1. **Model building in `scikit-learn`.** (These steps are from the tutorial available in scikit-learn.org.)

   (a) Fitting and predicting: estimator basics. *Estimators* are machine learning algorithms that can be fitted to some data. Test the following example, and inspect the parameters of the model created.

   ```
   from sklearn.ensemble import RandomForestClassifier
   clf = RandomForestClassifier(random_state=0)
   X = [[ 1,  2,  3],  # 2 samples, 3 features
        [11, 12, 13]]
   y = [0, 1]  # classes of each sample
   clf.fit(X, y)
   ```

   (b) X is a matrix, typically `n_samples` $\times$ `n_features`: observations are represented as rows and features as columns. In supervised learning, the target values y which are real numbers for regression tasks, or discrete values (typically) integers for classification. Both X and y are usually expected to be `numpy` arrays. Once the estimator is fitted, it can be used for predicting target values of new data; try:

   ```
   clf.predict(X)  # predict classes of the training data
   clf.predict([[4, 5, 6], [14, 15, 16]])  # predict classes of new data
   ```

   (c) A typical pipeline in machine learning consists of a pre-processing step that transforms or imputes the data, and a final predictor that predicts target values. A *transformer* is used, e.g., when we want to scale features to comparable ranges; *imputing* is used to fill empty values. In scikit-learn, pre-processors and transformers follow the same API as the estimator objects; instead of `predict`, they provide a `transform` method that outputs a newly transformed sample matrix X. Try:

   ```
   from sklearn.preprocessing import StandardScaler
   StandardScaler().fit(X).transform(X)
   ```

   (d) Transformers and estimators (predictors) can be combined into a unifying object called a *pipeline*, offering the same API as a regular estimator. Study the following code:

   ```
   from sklearn.preprocessing import StandardScaler
   from sklearn.linear_model import LogisticRegression
   from sklearn.pipeline import make_pipeline
   from sklearn.datasets import load_iris
   from sklearn.model_selection import train_test_split
   from sklearn.metrics import accuracy_score

   # create a pipeline object
   pipe = make_pipeline(
       StandardScaler(),
       LogisticRegression()
   )

   # load the iris dataset and split it into train and test sets
   X, y = load_iris(return_X_y=True)
   X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

   # fit the whole pipeline
   ```

```
pipe.fit(X_train, y_train)

# we can now use it like any other estimator
accuracy_score(pipe.predict(X_test), y_test)
```

(e) Model evaluation: fitting a model to some data does not entail that it will predict well on unseen data. This needs to be evaluated; the most common tool is *cross-validation*, which splits a dataset into train and test sets in a systematic way. Study the following 5-fold cross-validation procedure:

```
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_validate

X, y = make_regression(n_samples=1000, random_state=0)
lr = LinearRegression()

result = cross_validate(lr, X, y)  # defaults to 5-fold CV
result['test_score']  # r_squared score is high because dataset is easy
```

(f) All estimators have *hyper-parameters*, which cannot be learned; rather, they can be tuned. The generalization power of an estimator often critically depends on a few parameters. For example, a `RandomForestRegressor` has hyper-parameters `n_estimators` (determining the number of trees in the forest) and `max_depth` (determining the maximum depth of each tree). Good values for hyper-parameters are difficulty to know in advance, and hence are determined experimentally (using cross-validation). In the following example, parameters of a random forest are searched with `RandomizedSearchCV`. When the search is over, `RandomizedSearchCV` behaves as a RandomForestRegressor that has been fitted with the best set of parameters.

```
from sklearn.datasets import fetch_california_housing
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import train_test_split
from scipy.stats import randint

X, y = fetch_california_housing(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# define the parameter space that will be searched over
param_distributions = {'n_estimators': randint(1, 5), 'max_depth': randint(5, 10)}

# create a searchCV object and fit it to the data
search = RandomizedSearchCV(estimator=RandomForestRegressor(random_state=0),
                            n_iter=5,
                            param_distributions=param_distributions,
                            random_state=0)
search.fit(X_train, y_train)
print(search.best_params_)

# the search object now acts like a normal random forest estimator
# with max_depth=9 and n_estimators=4
search.score(X_test, y_test)
```

2. For this exercise, consider the dataset `iris`; it can be loaded in `scikit-learn` with

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
```

Information about this dataset is available in the attribute `iris.DESCR`, and the names of the classes in `iris.target_names`.

We may divide this dataset using the so-called *leave-one-out* method, making a training set with all the examples except one (which can then be used for testing) with

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=956, test_size=1)
```

A decision tree model may be trained with `X_train, y_train`, and then visualized with:

```
from matplotlib import pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
clf = DecisionTreeClassifier(random_state=0)
clf.fit(X_train, y_train)
tree.plot_tree(clf)
plt.show()
```

The class of a new example (e.g., `X_test`) can now be determined with

```
y_pred = clf.predict(X_test)
print(iris.target_names[y_pred])
```

The *"probability"* of that classification can be inspected with `y_prob = clf.predict_proba(X_test)`. Beware that Python starts indexing lists with zero.

(a) Inspect the example left for testing. Determine its true class.

(b) Construct a decision tree classifier with the hyperparameter `max_leaf_nodes=3`. By observing the tree, determine the class attibutted to the example left out.

(c) Determine how `scikit-learn` classifies the example left out, using the previous model.

(d) Construct a decision tree classifier with the hyperparameter `max_depth=3`. By observing the tree, determine how the example left out can be classified.

(e) Determine how `scikit-learn` classifies the example left out, using the previous model.

(f) Construct a decision tree classifier with the hyperparameter `min_samples_split=3`. By observing the tree, determine how the example left out is classified.

(g) Determine how `scikit-learn` classifies the example left out, using the previous model.