1. Basic Usage (Binary Classification)

```python
from sklearn.metrics import precision_score

# Assuming y_test (true labels) and y_pred (predicted labels)
precision = precision_score(y_test, y_pred)  # Correct for binary classification
```

- **Default Behavior**:
  - For **binary classification**, precision is calculated for the `1` class by default.
  - If your labels are `[0, 1]`, this measures precision for class `1`.

---

2. Key Parameters to Customize

A. Multiclass Classification

Use `average` to specify how to aggregate precision for multiple classes:

```python
precision = precision_score(y_test, y_pred, average='macro')  # Options: 'micro', 'weighted', None
```

- `'macro'`: Unweighted mean across classes.
- `'micro'`: Global precision (ignoring class imbalance).
- `'weighted'`: Weighted mean (accounts for class imbalance).
- `None`: Returns precision for each class individually.

B. Specify the Positive Label

If your positive class is not `1` (e.g., `"cat"` in `["dog", "cat"]`):

```python
precision = precision_score(y_test, y_pred, pos_label="cat")
```

C. Handling Zero-Division

If no predicted positives exist (division by zero), set a fallback value:

```python
precision = precision_score(y_test, y_pred, zero_division=0)  # Returns 0 instead of error
```

---

3. Full Example with MLflow Logging

```python
from sklearn.metrics import precision_score
import mlflow
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load data
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Train model
model = LogisticRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# Calculate precision (multiclass)
precision = precision_score(y_test, y_pred, average='macro')

# Log to MLflow
with mlflow.start_run():
    mlflow.log_metric("precision", precision)
    mlflow.sklearn.log_model(model, "model")
```

---

4. Common Pitfalls

1. **Binary vs. Multiclass**:
   - If `y_test` / `y_pred` contain >2 classes but you omit `average`, scikit-learn will raise a warning.
   - Always specify `average` for multiclass problems.
2. **Imbalanced Data**:
   - Precision can be misleading if the positive class is rare. Combine with `recall_score` or use `f1_score`.
3. **Probabilistic Predictions**:
   - If your model outputs probabilities (e.g., `model.predict_proba()`), threshold predictions first:

```python
y_pred = (model.predict_proba(X_test)[:, 1] > 0.5).astype(int)
```

---

5. Precision vs. Other Metrics

Metric	Formula	Use Case
Precision	`TP / (TP + FP)`	Focus on minimizing false positives (e.g., spam detection).
Recall	`TP / (TP + FN)`	Focus on minimizing false negatives (e.g., cancer screening).
F1-Score	`2 * (Precision * Recall) / (Precision + Recall)`	Balance precision/recall.

Best Practices

- For **binary classification**, ensure labels are `[0, 1]` or use `pos_label`.
- For **multiclass**, always specify `average`.
- Log precision alongside other metrics (e.g., recall, accuracy) in MLflow for full visibility.