Python AI/ML Essentials 2026 Cheatsheet

A comprehensive reference guide for developers and teachers covering essential Python libraries, syntax, and implementations for machine learning and artificial intelligence projects.

NumPy: Numerical Computing Fundamentals

NumPy provides efficient multidimensional arrays and mathematical operations core to all ML/AI work.

Core Array Operations

```
import numpy as np
```

Array creation

```
arr = np.array([1, 2, 3])
zeros = np.zeros((3, 4))
ones = np.ones(5)
range_arr = np.arange(0, 10, 2)
linspace = np.linspace(0, 1, 11)
random_arr = np.random.rand(3, 3)
```

Reshaping and transposing

```
reshaped = arr.reshape(2, -1)
flattened = arr.flatten()
transposed = arr.T
```

Indexing and slicing

```
element = arr[0]
slice_arr = arr[1:3]
fancy_index = arr[[0, 2]]
```

Mathematical Operations

Element-wise operations

```
sum_result = np.sum(arr)
mean_val = np.mean(arr)
std_val = np.std(arr)
max_val = np.max(arr)
min_val = np.min(arr)
```

Linear algebra

```
dot_product = np.dot(arr1, arr2)
matrix_mult = np.matmul(matrix1, matrix2)
inverse = np.linalg.inv(matrix)
eigenvalues, eigenvectors = np.linalg.eig(matrix)
```

Statistical operations

```
correlation = np.corrcoef(data1, data2)
sorted_arr = np.sort(arr)
unique_vals = np.unique(arr)
```

Pandas: Data Manipulation and Analysis

Pandas provides DataFrames for loading, cleaning, and exploring datasets essential for ML preprocessing.

DataFrame Basics

```
import pandas as pd
```

Creating DataFrames

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df = pd.read_csv('data.csv')
df = pd.read_excel('data.xlsx')
```

Inspecting data

```
df.head()
df.shape
df.info()
df.describe()
```

Accessing data

```
column = df['A']
row = df.iloc[0]
subset = df.loc[df['A'] > 1]
```

Data Cleaning and Transformation

Missing values

```
df.isnull().sum()
df.fillna(value=0)
df.dropna()
```

Encoding categorical variables

```
df['category'] = pd.Categorical(df['category']).codes
```

Scaling and normalization

```
df['normalized'] = (df['value'] - df['value'].mean()) /
df['value'].std()
```

Merging and concatenating

```
merged = pd.merge(df1, df2, on='key')
concatenated = pd.concat([df1, df2], axis=0)
```

Grouping and aggregation

```
grouped = df.groupby('category')['value'].mean()
pivot = df.pivot_table(values='value', index='date',
columns='category')
```

Feature engineering

```
df['new_feature'] = df['A'] * df['B']
df['squared'] = df['value'] ** 2
```

Scikit-Learn: Machine Learning Models

Scikit-Learn provides consistent APIs for building, evaluating, and deploying ML models.

Model Training Pipeline

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

Train-test split

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

Scaling features

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Pipeline for reproducibility

```
pipeline = Pipeline([
('scaler', StandardScaler()),
('model', LogisticRegression())
```

```
])
pipeline.fit(X_train, y_train)
```

Classification Models

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
```

Logistic Regression (binary/multiclass)

```
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

Random Forest (handles non-linear patterns)

```
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)
feature_importance = rf.feature_importances_
```

Support Vector Machine

```
svm = SVC(kernel='rbf', C=1.0)
svm.fit(X_train, y_train)
```

Regression Models

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.ensemble import GradientBoostingRegressor
```

Linear Regression

```
lr = LinearRegression()
lr.fit(X_train, y_train)
coefficients = lr.coef_
```


Ridge (L2 regularization) and Lasso (L1 regularization)

```
ridge = Ridge(alpha=1.0)
lasso = Lasso(alpha=0.1)
```

Gradient Boosting

```
gb = GradientBoostingRegressor(n_estimators=100)
gb.fit(X_train, y_train)
```

Clustering Models

```
from sklearn.cluster import KMeans, DBSCAN
from sklearn.decomposition import PCA
```

K-Means

```
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(X)
centers = kmeans.cluster_centers_
```

DBSCAN (density-based)

```
dbscan = DBSCAN(eps=0.5, min_samples=5)
labels = dbscan.fit_predict(X)
```

Dimensionality Reduction

```
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X)
explained_variance = pca.explained_variance_ratio_
```

Model Evaluation

```
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score
from sklearn.metrics import mean_squared_error, r2_score,
roc_auc_score
from sklearn.model_selection import cross_val_score
```

Classification metrics

```
accuracy = accuracy_score(y_test, predictions)
precision = precision_score(y_test, predictions)
recall = recall_score(y_test, predictions)
f1 = f1_score(y_test, predictions)
roc_auc = roc_auc_score(y_test, predictions_proba)
```

Regression metrics

```
mse = mean_squared_error(y_test, predictions)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, predictions)
```

Cross-validation

```
cv_scores = cross_val_score(model, X, y, cv=5)
print(f"Mean CV Score: {cv_scores.mean():.3f} (+/-
{cv_scores.std():.3f})")
```

Hyperparameter Tuning

```
from sklearn.model_selection import GridSearchCV,
RandomizedSearchCV
```

Grid Search

```
param_grid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf']}
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
```

Random Search (faster for large parameter spaces)

```
param_dist = {'n_estimators': [50, 100, 200], 'max_depth': [5,
10, 15]}
random_search = RandomizedSearchCV(RandomForestClassifier(),
param_dist, n_iter=10, cv=5)
random_search.fit(X_train, y_train)
```

Hugging Face Transformers: Modern NLP/GenAI

Hugging Face Transformers provides pre-trained models for text generation, classification, embeddings, and more.

Text Generation

```
from transformers import pipeline, AutoTokenizer,
AutoModelForCausalLM
```

Simple pipeline interface

```
generator = pipeline('text-generation', model='distilgpt2')
result = generator('The future of AI is', max_length=50,
num_return_sequences=1)
```

Low-level control

```
tokenizer = AutoTokenizer.from_pretrained('distilgpt2')
model = AutoModelForCausalLM.from_pretrained('distilgpt2')

inputs = tokenizer('Hello, I am', return_tensors='pt')
outputs = model.generate(**inputs, max_length=50)
generated_text = tokenizer.decode(outputs[0],
skip_special_tokens=True)
```

Text Classification

```
from transformers import pipeline
```

Sentiment analysis

```
classifier = pipeline('sentiment-analysis', model='distilbert-
base-uncased-finetuned-sst-2-english')
result = classifier('I love this product!')
```

Custom classification

```
classifier = pipeline('zero-shot-classification',
model='facebook/bart-large-mnli')
result = classifier('This is a great movie', ['positive',
'negative', 'neutral'])
```

Text Embeddings (Semantic Search)

```
from sentence_transformers import SentenceTransformer
```

Load pre-trained model

```
model = SentenceTransformer('all-MiniLM-L6-v2')
```

Generate embeddings

```
sentences = ['I love machine learning', 'AI is fascinating']
embeddings = model.encode(sentences)
```

Compute similarity

```
from sklearn.metrics.pairwise import cosine_similarity
similarity = cosine_similarity([embeddings[0]],
[embeddings[1]])[0][0]
```

Question Answering

```
from transformers import pipeline

qa_pipeline = pipeline('question-answering', model='distilbert-
base-cased-distilled-squad')

context = 'Machine learning is a subset of artificial
intelligence. It focuses on data-driven learning.'
question = 'What is machine learning?'

result = qa_pipeline(question=question, context=context)
print(f"Answer: {result['answer']}")
```

Chat/Conversational Models

```
from transformers import pipeline
import torch

chat = pipeline('text-generation', model='meta-llama/Meta-Llama-
3-8B-Instruct',
device_map='auto', torch_dtype=torch.bfloat16)

messages = [
{'role': 'system', 'content': 'You are a helpful AI assistant.'},
```

```
{'role': 'user', 'content': 'What are the fundamentals of machine
learning?'}
]

response = chat(messages, max_new_tokens=256)
```

TensorFlow/Keras: Deep Learning

TensorFlow with Keras API provides high-level interfaces for building neural networks.

Sequential Neural Networks

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Simple feedforward network

```
model = keras.Sequential([
layers.Dense(64, activation='relu', input_shape=(10,)),
layers.Dropout(0.2),
layers.Dense(32, activation='relu'),
layers.Dense(1, activation='sigmoid')
])
```

Compile and train

```
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=10, batch_size=32,
validation_split=0.2)
```

Predict

```
predictions = model.predict(X_test)
```


Convolutional Neural Networks (Image)

```
from tensorflow.keras.applications import MobileNetV2
```

Transfer learning with pre-trained model

```
base_model = MobileNetV2(input_shape=(224, 224, 3),
include_top=False, weights='imagenet')
base_model.trainable = False

model = keras.Sequential([
base_model,
layers.GlobalAveragePooling2D(),
```

```
layers.Dense(256, activation='relu'),
layers.Dropout(0.2),
layers.Dense(num_classes, activation='softmax')
])
```

Recurrent Neural Networks (Sequences)

LSTM for sequential data

```
model = keras.Sequential([
layers.LSTM(128, return_sequences=True, input_shape=(timesteps,
features)),
layers.LSTM(64),
layers.Dense(32, activation='relu'),
layers.Dense(num_classes, activation='softmax')
])
```

PyTorch: Deep Learning with Flexibility

PyTorch provides dynamic computation graphs for research and production ML.

Basic Tensor Operations

```
import torch
import torch.nn as nn
```

Tensor creation

```
tensor = torch.tensor([1.0, 2.0, 3.0])
zeros = torch.zeros(3, 3)
ones = torch.ones(2, 4)
random = torch.randn(3, 3)
```


Device management (CPU/GPU)

```
device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
tensor = tensor.to(device)
```

Autograd (automatic differentiation)

```
x = torch.tensor([2.0, 3.0], requires_grad=True)
y = x ** 2
loss = y.sum()
loss.backward()
print(x.grad)
```

Neural Network Definition

```
class SimpleMLP(nn.Module):
def init(self, input_size, hidden_size, output_size):
super(SimpleMLP, self).init()
self.fc1 = nn.Linear(input_size, hidden_size)
self.relu = nn.ReLU()
self.fc2 = nn.Linear(hidden_size, output_size)
```

```
def forward(self, x):
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)
    return x
```

```
model = SimpleMLP(10, 64, 2)
```

Training Loop

```
import torch.optim as optim
```

```
model = SimpleMLP(10, 64, 2)
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.CrossEntropyLoss()
```

```
for epoch in range(10):
for batch_X, batch_y in dataloader:
# Forward pass
outputs = model(batch_X)
loss = loss_fn(outputs, batch_y)
```

```
    # Backward pass
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Matplotlib & Seaborn: Visualization

Essential for exploratory data analysis and presenting model results.

Matplotlib Basics

```
import matplotlib.pyplot as plt
```

Line plots

```
plt.plot([1, 2, 3, 4], [1, 4, 2, 3])
plt.xlabel('X-axis')
```

```
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')
plt.show()
```

Scatter plots

```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')
plt.colorbar()
```

Subplots

```
fig, axes = plt.subplots(2, 2, figsize=(10, 8))
axes[0, 0].plot([1, 2, 3])
axes[0, 1].bar([1, 2, 3], [2, 3, 1])
plt.tight_layout()
```

Seaborn for Statistical Visualization

```
import seaborn as sns
```

Heatmap for correlation

```
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
```

Distribution plots

```
sns.histplot(data=df, x='value', kde=True)
sns.kdeplot(data=df, x='value')
```

Categorical plots

```
sns.boxplot(data=df, x='category', y='value')
sns.violinplot(data=df, x='category', y='value')
sns.stripplot(data=df, x='category', y='value', jitter=True)
```


Essential ML Workflow

Complete Pipeline Template

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report,
confusion_matrix
```

1. Load and explore data

```
df = pd.read_csv('data.csv')
print(df.head(), df.info(), df.describe())
```

2. Handle missing values and outliers

```
df.fillna(df.mean(), inplace=True)
Q1, Q3 = df.quantile([0.25, 0.75])
df = df[~((df < (Q1 - 1.5 * (Q3 - Q1))) | (df > (Q3 + 1.5 * (Q3 -
Q1)))).any(axis=1)]
```

3. Feature engineering

```
df['feature1_squared'] = df['feature1'] ** 2
df['feature_ratio'] = df['feature1'] / df['feature2']
```

4. Prepare data

```
X = df.drop('target', axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

5. Scale features

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

6. Train model

```
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train_scaled, y_train)
```

7. Evaluate

```
predictions = model.predict(X_test_scaled)
print(classification_report(y_test, predictions))
print(confusion_matrix(y_test, predictions))
```


8. Feature importance

```
importance = pd.DataFrame({'feature': X.columns, 'importance':
model.feature_importances_})
print(importance.sort_values('importance', ascending=False))
```

Key Concepts Quick Reference

Overfitting vs Underfitting

Overfitting: Model memorizes training data, poor test performance. Solutions: regularization, more data, simpler model. Underfitting: Model too simple for complexity. Solutions: add features, increase model complexity, train longer.

Cross-Validation

K-fold CV: Split data into K parts, train K models using K-1 parts, test on 1 part. Prevents overfitting, better generalization estimate.

Class Imbalance

When target classes unequal: use stratified split, class weights, oversampling (SMOTE), or appropriate metrics (precision, recall, F1) instead of accuracy.

Feature Scaling

StandardScaler: (X - mean) / std. MinMaxScaler: (X - min) / (max - min). Essential for distance-based (KNN, K-Means, SVM) and gradient-descent models.

Regularization

L1 (Lasso): Encourages sparsity, feature selection. L2 (Ridge): Shrinks coefficients, handles multicollinearity. Elastic Net: Combines both.

2026 Best Practices

- Use pre-trained transformers from Hugging Face Hub for NLP tasks rather than training from scratch.
- Implement MLOps pipelines with model versioning and monitoring for production.
- Always validate on unseen data; separate train/validation/test sets.
- Use modern libraries (transformers, datasets) optimized for multimodal and agentic workflows.
- Document experiments, hyperparameters, and results for reproducibility.