Python at a glance

QM1 bootcamp | Key rules and essential Python syntax

Key rules to remember

• **Count positions from zero.** `path[0]` is the first entry; an array of length 51 has indices 0-50. Slice stops are excluded: `path[:3]` selects indices 0, 1, 2.

• **Giving an object a second name does not copy it.** `other = par` refers to the same dictionary; changing `other["s"]` changes `par["s"]`. This also applies to shared lists and arrays. Use `par.copy()` or `path.copy()` for independent scalar parameters or numerical arrays. Reassignment affects only the assigned name: `a = 2; b = a; a = 3` leaves `b == 2`.

• **Calculations happen when you run them.** After changing `s`, rerun `investment = s * output` to update investment. Saving a file does not execute it.

01 · Values, names and calculations

• **Values and types:** `50`: integer; `0.2`: floating-point number; `"Capital"`: string (text); `True` / `False`: Boolean.

• **Assignment:** `s = 0.2` assigns the value `0.2` to `s`. The right-hand expression is evaluated before assignment.

• **Arithmetic:** `+ - * /`: addition, subtraction, multiplication, division. `k**alpha`: exponentiation. Parentheses group operations: `(a + b) / c`.

• **Comments:** # starts a comment; the rest of that line is not executed.

• **Display:** `print(s)` displays a value. An f-string inserts values into text: with `s = 0.2`, `print(f"Saving: {s:.0%}")` displays `Saving: 20%`.

02 · Libraries, calls and help

• **Imports and aliases:** `import numpy as np` imports NumPy as `np`; `import matplotlib.pyplot as plt` imports plotting tools as `plt`. Packages must already be installed.

• **Function call:** `np.sqrt(4.0)` calls the square-root function with input `4.0` and returns `2.0`.

• **Function inputs (arguments):** Inputs can be positional or named: `np.round(3.14159, 2)` and `np.round(3.14159, decimals=2)` both return `3.14`.

• **Dot syntax:** `grid_k.shape` accesses an attribute (dimensions); `grid_k.mean()` calls a method (the array's mean).

• **Documentation:** `help(np.linspace)` displays a function's inputs, defaults and examples.

03 · Arrays, grids and positions

• **Numerical array:** `np.array([1.0, 2.0])` constructs an array. Multiplying it by 2 gives an array with entries `2.0, 4.0`. Multiplying a Python list by 2 repeats the list.

• **Allocate storage:** `np.zeros(n_periods + 1)` constructs a one-dimensional array of `n_periods + 1` zeros. `np.zeros((3, 51))` constructs 3 rows × 51 columns.

• **Build grids:** `np.arange(4)` constructs `[0, 1, 2, 3]`; `np.linspace(0, 1, 5)` constructs five evenly spaced values from 0 to 1, including both endpoints.

• **Read / write:** `capital_path[0]` reads the first entry; `capital_path[0] = k0` assigns its value.

• **Slicing:** `capital_path[:3]` selects the first three entries; `capital_path[-1]` selects the last entry.

• **Size and shape:** `len(capital_path)` gives the number of entries in a one-dimensional array. `capital_path.shape` gives its dimensions, e.g. `(51,)`.

• **Calculate with whole arrays:** `productivity * capital_path**alpha` applies the calculation to each array entry.

• **Two dimensions:** `table[row, column]` selects one entry; `table[row, :]` selects a row. Both indices start at zero.

04 · Repeating and choosing

• **Blocks and indentation:** Function, loop and conditional headers end with `:`. Indent their bodies by four spaces; indentation determines which statements belong to the block.

• **For loop:** Repeats a block for each item. `range(n_periods)` produces integers from 0 to `n_periods - 1`.

```
for ind_t in range(n_periods):
    print(capital_path[ind_t])
```

• **Position and value:** `for ind_start, k_start in enumerate(starting_points):` iterates over each position `ind_start` and its value `k_start`.

• **Conditions:** `== != < <= > >=` compare values. For Boolean conditions, `and` requires both; `or` requires at least one; `not` reverses the truth value. `=` assigns; == compares.

• **Branches:** `if change < tol:` executes its block when the condition is true. `elif` tests an alternative condition; `else` handles the remaining case.

• **While loop:** `while change > tol and ind_iteration < max_iter:` repeats while both conditions hold. Update the tested values inside the loop; check the stopping condition afterwards.

05 · Functions and grouped inputs

• **Define a function:** `def` defines a function with inputs, allowing the same calculation to be reused in different parts of the code. `return` returns its result.

```
def fun_output(k, productivity, alpha):
    return productivity * k**alpha
```

• **Use the returned value:** `y = fun_output(2.0, 1.0, 0.3)` assigns the result to `y`. Function input names are local to the call. `print` displays a value; `return` passes a result to the caller.

• **Reuse another function:** `k_next = fun_capital_update(k_now, par)` calls the update function. Functions can call other functions.

• **Lists:** `starting_points = [0.5, 1.0, 3.0]` creates an ordered list. `history.append(k_next)` adds an entry at the end.

• **Dictionaries:** `par = {"s": 0.2, "alpha": 0.3}` stores key-value pairs. `par["s"]` retrieves a value; `par["s"] = 0.3` changes it.

• **Sharing and copying:** `other = par` shares the dictionary; `other = par.copy()` copies it. This copy is sufficient for our scalar parameters.

• **Several results:** `fig, ax = plt.subplots()` unpacks two returned objects into two names, in order. A tuple such as `(0.01, 5.0)` groups values in order.

06 · Plotting results

• **Create a plot:** `fig, ax = plt.subplots()` returns a figure and an axes object (plotting area).

• **Add a line:** `ax.plot(dates, capital_path, label="Capital")` plots vertical coordinates against horizontal coordinates: `plot(x, y)`. The arrays must have matching lengths.

• **Label and show:** `ax.set(xlabel="Date", ylabel="Capital")` sets axis labels; `ax.legend()` displays line labels; `plt.show()` displays the figure.

• **Several panels:** `fig, axes = plt.subplots(1, 2)` creates two side-by-side panels; `axes[0].plot(x, y)` plots on the first.

07 · Reproducible shocks

• **Random generator:** `rng = np.random.default_rng(seed)` creates a seeded generator. Recreating it with the same seed reproduces the sequence; further calls advance the existing generator.

• **Normal draws:** `rng.standard_normal(n_periods + 1)` returns `n_periods + 1` draws from the standard normal distribution (mean 0, standard deviation 1).

• **Exponential:** `np.exp(normal_draws)` computes the exponential of each entry. Use the productivity formula on the model sheet.

• **Controlled comparisons:** Reuse the same `normal_draws` across shock sizes; copy `par` before changing productivity within a scenario.

08 · Running, inspecting and checking

• **Execution order:** `# %%` marks a Positron code section. Run sections in order; rerun dependent calculations after changes.

• **Clean state:** Restart the Python Console and rerun from the top to check reproducibility. Clearing Console text does not clear variables.

• **Inspect a value:** `print(capital_path[:3])` displays the first three entries; `print(capital_path.shape)` displays the array dimensions.

• **Numerical comparison:** `abs(k_next - k_now) < tol` tests whether the absolute difference is below a tolerance. Use tolerances for floating-point results because of rounding error.

• **Check agreement:** For matching shapes, `np.allclose(path_a, path_b)` checks corresponding values within numerical tolerances. `assert condition` raises an error if the condition is false.

• **Read an error:** Read the final error message and locate the failing line. `NameError`: undefined name; `IndexError`: invalid index; `TypeError`: incompatible type.

• **Debug a loop:** Set a breakpoint to pause execution; step through individual statements. Inspect the index, inputs and stored output over two iterations.