Geo-Data and Geo-Data Management

Lecture 4: Raster Data — Grids, Values and Georeferencing

Yingjing Huang

yingjing.huang@univie.ac.at

2026W

Today’s plan

1



Lecture

- Python recap: nested structures, functions & chains
- Understand raster values and valid cells
- Georeference, save and map a GeoTIFF
- Apply the workflow to real DEM and land cover

2



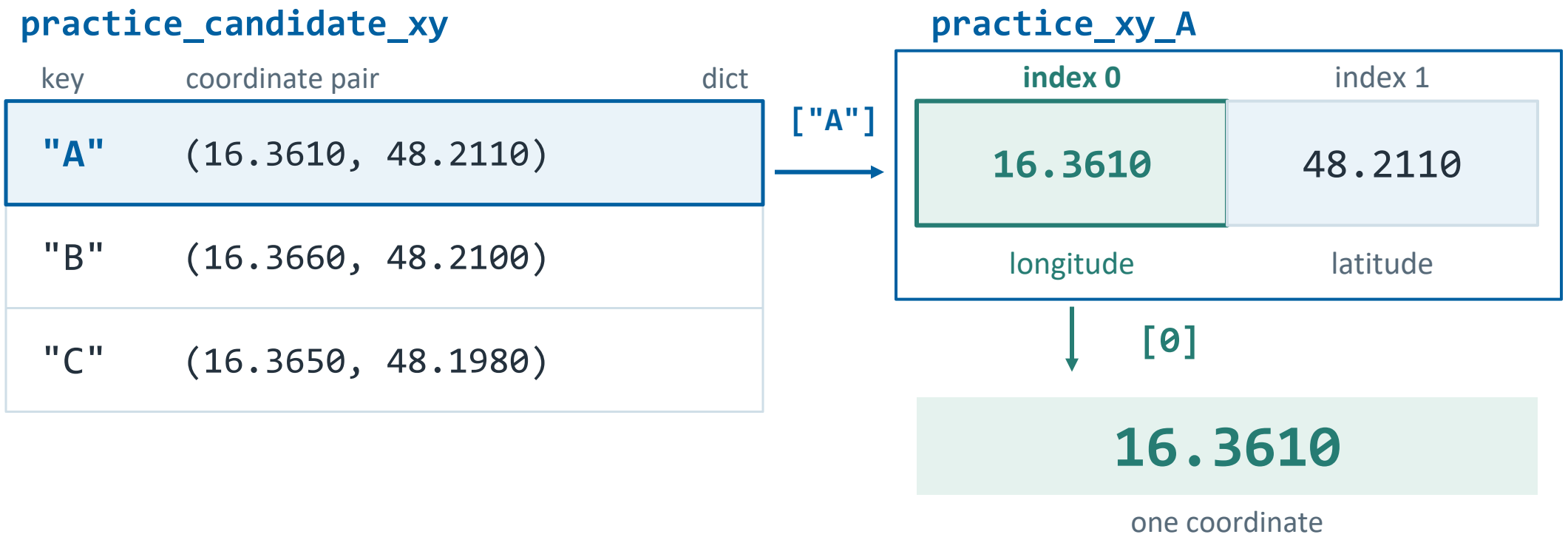
Lab

- Guided A: build & verify a synthetic raster (§§1–6)
- Guided B: analyse the Vienna DEM (§§7–9)
- Independent C: analyse WorldCover (§§10–11)

Python Recap: Nested Data Structures

A dictionary key retrieves a tuple; a tuple index retrieves one coordinate.

```
practice_xy_A_lon = practice_candidate_xy["A"][0]
print(practice_xy_A_lon)
```



Python Recap: Nested Functions

The inner function creates geometries; the outer function builds the GeoDataFrame.

WEEK 3 NOTEBOOK

```
fountains = gpd.GeoDataFrame(
    fountain_table.copy(),
    geometry=gpd.points_from_xy(
        fountain_table["longitude"],
        fountain_table["latitude"]
    ),
    crs="EPSG:4326"
)
```

1 **Create the Point geometries**

```
lon = fountain_table["longitude"]
lat = fountain_table["latitude"]
geometry = gpd.points_from_xy(
    lon, lat)
```

One Point per table row

2 **Build the GeoDataFrame**

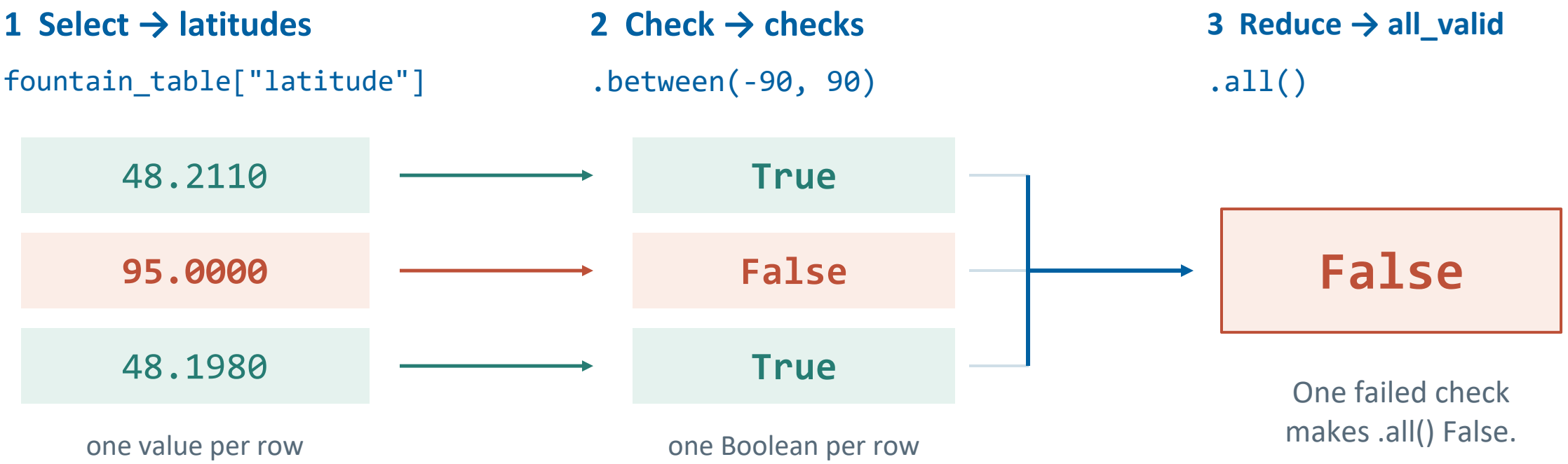
```
fountains = gpd.GeoDataFrame(
    fountain_table.copy(),
    geometry=geometry,
    crs="EPSG:4326")
```

Attributes + geometry + CRS

Python Recap: Chained Methods

A method chain: select values → check each row → summarise all results.

```
print("Valid latitude:", fountain_table["latitude"].between(-90, 90).all())
```



Python Recap: Unpacking Multi-Step Code

When several steps are packed into one expression, unpack them and inspect the intermediates.

1. **Split**
   Name each intermediate result
2. **Run**
   Execute one operation at a time
3. **Inspect**
   Check its type or first few rows

GROUPBY EXAMPLE

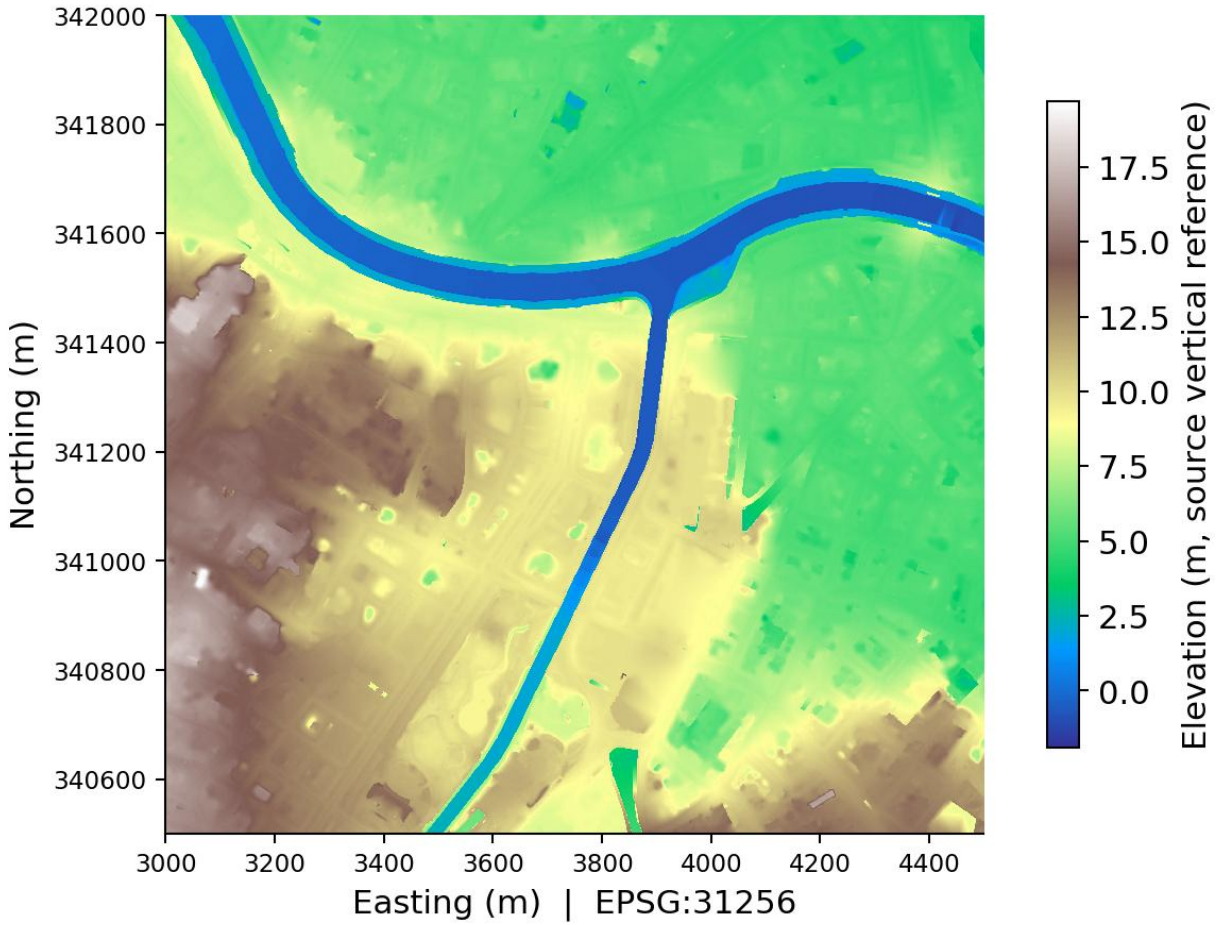
```
groups = matched.groupby("district_id")
counts = groups.size()

# Inspect before continuing
print(type(counts))
print(counts.head())

fountain_counts = counts.reset_index(
    name="fountain_count")
```

GroupBy groups rows. The counts are a Series. reset_index creates a DataFrame.

Raster Data



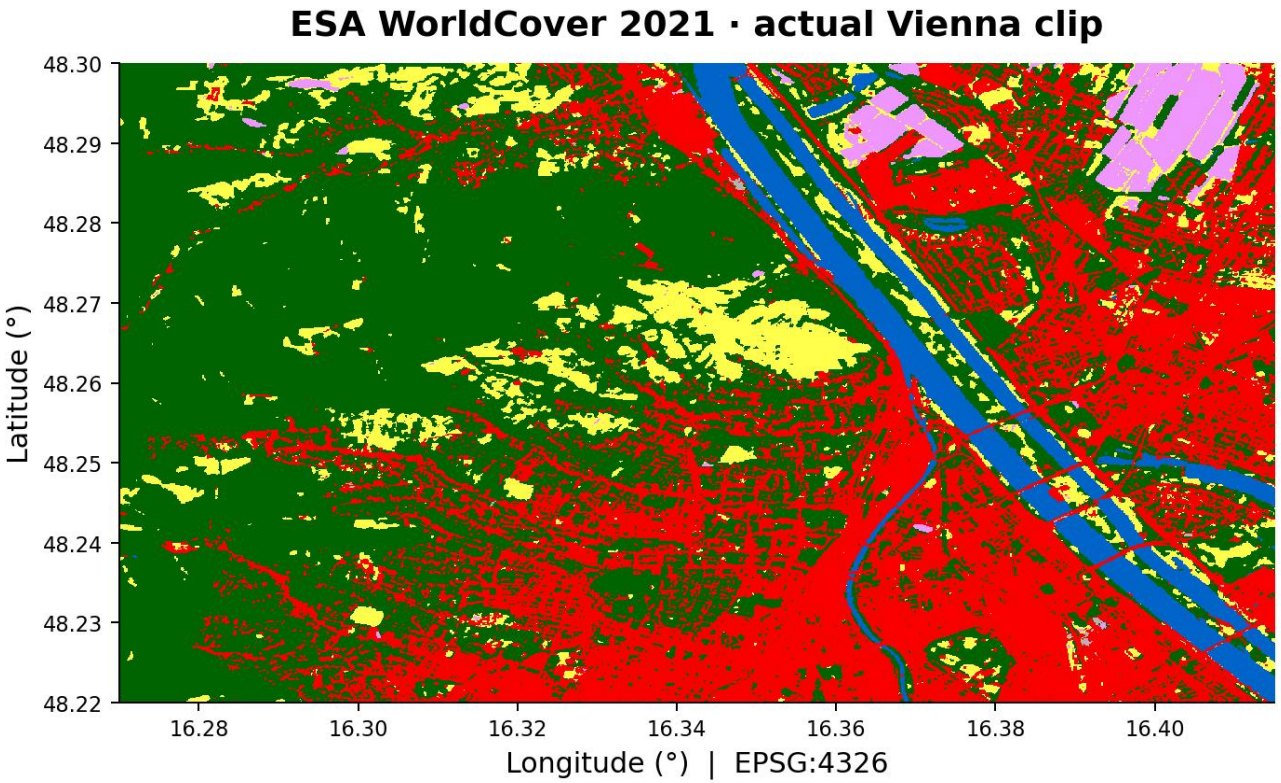
SIMPLIFIED ELEVATION GRID (m)

210	212	218
208	**215**	225
205	213	240

Each cell stores one elevation value.
Example: 215 m

A Land-Cover Raster Stores Category Codes

Each code represents a land-cover class.



SIMPLIFIED LAND-COVER GRID

10	**50**	50
10	30	50
30	80	80

- 10 Tree cover
- 30 Grassland
- 40 Cropland
- 50 Built-up
- 60 Bare/sparse
- 80 Water
- 90 Wetland

Sentinel-2 Provides Multiple Raster Bands

B4 and B8 record different spectral reflectance.

B4 · Red

0.12	0.11	0.10
0.13	**0.12**	0.11
0.15	0.14	0.13

0.12

Red reflectance

Same cell location.
Different measurements.

B8 · Near infrared

0.45	0.40	0.44
0.47	**0.46**	0.42
0.49	0.45	0.46

0.46

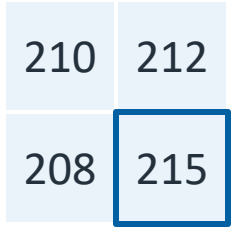
Near-infrared reflectance

Raster Values Have Different Meanings

Measurements carry units. Class codes identify categories.

Elevation

Measurement

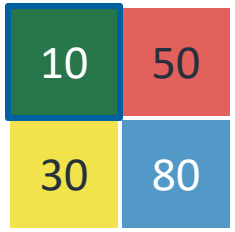


215 m

Elevation is a measured height.

Land cover

Class code

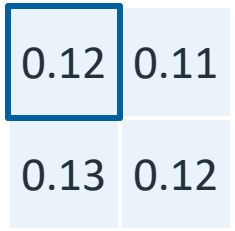


10 = tree cover

The code names a category.

Red band

Measurement

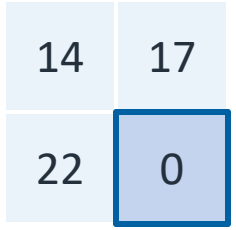


0.12 reflectance

A measurement in the red band.

Temperature

Measurement

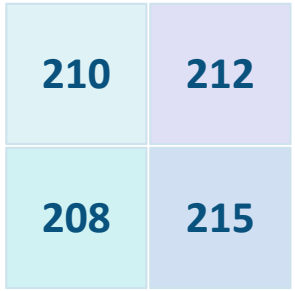


0 °C

Zero can be a valid temperature.

Python Tools for Raster Data

NumPy (np)



Here: values, masks & operations

Scientific computing with N-dimensional arrays.

Docs: numpy.org/doc/stable

Rasterio

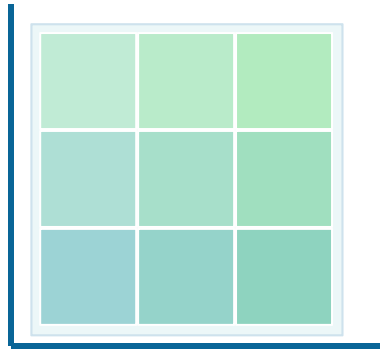


Here: GeoTIFF, CRS, transform & NoData

Python tools for reading, writing and georeferencing raster data.

Docs: rasterio.readthedocs.io

Matplotlib (plt)



Here: map-coordinate plotting & colourbars

A general-purpose library for creating data visualizations.

Docs: matplotlib.org/stable

One Synthetic DEM

A small digital elevation model (DEM)

210	212	218	-9999
208	215	225	229
205	213	240	235
200	208	219	230

CREATE THE DEM

```
arr = np.array([
    [210, 212, 218, -9999],
    [208, 215, 225, 229],
    [205, 213, 240, 235],
    [200, 208, 219, 230]
], dtype=np.float32)
nodata = -9999.0
```

Synthetic elevation values (m) · NoData = −9999 · all 16 cells shown

Array Shape and Data Type

shape gives grid dimensions. dtype describes stored values.

4 columns

4 rows

210	212	218	—
208	215	225	229
205	213	240	235
200	208	219	230

IN CODE

```
print(arr.shape)   # (4, 4)
print(arr.dtype)   # float32
```

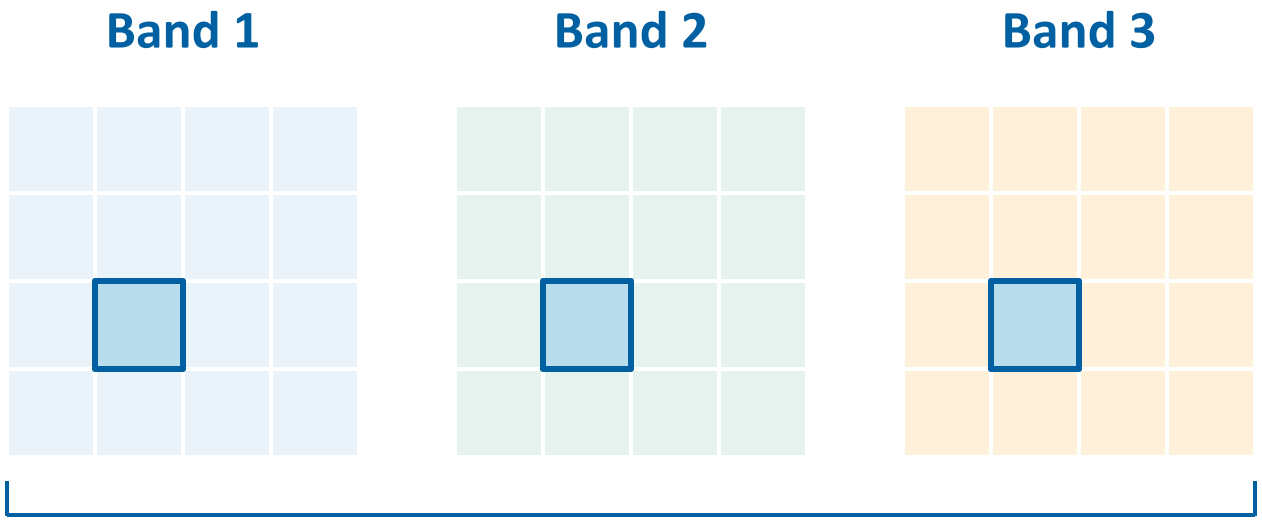
arr.shape
(4, 4)

arr.dtype
float32

32-bit floating-point values

Bands Add Layers of Values to a Grid

A band stack is indexed as [band, row, col].



IN CODE

```
band1 = stack[0]   # Band 1 grid
stack[0, 2, 1]   # Band 1 cell
stack[:, 2, 1]   # same cell, all bands
```

Same rows and columns
Different measurements

Array shape

(3 , 4 , 4)

bands rows columns

A Valid Python Result Can Still Be Misleading

Raw calculations can include invalid numbers.

STORED ARRAY

210	212	218	-9999
208	215	225	229
205	213	240	235
200	208	219	230

–9999 = NoData marker

PYTHON COMPARES THE NUMBERS

```
arr.max()  →  240
```

largest stored value

```
arr.min()  →  -9999
```

The marker is included in the calculation.

This is a missing-data marker,
not a measured elevation.

A Validity Mask Selects Legitimate Cells

A mask identifies which cells are valid.

Values

210	212	218	-9999
208	215	225	229
205	213	240	235
200	208	219	230

16 cells

```
nodata = -9999.0

valid = arr != nodata
```

IN CODE

```
valid = arr != nodata
print(valid[0, 3])   # False
print(valid[2, 1])   # True
print(valid.sum())     # 15
```

Validity mask

True	True	True	False
True	True	True	True
True	True	True	True
True	True	True	True

int(valid.sum()) → 15

15 True · 1 False

Here, NoData = −9999. A legitimate zero remains valid.

Statistics Must Exclude Invalid Cells

Statistics should use valid cells only.

210	212	218	-9999
208	215	225	229
205	213	240	235
200	208	219	230

BEFORE

```
arr.min() = -9999
```

AFTER MASKING

```
arr[valid].min() = 200 m
```

arr[valid] → 1D array of 15 valid values

Raster Cells Can Have Different Ground Sizes

Equal shapes can cover different ground areas.

1	2	3
4	5	6
7	8	9

10 m × 10 m

extent: 30 m × 30 m

VS

1	2	3
4	5	6
7	8	9

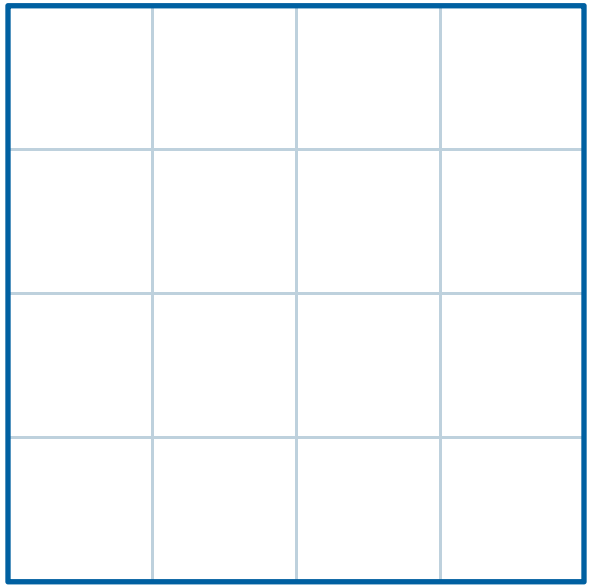
100 m × 100 m

extent: 300 m × 300 m

Extent and Cell Size Determine Grid Dimensions

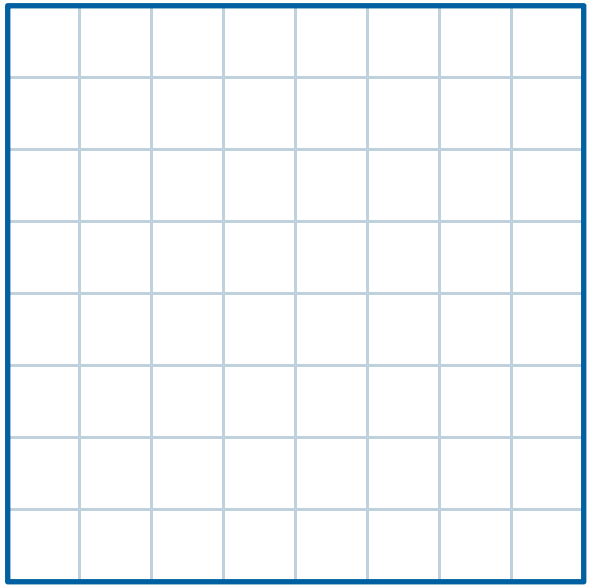
A smaller cell size requires more cells over the same extent.

SAME EXTENT: 400 m × 400 m



100 m per cell

4 × 4 = 16 cells



50 m per cell

8 × 8 = 64 cells

Cell Size Affects Spatial Detail

Finer cells can reveal finer spatial patterns.

Same ground extent

Coarse grid · 4 × 4

10	10	50	50
10	30	50	50
10	30	30	80
30	30	80	80

One label: 30 = grassland
Internal variation is not shown.

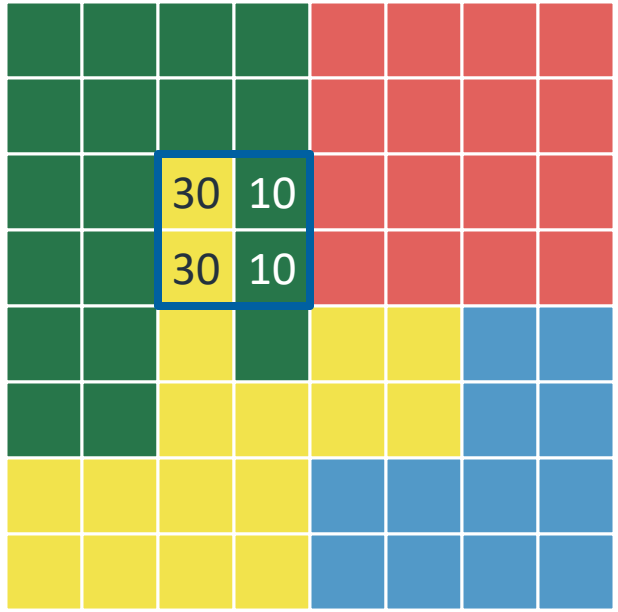
Same outlined area

One coarse cell

covers the area of

four finer cells

Finer grid · 8 × 8

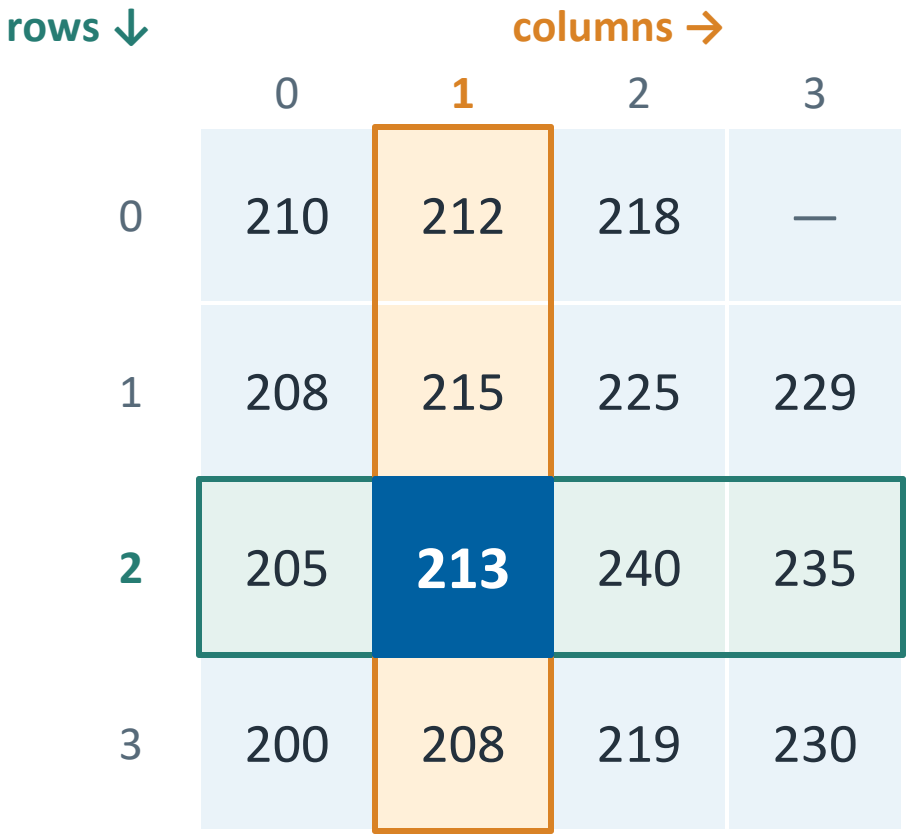


Finer cells distinguish
grassland (30) and trees (10).

10 Tree cover 30 Grassland 50 Built-up 80 Water

Row and Column Indexing

arr[row, col] returns the value at one grid cell.



1 Find row 2

2 Find column 1

```
arr[2, 1]
```

row column

→ 213.0

IN CODE

```
row, col = 2, 1
value = arr[row, col]
print(value)   # 213.0
```

Indices start at 0. Write the row first.

Finer Cells Do Not Create New Information

Smaller cells cannot recover detail that was never observed.

Same ground extent

Original grid · 4 × 4

10	10	50	50
10	30	50	50
10	30	30	80
30	30	80	80

One observed label: 30 = grassland

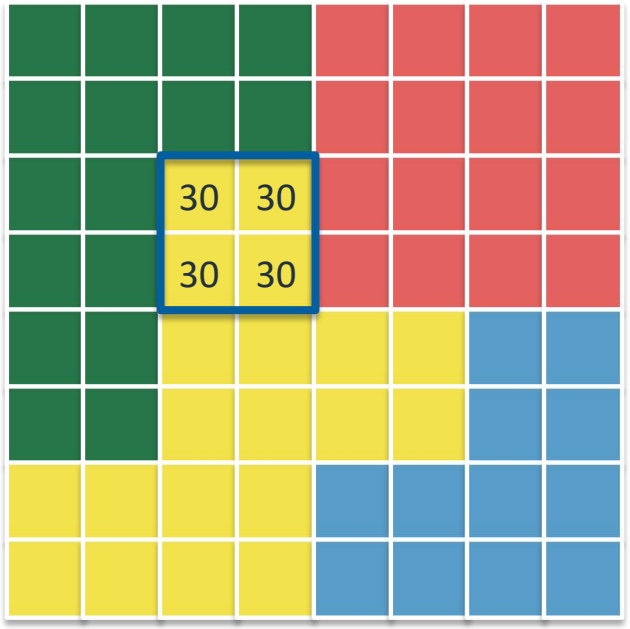
Divide each cell into four

4× more cells

0 new observations

More cells ≠ more information

Same values · 8 × 8

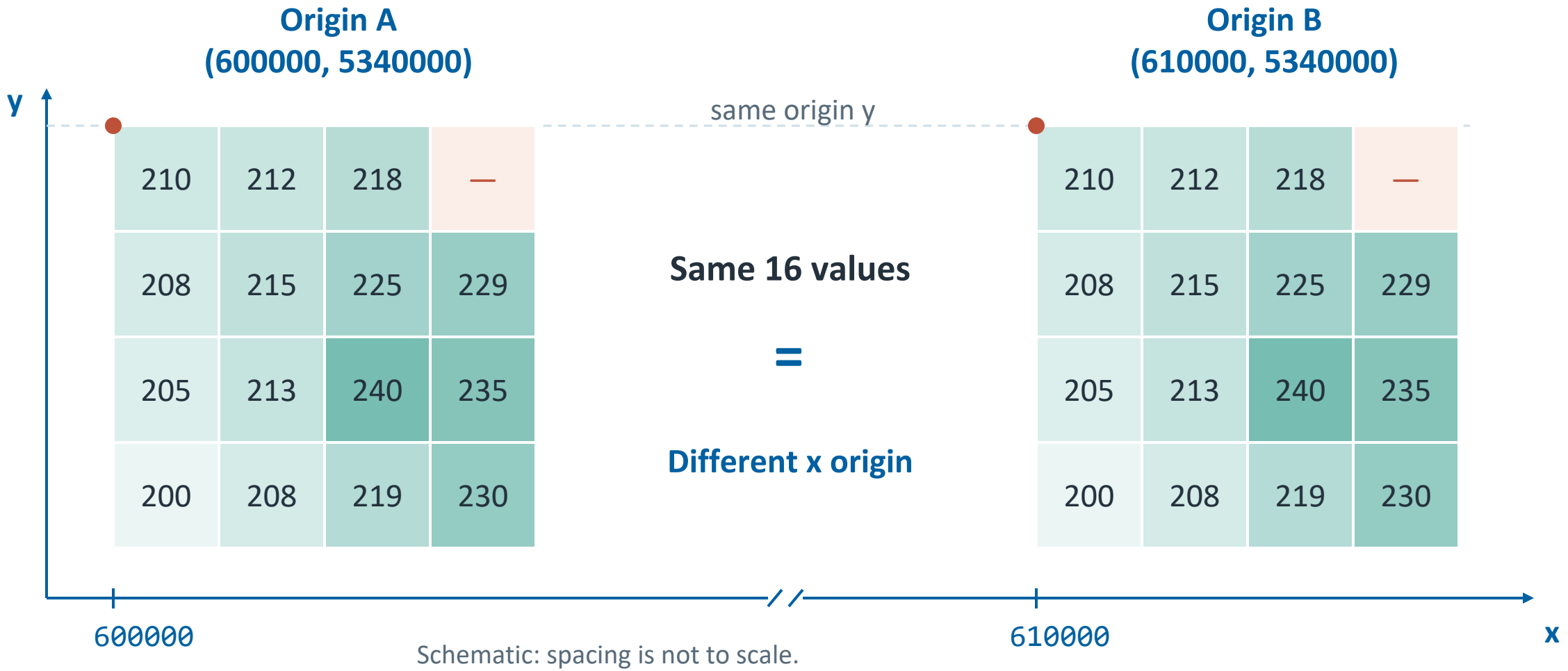


The same 30 is copied into four smaller cells.

Finer resolution only helps when the source data support finer detail.

Numbers Alone Do Not Make a Georeferenced Raster

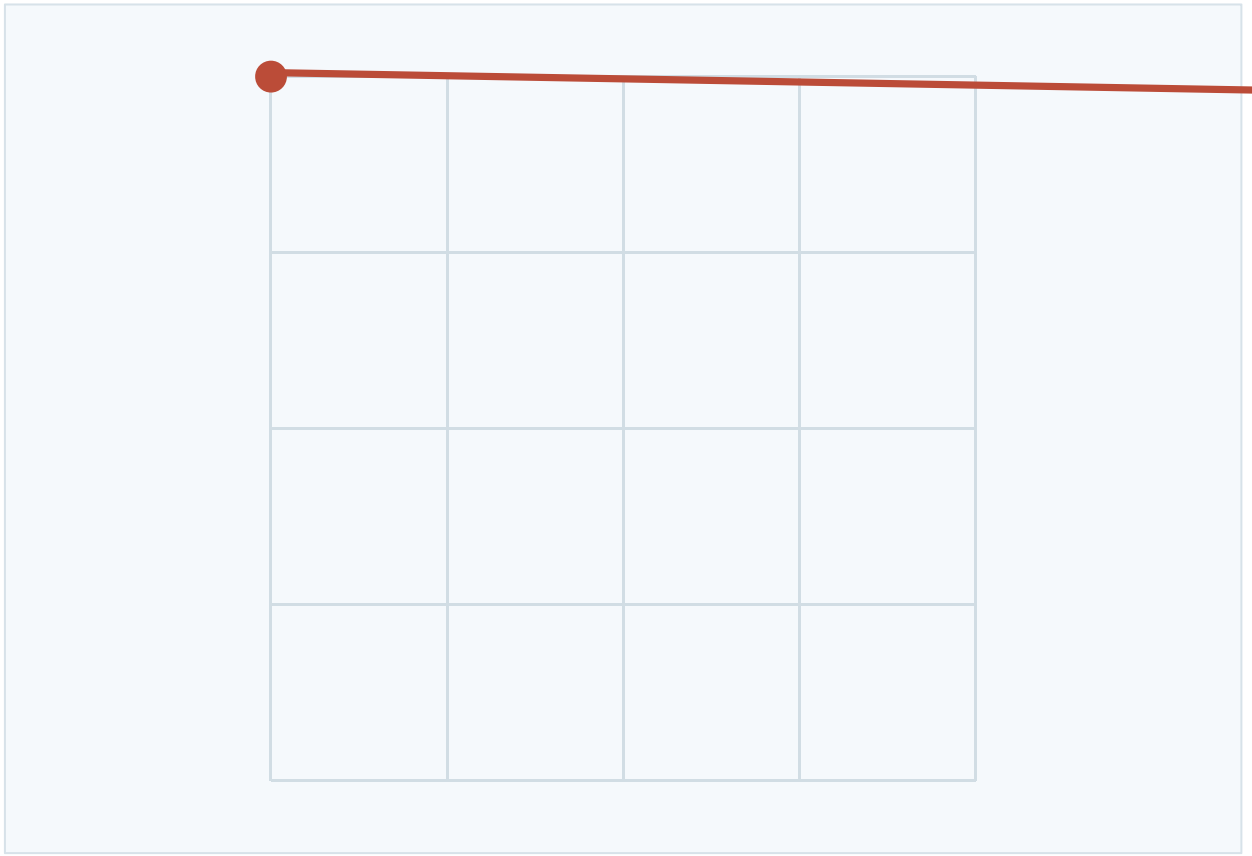
Values alone cannot locate a raster on Earth.



Schematic: spacing is not to scale.

The Origin Anchors the Grid in Coordinate Space

The upper-left grid corner anchors the raster.

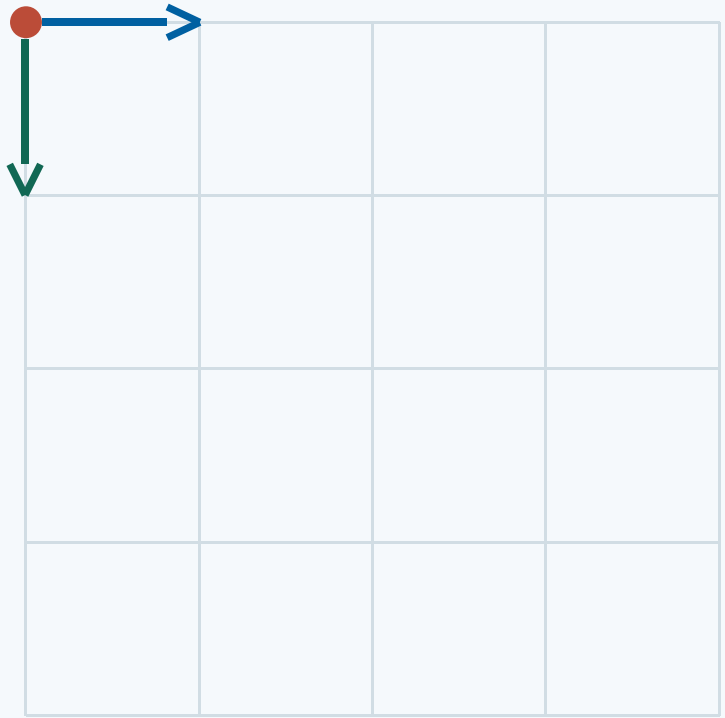


Origin
(600000, 5340000)

Upper-left grid corner

Pixel Size Determines Coordinate Steps

Cell size determines coordinate steps.



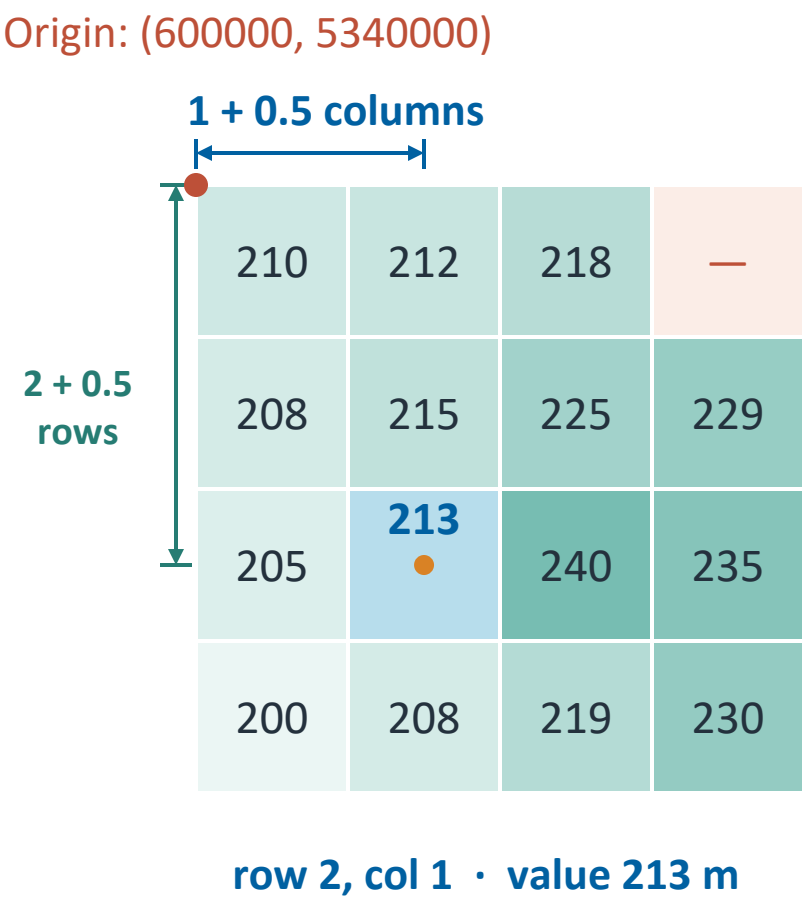
one column right
x increases by 100 m

one row down
y decreases by 100 m

For this north-up, unrotated grid

Calculate the Centre of One Cell

The half-cell offset turns an index into a cell centre.




```
row, col = 2, 1
```

```
x_manual = 600000 + (col + 0.5) * 100
```

column index + half a cell

```
y_manual = 5340000 - (row + 0.5) * 100
```

row index + half a cell

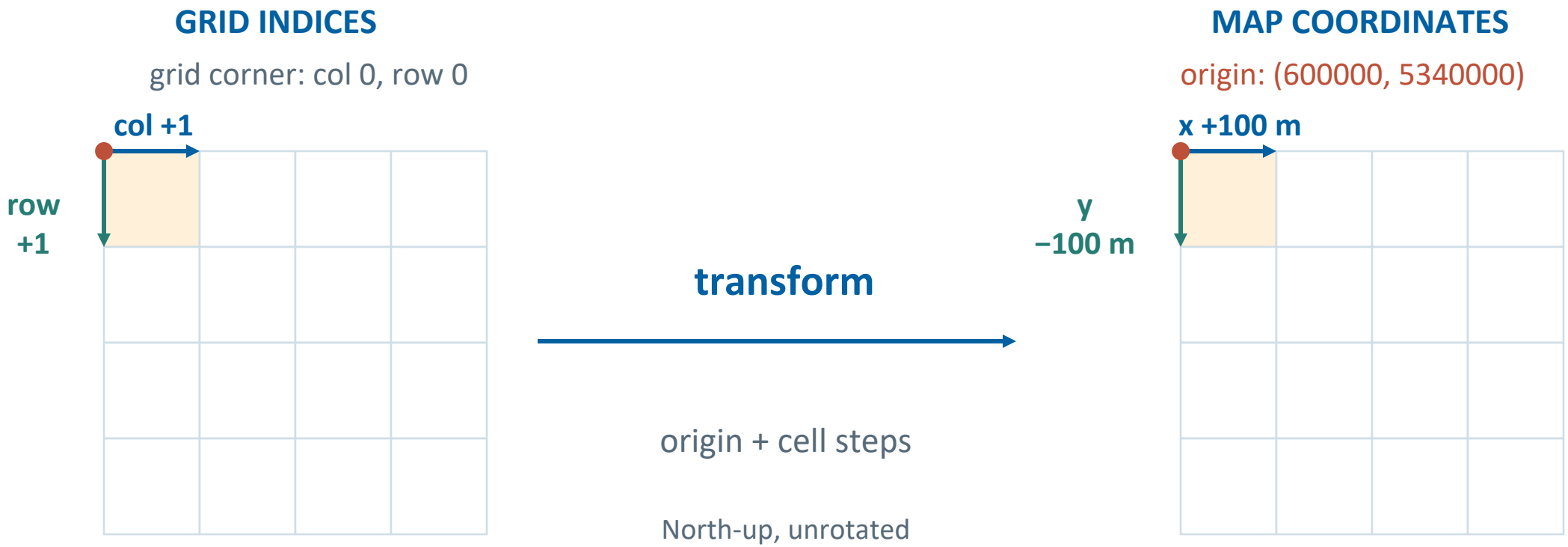
(600150, 5339750) m

- +0.5 locates the cell centre.

100 m cells · north-up, unrotated grid

An Affine Transform Maps Cells to Coordinates

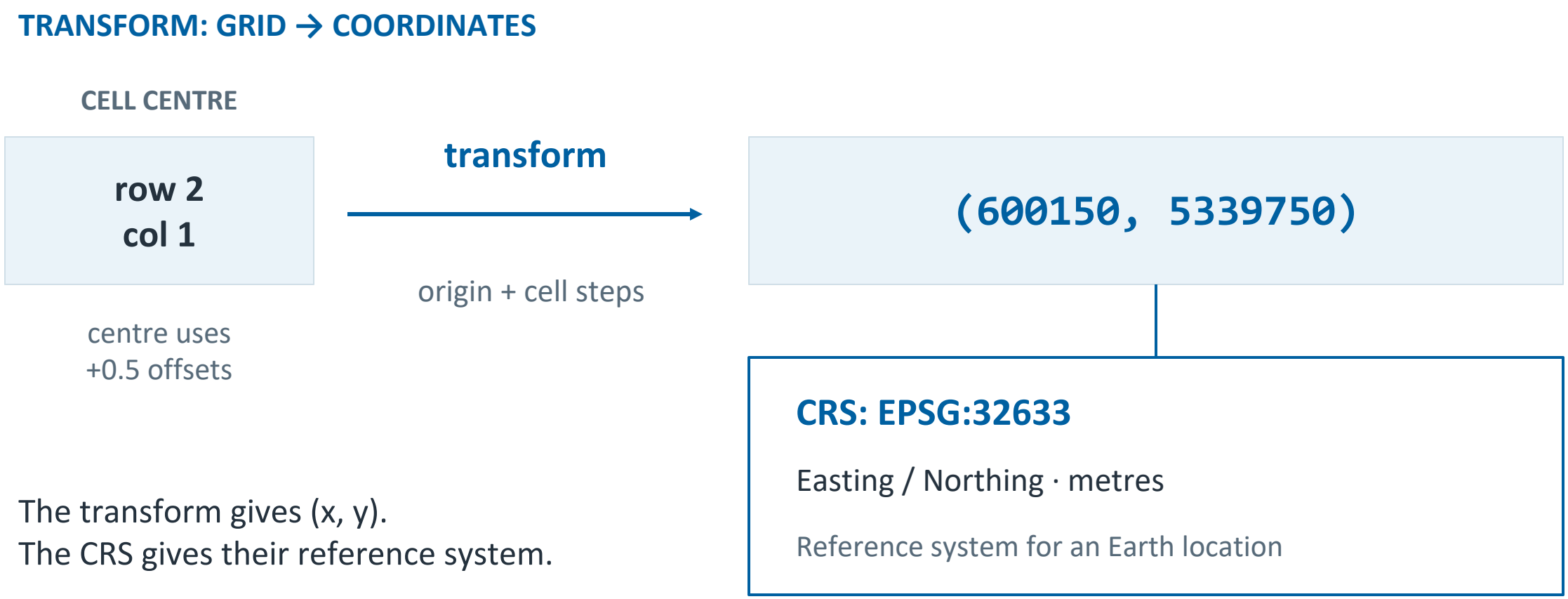
The CRS and transform give the grid its spatial reference.




```
crs = "EPSG:32633"
transform = from_origin(600000, 5340000, 100, 100)
```

Transform and CRS Answer Different Questions

Transform and CRS answer different questions.



The transform gives (x, y).
The CRS gives their reference system.

Projected Coordinates Are Not Longitude/Latitude

Projected coordinates are not latitude/longitude.

PROJECTED · EPSG:32633

(600150, 5339750)

Easting / Northing · metres

GEOGRAPHIC · EPSG:4326

(16.3479°, 48.2031°)

Longitude / Latitude · degrees

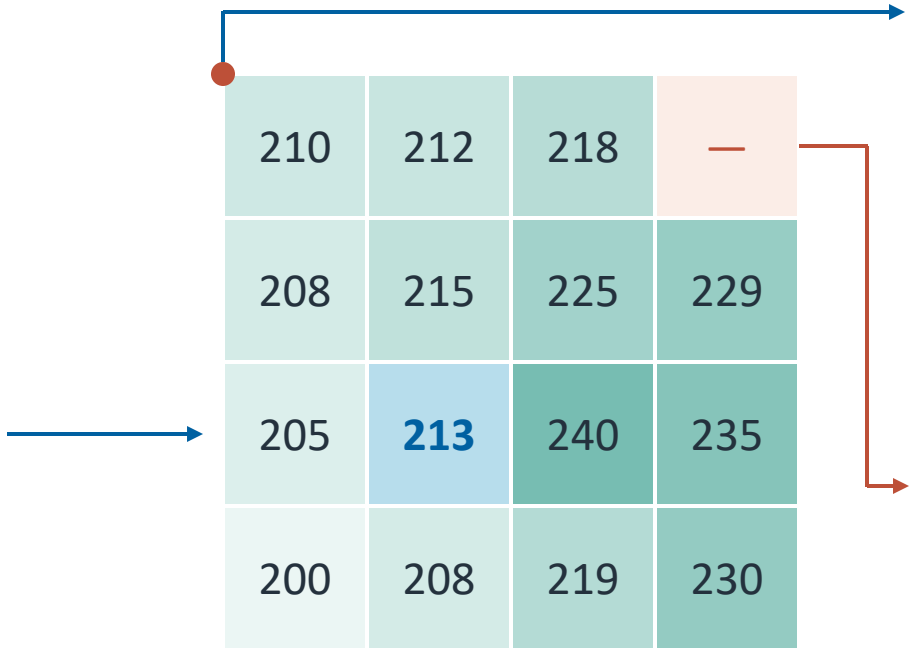
A coordinate pair needs its CRS and units to be meaningful.

Raster Needs Values, Placement and Meaning

A usable raster needs values, placement and meaning.

VALUES

4 × 4 elevation grid
One value per cell



PLACEMENT

origin (600000, 5340000)
100 m cell size
CRS: EPSG:32633

MEANING

Elevation in metres
NoData marker: **–9999**
Height reference: zero level

GeoTIFF: A Common Format for Georeferenced Rasters

A GeoTIFF (.tif) can store raster values and georeferencing information.



Writing a Raster Requires Explicit Metadata

Writing GeoTIFF requires explicit metadata.

arr

width = 4 columns

4 rows

210	212	218	—
208	215	225	229
205	213	240	235
200	208	219	230

```
height = 4 rows
dtype = float32
```

driver = "GTiff"

ARRAY STRUCTURE

```
width: 4   height: 4   count: 1
dtype: float32
```

REFERENCE & VALIDITY

```
crs: EPSG:32633
transform: from_origin(
    600000, 5340000, 100, 100)

nodata: -9999
```

Parameters supplied to rasterio.open()

Writing Within a Context Manager

The dataset stays open while its indented block runs.

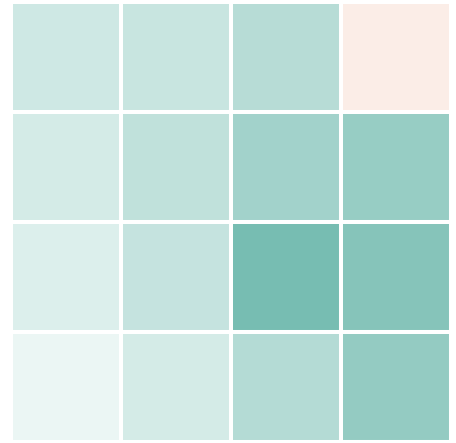
Writing elevation_demo.tif

```
with rasterio.open(
    tif_path, "w", ...) as dst:

    dst.write(arr, 1)
```

arr

NumPy array to write



dst closes when the block ends

arr remains available

Rasterio Can Write the Array into a GeoTIFF

Rasterio writes values and placement into a file.

WRITE GEOTIFF

```
tif_path = workdir / "elevation_demo.tif"
with rasterio.open(
    tif_path, "w",
    driver="GTiff", height=arr.shape[0],
    width=arr.shape[1], count=1,
    dtype=arr.dtype, crs=crs,
    transform=transform, nodata=nodata
) as dst:
    dst.write(arr, 1)
```



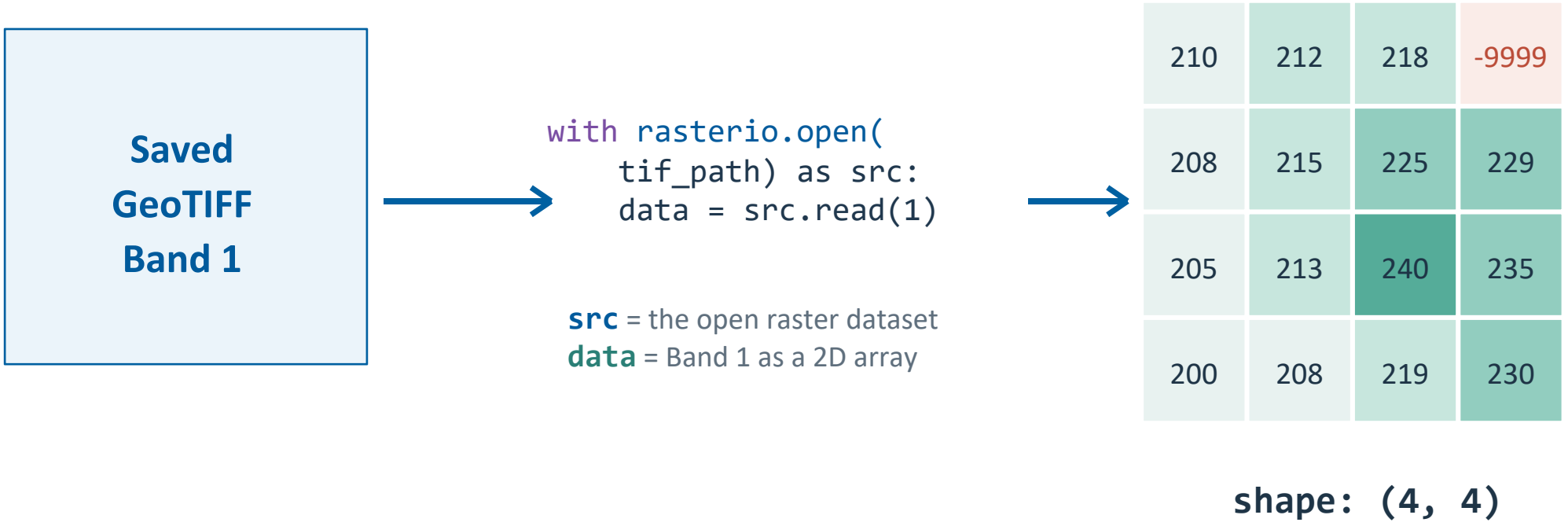
elevation_demo.tif

print("Saved:", tif_path) → week4_outputs/elevation_demo.tif

The array becomes band 1 of a georeferenced GeoTIFF.

Reopen the GeoTIFF and Read Band 1

Opening the file gives us src; reading band 1 gives us a NumPy array.



Rasterio band numbers start at 1; NumPy row/column indices start at 0.

Inspect the Open Raster

data contains the band values; src gives us the raster metadata.

WHILE src IS OPEN

```
with rasterio.open(tif_path) as src:
    data = src.read(1)
    print(data.shape, data.dtype)
    print(data[2, 1])
    print(src.width, src.height, src.count)
    print(src.crs, src.res)
    print(src.bounds, src.nodata)
```

WHAT COMES FROM WHERE

```
FROM data
shape       (4, 4)
cell[2,1]  213.0
dtype       float32

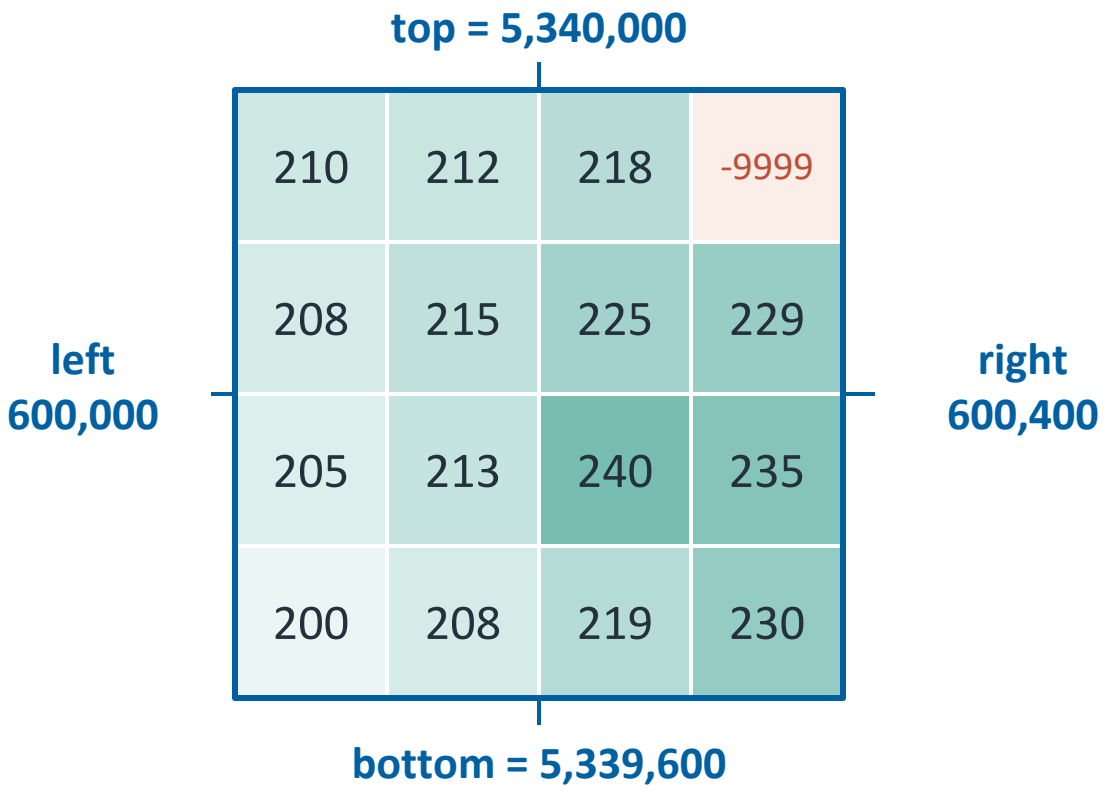
FROM src
size/bands  4, 4, 1
CRS         EPSG:32633
res         (100, 100)
NoData      -9999.0
```

src.bounds (L, B, R, T) (600000, 5339600, 600400, 5340000)

Values come from data; spatial metadata come from src.

Bounds and Pixel Spacing

Bounds locate the whole grid. Resolution describes one cell.



`src.bounds`

`left`	**600000**
`bottom`	**5339600**
`right`	**600400**
`top`	**5340000**

`src.res = (100, 100)`

`One cell: 100 m × 100 m`

shape = (4, 4)

`Whole extent: 400 m × 400 m`

The four bounds describe outer edges. src.res uses the dataset's map units.

Rasterio Confirms the Cell Centre

src.xy(row, col) returns the centre in the dataset CRS.

Array index

	0	1	2	3
0	210	212	218	-9999
1	208	215	225	229
2	205	**213**	240	235
3	200	208	219	230

arr[2, 1] returns 213 m

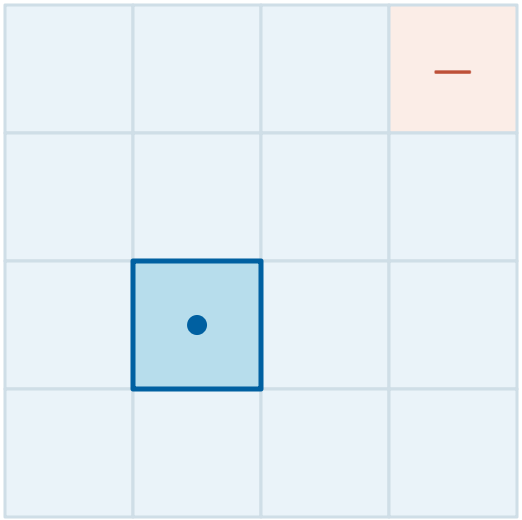
```
with rasterio.open(
    tif_path) as src:
    x_rasterio, y_rasterio = (
        src.xy(2, 1)
    )
```

```
(x_manual, y_manual) == (
    x_rasterio, y_rasterio)
```

True

Map location

EPSG:32633 (metres)



Cell-centre coordinates

x = 600150 m

y = 5339750 m

Keep Cell Positions When Finding the Maximum

Keep the 2D grid so the maximum can be located.

```
safe = np.where(valid, arr, -np.inf)
```

arr

210	212	218	-9999
208	215	225	229
205	213	**240**	235
200	208	219	230

Replace NoData

safe: same shape

210	212	218	−∞
208	215	225	229
205	213	**240**	235
200	208	219	230

`safe.max()`

240 m

The maximum value

Convert a Flat Index Back to Row and Column

Then convert the maximum index into row/column.

safe

	0	1	2	3
0	210	212	218	−∞
1	208	215	225	229
2	205	213	**240**	235
3	200	208	219	230

Find the position in the flattened array

```
flat_index = safe.argmax()
```

10 = 2 × 4 + 2

row × number of columns + column

```
max_row, max_col = np.unravel_index(
    flat_index, arr.shape)
```

(max_row, max_col) = (2, 2)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
210	212	218	−∞	208	215	225	229	205	213	**240**	235	200	208	219	230

src.xy(max_row, max_col)
(600250, 5339750) m at the 240 m cell

Rasterio Can Read Masked Values Directly

Read the file's values together with its own exclusion mask.

`src.read(1)`

`src.read(1, masked=True)`

Values

210	212	218	-9999
208	215	225	229
205	213	240	235
200	208	219	230

+

Validity information

✓	✓	✓	×
✓	✓	✓	✓
✓	✓	✓	✓
✓	✓	✓	✓

→

Masked array

210	212	218	—
208	215	225	229
205	213	240	235
200	208	219	230

−9999 stays in data

✓ included × excluded

— means masked

valid=True includes a cell. A masked array's mask=True excludes it.

Placing Raster Values in Map Coordinates

Use the dataset bounds to place the image in coordinate space.

```
with rasterio.open(tif_path) as src:
    masked = src.read(1, masked=True)
    b = src.bounds
    fig, ax = plt.subplots()
    im = ax.imshow(masked, origin="upper",
        extent=[b.left, b.right, b.bottom, b.top],
        cmap="terrain", interpolation="nearest")
```

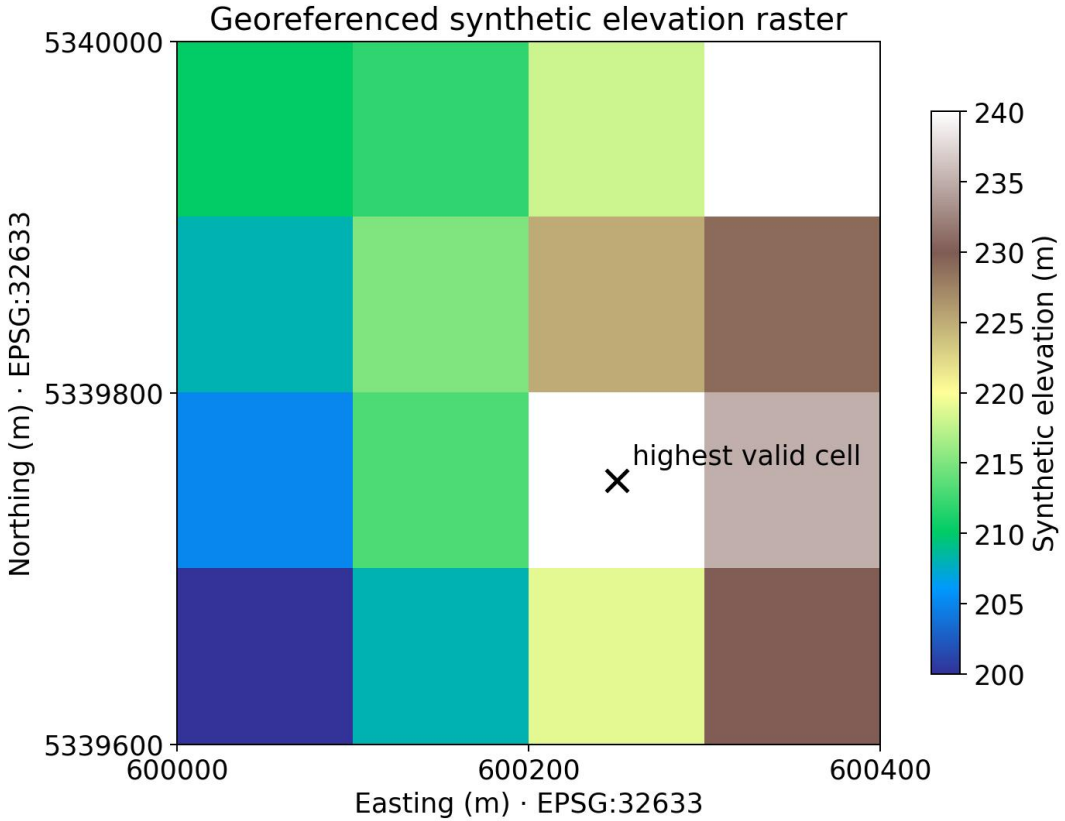
`extent = [left, right, bottom, top]`

fig: whole figure ax: map axes im: raster image for the colourbar

The notebook adds the marker, colourbar and axis labels.

Interpreting a Georeferenced Map

Coordinates, colour and the marker describe the same raster.




```
px, py = src.xy(max_row, max_col)
fig.colorbar(im, ax=ax, label="Elevation (m)")
```

READING THE MAP

Axes: easting / northing (m)
CRS: EPSG:32633
Colour: elevation in metres
Blank cell: NoData
×: highest valid cell
240 m at row 2, column 2

Reading a Real DEM's Own Metadata

The workflow carries over. The file supplies its spatial properties.

SYNTHETIC DEM

210	212	218	—
208	215	225	229
205	213	240	235
200	208	219	230

4 × 4 cells · 100 m · EPSG:32633

VS

REAL VIENNA DEM

shape	(1500, 1500)
resolution	(1, 1) m
CRS	EPSG:31256
NoData	-9999

Height reference: Wiener Null
Defines elevation zero · source: Stadt Wien

```
with rasterio.open("W04_Vienna_DEM_Clip.tif") as src:
    print(src.crs, src.res, src.nodata)
```

The synthetic parameters describe our example. Read real-file parameters before analysis.

Valid Statistics for Real Elevation

Use unmasked values that are finite: neither NaN nor infinity.

Valid observations

```
with rasterio.open("W04_Vienna_DEM_Clip.tif") as src:
    values = src.read(1, masked=True)
valid = ~np.ma.getmaskarray(values)
valid &= np.isfinite(np.asarray(values))
elevations = np.asarray(values)[valid]
elevations.min(), elevations.max(), elevations.mean()
```

1 ~ : excluded → included

2 &= : also require finite values

3 [valid] selects usable values

A negative elevation can be valid.

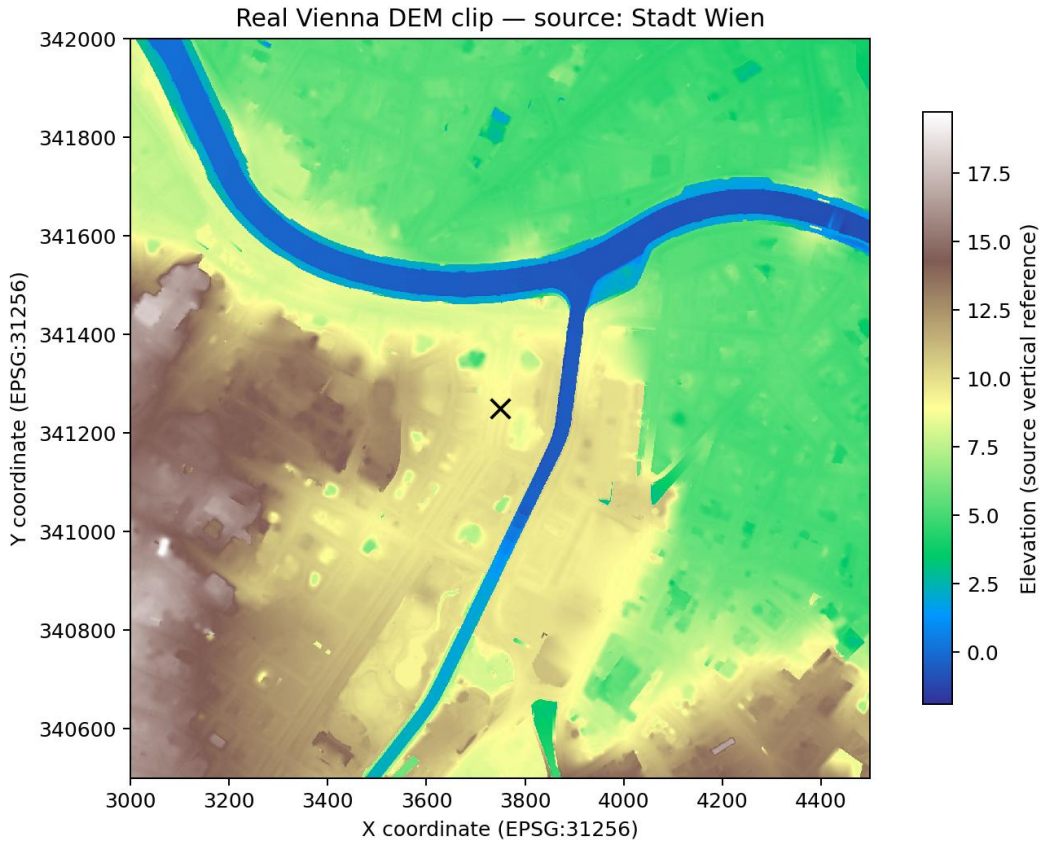
Negative values can lie below the reference zero.
–9999 is the declared NoData marker.

These statistics describe this clip, relative to Wiener Null.

Min -1.897 m Max 19.709 m Mean 8.053 m

Interpreting the Vienna DEM

Connect one cell's value, array index and map coordinates.



1 m pixel

EPSG:31256

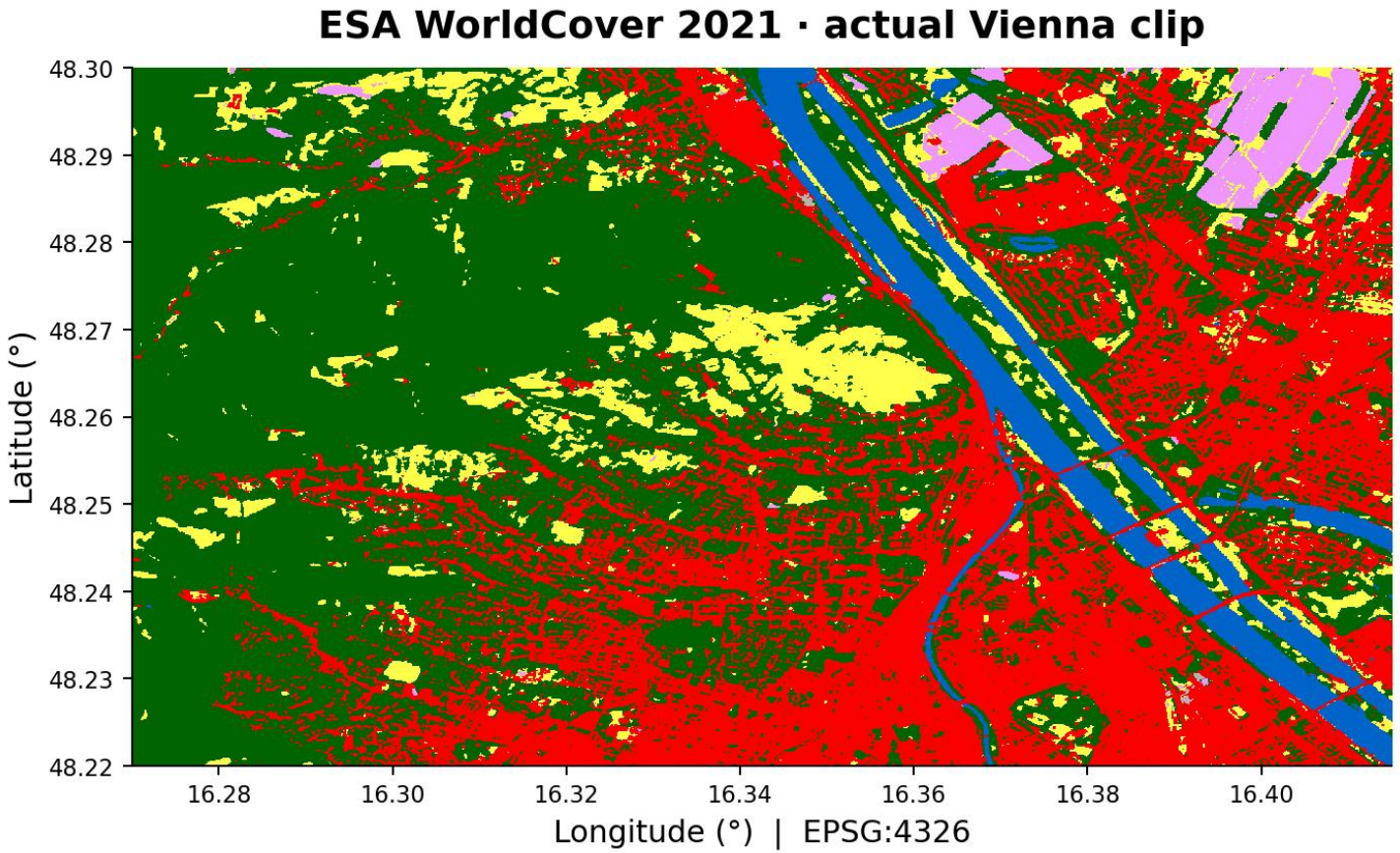
2.25 M valid cells

Example valid cell
row 750, col 750
value: 9.036 m
x: 3750 y: 341250

Stadt Wien DGM · Datenquelle: Stadt Wien - data.wien.gv.at · CC BY 4.0 · Vertical datum: Wiener Null

From Elevation to Land-Cover Analysis

Reuse read / inspect / mask; switch to counts / shares and a discrete legend.



ESA WORLDCOVER 2021

Official 11-class legend · not clip statistics

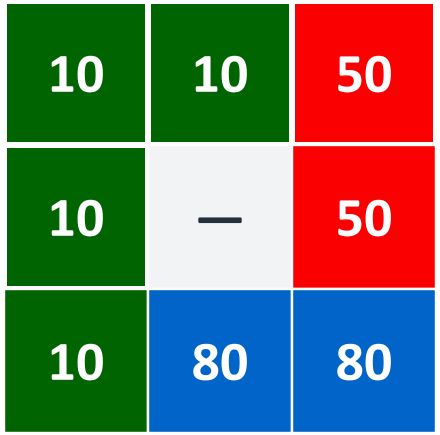
- 10 Tree cover
- 20 Shrubland
- 30 Grassland
- 40 Cropland
- 50 Built-up
- 60 Bare / sparse
- 70 Snow / ice
- 80 Permanent water
- 90 Herbaceous wetland
- 95 Mangroves
- 100 Moss / lichen

© ESA WorldCover project 2021 / Contains modified Copernicus Sentinel data (2021) processed by ESA WorldCover consortium

Class Counts and Shares

Count valid category codes, then divide by all valid cells.

Synthetic 3 × 3 example



8 valid cells

`NoData = -9999`

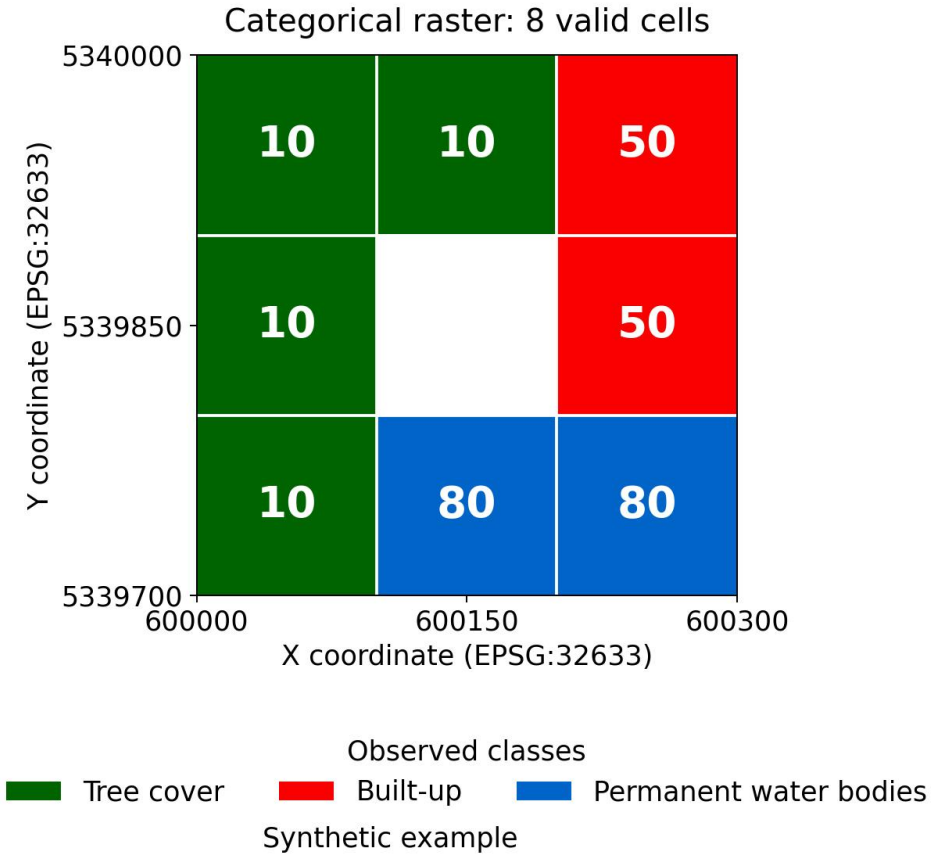
Class	Count	Share
10 Tree cover	4	50%
50 Built-up	2	25%
80 Permanent water	2	25%

```
classes, counts = np.unique(
    toy_data[toy_valid], return_counts=True)
shares = 100 * counts / counts.sum()
```

Share = class count / all valid cells.

Categorical Colours and Legends

Use a fixed colour for each code and show the observed classes in the legend.



USING THE PROVIDED TEMPLATE

```
toy_fig, toy_ax = plot_categorical(
    toy_data, toy_valid, classes,
    toy_labels, toy_colors, toy_extent,
    toy_crs, toy_title, toy_source
)
```

One code keeps one colour.
Only observed classes enter the legend.
NoData remains blank.

Section 10 of lab notebook supplies the reusable function and this toy example.

LAB TIME