Load data from dirctory:

```
from pathlib import Path
import pandas as pd

DATA_DIR = Path('data')
print(f"Path to data directory: {DATA_DIR}")
```

```
Path to data directory: data
```

Generic BLS flat-file loader to read in the data

```
def read_bls_tsv(path: str | Path):
    df = pd.read_csv(path, sep="\t", dtype={"series_id": "string", "period": "string"})
    df.columns = [c.strip() for c in df.columns]

    df["series_id"] = df["series_id"].astype("string").str.strip()
    df["period"] = df["period"].astype("string").str.strip()
    df["value"] = pd.to_numeric(df["value"], errors="coerce")

    # Keep monthly only (drop M13 annual)
    df = df[df["period"].str.match(r"^M(0[1-9]|1[0-2])$")].copy()
    df["month"] = df["period"].str[1:3].astype(int)
    df["date"] = pd.to_datetime(df["year"].astype(str) + "-" + df["month"].astype(str).str.zfill(2) + "-01")

    # Keep only the columns we need for plotting/merging
    keep_cols = ["series_id", "date", "value"]
    if "footnote_codes" in df.columns:
        df["footnote_codes"] = df["footnote_codes"].astype("string").str.strip()
        keep_cols.append("footnote_codes")

    return df[keep_cols]
```

Small helper for metadata("lookup") files

```
def read_meta_tsv(path: str | Path):
    df = pd.read_csv(path, sep="\t", dtype="string")
    df.columns = [c.strip() for c in df.columns]
    for c in df.columns:
        df[c] = df[c].astype("string").str.strip()
    return df
```

Filter-Load a big BLS data file by series_id (loads in chunks)

```
def read_bls_tsv_filtered(path: str | Path, keep_series_ids: set[str], chunksize: int = 2_000_000):
    chunks = pd.read_csv(path, sep="\t", dtype={"series_id": "string", "period": "string"}, chunksize=chunksize)
    out = []
    for c in chunks:
        c.columns = [x.strip() for x in c.columns]
        c["series_id"] = c["series_id"].astype("string").str.strip()
        c["period"] = c["period"].astype("string").str.strip()
        c = c[c["series_id"].isin(keep_series_ids)]
        if len(c):
            # Reuse the same logic to build date + value
            c["value"] = pd.to_numeric(c["value"], errors="coerce")
            c = c[c["period"].str.match(r"^M(0[1-9]|1[0-2])$")].copy()
            c["month"] = c["period"].str[1:3].astype(int)
            c["date"] = pd.to_datetime(c["year"].astype(str) + "-" + c["month"].astype(str).str.zfill(2) + "-01")
            keep_cols = ["series_id", "date", "value"]
            if "footnote_codes" in c.columns:
                c["footnote_codes"] = c["footnote_codes"].astype("string").str.strip()
                keep_cols.append("footnote_codes")
            out.append(c[keep_cols])

    return pd.concat(out, ignore_index=True) if out else pd.DataFrame(columns=["series_id", "date", "value"])
```

Loading all files (from Bureau of Labor Statistics)

```
# ---- JOLTS metadata (for mapping + drill-down) ----
jt_series     = read_meta_tsv(DATA_DIR / "jt.series.txt")
jt_industry   = read_meta_tsv(DATA_DIR / "jt.industry.txt")
jt_ratelevel  = read_meta_tsv(DATA_DIR / "jt.ratelevel.txt")
jt_period     = read_meta_tsv(DATA_DIR / "jt.period.txt")
jt_dataelem   = read_meta_tsv(DATA_DIR / "jt.dataelement.txt")
```

```
# ---- JOLTS data (openings / hires / quits) ----
jt_openings = read_bls_tsv(DATA_DIR / "jt.data.2.JobOpenings.txt")
jt_hires    = read_bls_tsv(DATA_DIR / "jt.data.3.Hires.txt")
jt_quits    = read_bls_tsv(DATA_DIR / "jt.data.5.Quits.txt")
```

```
# ---- CES metadata (for series discovery & labels) ----
ce_series    = read_meta_tsv(DATA_DIR / "ce.series.txt")
ce_datatype  = read_meta_tsv(DATA_DIR / "ce.datatype.txt")
ce_industry  = read_meta_tsv(DATA_DIR / "ce.industry.txt")
```

```
# ---- CES data (employment + earnings by industry) ----
ce_fin_emp  = read_bls_tsv(DATA_DIR / "ce.data.55a.FinancialActivities.Employment.txt")
ce_fin_ae   = read_bls_tsv(DATA_DIR / "ce.data.55b.FinancialActivities.AllEmployeeHoursAndEarnings.txt")

ce_pbs_emp  = read_bls_tsv(DATA_DIR / "ce.data.60a.ProfessionalBusinessServices.Employment.txt")
ce_pbs_ae   = read_bls_tsv(DATA_DIR / "ce.data.60b.ProfessionalBusinessServices.AllEmployeeHoursAndEarnings.txt")
```

Load Unemployment Rate

```
LN_PATH = DATA_DIR / "ln.data.1.AllData.txt"
UR_ID = "LNS14000000"  # headline unemployment rate (U-3)

ln_ur_raw = read_bls_tsv_filtered(LN_PATH, {UR_ID})
ur = (
    ln_ur_raw
    .rename(columns={"value": "ur"})
    [["date", "ur"]]
    .sort_values("date")
    .reset_index(drop=True)
)
```

```
/tmp/ipython-input-1970496410.py:8: DtypeWarning:

Columns (4) have mixed types. Specify dtype option on import or set low_memory=False.

/tmp/ipython-input-1970496410.py:8: DtypeWarning:

Columns (4) have mixed types. Specify dtype option on import or set low_memory=False.

/tmp/ipython-input-1970496410.py:8: DtypeWarning:

Columns (4) have mixed types. Specify dtype option on import or set low_memory=False.

/tmp/ipython-input-1970496410.py:8: DtypeWarning:

Columns (4) have mixed types. Specify dtype option on import or set low_memory=False.
```

Bundle into Dictionaries

Group into variables for convenience

- `data["jt"]` for Job Openings and Labor Turnover Survey (JOLTS)
- `data["ce"]` for Current Employment Statistics (CES)
- `data["ur"]` for unemployment data

```
data = {
    "jt": {
        "series": jt_series,
        "industry": jt_industry,
        "ratelevel": jt_ratelevel,
        "period": jt_period,
        "dataelement": jt_dataelem,
        "openings": jt_openings,
        "hires": jt_hires,
        "quits": jt_quits,
    },
    "ce": {
        "series": ce_series,
        "datatype": ce_datatype,
        "industry": ce_industry,
        "fin_emp": ce_fin_emp,
        "fin_ae": ce_fin_ae,
        "pbs_emp": ce_pbs_emp,
        "pbs_ae": ce_pbs_ae,  # None unless you set pbs_keep
    },
    "ur": ur
}

print("Loaded:")
print("  JOLTS openings rows:", len(jt_openings))
print("  JOLTS hires rows:", len(jt_hires))
print("  JOLTS quits rows:", len(jt_quits))
print("  CES fin emp rows:", len(ce_fin_emp))
print("  CES pbs emp rows:", len(ce_pbs_emp))
print("  Unemployment rate rows:", len(ur))
print("  PBS AE loaded?:", ce_pbs_ae is not None)
```

```
Loaded:
  JOLTS openings rows: 106596
  JOLTS hires rows: 106596
  JOLTS quits rows: 106596
  CES fin emp rows: 132478
  CES pbs emp rows: 209238
  Unemployment rate rows: 936
  PBS AE loaded?: True
```

Data Vis 1: Unemployment vs Openings (Belridge Curve 2021-2025)

Helper functions for plotting and series selection

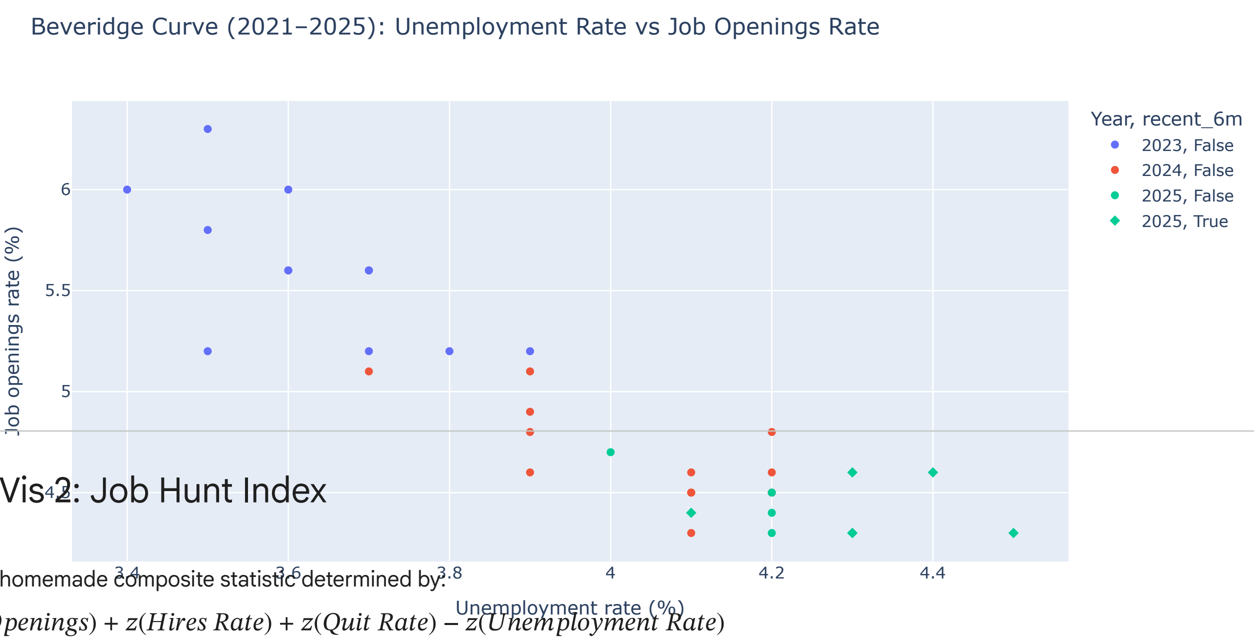
Use metadata to pick series, then use `series_ts()` to retrieve the actual time series.

```
import pandas as pd
import plotly.express as px

def series_ts(df: pd.DataFrame, series_id: str, value_name: str = "value") -> pd.DataFrame:
    """Return a single-series time series with columns: date, <value_name>."""
    out = df[df["series_id"] == series_id][["date", "value"]].copy()
    if out.empty:
        raise KeyError(f"series_id not found in dataframe: {series_id}")
    out = out.sort_values("date").rename(columns={"value": value_name}).reset_index(drop=True)
    return out

def latest_yoy_pct(s: pd.Series) -> float:
    """Latest YoY percent change (needs monthly frequency)."""
    yoy = (s / s.shift(12) - 1.0) * 100.0
    yoy = yoy.dropna()
    return float(yoy.iloc[-1]) if len(yoy) else float("nan")

def find_jolts_industry_code(jt_industry: pd.DataFrame, keyword: str) -> str:
    """Find the first JOLTS industry_code whose industry_text contains keyword."""
    m = jt_industry[jt_industry["industry_text"].str.contains(keyword, case=False, na=False)]
    if m.empty:
        raise KeyError(f"No JOLTS industry match for keyword: {keyword}")
    return m.iloc[0]["industry_code"]

def pick_jolts_series_id(
    jt_series: pd.DataFrame,
    dataelement_code: str,
    ratelevel_code: str,
    industry_code: str,
    state_code: str = "00",
    area_code: str = "00000",
    sizeclass_code: str = "00",
) -> str:
    """
    Pick a JOLTS series_id from jt_series using metadata codes.
    This is more robust than hardcoding series IDs.
    """
    df = jt_series.copy()

    # normalize whitespace just in case
    for c in ["dataelement_code","ratelevel_code","industry_code","state_code","area_code","sizeclass_code"]:
        if c in df.columns:
            df[c] = df[c].astype("string").str.strip()

    m = df[
        (df["dataelement_code"] == dataelement_code) &
        (df["ratelevel_code"] == ratelevel_code) &
        (df["industry_code"] == industry_code) &
        (df["state_code"] == state_code) &
        (df["area_code"] == area_code) &
        (df["sizeclass_code"] == sizeclass_code)
    ]

    if m.empty:
        raise KeyError(
            f"No JOLTS series found for dataelement={dataelement_code}, ratelevel={ratelevel_code}, industry={indust
        )

    # If multiple matches exist, just take the first.
    return m.iloc[0]["series_id"]
```

Beveridge curve: unemployment vs job openings rate

Industry openings rate over time (Business and Finance Sectors)

```
# --- Total nonfarm Job Openings Rate (JOR) ---
try:
    total_ind_code = "000000"
    total_jor_id = pick_jolts_series_id(
        jt_series,
        dataelement_code="JO",
        ratelevel_code="R",
        industry_code=total_ind_code
    )
except Exception:
    total_ind_code = find_jolts_industry_code(jt_industry, "Total")
    total_jor_id = pick_jolts_series_id(jt_series, "JO", "R", total_ind_code)

jor_total = series_ts(jt_openings, total_jor_id, value_name="jor")

# --- Window filter (2022–2025) ---
start = pd.Timestamp("2023-01-01")
end   = pd.Timestamp("2026-1-01")  # monthly data uses first-of-month dates

def in_window(df):
    return df[(df["date"] >= start) & (df["date"] <= end)].copy()

# --- Beveridge curve data: merge job openings rate with unemployment rate ---
bev = jor_total.merge(ur, on="date", how="inner").sort_values("date")
bev = in_window(bev)

# tag last 6 months (within the window) for highlighting
last_date = bev["date"].max()
bev["recent_6m"] = bev["date"] >= (last_date - pd.DateOffset(months=6))

fig_bev = px.scatter(
    bev,
    x="ur",
    y="jor"
    color=bev["date"].dt.year.astype(str),
    symbol="recent_6m",
    hover_data=["date"],
    title="Beveridge Curve (2021–2025): Unemployment Rate vs Job Openings Rate",
    labels={"ur": "Unemployment rate (%)", "jor": "Job openings rate (%)", "color": "Year"},
)
fig_bev.show()

# --- Industry drill-down for openings rate ---
fin_code  = find_jolts_industry_code(jt_industry, "Financial")
pbs_code  = find_jolts_industry_code(jt_industry, "Professional")

industry_list = [
    ("Financial activities", fin_code),
    ("Professional & business services", pbs_code),
]

drill = []
for name, code in industry_list:
    sid = pick_jolts_series_id(jt_series, dataelement_code="JO", ratelevel_code="R", industry_code=code)
    tmp = series_ts(jt_openings, sid, value_name="openings_rate")
    tmp["industry"] = name
    drill.append(tmp)

drill_df = pd.concat(drill, ignore_index=True)
drill_df = in_window(drill_df)



fig_drill = px.line(
    drill_df,
    x="date",
    y="openings_rate",
    color="industry",
    title="Job Openings Rate by Industry (2021–2025, selected)",
    labels={"openings_rate": "Job openings rate (%)"},
)
fig_drill.show()
```



Data Vis 2: Job Hunt Index

This is a homemade composite statistic determined by:

$z(Job\ Openings) + z(Hires\ Rate) + z(Quit\ Rate) - z(Unemployment\ Rate)$

where $z(x)$ is the z-score of column x

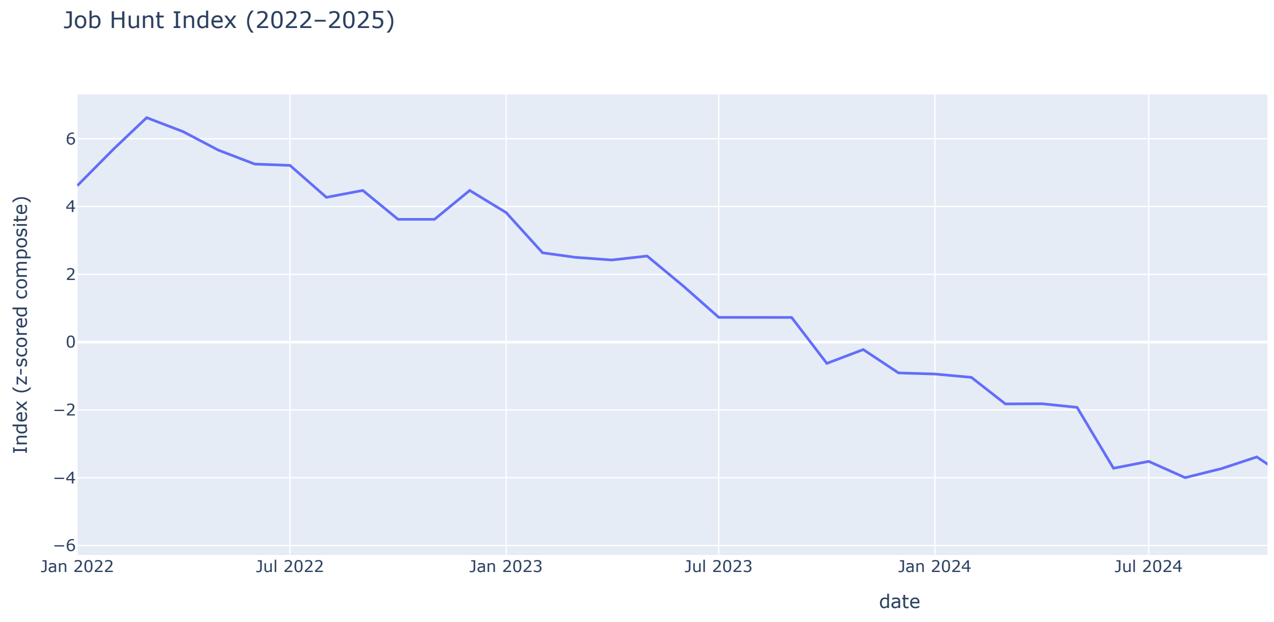
- If the index rises: conditions "feel" more favorable for job seekers
- If it falls: conditions "feel" tougher

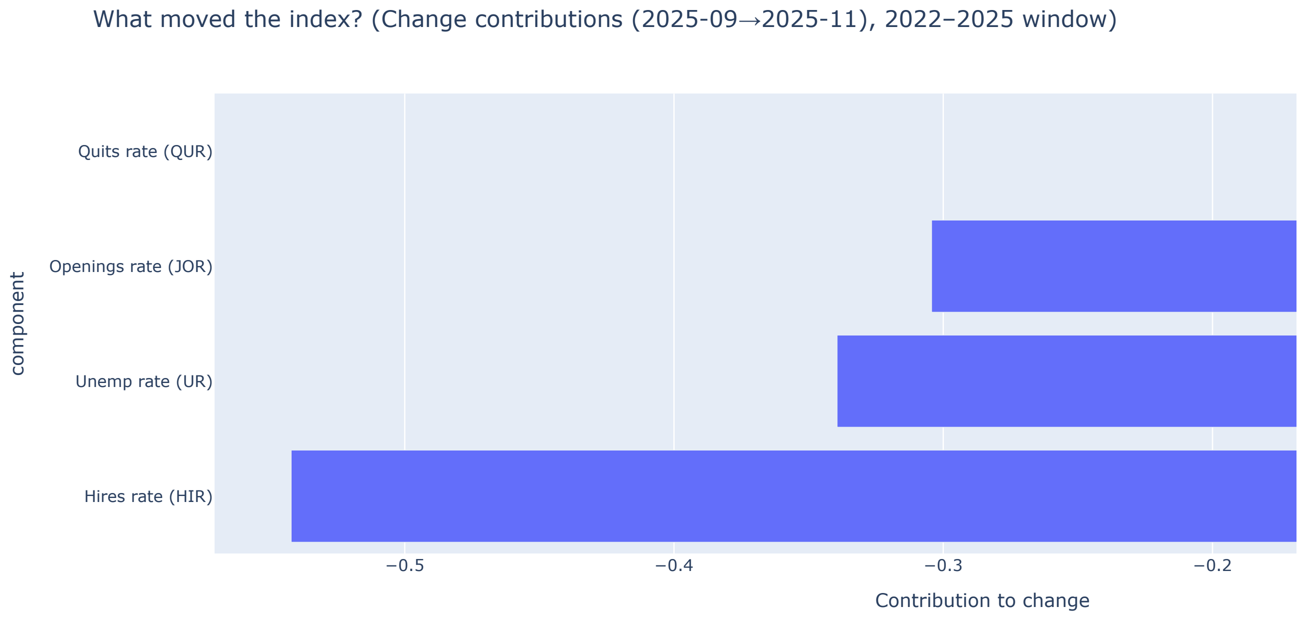
Job Openings Rate by Industry (2021–2025, selected)

```
# --- Pull total rates for openings, hires, quits ---
jor_id = pick_jolts_series_id(jt_series, dataelement_code="JO", ratelevel_code="R", industry_code=total_ind_code)
hir_id = pick_jolts_series_id(jt_series, dataelement_code="HI", ratelevel_code="R", industry_code=total_ind_code)
qur_id = pick_jolts_series_id(jt_series, dataelement_code="QU", ratelevel_code="R", industry_code=total_ind_code)

jor_ts = series_ts(jt_openings, jor_id, value_name="jor")
hir_ts = series_ts(jt_hires,    hir_id, value_name="hir")
qur_ts = series_ts(jt_quits,    qur_id, value_name="qur")

df = (
    jor_ts.merge(hir_ts, on="date", how="inner")
          .merge(qur_ts, on="date", how="inner")
          .merge(ur,     on="date", how="inner")
          .sort_values("date")
          .reset_index(drop=True)
)

# clean + drop missing components
for col in ["jor", "hir", "qur", "ur"]:
    df[col] = pd.to_numeric(df[col], errors="coerce")

df = (df.dropna(subset=["jor", "hir", "qur", "ur"])   # drops Oct 2025 UR missing, etc.
        .sort_values("date")
        .reset_index(drop=True))
# dropna(subset=...) is the standard way to remove rows with missing values in specific columns. :contentReference[o

# --- Window filter (2022–2025) ---
start = pd.Timestamp("2022-01-01")
end   = pd.Timestamp("2025-12-01")

df = df[(df["date"] >= start) & (df["date"] <= end)].copy()
df = df.sort_values("date").reset_index(drop=True)

def z(s: pd.Series) -> pd.Series:
    sd = s.std(ddof=0)
    if pd.isna(sd) or sd == 0:
        return pd.Series(0.0, index=s.index)
    return (s - s.mean()) / sd

# compute z-scores
df["z_jor"] = z(df["jor"])
df["z_hir"] = z(df["hir"])
df["z_qur"] = z(df["qur"])
```

```
df["z_ur"]  = z(df["ur"])

df["index"] = df["z_jor"] + df["z_hir"] + df["z_qur"] - df["z_ur"]

latest = df.iloc[-1]
prev   = df.iloc[-2]

index_level = float(latest["index"])
index_change = float(latest["index"] - prev["index"])

# detect whether this is truly month-over-month
gap = (latest["date"].year - prev["date"].year) * 12 + (latest["date"].month - prev["date"].month)
change_label = "MoM change" if gap == 1 else f"Change since previous available month ({prev['date']:%Y-%m}→{latest['

print(f"Job Hunt Index (latest): {index_level:.2f}")
print(f"{change_label}: {index_change:+.2f}")

# --- 2B) Sparkline ---
fig_idx = px.line(
    df,
    x="date",
    y="index",
    title="Job Hunt Index (2022–2025)",
    labels={"index": "Index (z-scored composite)"},
)
fig_idx.show()

# contributions using z-columns + label the window correctly
contrib = pd.Series({
    "Openings rate (JOR)": df["z_jor"].iloc[-1] - df["z_jor"].iloc[-2],
    "Hires rate (HIR)":    df["z_hir"].iloc[-1] - df["z_hir"].iloc[-2],
    "Quits rate (QUR)":    df["z_qur"].iloc[-1] - df["z_qur"].iloc[-2],
    "Unemp rate (UR)":    -(df["z_ur"].iloc[-1]  - df["z_ur"].iloc[-2]),  # minus because UR is subtracted in index
}).sort_values()

contrib_df = contrib.rename("contribution").reset_index().rename(columns={"index": "component"})

title_suffix = "MoM contributions" if gap == 1 else f"Change contributions ({prev['date']:%Y-%m}→{latest['date']:%Y-
fig_contrib = px.bar(
    contrib_df,
    x="contribution",
    y="component",
    orientation="h",
    title=f"What moved the index? ({title_suffix}, 2022–2025 window)",
    labels={"contribution": "Contribution to change"},
)
fig_contrib.show()
```

```
Job Hunt Index (latest): -5.59
Change since previous available month (2025-09→2025-11): -1.19
```





Data Vis 3: Where are the Jobs? (Business and Finance Sectors)

Focus is on

- Employment growth (are payrolls expanding?)
- Earnings growth (are wages rising?)

Select the right CES series IDs using metadata

```
def pick_ces_series_id_contains(ce_series, must_have, data_type_code, prefer_sa=True):
    s = ce_series.copy()
    s["series_title"] = s["series_title"].astype("string")
    s["data_type_code"] = s["data_type_code"].astype("string").str.strip()
    m = s[s["data_type_code"] == data_type_code]
    for token in must_have:
        m = m[m["series_title"].str.contains(token, case=False, na=False)]

    if m.empty:
        raise KeyError(f"No CES series match for tokens={must_have} and data_type_code='{data_type_code}'")

    if prefer_sa and "seasonal" in m.columns:
        sa = m[m["seasonal"].str.upper() == "S"]
        if not sa.empty:
            m = sa

    return m.iloc[0]["series_id"]
```

```
fin_emp_id  = pick_ces_series_id_contains(ce_series, ["financial activities", "all employees"], "01")
fin_ahe_id  = pick_ces_series_id_contains(ce_series, ["financial activities", "average hourly earnings"], "03")

pbs_emp_id  = pick_ces_series_id_contains(ce_series, ["professional and business services", "all employees"], "01")
pbs_ahe_id  = pick_ces_series_id_contains(ce_series, ["professional and business services", "average hourly earnings

# Pull time series from the sector files you loaded
fin_emp_ts  = series_ts(ce_fin_emp,  fin_emp_id,  "emp")
fin_ahe_ts  = series_ts(ce_fin_ae,   fin_ahe_id,  "ahe")

pbs_emp_ts  = series_ts(ce_pbs_emp,  pbs_emp_id,  "emp")
pbs_ahe_ts  = series_ts(ce_pbs_ae,   pbs_ahe_id,  "ahe")
```

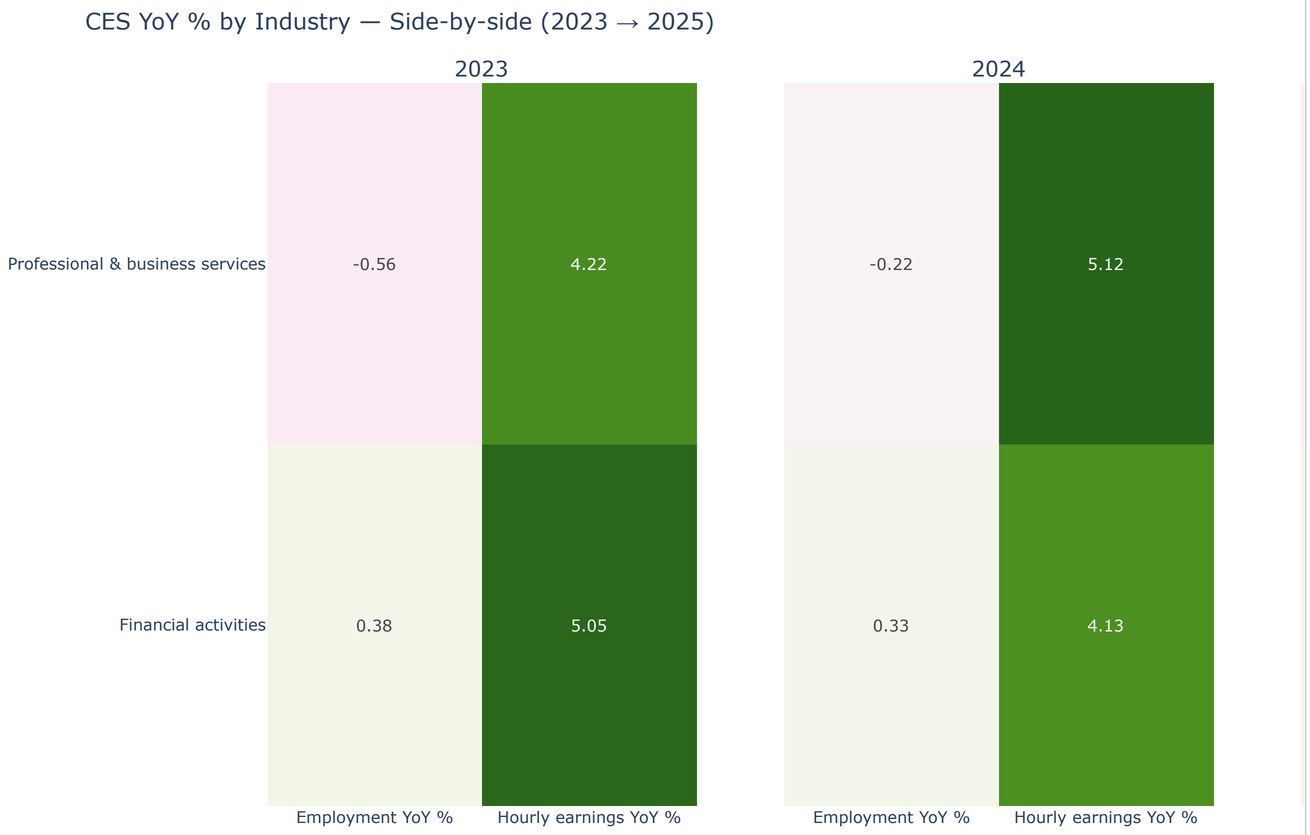
Create yearly “snapshots” (2023, 2024, 2025)

```
def year_snapshot_row(industry: str, emp_ts: pd.DataFrame, ahe_ts: pd.DataFrame, year: int, month: int | None = 12):
    e = emp_ts[["date", "emp"]].copy()
    w = ahe_ts[["date", "ahe"]].copy()
    both = e.merge(w, on="date", how="inner").sort_values("date")

    # try chosen month in that year (default Dec). If missing, fall back to latest month in that year.
    in_year = both[both["date"].dt.year == year]
    if month is not None:
        cand = in_year[in_year["date"].dt.month == month]
        snap_date = cand["date"].max() if not cand.empty else in_year["date"].max()
    else:
        snap_date = in_year["date"].max()

    if pd.isna(snap_date):
        raise ValueError(f"No overlapping emp/ahe data for {industry} in year {year}.")

    prev_date = (pd.Timestamp(snap_date) - pd.DateOffset(years=1)).to_period("M")
    both = both.set_index(both["date"].dt.to_period("M"))

    if prev_date not in both.index:
        raise ValueError(f"Missing prior-year month needed for YoY: {snap_date:%Y-%m} for {industry}.")

    emp_curr = float(both.loc[pd.Timestamp(snap_date).to_period("M"), "emp"])
    ahe_curr = float(both.loc[pd.Timestamp(snap_date).to_period("M"), "ahe"])
    emp_prev = float(both.loc[prev_date, "emp"])
    ahe_prev = float(both.loc[prev_date, "ahe"])

    emp_yoy = (emp_curr / emp_prev - 1) * 100
    ahe_yoy = (ahe_curr / ahe_prev - 1) * 100

    return {
        "Year": year,
        "Snapshot month": pd.Timestamp(snap_date).strftime("%Y-%m"),
        "Industry": industry,
        "Employment (thousands)": emp_curr,          # CES employment series are in thousands
        "Employment (people)": emp_curr * 1000,
        "Avg hourly earnings ($)": ahe_curr,
        "Employment YoY %": emp_yoy,
        "Hourly earnings YoY %": ahe_yoy,
    }

def build_year_outputs(year: int, sectors: list[tuple[str, pd.DataFrame, pd.DataFrame]], month: int | None = 12):
    rows = [year_snapshot_row(name, emp_ts, ahe_ts, year=year, month=month) for name, emp_ts, ahe_ts in sectors]
```

```
        table_df = pd.DataFrame(rows)

        heatmap_df = (table_df[["Industry", "Employment YoY %", "Hourly earnings YoY %"]]
                      .set_index("Industry")
                      .round(2))

        # format table
        table_df["Employment (thousands)"] = table_df["Employment (thousands)"].round(2)
        table_df["Employment (people)"] = table_df["Employment (people)"].round(0).astype("int64")
        table_df["Avg hourly earnings ($)"] = table_df["Avg hourly earnings ($)"].round(2)
        table_df["Employment YoY %"] = table_df["Employment YoY %"].round(2)
        table_df["Hourly earnings YoY %"] = table_df["Hourly earnings YoY %"].round(2)

        return heatmap_df, table_df
```

Heatmaps: side-by-side comparisons across years

```
    sectors = [
        ("Financial activities", fin_emp_ts, fin_ahe_ts),
        ("Professional & business services", pbs_emp_ts, pbs_ahe_ts),
    ]

    outputs = {}
    for y in [2025, 2024, 2023]:
        hm_y, table_y = build_year_outputs(y, sectors, month=12)  # tries Dec; falls back to latest month in that year
        outputs[y] = {"hm": hm_y, "table": table_y}

        fig = px.imshow(
            hm_y,
            text_auto=".2f",
            aspect="auto",
            title=f"YoY % Snapshot — {y} (month used: {table_y['Snapshot month'].iloc[0]})",
            labels={"color": "YoY %", "x": "Metric", "y": "Industry"},
        )

    import numpy as np
    import plotly.graph_objects as go
    from plotly.subplots import make_subplots

    # chronological order (oldest -> newest)
    years = [2023, 2024, 2025]

    # grab the heatmap dataframes you already built (index=Industry, columns=two YoY metrics)
    hms = {y: outputs[y]["hm"].copy() for y in years}

    # make sure all three share the same row order (union of industries, sorted)
    all_industries = sorted(set().union(*[hm.index.tolist() for hm in hms.values()]))

    # reindex so each subplot lines up row-by-row
    for y in years:
        hms[y] = hms[y].reindex(all_industries)

    # global color scale so the colors are comparable across years
    vals = np.concatenate([hms[y].to_numpy().ravel() for y in years])
    vals = vals[np.isfinite(vals)]
    abs_max = float(np.nanmax(np.abs(vals))) if len(vals) else 1.0

    fig = make_subplots(
        rows=1,
        cols=3,
        subplot_titles=[str(y) for y in years],
        specs=[[{"type": "heatmap"}, {"type": "heatmap"}, {"type": "heatmap"}]],
        horizontal_spacing=0.06
    )

    for i, y in enumerate(years, start=1):
        z = hms[y].to_numpy()
        x = list(hms[y].columns)
        ylabels = list(hms[y].index)

        fig.add_trace(
            go.Heatmap(
                z=z,
                x=x,
                y=ylabels,
                text=np.round(z, 2),
                texttemplate="%{text}",
                hovertemplate="Industry=%{y}<br>Metric=%{x}<br>YoY=%{z:.2f}%<extra></extra>",
                coloraxis="coloraxis"  # shared colorscale across subplots
            ),
            row=1, col=i
        )

    fig.update_layout(
        title="CES YoY % by Industry — Side-by-side (2023 → 2025)",
        height=650,
        # one shared colorbar/scale for all heatmaps
        coloraxis=dict(cmin=-abs_max, cmax=abs_max, colorbar=dict(title="YoY %")),
        margin=dict(l=20, r=20, t=80, b=20),
    )

    # If the y-axis labels feel too dense, you can hide them on the middle/right panels:
    fig.update_yaxes(showticklabels=True, row=1, col=1)
    fig.update_yaxes(showticklabels=False, row=1, col=2)
    fig.update_yaxes(showticklabels=False, row=1, col=3)

    fig.show()
```



Heat maps in tabular form

```
table_all = pd.concat([outputs[y]["table"] for y in [2025, 2024, 2023]], ignore_index=True)
table_all = table_all.sort_values(["Industry", "Year"], ascending = [True, False]).reset_index(drop=True)
table_all
```