Nice — let's go deep into **NumPy**. I'll:

- fix & explain your snippet,
- then walk you through the important concepts (with compact examples),
- and finish with **checkpoint exercises/questions** so I can quiz you after this topic (like you asked).

☑ Quick fix for your code & why it failed

Your code:

```
import numpy as np

arr = np.array([10,20,30,40,50])

print(arr * 2)

print(arr.mean())

array2d = arr.reshape((2, 3))
print(array2d)
```

`arr * 2` and `arr.mean()` are fine.

`arr.reshape((2, 3))` fails because you have 5 elements and you're asking to reshape into 6 slots → **ValueError: cannot reshape array of size 5 into shape (2,3)**.

Fixes:

- reshape to a compatible shape, e.g. `(5,1)` or `(1,5)`:

```
array2d = arr.reshape((5, 1))   # shape (5,1)
```

- or create a 6-element array and reshape to (2,3):

```
arr2 = np.array([10,20,30,40,50,60])
array2d = arr2.reshape((2,3))
```

Deep NumPy guide (practical, with examples)

1) Arrays & dtypes

```
import numpy as np
a = np.array([1, 2, 3])          # dtype inferred (int)
b = np.array([1.0, 2.0, 3.0])    # float
print(a.dtype, b.dtype)          # e.g. int64, float64

# convert dtype
a_float = a.astype(np.float32)
```

Why dtype matters: memory/performance and math behaviour (int division vs float).

2) Array creation convenience functions

```
np.zeros((2,3))
np.ones(5)
np.empty((3,3))        # uninitialized contents (fast)
np.arange(0, 10, 2)    # like range but returns array
np.linspace(0,1,5)     # 5 evenly spaced values in [0,1]
np.eye(4)              # identity matrix
```

3) Shape, size, memory info

```
arr = np.arange(12).reshape(3,4)
arr.shape      # (3,4)
arr.ndim       # 2
arr.size       # 12 (total elements)
arr.itemsize   # bytes per element
arr.nbytes     # total bytes = size * itemsize
arr.strides    # how many bytes to step to move along each axis
```

```
arr.flags      # tells if C-contiguous / F-contiguous etc
```

`strides` + `shape` determine how indexing reads memory (important for performance).

4) Views vs copies

- `view` = new array object pointing at same memory (fast, in-place changes reflected).
- `copy` = new memory (safe, more memory/time).

Examples:

```
x = np.arange(6)
y = x.reshape((3,2))      # often a view
z = x.copy()              # separate memory

y[0,0] = 999
print(x)                  # x changed if y is a view

flat = x.ravel()          # returns view when possible
flat2 = x.flatten()       # returns a copy
```

Use `np.shares_memory(x, y)` to test.

5) Indexing & slicing (powerful)

```
a = np.arange(10)
a[2]        # single element
a[2:8:2]    # slicing (start:stop:step)
a[-1]       # last element

M = np.arange(12).reshape(3,4)
M[1,2]      # row 1, col 2
M[:, 1]     # column 1
M[1, :]     # row 1

# boolean mask
```

```
mask = M > 5
M[mask]     # returns 1D array of elements > 5

# fancy indexing (copies)
idx = [0,2]
M[idx]      # select rows 0 and 2
```

Important: boolean indexing returns a new 1D array (not a view). Fancy indexing returns copies.

6) Broadcasting (one of NumPy's superpowers)

Rules: two arrays are compatible if for each trailing dimension they are equal or one of them is 1. NumPy virtually stretches size-1 dimensions.

Examples:

```
a = np.array([1,2,3])           # shape (3,)
b = np.array([[10],[20]])      # shape (2,1)
a + b  # result shape (2,3): adds row-wise
```

Common use: `X - X.mean(axis=0)` to center columns (works because mean has shape (n_features,) and broadcasts to (n_samples, n_features)).

7) Universal functions (ufuncs) & vectorized ops

Fast elementwise functions:

```
np.add(a, 5)
np.sin(a)
np.exp(a)
np.log(a)
```

Use `np.sqrt`, `np.maximum`, etc. These are implemented in C — much faster than Python loops.

Also: `np.vectorize` exists but is a convenience wrapper — still Python-level, so not as fast.

8) Reductions & axis semantics

```
A = np.arange(12).reshape(3,4)
A.sum()            # scalar
A.sum(axis=0)      # sum columns -> shape (4,)
A.sum(axis=1)      # sum rows -> shape (3,)
A.mean(axis=0, keepdims=True)  # keepdims preserves dimensions
```

Remember: `axis=0` collapses rows (operates down columns), `axis=1` collapses columns (operates across rows).

9) Linear algebra

```
A = np.array([[1,2],[3,4]])
B = np.array([[5,6],[7,8]])
C = A @ B                # matrix multiply (same as np.dot for 2D)
v = np.array([1,2])
A.dot(v)                 # matrix-vector

# solves and decompositions
np.linalg.inv(A)
np.linalg.solve(A, b)    # better than inv(A).dot(b)
np.linalg.svd(A)
np.linalg.eig(A)
```

For large-scale ML use `scipy.linalg` or specialized libraries (MKL-backed NumPy is fast).

10) Random numbers (new recommended API)

```
rng = np.random.default_rng(42)
rng.integers(0, 10, size=(3,3))
```

```
rng.normal(loc=0.0, scale=1.0, size=1000)
rng.permutation(10)
```

Use a `Generator` for reproducible, thread-safe RNG.

11) Useful helpers & advanced tools

- `np.where(condition, x, y)` — choose elements depending on condition.
- `np.unique`, `np.argsort`, `np.argmax`, `np.argmin`, `np.bincount`.
- `np.save`, `np.savez`, `np.load` — fast array persistence.
- `np.memmap` — memory-map large arrays stored on disk.
- `np.einsum` — very flexible tensor contraction / sums (powerful and concise). Example: matrix multiply `np.einsum('ik,kj->ij', A, B)`.
- `np.ma` (masked arrays) for missing-data-aware computations.
- `np.lib.stride_tricks.as_strided` — powerful but dangerous (advanced).

12) Interoperability with Pandas & ML

- `df = pd.DataFrame(arr)` and `arr = df.to_numpy()` / `df.values`.
- For ML pipelines: data normalization with NumPy is fast:

```
X = X - X.mean(axis=0)
X = X / (X.std(axis=0) + 1e-8)
```

- One-hot encoding: `np.eye(num_classes)[labels]`.

13) Performance tips

- Avoid Python loops; prefer vectorized ufuncs.
- Use in-place ops when you want to save memory: `arr *= 2` modifies in place.
- Check contiguity (`arr.flags['C_CONTIGUOUS']`) — many BLAS routines expect C-contiguous arrays. Use `np.ascontiguousarray` if needed.
- For heavy numerical kernels consider libraries: Numba, Cython, or use BLAS-backed NumPy.