LFP Analysis — 3-Layer Scaffold with Plugin Registry

This document contains a complete, runnable scaffold for a Python package called `lfp_analysis` that implements the **three-layer architecture** (Data Access / Business Logic / Presentation) plus a **plugin registry** for preprocessors, features and visualizers.

It is designed for learning — each file includes type hints, docstrings and clear input/output descriptions.

Project tree

```
lfp_analysis/
├── __init__.py
├── main.py
├── config.yaml
├── data_access/
│   ├── __init__.py
│   ├── base.py
│   ├── loader.py
│   └── filters.py
├── business_logic/
│   ├── __init__.py
│   ├── base.py
│   ├── spectral.py
│   └── statistics.py
└── presentation/
    ├── __init__.py
    ├── base.py
    └── trial_dynamics.py
```

> **Important:** Run the package from the project root (the directory that contains the `lfp_analysis` folder). Use the module runner to preserve package imports:

```
python -m lfp_analysis.main
```

`lfp_analysis/__init__.py`

```python
# lfp_analysis/__init__.py
"""Top-level package for the LFP analysis scaffold."""
__all__ = ["data_access", "business_logic", "presentation"]
```

`config.yaml`

```yaml
# config.yaml — choose the plugin names and their args
preprocessor:
  name: bandpass_filter
  args:
    low: 8.0
    high: 12.0
    sfreq: 1000.0

feature:
  name: band_power
  args:
    band: [8.0, 12.0]
    sfreq: 1000.0

visualizer:
  name: trial_dynamics
  args: {}

# dataset options for demo
dataset:
  n_sessions: 5
  n_channels: 4
  n_epochs: 20
  n_samples: 500
```

Data Access Layer

`data_access/__init__.py`

```python
# lfp_analysis/data_access/__init__.py
from .base import PREPROCESSOR_REGISTRY, register_preprocessor, Preprocessor
from .loader import load_dummy_dataset, load_from_files
from .filters import BandpassFilter
```

```
__all__ = [
    "PREPROCESSOR_REGISTRY",
    "register_preprocessor",
    "Preprocessor",
    "load_dummy_dataset",
    "load_from_files",
    "BandpassFilter",
]
```

---

data_access/base.py

```
# lfp_analysis/data_access/base.py
"""
Data access base: defines Preprocessor ABC and a registry for plugins.
"""
from typing import Callable, Dict, Type
import numpy as np

PREPROCESSOR_REGISTRY: Dict[str, Type["Preprocessor"]] = {}


def register_preprocessor(name: str) -> Callable:
    """Decorator to register a Preprocessor class with a name.

    Usage:
        @register_preprocessor("bandpass_filter")
        class BandpassFilter(Preprocessor): ...
    """
    def decorator(cls: Type["Preprocessor"]) -> Type["Preprocessor"]:
        PREPROCESSOR_REGISTRY[name] = cls
        return cls
    return decorator


class Preprocessor:
    """Abstract preprocessor.

    A preprocessor should accept and return numpy arrays with the following
canonical shapes:
      - Input: `np.ndarray` with shape (n_sessions, n_channels, n_epochs,
n_samples)
      - Output: same shape
    """
```

```
    def process(self, data: np.ndarray, **kwargs) -> np.ndarray:
        """Apply preprocessing to the entire dataset.

        Parameters
        ----------
        data : np.ndarray
            LFP data with shape (sessions, channels, epochs, samples)

        Returns
        -------
        np.ndarray
            Processed data with the same shape.
        """
        raise NotImplementedError
```

---

data_access/loader.py

```
# lfp_analysis/data_access/loader.py
"""Data loader utilities for the scaffold."""
from typing import Tuple
import numpy as np


def load_dummy_dataset(
    n_sessions: int = 2,
    n_channels: int = 4,
    n_epochs: int = 10,
    n_samples: int = 500,
) -> Tuple[np.ndarray, np.ndarray, np.ndarray, float]:
    """
    Create a synthetic dataset for demo/testing.

    Returns
    -------
    lfp : np.ndarray
        Shape (n_sessions, n_channels, n_epochs, n_samples)
    stimulus_types : np.ndarray
        Shape (n_sessions, n_epochs) with integer labels {1,2}
    time : np.ndarray
        1D array of time points for samples, length n_samples
    sfreq : float
        Sampling frequency (Hz)
    """
    sfreq = 1000.0
    t = np.linspace(-0.2, 0.8, n_samples)
```

```
    lfp = np.random.randn(n_sessions, n_channels, n_epochs, n_samples) * 1e-6
    # add a simple evoked oscillation for condition 1
    for s in range(n_sessions):
        for e in range(n_epochs):
            if (e % 2) == 0:  # condition 1
                lfp[s, :, e, :] += 1e-6 * np.sin(2 * np.pi * 10 * t)
    stimulus_types = (np.arange(n_epochs) % 2) + 1
    stimulus_types = np.tile(stimulus_types, (n_sessions, 1))
    return lfp, stimulus_types, t, sfreq


def load_from_files(path: str):
    """Placeholder: implement your dataset loader (e.g., .npy files, mat files).

    Keep the same output types as `load_dummy_dataset`.
    """
    raise NotImplementedError
```

---

data_access/filters.py

```
# lfp_analysis/data_access/filters.py
"""Example preprocessors (filters)."""
from typing import Any
import numpy as np
import scipy.signal as sps
from .base import Preprocessor, register_preprocessor


@register_preprocessor("bandpass_filter")
class BandpassFilter(Preprocessor):
    """
    Simple zero-phase Butterworth bandpass applied to each channel/epoch
independently.

    Input: (sessions, channels, epochs, samples)
    Output: same shape
    """

    def __init__(self, low: float, high: float, sfreq: float, order: int = 4):
        self.low = low
        self.high = high
        self.sfreq = sfreq
        self.order = order

    def _filter_1d(self, signal: np.ndarray) -> np.ndarray:
```

```
        b, a = sps.butter(self.order, [self.low, self.high], btype="band",
fs=self.sfreq)
        return sps.filtfilt(b, a, signal)

    def process(self, data: np.ndarray, **kwargs: Any) -> np.ndarray:
        sessions, channels, epochs, samples = data.shape
        out = np.empty_like(data)
        for s in range(sessions):
            for ch in range(channels):
                for e in range(epochs):
                    out[s, ch, e, :] = self._filter_1d(data[s, ch, e, :])
        return out
```

Business Logic Layer

`business_logic/__init__.py`

```
# lfp_analysis/business_logic/__init__.py
from .base import FEATURE_REGISTRY, register_feature, FeatureFunction
from .spectral import BandPowerFeature
from .statistics import compare_two_conditions

__all__ = [
    "FEATURE_REGISTRY",
    "register_feature",
    "FeatureFunction",
    "BandPowerFeature",
    "compare_two_conditions",
]
```

`business_logic/base.py`

```
# lfp_analysis/business_logic/base.py
from typing import Callable, Dict, Type
import numpy as np

FEATURE_REGISTRY: Dict[str, Type["FeatureFunction"]] = {}


def register_feature(name: str) -> Callable:
    def decorator(cls: Type["FeatureFunction"]) -> Type["FeatureFunction"]:
        FEATURE_REGISTRY[name] = cls
```

```
        return cls
    return decorator


class FeatureFunction:
    """Abstract feature function.

    Expected input shapes:
      - single signal: 1D array (n_samples,)
      - or full dataset: (n_sessions, n_channels, n_epochs, n_samples)

    Concrete implementations should document which inputs they accept.
    """

    def compute(self, signal: np.ndarray, **kwargs) -> np.ndarray:
        """Compute the feature.

        If `signal` is 1D: return a float (or 1-element array).
        If `signal` is a full dataset: return array shaped (n_sessions,
n_epochs) or similar.
        """
        raise NotImplementedError
```

---

business_logic/spectral.py

```
# lfp_analysis/business_logic/spectral.py
"""Spectral feature implementations (example: band power)."""
from typing import Tuple
import numpy as np
from .base import register_feature, FeatureFunction
from scipy.signal import welch


@register_feature("band_power")
class BandPowerFeature(FeatureFunction):
    """
    Band power computed with Welch's method.

    Parameters
    ----------
    band : Tuple[float, float]
        (low_hz, high_hz)
    sfreq : float
        sampling frequency
    """
```

```
    def __init__(self, band: Tuple[float, float], sfreq: float):
        self.band = band
        self.sfreq = sfreq

    def _bandpower_1d(self, signal: np.ndarray) -> float:
        freqs, psd = welch(signal, fs=self.sfreq, nperseg=min(256, len(signal)))
        mask = (freqs >= self.band[0]) & (freqs <= self.band[1])
        # Integrate PSD over band
        return float(np.trapz(psd[mask], freqs[mask]))

    def compute(self, signal: np.ndarray, **kwargs) -> np.ndarray:
        """Compute bandpower.

        If `signal` has shape (n_samples,) -> return float.
        If `signal` has shape (n_sessions, n_channels, n_epochs, n_samples) ->
        return (n_sessions, n_epochs) averaged across channels.
        """
        if signal.ndim == 1:
            return self._bandpower_1d(signal)

        # assume full dataset
        sessions, channels, epochs, samples = signal.shape
        out = np.zeros((sessions, epochs), dtype=float)
        for s in range(sessions):
            for e in range(epochs):
                # average channels for a single trial
                avg_signal = signal[s, :, e, :].mean(axis=0)
                out[s, e] = self._bandpower_1d(avg_signal)
        return out
```

---

business_logic/statistics.py

```
# lfp_analysis/business_logic/statistics.py
"""Simple statistical helpers.

This module belongs to business logic because statistical comparisons are part
of the analysis "business".
"""
from typing import Dict
import numpy as np
from scipy.stats import ttest_ind


def compare_two_conditions(features_by_condition: Dict[str, np.ndarray]) ->
```

```
Dict[str, float]:
    """Compare two groups ("cond1" and "cond2").

    Expects:
      features_by_condition = {"cond1": array_like, "cond2": array_like}

    Returns dict with statistic and p-value.
    """
    a = np.asarray(features_by_condition["cond1"]).ravel()
    b = np.asarray(features_by_condition["cond2"]).ravel()
    stat, p = ttest_ind(a, b, equal_var=False)
    return {"stat": float(stat), "p": float(p)}
```

Presentation Layer

`presentation/__init__.py`

```
# lfp_analysis/presentation/__init__.py
from .base import VISUALIZER_REGISTRY, register_visualizer, Visualizer
from .trial_dynamics import TrialDynamicsVisualizer

__all__ = [
    "VISUALIZER_REGISTRY",
    "register_visualizer",
    "Visualizer",
    "TrialDynamicsVisualizer",
]
```

`presentation/base.py`

```
# lfp_analysis/presentation/base.py
"""Visualizer base and registry."""
from typing import Callable, Dict, Type
import numpy as np

VISUALIZER_REGISTRY:
```