

A COURSE IN TEN PARTS

Python, from the Ground Up

*Logic and code for people who already think in maps,
layers and pixels*

ArcGIS Pro · QGIS · Google Earth Engine · Google Cloud

Written for a GIS professional who has never programmed before

What's inside

Before we start Why this is written the way it is

Part 0 Ground truths: what a program really is

Part 1 Thinking before typing: the three shapes of all logic

Part 2 Holding information: values, names, lists and dictionaries

Part 3 Decisions: teaching the computer to choose

Part 4 Repetition: doing it fifty times without getting bored

Part 5 Functions: building your own tools

Part 6 Objects and modules: things that know how to do things

Part 7 When things break: reading errors like a map

Part 8 Files, folders and tables of data

Part 9 Into the tools: Earth Engine, Google Cloud, QGIS, ArcGIS

Part 10 The capstone: one real workflow, start to finish

Appendix A one-page memory aid, and a rhythm for practice

Before we start

Most programming books begin with a sentence like "Type `print('Hello, World!')` and press Enter." Then they explain what just happened. That order is backwards for a grown professional. You end up memorising shapes of text without knowing why they look that way, and the moment something breaks, there is nothing underneath to stand on.

This document goes the other way round. Every idea starts with a situation you already know from your own work: a shapefile with a messy attribute table, a folder of fifty satellite scenes, a clip that needs doing for every district in a state. From that situation we ask a plain question: *what is actually going on here, underneath?* Only once the idea is clear in ordinary language does the Python show up, and when it does, it will feel less like new vocabulary and more like a short way of writing down something you already understood.

This is sometimes called learning from **first principles**. Instead of copying what others do, you break a thing down to the few facts that cannot be broken down further, and build back up from there. A physicist does it with energy and mass. We will do it with *values*, *names*, *decisions* and *repetition*. It turns out that nearly all of programming, including the code that drives Earth Engine's planet-scale computations, is made from a surprisingly small number of such pieces.

A few things about how this is written:

- **You already know more than you think.** If you have ever used *Select by Attributes*, you have written a condition. If you have ever used *Batch* in a geoprocessing tool, you have run a loop. If you have ever built a model in ModelBuilder, you have written a function. We will keep pointing at these connections, because they are your biggest head start.
- **Syntax comes last, and slowly.** Python's rules for commas and brackets are real, but they are the least interesting part. They come after the idea, never before.
- **Small code, read carefully.** The examples are short on purpose. A five-line example you fully understand teaches more than a fifty-line script you copy.
- **Moments to pause.** Now and then you'll see a line in italics beginning *Think about it*. These are not homework. They are the points where it is worth stopping for thirty seconds and answering in your own head before reading on.

The early parts (0 to 8) are pure Python and pure logic, with GIS used for examples. Part 9 is where it all meets the software you use every day. Part 10 puts everything together into one piece of real work. You can read it in order, and it is designed to be read in order: each part quietly relies on the one before.

A good pace is one part every two or three days, with some time in between spent playing with the examples in Google Colab (you'll meet it in Part 0). Reading without typing is like reading about swimming. The water is where it sticks.

Ground truths

What a program really is, what the computer is really doing, and where Python lives inside the tools you already use.

A computer is a very fast, very literal assistant

Imagine you've hired an assistant with two unusual qualities. First, they are unbelievably fast. They can measure the area of ten thousand polygons before you finish a sip of tea. Second, they have zero common sense. They will do exactly what you say, in exactly the order you say it, and nothing else. If you say "measure the fields" and forget to say which layer the fields are in, they will not guess. They will simply stop and tell you they don't understand.

That's a computer. Not a brain, not a mind — a fast, literal assistant.

When you use ArcGIS Pro or QGIS by clicking, you are giving this assistant instructions through buttons. Every click on *Buffer*, every value typed into *Distance*, every *Run* — those are instructions. The software translates your clicks into a precise list of steps and hands it over.

A **program** is simply that list of steps, written down as text instead of clicked. That's all. When people say "code", they mean the text of that list.

Think about it. *When you run the Buffer tool in QGIS, what exactly does the assistant need to know? Which layer, how far, in what units, where to save the result. Four facts. Keep that in mind; later you'll see the same four facts appear in one line of Python.*

Why write it down at all, if clicking works?

Clicking works beautifully for something you do once. It starts to hurt in three situations, and you've probably met all three.

Repetition. Clipping one raster to one district is a minute of work. Clipping forty rasters to seventy districts is 2,800 clips. Nobody should click 2,800 times.

Memory. Six months from now, someone asks how you produced a map. Which filter did you use? Which cloud threshold? With clicks, the answer lives in your head (or nowhere). With code, the answer is the code. It is a recipe that anyone can read and re-run.

Scale. Some work cannot be clicked at all. Earth Engine can compute the average greenness of every farm in a country across ten years of satellite images, but it has no button for that exact question. You have to describe the question, and the language you describe it in is code.

So a program is not a replacement for your GIS knowledge. It is a way of *recording* your GIS knowledge so it can be repeated, shared, and scaled.

A program is a recipe

The single most useful analogy in this whole document is this one: **a program is a recipe.**

A recipe has three kinds of things in it:

1. **Ingredients:** the things you work with (flour, onions; or in our world, a shapefile, a date range, a cloud percentage).
2. **Steps:** what you do with them, in order (chop, fry, stir; or filter, clip, calculate).
3. **A result:** what you end up with (a curry; or a map, a table, a number).

Keep this picture in your head. When we later meet *variables*, they are ingredients with labels on them. When we meet *functions*, they are small recipes you can reuse inside bigger recipes. When we meet Earth Engine, you'll discover that your Python code doesn't cook at all. It writes the recipe and posts it to Google's kitchen, which does the cooking. That one idea will save you weeks of confusion later.

What is Python, and why this one?

There are hundreds of programming languages, just as there are hundreds of spoken ones. Each is a set of agreed rules for writing down instructions. Python became the language of GIS for a few practical reasons:

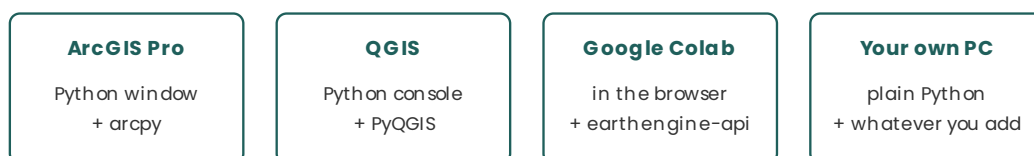
- It reads almost like English. `if cloud_cover < 20:` is not far from how you'd say it aloud.
- ArcGIS Pro, QGIS and Earth Engine all chose it. ArcGIS Pro ships with Python built in (the `arcpy` library). QGIS ships with its own Python (called PyQGIS). Google provides an official Python library for Earth Engine, and Google Cloud's tools all speak Python.
- Almost everything else you might need, such as reading Excel files, drawing charts, or doing statistics, already exists as a free Python add-on.

This means that learning Python once gives you a single language that talks to *every* tool on your desk. The words change from tool to tool, but the grammar underneath is the same.

Where does Python live?

This is a point that confuses nearly every beginner, so let's settle it now.

Python is not one thing sitting in one place on your computer. Think of it as an **engine** that can be installed inside different vehicles:



Same language, same grammar. Different add-ons plugged in, depending on where you are.

- **Inside ArcGIS Pro** there is a *Python window* (and notebooks). Here Python comes with `arcpy` already plugged in, so it can reach your geodatabases and every geoprocessing tool.
- **Inside QGIS** there is a *Python Console* (Plugins → Python Console). Here Python can talk to your open map, your layers, and the Processing toolbox.
- **In Google Colab**, which is just a web page from Google, Python runs on Google's computers, and you type into your browser. Nothing to install. This is where we will do most of our learning, and where Earth Engine work is most comfortable.

- **On your own computer**, Python can also be installed on its own. You won't need this for a while.

The key insight: *the grammar you learn in Colab is exactly the grammar you'll use in ArcGIS and QGIS.* Only the add-ons change. So we learn in the easiest place first.

Your first look at Colab

Open `colab.research.google.com` and sign in with a Google account. Choose *New notebook*. You'll see a page with a box in it. That box is called a **cell**. You type Python into a cell, press the ► button (or Shift + Enter), and the result appears right below.

A notebook is a mix of cells, some with code and some with notes, like a lab notebook where the experiments actually run. You can re-run any cell at any time.

Type this into the first cell and run it:

```
| print("The assistant is listening.")
```

The words appear underneath. `print` is an instruction meaning *show me this*. The text inside the quotation marks is what gets shown. That's your first program. It is one step long, and the assistant followed it exactly.

Now try something slightly more interesting:

```
| print(2 + 3)
| print(1250 * 0.0001)      # a 1,250 m² plot, in hectares
```

Notice the second line has a `#` followed by a note. Everything after a `#` on a line is ignored by Python. It is a **comment**, a note for the human reader. Good programmers leave lots of them. You will too.

Think about it. Why does `print(2 + 3)` show 5 and not `2 + 3`? Because without quotation marks, Python treats it as something to **work out**. With quotation marks, it treats it as text to show as-is. That difference, **something to calculate** versus **text to show**, is the first ground truth of Part 2.

The ground truths, gathered

Before moving on, here is everything this part established, in a few lines. These are the foundations the rest of the course stands on.

- A computer is a fast, literal assistant. It does exactly what it is told, in order, and never guesses.
- A program is a written list of instructions: a recipe with ingredients, steps and a result.
- Code is worth writing when work repeats, needs remembering, or is too big to click.
- Python is one language that lives inside many tools. Learn the grammar once and it works everywhere.
- Colab is the no-install place to learn, and it runs on Google's computers.

Thinking before typing

Every program ever written is built from three shapes. Once you can see them in your own work, the code is just transcription.

The hardest part of programming isn't the code

Here is something experienced programmers know and beginners rarely hear: most of the difficulty in programming is *not* Python. It is being clear, in plain language, about what you want done, step by step, with nothing left to guesswork.

Remember the literal assistant from Part 0. The assistant is never the problem. The problem is always the instructions. So before we learn a single rule of Python, we'll practise writing instructions that a literal assistant could follow.

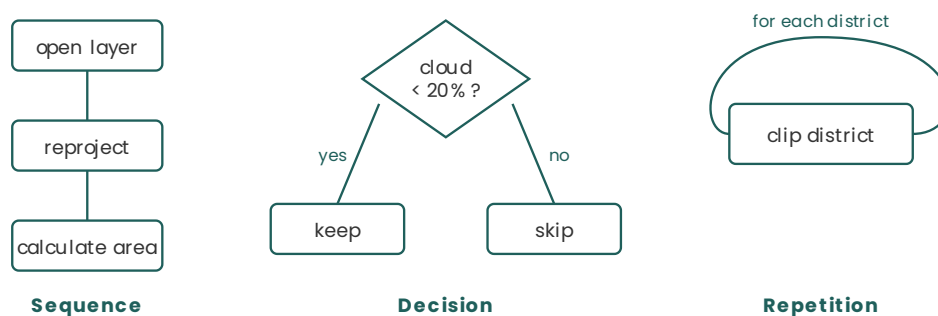
The three shapes of all logic

It is a remarkable fact, proven by computer scientists decades ago, that *any* procedure a computer can carry out can be built from just three shapes. Every script, every app, every Earth Engine analysis: all of it, underneath, is these three things combined.

1. Sequence: do this, then this, then this. Steps in order. Open the layer. Reproject it. Calculate the area. The order matters: you can't calculate area in metres before you've reprojected to a metre-based system.

2. Decision: only if something is true, do this. A fork in the road. If the cloud cover is below 20%, keep the image. If the land-use code is 'AGRI', colour it green; otherwise, colour it grey.

3. Repetition: do this for each one. A loop. For each district in the state, clip the raster. For each image from 2020 to 2024, compute NDVI.



That's the whole alphabet of logic. Python gives each shape a way of being written: sequence is just lines one after another, decision is `if`, repetition is `for`. Parts 3 and 4 cover the last two. But you can use all three *today*, in plain language.

Writing steps before writing code

Programmers have a name for writing out the logic in plain language first: **pseudocode**. It is not a real language. It has no rules. It is just numbered steps, clear enough that a literal assistant could follow them.

Let's take a real task and do it properly.

The task: "Find which villages in the district had healthy crops in August."

A first attempt, the way we'd say it to a colleague:

Get the satellite images for August, work out the greenness, and see which villages are green.

A colleague would understand this. The literal assistant would not. Which satellite? Which August? What is "greenness", exactly? What counts as "green enough"? Let's rewrite it for the assistant:

1. Take the district boundary.
2. Take all Sentinel-2 images that cover the district.
3. Keep only images taken between 1 and 31 August 2025.
4. Of those, keep only images with less than 20% cloud.
5. For each remaining image, calculate NDVI.
6. Combine all the NDVI images into one, using the median value per pixel.
7. For each village polygon:
 - work out the average NDVI inside it.
 - If the average is above 0.5, mark the village "healthy".
 - Otherwise, mark it "stressed".
8. Save the list of villages with their labels.

Look closely and you'll find all three shapes. Steps 1 to 6 are mostly *sequence*. Step 4 is a *decision* applied to every image. Step 5 and step 7 are *repetition* ("for each"). Inside step 7 there is another *decision* (above 0.5 or not).

This is, almost line for line, the Earth Engine script we'll write in Part 10. The hard thinking is already done. What remains is translation.

Think about it. Step 6 says "median", not "average". Why might a GIS person choose the median here? (Because a single cloudy or shadowed pixel that slipped through the filter can drag an average down badly, while a median ignores the odd outlier.) This is domain knowledge, and it belongs to **you**, not to Python. Code can only be as good as the thinking fed into it.

Four habits of clear instructions

When you write pseudocode, a few habits make the difference between instructions a literal assistant can follow and ones it can't.

Name the exact thing. Not "the layer" but "the village polygons layer". Not "recent images" but "images from 1–31 August 2025". Vague nouns become bugs later.

Put numbers on judgements. "Low cloud" means nothing to a computer. "Less than 20%" does. Wherever you find yourself using words like *good*, *high*, *nearby* or *recent*, replace them with a threshold.

Break big steps down until each one is obvious. "Calculate NDVI" is fine if you know NDVI is $(\text{NIR} - \text{Red}) / (\text{NIR} + \text{Red})$. If you don't, it needs breaking down further. A step is small enough when you could do it by hand on one pixel.

Say **what happens in the "otherwise" case**. Beginners write *if healthy, label green* and forget to say what happens to the rest. A literal assistant will simply leave them unlabelled.

Decomposition: the master skill

The general skill behind all of this has a name: **decomposition**. It means taking a problem too big to hold in your head and splitting it into smaller problems, then splitting those, until every piece is trivially easy.

You already do this in GIS. Nobody makes a flood risk map in a single step. You get elevation, derive slope, find areas near rivers, overlay land use, combine them with weights. Each of those is a smaller problem. A program is built the same way: a big recipe made of small recipes, each of which is made of single steps.

When you get stuck later in this course (and everyone does), the cure is almost always the same: step back from the code and ask, "*what are the smaller steps here, in plain words?*"

What you now own

- All logic is built from three shapes: **sequence, decision, repetition**.
- Pseudocode means writing the steps in plain, precise language before touching code.
- Clear instructions name exact things, put numbers on judgements, and handle the "otherwise".
- Decomposition, meaning breaking big problems into small ones, is the core skill. Python is just how we write the small pieces down.

Holding information

Values, names, lists and dictionaries. Or, as you already know them: cells, fields, columns and attribute rows.

Everything is a value

Open any attribute table. Each cell holds one piece of information: a district name, a population number, an area, a yes/no flag. In Python, one piece of information is called a **value**.

And just like attribute table fields have a type (Text, Short Integer, Double, Date), Python values have a type too. The type decides what you can do with the value. You can add two numbers. You can't sensibly add two district names, or at least, it doesn't mean the same thing.

There are only a handful of basic types you need to know, and you've met them all in field definitions already:

Python type	What it holds	Your field-type cousin	Example
int	whole numbers	Short / Long Integer	2025, -3, 144
float	numbers with decimals	Float / Double	0.62, 28.6139
str	text ("string" of characters)	Text	"Ghaziabad", "B4"
bool	yes or no	a 0/1 flag	True, False
None	nothing, empty	a NULL cell	None

A **string** always sits inside quotation marks. That's how Python tells the difference between text and a name (more on names in a moment). "2025" in quotes is text: four characters that happen to look like a number. 2025 without quotes is a number you can do arithmetic with. This distinction causes many beginner errors, and you'll see it again when reading CSV files, where everything arrives as text.

```
print(2025 + 1)      # 2026: arithmetic on a number
print("2025" + "1")  # 20251: text glued to text
```

Think about it. Have you ever joined a table in GIS and had it silently fail because one table stored district codes as text and the other as numbers? That's exactly this. The values **looked** the same but had different types, so they never matched.

Names: labels stuck on values

A value by itself isn't much use if you can't refer back to it. So we give it a name:

```
cloud_limit = 20
district = "Meerut"
ndvi_threshold = 0.5
```

Read the = sign here as **"is now labelled"**, not as "equals". The line `cloud_limit = 20` means *take the value 20 and stick the label `cloud_limit` on it*. Programmers call these labelled values **variables**.



A variable is a name-tag tied to a value. Use the name, and Python fetches the value.

Once labelled, you use the name wherever you'd use the value:

```
print(cloud_limit)          # 20
print(cloud_limit * 2)      # 40
```

Why bother? Because of the "memory" and "repetition" reasons from Part 0. If your script uses the cloud threshold in six places and you later decide 15% is better, you change *one* line, `cloud_limit = 15`, and every use updates. It's the same reason you'd put a threshold in a single cell of Excel and reference it, rather than typing 20 into fifty formulas.

A label can be moved to a new value at any time:

```
year = 2023
year = 2024      # the label now points at 2024; 2023 is forgotten
year = year + 1  # take the current value (2024), add 1, re-label: now 2025
```

That last line looks impossible as maths (*year cannot equal year + 1!*) but makes perfect sense once you read = as "is now labelled": *work out the right-hand side first, then move the label onto the result*.

Choosing good names

Python lets you name things almost anything, with a few rules: names can contain letters, digits and underscores, cannot start with a digit, and cannot have spaces. `ndvi_2024` is fine; `2024_ndvi` and `ndvi 2024` are not.

Beyond the rules, a good name tells a stranger what the thing is. `x = 0.5` tells nothing. `ndvi_threshold = 0.5` tells everything. The Python community writes names in lowercase with underscores between words (`village_count`), and you should too. It's the style every tool's documentation uses, so your code will look at home next to theirs.

Python also cares about capital letters: `District` and `district` are two different names. Many a beginner has spent half an hour on that.

Doing arithmetic

Python is a very good calculator. The symbols are mostly familiar:

```
area_m2 = 48250
area_ha = area_m2 / 10000      # divide → 4.825
perimeter = 2 * (120 + 85)     # multiply, brackets work as in school maths
pixels = 10980 ** 2            # power: a 10,980 × 10,980 Sentinel-2 tile
nir, red = 0.41, 0.08
ndvi = (nir - red) / (nir + red)
print(round(ndvi, 3))          # 0.673
```

Two you might not know: `//` divides and throws away the remainder (`17 // 5` is 3), and `%` gives only the remainder (`17 % 5` is 2). The remainder trick is handy for things like "every 10th image".

Notice line `nir, red = 0.41, 0.08`: Python can label two values at once. Small conveniences like this are part of why people enjoy the language.

Working with text

Strings are the second-most common thing you'll handle, after numbers: file names, district names, band names, dates, paths. A few things you can do with them:

```
state = "Uttar Pradesh"
print(len(state))          # 13: how many characters
print(state.upper())       # UTTAR PRADESH
print(state.replace(" ", "_")) # Uttar_Pradesh: handy for file names
```

The most useful string trick of all is the **f-string**, which lets you build text with values slotted in. Put an **f** before the opening quote, and anything inside `{curly brackets}` is worked out and inserted:

```
district = "Meerut"
year = 2025
filename = f"ndvi_{district}_{year}.tif"
print(filename)          # ndvi_Meerut_2025.tif
```

You'll use f-strings constantly, for output file names, for messages, and for building labels on maps. They're the Python version of the *Concatenate* expression in Field Calculator.

Lists: a column of values

So far each name holds one value. But GIS is rarely about one thing. It's about many districts, many images, many bands. For an ordered collection of values, Python has the **list**.

```
bands = ["B2", "B3", "B4", "B8"]
years = [2021, 2022, 2023, 2024]
```

A list is written in square brackets, with values separated by commas. Think of it as **one column of an attribute table**: a sequence of values, in order, under one name.

Every item in a list has a position, called its **index**. And here comes the one rule that trips up absolutely everyone: **counting starts at 0, not 1**.

```
bands = ["B2", "B3", "B4", "B8"]
#         0      1      2      3      ← the index of each item

print(bands[0])    # B2: the first item
print(bands[2])    # B4
print(bands[-1])   # B8: negative numbers count from the end
print(len(bands))  # 4: how many items
```

Why zero? The index is really an *offset*: how many steps from the start. The first item is zero steps from the start. It feels odd for about a week and then it doesn't. (The same convention runs through Earth Engine, NumPy, and nearly every programming tool you'll ever meet.)

You can also take a slice of a list: `bands[0:2]` gives items at positions 0 and 1, so `["B2", "B3"]`. The rule is *start included, end excluded*. That makes `bands[0:2]` exactly 2 items long.

Lists can change. You can add to them and change items:

```
districts = ["Meerut", "Ghaziabad"]
districts.append("Hapur")    # add to the end
districts[0] = "Meerut (urban)" # replace the first item
print(districts)
```

Dictionaries: one row of an attribute table

A list is good for "many values of the same kind". But what about *one feature*, which has many *different* attributes? A village has a name, a population, an area, a district code. For that, Python has the **dictionary**.

```
village = {  
    "name": "Rasoolpur",  
    "population": 4120,  
    "area_ha": 312.5,  
    "irrigated": True  
}
```

A dictionary is written in curly brackets. Inside are pairs of **key : value**. The key is the field name, the value is what's in the cell. It is precisely **one row of an attribute table**.

You look things up by key, not by position:

```
print(village["name"])      # Rasoolpur  
print(village["population"]) # 4120  
  
village["population"] = 4200 # update a field  
village["ndvi_aug"] = 0.58  # add a brand-new field
```

That last line is like Add Field followed by Calculate Field, for a single row, in one go.

Think about it. Why would a dictionary look things up by name rather than by position? Because with attributes, position means nothing. Is population "field 2" or "field 3"? Nobody cares, and it might change. What matters is the name. A list is for order; a dictionary is for meaning.

Putting them together: the attribute table, rebuilt

Now watch what happens when we combine the two containers. A list of dictionaries, in which each dictionary is one row, is **an entire attribute table**:

```
villages = [  
    {"name": "Rasoolpur", "population": 4120, "ndvi": 0.58},  
    {"name": "Kheda",      "population": 2890, "ndvi": 0.41},  
    {"name": "Daurala",    "population": 6310, "ndvi": 0.66},  
]  
  
print(villages[1]["name"]) # Kheda: row 1, field "name"  
print(len(villages))      # 3 rows
```

name	population	ndvi
Rasoolpur	4120	0.58
Kheda	2890	0.41
Daurala	6310	0.66



```
[  
    {"name": "Rasoolpur", "population": ...},  
    {"name": "Kheda",      "population": ...},  
    {"name": "Daurala",    "population": ...},  
]
```

This is not just a teaching trick. It's genuinely how data comes back to you from many tools. When Earth Engine hands you the properties of a feature, it hands you a dictionary. When you read a CSV row by row, each row can be a dictionary. When ArcGIS gives you field values, they come in a row that behaves like a list. You now know the shape of nearly all the data you'll touch.

Changing one type into another

Sometimes a value arrives as the wrong type (very often text, when you needed a number). Python has converters named after the types:

```
pop_text = "4120"          # e.g. read from a CSV
pop = int(pop_text)        # now the number 4120
ndvi = float("0.58")      # now the number 0.58
label = str(2025)         # now the text "2025"
print(type(pop))          # <class 'int': ask Python what type something is
```

`type()` is a good friend while learning. Whenever something behaves strangely, ask Python what type it thinks the value is. Very often, that's the answer.

What you now own

- A **value** is one piece of information, and its **type** (`int`, `float`, `str`, `bool`, `None`) decides what you can do with it.
- A **variable** is a name-tag on a value. `=` means "is now labelled", not "equals".
- **Lists** are ordered collections (a column), indexed from **0**.
- **Dictionaries** are key-to-value pairs (a row).
- A **list of dictionaries** is an attribute table. You now know the shape of most GIS data.
- **f-strings** build text with values inside: `f"ndvi_{year}.tif"`.

Decisions

Teaching the literal assistant to choose. You've been doing this for years in Select by Attributes.

Every decision starts with a question that has a yes-or-no answer

Open *Select by Attributes* in ArcGIS or QGIS and write "POPULATION" > 5000. What does the software do? It goes to each row and asks a question: *is this row's population greater than 5000?* The answer is always either yes or no. Rows where the answer is yes get selected.

That yes-or-no answer is the `bool` type from Part 2: `True` or `False`. And the question itself is called a **condition**, or a comparison.

```
population = 6310
print(population > 5000)    # True
print(population < 1000)   # False
```

Python's comparison symbols:

Symbol	Means	Example that is True
<code>==</code>	is equal to	<code>district == "Meerut"</code>
<code>!=</code>	is not equal to	<code>landuse != "WATER"</code>
<code>></code> <code><</code>	greater than, less than	<code>ndvi > 0.5</code>
<code>>=</code> <code><=</code>	at least, at most	<code>cloud <= 20</code>
<code>in</code>	is found inside	<code>"B8" in bands</code>

Notice the **double** equals sign for "is equal to". A single `=` already has a job: sticking labels on values. So asking a question needs a different symbol. Mixing them up (writing `if district = "Meerut"`) is the most common beginner mistake in this part, and Python will stop you with an error when you do.

The `in` check is wonderfully readable, and it works on lists, strings and dictionaries:

```
bands = ["B2", "B3", "B4", "B8"]
print("B8" in bands)    # True
print("B11" in bands)   # False
```

if: the fork in the road

A condition on its own just produces `True` or `False`. To make the assistant *act* on it, we use `if`:

```
cloud = 12

if cloud < 20:
    print("Keep this image.")
```

Read it aloud: *if cloud is less than 20, then print "Keep this image."* Three details carry the whole meaning:

- The line starts with `if` and ends with a **colon** `:`. The colon means *"here comes what to do"*.
- The line underneath is **indented**, pushed in by four spaces. The indentation is how Python knows which lines belong to the `if`. Everything indented under it happens only when the condition is `True`.
- When the indentation stops, the `if` is over. Lines after that happen regardless.

This use of indentation is Python's most distinctive feature. Other languages use brackets to group lines. Python uses the visual shape of the text itself. It means well-organised code looks well-organised: you can see the structure of the logic from across the room.

```
cloud = 35

if cloud < 20:
    print("Keep this image.")      # indented: only if True
    print("Adding to collection.") # indented: only if True
print("Checked one image.")       # not indented: always happens
```

Here, since 35 is not less than 20, only the last line prints.

else: what happens otherwise

In Part 1 we said clear instructions always handle the "otherwise". Python's word for it is `else`:

```
ndvi = 0.41

if ndvi > 0.5:
    status = "healthy"
else:
    status = "stressed"

print(status)    # stressed
```

Exactly one of the two branches will run, never both, never neither. That guarantee is useful: after this block, `status` is *always* labelled with something.

elif: more than two roads

Real classification rarely has only two classes. NDVI is commonly split into several:

```
ndvi = 0.34

if ndvi < 0:
    cover = "water"
elif ndvi < 0.2:
    cover = "bare soil or built-up"
elif ndvi < 0.5:
    cover = "sparse vegetation"
else:
    cover = "dense vegetation"

print(cover)    # sparse vegetation
```

`elif` is short for "else if". Python checks the conditions **from top to bottom and stops at the first one that is True**. That's why the second condition can simply say `ndvi < 0.2`: by the time Python gets there, it already knows the value isn't below 0. The order of the branches is part of the logic.

This is precisely a Reclassify operation, written out by hand. In fact, it's what the Reclassify tool does internally, cell by cell.

Think about it. What would happen if we put the `ndvi < 0.5` branch **first**? Then a value of -0.1 (water) would be caught by `< 0.5` and wrongly labelled "sparse vegetation". Order matters. When you write a chain of `elif`s, check each branch assuming everything above it was `False`.

Combining conditions: and, or, not

Selections often need more than one test. In GIS you'd write `"LANDUSE" = 'AGRI' AND "AREA_HA" > 10`. Python uses the same words, in lowercase:

```
landuse = "AGRI"
area_ha = 14.2
irrigated = False

if landuse == "AGRI" and area_ha > 10:
    print("Large farm.")

if landuse == "AGRI" and not irrigated:
    print("Rain-fed farm: watch NDVI in dry spells.")
```

- `and` is True only when **both** sides are True.
- `or` is True when **at least one** side is True.
- `not` flips True to False and back.

When conditions get long, use brackets to make the grouping obvious, just as you would in a query: `(landuse == "AGRI" or landuse == "ORCHARD") and area_ha > 10`.

Deciding with dictionaries

Now let's join this to Part 2. With one village as a dictionary, decisions read beautifully:

```
village = {"name": "Kheda", "population": 2890, "ndvi": 0.41}

if village["ndvi"] > 0.5:
    village["status"] = "healthy"
else:
    village["status"] = "stressed"

print(village)
```

We looked up a field, asked a question about it, and wrote a new field based on the answer. That is *Calculate Field with a condition*, for one row. What's missing is doing it for *every* row, and that's exactly what Part 4 gives us.

The one-line version

When an `if/else` only chooses between two values, Python allows a compact form that reads like a sentence:

```
status = "healthy" if ndvi > 0.5 else "stressed"
```

Status is "healthy" if NDVI is above 0.5, else "stressed". It means exactly the same as the four-line version. You'll meet it again inside pandas and ArcPy code.

A preview of a twist

Hold on to this thought for later: in Earth Engine, Python's `if` **does not work** on satellite images or on the numbers computed from them. Not because `if` is broken, but because of *where* those numbers live. We'll get to the reason in Part 9. For now, just note that you'll learn a different way of asking yes-or-no questions of pixels, and that it's based on exactly the same idea as this part.

What you now own

- A **condition** is a question with a True/False answer. Compare with `==`, `!=`, `<`, `>`, `<=`, `>=`, `in`.
- `if` runs indented lines only when the condition is True. The colon and the indentation *are* the grammar.
- `else` handles the otherwise. `elif` adds more branches, checked top to bottom, and the first True one wins.
- `and`, `or`, `not` combine conditions, just like in attribute queries.
- An `if/elif/else` chain is a hand-written Reclassify.

Repetition

*Doing a thing fifty times without getting bored, tired or careless.
This is where code starts paying back the effort.*

The batch button, opened up

Most geoprocessing tools in ArcGIS Pro have a *Batch* option, and QGIS has *Run as Batch Process*. You give it a list of inputs; it runs the same tool on each. You've used this. What you've used is a **loop**.

A loop is the third shape of logic from Part 1: *repetition*. It says: **for each item in a collection, do these steps**.

```
districts = ["Meerut", "Ghaziabad", "Hapur", "Bulandshahr"]

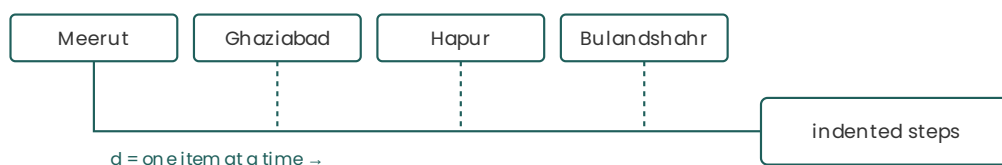
for d in districts:
    print(f"Clipping raster to {d}")
```

Output:

```
Clipping raster to Meerut
Clipping raster to Ghaziabad
Clipping raster to Hapur
Clipping raster to Bulandshahr
```

Let's slow down and read that `for` line word by word, because it is doing something quietly clever:

- `for d in districts:` means *take the list `districts`; one at a time, stick the label `d` on the next item; then run the indented lines*.
- First pass: `d` is labelled "Meerut". The indented line runs.
- Second pass: the label `d` moves to "Ghaziabad". The indented line runs again.
- ...and so on, until the list is used up. Then the loop ends, and Python carries on with whatever comes after.



The name `d` is your choice. `for district in districts:` is clearer and is what most people would write. The convention is a plural name for the collection and the singular for each item: `for image in images`, `for village in villages`, `for band in bands`.

Think about it. How many lines would it take to clip 70 districts without a loop? At least 70. With a loop, two. And if a 71st district is added next year, the loop needs **no change at all**. You just add a name to the list. That is the real power: the logic is written once, separately from the data it runs on.

Counting with `range`

Sometimes you don't have a list of things, just a number of times. `range` produces a sequence of numbers for you:

```
for year in range(2019, 2025):  
    print(f"Processing {year}")
```

This prints 2019 up to **2024**, not 2025. You've met this rule before, with list slices: *start included, end excluded*. `range(2019, 2025)` has exactly $2025 - 2019 = 6$ numbers. Python is consistent about this everywhere, which is a comfort once you know it.

Loops and decisions, together

Now the three shapes combine. Remember our list of villages from Part 2? Let's label every village, which is what we couldn't quite do at the end of Part 3:

```
villages = [  
    {"name": "Rasoolpur", "ndvi": 0.58},  
    {"name": "Kheda", "ndvi": 0.41},  
    {"name": "Daurala", "ndvi": 0.66},  
]  
  
for village in villages:  
    if village["ndvi"] > 0.5:  
        village["status"] = "healthy"  
    else:  
        village["status"] = "stressed"  
  
print(villages)
```

Look at the indentation: it now has two levels. The `if` is indented under the `for` (it happens for each village), and the assignments are indented under the `if/else` (they happen depending on the answer). Indentation is how Python shows *what lives inside what*. Read it like the outline of a report: headings, sub-points, sub-sub-points.

That short block is *Calculate Field with a condition, over the whole table*. You've rebuilt one of your most-used GIS operations from nothing but a list, a dictionary, a loop and an `if`.

Three patterns that cover most loops

In practice, nearly every loop you'll write in GIS work is one of three patterns. Learn to recognise them and you'll know what to write before you write it.

Pattern 1: Transform each item. Do something to every item: rename, compute, label. The example above is this pattern.

Pattern 2: Collect the ones that pass (a filter). Start with an empty list, and add items to it only when they meet a condition:

```
healthy = [] # an empty basket  
  
for village in villages:  
    if village["ndvi"] > 0.5:  
        healthy.append(village["name"])  
  
print(healthy) # ['Rasoolpur', 'Daurala']
```

Pattern 3: Build up a total (an accumulator). Start with zero, and add to it on every pass:

```
total = 0
for village in villages:
    total = total + village["ndvi"]

average = total / len(villages)
print(round(average, 2))          # 0.55
```

That's a *Summary Statistics* operation. Filter, transform, summarise: these three verbs will come back in Part 9 in a new form, as `filter`, `map` and `reduce`, which are the three heartbeats of every Earth Engine script. Everything you're learning here carries over directly.

A few more loop tools

Knowing the position. Sometimes you want both the item *and* its number. `enumerate` gives you both:

```
for i, band in enumerate(["B2", "B3", "B4"]):
    print(i, band)          # 0 B2, then 1 B3, then 2 B4
```

Looping over a dictionary. `.items()` hands you each key and its value together, which is perfect for printing a feature's attributes:

```
village = {"name": "Kheda", "population": 2890, "ndvi": 0.41}
for field, value in village.items():
    print(f"{field}: {value}")
```

Skipping and stopping. Inside a loop, `continue` means *skip the rest of this pass and go to the next item*, and `break` means *stop the loop entirely*. Use them sparingly; they're handy for things like "skip files that aren't TIFFs" or "stop once you find the first match".

Loops inside loops. A loop can live inside another loop. For every district, for every year:

```
for district in ["Meerut", "Hapur"]:
    for year in range(2022, 2025):
        print(f"ndvi_{district}_{year}.tif")
```

This prints six file names: every combination. Nested loops are how you process "every X for every Y", but they multiply quickly (70 districts × 10 years × 12 months = 8,400 passes), so it's worth pausing to count before running one on real data.

A short form you'll see everywhere

Once people are comfortable with Pattern 1 and Pattern 2, they often write them in a single line called a **list comprehension**. You'll see these in documentation and in other people's code, so it's worth being able to read one:

```
names = [v["name"] for v in villages]          # transform
healthy = [v["name"] for v in villages if v["ndvi"] > 0.5] # filter
```

Read it as a sentence: "make a list of *v*'s name, for each *v* in villages, if *v*'s NDVI is above 0.5." It means exactly the same as the four-line version. Use whichever form you find clearer. The long form is never wrong.

while: repeat until something changes

There is a second kind of loop, `while`, that repeats *as long as a condition stays True*, without a list to walk through:

```
waiting = 0
while waiting < 3:
```

```
print("Export still running...")  
waiting = waiting + 1
```

You'll rarely need it in GIS work. Its main use is waiting for something, such as checking whether a long Earth Engine export has finished. The one danger: if the condition never becomes False, the loop never ends. (If that happens in Colab, press the stop button next to the cell.)

What you now own

- `for item in collection:` runs the indented lines once per item, moving the label along.
- `range(a, b)` counts from a up to, but not including, b.
- Loops plus decisions give you filtering, labelling and totals: most of attribute-table work.
- The three loop patterns: **transform**, **filter** (collect into an empty list), **accumulate** (add into a starting total).
- Nested indentation shows what lives inside what. Read it like an outline.

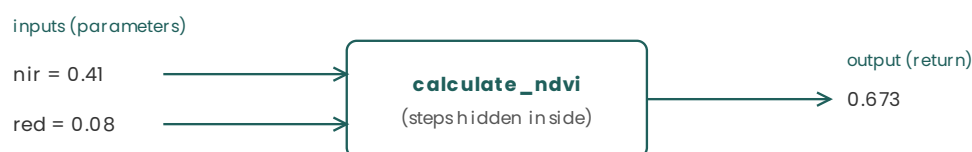
Functions

Building your own tools. This is the idea that turns scripts into something you can reuse, and it's the key that unlocks Earth Engine.

You've already built functions

Think about a model you've made in ModelBuilder, or a graphical model in QGIS. You chained a few tools, say *Clip* → *Calculate NDVI* → *Zonal Statistics*, and then exposed some of the values as **parameters**: the input raster, the boundary, the output folder. After that, the model appeared in your toolbox like any other tool. Anyone could double-click it, fill in the blanks, and get a result, without ever seeing the steps inside.

That is exactly what a **function** is: a named, reusable recipe with blanks to fill in. You give it inputs; it does its steps; it hands back a result.



You've also used dozens of functions already in this document without being told: `print()`, `len()`, `round()`, `int()`, `type()`. Each is a tool someone else built. You fill in the brackets, and it does its job. Now you'll build your own.

Writing one

```
def calculate_ndvi(nir, red):
    ndvi = (nir - red) / (nir + red)
    return ndvi
```

Line by line:

- `def` is short for *define*. It announces: *I'm building a new tool*.
- `calculate_ndvi` is the tool's name. Same naming rules as variables.
- `(nir, red)` are the **parameters**, the blanks to be filled in. Inside the function, they behave like ordinary variables.
- The colon and the indented lines should look familiar by now: they're the steps that belong to this tool.
- `return` hands the result back to whoever used the tool. It's the output parameter of your model.

Writing `def` builds the tool and puts it in your toolbox. **It does not run it**. Nothing happens yet, just as saving a model doesn't process any data. To run it, you **call** it by name, with actual values in the brackets:

```

result = calculate_ndvi(0.41, 0.08)
print(round(result, 3))          # 0.673

print(calculate_ndvi(0.30, 0.12)) # works again, with new inputs

```

When you call `calculate_ndvi(0.41, 0.08)`, Python temporarily labels `nir` as 0.41 and `red` as 0.08, runs the indented steps, and the return value comes back out and gets labelled `result`.

Why functions matter so much

You name an idea once. "NDVI" is a concept you think in. Once it's a function, your code can think in it too: `calculate_ndvi(nir, red)` reads like the concept, not like the arithmetic.

You fix things in one place. If you discover a mistake in the formula, you correct the function, and every place that uses it is corrected.

Big problems become small ones. This is decomposition from Part 1, made real. A long analysis becomes a handful of well-named functions (`load_images`, `mask_clouds`, `add_ndvi`, `summarise_by_village`) and the main script reads almost like pseudocode.

Think about it. Look back at the eight pseudocode steps in Part 1. Could each numbered step become one function? Most could. That's not a coincidence. Good pseudocode and good functions are the same thinking, written at different levels of detail.

return VERSUS print

This trips up many beginners, so let's make it sharp. `print` shows something on screen, for a human to look at. `return` hands something back, for the program to keep using.

```

def area_ha_print(area_m2):
    print(area_m2 / 10000)    # shows it, then the value is gone

def area_ha_return(area_m2):
    return area_m2 / 10000    # hands it back to be used

total = area_ha_return(48250) + area_ha_return(12000) # 6.025: works

```

Try the same sum with `area_ha_print` and Python will complain, because a function without a `return` hands back `None`: nothing. If a tool needs to feed its result into the next tool, it must return it. `Print` is for your eyes; `return` is for the recipe.

Parameters with default values

Some model parameters have sensible defaults: the Buffer tool defaults to "Full" side type, for instance. Functions can do the same:

```

def is_clear(cloud_percent, limit=20):
    return cloud_percent < limit

print(is_clear(12))          # True: uses the default limit of 20
print(is_clear(12, limit=10)) # False: we overrode the default

```

The second call shows another useful habit: **naming the argument** when you pass it (`limit=10`). This is called a *keyword argument*. It makes calls self-explanatory, and you'll see it everywhere in GIS libraries. An Earth Engine export, for instance, is called with named arguments like `image=...`, `region=...`, `scale=10`. When you see `name=value` inside a call's brackets, it's simply filling in a named blank.

What happens inside stays inside

Variables created inside a function exist only while the function runs. Once it returns, they vanish. The `ndvi` label inside `calculate_ndvi` doesn't exist outside it:

```
def calculate_ndvi(nir, red):
    ndvi = (nir - red) / (nir + red)
    return ndvi

calculate_ndvi(0.41, 0.08)
print(ndvi)          # Error: name 'ndvi' is not defined
```

This is deliberate, and it's a good thing. A tool's internal workings shouldn't spill onto your desk. A ModelBuilder model's intermediate layers don't clutter your map either. The only thing that comes out is what you return.

Leave a note for the next person

The first line inside a function can be a description in triple quotes, called a **docstring**. It's what the tool's help panel would say:

```
def calculate_ndvi(nir, red):
    """Return NDVI from near-infrared and red reflectance (range -1 to 1)."""
    return (nir - red) / (nir + red)
```

Six months from now, the "next person" is usually you.

The idea that unlocks Earth Engine: tools can be handed to other tools

Here is the most important idea in this part, and it's worth reading twice.

A function is itself a value, just like a number or a string. That means you can **hand a function to another function**, the way you'd hand a tool to a batch processor and say "run this on every input."

Python has a built-in called `map` that does exactly this. You give it a function and a list, and it applies the function to every item:

```
def to_hectares(area_m2):
    return area_m2 / 10000

areas_m2 = [48250, 12000, 90500]
areas_ha = list(map(to_hectares, areas_m2))
print(areas_ha)          # [4.825, 1.2, 9.05]
```

Notice that we wrote `to_hectares` **without brackets**. With brackets, `to_hectares(...)` means *run the tool now*. Without brackets, `to_hectares` means *the tool itself*: we're handing it over, not using it. `map` then does the running, once per item.

Why does this matter so much? Because in Earth Engine, you can't loop over satellite images with `for` (Part 9 explains why). Instead, you write a function that does something to *one* image, and hand that function to the image collection's own `.map()`. Earth Engine then applies it to every image, on Google's computers, in parallel. The shape is identical to what you just wrote:

```
# A preview of Part 9, just to see the shape:
def add_ndvi(image):
    ndvi = image.normalizedDifference(["B8", "B4"]).rename("NDVI")
    return image.addBands(ndvi)
```

```
with_ndvi = images.map(add_ndvi)    # the tool, handed over, not called
```

A function that takes *one* item and returns *one* item, handed to `map`. If that pattern makes sense to you now, the hardest idea in Earth Engine is already behind you.

Tiny tools without a name: `lambda`

For very small, one-line tools, Python lets you write a function without `def` and without a name, called a **lambda**:

```
areas_ha = list(map(lambda a: a / 10000, areas_m2))
```

`lambda a: a / 10000` means *a tool that takes a and returns a / 10000*. You'll see lambdas in Earth Engine examples online. They're a shorthand, never a necessity. A named `def` function is always allowed and usually easier to read.

What you now own

- A **function** is a named, reusable recipe: parameters in, return value out. It's a ModelBuilder model written as text.
- `def` builds the tool, and calling it by name with brackets runs it.
- `return` hands a value back for further use; `print` only shows it.
- Parameters can have **defaults**, and arguments can be passed **by name** (`scale=10`).
- Functions are values: they can be **handed to other functions**. `map(tool, items)` runs a tool on every item. This is exactly how Earth Engine's `.map()` works.

Objects and modules

Things that know things, and know how to do things. After this part, you'll be able to read any tool's documentation and understand what it's telling you.

A layer knows things, and can do things

Right-click a layer in QGIS or ArcGIS and open its *Properties*. You'll see what the layer **knows** about itself: its name, its coordinate system, its extent, how many features it has, where its file lives. Now look at the right-click menu itself. That's what the layer **can do**: zoom to itself, open its attribute table, export, be styled, be filtered.

Every "thing" in modern software is like this: a bundle of **information** plus **actions** that belong to it. Python calls such a bundle an **object**.

- The information an object holds is called its **attributes** (or properties).
- The actions it can perform are called its **methods**.

And here's the surprise: you've been using objects all along. Every value in Python is an object. A string knows things and can do things:

```
name = "ghaziabad"
print(name.upper())      # GHAZIABAD: an action the string can do
print(name.capitalize()) # Ghaziabad
print(name.startswith("gh")) # True
```

A list is an object too. `districts.append("Hapur")` from Part 2 was calling the list's `append` method.

The dot: "this thing's ..."

The single piece of grammar that makes objects work is the **dot**. Read it as the possessive 's:

- `name.upper()` is *name's upper-case action*.
- `layer.name()` is *the layer's name*.
- `image.select("B4")` is *the image's select action, choosing band B4*.

If there are brackets after the dot-word, it's an action (a method, which is a function that belongs to the object). If there are no brackets, it's usually a piece of information (an attribute). That's why `append` has brackets: it's something the list *does*.

This is the key to reading GIS code. Look at any line from ArcPy, PyQGIS or Earth Engine and read the dots aloud as 's:

```
layer.featureCount()      # the layer's feature count
image.bandNames()         # the image's band names
collection.filterDate("2025-08-01", "2025-09-01") # the collection's date filter
```

Suddenly they're not cryptic. They're sentences.

Chaining: a pipeline of actions

When a method returns a new object, you can immediately call a method on *that*. This is called **chaining**, and Earth Engine code is full of it:

```
images = (ee.ImageCollection("COPERNICUS/S2_SR_HARMONIZED")
          .filterBounds(district)
          .filterDate("2025-08-01", "2025-09-01")
          .filter(ee.Filter.lt("CLOUDY_PIXEL_PERCENTAGE", 20)))
```

Don't worry about the details yet; just see the shape. Read it top to bottom: *take the Sentinel-2 collection, and from that keep only images over the district, and from that only August, and from that only the ones under 20% cloud*. Each dot takes the result of the line above and does one more thing. It's pseudocode steps 2, 3 and 4 from Part 1, written as a chain. (The outer brackets just let the chain spread over several lines for readability.)

Blueprints and the things built from them

Where do objects come from? Every object is built from a **blueprint**, which Python calls a **class**. The class describes what information the object will hold and what actions it will have; each object built from it is an **instance** with its own particular values.

You know this relationship from geodatabases. A *feature class* defines the schema: the fields, the geometry type. Each *feature* in it is one instance with its own values. Many features, one blueprint.

In Python, you create an instance by calling the class's name like a function:

```
# Earth Engine: build a point object from its blueprint
point = ee.Geometry.Point([77.45, 28.67]) # longitude, latitude

# PyQGIS: build a layer object from a file
layer = QgsVectorLayer("villages.shp", "Villages", "ogr")
```

You won't need to write your own classes for GIS scripting. Almost everything you need already exists. But recognising that `ee.Geometry.Point(...)` or `QgsVectorLayer(...)` means "*build me a new one of these*" makes the documentation much easier to follow.

Modules: toolboxes you open when you need them

Python out of the box knows the basics: numbers, strings, lists, `if`, `for`. Everything else lives in **modules**, which are collections of ready-made functions and classes, grouped by topic. It's exactly the ArcToolbox idea: you don't have every tool in front of you at once. You have toolboxes, and you open the one you need.

To use a module, you **import** it:

```
import math
print(math.sqrt(10980 * 10980)) # 10980.0
print(math.pi)
```

After `import math`, everything in that toolbox is reachable through the dot: `math.sqrt` means *math's square-root tool*. The dot works for modules exactly as it does for objects.

There are three ways you'll see imports written, and they all mean roughly the same thing:

```
import datetime # use as datetime.date(...)
import geopandas as gpd # a short nickname: gpd.read_file(...)
from pathlib import Path # take one tool out: Path(...)
```

The nickname form (`as`) is a strong convention for some libraries. The world writes `import pandas as pd` and `import numpy as np`, and following the convention makes your code look like the documentation.

For your work, the important modules are:

Module	Where it lives	What it gives you
<code>arcpy</code>	inside ArcGIS Pro	every geoprocessing tool, data access, map documents
<code>qgis.core</code> , <code>processing</code>	inside QGIS	layers, features, the Processing toolbox
<code>ee</code>	install <code>earthengine-api</code>	Earth Engine: images, collections, reducers
<code>geemap</code>	install <code>geemap</code>	interactive maps for Earth Engine in notebooks
<code>pandas</code>	usually pre-installed	tables: read CSV/Excel, filter, group, summarise
<code>pathlib</code> , <code>os</code>	built in	files and folders

Some modules come with Python; others are **packages** you install once. In Colab, you install with a line starting with an exclamation mark, which means "run this as a system command, not as Python":

```
!pip install geemap
```

(Colab already has `earthengine-api` and `pandas`, so often you won't need even this.)

Reading documentation without fear

Here is where this part pays off. Documentation for any GIS library is, at heart, a list of classes, their methods, and the parameters each method takes. With what you now know, you can read it.

Take this entry, the kind you'll find in the Earth Engine API reference:

```
ee.Image.normalizedDifference(bandNames)
    Computes the normalized difference between two bands.
    Arguments:
        this:image (Image)    The input image.
        bandNames (List)     A list of names specifying the bands to use.
    Returns: Image
```

Translate it with your new vocabulary: *this is a method that belongs to Image objects. It takes one parameter, `bandNames`, which should be a list. It hands back (returns) a new Image. So: `image.normalizedDifference(["B8", "B4"])`. You just wrote a line of Earth Engine from its documentation, without an example to copy.*

Python can also tell you about objects directly, from inside Colab:

```
help(len)          # the built-in manual page
dir("hello")       # list everything a string can do
```

And typing an object's name followed by a dot and then pressing Tab in Colab pops up the list of its methods. That's the quickest way to explore.

Think about it. Why does it matter to know that a method **returns** something? Because it tells you what you can chain next. `normalizedDifference` returns an `Image`, so you can immediately call `Image` methods on the result, like `.rename("NDVI")`. The "Returns" line in documentation is what makes chaining possible.

What you now own

- An **object** bundles information (attributes) with actions (methods). Every value in Python is one.
- The **dot** means 's. `image.select("B4")` is the *image's select action*.
- **Chaining** strings actions into a readable pipeline, each step working on the last one's result.
- A **class** is a blueprint; `SomeClass(...)` builds a new instance, like a feature from a feature class.
- **Modules** are toolboxes opened with `import`. `arcpy`, `ee`, `processing` and `pandas` are yours.
- Documentation is a map of classes, methods, parameters and return types, and you can now read it.

When things break

Errors aren't failures. They're the literal assistant telling you exactly where it got confused. Learning to read them is what separates people who get stuck from people who get unstuck.

A change of attitude first

Every programmer, at every level, sees error messages constantly. The difference between a beginner and an expert isn't that the expert makes fewer mistakes. It's that the expert reads the error calmly and fixes it in a minute, while the beginner sees red text and feels something has gone badly wrong.

Nothing has gone wrong. Remember the literal assistant: when an instruction doesn't make sense, it stops and tells you. An error message is that report. It's the most helpful thing Python ever says to you, because it tells you **what** it couldn't do, and **where**.

Think of it like a GPS saying "route not possible: road closed at km 14." It isn't the GPS failing. It's the GPS telling you precisely where to look.

Anatomy of an error message

When Python stops, it prints a **traceback**. Here's a typical one:

```
Traceback (most recent call last):
  File "<ipython-input-7>", line 9, in <cell line: 9>
    label_village(villages[3])
  File "<ipython-input-7>", line 3, in label_village
    if village["NDVI"] > 0.5:
  KeyError: 'NDVI'
```

It looks intimidating, but there's a simple way to read it: **start from the bottom**.

- **The last line** is the headline: the type of error, then a short message. `KeyError: 'NDVI'` means you asked a dictionary for a key called 'NDVI', and it doesn't have one.
- **The lines just above** show exactly which line of code was running when it happened: `if village["NDVI"] > 0.5:`, line 3.
- **Further up** is the trail of how Python got there: line 9 called the function `label_village`, and inside that function, line 3 failed. It's the breadcrumb trail.

And the fix, here? Our dictionaries use the key `"ndvi"`, in lowercase. Python is case-sensitive (Part 2). One character.

The errors you'll meet most

About ten error types cover nearly everything a beginner will see. Once you know what each headline *means*, most fixes become obvious.

Headline	What it's really saying	Usual cause
<code>SyntaxError</code>	"I can't even read this line."	missing colon, bracket or quote
<code>IndentationError</code>	"The shape of the code is off."	mixed or missing indentation after :
<code>NameError</code>	"I don't know that name."	typo, wrong capitals, or used before it was labelled
<code>TypeError</code>	"These things don't go together."	adding text to a number; wrong number of arguments
<code>KeyError</code>	"That dictionary has no such key."	field name misspelled or different case
<code>IndexError</code>	"There's no item at that position."	asking for item 5 of a 5-item list (last is index 4)
<code>AttributeError</code>	"This object can't do that."	method name misspelled, or the object isn't what you think
<code>ValueError</code>	"Right type, impossible value."	<code>int("forty")</code>
<code>FileNotFoundError</code>	"Nothing is at that path."	wrong folder, wrong extension, or a typo in the path
<code>ModuleNotFoundError</code>	"I don't have that toolbox."	package not installed, or you're in the wrong environment

Earth Engine adds one of its own: `EEException`, meaning the error happened on Google's side, not in your Python. Its message usually names the problem plainly (a band that doesn't exist, a collection that's empty, a computation that's too large). The same bottom-up reading applies.

Think about it. `AttributeError: 'NoneType' object has no attribute 'addBands'`. Using only the table, what's going on? Some variable you expected to be an image is actually *None*, meaning **nothing**. The usual culprit is a function that forgot to *return* (Part 5), so it handed back nothing. The error appears on one line, but the cause is in another. That's common, and the traceback's breadcrumbs help you walk back to it.

A method for getting unstuck

When code doesn't work, and especially when it runs without an error but gives the wrong answer, use this sequence. It works at every level of experience.

- 1. Read the last line of the error, slowly.** Say it out loud in plain words. Half of all bugs are solved here.
- 2. Look at the exact line it points to.** Check spelling, capitals, colons, brackets, quotes.
- 3. Check your assumptions with `print` and `type`.** Most bugs are a variable that isn't what you think it is. So ask:

```
print(village)           # what's actually in here?
print(type(area))        # is it a number, or text that looks like a number?
print(len(images))       # did the filter leave anything at all?
```

In Earth Engine, the equivalent check is asking for a small piece of information back, such as `print(images.size().getInfo())` to see how many images survived your filters. Zero is a very common answer, and a very useful one to know.

- 4. Make the problem smaller.** If a loop over 70 districts fails, run it on one district. If a ten-step chain fails, run the first three steps and look at the result. Then four. The step where it breaks is the step to fix.

5. Explain it to someone, or something. Programmers talk about "rubber duck debugging": explaining your code line by line to a rubber duck on the desk. Halfway through explaining, you hear yourself say the wrong thing. Writing the logic out as pseudocode again (Part 1) works the same way.

6. Search the exact error message. Copy the last line, minus your own variable names, into a search engine. Someone has had this exact problem before, very probably with the same library. Earth Engine's developer forum and GIS Stack Exchange are full of answered questions.

Carrying on when one item fails

In batch work there's a special situation: you're processing 200 files, and file 137 is corrupt. By default, the error stops the whole loop, and files 138 to 200 never run. Python lets you catch the error, note it, and carry on, with `try` and `except`:

```
failed = []

for path in raster_paths:
    try:
        process_raster(path)
    except Exception as err:
        print(f"Skipped {path}: {err}")
        failed.append(path)

print(f"Done. {len(failed)} files need a look.")
```

Read it as: *try these steps; if any error happens, don't stop, run the `except` block instead*. This is the scripted version of a batch tool that logs failures and keeps going.

One caution: `try/except` is for **expected, isolated** failures in batch runs. Don't wrap everything in it to make errors disappear. An error you hide is an error you can't fix. While you're developing, let the errors show.

What you now own

- An error is a precise report of where the literal assistant got confused. It's not a verdict on you.
- Read a traceback **from the bottom**: the headline, then the line, then the trail.
- About ten error types cover most problems, and each headline tells you where to look.
- The unsticking method: read, inspect the line, check assumptions with `print/type`, shrink the problem, explain it, search it.
- `try/except` lets batch loops survive a bad item. Use it deliberately, not to hide things.

Files, folders and tables

Your work lives in files. Here's how Python finds them, reads them, and writes new ones, and a first look at pandas, the spreadsheet engine inside Python.

A path is an address

Every file on a computer has an address, called its **path**: the chain of folders you'd click through to reach it, ending in the file's name.

```
Windows:  C:\GIS\Meerut\rasters\ndvi_2025.tif
Mac/Linux and Colab: /content/drive/MyDrive/GIS/ndvi_2025.tif
```

In Python, a path is just a string. But Windows paths bring one trap. In Python strings, the backslash `\` has a special meaning (for example, `\n` means "new line"). So `"C:\GIS\new_data"` quietly gets mangled. The cure is to put an `r` before the quote, which makes it a **raw string** in which backslashes are just backslashes:

```
folder = r"C:\GIS\Meerut\rasters"    # safe on Windows
```

You'll see `r"..."` all over ArcPy scripts for exactly this reason.

Paths come in two flavours. An **absolute** path starts from the very top (`C:\...` or `/...`) and works from anywhere. A **relative** path, like `rasters/ndvi_2025.tif`, is measured from the folder your script is currently "standing in". If a relative path works in one place and not another, that's usually why.

pathlib: paths as objects

Rather than gluing strings together, Python's built-in `pathlib` module treats a path as an object (Part 6) that knows useful things about itself:

```
from pathlib import Path

folder = Path("/content/drive/MyDrive/GIS/rasters")
tif = folder / "ndvi_2025.tif"    # the / joins path pieces

print(tif.name)      # ndvi_2025.tif
print(tif.stem)      # ndvi_2025, the name without extension
print(tif.suffix)    # .tif
print(tif.exists())  # True or False
```

The `/` between a folder and a name is `pathlib`'s way of saying "inside". It works the same on Windows and everywhere else, so you never have to worry about which slash to use.

Finding all the files: the start of every batch job

The most common first step in real scripting is "get me every file of a certain kind in this folder." That's what `glob` does. It matches a pattern, where `*` means anything:

```
tifs = list(folder.glob("*.tif"))          # every .tif in the folder
tifs_2025 = list(folder.glob("*_2025.tif")) # only those ending _2025
every_tif = list(folder.rglob("*.tif"))    # also look inside sub-folders

for tif in tifs:
    print(tif.name)
```

Now join that to Part 4. A list of files plus a for loop is a batch process over a folder, the thing you'd otherwise do with *Batch* and fifty rows of clicking:

```
for tif in folder.glob("*.tif"):
    out = folder / "clipped" / f"{tif.stem}_clip.tif"
    print(f"{tif.name} → {out.name}")
    # clip_raster(tif, out) ← whichever tool does the real work
```

Opening a file and closing it again

To read a plain text file, you open it, read it, and close it. Python's `with` statement handles the closing automatically, even if something goes wrong halfway:

```
with open("notes.txt") as f:
    text = f.read()

with open("log.txt", "w") as f:          # "w" = write (replaces the file)
    f.write("Processing finished.\n")
```

The indented block under `with` is the time the file is open. When the indentation ends, the file is closed. It's the same indentation grammar you already know, with a new meaning: *while this is open, do these things*.

CSV files: the attribute table on disk

CSV (comma-separated values) is the plainest table format there is: one row per line, fields separated by commas. It's how many GIS tools export attribute tables and how Earth Engine exports statistics.

Python's `csv` module can read each row as a **dictionary**, which means every CSV turns straight into the list-of-dictionaries attribute table from Part 2:

```
import csv

with open("villages.csv") as f:
    villages = list(csv.DictReader(f))

print(villages[0])
# {'name': 'Rasoolpur', 'population': '4120', 'ndvi': '0.58'}
```

Look closely at the numbers. `'4120'` and `'0.58'` are in quotes, so they're **strings**. A CSV has no idea what type anything is; everything arrives as text. This is the Part 2 type trap, in its natural habitat. Convert before you calculate:

```
for v in villages:
    v["ndvi"] = float(v["ndvi"])
    v["population"] = int(v["population"])
```

pandas: a whole attribute table as one object

Working row by row with loops is excellent for understanding. For day-to-day table work, though, Python has **pandas**, a library that holds an entire table as one object called a **DataFrame**. If a list of dictionaries is an attribute table you built by hand, a `DataFrame` is an attribute table with every tool

already attached.

```
import pandas as pd

df = pd.read_csv("villages.csv")  # types are detected automatically
print(df.head())                 # first five rows
print(df.shape)                  # (rows, columns)
print(df["ndvi"].mean())          # average of one column
```

Everything you did with loops in Part 4 has a one-line pandas equivalent. And because you've built them by hand, you understand what these lines are doing:

```
healthy = df[df["ndvi"] > 0.5]  # filter (Select by Attributes)
df["status"] = df["ndvi"].apply(lambda n: "healthy" if n > 0.5 else "stressed") # calculate field
by_district = df.groupby("district")["ndvi"].mean() # summary statistics by group
healthy.to_csv("healthy_villages.csv", index=False) # export
```

Notice `df["ndvi"] > 0.5` inside the square brackets. It produces a column of True/False values (one per row), and the outer `df[...]` keeps only the rows marked True. That's a selection set, exactly as a GIS highlights selected rows. `apply` is `map` from Part 5, working on a column. `groupby` is *Summary Statistics with a case field*.

pandas also reads and writes Excel (`pd.read_excel`, `df.to_excel`), which is often how results need to leave your hands.

A step further, when you're ready: `geopandas` is pandas with a geometry column added.

`gpd.read_file("villages.shp")` gives you a `DataFrame` where each row also carries its polygon, with reprojection, spatial joins and area calculation built in. It's the natural next library after this course, and everything you know about `DataFrames` carries straight over.

JSON and GeoJSON: dictionaries on disk

One last format, because it's everywhere in web and cloud GIS. **JSON** is a text format that looks almost exactly like Python dictionaries and lists. **GeoJSON** is JSON with an agreed structure for features and geometries:

```
{ "type": "Feature",
  "properties": { "name": "Rasoolpur", "ndvi": 0.58 },
  "geometry": { "type": "Point", "coordinates": [ 77.61, 29.02 ] } }
```

You can read that as nested dictionaries straight away. Python's `json` module turns it into real ones:

```
import json
with open("villages.geojson") as f:
    data = json.load(f)

for feature in data["features"]:
    print(feature["properties"]["name"])
```

Earth Engine speaks this shape too. When you ask it for a feature's information, what comes back is essentially GeoJSON as Python dictionaries.

Colab and Google Drive

Colab runs on Google's computers, so it can't see your laptop's folders. The usual bridge is Google Drive. Mount it once per session:

```
from google.colab import drive
drive.mount("/content/drive")
```

```
| # your files now appear under /content/drive/MyDrive/
```

Earth Engine can export results straight to Drive as well, which makes Drive the natural meeting point: Earth Engine writes there, Colab reads from there, and you can open the same files in QGIS or ArcGIS on your desktop.

What you now own

- A **path** is a file's address. Use `r"..."` for Windows paths, and `pathlib.Path` to join and inspect them.
- `folder.glob("*.tif")` + a for loop = a batch process over a folder.
- `with open(...)` opens and safely closes files.
- CSVs read as lists of dictionaries, **with every value as text**, so convert before you calculate.
- **pandas** DataFrames are attribute tables with tools attached: filter, calculate, group, export.
- JSON and GeoJSON are dictionaries and lists on disk.

Into the tools

Earth Engine, Google Cloud, QGIS and ArcGIS Pro. Four dialects of one language. Everything from Parts 0 to 8 now gets put to work.

One grammar, four dialects

Before we visit each tool, here's the picture that ties them together. The *ideas* you've learned stay exactly the same. Only the names of things change.

The idea	Plain Python	Earth Engine	QGIS	ArcGIS Pro
a feature's attributes	dictionary	<code>feature.get("name")</code>	<code>feature["name"]</code>	a cursor row
many features	list of dicts	<code>ee.FeatureCollection</code>	<code>layer.getFeatures()</code>	<code>arcpy.da.SearchCursor</code>
a grid of values	list of lists	<code>ee.Image</code>	<code>QgsRasterLayer</code>	<code>arcpy.Raster</code>
do it for each	for loop	<code>.map(function)</code>	for loop	for loop
keep only some	if inside a loop	<code>.filter(...)</code>	<code>selectByExpression</code>	where clause
summarise	accumulator	<code>.reduce... (ee.Reducer...)</code>	processing tools	Summary Statistics
run a tool	call a function	call a method	<code>processing.run(...)</code>	<code>arcpy.analysis.Buffer(...)</code>

Three of the four tools (QGIS, ArcGIS, plain Python) run your code *on your machine*, where everything from the earlier parts applies exactly. Earth Engine is the odd one out, and understanding why is the biggest single idea in this document. So we start there.

Earth Engine

Getting in the door

Earth Engine is a Google service, so it needs to know *who you are* and *which Google Cloud project* your work belongs to. Every Earth Engine user now works inside a registered Cloud project (non-commercial and research use is free, but still needs a registered project). Once the project exists, the start of every notebook looks like this:

```
import ee
ee.Authenticate() # "who are you?": opens a Google sign-in
ee.Initialize(project="your-project-id") # "which project is this work for?"
```

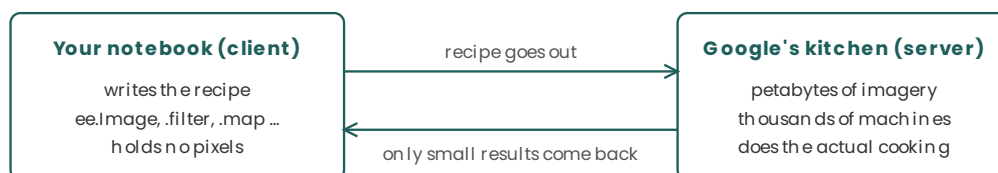
Authenticate is showing your ID card; Initialize is saying which office you're working in today. You'll run these two lines at the top of every session.

The kitchen: why Earth Engine code is different

Here is the idea that everything else in Earth Engine rests on.

When you write `2 + 3` in Python, your computer does the arithmetic, right there. But a year of Sentinel-2 images over a district is hundreds of gigabytes. Your laptop can't download that, let alone compute on it. So Earth Engine works differently:

Your Python code does not process any data. It writes a recipe and sends it to Google's kitchen. Google's computers do the cooking, near where the data is stored, and only send back what you ask to see.



.getInfo(), a map tile, or an export is how a result travels back.

Google's documentation calls these the **client** (your notebook) and the **server** (Google's machines). Every Earth Engine object, anything starting with `ee.`, is really a *line in a recipe*, not the data itself. Watch:

```
x = ee.Number(5)
y = x.add(3)
print(y)           # prints a description of the recipe, not 8
print(y.getInfo()) # 8: "cook this and send me the answer"
```

`y` is an instruction: "take 5, add 3." It isn't 8 until you ask the kitchen to cook it. `.getInfo()` does exactly that: it sends the recipe, waits, and brings the answer back as an ordinary Python value.

From this one idea, three rules follow. They explain nearly every "why doesn't this work?" moment in Earth Engine:

Rule 1: Use Earth Engine's methods, not Python's operators, on ee objects. `x + 3` doesn't work on an `ee.Number`; `x.add(3)` does. Python's `+` only knows how to add things *on your computer*. `.add()` adds a step to the recipe.

Rule 2: Don't use Python's for or if on server data. A `for` loop runs on your computer, but the images are in the kitchen. Instead of looping, you hand a function to `.map()` (Part 5!) and the kitchen applies it to every item, in parallel, on its own machines. Instead of `if ndvi > 0.5`, you ask a question of *every pixel at once*: `ndvi.gt(0.5)` produces a new image where each pixel is 1 (yes) or 0 (no). The decision is still there. It's just cooked in the kitchen.

Rule 3: Use .getInfo() sparingly. Each call is a round trip to the kitchen, and asking for something large (like every pixel) will be slow or fail. Ask for small things: a count, a list of dates, a single average. Get big results out through maps (they're drawn tile by tile) and exports (covered just below).

Think about it. Why is `.map()` better than a loop here, even ignoring where the data lives? A loop does items one after another. `.map()` tells the kitchen "do this to all of them", and the kitchen, with thousands of machines, can do them all at the same time. You're not just writing it differently. You're letting the work be split up.

The main ingredients

Earth Engine has a small set of object types. You'll recognise each from GIS:

- `ee.Image`: one raster, with one or more named bands. One Sentinel-2 scene.
- `ee.ImageCollection`: a stack of images. All Sentinel-2 scenes, ever.
- `ee.Geometry`: a point, line or polygon.
- `ee.Feature`: a geometry plus a dictionary of properties. One village.
- `ee.FeatureCollection`: many features. A village layer.
- `ee.Reducer`: a summarising operation (mean, median, sum, count...).

And they're made from blueprints (Part 6):

```
point = ee.Geometry.Point([77.70, 28.98])    # longitude first, then latitude
s2 = ee.ImageCollection("COPERNICUS/S2_SR_HARMONIZED")
districts = ee.FeatureCollection("FAO/GAUL/2015/level2")
meerut = districts.filter(ee.Filter.eq("ADM2_NAME", "Meerut"))
```

The long names in quotes are dataset IDs from the Earth Engine Data Catalog. Each dataset's catalog page lists its ID, its bands and its properties. You'll keep that catalog open in a browser tab constantly. Your own shapefiles can be uploaded as **assets** and loaded the same way, as `ee.FeatureCollection("projects/your-project-id/assets/villages")`.

The rhythm: filter → map → reduce

Almost every Earth Engine analysis follows the same three-beat rhythm. You've already met all three beats as loop patterns in Part 4.

Filter: narrow down the collection (Part 4, Pattern 2):

```
images = (s2.filterBounds(meerut)
          .filterDate("2025-08-01", "2025-09-01")
          .filter(ee.Filter.lt("CLOUDY_PIXEL_PERCENTAGE", 20)))

print(images.size().getInfo())    # how many survived? A small, cheap question.
```

Map: do something to every image (Part 4, Pattern 1, via Part 5's handing-over):

```
def add_ndvi(image):
    ndvi = image.normalizedDifference(["B8", "B4"]).rename("NDVI")
    return image.addBands(ndvi)

images = images.map(add_ndvi)
```

The function takes *one* image and returns *one* image. It never loops. It only describes what to do to a single image, and `.map()` handles the "for each". (For Sentinel-2, B8 is near-infrared and B4 is red.)

Reduce: summarise (Part 4, Pattern 3, the accumulator):

```
# collapse the stack through time: one median value per pixel
ndvi_aug = images.select("NDVI").median().clip(meerut)

# collapse the pixels through space: one average for the whole district
stats = ndvi_aug.reduceRegion(
    reducer=ee.Reducer.mean(),
    geometry=meerut.geometry(),
    scale=10,
    maxPixels=1e10
)
print(stats.getInfo())    # e.g. {'NDVI': 0.54}, a small dictionary comes home
```

Two kinds of reducing are happening, and it's worth seeing them clearly. `.median()` reduces *through time*: many images become one. `.reduceRegion()` reduces *through space*: many pixels become one number. Notice the keyword arguments (Part 5): `scale=10` means *work at 10-metre pixels*, which is Sentinel-2's native resolution for these bands.

Pixel-level decisions follow Rule 2. Here's the "healthy vs stressed" question from Part 3, asked of every pixel at once:

```
healthy_mask = ndvi_aug.gt(0.5) # 1 where NDVI > 0.5, else 0
healthy_only = ndvi_aug.updateMask(healthy_mask) # hide everything else
```

Seeing it on a map

Numbers are good; maps are better. The `geemap` library gives you an interactive map inside the notebook:

```
import geemap

m = geemap.Map()
m.centerObject(meerut, 10)
m.addLayer(ndvi_aug, {"min": 0, "max": 0.8,
                    "palette": ["#a6611a", "#f5f5a0", "#1a9641"]}, "NDVI Aug 2025")
m.addLayer(meerut.style(color="black", fillColor="00000000"), {}, "Meerut boundary")
m
```

The dictionary in the middle (`min`, `max`, `palette`) is the symbology: the same stretch and colour ramp you'd set in layer properties. The map is drawn by the kitchen tile by tile as you pan and zoom, which is why it can display a whole district's worth of computation without anything large coming down to your notebook.

Getting results out: exports

When you need an actual file (a GeoTIFF to open in QGIS, or a table for Excel), you **export**. Exports are sent to the kitchen as a *task* that runs in the background, because big jobs can take minutes or hours:

```
task = ee.batch.Export.image.toDrive(
    image=ndvi_aug,
    description="ndvi_meerut_aug2025",
    folder="GEE_exports",
    region=meerut.geometry(),
    scale=10,
    maxPixels=1e10
)
task.start()
print(task.status()["state"]) # READY → RUNNING → COMPLETED
```

Every blank is a named parameter, exactly the same as filling in a geoprocessing tool's dialog. Tables export the same way with `Export.table.toDrive(collection=..., fileFormat="CSV")`. You can also watch tasks in the *Tasks* tab of the Earth Engine Code Editor. There are equivalents for exporting to Cloud Storage and to your Earth Engine assets, and the next section is about the first of those.

Google Cloud

You don't need to become a cloud engineer. But because Earth Engine now lives inside Google Cloud, a few ideas are worth owning properly.

A project is a workspace with a key-ring and a meter. Everything in Google Cloud belongs to a *project*: a named container that holds resources (storage, Earth Engine access), decides *who* is allowed to use them, and tracks usage. When you called `ee.Initialize(project="...")`, you were saying *charge and authorise this work under this workspace*. Your project ID is shown on the Google Cloud Console's dashboard.

APIs are doors that start closed. Each Google service must be switched on for a project before code can use it (for example, the Earth Engine API). If you get an error saying an API "has not been used in project ... or it is disabled", the fix is simply enabling it in the Console.

Authentication is "who is asking?" Your code acts *as someone*. In Colab, that's you, after signing in:

```
from google.colab import auth
auth.authenticate_user()
```

For automated jobs that run without a person present, Google uses **service accounts**, which are robot identities with their own permissions. You'll only need those if your work becomes a scheduled pipeline.

Cloud Storage is a very large, very reliable shared drive. Its folders-at-the-top are called **buckets**, and its files are called **objects**. It's where large Earth Engine exports usually go in professional work, because it's faster than Drive for big files, and other cloud tools can read straight from it:

```
task = ee.batch.Export.image.toCloudStorage(
    image=ndvi_aug,
    description="ndvi_meerut_aug2025_gcs",
    bucket="your-bucket-name",
    fileNamePrefix="exports/ndvi_meerut_aug2025",
    region=meerut.geometry(),
    scale=10,
    maxPixels=1e10
)
task.start()
```

And from Python you can list and download what's there, using Google's storage library, in the same object-and-method grammar as everything else:

```
from google.cloud import storage

client = storage.Client(project="your-project-id")
for blob in client.list_blobs("your-bucket-name", prefix="exports/"):
    print(blob.name, blob.size)

bucket = client.bucket("your-bucket-name")
bucket.blob("exports/ndvi_meerut_aug2025.tif").download_to_filename("ndvi.tif")
```

Read it with Part 6's eyes: a *client* object (your connection), which can list *blobs* (objects in the bucket); a *bucket* object, whose *blob* can download itself to a file. Nothing new, just new nouns.

QGIS

In QGIS, open **Plugins → Python Console**. Python here is already connected to the running application, and a special object called *iface* (the *interface*) is waiting for you. It's your handle on the QGIS window itself.

Layers and features

Select a vector layer in the Layers panel, then:

```

layer = iface.activeLayer()
print(layer.name(), layer.featureCount(), layer.crs().authid())

for field in layer.fields():
    print(field.name(), field.typeName())

```

Now loop through the features: the attribute table, row by row. Each feature behaves like a dictionary for its attributes, plus a geometry:

```

for f in layer.getFeatures():
    area_ha = f.geometry().area() / 10000
    print(f["name"], round(area_ha, 2))

```

(Area is in the layer's CRS units, so this is only hectares if the layer uses a metre-based projection. That's Part 1's "name the exact thing", showing up in real life.)

Selecting uses the very same expression language as the *Select by Expression* dialog:

```

layer.selectByExpression('"ndvi" > 0.5')
print(layer.selectedFeatureCount())

```

Loading a layer from disk is building an object from a blueprint and adding it to the map:

```

villages = iface.addVectorLayer(r"C:\GIS\Meerut\villages.shp", "Villages", "ogr")

```

Running Processing tools

Every tool in the Processing Toolbox can be run from Python with `processing.run`. It takes the tool's ID and a **dictionary** of parameters, which is Part 2's dictionary doing real work:

```

result = processing.run("native:buffer", {
    "INPUT": villages,
    "DISTANCE": 500,
    "OUTPUT": "memory:"          # a temporary layer, like a scratch output
})
buffered = result["OUTPUT"]
QgsProject.instance().addMapLayer(buffered)

```

It even hands back a dictionary of results. And here's the best learning shortcut in QGIS: open any tool's dialog, fill it in the normal way, and use **Advanced** → **Copy as Python Command**. QGIS writes the `processing.run(...)` line for you, with the correct tool ID and parameter names. The *Processing* → *History* panel also keeps the Python version of every tool you've run by clicking. Your clicks, translated.

Batch over a folder

Put Part 8 and this together, and you have a genuine time-saver:

```

from pathlib import Path

folder = Path(r"C:\GIS\Meerut\rasters")
boundary = r"C:\GIS\Meerut\district.shp"

for tif in folder.glob("*.tif"):
    processing.run("gdal:cliprasterbymasklayer", {
        "INPUT": str(tif),
        "MASK": boundary,
        "OUTPUT": str(folder / "clipped" / f"{tif.stem}_clip.tif")
    })
    print("done:", tif.name)

```

(Create the `clipped` folder first. And note `str(...)`: some tools want paths as plain strings rather than `Path` objects, which is a small conversion from Part 2.)

ArcGIS Pro and ArcPy

In ArcGIS Pro, open the **Python window** (Analysis tab → Python), or create a **Notebook** from the Insert tab. `arcpy` is ready to use.

Workspace and listing data

ArcPy leans heavily on a *workspace*: the current geodatabase or folder, so you can refer to datasets by name instead of full path. Settings like this live in `arcpy.env`, an object of environment settings (the same as *Environments* in a tool dialog):

```
import arcpy

arcpy.env.workspace = r"C:\GIS\Meerut\Meerut.gdb"
arcpy.env.overwriteOutput = True      # allow outputs to replace old ones

for fc in arcpy.ListFeatureClasses():
    count = arcpy.management.GetCount(fc)[0]
    print(fc, count)
```

`ListFeatureClasses()` hands back a Python list of names. There are siblings: `ListRasters()`, `ListFields(fc)`, `ListTables()`. Each is the start of a batch job.

Running tools

Every tool in the toolbox is a function, filed under its toolbox's short name: `arcpy.analysis`, `arcpy.management`, `arcpy.conversion`, and so on. Parameters come in the same order as in the tool dialog:

```
arcpy.analysis.Buffer("villages", "villages_buf500", "500 Meters")
arcpy.analysis.Clip("roads", "district_boundary", "roads_meerut")
```

The ArcGIS equivalent of QGIS's shortcut: run a tool once by hand, then open **History** in the Catalog pane, right-click the entry and choose **Copy Python Command**. Paste it, and you have the exact line, parameters and all.

Cursors: walking the attribute table

To read or change attribute values row by row, ArcPy uses **cursors**. You name the fields you want, and each row comes back as a sequence of values *in that order*, so you use positions (from 0, of course):

```
with arcpy.da.SearchCursor("villages", ["NAME", "NDVI"]) as cursor:
    for row in cursor:
        print(row[0], row[1])      # NAME, then NDVI
```

To write, use an `UpdateCursor`. This is *Calculate Field with a condition*, which you first built by hand in Part 4:

```
arcpy.management.AddField("villages", "STATUS", "TEXT")

with arcpy.da.UpdateCursor("villages", ["NDVI", "STATUS"]) as cursor:
    for row in cursor:
        row[1] = "healthy" if row[0] > 0.5 else "stressed"
        cursor.updateRow(row)      # save this row's change
```

Everything in those lines is familiar: `with` from Part 8, `for` from Part 4, the one-line `if/else` from Part 3, list positions from Part 2, and a method call from Part 6. That's the point of the whole course: there is no "ArcPy language". There's Python, with ArcGIS's nouns.

What you now own

- The four tools are **one grammar with different nouns**. The table at the start of this part is your translation key.
- **Earth Engine code writes recipes; Google's servers cook them**. So: use ee methods, use `.map()` instead of loops, ask pixels questions with `.gt()` and friends, and `.getInfo()` only for small answers.
- Earth Engine's rhythm is **filter** → **map** → **reduce**, then **visualise** with `geemap` or **export** as a background task.
- **Google Cloud**: projects are workspaces, APIs must be enabled, authentication says who's asking, and buckets hold big exports.
- **QGIS**: `iface`, layers and features, and `processing.run(tool_id, {parameters})`. Copy as Python Command teaches you the rest.
- **ArcGIS**: `arcpy.env.workspace`, `List...` functions, tools as functions, and cursors for rows. Copy Python Command from History.

The capstone

One real question, answered from start to finish. Every idea from the course shows up at least once. Watch for them.

The question

"How did crop greenness in Meerut district change month by month through the 2025 kharif season, and which villages were lagging behind in August?"

This is the kind of question that arrives on a real desk. Before any code, let's do what Part 1 taught and write the plan in plain, precise language:

1. Set the inputs: district, year, months, cloud limit, NDVI threshold.
2. Build a tool that, for any date range, makes a cloud-filtered median NDVI image over the district.
3. For each month June–October:
 - make that month's NDVI image, and
 - reduce it to one district-wide average.
4. Bring those five small numbers home and draw them as a line chart.
5. For August: average NDVI inside every village polygon.
6. Label each village "healthy" (> threshold) or "stressed".
7. Export the labelled villages as a shapefile.
8. Open it in QGIS or ArcGIS and colour by label.

Each step is small enough to be obvious. Now we translate.

Step 1: Ingredients, labelled once

```
import ee, geemap
import pandas as pd

ee.Authenticate()
ee.Initialize(project="your-project-id")

# — Inputs: change these, and the whole analysis follows —
YEAR = 2025
MONTHS = [6, 7, 8, 9, 10]           # June to October
CLOUD_LIMIT = 40                    # monsoon skies are rarely clear
NDVI_THRESHOLD = 0.5

district = (ee.FeatureCollection("FAO/GAUL/2015/level2")
            .filter(ee.Filter.eq("ADM2_NAME", "Meerut")))
aoi = district.geometry()
villages = ee.FeatureCollection("projects/your-project-id/assets/meerut_villages")
s2 = ee.ImageCollection("COPERNICUS/S2_SR_HARMONIZED")
```

Part 2: variables as labelled ingredients. The names in capitals are a common convention meaning "a setting you might tune, which the code itself never changes". Putting them all at the top is the "change one line, everything updates" idea. Part 6: modules imported, objects built from blueprints, and a chained filter. The village layer is your own shapefile, uploaded once as an Earth Engine asset.

Step 2: A tool for making NDVI composites

```
def add_ndvi(image):
    """Add an NDVI band to one Sentinel-2 image."""
    ndvi = image.normalizedDifference(["B8", "B4"]).rename("NDVI")
    return image.addBands(ndvi)

def ndvi_composite(start, end):
    """Median NDVI over the district for a date range."""
    return (s2.filterBounds(aoi)
            .filterDate(start, end)
            .filter(ee.Filter.lt("CLOUDY_PIXEL_PERCENTAGE", CLOUD_LIMIT))
            .map(add_ndvi)
            .select("NDVI")
            .median()
            .clip(aoi))
```

Part 5: two functions, each with a docstring. `add_ndvi` takes one image and returns one image, which is the exact shape `.map()` needs. `ndvi_composite` bundles the entire filter → map → reduce-through-time rhythm (Part 9) into one reusable tool with two blanks: a start and an end date. We'll use it five times for the chart and once more for the villages, and we wrote it once.

Step 3: The monthly series

```
rows = []

for month in MONTHS:
    start = ee.Date.fromYMD(YEAR, month, 1)
    end = start.advance(1, "month")
    composite = ndvi_composite(start, end)

    stats = composite.reduceRegion(
        reducer=ee.Reducer.mean(), geometry=aoi, scale=20, maxPixels=1e10)

    rows.append({"month": month, "ndvi": stats.get("NDVI").getInfo()})
    print(f"Month {month} done")
```

Wait. A Python `for` loop, in Earth Engine code? Didn't Part 9 say not to?

Look closely at *what* is being looped over. `MONTHS` is an ordinary Python list of five numbers, sitting on your computer. The loop runs five times, and each time it *writes a recipe* and asks for **one small number** back with `.getInfo()`. We're not looping over pixels or images in the kitchen. We're sending five small, separate orders. That's perfectly fine. The rule was never "no loops ever". It was "don't loop over server data". Understanding *why* a rule exists lets you know when it applies, and that is what first principles buy you.

(`scale=20` here, rather than 10, simply makes each district-wide average quicker to compute. For an average over thousands of square kilometres, the difference is negligible. Choosing that is a GIS judgement, not a Python one.)

Part 4: an empty list, filled one dictionary per pass. That's the collect pattern. Part 2: the result is a list of dictionaries, which is an attribute table.

Think about it. What if one month has **no** images under the cloud limit? The composite has no bands, `stats` comes back empty, and `.get("NDVI")` fails. You'd see an `EEException` complaining that the dictionary doesn't contain the key. Using Part 7, you'd read the last line, check `images.size()` for that month, and either raise `CLOUD_LIMIT` or widen the date window. Now you know what the error means *before* you ever see it.

Step 4: From numbers to a chart

```
df = pd.DataFrame(rows)
df["month_name"] = ["Jun", "Jul", "Aug", "Sep", "Oct"]
print(df)

df.plot(x="month_name", y="ndvi", marker="o", legend=False,
        title=f"Mean NDVI, Meerut district, kharif {YEAR}")
```

Part 8: a list of dictionaries becomes a DataFrame in one line, and a DataFrame can draw itself. You should see greenness rise through the monsoon, peak around August or September as the kharif crop matures, and fall as harvest begins. If it doesn't, that's not a bug to fix. It's a finding to investigate, and that's where your GIS expertise takes over from the code.

Step 5 and 6: Every village, labelled

```
ndvi_aug = ndvi_composite(f"{YEAR}-08-01", f"{YEAR}-09-01")

village_stats = ndvi_aug.reduceRegions(
    collection=villages, reducer=ee.Reducer.mean(), scale=10)

def label_village(feature):
    ndvi = ee.Number(feature.get("mean"))
    status = ee.Algorithms.If(ndvi.gt(NDVI_THRESHOLD), "healthy", "stressed")
    return feature.set({"ndvi_aug": ndvi, "status": status})

labelled = village_stats.map(label_village)

print(labelled.aggregate_histogram("status").getInfo())
# e.g. {'healthy': 412, 'stressed': 97}
```

`reduceRegions` (with an *s*) is `reduceRegion`'s big sibling. Instead of one average for one area, it computes an average *inside every feature* of a collection, and writes it onto each feature as a property called `mean`. It's Zonal Statistics as Table, done in the kitchen.

Then look at `label_village`. It's Part 3's healthy/stressed decision, but written the server-side way (Part 9, Rule 2): `ndvi.gt(...)` instead of `>`, and `ee.Algorithms.If(condition, yes, no)` instead of Python's `if`. The same logic, cooked in the kitchen. And it's handed to `.map()`, so every village is labelled in parallel.

The final line asks for a tiny summary, a count per label, instead of all 500 villages. That's Rule 3: bring home only what you need to see.

Step 7: Out to a file

```
task = ee.batch.Export.table.toDrive(
    collection=labelled,
    description=f"meerut_villages_ndvi_aug{YEAR}",
    folder="GEE_exports",
    fileFormat="SHP"
)
task.start()
```

A few minutes later, a shapefile appears in the `GEE_exports` folder of your Google Drive. (For large jobs, `Export.table.toCloudStorage` with a bucket works the same way.)

And for a quick look without waiting:

```
m = geemap.Map()
m.centerObject(district, 10)
```

```
m.addLayer(ndvi_aug, {"min": 0, "max": 0.8,
                    "palette": ["#a6611a", "#f5f5a0", "#1a9641"]}, "NDVI Aug")
m.addLayer(labelled.filter(ee.Filter.eq("status", "stressed")),
           {"color": "red"}, "Stressed villages")
m
```

Step 8: Back on the desktop

Open the exported shapefile in QGIS or ArcGIS and apply a categorised style on `status`: green for healthy, red for stressed. The map a manager can read is done.

And because you now speak the dialects, you could do the last mile in code too. In ArcGIS Pro, for example, counting stressed villages per tehsil is a cursor and a dictionary:

```
counts = {}
with arcpy.da.SearchCursor("meerut_villages_ndvi_aug2025",
                          ["TEHSIL", "status"]) as cursor:
    for tehsil, status in cursor:
        if status == "stressed":
            counts[tehsil] = counts.get(tehsil, 0) + 1
print(counts)
```

(`counts.get(tehsil, 0)` means "the current count, or 0 if this tehsil hasn't been seen yet". It's a tidy accumulator for counting by category. The `TEHSIL` field is whatever your own village layer calls it.)

Looking back at what you just did

Read the whole capstone again, and count what's in it: values and variables, lists and dictionaries, a list of dictionaries used as a table, conditions (both kinds), a `for` loop and `.map()`, functions with parameters and docstrings, functions handed to other functions, objects and chained methods, four imported modules, an f-string in a file name, a potential error understood in advance, a `DataFrame`, a chart, a cloud export, and a cursor. That is essentially the entire course, in about seventy lines, doing real work.

Nothing in it is magic, and nothing in it was memorised. Every line is a small, understood step, which is exactly how it should feel.

Next time a question arrives on your desk, the method is the same. Write the plan in plain words. Decompose it until each step is obvious. Translate one step at a time, checking small results as you go. When it breaks, read the error from the bottom.

Where to go from here

- **Rewrite the capstone for a different question.** Change the district, the season, or the index (NDWI for water, for example). Changing a working script is the fastest way to learn.
- **Automate one real, boring task from your own week.** Pick the most repetitive clicking you do, write its pseudocode, and script it in whichever tool it lives in.
- **Learn `geopandas`:** `pandas` with geometry. Your `DataFrame` skills carry over directly.
- **The official Earth Engine guides** (developers.google.com/earth-engine) are excellent, and you can now read them. The *Client vs. Server* and *Coding Best Practices* pages will feel like revision.
- **Free structured courses.** Spatial Thoughts' Earth Engine courses, the `geemap` tutorials, and the `ArcPy` and `PyQGIS` cookbooks all make natural next steps.

APPENDIX

Memory aid and practice rhythm

One page to keep beside the keyboard, and a pace that makes it stick.

The whole course on one page

Idea	Python shape	Your GIS cousin
a value and its type	4120, 0.58, "Meerut", True, None	a cell and its field type
a labelled value	cloud_limit = 20	a model parameter
text with values inside	f"ndvi_{year}.tif"	Concatenate in Field Calculator
an ordered collection	["B2", "B3", "B4"], index from 0	one column
key → value pairs	{"name": "Kheda", "ndvi": 0.41}	one row
a whole table	a list of dictionaries, or a pandas DataFrame	an attribute table
a decision	if ... : / elif ... : / else:	Select by Attributes, Reclassify
repetition	for item in items:	Batch processing
a reusable recipe	def tool(a, b): ... return result	a ModelBuilder model
do a tool to every item	map(tool, items), .map(tool) in Earth Engine	Batch processing, in parallel
a thing that knows and does	object.attribute, object.method()	a layer's Properties and right-click menu
a toolbox	import arcpy, import ee	ArcToolbox / Processing Toolbox
a file's address	Path(r"C:\GIS\data.tif")	a data source path
surviving a bad item	try: ... except Exception as err:	a batch run that logs failures

Earth Engine's three rules. Use ee methods, not Python operators, on ee objects. Use `.map()`, not `for`, over server collections, and ask pixels with `.gt()`, `.lt()` and `.eq()`. Call `.getInfo()` only for small answers.

Reading any error. Start at the last line (error type and message), then the line it points to, then the trail above it.

When stuck. Write the plan in plain words. Make the problem smaller. `print` the thing and its type.

A rhythm for practice

Reading builds understanding. Only typing builds fluency. A sustainable pace, alongside regular work:

Week	Read	Practise in Colab (about 30 minutes a day)
1	Parts 0–1	Run every example. Write pseudocode for three tasks from your own week.
2	Part 2	Rebuild one of your attribute tables as a list of dictionaries, by hand.
3	Parts 3–4	Write the "healthy/stressed" labelling for your table. Then try a total and a filter.
4	Part 5	Turn every calculation from Weeks 2 and 3 into a named function. Use <code>map</code> .
5	Parts 6–7	Explore <code>dir()</code> and Tab-completion on strings and lists. Break code on purpose and read the errors.
6	Part 8	Read a real CSV export from your work with <code>csv</code> , then with <code>pandas</code> . Filter, group and export it.
7	Part 9 (Earth Engine, Google Cloud)	Filter → map → reduce on your own area. Export one image and one table.
8	Part 9 (QGIS, ArcGIS)	Use Copy as Python Command on three tools you use weekly, then wrap one in a loop over a folder.
9	Part 10	Run the capstone on your own district, then change one thing and run it again.

One honest note about practice: understanding a line when you read it and being able to write it from nothing are different skills, and the second only comes from doing. Expect the early weeks to feel slow. They're laying the ground that everything later stands on, and the speed arrives all at once, usually somewhere around Week 5, when the pieces start connecting on their own.