

Python Review

لا تنسونا من دعواتكم

اللَّهُ يوفقنا جميعًا ونتقابل في الصيفيه 🙏
هذي حساباتنا لو عندكم اي اقتراحات او ملفات ممكن ننشرها :
Ice Sara Muath



The Atoms of Code: Storing Data and Performing Operations.

Variables & Dynamic Typing

In Python, you don't need to declare a variable's type. The interpreter figures it out automatically.

```
# No need for "int x = 5"
x = 5
y = "hello"
z = True
```

Essential Operators

Arithmetic		Comparison		Logical	
+, -, *, /	Addition, Subtraction, Multiplication, Division	==	Equal to	and	True if both are true
//	Floor Division: rounds down	!=	Not equal to	or	True if one is true
%	Modulo: remainder	>, <	Greater/Less than	not	Reverses the result
**	Exponentiation: 6 ** 2 is 36				

Handling Collections: The Power of Lists

In AI, we rarely work with a single data point. We deal with vast collections. The `list` is our primary tool for organizing this data. Lists are ordered, mutable (changeable), and can hold items of different types.

Creation

```
my_list = [1, "hello", 3.14, True]
```

Positive Indexing

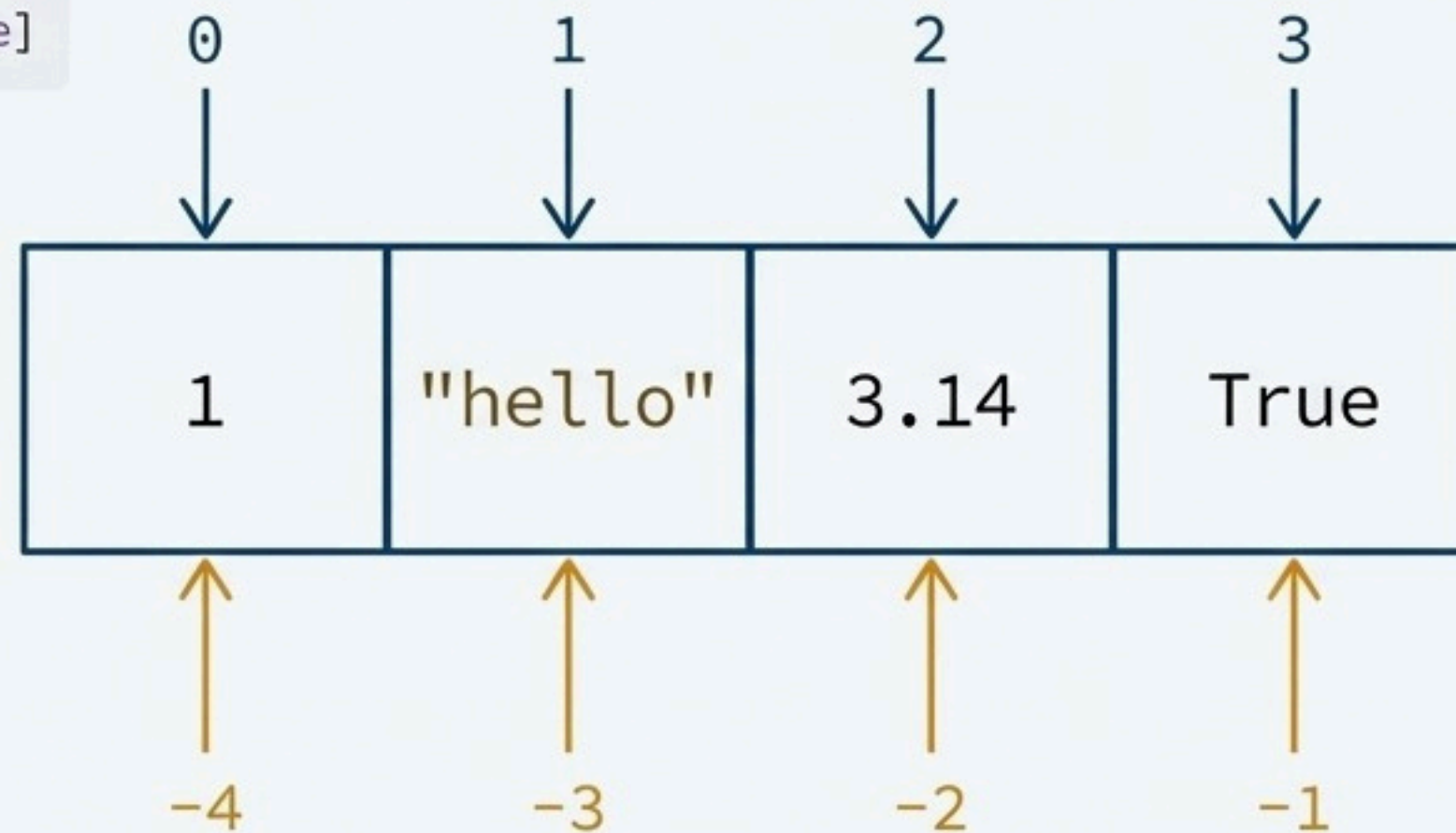
```
my_list[0]  
returns 1
```

Negative Indexing

```
my_list[-1]  
returns True
```

Slicing: The powerful
[start:end:step] **syntax for**
selecting subsets

```
my_list[1:3]  
returns ['hello', 3.14]  
  
my_list[:2]  
returns [1, 'hello']  
  
my_list[1:]  
returns ['hello', 3.14, True]
```



Modifying and Querying Lists.

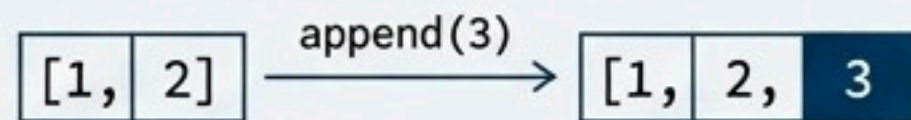
Adding an element

`.append()`

Before `my_list = [1, 2]`

Code `my_list.append(3)`

After `[1, 2, 3]`



append adds its argument as a single element. **extend** iterates over its argument, adding each item individually.

`[1, 2].append([3, 4]) → [1, 2, [3, 4]]`

A diagram showing the transformation of `[1, 2]` to `[1, 2, [3, 4]]` using `append`. The inner list `[3, 4]` is shown in a box with an arrow pointing to its position in the outer list.

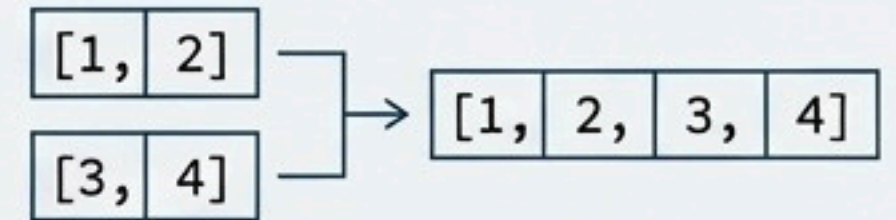
Merging lists

`.extend()`

Before `my_list = [1, 2]`
`other_list = [3, 4]`

Code `my_list.extend(other_list)`

After `[1, 2, 3, 4]`



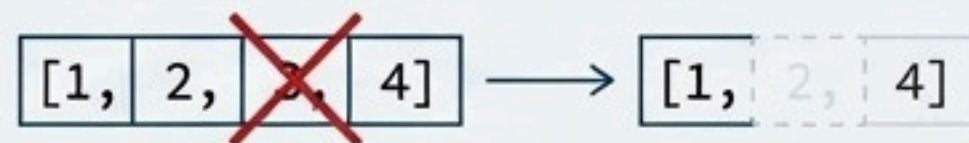
Removing an element

`del`

Before `my_list = [1, 2, 3, 4]`

Code `del my_list[2]`

After `[1, 2, 4]`



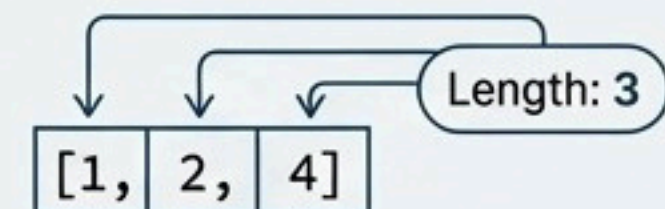
Getting the length

`len()`

Code `len([1, 2, 4])`

Result

Returns 3



When Order Isn't Enough: Structuring Data with Dictionaries.

Sometimes, accessing data by a numeric position (`[0]`, `[1]`) isn't meaningful. We need to access it by a descriptive label. Dictionaries store data as **key-value pairs**, just like a real dictionary. This is how we represent feature vectors in AI.

A Single Data Point for a House Price Model

```
house = {  
    "area_sqft": 1500,  
    "num_rooms": 4,  
    "location": "Riyadh",  
    "price_usd": 500000  
}
```

```
# Accessing Data  
price = house["price_usd"]
```

```
# Modifying Data  
house["num_rooms"] = 5
```

```
# Adding New Data  
house["has_garage"] = True
```



A Quick Look at Tuples: Immutable Collections.

A tuple is like a list, but it cannot be changed after it's created. This is known as being **immutable**. You use them when you want to guarantee that a collection of values remains constant.

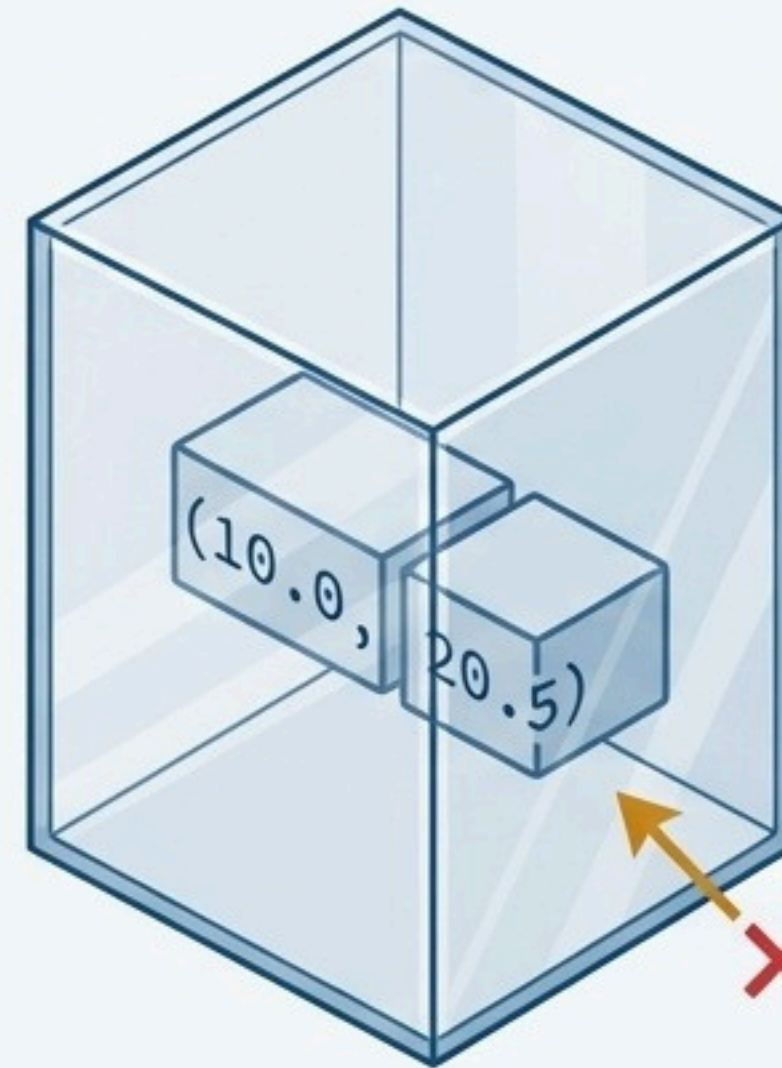
Syntax

Note the use of parentheses ().

```
coordinates = (10.0, 20.5)
```

Primary Use Case

Data integrity. Perfect for things that shouldn't change, like geographic coordinates, RGB color values, or fixed configuration settings.



```
# This will work:  
x = coordinates[0]
```

```
# This will cause an error:  
coordinates[0] = 15.0
```

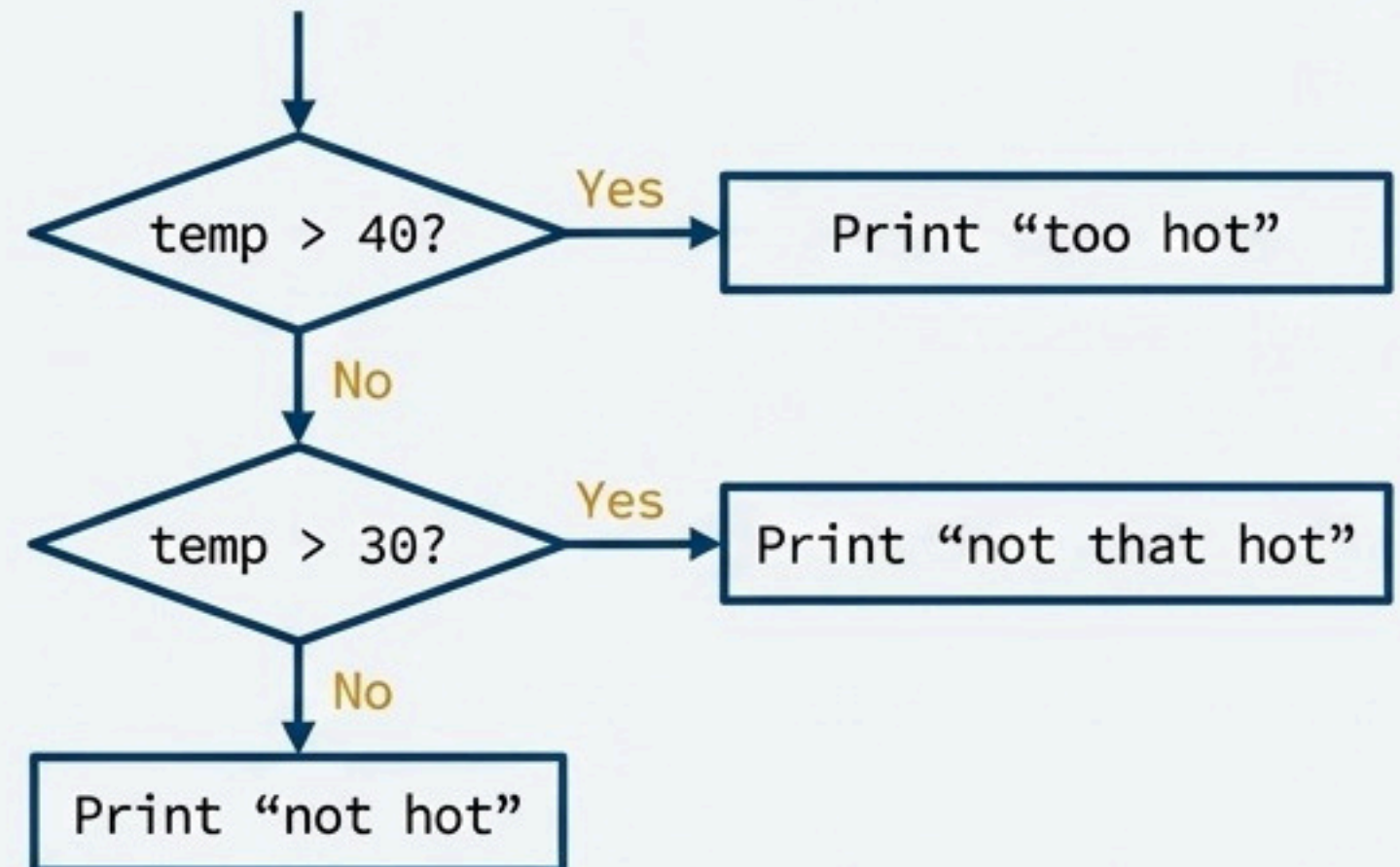
Making Decisions with Conditional Logic.

Intelligent systems constantly make decisions. In Python, we use `if`, `elif` (else if), and `else` statements to embed this logic into our code.

****The Crucial Rule: Indentation Defines Structure.****

Python uses whitespace (indentation), not brackets `{}`, to define code blocks. This is not optional; it is a core part of the syntax.

```
temperature = 35
if temperature > 40:
    print("It's too hot.")
elif temperature > 30:
    print("It's not that hot.")
else:
    print("It's not hot.")
```



Repetition with Purpose: The Power of Loops.

Data processing involves repeating the same operation thousands or millions of times. Loops are our engine for this automation.

The Pythonic `for` Loop

Iterating directly over items is clean and readable.

Verbose Java-style loop

```
for (int i = 0; i < numbers.length; i++) {  
    System.out.println(numbers[i]);  
}
```

Elegant Python loop

```
numbers = [1, 2, 3, 4]  
for num in numbers:  
    print(num * num)
```

→
1
4
9
16

Generating Sequences with `range()`

For looping a specific number of times.

`range(5)` → 0, 1, 2, 3, 4

`range(2, 7)` → 2, 3, 4, 5, 6

`range(0, 10, 2)` → 0, 2, 4, 6, 8

The Pythonic Way: Elegant Loops with List Comprehensions

This is a signature feature of Python that lets you create lists concisely. It combines the loop and creation of a new list into a single, readable line.

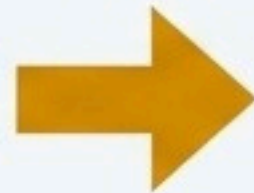
The Formula

```
new_list = [ output_expression for item in iterable if condition ]
```

The Transformation

The Standard Way

```
squares = []  
for x in range(10):  
    if x % 2 == 0: # only even numbers  
        squares.append(x ** 2)
```



The Pythonic Way

```
squares = [x ** 2 for x in range(10) if x % 2 == 0]
```

Building Reusable Blocks: An Introduction to Functions

Great programmers don't repeat themselves. They build reusable tools to perform specific tasks. In programming, these tools are called **functions**.

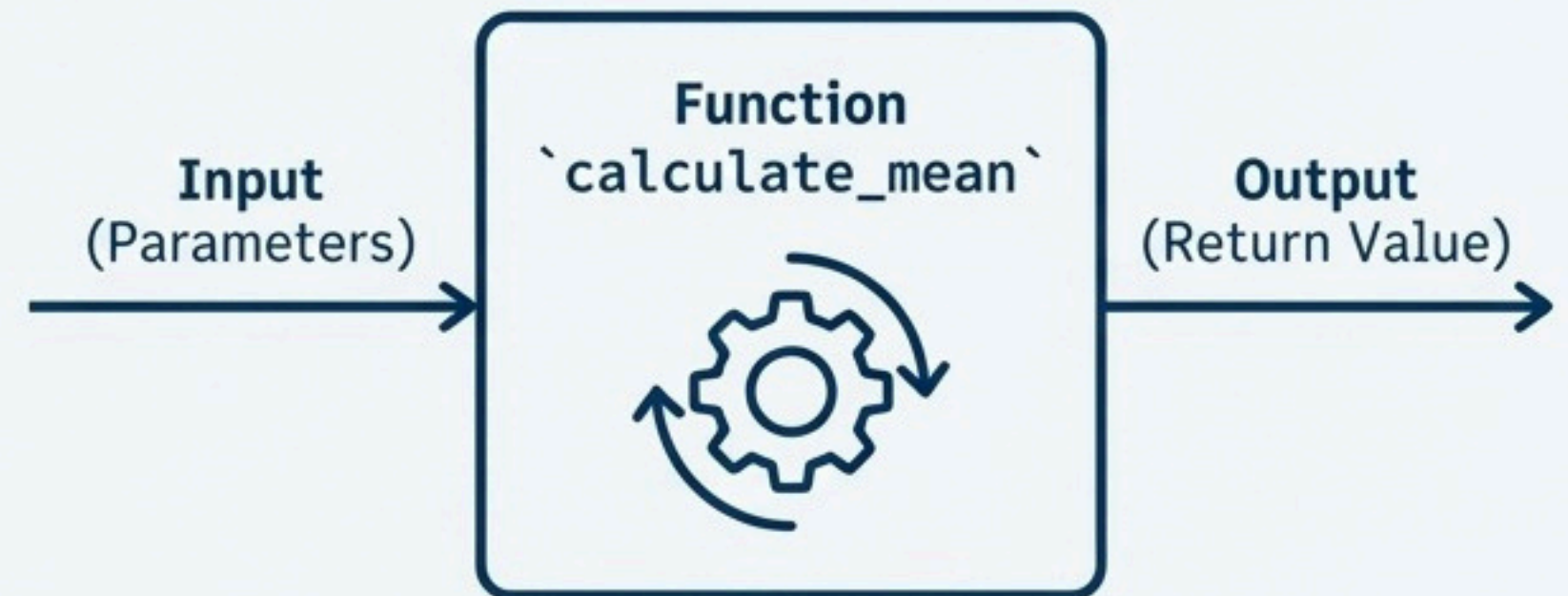
Anatomy of a Function

- **def**: The keyword to define a function.
- **Parameters (Inputs)**: Data the function needs.
- **Function Body**: The indented block of code.
- **return (Output)**: The value sent back.

```
def calculate_mean(data_list):  
    total = sum(data_list)  
    count = len(data_list)  
    return total / count
```

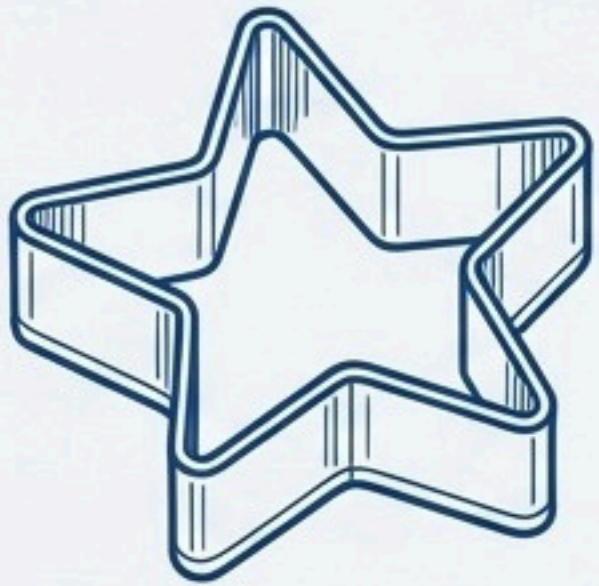
Using the function

```
my_grades = [88, 92, 100, 78]  
average_grade = calculate_mean(my_grades)  
print(average_grade) # Output: 89.5
```

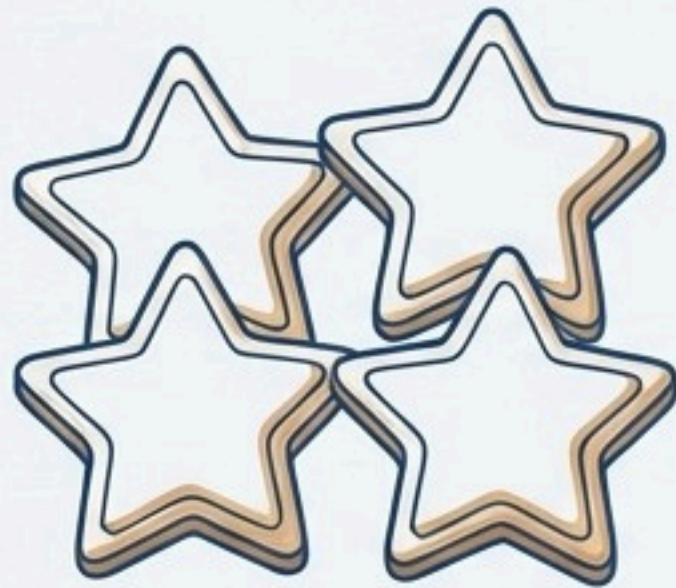


Blueprints for Data: Structuring Code with Classes and Objects.

As programs grow, we need a way to organize related data and functions. **Classes** are blueprints for creating **objects**—self-contained instances that bundle data (attributes) and the functions that operate on that data (methods).



Class (CookieCutter)



Objects (cookie1,
cookie2, cookie3)

Key Concepts

- **Class**: The blueprint. (e.g., a Car blueprint).
- **Object**: A specific instance created from the blueprint. (e.g., my_tesla, my_corolla).
- **`__init__`**: A special method called a **constructor**. It runs automatically when you create a new object to initialize its attributes.
- **`self`**: A special variable that represents the instance of the object itself. It's used to access attributes and methods within the class.

Classes in Action: Building a `DataSplitter` for AI.

Nearly all tools in major AI libraries are implemented as classes. Let's build a simplified class that performs a common machine learning task: splitting data into training and testing sets.

```
class DataSplitter:
    # The constructor initializes the object with a state
    def __init__(self, train_ratio=0.8):
        self.train_ratio = train_ratio

    # A method that performs an action using the object's state
    def split(self, data):
        split_index = int(len(data) * self.train_ratio)
        train_data = data[:split_index]
        test_data = data[split_index:]
        return train_data, test_data

# How it's used:
my_data = list(range(100))
splitter = DataSplitter(train_ratio=0.7) # Create the object
train, test = splitter.split(my_data)    # Use its method
```

Stores the "state" of this specific splitter object.

A function "bound" to the object that can use its state.