data:", clean_line)
        # 3. UPDATE: Read the next line for the next loop
        line = infile.readline()

    # 5. CLOSE the file
    infile.close()
```

7.6.2 Writing to a File

Writing is done using mode `'w'`.

Warning: Overwriting

Opening a file in `'w'` mode will **erase** any existing file with that name immediately. Use with caution!

```
def save_report(filename):
    # 1. OPEN in write mode ('w')
    output = open(filename, 'w')

    # 2. WRITE strings. Note: You must manually add '\n' for newlines
    output.write("HELLO WORLD\n")
    output.write("End of Report\n")

    # 3. CLOSE to flush data to disk
    output.close()
```

7.6.3 Working with CSV Files

CSV (Comma-Separated Values) is the standard format for simple database records. It mimics a spreadsheet structure.

In particular, in older Practical Examinations, you will often see a question on processing CSV files.

CSV files are usually structured like spreadsheets, and usually have the following format:

- **Header Row (Index 0)**: Describes the columns (e.g. `Date`, `Tweet`).
- **Data Rows (Index 1+)**: The actual records.

This is an example on how to read and work with CSV files, using the `csv` library. Do note that the `read_csv()` function is often provided for you.

```
import csv

def read_csv(csvfilename):
    """
    Reads a CSV file and returns a list of lists
    containing rows in the csv file and its entries.
    """
    rows = []

    # Open the file using 'with' (automatically closes file)
    with open(csvfilename, 'r') as csvfile:
        file_reader = csv.reader(csvfile)

        # Iterate over each row in the CSV
        for row in file_reader:
            rows.append(row)

    return rows

def get_tweets(target_date):
    # Assume read_csv returns a list of lists (rows/cols)
    data = read_csv('tweets.csv')

    # Skip the header (data[0]) and look at records
    for row in data[1:]:
        tweet_date = row[6]  # Date is at index 6
        tweet_text = row[2]  # Text is at index 2

        if tweet_date == target_date:
            print(tweet_text)
```

Looking Forward: Databases (SQL)

While text files and CSVs are useful for simple storage, large-scale systems (like the NUS Registrar) use **databases**. Databases allow for efficient querying without loading the entire file into memory.
Python provides built-in support for databases via the `sqlite3` library. Instead of reading lines, we execute **SQL (Structured Query Language)** commands.

```
import sqlite3

# 1. Connect to the database file
con = sqlite3.connect('university.db')
cur = con.cursor()

# 2. Execute a SQL Query (Find all students with grade 'A')
# This is much faster than looping through a text file!
cur.execute("SELECT name FROM students WHERE grade = 'A'")

# 3. Fetch and print results
for row in cur.fetchall():
    print(row)

con.close()
```

You will learn more about databases in courses like CS2102 and CS3223.