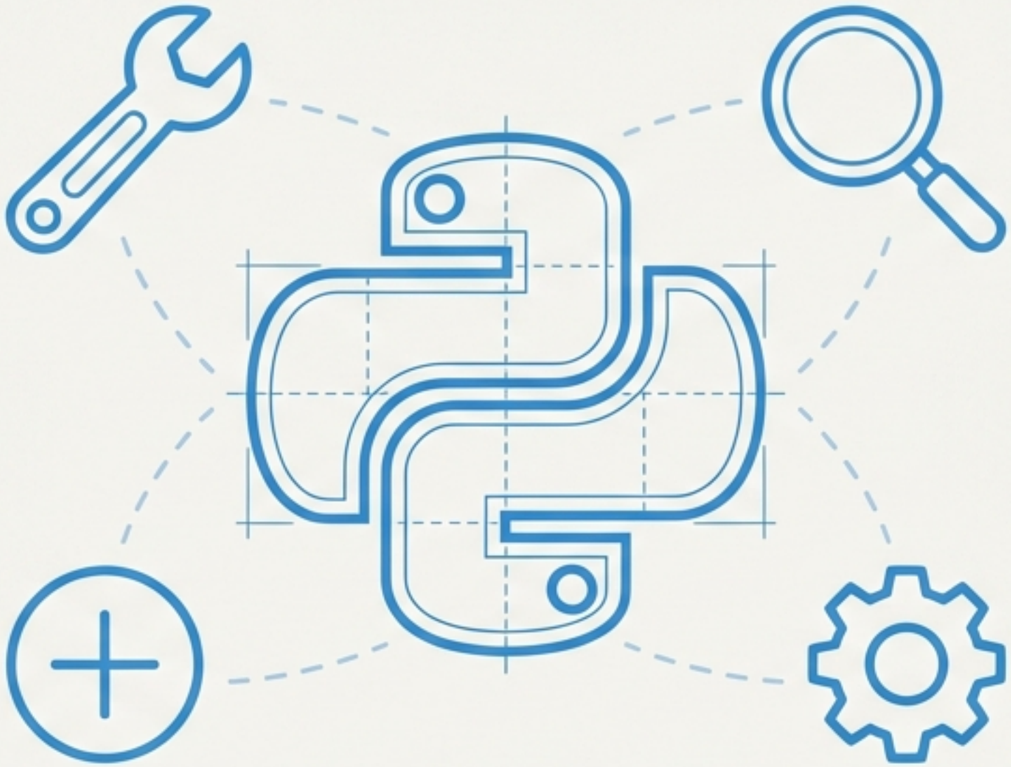


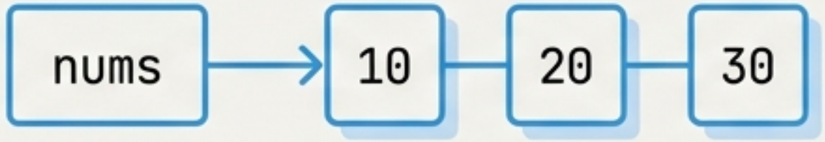
THE DEVELOPER'S TOOLKIT: MASTERING THE PYTHON LIST

A practical guide to Python's most fundamental data structure.

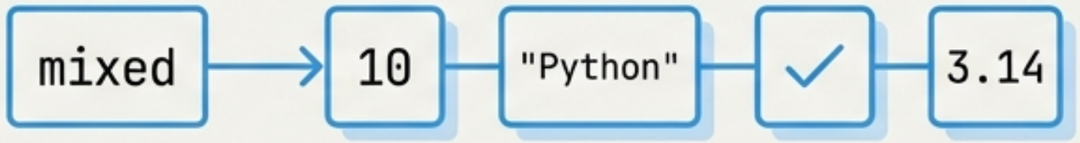
AT ITS CORE, A LIST IS AN ORDERED AND CHANGEABLE COLLECTION

- **Ordered**: Items maintain a specific sequence. Their position matters.
- **Mutable**: You can change, add, and remove items after the list has been created. It's a dynamic tool.
- **Versatile**: A single list can hold items of different data types.

```
nums = [10, 20, 30]
```

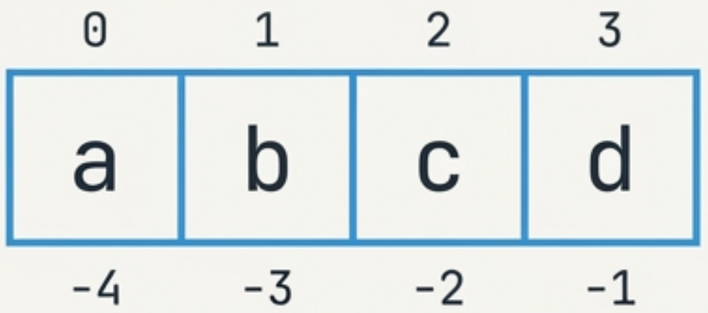



```
mixed = [10, "Python", True, 3.14]
```



INSPECTING YOUR DATA: ACCESSING ELEMENTS BY INDEX

Every item in a list has a position, or “index”, starting from 0. You can use this index to retrieve a specific item.



Negative Indexing (from the end)

A powerful shortcut for accessing items at the end of the list without needing to know its length.

Positive Indexing (from the start)

```
items = ["a", "b", "c", "d"]
items[0] ---------------- "a"
items[2] ------------------ "c"
```

```
items[-1]  "d"
items[-2]  "c"
```

PRECISION EXTRACTION: ACCESSING SUBSETS WITH SLICING

Slicing creates a new list containing a portion of the original. It's the primary tool for getting a "slice" of your data.

Syntax Breakdown

```
list[start:end:step]
```

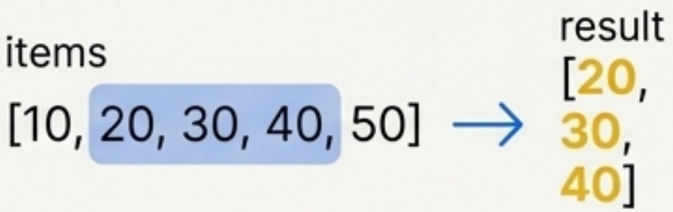
start: The beginning index (inclusive). Defaults to 0.

end: The ending index (exclusive). Defaults to list length.

step: The increment/decrement value. Defaults to 1.

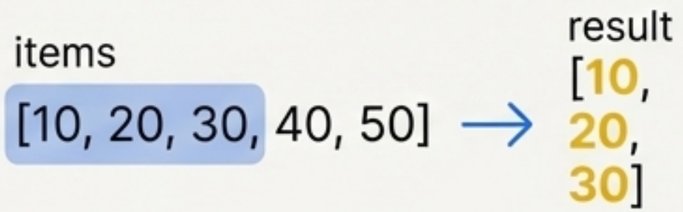
Get a middle chunk

```
items[1:4]
```



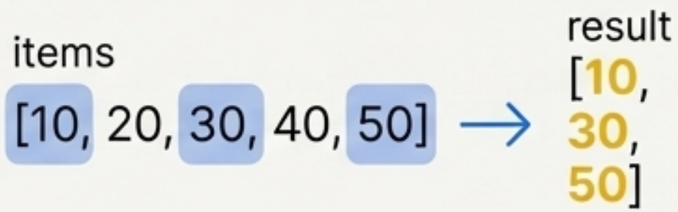
From the beginning

```
items[:3]
```



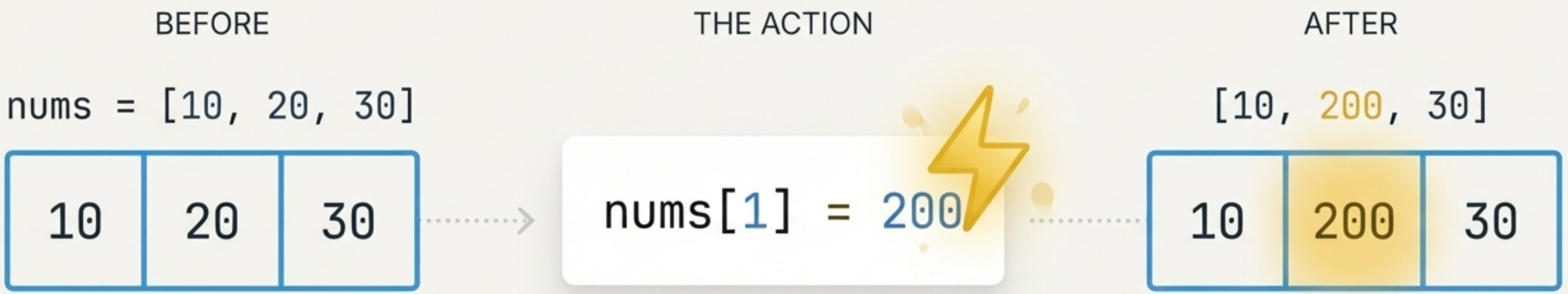
Every other element

```
items[::2]
```



SHAPING YOUR DATA: DIRECTLY UPDATING LIST ELEMENTS

Because lists are mutable, you can replace any element by assigning a new value to its index. This changes the list "in-place".

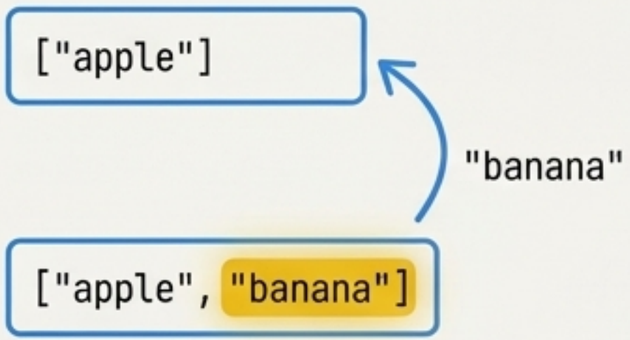


THE CONSTRUCTION TOOLS: THREE WAYS TO ADD ELEMENTS

1. `append()` - Add to the End

Use this to add a single item to the end of the list.

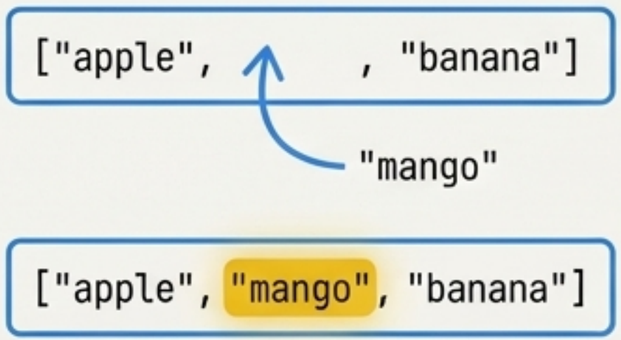
```
fruits = ["apple"]
fruits.append("banana")
```



2. `insert()` - Add at a Specific Position

Use this when the position of the new item matters.

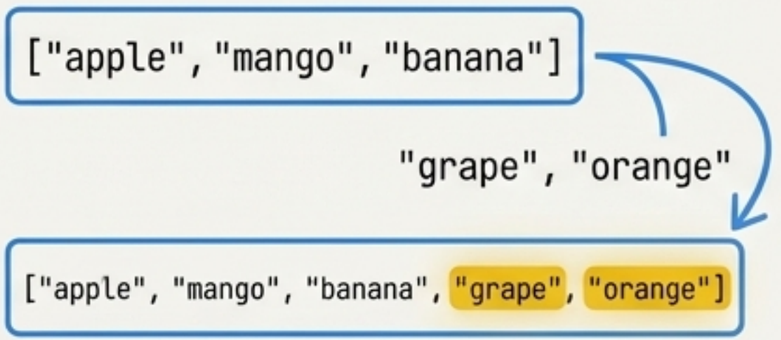
```
# assuming fruits is ["apple", "banana"]
fruits.insert(1, "mango")
```



3. `extend()` - Add Multiple Items

Use this to add all items from another collection to the end.

```
# assuming fruits is now ["apple", "mango",
fruits.extend(["grape", "orange"])
```

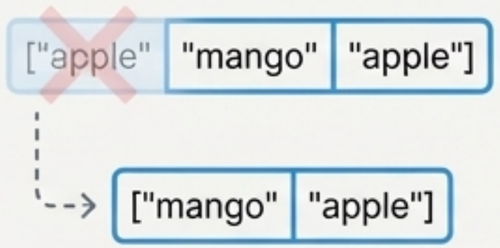


THE REFINEMENT TOOLS: FOUR WAYS TO REMOVE ELEMENTS

`remove(value)` - By Value

Removes the *first* occurrence of a specific value.

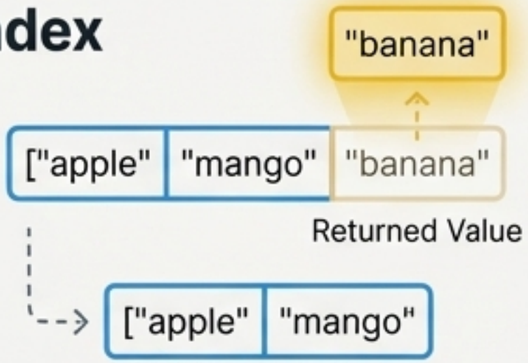
```
fruits.remove("apple")
```



`pop(index)` - By Index

Removes the item at the specified index (or the last) and returns it.

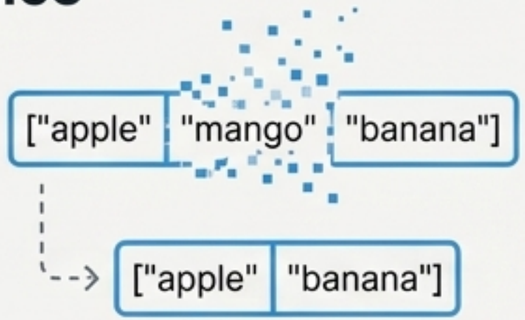
```
fruits.pop(2)
```



`del` - By Index or Slice

A powerful statement that removes items by index or even the entire variable.

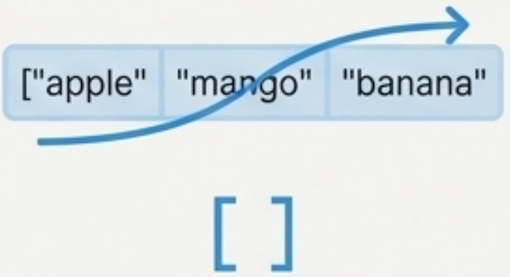
```
del fruits[1]
```



`clear()` - Empties the List

Removes all items, leaving an empty list `[]`.

```
fruits.clear()
```

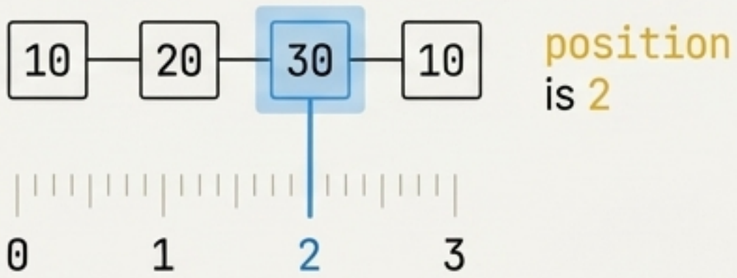


FINDING WHAT YOU NEED: SEARCHING AND MEMBERSHIP

Find Position with `.index()`

Returns the index of the first occurrence of an element. Raises an error if the element is not found.

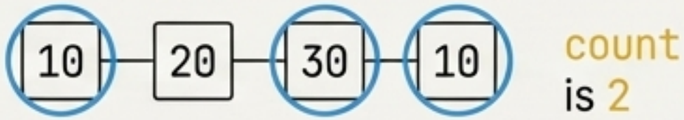
```
numbers = [10, 20, 30, 10]
position = numbers.index(30)
```



Count Occurrences with `.count()`

Returns how many times an element appears in the list.

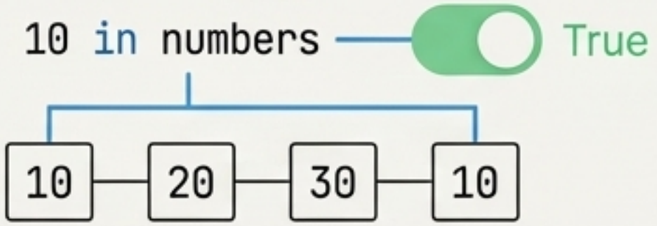
```
count = numbers.count(10)
```



Check Membership with `in`

The most Pythonic way to check if an item exists. Returns `True` or `False`.

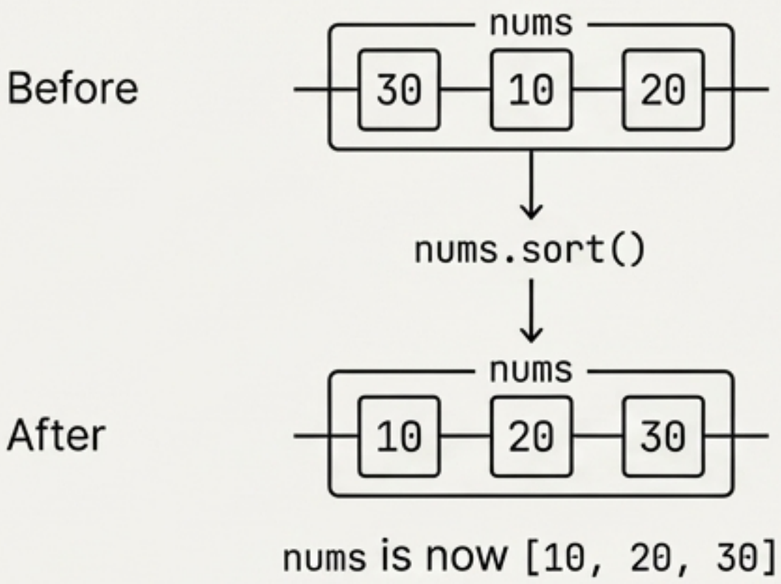
```
if 10 in numbers:
    print("It's in the list!")
```



ORGANISING YOUR COLLECTION: IN-PLACE VS. NEW LISTS

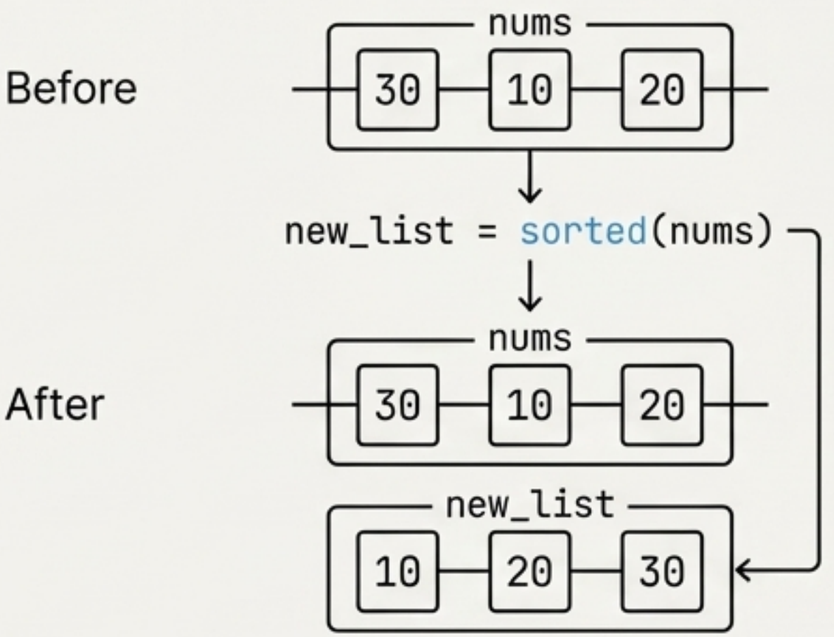
`list.sort()` - Modifies the Original

Sorts the list directly. The original list is changed, and nothing is returned.



`sorted(list)` - Creates a New List

Returns a *new*, sorted list, leaving the original unchanged.



**Pro Tip*: Use the `.reverse()` method to reverse the order of a list in-place.

EFFICIENT ITERATION: WALKING THROUGH YOUR LIST

The `for` loop is the standard way to perform an action on every item in a list.

Method 1: Simple Iteration (Accessing Values)

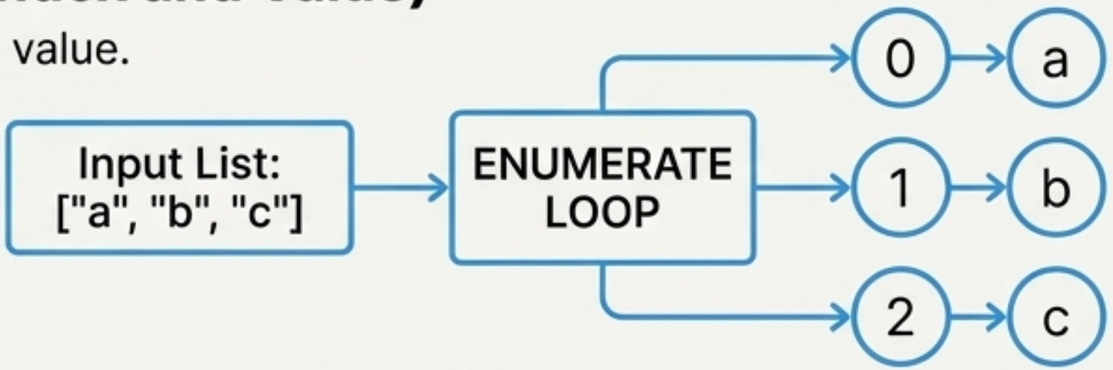
Use Case: When you only need each item's value.

```
items = ["a", "b", "c"]
for item in items:
    print(item)
```

Method 2: Iterating with Index (Accessing Index and Value)

Use Case: When you need both the item's position and its value.

```
for i, value in enumerate(items):
    print(f"Index {i}: {value}")
```



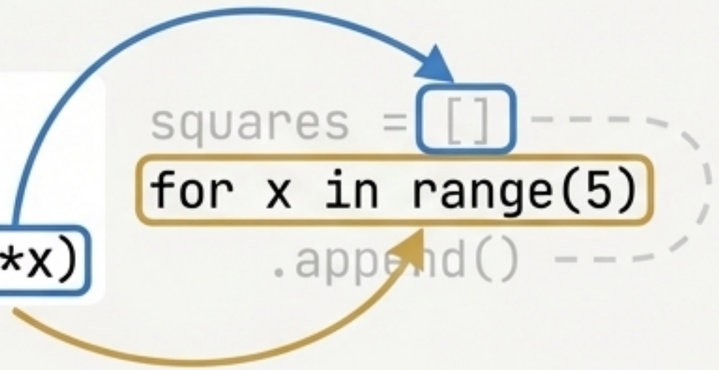
THE POWER TOOL: BUILDING LISTS WITH COMPREHENSIONS

A concise and readable way to create lists based on existing lists. It combines the `for` loop and the creation of elements into one line.

Example 1: Basic Creation | Goal: Create a list of squares.

The Old Way (for loop)

```
squares = []
for x in range(5):
    squares.append(x*x)
```



The Pythonic Way (comprehension)

```
squares = [x*x for x in range(5)]
```

```
[0, 1, 4, 9, 16]
```

Example 2: Creation with Filtering | Goal: Create a list of even numbers from a range.

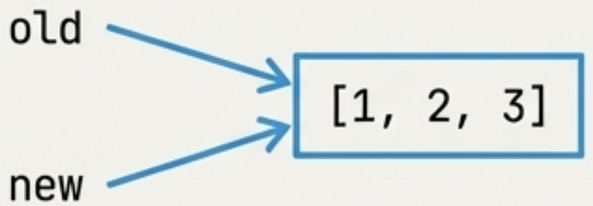
The Pythonic Way

```
even = [n for n in range(10) if n % 2 == 0]
```

```
[0, 2, 4, 6, 8]
```

A MASTER'S TASK: CREATING SAFE COPIES OF YOUR DATA

The Problem: Simple assignment (`new = old`) doesn't create a new list; it just creates another name pointing to the *same* list.

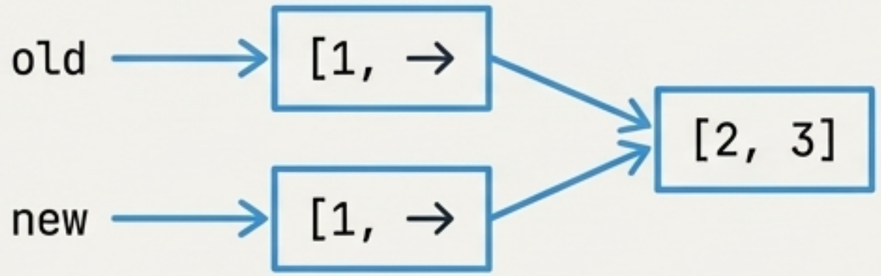


Solution 1: Shallow Copy

ℹ️ **When to use:** For simple lists (without nested lists).

```
new = old.copy() or new = old[:]
```

Creates a new list object, but if the original list contains other objects (like other lists), it copies only the references to them.

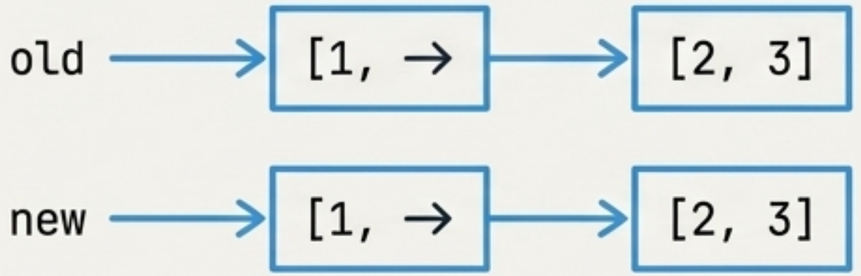


Solution 2: Deep Copy

ℹ️ **When to use:** Essential for lists containing other lists (nested lists).

```
from copy import deepcopy
new = deepcopy(old)
```

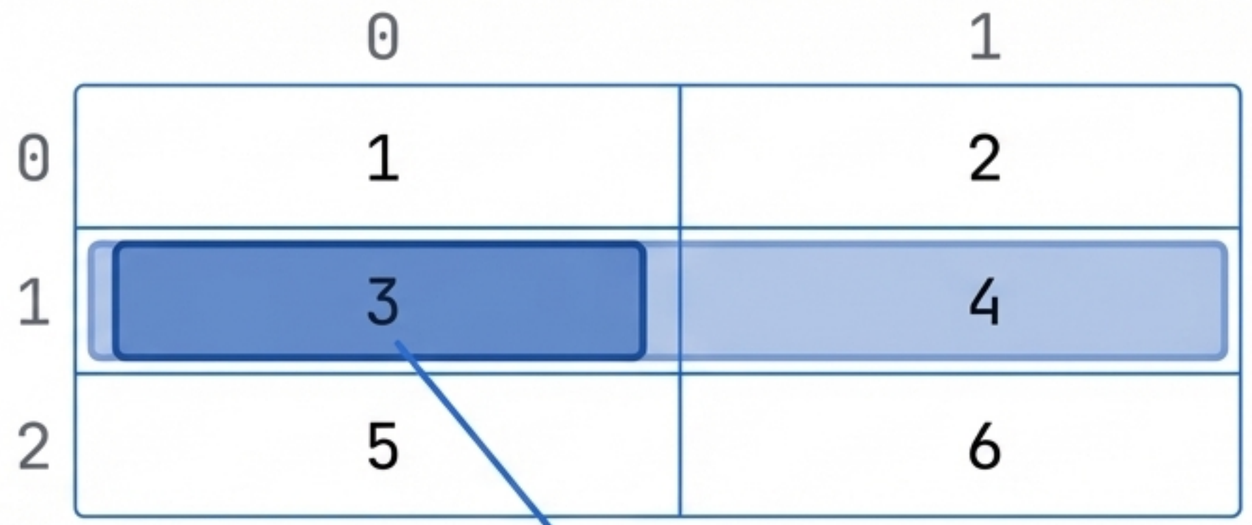
Creates a completely independent copy of the list and all the objects it contains, recursively.



HANDLING COMPLEXITY: WORKING WITH NESTED LISTS

A list can contain other lists as its elements. This is a common way to represent 2D data, like a grid, a matrix, or a tic-tac-toe board.

```
lists = [[1, 2],
         [3, 4],
         [5, 6]]
```



Accessing Nested Elements

Access works in layers: the first index [] selects the inner list, and the second index [] selects the element within that list.

```
element = lists[1][0]    element = "3"
```

```
element = "3"
```

QUICK REFERENCE: ESSENTIAL BUILT-IN FUNCTIONS

`len(list)`

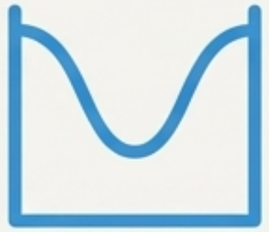
Gets the number of items in the list.

`len([10, 20, 30])` → `3`

`max(list)`

Returns the largest item in the list.

`max([10, 20, 30])` → `30`



`min(list)`

Returns the smallest item in the list.

`min([10, 20, 30])` → `10`

`sum(list)`

Calculates the sum of all numeric items in the list.

`sum([10, 20, 30])` → `60`

THE TOOLKIT IN ACTION: REAL-WORLD USE CASES

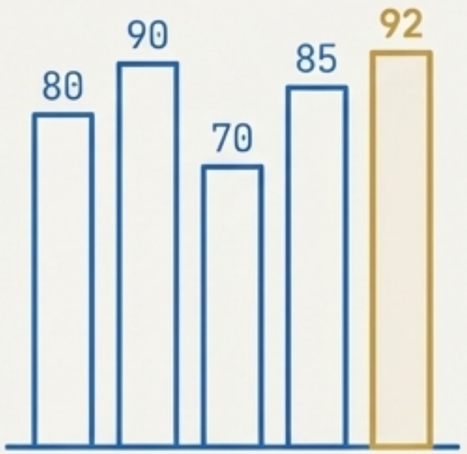
The Python list is not just a data structure; it’s a direct solution to common programming challenges.

STORING A COLLECTION OF DATA

Managing a class’s test scores.

```
marks = [80, 90, 70, 85, 92]
```

You can now easily calculate the average, find the highest score (`max(marks)`), or sort them.

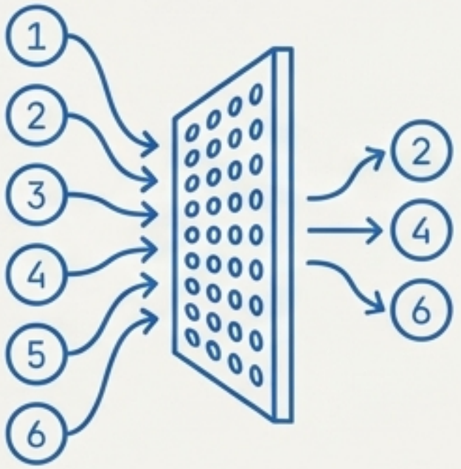


FILTERING DATA

Extracting all even numbers from a list of inputs.

```
nums = [1, 2, 3, 4, 5, 6]
evens = [n for n in nums if
n % 2 == 0]
```

This uses a list comprehension, the ‘power tool’, for an elegant and efficient solution.



Lists are Python's built-in implementation of dynamic arrays. Mastering them means you have a powerful, flexible tool ready for nearly any task involving a collection of data.