Explaining some `.assign()` idioms

Ian D. Gow

26 October 2026

Users of pandas will know about the `.assign()` method of the DataFrame. But some code using it can look a little mysterious, particularly when `lambda`, `**`, and dictionary comprehensions appear together. In this note, I work through some examples to explain what these pieces do. I ran the examples with pandas 3.1.0.

1 Assigning a column

I borrow the toy data frame below from the pandas documentation.

```
import pandas as pd

df = pd.DataFrame(
    {"temp_c": [17.0, 25.0]},
    index=["Portland", "Berkeley"],
)
df
```

```
          temp_c
Portland    17.0
Berkeley    25.0
```

The `.assign()` method returns a new DataFrame containing the existing columns and any columns we supply. Supplying an existing column name replaces that column in the returned DataFrame. Here I add a `state` column:

```
with_state = df.assign(state=["OR", "CA"])
with_state
```

```
          temp_c state
Portland    17.0    OR
Berkeley    25.0    CA
```

The original DataFrame is unchanged:

```
df
```

```
          temp_c
Portland    17.0
Berkeley    25.0
```

We can make the name `df` refer to the returned DataFrame:

```
df = df.assign(state=["OR", "CA"])
```

This replaces what the name `df` refers to; it does not modify the old object. If another name referred to that old DataFrame, it would still see the original data. In contrast, the more traditional assignment below changes the DataFrame that `df` refers to:

```
df["state"] = ["OR", "CA"]
```

Pandas 3 uses Copy-on-Write, so a new DataFrame can initially share data with the original while behaving as a separate copy. Changing one does not change the other. This does not make DataFrames immutable: direct assignment such as `df["state"] = ...` still changes the object, and two names referring to that same object still see the same changes.

2 Why the lambda?

The signature `.assign(**kwargs)` tells us that column names are supplied as keyword arguments. In the example above, the keyword is `state` and its value is a list. Pandas assigns that list to the new column.

The value can also be a **callable**, such as an anonymous function created using `lambda`:

```
df.assign(temp_f=lambda d: d["temp_c"] * 9 / 5 + 32)
```

```
          temp_c state  temp_f
Portland    17.0    OR    62.6
Berkeley    25.0    CA    77.0
```

Here `.assign()` calls the function with the DataFrame as its argument. The function returns the values for `temp_f`. The name `d` has no special meaning; I could have used `x` or another name.

For this simple example, we could calculate those values directly:

```
df.assign(temp_f=df["temp_c"] * 9 / 5 + 32)
```

```
          temp_c state  temp_f
Portland    17.0    OR    62.6
Berkeley    25.0    CA    77.0
```

The distinction matters in a pipeline. A callable receives the DataFrame at that point in the pipeline, including columns created in earlier steps. It can even use a column created earlier in the *same* `.assign()` call:

```
temperatures = (
    pd.DataFrame({"temp_c": [17.0, 25.0]})
    .assign(
        temp_f=lambda d: d["temp_c"] * 9 / 5 + 32,
        temp_k=lambda d: (d["temp_f"] - 32) * 5 / 9 + 273.15,
    )
)
temperatures
```

```
   temp_c  temp_f  temp_k
0    17.0    62.6  290.15
1    25.0    77.0  298.15
```

The keyword arguments are processed in order, so `temp_f` exists when the second function runs. Using the name `df` inside these functions would instead refer to whatever object that name happened to identify outside the pipeline.

The documentation also says that a callable should not modify the DataFrame it receives. For example, `lambda d: d.pop("temp_c")` would remove `temp_c` while calculating the new column. The point of the callable is to calculate values, not to change the data structure as a side effect.

3 Cleaning several columns

Now consider some messier toy data. The temperatures include a `C` suffix, the areas include units, and the populations include commas.

```
raw_data = {
    "city": ["Portland", "Berkeley"],
    "temp_c": ["17C", "25C"],
    "area_land": ["345.73 sq km", "27.02 sq km"],
    "area_water": ["29.82 sq km", "18.71 sq km"],
    "area_total": ["375.55 sq km", "45.73 sq km"],
    "pop": ["652,503", "124,321"],
}

pd.DataFrame(raw_data)
```

```
       city temp_c     area_land   area_water    area_total      pop
0  Portland    17C  345.73 sq km  29.82 sq km  375.55 sq km  652,503
1  Berkeley    25C   27.02 sq km  18.71 sq km   45.73 sq km  124,321
```

We can clean each column separately:

```
clean_explicit = (
    pd.DataFrame(raw_data)
    .assign(
        temp_c=lambda d: pd.to_numeric(
            d["temp_c"].str.removesuffix("C"), errors="raise"
        ),
        area_land=lambda d: pd.to_numeric(
            d["area_land"].str.removesuffix(" sq km"), errors="raise"
        ),
        area_water=lambda d: pd.to_numeric(
            d["area_water"].str.removesuffix(" sq km"), errors="raise"
        ),
        area_total=lambda d: pd.to_numeric(
            d["area_total"].str.removesuffix(" sq km"), errors="raise"
        ),
        pop=lambda d: pd.to_numeric(
            d["pop"].str.replace(",", "", regex=False), errors="raise"
        ),
    )
)
clean_explicit
```

```
       city  temp_c  area_land  area_water  area_total     pop
0  Portland      17     345.73       29.82      375.55  652503
1  Berkeley      25      27.02       18.71       45.73  124321
```

Ew! That's some ugly code. One improvement is to put the repeated operations into functions. The `errors="raise"` arguments make unexpected non-numeric values cause an error rather than quietly become missing values.

```
def parse_area(s: pd.Series) -> pd.Series:
    return pd.to_numeric(
        s.str.removesuffix(" sq km"), errors="raise"
    )


def parse_pop(s: pd.Series) -> pd.Series:
    return pd.to_numeric(
        s.str.replace(",", "", regex=False), errors="raise"
    )
```

We can now write:

```
clean_helpers = (
    pd.DataFrame(raw_data)
    .assign(
        temp_c=lambda d: pd.to_numeric(
            d["temp_c"].str.removesuffix("C"), errors="raise"
        ),
        area_land=lambda d: parse_area(d["area_land"]),
        area_water=lambda d: parse_area(d["area_water"]),
        area_total=lambda d: parse_area(d["area_total"]),
        pop=lambda d: parse_pop(d["pop"]),
    )
)
```

This is better, but we are still repeating ourselves for the three area columns.

4 Unpacking a dictionary of functions

The three area arguments above can be generated using a dictionary comprehension:

```
area_cols = ["area_land", "area_water", "area_total"]

clean_dict = (
    pd.DataFrame(raw_data)
    .assign(
        temp_c=lambda d: pd.to_numeric(
            d["temp_c"].str.removesuffix("C"), errors="raise"
        ),
        pop=lambda d: parse_pop(d["pop"]),
        **{c: (lambda d, c=c: parse_area(d[c])) for c in area_cols},
    )
)
```

There are two pieces of Python syntax to unpack here. First, the comprehension creates a dictionary whose keys are the area-column names and whose values are functions. The `**` then unpacks that dictionary into keyword arguments. For example, `**{"area_land": some_function}` supplies the keyword argument `area_land=some_function`.

Second, `c=c` gives each function its own default value for the column name. That value is set when the function is created. Without it, the functions would look up `c` when called, by which time the loop would have reached `"area_total"`. We would therefore use the total-area values for all three columns. This is a standard Python closure issue, rather than something specific to pandas.

5 A pandas 3 alternative: `pd.col()`

Pandas 3 introduced `pd.col()`, which lets us build expressions referring to columns by name. For example, the temperature calculation can be written as:

```
df.assign(temp_f=pd.col("temp_c") * 9 / 5 + 32)
```

```
          temp_c state  temp_f
Portland    17.0    OR    62.6
Berkeley    25.0    CA    77.0
```

The expression is evaluated against the DataFrame supplied by `.assign()`. It therefore serves the same purpose as `lambda d: d["temp_c"] * 9 / 5 + 32`.

Expressions can also use Series methods. Here `.pipe(parse_area)` passes the selected Series to `parse_area()` when the expression is evaluated:

```
clean_expr = (
    pd.DataFrame(raw_data)
    .assign(
        temp_c=pd.col("temp_c").str.removesuffix("C").pipe(pd.to_numeric),
        pop=pd.col("pop").pipe(parse_pop),
        **{c: pd.col(c).pipe(parse_area) for c in area_cols},
    )
)
clean_expr
```

```
       city  temp_c  area_land  area_water  area_total     pop
0  Portland      17     345.73       29.82      375.55  652503
1  Berkeley      25      27.02       18.71       45.73  124321
```

Each expression records its column name when it is constructed, so there is no `c=c` to explain in this version. I find this easier to read, though understanding the callable version is still useful when reading existing pandas code.

6 Something like `across()`

R users might wonder why we are working so hard. Using `dplyr` and `readr`, the cleaning can be written as follows:

```
library(dplyr)
library(readr)

raw_data_r <- list(
  city = c("Portland", "Berkeley"),
  temp_c = c("17C", "25C"),
  area_land = c("345.73 sq km", "27.02 sq km"),
  area_water = c("29.82 sq km", "18.71 sq km"),
  area_total = c("375.55 sq km", "45.73 sq km"),
  pop = c("652,503", "124,321")
)

clean_r <-
  tibble::as_tibble(raw_data_r) |>
  mutate(across(-city, parse_number))

clean_r
```

```
# A tibble: 2 × 6
  city     temp_c area_land area_water area_total    pop
  <chr>     <dbl>     <dbl>      <dbl>      <dbl>  <dbl>
1 Portland     17    345.73      29.82     375.55 652503
2 Berkeley     25     27.02      18.71      45.73 124321
```

The `across()` call applies `parse_number()` to every column except `city`. The parser handles the units and commas in these values.

For this particular data set, we can write a small Python helper that removes everything except digits, decimal points, and minus signs before converting to numbers:

```
def parse_number(s: pd.Series) -> pd.Series:
    return (
        s.astype("string")
        .str.replace(r"[^\d.\-]+", "", regex=True)
        .pipe(pd.to_numeric, errors="raise")
    )
```

This is not a general replacement for `readr::parse_number()`. For example, it would turn `"12 to 15"` into `1215` and would not correctly interpret scientific notation. It is suitable here because we know the limited formats used in our toy data.

To select columns from the current DataFrame, I use `.pipe()`. It passes that DataFrame to a function; here, the function calls `.assign()` with a generated argument for every column except `city`.

```
clean_all = (
    pd.DataFrame(raw_data)
    .pipe(
        lambda d: d.assign(
            **{
                c: pd.col(c).pipe(parse_number)
                for c in d.columns if c != "city"
            }
        )
    )
    .assign(
        temp_f=pd.col("temp_c") * 9 / 5 + 32,
        temp_k=(pd.col("temp_f") - 32) * 5 / 9 + 273.15,
    )
)
clean_all[["city", "temp_c", "temp_f", "temp_k"]]
```

```
       city  temp_c  temp_f  temp_k
0  Portland      17    62.6  290.15
1  Berkeley      25    77.0  298.15
```

The output shows the temperature columns; the other cleaned columns remain in `clean_all`. The last `.assign()` illustrates the point about evaluation order again: the expression for `temp_k` uses the newly created `temp_f`.

7 Some choices

For one or two columns, spelling out the arguments to `.assign()` is often clearest. When several columns need the same treatment, a dictionary comprehension can avoid repetition. Callables let those calculations use the DataFrame at the current step in a pipeline, and pandas 3's `pd.col()` expressions can make some of them easier to read.

The goal is readable code. A pipeline is useful when it makes the sequence of transformations easier to follow; there is no need to put an entire analysis into one expression.