1.import random

```
# Pre-defined evidence (modify as needed)
test_results = {"WBC Count": 15000, "Imaging": "Possible appendicitis"}
intraoperative_findings = "Bleeding encountered during initial incision"
expert_consultation = "Proceed cautiously, consider potential for atypical presentation"

# Decision point and answer options
decision_prompt = "Based on the available evidence, what is the best course of action?"
answer_options = [
    "Continue with the laparoscopic appendectomy as planned.",
    "Convert to an open procedure to gain better visibility.",
    "Consult with another surgical specialist for further guidance."
]

def simulate_experiment():
    # Present background information
    print("Simulating an appendectomy...")
    print("Background: Patient presents with classic appendicitis symptoms.")
    # Present evidence
    print("Pre-operative Test Results:", test_results)
    print("Intra-operative Findings:", intraoperative_findings)
    print("Expert Consultation:", expert_consultation)
    # Present decision point
    print(decision_prompt)
    for i, option in enumerate(answer_options):
        print(f"{i+1}. {option}")

    # Simulate user input (replace with actual user input mechanism)
    user_choice = input("Select an option (1, 2, 3): ")  # Modified to accept user input
    while user_choice not in ['1', '2', '3']:
        print("Invalid choice. Please select 1, 2, or 3.")
        user_choice = input("Select an option (1, 2, 3): ")
    user_choice = int(user_choice)

    print("Selected option:", user_choice)
    # Prompt for justification
    justification = input("Explain your reasoning for this choice: ")
    print("Justification:", justification)
    evaluate_justification(justification, user_choice)

def evaluate_justification(justification, user_choice):
    # Implement your evaluation logic here
    # For simplicity, let's provide basic feedback based on the chosen option
    if user_choice == 1:
        print("Your choice: Continue with the laparoscopic appendectomy.")
        print("Feedback: Proceeding cautiously with the planned procedure.")
    elif user_choice == 2:
        print("Your choice: Convert to an open procedure.")
        print("Feedback: Opting for better visibility due to encountered bleeding.")
    elif user_choice == 3:
        print("Your choice: Consult with another surgical specialist.")
        print("Feedback: Seeking further guidance due to uncertain circumstances.")

# Run the simulation
simulate_experiment()
```

2.import pandas as pd
```
from scipy import stats

# Data
fertilizer_conc = ["Control", "Low", "Medium", "High"]
plant_height = [10, 12, 15, 18]  # Average height for each group

# Create dataframe
data = pd.DataFrame({"Fertilizer": fertilizer_conc, "Height": plant_height})

# Descriptive statistics for each group
print(data.groupby('Fertilizer')['Height'].describe())

# Hypothesis test (ANOVA)
anova_results = stats.f_oneway(
    data[data['Fertilizer'] == "Control"]["Height"],
    data[data['Fertilizer'] == "Low"]["Height"],
    data[data['Fertilizer'] == "Medium"]["Height"],
    data[data['Fertilizer'] == "High"]["Height"]
)

# Print ANOVA results
print("ANOVA p-value:", anova_results.pvalue)

# Interpretation based on results
if anova_results.pvalue < 0.05:  # Adjust significance level as needed
    print("There is a statistically significant difference in plant height between groups. Further analysis is needed to identify specific relationships between fertilizer concentration and plant growth.")
else:
    print("There is no statistically significant evidence that fertilizer concentration affects plant height in this data set.")
```

3.import random
```
# Define events
event_A = "It will rain today"
event_B = "The bus will be late"

# Assign probabilities to events (P(A) and P(B))
probability_A = 0.6
probability_B = 0.4

# Function to calculate the probability of an event (P(X))
def get_probability(event):
    if event == event_A:
        return probability_A
    elif event == event_B:
        return probability_B
    else:
        print("Invalid event")
        return None

# Simulate 1000 trials
trials = 1000
rain_count = 0
late_bus_count = 0
```

```
both_count = 0

for _ in range(trials):
    # Simulate events happening
    if random.random() < probability_A:
        rain_count += 1
    if random.random() < probability_B:
        late_bus_count += 1
    if rain_count and late_bus_count:
        both_count += 1

# Estimated probabilities based on simulations
estimated_probability_A = rain_count / trials
estimated_probability_B = late_bus_count / trials
estimated_intersection_probability = both_count / trials
estimated_union_probability = estimated_probability_A + estimated_probability_B - estimated_intersection_probability

# Print results
print("Event A:", event_A)
print("Estimated P(A):", estimated_probability_A)
print("Event B:", event_B)
print("Estimated P(B):", estimated_probability_B)
print("Estimated P(A and B):", estimated_intersection_probability)
print("Estimated P(A or B):", estimated_union_probability)
```

```
4.def analyze_believability(claim, source, sensationalism=0):
    """
    This function analyzes the believability of a claim based on source and sensationalism.

    Args:
        claim (str): The textual claim to be analyzed.
        source (str): The source of the claim (e.g., news website name).
        sensationalism (int, optional): A score indicating the level of sensational language (default 0).

    Returns:
        str: A basic believability classification (e.g., "Likely Believable", "Needs Verification").
    """
    source_credibility = {"Reputable News": 1, "Social Media": 0.5, "Anonymous": 0.2}
    credibility_score = source_credibility.get(source, 0.5)  # Default 0.5 for unknown sources
    # Adjust score based on sensationalism (more exclamation points = less believable)
    credibility_score -= sensationalism * 0.1
    if credibility_score > 0.7:
        return "Likely Believable"
    elif credibility_score > 0.4:
        return "Needs Verification"
    else:
        return "Likely Unbelievable"

# Example usage
claim = "Giant lizards discovered living in the Amazon rainforest!"
source = "Unknown Social Media Post"
believability = analyze_believability(claim, source, sensationalism=2)  # High sensationalism score (2)
print(f"Claim: {claim}")
print(f"Source: {source}")
print(f"Believability: {believability}")
```

5.import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

```
# Load the dataset
data = pd.read_csv("iris.csv")  # Replace with your dataset path

# Preprocess data (handle missing values, encode categorical features)
# For simplicity, let's assume no missing values and no categorical features to encode
# You may need to perform preprocessing steps here

# Separate features (X) and target variable (y)
X = data.drop("target", axis=1)
y = data["target"]

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train the decision tree model
model = DecisionTreeClassifier()
model.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = model.predict(X_test)

# Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Placeholder for iterative refinement process
# Analyze model's performance, gather user feedback, and refine rules
# Repeat iteration until desired performance is achieved
# You may want to visualize the decision tree and analyze its structure

# Retrain the model with refined rules (if applicable)
# For simplicity, let's skip this step in the placeholder

# Evaluate the performance of the refined model (if applicable)
# For simplicity, let's skip this step in the placeholder
```

6.import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

```
# Load and preprocess data
data = pd.read_csv('your_data.csv')  # Replace 'your_data.csv' with your actual file
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)

# Choose number of clusters
n_clusters = 4  # Adjust as needed

# Create and train KMeans model
```

```
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
kmeans.fit(scaled_data)

# Assign cluster labels to data points
data['cluster'] = kmeans.labels_

# Analyze clusters: Print cluster centers
cluster_centers = kmeans.cluster_centers_
print("Cluster centers:", cluster_centers)

# Visualize clusters
plt.scatter(scaled_data[:, 0], scaled_data[:, 1], c=data['cluster'], cmap='viridis')
plt.title('Yeast Gene Expression Clusters')
plt.xlabel('Gene 1')
plt.ylabel('Gene 2')
plt.show()

# Further analysis: Compute summary statistics of cluster means
clustered_data = data.groupby('cluster')
print(clustered_data.mean().describe())
```

```
7.from owlready2 import *

# Define the ontology
plant_phenotype_ontology = Ontology("http://plantphenotype.org/ontology.owl")

# Define classes
class Leaf(Thing):
    pass

class Shape(Thing):
    pass

class Size(DataRange):
    pass

# Define relationships (object properties & data properties)
has_shape = ObjectProperty(domain=Leaf, range=Shape)
has_size = DataProperty(domain=Leaf, range=Size)

# Define data types for attributes
float_datatype = DatatypeFactory.getBuiltInDatatype(xmlschema="float")

# Example instance
oak_leaf = Leaf(name="Oak Leaf")
oak_leaf.has_shape = "elliptical"
oak_leaf.has_size = 10.5  # using float data type

# Save the ontology
plant_phenotype_ontology.save(file="plant_phenotype.owl", format="owl")
print("Ontology created and saved successfully!")
```