**Identifiers**

In Python, an **identifier** is a name used to identify a variable, function, class, module, or other object.

R**ules for Identifiers:**

* Can only contain letters (a-z, A-Z), numbers (0-9), and the underscore (\_).
* Must start with a letter or an underscore (e.g., \_my\_var, name).
* Cannot start with a number (e.g., 1variable is invalid).
* Case-sensitive (myVar is different from myvar).
* Cannot be one of Python's **keywords** (reserved words like if, else, for, while, def, class, import, True, False, None, etc.).

**Examples:**

**Instructions (Statements)**

In programming, an instruction or statement is a command that the Python interpreter can execute. It's a single logical line of code that performs an action.

**Examples:**

Python

# This is an instruction (assignment statement)

x = 10

# This is an instruction (function call statement)

print("Hello, Python!")

# This is an instruction (conditional statement)

if x > 5:

print("x is greater than 5")

# This is an instruction (loop statement)

for i in range(3):

print(i)

**Indentation:**

Whitespace (spaces or tabs) used to define blocks of code (like inside if statements or functions).

**Rule:** All lines in the same block must have the same indentation level.

**Docstrings**

**What it is:** A multi-line string used as documentation right after a function, class, or module definition. It explains what the code does.

**Access:** You can see it using help() or .\_\_doc\_\_.

Example: def multiply\_numbers(a, b):

""" This function takes two numbers and returns their product. It's a simple multiplication utility. """

return a \* b

# You can access the docstring like this: print(multiply\_numbers.\_\_doc\_\_)

**Variables**

A **variable** is essentially a named storage location.

Imagine a **box** where you can put a piece of information. A **variable** is just a **name** you give to that box.

**Example:**

Here, 'item\_name' is a variable storing the string "Laptop"

item\_name = "Laptop"

**Boolean**

A Boolean is a super simple type of data that can only have one of two possible values:

* True
* False

Booleans are fundamental for making **decisions** in your code, especially with if statements and loops. They are the result of comparison operations (like > greater than, == equal to, etc.).

**String:**

A string is Python's way of handling text. It's a sequence of characters (like letters, numbers, symbols, spaces) enclosed within quotes.

we can define string in single, double, or triple quotes.

Strings are immutable: Once you create a string, you cannot change individual characters within it.

**String Indexing**

String indexing is how you access a single character within a string.

Imagine the word **"UBER"** is like a **line of cars, and each letter is a person sitting in a specific car.**

my\_ride = "UBER"

Think of it like this:

|  |  |  |
| --- | --- | --- |
| **Car (Letter)** | **Position (Index) - from the start** | **Position (Index) - from the end** |
| **U** | **0** (The very first car) | **-4** |
| **B** | **1** | **-3** |
| **E** | **2** | **-2** |
| **R** | **3** (The last car) | **-1** (The very last car) |

When you do **String Indexing**, you're telling Python: "Go to this specific car number in the line and tell me who's in it!"

* my\_ride[0] means: "Who's in **car number 0** (the first car)?" **Answer:** U
* my\_ride[2] means: "Who's in **car number 2**?" **Answer:** E
* my\_ride[-1] means: "Who's in the **last car** (-1)?" **Answer:** R
* my\_ride[-4] means: "Who's in the **fourth car from the end** (-4)?" **Answer:** U

**Log Files and its Types**

Text files that record events from your program (messages, warnings, errors). Essential for debugging and monitoring.

**Appending to a Log File**

You write today's entry on the next empty page in your diary. You don't erase yesterday's entry; you just add to the end.

When a program appends to a log file, it means it adds new messages to the end of the existing log file, without deleting what was already there. This is the most common way to write logs.

**Cycling (Log Rotation)**

What happens when your diary gets **too thick** to hold, or you want to **start a fresh one** each month/year? You put the old, full diary on a shelf (maybe label it "January 2025") and start a brand new, empty diary for the new month.

**Cycling** (**Log Rotation**) is when a log file gets too big, or too old, and the program decides to:

Close the current log file

Rename the old log file

Start a brand new, empty log file with the original name

It might even delete very old rotated log files to save space

**List indexing**

List indexing is how you access a single item within a list using its position or number.

Unlike strings, lists are mutable, which means you can change the items in a list using their index.

daily\_tasks = ["Wake up", "Eat breakfast", "Go to work", "Attend meeting", "Buy groceries", "Cook dinner", "Relax"]

# Python sees this list like:

# Index: 0 1 2 3 4 5 6

# Task: "Wake up" "Eat breakfast" "Go to work" "Attend meeting" "Buy groceries" "Cook dinner" "Relax"

| **🔤 Topic** | **📝 Simple Code** | **📤 Output** | **🚗 Real-Life Example** |
| --- | --- | --- | --- |
| **List Indexing** | rides = ["Airport", "Office", "Mall"]print(rides[1]) | Office | Shows your 2nd trip |
| **List Slicing** | print(rides[0:2]) | ['Airport', 'Office'] | Shows first 2 trips |
| **Tuple Indexing** | places = ("Home", "Gym", "Store")print(places[0]) | Home | Shows 1st saved location |
| **Tuple Slicing** | print(places[1:]) | ('Gym', 'Store') | Shows places from 2nd onward |
| **Tuple Reindexing** | temp = list(places)temp[0] = "New Home"places = tuple(temp)print(places) | ('New Home', 'Gym', 'Store') | Updating your saved place |
| **Sorting** | fares = [300, 150, 450]fares.sort()print(fares) | [150, 300, 450] | Sorting rides by price |
| **Set Operation** | a = {"Ali", "Ravi"}b = {"Ali", "John"}print(a & b) | {'Ali'} | Same driver both days |
| **Lambda** | fare = lambda km: km \* 10print(fare(5)) | 50 | Fare per km |
| **filter()** | ratings = [4.8, 3.9, 5.0]good = list(filter(lambda r: r > 4.5, ratings))print(good) | [4.8, 5.0] | Only best-rated trips |
| **map()** | fares = [100, 200]tips = list(map(lambda x: x+20, fares))print(tips) | [120, 220] | Add tips to fares |
| **reduce()** | from functools import reducefares = [100, 200, 300]total = reduce(lambda x, y: x + y, fares)print(total) | 600 | Total daily earnings |

# Uber-Style Python Basics Examples

# List Indexing

rides = ["Airport", "Office", "Mall"]

print("List Indexing:", rides[1])  # Output: Office

# List Slicing

print("List Slicing:", rides[0:2])  # Output: ['Airport', 'Office']

# Tuple Indexing

places = ("Home", "Gym", "Store")

print("Tuple Indexing:", places[0])  # Output: Home

# Tuple Slicing

print("Tuple Slicing:", places[1:])  # Output: ('Gym', 'Store')

# Tuple Reindexing (modifying tuple)

temp = list(places)

temp[0] = "New Home"

places = tuple(temp)

print("Tuple Reindexing:", places)  # Output: ('New Home', 'Gym', 'Store')

# Sorting

fares = [300, 150, 450]

fares.sort()

print("Sorted Fares:", fares)  # Output: [150, 300, 450]

# Set Operations

day1 = {"Ali", "Ravi"}

day2 = {"Ali", "John"}

print("Common Driver:", day1 & day2)  # Output: {'Ali'}

# Lambda Function

fare = lambda km: km \* 10

print("Lambda Fare (5 km):", fare(5))  # Output: 50

# filter() Example

ratings = [4.8, 3.9, 5.0]

good = list(filter(lambda r: r > 4.5, ratings))

print("High Ratings:", good)  # Output: [4.8, 5.0]

# map() Example

fares = [100, 200]

tips = list(map(lambda x: x + 20, fares))

print("Fares with Tips:", tips)  # Output: [120, 220]

# reduce() Example

from functools import reduce

fares = [100, 200, 300]

total = reduce(lambda x, y: x + y, fares)

print("Total Fare:", total)  # Output: 600

**Problem Statement: Uber Ride Dispatch and Analytics System**

Uber wants to build a lightweight **Python-based backend prototype** to:

* Handle **ride dispatching**
* Track **trip details**
* Analyze **driver and trip data**
* Support **real-time filtering and processing**

You are tasked with creating a system using core Python concepts.

**🎯 Functional Requirements (Mapped to Concepts):**

1. **Identifiers & Variables**
   * Define valid identifiers for drivers, riders, and trip status.
   * Store current locations, destination, trip duration, and fare.
2. **Indentation & Instructions**
   * Follow proper indentation to structure ride assignment and payment logic.
3. **Docstrings**
   * Use docstrings to explain functions like assign\_driver(), calculate\_fare(), etc.
4. **Strings & Boolean**
   * Use strings to store locations, names, and statuses like "available", "on\_trip".
   * Use booleans for trip state (e.g., is\_completed = True).
5. **String Indexing**
   * Extract city codes from trip location strings like "NYC-TimesSquare" → "NYC".
6. **Log Files and Types**
   * Write trip logs to a file: "trip\_log.txt".
   * Support log types: "INFO", "ERROR", "TRIP\_COMPLETED".
7. **List Indexing & Slicing**
   * Use lists to manage active drivers and recent trips.
   * Slice the last 5 trips for report generation.
8. **Tuple Indexing & Slicing**
   * Use tuples for immutable driver profiles like: (driver\_id, name, rating).
   * Slice tuples to get partial data (e.g., name and rating only).
9. **Sort & Set Operations**
   * Sort drivers by ratings or trip counts.
   * Use sets to get unique zones with active trips.
10. **Lambda, Filter, Map, Reduce**

* lambda: Calculate surge pricing
* filter: Find drivers above 4.8 rating
* map: Convert list of trip durations from minutes to hours
* reduce: Compute total earnings of all drivers in a day