Quantitative Macroeconomics I

Bootcamp 1: Introduction to Python

Moritz Scheidenberger
Paris School of Economics

Nice to meet you!

- Did you receive the course email? Let us know if you missed it.
- Any remaining Python setup problems?
  - Open the course project in Positron and select its `.venv`.
  - Check that the welcome script runs and its plot appears.
- **Check your emails** regularly for updates and schedules.

What **you** should expect from the tutorials

Format of the tutorial sessions

- Complementary to Prof. Broer's class — attend both classes!
- A mix of theory, coding and problem sets — expect **a lot of work**.

Learning objectives

1. Create your toolbox to solve macroeconomic models (QM1: representative agent).
   - Various **numerical methods**, with advantages and drawbacks.
   - **QM2 will build on QM1**, focusing on heterogeneous-agent models.
2. Become familiar with **dynamic programming / recursive methods**.
   - Widely used in macro, labor, theory and structural econometrics.

Be able to solve models used by central banks and academic researchers.

What **we** expect from you

How to succeed in this course?

1. Attend classes, and **ask questions** if you don't understand!
2. Study the exercises and problem sets, **do the thinking yourself**, code regularly.
   - Discuss with peers and help each other understand the solution.
3. Connect computational methods to economic models!
   - Ask us if you don't see the connection.
4. If you notice typos or mistakes in the slides or assignments, email us!
   - We keep improving the materials; your feedback helps us correct mistakes.

One more thing...

Some advice

- Quantitative macro has a **large fixed cost** — invest early.
- Help each other understand the course and methods.

Useful resources — more on the syllabus

- Heer and Maussner, *Dynamic General Equilibrium Modeling*.
- Azzimonti et al., *Macroeconomics*.
- QuantEcon lecture notes — Sargent and Stachurski.
- Advanced computational-methods course materials:
  - Jesús Fernández-Villaverde.
  - Fatih Guvenen.

Grading

- An end-of-semester exam.
- Problem sets are **not graded**.
- **Do the problem sets**: they build the skills you need for the exam.
  - Work through the questions and use difficulties to guide your revision.

Python

Outline for today's bootcamp

1. **Why Python?**
2. Python basics: short examples and a live demonstration
3. Build and solve our first dynamic growth model
   - Fill contained parts of the Solow code.
   - Compare a steady state, a transition and productivity shocks.

Why use Python?

- A **general-purpose language** with tools for many research tasks.
  - Data analysis, empirical work, deep learning and web scraping.
- We learn a focused subset: numerical calculations, simulations and figures.
  - The same skills carry over to other projects.
- **Packages** extend the language with specialised tools.
- A **project environment** keeps its interpreter and packages together.
  - The saved environment specification helps us recreate the setup later.

Python basics

Outline for today's bootcamp

1. Why Python?
2. **Python basics: short examples and a live demonstration**
3. Build and solve our first dynamic growth model
   - Fill contained parts of the Solow code.
   - Compare a steady state, a transition and productivity shocks.

Python · Working in Positron

- **Files** opens project files; the **editor** contains the saved instructions.
- **Variables** shows live values; **Console** runs code and displays output.
- **Plots** displays figures; **Source Control** tracks saved changes with Git.
- **Quick demonstration:** run the file, then one `# %%` section.
  - Run prerequisites first; inspect the values and the plot.
  - Restart Python and rerun: saving preserves code, not the live session.

Values have types

- An **integer** represents whole numbers; a **float** represents floating-point numbers.
  - `4` is an integer; `4.0` is a float, even though its value is whole.
- A **string** stores quoted text; a **Boolean** is `True` or `False`.
- `type(...)` reports the type. Try changing `count` from `4` to `4.0`.

```
1 count = 4
2 rate = 0.2
3 label = "Baseline"
4 finished = False
5 print(type(count), type(rate))        # <class 'int'> <class 'float'>
6 print(type(label), type(finished))    # <class 'str'> <class 'bool'>
```

Assignment and recalculation

- = evaluates the right-hand side, then binds the left-hand name to the result.
  - # starts a comment; Python ignores the rest of that line.
- Changing an input leaves an earlier calculation unchanged.
  - Rerun the expression to calculate with the new input.

```
saving = 0.2
amount = 10 * saving
saving = 0.3
print(amount)          # 2.0: the earlier result
amount = 10 * saving
print(amount)          # 3.0: calculated again
```

Two names and a shared object

- Reassigning a name changes what that name refers to.
- A **list** holds entries in brackets; `[0]` selects its first entry.
  - Two names can refer to the same list; replacing an entry changes that shared list.
- Compare rebinding `a` with changing an entry through `other`.

```
a = 2
b = a
a = 3
print(b)                    # 2: b was not rebound
items = [1, 2]
other = items
other[0] = 9
print(items)                # [9, 2]: the shared object changed
```

Arithmetic and readable expressions

- `*` multiplies and `/` divides; write multiplication explicitly.
- `**` raises to a power: `ratio**2` squares `ratio`. `^` is not exponentiation.
- Parentheses group operations. Compare the two ratios below.
  - Long expressions can continue inside parentheses.

```
numerator = 2 * 3.0
print(numerator / (1 + 0.5))    # 4.0
print(numerator / 1 + 0.5)      # 6.5
ratio = numerator / (1 + 0.5)
print(ratio**2)                 # 16.0
```

Inspect and print values

- Inspect stored values in **Variables**; use `print(...)` to display them in Console.
  - Printing inside an algorithm helps us follow intermediate results.
- An **f-string** inserts values into text: `f"...{value}..."`.
  - `:.1f` displays one decimal place; the stored value stays unchanged.

```
price = 12.345
label = "Price"
print(label, price)                 # Price 12.345
print(f"{label}: {price:.1f}")      # Price: 12.3
print(price)                        # 12.345
```

Python · Packages and imports

Goal: Access additional tools, then call them with inputs.

- Python has built-ins such as `print`; **packages** add tools for particular jobs.
  - NumPy supplies numerical arrays; Matplotlib supplies plotting.
- `as np` gives NumPy a short **alias**; `plt` abbreviates its plotting companion.
  - These are conventional names, used consistently throughout our code.
- Importing uses an installed package; it does not install it.
  - The course environment is prepared before class.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
```

Functions, arguments and help

- The dot selects a function; parentheses call it with **arguments** (inputs).
  - `np.sqrt(9.0)` returns a result that we can assign to a name.
- Inputs can be given **by position** or **by keyword name**.
  - The two rounding calls below both request two decimal places.
- `help(np.round)` shows the inputs, defaults and examples; read `decimals=0`.

```
result = np.sqrt(9.0)
print(result)                          # 3.0
print(np.round(3.14159, 2))              # 3.14
print(np.round(3.14159, decimals=2))     # 3.14
help(np.round)                          # Inspect the documentation
```

Understand errors

- Read the **failing line**, then the **exception name and message**.
  - Here `delivery_fee` has not been defined in the fresh session.
- Use the message to find a missing input or typo; correct it and rerun in order.
- AI can help explain a message: understand the cause before accepting a fix.

```
price = 12.0
total = price + delivery_fee
```

```
Traceback (most recent call last):
  File "traceback_demo.py", line 2, in <module>
    total = price + delivery_fee
NameError: name 'delivery_fee' is not defined
```

First exercise: Solow

The Solow economy

- A closed economy uses capital K_t and workers L_t to produce one good.
- Work **per worker**: $k_t = K_t/L_t$ and $y_t = Y_t/L_t$.
  - $A > 0$ is productivity; $0 < \alpha < 1$ gives diminishing returns to capital.
- Save a fixed fraction s as investment; consume the remainder.

$$Y_t = AK_t^{\alpha} L_t^{1-\alpha} \quad \Rightarrow \quad y_t = Ak_t^{\alpha}$$

$$i_t = sy_t, \qquad c_t = (1-s)y_t, \qquad y_t = c_t + i_t$$

Capital accumulation

- A fraction δ of existing capital depreciates each period.
- Today's investment becomes available for production **next period**.
- The workforce grows: $L_{t+1} = (1+n)L_t$.
  - Divide tomorrow's capital by tomorrow's workforce.

$$K_{t+1} = (1-\delta)K_t + sY_t$$

$$k_{t+1} = \frac{(1-\delta)k_t + sAk_t^{\alpha}}{1+n}$$

Steady state and convergence

- A **steady state** has $k_{t+1} = k_t = k^*$.
  - Investment covers depreciation and population dilution.
- Below the positive steady state, capital rises; above it, capital falls.
  - With the same parameters, positive starting points approach the same k^*.
- Constant productivity implies no sustained growth **per worker**.
  - Aggregate output can still grow with the workforce.

$$k_{t+1} - k_t = \frac{sAk_t^{\alpha} - (\delta + n)k_t}{1 + n}$$

Our route

1 · Define and calibrate the economy

2 · Analytical steady state and diagrams

3 · Simulate transition paths

- 3a · Trace a fixed-horizon path
- 3b · Reuse the simulation

4a · Iterate until convergence

7 · Simulate productivity shocks

Section 1 · Calibrate the economy

- `productivity` is A, `alpha` is α and `s` is the saved fraction of output.
- `delta` is depreciation and `n` is population growth per period. These scalar parameters are supplied.
- Capital k_t, output $y_t = Ak_t^\alpha$ and consumption $c_t = (1-s)y_t$ are **per worker**.

```
productivity = 1.0
alpha = 0.3
s = 0.2
delta = 0.1
n = 0.02
```

Our route

1 · Define and calibrate the economy

2 · Analytical steady state and diagrams

3 · Simulate transition paths

- 3a · Trace a fixed-horizon path
- 3b · Reuse the simulation

4a · Iterate until convergence

7 · Simulate productivity shocks

Apply Python: translate the steady-state equations into arithmetic.

- Investment covers depreciation and population dilution at a positive steady state.
- Match k^* to `k_analytical`, y^* to `y_steady` and c^* to `c_steady`.

$$sA(k^*)^{\alpha} = (n+\delta)k^*$$

$$k^* = \left(\frac{sA}{n+\delta}\right)^{1/(1-\alpha)}, \quad y^* = A(k^*)^{\alpha}, \quad c^* = (1-s)y^*.$$

S02-EQ · Student task

1. **Where:** `main_solow.py`, section 2, **S02-EQ**.
2. **Write:** Complete the right-hand sides of `k_analytical`, `y_steady` and `c_steady` using the preceding equations.
3. **Given:** Names, scalar calibration and print statements.
4. **Run/check:** Run section 2 through the print block. Capital ≈ 2.075, output ≈ 1.245 and consumption ≈ 0.996, all per worker.

Solution

```
k_analytical = (s * productivity / (n + delta)) ** (1 / (1 - alpha))
y_steady     = productivity * k_analytical**alpha
c_steady     = (1 - s) * y_steady
```

Python · Numerical arrays

- Arrays store many numerical values together: **grid inputs** or a **time path**.
  - Apply a formula to the whole grid and store its function values.
- `np.array(...)` builds an array from a list of values in brackets.
- Arithmetic acts **entry by entry**; predict `values + 1`.

```
1 values = np.array([1.0, 2.0, 3.0])
2 print(values * 2)      # [2. 4. 6.]
3 print(values**2)       # [1. 4. 9.]
```

Inspect an array

- `values.size` counts entries; `values.shape` gives dimensions.
  - These **attributes** need no parentheses.
- `values.mean()` computes the mean.
  - A **method** belongs to the object; parentheses call it.

```
values = np.array([1.0, 2.0, 3.0])
print(values.size)      # 3
print(values.shape)     # (3,): one dimension, three entries
print(values.mean())    # 2.0
```

Read and replace entries

- Positions **start at zero**: `[0]` reads the first entry.
- `[-1]` reads the last entry; assignment replaces an entry.
  - It does not insert a new entry or change the array's length.
- Predict the result if we replace position 0 instead of position 1.

```
values = np.array([1.0, 2.0, 3.0])
print(values[0], values[-1])    # 1.0 3.0
values[1] = 8.0
print(values)                   # [1. 8. 3.]
```

Select a slice

- `[start:stop]` includes `start` and **excludes `stop`**.
- Omit `start` to begin at zero; omit `stop` to continue to the end.
  - Count positions, not the values stored there.

```
values = np.array([1.0, 2.0, 3.0])
print(values[:2])     # [1. 2.]
print(values[1:3])    # [2. 3.]
print(values[1:])     # [2. 3.]
```

Arrays with rows and columns

- A matrix stores rows and columns; both positions **start at zero**.
- Shape `(2, 3)` means two rows and three columns.
- `[row, column]` reads one entry; `:` selects an entire row or column.

```
table = np.array([[10, 20, 30], [40, 50, 60]])
print(table.shape)       # (2, 3)
print(table[0, 1])       # 20
print(table[1, :])       # [40 50 60]
```

Four useful array constructors

- `array` uses given values; `linspace` chooses a **number of points**.
- `arange` supplies positions; `zeros` reserves space for later results.
- Use **`help(np.linspace)`** as with `help(np.round)`.

```
print(np.array([2, 5, 9]))      # [2 5 9]
print(np.linspace(0, 2, 3))     # [0. 1. 2.]: endpoints included
print(np.arange(4))             # [0 1 2 3]: stop excluded
print(np.zeros(4))              # [0. 0. 0. 0.]: placeholders
```

Apply Python: evaluate economic formulas over a capital grid.

- `grid_k` contains candidate capital stocks, not economic dates.
- Evaluate investment, replacement and next-period capital at each candidate.
- Positive investment above replacement implies rising capital per worker.

$$i(k) = sAk^{\alpha}, \qquad r(k) = (n + \delta)k$$

$$g(k) = \frac{i(k) + (1 - \delta)k}{1 + n}$$

- `grid_invest`, `grid_replace` and `grid_kprime` store these three arrays.

S02-GRID · Student task

1. **Where:** `main_solow.py`, section 2, **S02-GRID**.
2. **Write:** Use `help(np.linspace)` to construct `grid_k`: 300 points from zero to twice `k_analytical`. Complete all three derived arrays.
3. **Given:** The preceding formulas and all four names.
4. **Run/check:** Use the shape check below for each array. Every shape is `(300,)`; endpoints are 0 and ≈ 4.14913.

```
print(grid_k.shape)    # Repeat for grid_invest, grid_replace, grid_kprime
```

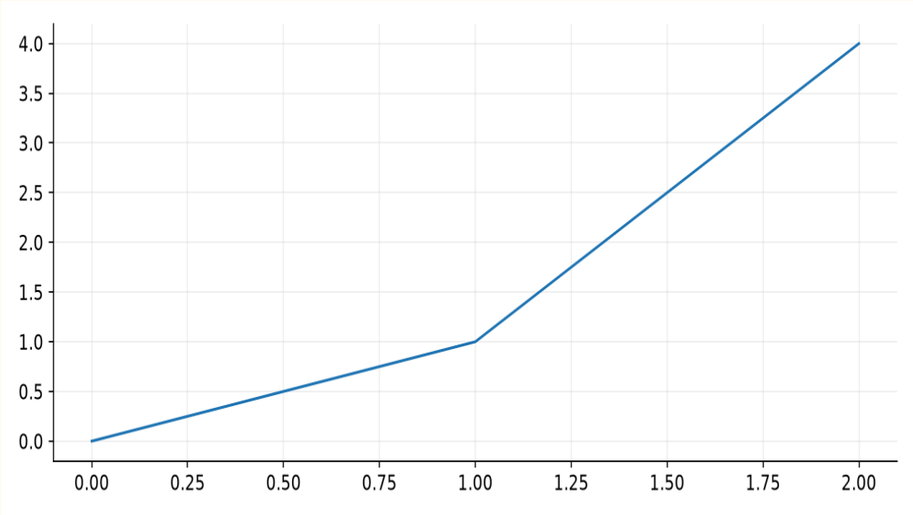
Solution

```
grid_k       = np.linspace(0, 2 * k_analytical, 300)
grid_invest  = s * productivity * grid_k**alpha
grid_replace = (n + delta) * grid_k
grid_kprime  = (grid_invest + (1 - delta) * grid_k) / (1 + n)
```

Python · Draw a line

- `plt.subplots()` creates a **figure** and one **axes** (plotting panel).
  - `fig, ax = ...` receives the two returned objects in order.
- `ax.plot(x, y)` pairs horizontal and vertical entries of matching length.
  - It connects the points in their array order.

```
1 x = np.array([0., 1., 2.])
2 y = x**2
3 fig, ax = plt.subplots()
4 ax.plot(x, y)
```



Labels and plotting panels

- Add another line to the **same panel** with another `ax.plot(...)` call.
  - `label=` names the line; `ax.legend()` shows the names.
- `ax.set(...)` sets the axis labels.

```
1 fig, ax = plt.subplots()
2 ax.plot(x, y, label="Squared")
3 ax.plot(x, x, label="Unchanged")
4 ax.set(xlabel="Input", ylabel="Value")
5 ax.legend()
```

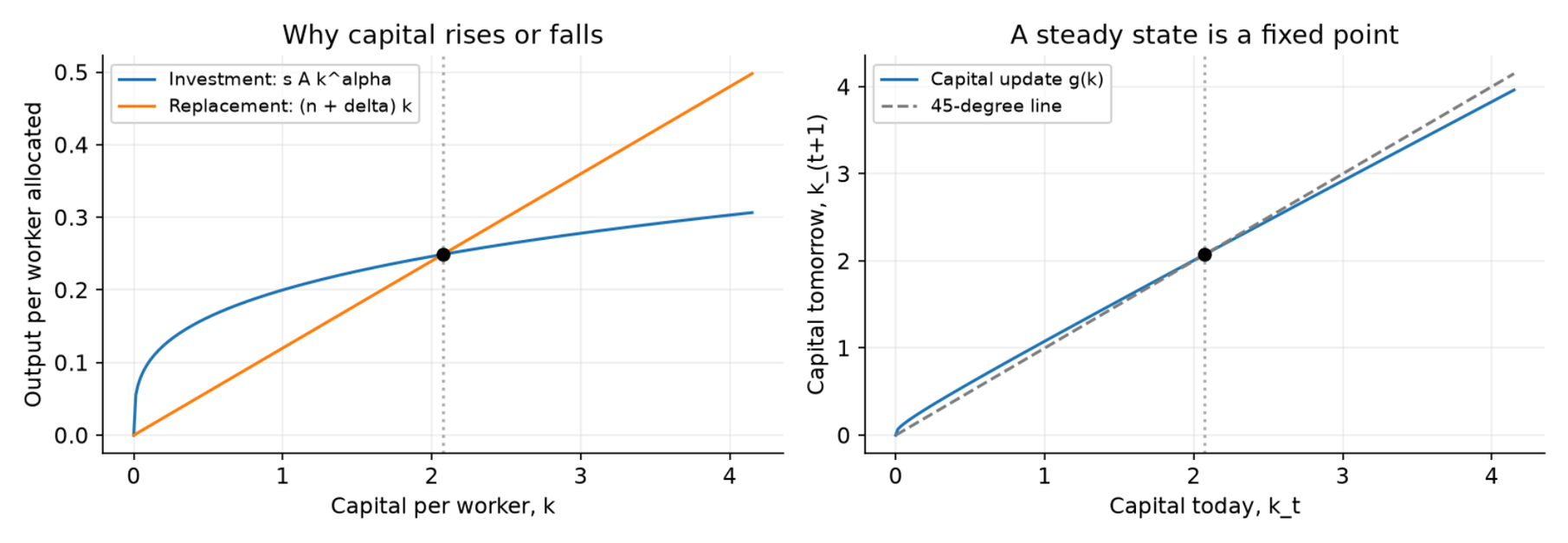
Choose a panel

- `plt.subplots(1, 2)` creates one row with **two panels**.
- `axes[0]` selects the left; `axes[1]` selects the right.
  - With one panel, use `ax` directly; no index is needed.

```
1 fig, axes = plt.subplots(1, 2)
2 axes[0].plot(x, y, label="Squared")
3 axes[1].plot(x, x, label="Unchanged")
```

Section 2 · Choose the diagram coordinates

- Write two `plot` calls on **`axes[0]`**: investment and replacement against capital.
  - Horizontal axis: `grid_k`; vertical axis: investment or replacement.
- The right panel shows next capital and the 45-degree line.



S02-PLOT · Student task

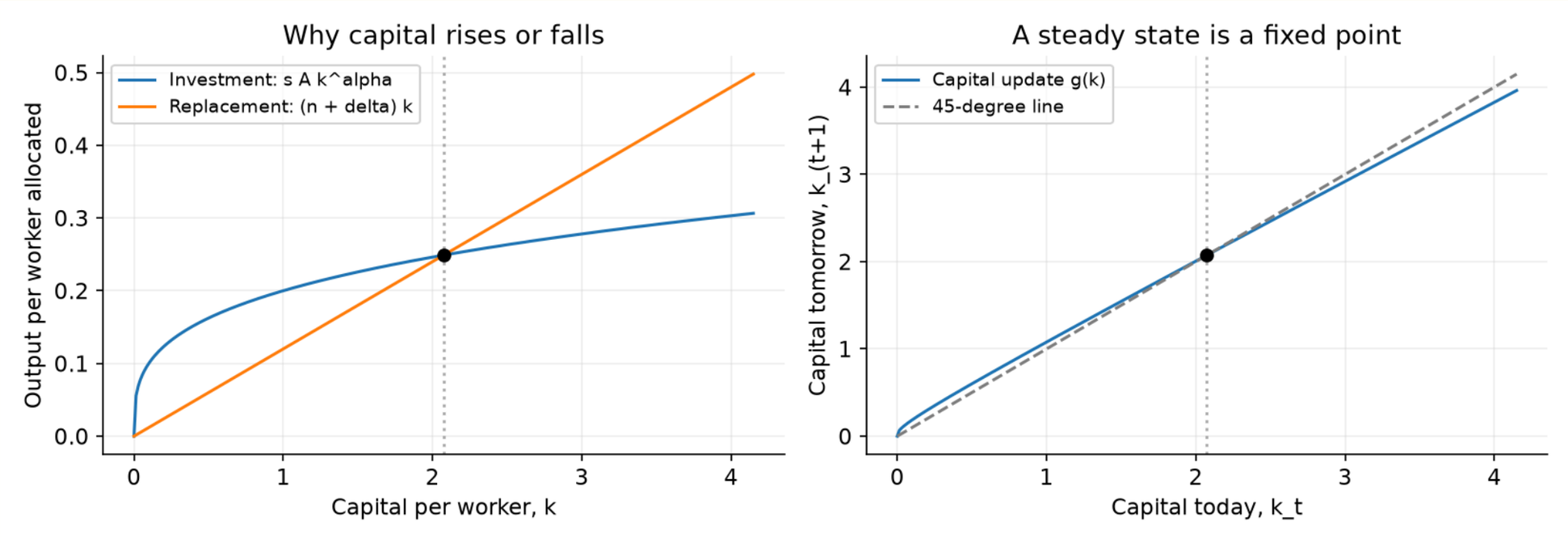
1. **Where:** `main_solow.py`, section 2, **S02-PLOT**.
2. **Write:** Write two `axes[0].plot(...)` calls using `grid_k` horizontally and `grid_invest` or `grid_replace` vertically. Give each curve a descriptive label.
3. **Given:** Figure/axes creation, scatter markers, panel 1 and all formatting.
4. **Run/check:** Run section 2. The investment and replacement lines intersect at positive capital near 2.075. Panel 1 shows g(k) and the 45-degree line.

Solution

```
axes[0].plot(grid_k, grid_invest, label="Investment: s A k^alpha")
axes[0].plot(grid_k, grid_replace, label="Replacement: (n + delta) k")
```

Section 2 · Two steady-state diagrams

- **Left:** investment and replacement intersect at the positive steady state.
- **Right:** next-period capital equals current capital where $g(k) = k$.
- Zero is also a fixed point. Our analytical benchmark is the positive one.



Our route

1 · Define and calibrate the economy

2 · Analytical steady state and diagrams

3 · Simulate transition paths

- **3a · Trace a fixed-horizon path**
- 3b · Reuse the simulation

4a · Iterate until convergence

7 · Simulate productivity shocks

Section 3a · Prepare and simulate a time path

1. **Prepare storage:** reserve entries and record initial capital.
   - 50 updates need 51 states: date zero plus dates 1–50.
2. **Repeat:** read today's capital, calculate tomorrow's, store it.
   - Position `t` represents economic date `t` in this array.

```
example_path = np.zeros(4)    # Space for date 0 and three updates
example_path[0] = 8.0
example_path[1] = example_path[0] / 2
print(example_path)           # [8. 4. 0. 0.]
```

Python · The for loop

Example: repeated updates

- Start with 8 units; halve the remaining amount three times.

- `for` repeats the indented body; `range(3)` supplies **0, 1, 2**.
- `ind_t` receives each value in turn; the colon starts the body.
  - Unindented code runs after the loop. Predict the final amount.

```
1 amount = 8.0
2 for ind_t in range(3):
3     amount = amount / 2
4     print(ind_t, amount)    # 0 4.0, then 1 2.0, then 2 1.0
5 print("Done")
```

Section 3a · Read, calculate and store

- Read `capital_path[ind_t]`; write the new value at **`ind_t + 1`**.
- Calculate investment, retained capital, then next-period capital.
  - The first two updates are the check before running all 50.

$$k_{t+1} = \frac{sAk_t^{\alpha} + (1-\delta)k_t}{1+n}$$

Pass	Read position	Capital today	Capital tomorrow	Write position
1	0	1.00000	1.07843	1
2	1	1.07843	1.15213	2

Section 3a · Locate the exercise

- Keep the supplied starting value and integer horizon.
- Write the storage setup and a **`for` loop** using `ind_t` as its index.
  - Four spaces group the body; each pass uses the result of the previous pass.

```
k0 = 1.0
n_periods = 50
# Construct dates and initialise capital_path.
# Write a for loop using ind_t.
# Calculate and store capital at each next date.
# Supplied output, consumption and figure follow.
```

S03A-PATH · Student task

1. **Where:** `main_solow.py`, section 3a, **S03A-PATH**.
2. **Write:** Construct `dates`, allocate and initialise `capital_path`, then write the loop header and body.
3. **Given:** `k0 = 1.0`, `n_periods = 50`, required variable names, comments and later figures.
4. **Run/check:** Try two updates: capital ≈ 1.07843 then 1.15213. Restore 50 and rerun storage + loop: shape `(51,)`, last value ≈ 2.05829.

Solution

```
k0 = 1.0
n_periods = 50
dates = np.arange(n_periods + 1)
capital_path = np.zeros(n_periods + 1)
capital_path[0] = k0
for ind_t in range(n_periods):
    k_now = capital_path[ind_t]
    investment = s * productivity * k_now**alpha
    retained_capital = (1 - delta) * k_now
    k_next = (investment + retained_capital) / (1 + n)
    capital_path[ind_t + 1] = k_next
```

Section 3a · Inspect a stored path

- Use the familiar slice `capital_path[:3]` to inspect the initial state and first two updates.
- Recover output and consumption entrywise after simulating capital. These lines and the plot are supplied.
- The horizon ends at date 50. A finite horizon alone does not establish convergence.

```
output_path = productivity * capital_path**alpha
consumption_path = (1 - s) * output_path
print(capital_path[:3])     # [1.         1.07843137 1.15212783]
```

Section 3a · A repeated-capital read

- This temporary example always reads **initial capital**, so its path is wrong.
  - The working loop reads the current entry instead; compare the two lines.
- `demo_path` keeps the demonstration separate from your working path.

```
demo_path = np.zeros(3)
demo_path[0] = k0
for ind_t in range(2):
    k_now = k0                      # Wrong: always the initial value.
    # k_now = demo_path[ind_t]      # Working read: current position.
    investment = s * productivity * k_now**alpha
    retained_capital = (1 - delta) * k_now
    k_next = (investment + retained_capital) / (1 + n)
    demo_path[ind_t + 1] = k_next
```

Section 3a · Pause and inspect the loop

1. In your **correct section 3a loop**, find `k_now = capital_path[ind_t]`.
2. Click the margin beside that line to set a **breakpoint**.
3. Start the file with the debugger; it pauses **before** the marked line.
4. Use the **Debug Console** to print values; then choose **Step Over**.
   - Step Over executes one line. Print again to inspect its effect.

```
# At the first pause, before the read:
print(ind_t, capital_path[ind_t])    # 0 1.0
```

```
# After Step Over has executed the read:
print(k_now)                         # 1.0
```

Section 3a · Inspect the next pass

1. Choose **Continue**: the breakpoint pauses at the read on the next pass.
2. Print the index, stored capital and old `k_now` in the **Debug Console**.
3. Choose **Step Over**, then print `k_now` again.
   - In the working loop it changes; the temporary faulty read would keep 1.0.

```
# Before the second read: k_now still belongs to the previous pass.
print(ind_t, np.round(capital_path[ind_t], 5), k_now)
# 1 1.07843 1.0
```

```
# After stepping over the second read:
print(np.round(k_now, 5))            # 1.07843
```

Back to Solow: our Python toolbox

Common starting file: **`main_solow_restart.py`**, complete through **3b**.

- **2 · Formulas, arrays and plots**
  - Calculate k^*, y^* and c^*; evaluate a grid.
  - Plot investment and the capital update.
- **3a · `for`: follow a transition**
  - Read k_t; calculate and store k_{t+1}.
  - Repeat for a fixed number of dates.
- **3b · Functions: supplied bridge**
  - `def` / `return`: inputs and returned path.
  - `par`: group inputs; vary starting capital.
- **4a · `while`: stop at convergence**
  - Update; measure change; append; advance.
  - Count updates; cap at `max_iter`.
  - Compare with analytical k^*.
- **7 · Functions and `for`: shocks**
  - Give a productivity path to a function.
  - Draw seeded normals; compare sigmas.
  - Simulate/store; run supplied **7b**.

Study at home: 4b · root finding; 5 · saving scenarios; 6 · optimisation.

Second half: use arrays, loops and functions to begin PS1.

Section 3b · Supplied functions at the restart

- **`par`** groups the five economy parameters, e.g. `par["s"]`.
- **`fun_simulate`** takes starting capital, the economy and a horizon.
  - It calls `fun_capital_update(k_now, par)` for each date.
  - `return capital_path` sends back the complete capital array.
- Run **0–3b** and inspect the supplied starting-capital comparison in Positron.
  - Restart route: continue with **4a**.
  - Full bootcamp: continue with the detailed **3b** lessons and exercises below.

```
1 reused_path = fun_simulate(k0, par, n_periods)
2 print(reused_path.shape)    # (51,): date 0 plus 50 updates
```

Full bootcamp route · Section 3b

1 · Define and calibrate the economy

2 · Analytical steady state and diagrams

3 · Simulate transition paths

- 3a · Trace a fixed-horizon path
- **3b · Reuse the simulation**

4a · Iterate until convergence

7 · Simulate productivity shocks

Restart: 3b is supplied. Skip its writing/refactoring activities; use the brief function bridge, then continue to 4a.

Python · Define a reusable calculation

- def names a function and its inputs; indent the body.
  - Input names belong to that call; they need no prior assignment.
- The body runs when we **call** the function, not when we define it.
- **return** sends a result back to the caller for storage or further use.

```
1 def order_total(price, fee):
2     subtotal = price + fee
3     return subtotal * 1.1
4
5 answer = order_total(12.0, 3.0)
6 print(answer)    # 16.5
```

Return two results

- A function can return numbers, arrays or several objects.
- `return first, second` returns a **tuple**: an ordered pair here.
  - `base, with_tax = ...` unpacks its entries in the same order.
- This is the same assignment pattern as `fig, ax = plt.subplots()`.

```
def order_amounts(price, fee):
    subtotal = price + fee
    return subtotal, subtotal * 1.1

base, with_tax = order_amounts(12.0, 3.0)
print(base, with_tax)    # 15.0 16.5
```

Return and print

- `print` displays a value; **`return`** lets the caller use it.
- Without a `return`, a function returns `None` (no result).
  - A printed number is not automatically the returned result.

```
1 def show_total(price, fee):
2     print((price + fee) * 1.1)
3
4 shown = show_total(12.0, 3.0)    # Displays 16.5
5 print(shown)                     # None
```

Python · Group named inputs

- A dictionary maps **keys** to values; look up a quoted key in brackets.
- `settings["tax"]` reads a value; assigning to it replaces that value.
  - Key spelling matters; an unknown key raises `KeyError`.
- Group related inputs when long argument lists become hard to read.

```
1 settings = {"tax": 0.1, "fee": 3.0}
2 print(settings["tax"])     # 0.1
3 settings["fee"] = 4.0
4 print(settings["fee"])     # 4.0
```

Python · Copy a dictionary before changing it

- `alias = baseline` gives one dictionary a second name.
- `.copy()` makes an independent dictionary for our **scalar-valued** setup.
  - Change the copy's productivity; preserve the baseline for the next scenario.

```
baseline = {"productivity": 1.0, "s": 0.2}
alias = baseline
independent = baseline.copy()
alias["productivity"] = 1.2
print(baseline["productivity"])       # 1.2: shared object
print(independent["productivity"])    # 1.0: separate dictionary
```

Section 3b · Group the economy parameters

- Repeated calls would otherwise need all five economy inputs.
  - Keep the shorter interface **`fun_simulate(k_start, par, n_periods)`**.
- The supplied `par` groups the calibration; `par["s"]` reads the saving rate.
- Starting capital and the horizon remain separate experiment choices.

```
par = {
    "productivity": productivity,
    "alpha": alpha,
    "s": s,
    "delta": delta,
    "n": n,
}
```

Section 3b · Package the known simulation

- Put the section-3a allocation and update loop inside one function.
- Use `k_start` for the initial entry and dictionary keys for the parameters.
  - For example, `s * productivity` becomes `par["s"] * par["productivity"]`.
- **Return the full path after the loop**, aligned with `for`.
  - First keep the update arithmetic inside; we will extract it for reuse next.

```
def fun_simulate(k_start, par, n_periods):
    # Allocate and initialise the path.
    for ind_t in range(n_periods):
        # Read, compute and store using par's named values.
    # Return the completed path, outside the loop.
```

S03B-SIM · Student task

1. **Where:** `main_solow.py`, section 3b, **S03B-SIM**.
2. **Write:** Write the full `fun_simulate(k_start, par, n_periods)` function using the familiar update arithmetic inside the loop.
3. **Given:** The supplied `par`, signature and section-3a calculation. This first teaching state uses no update helper.
4. **Run/check:** Run `fun_simulate(k0, par, n_periods)`. Shape `(51,)`; compare every entry with `capital_path` using the supplied check.

Solution

```python
def fun_simulate(k_start, par, n_periods):
    capital_path = np.zeros(n_periods + 1)
    capital_path[0] = k_start
    for ind_t in range(n_periods):
        k_now = capital_path[ind_t]
        investment = par["s"] * par["productivity"] * k_now**par["alpha"]
        retained_capital = (1 - par["delta"]) * k_now
        k_next = (investment + retained_capital) / (1 + par["n"])
        capital_path[ind_t + 1] = k_next
    return capital_path
```

Section 3b · Check the packaged simulation

- Call your function with the **same start, economy and horizon** as section 3a.
- The supplied check compares every entry; no output means a pass.
  - `rtol=0, atol=1e-12` allows only tiny numerical differences.

```
reused_path = fun_simulate(k0, par, n_periods)
np.testing.assert_allclose(reused_path, capital_path, rtol=0, atol=1e-12)
print(reused_path.shape)    # (51,)
```

Section 3b · Extract the repeated calculation

- The simulator works. Convergence needs the **same capital update**.
- Move these three calculations into one helper; the helper is supplied.
  - `k` is one capital value; `return` sends back the next value.

```
def fun_capital_update(k, par):
    investment = par["s"] * par["productivity"] * k**par["alpha"]
    retained_capital = (1 - par["delta"]) * k
    return (investment + retained_capital) / (1 + par["n"])
```

Section 3b · Follow a helper call

- The simulator passes **one number and the parameters** to the helper.
- Python runs the helper, returns a number, then continues the simulator.
  - This calls another function; neither function calls itself.

```
k_now = 1.0
k_next = fun_capital_update(k_now, par)
print(k_now, np.round(k_next, 8))    # 1.0 1.07843137
```

Section 3b · Replace the inline arithmetic

- Keep allocation, initialisation, the loop and the returned path.
- **Replace** these three lines with one call to the supplied helper.
  - Remove the old calculations; do not keep both versions.

```
        investment = par["s"] * par["productivity"] * k_now**par["alpha"]
        retained_capital = (1 - par["delta"]) * k_now
        k_next = (investment + retained_capital) / (1 + par["n"])
```

S03B-CALL · Student task

1. **Where:** `main_solow.py`, section 3b, inside `fun_simulate`.
2. **Write:** Replace the three arithmetic lines with one assignment to `k_next`, calling `fun_capital_update` with `k_now` and `par`.
3. **Given:** The helper, read/store lines and the rest of your simulator.
4. **Run/check:** Rerun definitions and the supplied path check. Every entry should still match section 3a.

Solution

```
1          k_next = fun_capital_update(k_now, par)
```

Section 3b · One simulator, with a shared update

- **Only the update calculation moved**; inputs and returned path stay the same.
- Use this version from now on. Rerun the whole-path check after defining it.

```
def fun_simulate(k_start, par, n_periods):
    capital_path = np.zeros(n_periods + 1)
    capital_path[0] = k_start
    for ind_t in range(n_periods):
        k_now = capital_path[ind_t]
        k_next = fun_capital_update(k_now, par)
        capital_path[ind_t + 1] = k_next
    return capital_path
```

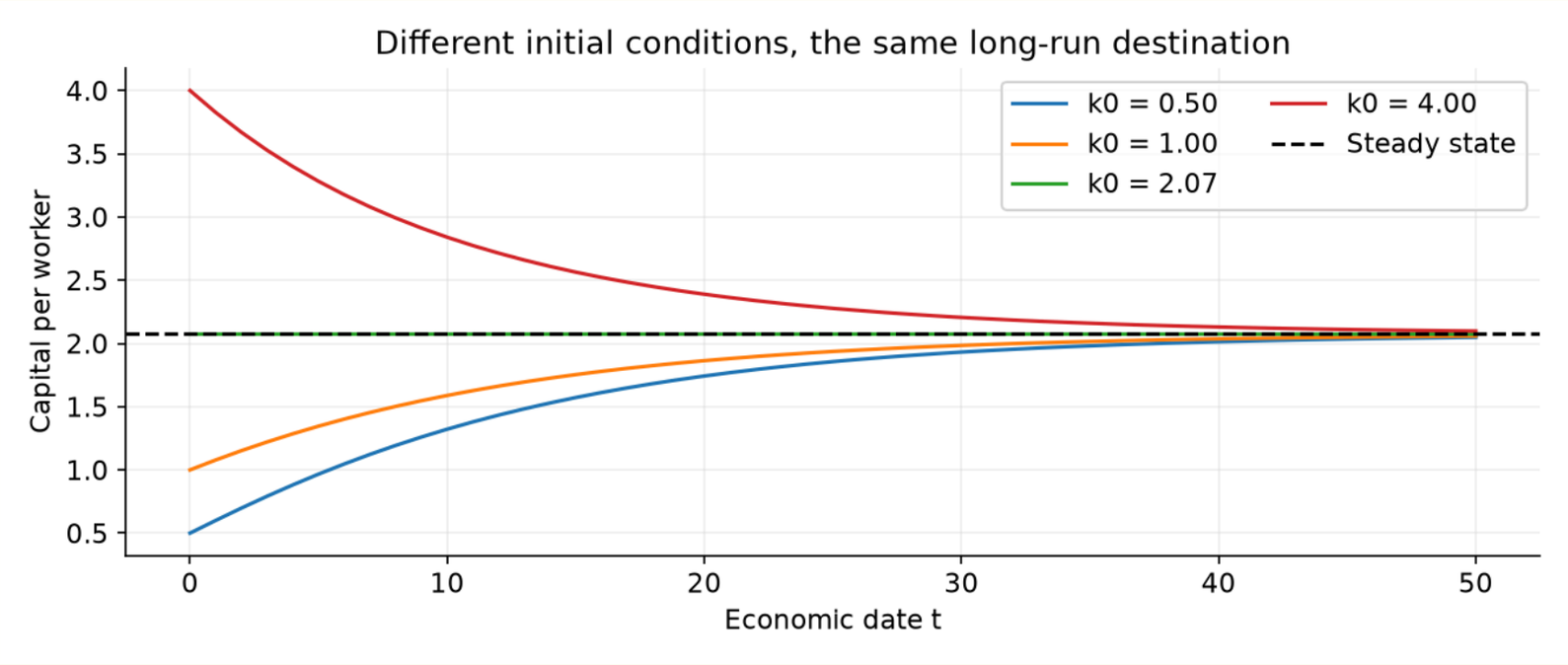
Section 3b · Read the supplied scenario storage

- A **list** stores starting values in order; `len` counts them.
- `enumerate` supplies a position and its value on each pass.
- Shape `(4, 51)`: one scenario per row, one date per column.
  - `[ind_scenario, :]` selects a whole row. Storage and plots are supplied.

```
initial_capitals = [0.5, 1.0, k_analytical, 4.0]
scenario_paths = np.zeros((len(initial_capitals), n_periods + 1))
for ind_scenario, k_start in enumerate(initial_capitals):
    scenario_paths[ind_scenario, :] = fun_simulate(k_start, par, n_periods)
```

Section 3b · Compare starting points

- Every row starts at its own initial condition, with the same calibration.
- The analytical steady-state row remains constant. Other paths approach it.
- Figures are supplied; simulation precedes plotting.



Our route · Restart at 4a

1 · Define and calibrate the economy

2 · Analytical steady state and diagrams

3 · Simulate transition paths

- 3a · Trace a fixed-horizon path
- 3b · Reuse the simulation

4a · Iterate until convergence

7 · Simulate productivity shocks

Today: complete 4a, then jump to 7. Study the complete examples in 4b–6 at home.

Python · Conditions and repetition

- Comparisons produce **`True` or `False`**: `==`, `!=`, `<`, `<=`, `>` and `>=`.
- `and` requires both conditions; `or` needs at least one; `not` reverses one.
- `if` runs an indented block when its condition is true.
  - `=` assigns; `==` compares equality.

```
amount = 2.0
print(amount == 2.0, amount <= 1.0)    # True False
if amount > 1.0 and amount < 3.0:
    print("Inside the interval")
```

Python · The while loop

- while checks a condition **before each pass**.
- Update the values used by the condition so the loop can stop.
- `counter += 1` adds one; a second condition limits the number of passes.

```
amount = 8.0
counter = 0
while amount > 1.0 and counter < 10:
    amount = amount / 2
    counter += 1
print(amount, counter)    # 1.0 3
```

Section 4a · Stop at the requested accuracy

- Section 3a chose **50 updates** and inspected the resulting path.
  - Now repeat until the latest capital change is below a tolerance.
- `abs(k_next - k_now)` measures the latest change.
  - `.append(k_next)` adds the accepted value to the history list.
- Start above tolerance; count updates against a safety budget.

```
tol = 1e-10
max_iter = 10000
k_now = k0
forward_history = [k_now]
change = 1.0
ind_iteration = 0
```

Section 4a · Plan the stopping loop

- Continue while **accuracy is unmet and budget remains**.
- On each pass, calculate, measure, store and advance.
  - Update the counter too; both conditions must permit another pass.

```
while ...:
    # Calculate the next capital value.
    # Measure its change from today's capital.
    # Append the new value to the history.
    # Make it current and count the update.
```

S04A-WHILE · Student task

1. **Where:** `main_solow_restart.py` (or full starter `main_solow.py`), section 4a, **S04A-WHILE**.
2. **Write:** The `while` condition and body: update capital, measure change, store the result, advance `k_now` and count the update.
3. **Given:** Initialisation, `fun_capital_update`, failure check and reporting.
4. **Run/check:** With `tol=1e-10`, reach 242 updates and 243 stored states. With budget 1, the supplied check must report failure.

Solution

```
while change > tol and ind_iteration < max_iter:
    k_next = fun_capital_update(k_now, par)
    change = abs(k_next - k_now)
    forward_history.append(k_next)
    k_now = k_next
    ind_iteration += 1
```

Section 4a · Check that stopping means success

- A loop can use its budget **before reaching accuracy**.
- The supplied guard refuses that result instead of reporting convergence.
  - `np.isfinite` checks for a finite number; `not` reverses the result.
  - `or` rejects either failure; `raise RuntimeError` stops with a message.

```
if not np.isfinite(change) or change > tol:
    raise RuntimeError("Forward iteration exhausted its update budget.")
```

Section 4a · Inspect convergence

- **242 updates** meet `tol=1e-10`; the actual capital error is about 1.11×10^{-9}.
  - Last change compares neighbours; analytical error compares with k^*.
- `len` counts stored states; `np.array` converts the list; `np.abs` acts entrywise.
- Inspect the supplied convergence figure: does the error shrink?
  - A budget of 1 must report failure; restore it before continuing.

```
n_updates = len(forward_history) - 1
forward_history = np.array(forward_history)
forward_error = np.abs(forward_history - k_analytical)
print(n_updates, forward_error[-1])    # 242, approximately 1.11e-9
```

Our route · Continue with 7

1 · Define and calibrate the economy

2 · Analytical steady state and diagrams

3 · Simulate transition paths

- 3a · Trace a fixed-horizon path
- 3b · Reuse the simulation

4a · Iterate until convergence

7 · Simulate productivity shocks

Restart: jump directly from 4a to 7. Sections 4b–6 remain home study.

Section 7 · A productivity-shock process

- Productivity now fluctuates: use **independent positive draws** by date.
- $\overline{A}$ is mean productivity; $\sigma \geq 0$ controls shock size.
  - ε_t are independent standard-normal draws: mean 0, variance 1.
- The correction fixes $E[A_t] = \overline{A}$; a sample mean can still differ.
  - At $\sigma = 0$, productivity is constant at $\overline{A}$.

$$A_t = \overline{A}\exp(\sigma\varepsilon_t - \sigma^2/2), \qquad \varepsilon_t \overset{\text{iid}}{\sim} N(0,1)$$

Section 7 · From productivity to the capital path

- Today's capital is **inherited**; productivity changes current output.
  - Investment changes next-period capital through the familiar update.
- Keep saving, depreciation, population growth and α fixed.
- Use **150 updates and 151 observations** for each scenario.
  - These are productivity-level shocks, with no technological trend.

$$y_t = A_t k_t^{\alpha}, \qquad c_t = (1 - s) y_t$$

$$k_{t+1} = \frac{s A_t k_t^{\alpha} + (1 - \delta) k_t}{1 + n}$$

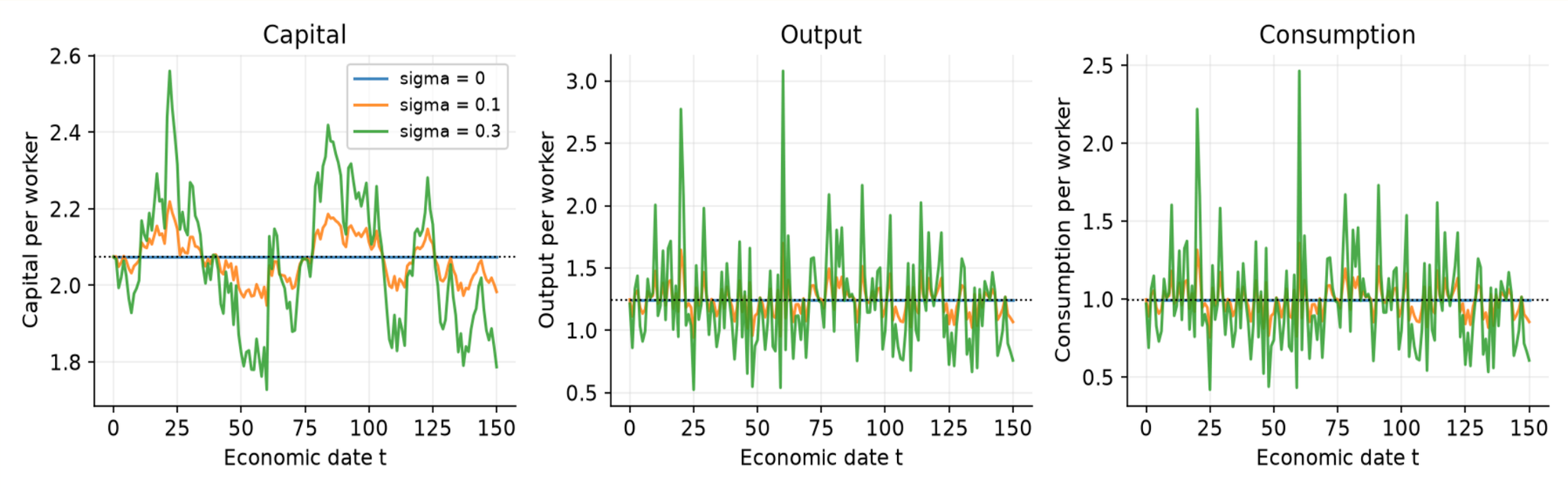
Python · Tools for the experiment

- A **seed** makes a random sequence reproducible; generator setup is supplied.
  - Recreate the generator with the same seed to repeat the sequence.
- Explore `rng.standard_normal` with **`help`**: how do you request an array?
- `np.exp(...)` applies the exponential to every array entry.
- Preserve the baseline parameters; recall dictionary **`.copy()`**.

```
1 seed = 20261019
2 rng = np.random.default_rng(seed)    # Supplied in section 7
3 help(rng.standard_normal)
```

Section 7 · The experiment you will build

- Compare $\sigma = 0, 0.1, 0.3$ using **one shared array of normal draws**.
- Start from the deterministic steady state: **150 updates, 151 observations**.
- Simulate/store in **7**; run supplied plotting **7b** for the panels.



S07-CAPSTONE · Your experiment

1. **Where:** `main_solow_restart.py` (or full starter `main_solow.py`), section **7**.
2. **Build:** `fun_simulate_productivity(k_start, par, productivity_path)`; return capital for every observation. Draw shared normals once.
   - Inside `for sigma in shock_sizes:`, construct productivity, capital, output and consumption. Supplied storage saves them in `shock_results`.
3. **Given:** Equations, settings, update helper, sigma-loop header and storage.
   - **7b is supplied plotting:** run as provided, without studying or editing it.
4. **Check:** The supplied constant-productivity comparison with `fun_simulate` must pass. Then run **7b** and inspect the three panels.

Solution

```python
def fun_simulate_productivity(k_start, par, productivity_path):
    n_updates = len(productivity_path) - 1
    capital_path = np.zeros(n_updates + 1)
    capital_path[0] = k_start
    par_period = par.copy()
    for ind_t in range(n_updates):
        par_period["productivity"] = productivity_path[ind_t]
        capital_path[ind_t + 1] = fun_capital_update(
            capital_path[ind_t], par_period
        )
    return capital_path
```

Solution

```
normal_draws = rng.standard_normal(shock_periods + 1)
for sigma in shock_sizes:
    productivity_path = productivity * np.exp(
        sigma * normal_draws - sigma**2 / 2
    )
    shock_capital = fun_simulate_productivity(
        k_analytical, par, productivity_path
    )
    shock_output = productivity_path * shock_capital**alpha
    shock_consumption = (1 - s) * shock_output
```

Simple checks save you a lot of time!

- With constant productivity, both simulators must give the **same capital path**.
  - Use the same initial capital, parameters and horizon; no output means a pass.
- Clean restart: run **0–3b, completed 4a, then completed 7** in the restart file.
- Then run supplied **7b**: it only plots stored `shock_results`.

```
constant_productivity = np.full(n_periods + 1, productivity)
constant_capital = fun_simulate_productivity(
    k0, par, constant_productivity
)
np.testing.assert_allclose(
    constant_capital, fun_simulate(k0, par, n_periods),
    rtol=0, atol=1e-12
)
```

Transfer the operations

- An input changes after a total was stored. Which assignment must run again?
- A loop receives 10, 20 and 30. Are these positions in a three-entry array?
- A function displays the right number but returns `None`. What is missing?
- A scenario changes the baseline dictionary. Where should `.copy()` be used?

End

Optional extra · Watch the solver

- **Time left after section 7?** Solve the deterministic steady state again.
  - Use bisection and Newton from lecture: $r(k) = g(k) - k = 0$.
- Run `extensions/root_finding.py`; write your loops at its bottom.
  - Calibration, residual, derivative and plotting tools are **supplied**.
- Pass each **evaluated guess**, its residual, its number and the same axes.
  - Left: where the equation is evaluated. Right: the sequence of guesses.
- Pause for **0.5 seconds** per evaluation; set `pause_seconds = 0` for speed.

```
axes = fun_start_plot("Bisection")        # Once, before your loop
# Inside the loop, after value = fun_residual(guess):
fun_plot_iteration(guess, value, iteration, axes, pause_seconds)
```

Optional extra · Write bisection

- **Where:** optional workspace in `extensions/root_finding.py`, after setup.
- Write the loop: midpoint → residual → plot → stop or retain one half.
  - Start with `capital_bounds`; preserve **opposite residual signs**.
  - Use `range(1, max_iter + 1)` and stop at `abs(value) <= tol`.
- Call `fun_plot_iteration(...)` **before** the stopping test.
  - The accepted candidate must appear in the figure too.
- **Check:** capital ≈ 2.07456375; `abs(value) <= 1e-10`.
  - Compare with `k_analytical`; stop with an error if the budget runs out.

Solution

```
lower, upper = capital_bounds
assert fun_residual(lower) * fun_residual(upper) < 0
axes = fun_start_plot("Bisection")
for iteration in range(1, max_iter + 1):
    guess = lower + (upper - lower) / 2
    value = fun_residual(guess)
    fun_plot_iteration(guess, value, iteration, axes, pause_seconds)
    if abs(value) <= tol:
        break
    if fun_residual(lower) * value > 0:
        lower = guess
    else:
        upper = guess
if abs(value) > tol:
    raise RuntimeError("Bisection exhausted its evaluation budget.")
print(f"Bisection: k={guess:.8f}, residual={value:.2e}")
```

Optional extra · Write Newton

- **Where:** a new section below your bisection loop; create a fresh figure.
- Start at `starting_guess = 1.0`; use the **supplied derivative**.
  - Evaluate → plot → test tolerance → take a Newton step.
- Keep the same tolerance and evaluation budget as bisection.
  - Invalid capital or a near-zero slope raises an error; inspect your start.
- **Check:** capital ≈ 2.07456375; `abs(value) <= 1e-10`.
  - Compare both sequences. Why do they visit different candidates?

$$k_{\text{new}} = k - \frac{r(k)}{r'(k)}$$

Solution

```
guess = starting_guess
axes = fun_start_plot("Newton")
for iteration in range(1, max_iter + 1):
    value = fun_residual(guess)
    fun_plot_iteration(guess, value, iteration, axes, pause_seconds)
    if abs(value) <= tol:
        break
    guess = guess - value / fun_derivative(guess)
if abs(value) > tol:
    raise RuntimeError("Newton exhausted its evaluation budget.")
print(f"Newton: k={guess:.8f}, residual={value:.2e}")
```