CCUS Simulator Textbook

Chapter 1 — Imports & Dataclass

Code: Imports

```
# === CCUS GUI: Default scenario + Agent-based, with cost & energy, ppm, and live plotting ===

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from dataclasses import dataclass

from ipywidgets import (
```

Explanation

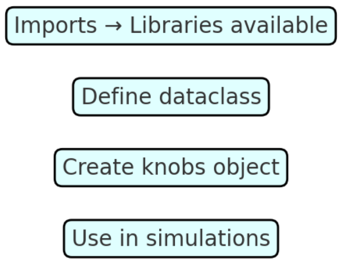
This block loads all external libraries used by the notebook. NumPy (`np`) provides fast numerical arrays; Pandas (`pd`) provides tables (DataFrames); Matplotlib (`plt`) draws charts; ipywidgets provides sliders and dropdowns for the UI. The `dataclass` decorator lets us define a compact class of parameters without writing boilerplate constructors.

Code: CostEnergyKnobs

```
# ---------- helpers & data "cards" ----------

@dataclass

class CostEnergyKnobs:

    ccus_cost_usd_per_t0: float = 120.0   # starting $/t captured (tech-only)

    ccus_learning_rate_pct: float = 5.0   # cost ↓ %/yr

    ccus_energy_kWh_per_t0: float = 300.0 # kWh per t captured (start)

    ccus_energy_improve_pct: float = 2.0  # energy ↓ %/yr

    power_price_usd_per_kwh: float = 0.07 # $/kWh
    grid_kgCO2_per_kwh: float = 0.40      # kg CO2 / kWh
```

Explanation

`CostEnergyKnobs` stores all CCUS cost/energy assumptions (cost per ton, learning rate, energy per ton, grid intensity, etc.). Each field has a type hint and a default value so you can create an instance and override any field by name. This object is passed to simulators, keeping all assumptions consistent.



Chapter 1 — Quiz

Q1. What does `as np` mean in `import numpy as np`?
Q2. Why use a `@dataclass` for the parameter knobs?

- ➔ It's just an easier way to create an object and its parameters

Q3. What's the difference between a list and a NumPy array?

- ➔ Basically good practice for vectorized maths

Chapter 2 — Helper Functions

Code: _ppm_from_net_series()

```
def _ppm_from_net_series(net_mt_series, starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2):

    ppm_vals, ppm_now = [], float(starting_ppm)

    for net_mt in np.asarray(net_mt_series, dtype=float):

        add_ppm = (net_mt / 1000.0) * ppm_per_GtCO2

        excess  = ppm_now - preindustrial_ppm

        sink    = k_sink * excess
        ppm_now = ppm_now + add_ppm - sink

        ppm_vals.append(ppm_now)

    return np.array(ppm_vals, dtype=float)
```

Explanation

This function converts a series of yearly net CO_2 emissions (in $MtCO_2$) into atmospheric CO_2 concentrations (ppm) using a simple linear-sink model.

Line by line:

def _ppm_from_net_series(...):
- `def` defines a function. `_ppm_from_net_series` is its name (underscore means "helper").
- The parameters are inputs: net emissions, starting ppm, preindustrial baseline, sink rate, and conversion factor.

ppm_vals, ppm_now = [], float(starting_ppm)
- Tuple assignment: creates an empty list (to store yearly ppm values) and initializes current ppm from the starting value.

for net_mt in np.asarray(net_mt_series, dtype=float):
- A for-loop: converts the input series into a NumPy array of floats and iterates year by year. `net_mt` is the emissions that year ($MtCO_2$).

add_ppm = (net_mt / 1000.0) * ppm_per_GtCO2
- Converts Mt to Gt by dividing by 1000, then multiplies by ppm_per_$GtCO_2$ to get the ppm increment.

excess = ppm_now - preindustrial_ppm
- Difference between current ppm and the preindustrial baseline.

sink = k_sink * excess
- Natural sink removes a fraction of that excess each year.

ppm_now = ppm_now + add_ppm - sink
- Euler update: new ppm = old + emissions increment - sink removal.

ppm_vals.append(ppm_now)
- Appends the new ppm to the results list.

return np.array(ppm_vals, dtype=float)

- Returns the ppm values as a NumPy array for later plotting or math.

Worked Example:
- starting_ppm = 420, preindustrial_ppm = 280, k_sink = 0.012, ppm_per_GtCO$_2$ = 0.1282
- net_mt_series = [35000]
- Convert: 35000 Mt = 35 Gt → add_ppm ≈ 4.49 ppm
- excess = 140 ppm → sink = 1.68 ppm
- new ppm = 420 + 4.49 − 1.68 ≈ 422.81

Pitfalls:
1. Units: net_mt_series is in Mt, not Gt.
2. k_sink too large → ppm falls unrealistically.
3. If starting_ppm < preindustrial, sink becomes negative (source).
4. Return length: one ppm per year, no extra starting point.

Worked example

If net emissions are [30, 25, 20] Mt and 1 ppm ≈ 7.8 GtCO$_2$, then each 1 Mt raises ppm by ~0.000128. Starting at 420 ppm with preindustrial 280 ppm and a 1% sink of the excess, ppm declines modestly each year unless emissions are large.

Code: _cost_energy_from_captured()

```
def _cost_energy_from_captured(captured_mt, knobs: CostEnergyKnobs,
                               cost0, learn_pct, kwh0, improve_pct, power_price, grid_kg_per_kwh):
    cap = np.asarray(captured_mt, dtype=float)
    n = len(cap)
    cost_per_t = cost0 * (1.0 - learn_pct/100.0) ** np.arange(n)
    kwh_per_t  = kwh0  * (1.0 - improve_pct/100.0) ** np.arange(n)
    tech_cost_usd_b   = cap * cost_per_t / 1e6
    energy_twh        = cap * kwh_per_t / 1000.0
    energy_bill_usd_b = (cap * kwh_per_t * power_price) / 1e9
    energy_emis_mt    = cap * kwh_per_t * grid_kg_per_kwh / 1000.0
    return (cost_per_t, kwh_per_t, tech_cost_usd_b, energy_twh, energy_bill_usd_b, energy_emis_mt)


# ---------- simulators ----------
def simulate_default_with_cost_energy(start_year, end_year,
                                      gross_start_mt, gross_decline_rate_pct,
                                      ccus_start_mt, ccus_growth_rate_pct,
                                      starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,
                                      knobs: CostEnergyKnobs) -> pd.DataFrame:
    years = list(range(int(start_year), int(end_year)+1))
    g = float(gross_start_mt)
    c = float(ccus_start_mt)
    decline = 1.0 - float(gross_decline_rate_pct)/100.0
    growth  = 1.0 + float(ccus_growth_rate_pct)/100.0

    gross, captured = [], []
    for i, _ in enumerate(years):
```

```
    if i > 0:
        g *= decline
        c *= growth
    gross.append(g)
    captured.append(c)
gross = np.array(gross, dtype=float)
captured = np.array(captured, dtype=float)
net = gross - captured


ppm_vals = _ppm_from_net_series(net, starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2)


(cost_per_t, kwh_per_t, tech_cost_b,
 energy_twh, energy_bill_b, energy_emis_mt) = _cost_energy_from_captured(
    captured, knobs,
    knobs.ccus_cost_usd_per_t0, knobs.ccus_learning_rate_pct,
    knobs.ccus_energy_kWh_per_t0, knobs.ccus_energy_improve_pct,
    knobs.power_price_usd_per_kwh, knobs.grid_kgCO2_per_kwh
)


df = pd.DataFrame({
    "Year": years,
    "Gross_Mt": gross,
    "Captured_Mt": captured,
    "Net_Mt": net,
    "Atmospheric_CO2_ppm": ppm_vals,
    "CCUS_Cost_USD_per_t": cost_per_t,
    "CCUS_Energy_kWh_per_t": kwh_per_t,’
    "CCUS_Tech_Cost_USD_B": tech_cost_b,
    "CCUS_Energy_TWh": energy_twh,
        "CCUS_Energy_Bill_USD_B": energy_bill_b,
        "Energy_Emissions_Mt": energy_emis_mt,
    })
    df["Effective_Net_Mt"] = df["Net_Mt"] + df["Energy_Emissions_Mt"]
    df["CCUS_Total_Spend_USD_B"] = df["CCUS_Tech_Cost_USD_B"] +
df["CCUS_Energy_Bill_USD_B"]
    return df


ABM_SECTORS = ["Power","Industry","Transport","Buildings","Other"]


def simulate_abm_with_cost_energy(start_year, end_year,
                                  sector_gross_start_mt: dict, sector_decline_pct: dict,
                                  ccus_start_mt, ccus_growth_rate_pct,
                                  starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,
                                  knobs: CostEnergyKnobs) -> pd.DataFrame:
    years = list(range(int(start_year), int(end_year)+1))
    current  = {s: float(sector_gross_start_mt[s]) for s in ABM_SECTORS}
    declines = {s: 1.0 - float(sector_decline_pct[s])/100.0 for s in ABM_SECTORS}
    c = float(ccus_start_mt)
    growth = 1.0 + float(ccus_growth_rate_pct)/100.0

    gross_list, captured_list, net_list = [], [], []
    for i, _ in enumerate(years):
        if i > 0:
            for s in ABM_SECTORS:
                current[s] *= declines[s]
            c *= growth
        g_total = sum(current.values())
        gross_list.append(g_total)
    captured_list.append(c)
    net_list.append(g_total - c)


gross = np.array(gross_list, dtype=float)
captured = np.array(captured_list, dtype=float)
net = np.array(net_list, dtype=float)


ppm_vals = _ppm_from_net_series(net, starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2)


(cost_per_t, kwh_per_t, tech_cost_b,
 energy_twh, energy_bill_b, energy_emis_mt) = _cost_energy_from_captured(
    captured, knobs,
    knobs.ccus_cost_usd_per_t0, knobs.ccus_learning_rate_pct,
    knobs.ccus_energy_kWh_per_t0, knobs.ccus_energy_improve_pct,
    knobs.power_price_usd_per_kwh, knobs.grid_kgCO2_per_kwh
)


df = pd.DataFrame({
    "Year": years,
    "Gross_Mt": gross,
    "Captured_Mt": captured,
    "Net_Mt": net,
    "Atmospheric_CO2_ppm": ppm_vals,
    "CCUS_Cost_USD_per_t": cost_per_t,
    "CCUS_Energy_kWh_per_t": kwh_per_t,
    "CCUS_Tech_Cost_USD_B": tech_cost_b,
    "CCUS_Energy_TWh": energy_twh,
    "CCUS_Energy_Bill_USD_B": energy_bill_b,
        "Energy_Emissions_Mt": energy_emis_mt,
    })
    df["Effective_Net_Mt"] = df["Net_Mt"] + df["Energy_Emissions_Mt"]
    df["CCUS_Total_Spend_USD_B"] = df["CCUS_Tech_Cost_USD_B"] +
df["CCUS_Energy_Bill_USD_B"]
    return df


# ---------- widgets ----------
mode_dd = Dropdown(options=["Default scenario (live)", "Agent-based (live)"],
                   value="Default scenario (live)")


start_year = IntSlider(min=2020, max=2030, step=1, value=2025)
end_year   = IntSlider(min=2030, max=2060, step=1, value=2050)


gross_start_mt         = IntSlider(min=10000, max=60000, step=500, value=35000)
gross_decline_rate_pct = FloatSlider(min=0.0, max=5.0, step=0.1, value=0.7)
ccus_start_mt          = IntSlider(min=0, max=500, step=5, value=40)
ccus_growth_rate_pct   = FloatSlider(min=0.0, max=50.0, step=0.5, value=15.0)


sector_start = {s: IntSlider(min=0, max=30000, step=250, value=v) for s, v in
                zip(ABM_SECTORS, [12000, 9000, 7000, 5000, 2000])}
sector_decl  = {s: FloatSlider(min=0.0, max=10.0, step=0.1, value=v) for s, v in
                zip(ABM_SECTORS, [2.5, 1.5, 1.0, 1.8, 0.5])}


starting_ppm        = FloatSlider(min=350, max=500, step=1, value=420)
preindustrial_ppm   = FloatSlider(min=250, max=300, step=0.5, value=280)
k_sink              = FloatSlider(min=0.0, max=0.05, step=0.001, value=0.012)
ppm_per_GtCO2       = FloatSlider(min=0.08, max=0.20, step=0.001, value=1.0/7.8)
```

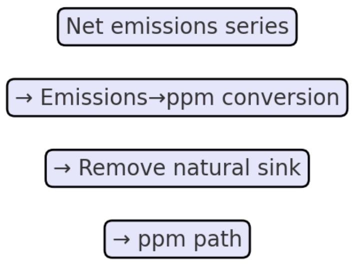
```
ccus_cost_usd_per_t0   = FloatSlider(min=20,  max=600,  step=5,   value=120)

ccus_learning_rate_pct = FloatSlider(min=0,   max=30,   step=0.5, value=5.0)

ccus_energy_kwh_per_t0 = FloatSlider(min=50,  max=1200, step=10,  value=300)

ccus_energy_improve_pct= FloatSlider(min=0,   max=20,   step=0.5, value=2.0)

power_price_usd_per_kwh= FloatSlider(min=0.02,max=0.3,  step=0.005,value=0.07)

grid_kgCO2_per_kwh     = FloatSlider(min=0.0, max=0.9,  step=0.01, value=0.40)
```

Explanation

Maps the captured series ($MtCO_2$) into technology cost, energy demand, and energy-related emissions. Cost = captured × cost_per_ton; Energy = captured × energy_per_ton; Energy emissions = Energy × grid_intensity.

Worked example

If captured = 10 Mt and cost_per_ton = \$120/t, total tech cost = $10\times10^6\times120$ = \$1.2B. If energy_per_ton = 0.5 MWh/t and grid_intensity = 0.4 t/MWh, energy-related emissions = $10\times10^6\times0.5\times0.4$ = 2 Mt.



Chapter 2 — Quiz

Q1. Why do we subtract a fraction of the excess ppm each year?
Q2. How do you convert Mt to t when computing total cost?
Q3. What happens to energy-related emissions if grid_kgCO2_per_kwh is reduced?

Chapter 3a — Default Simulator

Full Code

```
def simulate_default_with_cost_energy(start_year, end_year,

                                      gross_start_mt, gross_decline_rate_pct,

                                      ccus_start_mt, ccus_growth_rate_pct,

                                      starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,

                                      knobs: CostEnergyKnobs) -> pd.DataFrame:

    years = list(range(int(start_year), int(end_year)+1))

    g = float(gross_start_mt)

    c = float(ccus_start_mt)

    decline = 1.0 - float(gross_decline_rate_pct)/100.0

    growth  = 1.0 + float(ccus_growth_rate_pct)/100.0



    gross, captured = [], []

    for i, _ in enumerate(years):

        if i > 0:

            g *= decline

            c *= growth

        gross.append(g)

        captured.append(c)

    gross = np.array(gross, dtype=float)

    captured = np.array(captured, dtype=float)

    net = gross - captured



    ppm_vals = _ppm_from_net_series(net, starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2)



    (cost_per_t, kwh_per_t, tech_cost_b,

     energy_twh, energy_bill_b, energy_emis_mt) = _cost_energy_from_captured(
        captured, knobs,

        knobs.ccus_cost_usd_per_t0, knobs.ccus_learning_rate_pct,

        knobs.ccus_energy_kWh_per_t0, knobs.ccus_energy_improve_pct,

        knobs.power_price_usd_per_kwh, knobs.grid_kgCO2_per_kwh

    )



    df = pd.DataFrame({

        "Year": years,

        "Gross_Mt": gross,

        "Captured_Mt": captured,

        "Net_Mt": net,

        "Atmospheric_CO2_ppm": ppm_vals,

        "CCUS_Cost_USD_per_t": cost_per_t,

        "CCUS_Energy_kWh_per_t": kwh_per_t,’

        "CCUS_Tech_Cost_USD_B": tech_cost_b,

        "CCUS_Energy_TWh": energy_twh,

        "CCUS_Energy_Bill_USD_B": energy_bill_b,

        "Energy_Emissions_Mt": energy_emis_mt,

    })

    df["Effective_Net_Mt"] = df["Net_Mt"] + df["Energy_Emissions_Mt"]

    df["CCUS_Total_Spend_USD_B"] = df["CCUS_Tech_Cost_USD_B"] +
df["CCUS_Energy_Bill_USD_B"]

    return df
```

Overarching explanation

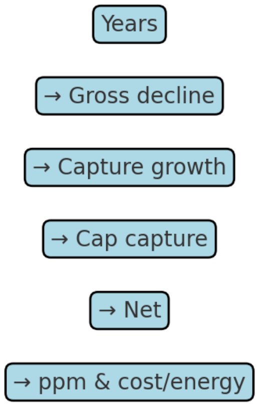
Represents the world as a single emissions system. Each year, gross emissions shrink by a decline multiplier (e.g., 0.98 for −2%/yr) and CCUS capture grows by a growth multiplier (e.g., 1.20 for +20%/yr). Capture is capped at gross. Net = gross − capture. Net is converted to ppm; capture maps to cost and energy.

Narrative walkthrough

The function builds a list of years, initializes `g` and `c` with starting values, and computes `decline` and `growth` multipliers from percentages. It loops over the years using `enumerate(years)`. On the first iteration (`i == 0`) it preserves the starting values. From the second iteration onward (`i > 0`), `g` is multiplied by `decline` and `c` by `growth`, compounding both series. It stores yearly gross and captured in lists, converts them to NumPy arrays, computes `net = gross - captured`, then calls helpers for ppm and costs.

Worked numeric example

With gross_start=30 Mt, decline=0.98, ccus_start=2 Mt, growth=1.5 over 2025–2027: 2025 gross=30, capture=2 → net=28; 2026 gross=29.4, capture=3 → net=26.4; 2027 gross≈28.812, capture=4.5 → net≈24.312.



Chapter 3a — Quiz

Q1. Why does the code use `if i > 0` before applying decline/growth?
Q2. What does 0.98 represent when applied each year to gross?
Q3. Why must capture be capped at gross?

Chapter 3b — ABM (Agent-Based) Simulator

Full Code

```
ABM_SECTORS = ["Power","Industry","Transport","Buildings","Other"]


def simulate_abm_with_cost_energy(start_year, end_year,
                                 sector_gross_start_mt: dict, sector_decline_pct: dict,
                                 ccus_start_mt, ccus_growth_rate_pct,
                                 starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,
                                 knobs: CostEnergyKnobs) -> pd.DataFrame:
    years = list(range(int(start_year), int(end_year)+1))
    current  = {s: float(sector_gross_start_mt[s]) for s in ABM_SECTORS}
    declines = {s: 1.0 - float(sector_decline_pct[s])/100.0 for s in ABM_SECTORS}
    c = float(ccus_start_mt)
    growth = 1.0 + float(ccus_growth_rate_pct)/100.0

    gross_list, captured_list, net_list = [], [], []
    for i, _ in enumerate(years):
        if i > 0:
            for s in ABM_SECTORS:
                current[s] *= declines[s]
            c *= growth
        g_total = sum(current.values())
        gross_list.append(g_total)
        captured_list.append(c)
        net_list.append(g_total - c)

    gross = np.array(gross_list, dtype=float)
    captured = np.array(captured_list, dtype=float)
    net = np.array(net_list, dtype=float)



    ppm_vals = _ppm_from_net_series(net, starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2)



    (cost_per_t, kwh_per_t, tech_cost_b,
     energy_twh, energy_bill_b, energy_emis_mt) = _cost_energy_from_captured(
        captured, knobs,
        knobs.ccus_cost_usd_per_t0, knobs.ccus_learning_rate_pct,
        knobs.ccus_energy_kWh_per_t0, knobs.ccus_energy_improve_pct,
        knobs.power_price_usd_per_kwh, knobs.grid_kgCO2_per_kwh
    )



    df = pd.DataFrame({
        "Year": years,
        "Gross_Mt": gross,
        "Captured_Mt": captured,
        "Net_Mt": net,
        "Atmospheric_CO2_ppm": ppm_vals,
        "CCUS_Cost_USD_per_t": cost_per_t,
        "CCUS_Energy_kWh_per_t": kwh_per_t,
        "CCUS_Tech_Cost_USD_B": tech_cost_b,
        "CCUS_Energy_TWh": energy_twh,
        "CCUS_Energy_Bill_USD_B": energy_bill_b,
        "Energy_Emissions_Mt": energy_emis_mt,
    })
    df["Effective_Net_Mt"] = df["Net_Mt"] + df["Energy_Emissions_Mt"]
    df["CCUS_Total_Spend_USD_B"] = df["CCUS_Tech_Cost_USD_B"] +
df["CCUS_Energy_Bill_USD_B"]
    return df
```

Overarching explanation

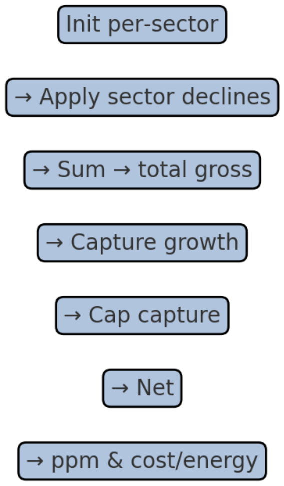
Splits emissions into sectors (Power, Industry, Transport, Buildings, Other). Each sector declines at its own rate. Each year, sector emissions compound down, are summed to total gross, capture grows, cap is applied, net is computed, then ppm and costs are derived.

Narrative walkthrough

The function creates dictionaries `current` (sector → current gross) and `declines` (sector → multiplier). In each year after the first, it multiplies each `current[s]` by `declines[s]` and scales `c` by `growth`. It sums `current.values()` to get `g_total`, appends gross/captured/net to lists, and after the loop converts lists to arrays. Helpers compute ppm and costs before assembling a DataFrame.

Worked numeric example

With Power=10 (0.95), Industry=20 (0.90), ccus_start=2 (1.5 growth) over 2025–2027: 2025 gross=30, capture=2 → net=28; 2026 gross=27.5, capture=3 → net=24.5; 2027 gross≈25.225, capture=4.5 → net≈20.725.



Chapter 3b — Quiz

Q1. What does `current[s] *= declines[s]` do conceptually?
Q2. If a sector declines 10%/yr, what is the yearly multiplier?
Q3. Why is capture compared to the sum of sectors (total gross), not to a single sector?

Chapter 4 — Widgets & UI

Full Code

```
# ---------- widgets ----------

mode_dd = Dropdown(options=["Default scenario (live)", "Agent-based (live)"],

                   value="Default scenario (live)")



start_year = IntSlider(min=2020, max=2030, step=1, value=2025)

end_year   = IntSlider(min=2030, max=2060, step=1, value=2050)



gross_start_mt         = IntSlider(min=10000, max=60000, step=500, value=35000)

gross_decline_rate_pct = FloatSlider(min=0.0, max=5.0, step=0.1, value=0.7)

ccus_start_mt          = IntSlider(min=0, max=500, step=5, value=40)

ccus_growth_rate_pct   = FloatSlider(min=0.0, max=50.0, step=0.5, value=15.0)



sector_start = {s: IntSlider(min=0, max=30000, step=250, value=v) for s, v in

                zip(ABM_SECTORS, [12000, 9000, 7000, 5000, 2000])}

sector_decl  = {s: FloatSlider(min=0.0, max=10.0, step=0.1, value=v) for s, v in

                zip(ABM_SECTORS, [2.5, 1.5, 1.0, 1.8, 0.5])}



starting_ppm        = FloatSlider(min=350, max=500, step=1, value=420)

preindustrial_ppm   = FloatSlider(min=250, max=300, step=0.5, value=280)

k_sink              = FloatSlider(min=0.0, max=0.05, step=0.001, value=0.012)

ppm_per_GtCO2       = FloatSlider(min=0.08, max=0.20, step=0.001, value=1.0/7.8)



ccus_cost_usd_per_t0   = FloatSlider(min=20,  max=600,  step=5,   value=120)

ccus_learning_rate_pct = FloatSlider(min=0,   max=30,   step=0.5, value=5.0)

ccus_energy_kwh_per_t0 = FloatSlider(min=50,  max=1200, step=10,  value=300)

ccus_energy_improve_pct= FloatSlider(min=0,   max=20,   step=0.5, value=2.0)
```

```
power_price_usd_per_kwh= FloatSlider(min=0.02,max=0.3,  step=0.005,value=0.07)

grid_kgCO2_per_kwh     = FloatSlider(min=0.0, max=0.9,  step=0.01, value=0.40)



def _grid_rows(rows):

    children = []

    for label, widget in rows:

        children += [Label(label), widget]

    return GridBox(children=children,

                   layout=Layout(grid_template_columns="220px 520px", grid_gap="6px 12px",
width="800px"))



grid_mode = _grid_rows([("Mode:", mode_dd)])



grid_default = _grid_rows([

    ("Start year:", start_year), ("End year:", end_year),

    ("Gross start (Mt):", gross_start_mt),

    ("Gross decline %/yr:", gross_decline_rate_pct),

    ("CCUS start (Mt):", ccus_start_mt),

    ("CCUS growth %/yr:", ccus_growth_rate_pct),

])



grid_abm = _grid_rows(

    [("Start year:", start_year), ("End year:", end_year)] +

    [(f"{s} start (Mt):", sector_start[s]) for s in ABM_SECTORS] +

    [(f"{s} decline %/yr:", sector_decl[s]) for s in ABM_SECTORS] +

    [("CCUS start (Mt):", ccus_start_mt), ("CCUS growth %/yr:", ccus_growth_rate_pct)]

)



grid_ppm = _grid_rows([
    ("Start ppm:", starting_ppm),
    ("Preindustrial ppm:", preindustrial_ppm),
    ("Sink rate /yr:", k_sink),
    ("ppm per GtCO₂:", ppm_per_GtCO2),
])


grid_cost = _grid_rows([
    ("CCUS $/t (start):", ccus_cost_usd_per_t0),
    ("Learning %/yr:", ccus_learning_rate_pct),
    ("Energy kWh/t (start):", ccus_energy_kwh_per_t0),
    ("Energy improve %/yr:", ccus_energy_improve_pct),
    ("Power price $/kWh:", power_price_usd_per_kwh),
    ("Grid kgCO₂/kWh:", grid_kgCO2_per_kwh),
])


def _toggle_grids(*_):
    if mode_dd.value == "Default scenario (live)":
        grid_default.layout.display = ""
        grid_abm.layout.display = "none"
    else:
        grid_default.layout.display = "none"
        grid_abm.layout.display = ""
mode_dd.observe(_toggle_grids, names="value")
_toggle_grids()


def _read_sector_widgets():
    sector_gross = {s: float(sector_start[s].value) for s in ABM_SECTORS}
    sector_declp = {s: float(sector_decl[s].value)  for s in ABM_SECTORS}
    return sector_gross, sector_declp
```

```
def _ui(mode,
        gross_start_mt, gross_decline_rate_pct,
        ccus_start_mt, ccus_growth_rate_pct,
        start_year, end_year,
        starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,
        ccus_cost_usd_per_t0, ccus_learning_rate_pct,
        ccus_energy_kwh_per_t0, ccus_energy_improve_pct,
        power_price_usd_per_kwh, grid_kgCO2_per_kwh):
    knobs = CostEnergyKnobs(
        ccus_cost_usd_per_t0=float(ccus_cost_usd_per_t0),
        ccus_learning_rate_pct=float(ccus_learning_rate_pct),
        ccus_energy_kWh_per_t0=float(ccus_energy_kwh_per_t0),
        ccus_energy_improve_pct=float(ccus_energy_improve_pct),
        power_price_usd_per_kwh=float(power_price_usd_per_kwh),
        grid_kgCO2_per_kwh=float(grid_kgCO2_per_kwh),
    )

    if mode == "Default scenario (live)":
        df = simulate_default_with_cost_energy(
            start_year, end_year,
            gross_start_mt, gross_decline_rate_pct,
            ccus_start_mt, ccus_growth_rate_pct,
            starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,
            knobs
        )
        used = "default scenario (live)"
    else:
        sector_gross, sector_declp = _read_sector_widgets()
        df = simulate_abm_with_cost_energy(
            start_year, end_year,
            sector_gross, sector_declp,
            ccus_start_mt, ccus_growth_rate_pct,
            starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,
            knobs
        )
        used = "agent-based (live)"


    # plot
    plt.figure(figsize=(9,5))
    ax1 = plt.gca()
    ax2 = ax1.twinx()
    ax1.plot(df["Year"], df["Gross_Mt"],    label="Gross (Mt)")
    ax1.plot(df["Year"], df["Captured_Mt"], label="Captured (Mt)")
    ax1.plot(df["Year"], df["Net_Mt"],      label="Net (Mt)")
    ax1.plot(df["Year"], df["Effective_Net_Mt"], label="Effective Net (Mt)", linestyle="--")
    ax1.set_xlabel("Year"); ax1.set_ylabel("Emissions (Mt/yr)"); ax1.grid(True)
    ax2.plot(df["Year"], df["Atmospheric_CO2_ppm"], label="CO₂ (ppm)")
    ax2.set_ylabel("CO₂ (ppm)")
    l1, lab1 = ax1.get_legend_handles_labels()
    l2, lab2 = ax2.get_legend_handles_labels()
    ax1.legend(l1 + l2, lab1 + lab2, loc="upper right")  # combine legends
    plt.title(f"Emissions, Effective Net & CO₂ (ppm) — {used}")
    plt.show()


    last = df.iloc[-1]
    print(
        f"Final {int(last['Year'])}: "
```

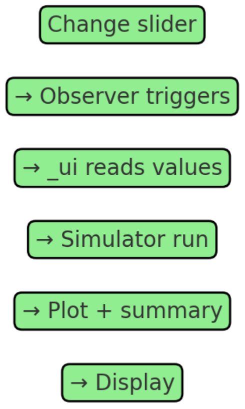
```
        f"Captured={last['Captured_Mt']:,.0f} Mt | "
        f"CCUS $/t={last['CCUS_Cost_USD_per_t']:,.0f} | "
        f"Tech cost=${last['CCUS_Tech_Cost_USD_B']:,.2f} B | "
        f"Energy={last['CCUS_Energy_TWh']:,.1f} TWh | "
        f"Energy bill=${last['CCUS_Energy_Bill_USD_B']:,.2f} B | "
        f"Energy emis={last['Energy_Emissions_Mt']:,.1f} Mt | "
        f"Total spend=${last['CCUS_Total_Spend_USD_B']:,.2f} B | "
        f"ppm={last['Atmospheric_CO2_ppm']:,.1f}"
    )


# wire sliders -> UI and display
out = interactive_output(
    _ui,
    {
        "mode": mode_dd,
        "gross_start_mt": gross_start_mt,
        "gross_decline_rate_pct": gross_decline_rate_pct,
        "ccus_start_mt": ccus_start_mt,
        "ccus_growth_rate_pct": ccus_growth_rate_pct,
        "start_year": start_year,
        "end_year": end_year,
        "starting_ppm": starting_ppm,
        "preindustrial_ppm": preindustrial_ppm,
        "k_sink": k_sink,
        "ppm_per_GtCO2": ppm_per_GtCO2,
        "ccus_cost_usd_per_t0": ccus_cost_usd_per_t0,
        "ccus_learning_rate_pct": ccus_learning_rate_pct,
        "ccus_energy_kwh_per_t0": ccus_energy_kwh_per_t0,
        "ccus_energy_improve_pct": ccus_energy_improve_pct,
        "power_price_usd_per_kwh": power_price_usd_per_kwh,

        "grid_kgCO2_per_kwh": grid_kgCO2_per_kwh,

    }

)



# assemble UI

ui = VBox([grid_mode, grid_default, grid_abm, grid_ppm, grid_cost, out])

display(ui)
```

Overarching explanation

Defines interactive controls (Dropdowns and Sliders) grouped into grids. A toggle shows either default or ABM inputs. `_ui` reads values, calls the simulators, plots results, and prints a one-line summary. Everything is connected with `interactive_output` and displayed in a vertical layout.

Narrative walkthrough

The mode dropdown flips which grid is visible via `_toggle_grids` and an observer on `mode_dd`. Sector sliders are created with dictionary comprehensions so each sector has its own start and decline controls. `_read_sector_widgets` converts slider values into dictionaries for the ABM simulator. `_ui` constructs the `CostEnergyKnobs`, chooses a simulator based on mode, then generates the plot and summary. Finally, `interactive_output` wires each widget to `_ui`, and a `VBox` assembles the whole UI.



Chapter 4 — Quiz

Q1. Why attach `mode_dd.observe(_toggle_grids, names='value')`?
Q2. What data structure does `_read_sector_widgets` return and why?
Q3. What does `interactive_output(_ui, {...})` accomplish?

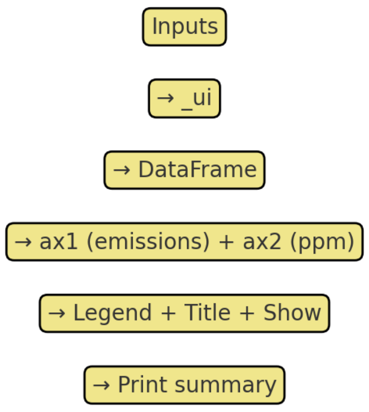
Chapter 5 — Plotting & Output

Code: Plotting & Printout

```
# plot

plt.figure(figsize=(9,5))

ax1 = plt.gca()

ax2 = ax1.twinx()

ax1.plot(df["Year"], df["Gross_Mt"],    label="Gross (Mt)")

ax1.plot(df["Year"], df["Captured_Mt"], label="Captured (Mt)")

ax1.plot(df["Year"], df["Net_Mt"],      label="Net (Mt)")

ax1.plot(df["Year"], df["Effective_Net_Mt"], label="Effective Net (Mt)", linestyle="--")

ax1.set_xlabel("Year"); ax1.set_ylabel("Emissions (Mt/yr)"); ax1.grid(True)

ax2.plot(df["Year"], df["Atmospheric_CO2_ppm"], label="CO₂ (ppm)")

ax2.set_ylabel("CO₂ (ppm)")

l1, lab1 = ax1.get_legend_handles_labels()

l2, lab2 = ax2.get_legend_handles_labels()

ax1.legend(l1 + l2, lab1 + l2, loc="upper right")  # combine legends

plt.title(f"Emissions, Effective Net & CO₂ (ppm) — {used}")

plt.show()



last = df.iloc[-1]

print(

    f"Final {int(last['Year'])}: "

    f"Captured={last['Captured_Mt']:,.0f} Mt | "

    f"CCUS $/t={last['CCUS_Cost_USD_per_t']:,.0f} | "

    f"Tech cost=${last['CCUS_Tech_Cost_USD_B']:,.2f} B | "
    f"Energy={last['CCUS_Energy_TWh']:,.1f} TWh | "
    f"Energy bill=${last['CCUS_Energy_Bill_USD_B']:,.2f} B | "
    f"Energy emis={last['Energy_Emissions_Mt']:,.1f} Mt | "
    f"Total spend=${last['CCUS_Total_Spend_USD_B']:,.2f} B | "
    f"ppm={last['Atmospheric_CO2_ppm']:,.1f}"
)
```

Explanation

Creates a Matplotlib figure, uses `ax1` for emissions (left axis) and `ax2` for CO_2 ppm (right axis). Merges legends, sets labels, adds a title, and renders with `plt.show()`. Obtains the final year row with `df.iloc[-1]` and prints a formatted summary using f-strings.



Chapter 5 — Quiz

Q1. Why use `twinx()`? What does it give you?
Q2. How are legends from both axes combined?
Q3. What does `df.iloc[-1]` return and why is it used here?

Glossary of Python Terms

Function — a reusable block of code that takes inputs (parameters) and can return outputs.
Loop — repeats a block of code multiple times.
List — an ordered collection of items, e.g., [1,2,3].
NumPy array — like a list but optimized for fast math.
DataFrame — a table with rows and columns (from Pandas).

Decorator — `@something` that modifies the behavior of a function/class definition.
Dictionary — key→value map, e.g., {'Power': 10, 'Industry': 20}.
Observer — a callback tied to a widget that runs when its value changes.

Appendix — Quiz Answers

Ch1 A1: `as np` creates a short alias for numpy; you can type `np.array` instead of `numpy.array`.
Ch1 A2: Dataclasses bundle parameters cleanly and auto-generate constructors.
Ch1 A3: Arrays enable fast vectorized math; lists are more general but slower for large numeric ops.

Ch2 A1: To reflect oceans/land absorbing a fraction of excess CO_2 each year.
Ch2 A2: Multiply by 1e6 (Mt→t) before applying $/t.
Ch2 A3: Energy emissions drop because each kWh causes fewer kg CO_2.

Ch3a A1: Keeps year-0 as input values; updates start from year-1.
Ch3a A2: A 2% annual decline multiplier; compounding over time.
Ch3a A3: Prevents net from going negative (physically inconsistent).

Ch3b A1: Applies the sector's annual decline multiplier to its current value.
Ch3b A2: 0.90 (since −10%/yr).
Ch3b A3: Because capture applies to total gross across all sectors.

Ch5 A1: `twinx()` gives a second y-axis to plot variables with different units/scales.
Ch5 A2: By concatenating handles/labels from `ax1` and `ax2` and passing to `ax1.legend`.
Ch5 A3: The last row (final year), used to summarize end-state values.

CCUS Simulator — Line-by-Line Manual

Preamble (before first section header)

```
# === CCUS GUI: Default scenario + Agent-based, with cost & energy, ppm, and live plotting ===
```

Comment: === CCUS GUI: Default scenario + Agent-based, with cost & energy, ppm, and live plotting ===

```
import numpy as np
```

Import: loads NumPy (high-performance arrays and math). Used for numeric vectors and element-wise operations.

```
import pandas as pd
```

Import: loads Pandas (tables / DataFrames). Used to assemble and return tidy tabular results.

```
import matplotlib.pyplot as plt
```

Import: loads Matplotlib's plotting API as `plt` to draw charts (lines, labels, legends).

```
from dataclasses import dataclass
```

Import: brings in the `dataclass` decorator, which auto-generates init and other methods for simple classes.

```
from ipywidgets import (
```

Import: loads ipywidgets widgets (sliders, dropdowns) for interactive controls in Jupyter.

```
    FloatSlider, IntSlider, Dropdown, Label, GridBox, Layout,
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
    interactive_output, VBox
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
)
```

This line contributes to the overall logic using standard Python syntax.

helpers & data "cards"

```

```

Blank line: separates logical blocks to improve readability.

```
@dataclass
```

Decorator: marks the following class as a dataclass; Python will auto-create boilerplate like the constructor.

```
class CostEnergyKnobs:
```

Class definition: creates a custom data structure (fields defined on the following indented lines).

```
    ccus_cost_usd_per_t0: float = 120.0   # starting $/t captured (tech-only)
```

Assignment: sets `ccus_cost_usd_per_t0: float` to `120.0   # starting $/t captured (tech-only)` to store an intermediate or final value.

```
    ccus_learning_rate_pct: float = 5.0   # cost ↓ %/yr
```

Assignment: sets `ccus_learning_rate_pct: float` to `5.0   # cost ↓ %/yr` to store an intermediate or final value.

```
    ccus_energy_kWh_per_t0: float = 300.0 # kWh per t captured (start)
```

Assignment: sets `ccus_energy_kWh_per_t0: float` to `300.0 # kWh per t captured (start)` to store an intermediate or final value.

```
    ccus_energy_improve_pct: float = 2.0  # energy ↓ %/yr
```

Assignment: sets `ccus_energy_improve_pct: float` to `2.0  # energy ↓ %/yr` to store an intermediate or final value.

```
    power_price_usd_per_kwh: float = 0.07 # $/kWh
```

Assignment: sets `power_price_usd_per_kwh: float` to `0.07 # $/kWh` to store an intermediate or final value.

```
    grid_kgCO2_per_kwh: float = 0.40      # kg CO2 / kWh
```

Assignment: sets `grid_kgCO2_per_kwh: float` to `0.40      # kg CO2 / kWh` to store an intermediate or final value.

```

```

Blank line: separates logical blocks to improve readability.

```
def _ppm_from_net_series(net_mt_series, starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2):
```

Function definition: declares a reusable block of code. Parameters in parentheses are the inputs.

```
    ppm_vals, ppm_now = [], float(starting_ppm)
```

Assignment: sets `ppm_vals, ppm_now` to `[], float(starting_ppm)` to store an intermediate or final value.

```
    for net_mt in np.asarray(net_mt_series, dtype=float):
```

Loop: repeats the indented block for each item or while a condition holds.

```
        add_ppm = (net_mt / 1000.0) * ppm_per_GtCO2
```

Assignment: sets `add_ppm` to `(net_mt / 1000.0) * ppm_per_GtCO2` to store an intermediate or final value.

```
        excess  = ppm_now - preindustrial_ppm
```

Assignment: sets `excess` to `ppm_now - preindustrial_ppm` to store an intermediate or final value.

```
        sink    = k_sink * excess
```

Assignment: sets `sink` to `k_sink * excess` to store an intermediate or final value.

```
        ppm_now = ppm_now + add_ppm - sink
```

Assignment: sets `ppm_now` to `ppm_now + add_ppm - sink` to store an intermediate or final value.

```
        ppm_vals.append(ppm_now)
```

This line contributes to the overall logic using standard Python syntax.

```
    return np.array(ppm_vals, dtype=float)
```

Return: sends a value back to the caller and exits the function.

```

```

Blank line: separates logical blocks to improve readability.

```
def _cost_energy_from_captured(captured_mt, knobs: CostEnergyKnobs,
```

Function definition: declares a reusable block of code. Parameters in parentheses are the inputs.

```
                               cost0, learn_pct, kwh0, improve_pct, power_price, grid_kg_per_kwh):
```

This line contributes to the overall logic using standard Python syntax.

```
    cap = np.asarray(captured_mt, dtype=float)
```

Assignment: sets `cap` to `np.asarray(captured_mt, dtype=float)` to store an intermediate or final value.

```
    n = len(cap)
```

Assignment: sets `n` to `len(cap)` to store an intermediate or final value.

```
    cost_per_t = cost0 * (1.0 - learn_pct/100.0) ** np.arange(n)
```

Assignment: sets `cost_per_t` to `cost0 * (1.0 - learn_pct/100.0) ** np.arange(n)` to store an intermediate or final value.

```
    kwh_per_t  = kwh0  * (1.0 - improve_pct/100.0) ** np.arange(n)
```

Assignment: sets `kwh_per_t` to `kwh0  * (1.0 - improve_pct/100.0) ** np.arange(n)` to store an intermediate or final value.

```
    tech_cost_usd_b   = cap * cost_per_t / 1e6
```

Assignment: sets `tech_cost_usd_b` to `cap * cost_per_t / 1e6` to store an intermediate or final value.

```
    energy_twh        = cap * kwh_per_t / 1000.0
```

Assignment: sets `energy_twh` to `cap * kwh_per_t / 1000.0` to store an intermediate or final value.

```
    energy_bill_usd_b = (cap * kwh_per_t * power_price) / 1e9
```

Assignment: sets `energy_bill_usd_b` to `(cap * kwh_per_t * power_price) / 1e9` to store an intermediate or final value.

```
    energy_emis_mt    = cap * kwh_per_t * grid_kg_per_kwh / 1000.0
```

Assignment: sets `energy_emis_mt` to `cap * kwh_per_t * grid_kg_per_kwh / 1000.0` to store an intermediate or final value.

```
    return (cost_per_t, kwh_per_t, tech_cost_usd_b, energy_twh, energy_bill_usd_b, energy_emis_mt)
```

Return: sends a value back to the caller and exits the function.

simulators

```

```

Blank line: separates logical blocks to improve readability.

```
def simulate_default_with_cost_energy(start_year, end_year,
```

Function definition: declares a reusable block of code. Parameters in parentheses are the inputs.

```
                        gross_start_mt, gross_decline_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
                        ccus_start_mt, ccus_growth_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
                        starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,
```

This line contributes to the overall logic using standard Python syntax.

```
                        knobs: CostEnergyKnobs) -> pd.DataFrame:
```

Pandas: constructs a DataFrame (table) from a dictionary of column name → data series.

```
    years = list(range(int(start_year), int(end_year)+1))
```

Assignment: builds a list of integer years from start to end inclusive.

```
    g = float(gross_start_mt)
```

Assignment: sets `g` to `float(gross_start_mt)` to store an intermediate or final value.

```
    c = float(ccus_start_mt)
```

Assignment: sets `c` to `float(ccus_start_mt)` to store an intermediate or final value.

```
    decline = 1.0 - float(gross_decline_rate_pct)/100.0
```

Assignment: computes `decline` as a yearly multiplier (1.0 - float(gross_decline_rate_pct)/100.0); e.g., 0.98 for −2%/yr or 1.20 for +20%/yr.

```
    growth  = 1.0 + float(ccus_growth_rate_pct)/100.0
```

Assignment: computes `growth` as a yearly multiplier (1.0 + float(ccus_growth_rate_pct)/100.0); e.g., 0.98 for −2%/yr or 1.20 for +20%/yr.

```

```

Blank line: separates logical blocks to improve readability.

```
    gross, captured = [], []
```

Assignment: sets `gross, captured` to `[], []` to store an intermediate or final value.

```
    for i, _ in enumerate(years):
```

Loop: iterates over the sequence with both index and value. `enumerate` yields (index, value).

```
        if i > 0:
```

Conditional: runs the following indented block only if this condition is True.

```
            g *= decline
```

In-place multiplication: updates `g` by multiplying it by `decline` (compounding effect across iterations).

```
            c *= growth
```

In-place multiplication: updates `c` by multiplying it by `growth` (compounding effect across iterations).

```
        gross.append(g)
```

This line contributes to the overall logic using standard Python syntax.

```
        captured.append(c)
```

This line contributes to the overall logic using standard Python syntax.

```
    gross = np.array(gross, dtype=float)
```

NumPy: converts a Python list into a NumPy array (fast numeric container). `dtype=float` ensures floating-point math.

```
    captured = np.array(captured, dtype=float)
```

NumPy: converts a Python list into a NumPy array (fast numeric container). `dtype=float` ensures floating-point math.

```
    net = gross - captured
```

Assignment: `net` stores a series used in the simulation.

```

```

Blank line: separates logical blocks to improve readability.

```
    ppm_vals = _ppm_from_net_series(net, starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2)
```

Assignment: sets `ppm_vals` to `_ppm_from_net_series(net, starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2)` to store an intermediate or final value.

```

```

Blank line: separates logical blocks to improve readability.

```
    (cost_per_t, kwh_per_t, tech_cost_b,
```

This line contributes to the overall logic using standard Python syntax.

```
     energy_twh, energy_bill_b, energy_emis_mt) = _cost_energy_from_captured(
```

Assignment: sets `energy_twh, energy_bill_b, energy_emis_mt)` to `_cost_energy_from_captured(` to store an intermediate or final value.

```
        captured, knobs,
```

This line contributes to the overall logic using standard Python syntax.

```
        knobs.ccus_cost_usd_per_t0, knobs.ccus_learning_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        knobs.ccus_energy_kWh_per_t0, knobs.ccus_energy_improve_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        knobs.power_price_usd_per_kwh, knobs.grid_kgCO2_per_kwh
```

This line contributes to the overall logic using standard Python syntax.

```
    )
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
    df = pd.DataFrame({
```

Pandas: constructs a DataFrame (table) from a dictionary of column name → data series.

```
        "Year": years,
```

This line contributes to the overall logic using standard Python syntax.

```
        "Gross_Mt": gross,
```

This line contributes to the overall logic using standard Python syntax.

```
        "Captured_Mt": captured,
```

This line contributes to the overall logic using standard Python syntax.

```
        "Net_Mt": net,
```

This line contributes to the overall logic using standard Python syntax.

```
        "Atmospheric_CO2_ppm": ppm_vals,
```

This line contributes to the overall logic using standard Python syntax.

```
        "CCUS_Cost_USD_per_t": cost_per_t,
```

This line contributes to the overall logic using standard Python syntax.

```
        "CCUS_Energy_kWh_per_t": kwh_per_t,'
```

This line contributes to the overall logic using standard Python syntax.

```
        "CCUS_Tech_Cost_USD_B": tech_cost_b,
```

This line contributes to the overall logic using standard Python syntax.

```
        "CCUS_Energy_TWh": energy_twh,
```

This line contributes to the overall logic using standard Python syntax.

```
        "CCUS_Energy_Bill_USD_B": energy_bill_b,
```

This line contributes to the overall logic using standard Python syntax.

```
        "Energy_Emissions_Mt": energy_emis_mt,
```

This line contributes to the overall logic using standard Python syntax.

```
    })
```

This line contributes to the overall logic using standard Python syntax.

```
    df["Effective_Net_Mt"] = df["Net_Mt"] + df["Energy_Emissions_Mt"]
```

Assignment: sets `df["Effective_Net_Mt"]` to `df["Net_Mt"] + df["Energy_Emissions_Mt"]` to store an intermediate or final value.

```
    df["CCUS_Total_Spend_USD_B"] = df["CCUS_Tech_Cost_USD_B"] +
df["CCUS_Energy_Bill_USD_B"]
```

Assignment: sets `df["CCUS_Total_Spend_USD_B"]` to `df["CCUS_Tech_Cost_USD_B"] + df["CCUS_Energy_Bill_USD_B"]` to store an intermediate or final value.

```
    return df
```

Return: sends a value back to the caller and exits the function.

```

```

Blank line: separates logical blocks to improve readability.

```
ABM_SECTORS = ["Power","Industry","Transport","Buildings","Other"]
```

Assignment: sets `ABM_SECTORS` to `["Power","Industry","Transport","Buildings","Other"]` to store an intermediate or final value.

```

```

Blank line: separates logical blocks to improve readability.

```
def simulate_abm_with_cost_energy(start_year, end_year,
```

Function definition: declares a reusable block of code. Parameters in parentheses are the inputs.

```
                              sector_gross_start_mt: dict, sector_decline_pct: dict,
```

This line contributes to the overall logic using standard Python syntax.

```
                              ccus_start_mt, ccus_growth_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
                              starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,
```

This line contributes to the overall logic using standard Python syntax.

```
                              knobs: CostEnergyKnobs) -> pd.DataFrame:
```

Pandas: constructs a DataFrame (table) from a dictionary of column name → data series.

```
    years = list(range(int(start_year), int(end_year)+1))
```

Assignment: builds a list of integer years from start to end inclusive.

```
    current  = {s: float(sector_gross_start_mt[s]) for s in ABM_SECTORS}
```

Dictionary comprehension: creates a dictionary in one expression by looping and assigning key→value pairs.

```
    declines = {s: 1.0 - float(sector_decline_pct[s])/100.0 for s in ABM_SECTORS}
```

Dictionary comprehension: creates a dictionary in one expression by looping and assigning key→value pairs.

```
    c = float(ccus_start_mt)
```

Assignment: sets `c` to `float(ccus_start_mt)` to store an intermediate or final value.

```
    growth = 1.0 + float(ccus_growth_rate_pct)/100.0
```

Assignment: computes `growth` as a yearly multiplier (1.0 + float(ccus_growth_rate_pct)/100.0); e.g., 0.98 for −2%/yr or 1.20 for +20%/yr.

```

```

Blank line: separates logical blocks to improve readability.

```
    gross_list, captured_list, net_list = [], [], []
```

Assignment: sets `gross_list, captured_list, net_list` to `[], [], []` to store an intermediate or final value.

```
    for i, _ in enumerate(years):
```

Loop: iterates over the sequence with both index and value. `enumerate` yields (index, value).

```
        if i > 0:
```

Conditional: runs the following indented block only if this condition is True.

```
            for s in ABM_SECTORS:
```

Loop: repeats the indented block for each item or while a condition holds.

```
                current[s] *= declines[s]
```

In-place multiplication: updates `current[s]` by multiplying it by `declines[s]` (compounding effect across iterations).

```
            c *= growth
```

In-place multiplication: updates `c` by multiplying it by `growth` (compounding effect across iterations).

```
        g_total = sum(current.values())
```

Assignment: sets `g_total` to `sum(current.values())` to store an intermediate or final value.

```
        gross_list.append(g_total)
```

This line contributes to the overall logic using standard Python syntax.

```
        captured_list.append(c)
```

This line contributes to the overall logic using standard Python syntax.

```
        net_list.append(g_total - c)
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
    gross = np.array(gross_list, dtype=float)
```

NumPy: converts a Python list into a NumPy array (fast numeric container). `dtype=float` ensures floating-point math.

```
    captured = np.array(captured_list, dtype=float)
```

NumPy: converts a Python list into a NumPy array (fast numeric container). `dtype=float` ensures floating-point math.

```
    net = np.array(net_list, dtype=float)
```

NumPy: converts a Python list into a NumPy array (fast numeric container). `dtype=float` ensures floating-point math.

```

```

Blank line: separates logical blocks to improve readability.

```
    ppm_vals = _ppm_from_net_series(net, starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2)
```

Assignment: sets `ppm_vals` to `_ppm_from_net_series(net, starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2)` to store an intermediate or final value.

```

```

Blank line: separates logical blocks to improve readability.

```
    (cost_per_t, kwh_per_t, tech_cost_b,
```

This line contributes to the overall logic using standard Python syntax.

```
     energy_twh, energy_bill_b, energy_emis_mt) = _cost_energy_from_captured(
```

Assignment: sets `energy_twh, energy_bill_b, energy_emis_mt)` to `_cost_energy_from_captured(` to store an intermediate or final value.

```
        captured, knobs,
```

This line contributes to the overall logic using standard Python syntax.

```
        knobs.ccus_cost_usd_per_t0, knobs.ccus_learning_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        knobs.ccus_energy_kWh_per_t0, knobs.ccus_energy_improve_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        knobs.power_price_usd_per_kwh, knobs.grid_kgCO2_per_kwh
```

This line contributes to the overall logic using standard Python syntax.

```
    )
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
    df = pd.DataFrame({
```

Pandas: constructs a DataFrame (table) from a dictionary of column name → data series.

```
        "Year": years,
```

This line contributes to the overall logic using standard Python syntax.

```
        "Gross_Mt": gross,
```

This line contributes to the overall logic using standard Python syntax.

```
        "Captured_Mt": captured,
```

This line contributes to the overall logic using standard Python syntax.

```
        "Net_Mt": net,
```

This line contributes to the overall logic using standard Python syntax.

```
        "Atmospheric_CO2_ppm": ppm_vals,
```

This line contributes to the overall logic using standard Python syntax.

```
        "CCUS_Cost_USD_per_t": cost_per_t,
```

This line contributes to the overall logic using standard Python syntax.

```
        "CCUS_Energy_kWh_per_t": kwh_per_t,
```

This line contributes to the overall logic using standard Python syntax.

```
        "CCUS_Tech_Cost_USD_B": tech_cost_b,
```

This line contributes to the overall logic using standard Python syntax.

```
        "CCUS_Energy_TWh": energy_twh,
```

This line contributes to the overall logic using standard Python syntax.

```
        "CCUS_Energy_Bill_USD_B": energy_bill_b,
```

This line contributes to the overall logic using standard Python syntax.

```
        "Energy_Emissions_Mt": energy_emis_mt,
```

This line contributes to the overall logic using standard Python syntax.

```
    })
```

This line contributes to the overall logic using standard Python syntax.

```
    df["Effective_Net_Mt"] = df["Net_Mt"] + df["Energy_Emissions_Mt"]
```

Assignment: sets `df["Effective_Net_Mt"]` to `df["Net_Mt"] + df["Energy_Emissions_Mt"]` to store an intermediate or final value.

```
    df["CCUS_Total_Spend_USD_B"] = df["CCUS_Tech_Cost_USD_B"] +
df["CCUS_Energy_Bill_USD_B"]
```

Assignment: sets `df["CCUS_Total_Spend_USD_B"]` to `df["CCUS_Tech_Cost_USD_B"] + df["CCUS_Energy_Bill_USD_B"]` to store an intermediate or final value.

```
    return df
```

Return: sends a value back to the caller and exits the function.

widgets

```

```

Blank line: separates logical blocks to improve readability.

```
mode_dd = Dropdown(options=["Default scenario (live)", "Agent-based (live)"],
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
                   value="Default scenario (live)")
```

Assignment: sets `value` to `"Default scenario (live)")` to store an intermediate or final value.

```

```

Blank line: separates logical blocks to improve readability.

```
start_year = IntSlider(min=2020, max=2030, step=1, value=2025)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
end_year   = IntSlider(min=2030, max=2060, step=1, value=2050)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```

```

Blank line: separates logical blocks to improve readability.

```
gross_start_mt         = IntSlider(min=10000, max=60000, step=500, value=35000)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
gross_decline_rate_pct = FloatSlider(min=0.0, max=5.0, step=0.1, value=0.7)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
ccus_start_mt         = IntSlider(min=0, max=500, step=5, value=40)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
ccus_growth_rate_pct   = FloatSlider(min=0.0, max=50.0, step=0.5, value=15.0)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```

```

Blank line: separates logical blocks to improve readability.

```
sector_start = {s: IntSlider(min=0, max=30000, step=250, value=v) for s, v in
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
                zip(ABM_SECTORS, [12000, 9000, 7000, 5000, 2000])}
```

This line contributes to the overall logic using standard Python syntax.

```
sector_decl  = {s: FloatSlider(min=0.0, max=10.0, step=0.1, value=v) for s, v in
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
                zip(ABM_SECTORS, [2.5, 1.5, 1.0, 1.8, 0.5])}
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
starting_ppm       = FloatSlider(min=350, max=500, step=1, value=420)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
preindustrial_ppm   = FloatSlider(min=250, max=300, step=0.5, value=280)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
k_sink             = FloatSlider(min=0.0, max=0.05, step=0.001, value=0.012)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
ppm_per_GtCO2       = FloatSlider(min=0.08, max=0.20, step=0.001, value=1.0/7.8)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```

```

Blank line: separates logical blocks to improve readability.

```
ccus_cost_usd_per_t0   = FloatSlider(min=20,  max=600,  step=5,   value=120)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
ccus_learning_rate_pct = FloatSlider(min=0,   max=30,   step=0.5, value=5.0)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
ccus_energy_kwh_per_t0 = FloatSlider(min=50,  max=1200, step=10,  value=300)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
ccus_energy_improve_pct= FloatSlider(min=0,   max=20,   step=0.5, value=2.0)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
power_price_usd_per_kwh= FloatSlider(min=0.02,max=0.3,  step=0.005,value=0.07)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
grid_kgCO2_per_kwh     = FloatSlider(min=0.0, max=0.9,  step=0.01, value=0.40)
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```

```

Blank line: separates logical blocks to improve readability.

```
def _grid_rows(rows):
```

Function definition: declares a reusable block of code. Parameters in parentheses are the inputs.

```
    children = []
```

Assignment: sets `children` to `[]` to store an intermediate or final value.

```
    for label, widget in rows:
```

Loop: repeats the indented block for each item or while a condition holds.

```
        children += [Label(label), widget]
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
    return GridBox(children=children,
```

Return: sends a value back to the caller and exits the function.

```
                   layout=Layout(grid_template_columns="220px 520px", grid_gap="6px 12px", 
width="800px"))
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```

```

Blank line: separates logical blocks to improve readability.

```
grid_mode = _grid_rows([("Mode:", mode_dd)])
```

Assignment: sets `grid_mode` to `_grid_rows([("Mode:", mode_dd)])` to store an intermediate or final value.

```

```

Blank line: separates logical blocks to improve readability.

```
grid_default = _grid_rows([
```

Assignment: sets `grid_default` to `_grid_rows([` to store an intermediate or final value.

```
    ("Start year:", start_year), ("End year:", end_year),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("Gross start (Mt):", gross_start_mt),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("Gross decline %/yr:", gross_decline_rate_pct),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("CCUS start (Mt):", ccus_start_mt),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("CCUS growth %/yr:", ccus_growth_rate_pct),
```

This line contributes to the overall logic using standard Python syntax.

```
])
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
grid_abm = _grid_rows(
```

Assignment: sets `grid_abm` to `_grid_rows(` to store an intermediate or final value.

```
    [("Start year:", start_year), ("End year:", end_year)] +
```

This line contributes to the overall logic using standard Python syntax.

```
    [(f"{s} start (Mt):", sector_start[s]) for s in ABM_SECTORS] +
```

List comprehension: creates a list from an expression applied to each item in a loop.

```
    [(f"{s} decline %/yr:", sector_decl[s]) for s in ABM_SECTORS] +
```

List comprehension: creates a list from an expression applied to each item in a loop.

```
    [("CCUS start (Mt):", ccus_start_mt), ("CCUS growth %/yr:", ccus_growth_rate_pct)]
```

This line contributes to the overall logic using standard Python syntax.

```
)
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
grid_ppm = _grid_rows([
```

Assignment: sets `grid_ppm` to `_grid_rows([` to store an intermediate or final value.

```
    ("Start ppm:", starting_ppm),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("Preindustrial ppm:", preindustrial_ppm),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("Sink rate /yr:", k_sink),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("ppm per GtCO₂:", ppm_per_GtCO2),
```

This line contributes to the overall logic using standard Python syntax.

```
])
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
grid_cost = _grid_rows([
```

Assignment: sets `grid_cost` to `_grid_rows([` to store an intermediate or final value.

```
    ("CCUS $/t (start):", ccus_cost_usd_per_t0),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("Learning %/yr:", ccus_learning_rate_pct),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("Energy kWh/t (start):", ccus_energy_kwh_per_t0),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("Energy improve %/yr:", ccus_energy_improve_pct),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("Power price $/kWh:", power_price_usd_per_kwh),
```

This line contributes to the overall logic using standard Python syntax.

```
    ("Grid kgCO₂/kWh:", grid_kgCO2_per_kwh),
```

This line contributes to the overall logic using standard Python syntax.

```
])
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
def _toggle_grids(*_):
```

Function definition: declares a reusable block of code. Parameters in parentheses are the inputs.

```
    if mode_dd.value == "Default scenario (live)":
```

Conditional: runs the following indented block only if this condition is True.

```
        grid_default.layout.display = ""
```

Widget layout: configures how a widget is displayed (e.g., show/hide with `display` or set grid width).

```
        grid_abm.layout.display = "none"
```

Widget layout: configures how a widget is displayed (e.g., show/hide with `display` or set grid width).

```
    else:
```

Conditional (else): runs if none of the prior conditions were True.

```
        grid_default.layout.display = "none"
```

Widget layout: configures how a widget is displayed (e.g., show/hide with `display` or set grid width).

```
        grid_abm.layout.display = ""
```

Widget layout: configures how a widget is displayed (e.g., show/hide with `display` or set grid width).

```
mode_dd.observe(_toggle_grids, names="value")
```

Observer: when the dropdown changes, call the given function to show/hide relevant grids.

```
_toggle_grids()
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
def _read_sector_widgets():
```

Function definition: declares a reusable block of code. Parameters in parentheses are the inputs.

```
    sector_gross = {s: float(sector_start[s].value) for s in ABM_SECTORS}
```

Dictionary comprehension: creates a dictionary in one expression by looping and assigning key→value pairs.

```
    sector_declp = {s: float(sector_decl[s].value)  for s in ABM_SECTORS}
```

Dictionary comprehension: creates a dictionary in one expression by looping and assigning key→value pairs.

```
    return sector_gross, sector_declp
```

Return: sends a value back to the caller and exits the function.

```

```

Blank line: separates logical blocks to improve readability.

```
def _ui(mode,
```

Function definition: declares a reusable block of code. Parameters in parentheses are the inputs.

```
        gross_start_mt, gross_decline_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        ccus_start_mt, ccus_growth_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        start_year, end_year,
```

This line contributes to the overall logic using standard Python syntax.

```
        starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,
```

This line contributes to the overall logic using standard Python syntax.

```
        ccus_cost_usd_per_t0, ccus_learning_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        ccus_energy_kwh_per_t0, ccus_energy_improve_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        power_price_usd_per_kwh, grid_kgCO2_per_kwh):
```

This line contributes to the overall logic using standard Python syntax.

```
    knobs = CostEnergyKnobs(
```

Assignment: sets `knobs` to `CostEnergyKnobs(` to store an intermediate or final value.

```
        ccus_cost_usd_per_t0=float(ccus_cost_usd_per_t0),
```

Assignment: sets `ccus_cost_usd_per_t0` to `float(ccus_cost_usd_per_t0),` to store an intermediate or final value.

```
        ccus_learning_rate_pct=float(ccus_learning_rate_pct),
```

Assignment: sets `ccus_learning_rate_pct` to `float(ccus_learning_rate_pct),` to store an intermediate or final value.

```
        ccus_energy_kWh_per_t0=float(ccus_energy_kwh_per_t0),
```

Assignment: sets `ccus_energy_kWh_per_t0` to `float(ccus_energy_kwh_per_t0),` to store an intermediate or final value.

```
        ccus_energy_improve_pct=float(ccus_energy_improve_pct),
```

Assignment: sets `ccus_energy_improve_pct` to `float(ccus_energy_improve_pct),` to store an intermediate or final value.

```
        power_price_usd_per_kwh=float(power_price_usd_per_kwh),
```

Assignment: sets `power_price_usd_per_kwh` to `float(power_price_usd_per_kwh),` to store an intermediate or final value.

```
        grid_kgCO2_per_kwh=float(grid_kgCO2_per_kwh),
```

Assignment: sets `grid_kgCO2_per_kwh` to `float(grid_kgCO2_per_kwh),` to store an intermediate or final value.

```
    )
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
    if mode == "Default scenario (live)":
```

Conditional: runs the following indented block only if this condition is True.

```
        df = simulate_default_with_cost_energy(
```

Assignment: sets `df` to `simulate_default_with_cost_energy(` to store an intermediate or final value.

```
            start_year, end_year,
```

This line contributes to the overall logic using standard Python syntax.

```
            gross_start_mt, gross_decline_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
            ccus_start_mt, ccus_growth_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
            starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,
```

This line contributes to the overall logic using standard Python syntax.

```
            knobs
```

This line contributes to the overall logic using standard Python syntax.

```
        )
```

This line contributes to the overall logic using standard Python syntax.

```
        used = "default scenario (live)"
```

Assignment: sets `used` to `"default scenario (live)"` to store an intermediate or final value.

```
    else:
```

Conditional (else): runs if none of the prior conditions were True.

```
        sector_gross, sector_declp = _read_sector_widgets()
```

Assignment: sets `sector_gross, sector_declp` to `_read_sector_widgets()` to store an intermediate or final value.

```
        df = simulate_abm_with_cost_energy(
```

Assignment: sets `df` to `simulate_abm_with_cost_energy(` to store an intermediate or final value.

```
            start_year, end_year,
```

This line contributes to the overall logic using standard Python syntax.

```
            sector_gross, sector_declp,
```

This line contributes to the overall logic using standard Python syntax.

```
            ccus_start_mt, ccus_growth_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
            starting_ppm, preindustrial_ppm, k_sink, ppm_per_GtCO2,
```

This line contributes to the overall logic using standard Python syntax.

```
            knobs
```

This line contributes to the overall logic using standard Python syntax.

```
        )
```

This line contributes to the overall logic using standard Python syntax.

```
        used = "agent-based (live)"
```

Assignment: sets `used` to `"agent-based (live)"` to store an intermediate or final value.

```

```

Blank line: separates logical blocks to improve readability.

```
    # plot
```

Comment: plot

```
    plt.figure(figsize=(9,5))
```

Matplotlib plotting: configures the figure, axes, lines, labels, legend, grid, and rendering.

```
    ax1 = plt.gca()
```

Assignment: sets `ax1` to `plt.gca()` to store an intermediate or final value.

```
    ax2 = ax1.twinx()
```

Matplotlib: creates a second y-axis sharing the same x-axis, useful for plotting variables with different units.

```
    ax1.plot(df["Year"], df["Gross_Mt"],    label="Gross (Mt)")
```

Matplotlib plotting: configures the figure, axes, lines, labels, legend, grid, and rendering.

```
    ax1.plot(df["Year"], df["Captured_Mt"], label="Captured (Mt)")
```

Matplotlib plotting: configures the figure, axes, lines, labels, legend, grid, and rendering.

```
    ax1.plot(df["Year"], df["Net_Mt"],      label="Net (Mt)")
```

Matplotlib plotting: configures the figure, axes, lines, labels, legend, grid, and rendering.

```
    ax1.plot(df["Year"], df["Effective_Net_Mt"], label="Effective Net (Mt)", linestyle="--")
```

Matplotlib plotting: configures the figure, axes, lines, labels, legend, grid, and rendering.

```
    ax1.set_xlabel("Year"); ax1.set_ylabel("Emissions (Mt/yr)"); ax1.grid(True)
```

Matplotlib plotting: configures the figure, axes, lines, labels, legend, grid, and rendering.

```
    ax2.plot(df["Year"], df["Atmospheric_CO2_ppm"], label="CO₂ (ppm)")
```

Matplotlib plotting: configures the figure, axes, lines, labels, legend, grid, and rendering.

```
    ax2.set_ylabel("CO₂ (ppm)")
```

Matplotlib plotting: configures the figure, axes, lines, labels, legend, grid, and rendering.

```
    l1, lab1 = ax1.get_legend_handles_labels()
```

Assignment: sets `l1, lab1` to `ax1.get_legend_handles_labels()` to store an intermediate or final value.

```
    l2, lab2 = ax2.get_legend_handles_labels()
```

Assignment: sets `l2, lab2` to `ax2.get_legend_handles_labels()` to store an intermediate or final value.

```
    ax1.legend(l1 + l2, lab1 + l2, loc="upper right")  # combine legends
```

Matplotlib plotting: configures the figure, axes, lines, labels, legend, grid, and rendering.

```
    plt.title(f"Emissions, Effective Net & CO₂ (ppm) — {used}")
```

Matplotlib plotting: configures the figure, axes, lines, labels, legend, grid, and rendering.

```
    plt.show()
```

Matplotlib plotting: configures the figure, axes, lines, labels, legend, grid, and rendering.

```

```

Blank line: separates logical blocks to improve readability.

```
    last = df.iloc[-1]
```

Assignment: sets `last` to `df.iloc[-1]` to store an intermediate or final value.

```
    print(
```

Output: prints a formatted summary line using f-strings (values inserted inside {} with numeric formatting).

```
        f"Final {int(last['Year'])}: "
```

Output: prints a formatted summary line using f-strings (values inserted inside {} with numeric formatting).

```
        f"Captured={last['Captured_Mt']:,.0f} Mt | "
```

Assignment: sets `f"Captured` to `{last['Captured_Mt']:,.0f} Mt | "` to store an intermediate or final value.

```
        f"CCUS $/t={last['CCUS_Cost_USD_per_t']:,.0f} | "
```

Assignment: sets `f"CCUS $/t` to `{last['CCUS_Cost_USD_per_t']:,.0f} | "` to store an intermediate or final value.

```
        f"Tech cost=${last['CCUS_Tech_Cost_USD_B']:,.2f} B | "
```

Assignment: sets `f"Tech cost` to `${last['CCUS_Tech_Cost_USD_B']:,.2f} B | "` to store an intermediate or final value.

```
        f"Energy={last['CCUS_Energy_TWh']:,.1f} TWh | "
```

Assignment: sets `f"Energy` to `{last['CCUS_Energy_TWh']:,.1f} TWh | "` to store an intermediate or final value.

```
        f"Energy bill=${last['CCUS_Energy_Bill_USD_B']:,.2f} B | "
```

Assignment: sets `f"Energy bill` to `${last['CCUS_Energy_Bill_USD_B']:,.2f} B | "` to store an intermediate or final value.

```
        f"Energy emis={last['Energy_Emissions_Mt']:,.1f} Mt | "
```

Assignment: sets `f"Energy emis` to `{last['Energy_Emissions_Mt']:,.1f} Mt | "` to store an intermediate or final value.

```
        f"Total spend=${last['CCUS_Total_Spend_USD_B']:,.2f} B | "
```

Assignment: sets `f"Total spend` to `${last['CCUS_Total_Spend_USD_B']:,.2f} B | "` to store an intermediate or final value.

```
        f"ppm={last['Atmospheric_CO2_ppm']:,.1f}"
```

Assignment: sets `f"ppm` to `{last['Atmospheric_CO2_ppm']:,.1f}"` to store an intermediate or final value.

```
    )
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
# wire sliders -> UI and display
```

Comment: wire sliders -> UI and display

```
out = interactive_output(
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
    _ui,
```

This line contributes to the overall logic using standard Python syntax.

```
    {
```

This line contributes to the overall logic using standard Python syntax.

```
        "mode": mode_dd,
```

This line contributes to the overall logic using standard Python syntax.

```
        "gross_start_mt": gross_start_mt,
```

This line contributes to the overall logic using standard Python syntax.

```
        "gross_decline_rate_pct": gross_decline_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        "ccus_start_mt": ccus_start_mt,
```

This line contributes to the overall logic using standard Python syntax.

```
        "ccus_growth_rate_pct": ccus_growth_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        "start_year": start_year,
```

This line contributes to the overall logic using standard Python syntax.

```
        "end_year": end_year,
```

This line contributes to the overall logic using standard Python syntax.

```
        "starting_ppm": starting_ppm,
```

This line contributes to the overall logic using standard Python syntax.

```
        "preindustrial_ppm": preindustrial_ppm,
```

This line contributes to the overall logic using standard Python syntax.

```
        "k_sink": k_sink,
```

This line contributes to the overall logic using standard Python syntax.

```
        "ppm_per_GtCO2": ppm_per_GtCO2,
```

This line contributes to the overall logic using standard Python syntax.

```
        "ccus_cost_usd_per_t0": ccus_cost_usd_per_t0,
```

This line contributes to the overall logic using standard Python syntax.

```
        "ccus_learning_rate_pct": ccus_learning_rate_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        "ccus_energy_kwh_per_t0": ccus_energy_kwh_per_t0,
```

This line contributes to the overall logic using standard Python syntax.

```
        "ccus_energy_improve_pct": ccus_energy_improve_pct,
```

This line contributes to the overall logic using standard Python syntax.

```
        "power_price_usd_per_kwh": power_price_usd_per_kwh,
```

This line contributes to the overall logic using standard Python syntax.

```
        "grid_kgCO2_per_kwh": grid_kgCO2_per_kwh,
```

This line contributes to the overall logic using standard Python syntax.

```
    }
```

This line contributes to the overall logic using standard Python syntax.

```
)
```

This line contributes to the overall logic using standard Python syntax.

```

```

Blank line: separates logical blocks to improve readability.

```
# assemble UI
```

Comment: assemble UI

```
ui = VBox([grid_mode, grid_default, grid_abm, grid_ppm, grid_cost, out])
```

ipywidgets: builds UI controls or layout. Sliders have min/max/step/value. Layout widgets arrange controls on screen.

```
display(ui)
```

This line contributes to the overall logic using standard Python syntax.