📊 STATISTICS FOR DATA SCIENCE

Section 4: Inferential Statistics - Making Conclusions from Samples

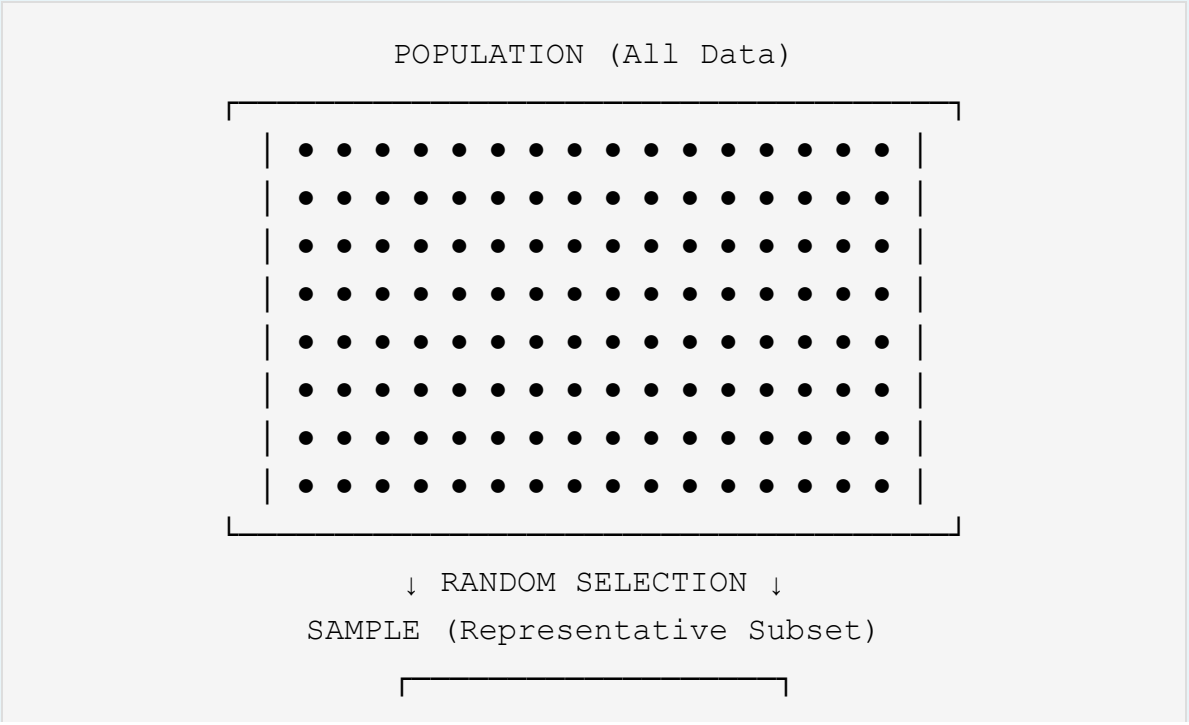
4.1 Population vs Sample

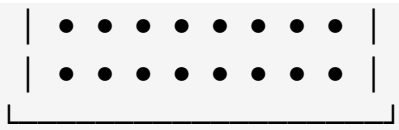
🎯 Core Concept

Definition: Inferential statistics lets you make conclusions about an entire population based on a smaller sample.
Population: The entire group you want to study (all possible data points)
Sample: A smaller group selected from the population (representative subset)

Real-World Examples:
• *Population: All 1.4 billion Indians | Sample: 10,000 randomly selected Indians*
• *Population: All smartphones ever made | Sample: 500 phones tested for quality*
• *Population: All students in India | Sample: 5,000 students surveyed*

📊 Population vs Sample Visualization





Population = 100% of data (often too large to study)
Sample = Small, representative subset (practical to study)

🤔 Why is this important & what is the use?

Key Insight: You can't always study everyone. A sample saves time, money, and effort while still providing reliable insights.

What it tells about data:

- **Population parameters (true values for entire group)**
- **Sample statistics (estimated values from sample)**

How it helps in decision-making:

Application	Population	Sample	Insight Gained
Political Polls	**100 million voters**	**5,000 voters**	**Predicts election outcome**
Clinical Trials	**Millions of patients**	**1,000 people**	**Predicts drug effect on millions**
Product Testing	**Entire production batch**	**100 products**	**Predicts quality of entire batch**
Market Research	**1 million customers**	**1,000 customers**	**Understands preferences of all**

```
import numpy as np

# Simulating population and sample
population_size = 1000000  # 1 million population
```

```
sample_size = 1000          # 1,000 sample

# Generate population data (e.g., incomes with mean 50000, std
15000)
np.random.seed(42)
population_incomes = np.random.normal(50000, 15000,
population_size)

# Take a random sample
sample_indices = np.random.choice(population_size, sample_size,
replace=False)
sample_incomes = population_incomes[sample_indices]

# Calculate statistics
pop_mean = np.mean(population_incomes)
sample_mean = np.mean(sample_incomes)
pop_std = np.std(population_incomes)
sample_std = np.std(sample_incomes)

print("=== POPULATION vs SAMPLE COMPARISON ===")
print(f"Population Size: {population_size:,}")
print(f"Sample Size: {sample_size:,}")
print(f"\nPopulation Mean Income: ${pop_mean:,.2f}")
print(f"Sample Mean Income: ${sample_mean:,.2f}")
print(f"Difference: ${abs(pop_mean - sample_mean):,.2f}")
print(f"\nPopulation Std Dev: ${pop_std:,.2f}")
print(f"Sample Std Dev: ${sample_std:,.2f}")
print(f"\nAccuracy: {100*(1 - abs(pop_mean -
sample_mean)/pop_mean):.2f}%")

# Calculate confidence interval
from scipy import stats
z_score = 1.96  # 95% confidence
margin_error = z_score * (sample_std / np.sqrt(sample_size))
ci_lower = sample_mean - margin_error
ci_upper = sample_mean + margin_error

print(f"\n95% Confidence Interval for Population Mean:")
print(f"${ci_lower:,.2f} to ${ci_upper:,.2f}")
print(f"True Population Mean (${pop_mean:,.2f}) falls within:
```

```
{ci_lower <= pop_mean <= ci_upper}")
```

4.2 Sampling Techniques

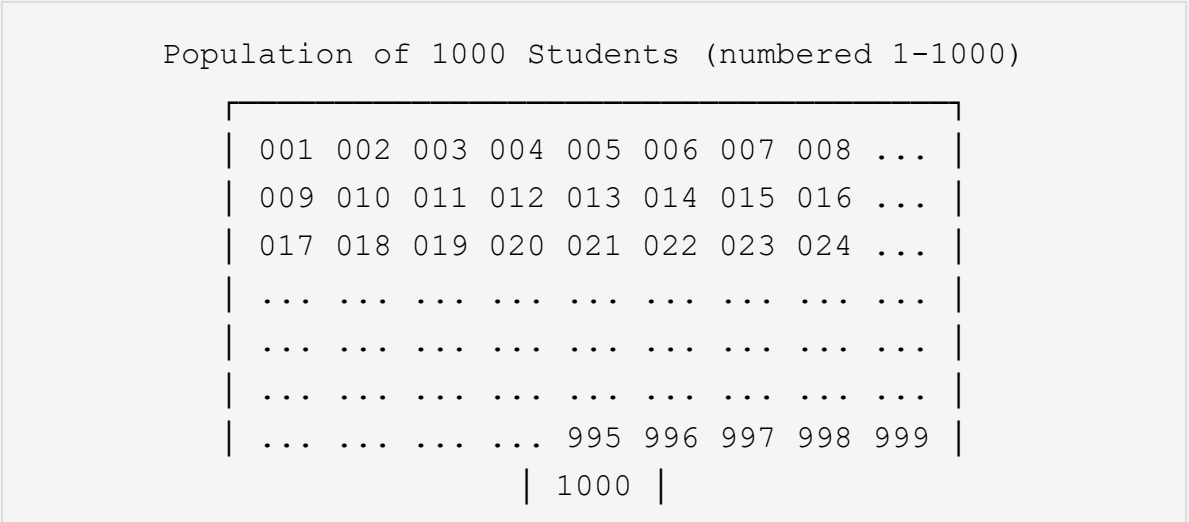
Random Sampling

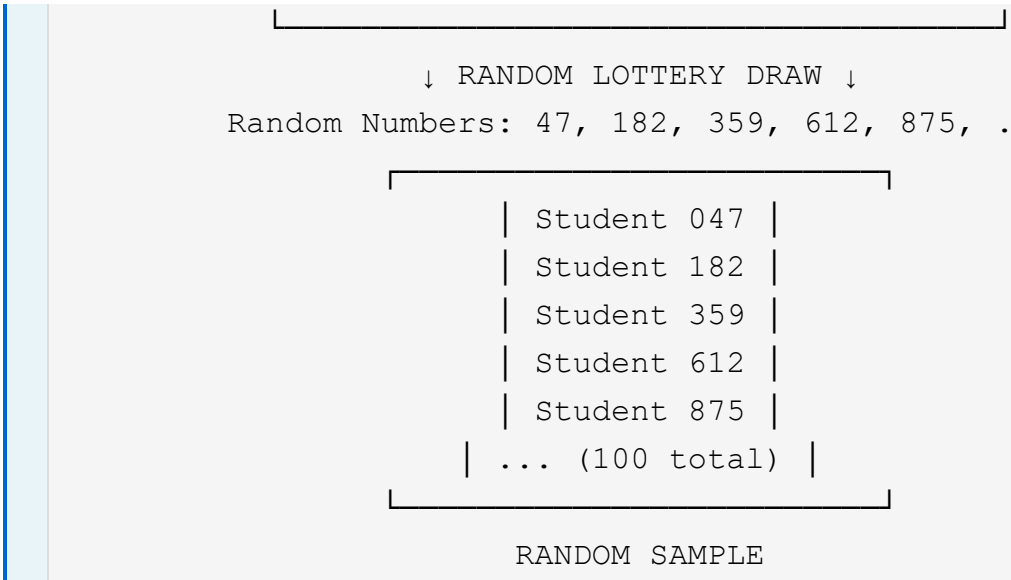
Definition: Every member of the population has an equal chance of being selected. Selection is completely random with no bias.

Method:
1. List all members of population
2. Assign each a number
3. Randomly select numbers (like lottery)

Example: *Select 100 students from 1,000*
• Number all students 1-1000
• Randomly pick 100 numbers
• Those 100 students are your sample

Random Sampling Visualization





🎯 Why is this important & what is the use?

Key Insight: Random sampling eliminates bias. Each person is equally likely to be selected, so the sample fairly represents the population.

What it tells about data: Results from random samples are statistically valid and can be generalized to the population.

How it helps in decision-making:

Application	Method	Benefit
Market Research	**Random sample of customers**	**Gives unbiased view of preferences**
Quality Control	**Random selection of products**	**Tests if entire batch meets standards**
Medical Research	**Random assignment to treatment/control**	**Ensures fair comparison**

```
import numpy as np
import pandas as pd
```

```
# Create a population dataset
np.random.seed(42)
population_size = 10000
population_data = pd.DataFrame({
    'id': range(1, population_size + 1),
    'age': np.random.randint(18, 70, population_size),
    'income': np.random.normal(50000, 15000, population_size),
    'city': np.random.choice(['Delhi', 'Mumbai', 'Chennai',
'Kolkata', 'Bangalore'], population_size)
})

print("=== POPULATION CHARACTERISTICS ===")
print(f"Total Population: {population_size:,}")
print(f"Age Range: {population_data['age'].min()} to
{population_data['age'].max()}")
print(f"Income Mean: ${population_data['income'].mean():,.2f}")
print(f"Cities: {population_data['city'].unique()}")

# Simple Random Sampling
sample_size = 1000
random_sample = population_data.sample(n=sample_size,
random_state=42)

print(f"\n=== RANDOM SAMPLE RESULTS (n={sample_size}) ===")
print(f"Sample Age Range: {random_sample['age'].min()} to
{random_sample['age'].max()}")
print(f"Sample Income Mean:
${random_sample['income'].mean():,.2f}")
print(f"Sample Cities: {random_sample['city'].unique()}")

# Check representativeness
print(f"\n=== REPRESENTATIVENESS CHECK ===")
print("Population City Distribution:")
pop_city_counts =
population_data['city'].value_counts(normalize=True) * 100
for city, percent in pop_city_counts.items():
    print(f"  {city}: {percent:.1f}%")

print("\nSample City Distribution:")
sample_city_counts =
```

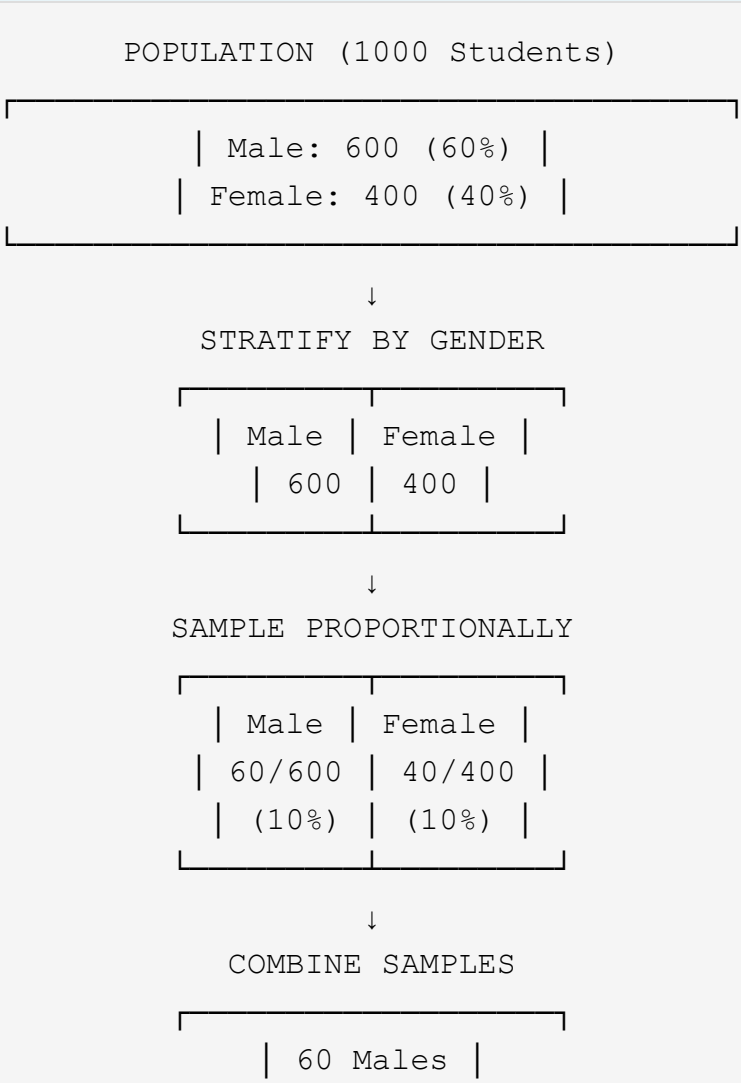
```
random_sample['city'].value_counts(normalize=True) * 100
for city, percent in sample_city_counts.items():
    print(f"  {city}: {percent:.1f}%")

# Calculate sampling error
print(f"\n=== SAMPLING ERROR ANALYSIS ===")
pop_mean = population_data['income'].mean()
sample_mean = random_sample['income'].mean()
sampling_error = abs(pop_mean - sample_mean)
print(f"Population Mean Income: ${pop_mean:,.2f}")
print(f"Sample Mean Income: ${sample_mean:,.2f}")
print(f"Sampling Error: ${sampling_error:,.2f}")
print(f"Error Percentage:
{(sampling_error/pop_mean)*100:.2f}%")
```

Stratified Sampling

Definition: Divide population into subgroups (strata) and randomly sample from each subgroup in proportion to its size in the population.

Example:
Population of 1,000 students: 600 male, 400 female
Sample size: 100 students
Stratified sample: 60 male, 40 female (same proportions as population)

Stratified Sampling Visualization



```
        | 40 Females |
        | Total: 100 |
    | (Maintains 60:40 |
        | proportion) |
    └──────────────────┘
```

🎯 Why is this important & what is the use?

Key Insight: Stratified sampling ensures all important subgroups are represented.
Useful when population has distinct groups.
What it tells about data: Results capture perspectives of all important demographic groups.
How it helps in decision-making:

Application	Strata Used	Benefit
Political Surveys	Male/female, age groups, regions	Accurate representation of all voter groups
Education Research	Classes, grades, schools	Equal representation of all student types
Marketing Research	Customer segments	Understands different preferences
Healthcare Study	Age groups, health conditions	Ensures all groups represented

```
import numpy as np
import pandas as pd
from sklearn.model_selection import StratifiedShuffleSplit

# Create population with multiple strata
np.random.seed(42)
population_size = 10000
```

```
# Create population with age groups and income levels
population_data = pd.DataFrame({
    'id': range(1, population_size + 1),
    'age_group': np.random.choice(['18-25', '26-35', '36-45',
'46-55', '56+'], population_size, p=[0.2, 0.3, 0.25, 0.15,
0.1]),
    'income_level': np.random.choice(['Low', 'Medium', 'High'],
population_size, p=[0.4, 0.5, 0.1]),
    'income': np.random.normal(50000, 20000, population_size)
})

# Add income adjustments based on groups
population_data.loc[population_data['age_group'] == '18-25',
'income'] *= 0.6
population_data.loc[population_data['age_group'] == '56+',
'income'] *= 1.3
population_data.loc[population_data['income_level'] == 'High',
'income'] *= 2.0

print("=== POPULATION STRATA DISTRIBUTION ===")
strata_counts = population_data.groupby(['age_group',
'income_level']).size().unstack()
print(strata_counts)

# Perform stratified sampling
sample_size = 1000
split = StratifiedShuffleSplit(n_splits=1,
test_size=sample_size/population_size, random_state=42)

# Create a combined strata column for sampling
population_data['strata'] = population_data['age_group'] + '_'
+ population_data['income_level']

for train_index, test_index in split.split(population_data,
population_data['strata']):
    stratified_sample = population_data.iloc[test_index]

print(f"\n=== STRATIFIED SAMPLE RESULTS (n={sample_size}) ===")
sample_strata_counts = stratified_sample.groupby(['age_group',
```

```
'income_level']).size().unstack()
print(sample_strata_counts)

# Compare proportions
print(f"\n=== PROPORTION COMPARISON ===")
print("Population Proportions:")
for (age, income), count in
population_data.groupby(['age_group',
'income_level']).size().items():
    proportion = count / population_size * 100
    print(f"  {age}-{income}: {proportion:.2f}%")

print("\nSample Proportions:")
for (age, income), count in
stratified_sample.groupby(['age_group',
'income_level']).size().items():
    proportion = count / sample_size * 100
    print(f"  {age}-{income}: {proportion:.2f}%")

# Check mean income by group
print(f"\n=== INCOME COMPARISON BY STRATA ===")
print("Population Mean Income by Age Group:")
for group in population_data['age_group'].unique():
    mean_income = population_data[population_data['age_group']
== group]['income'].mean()
    print(f"  {group}: ${mean_income:,.2f}")

print("\nSample Mean Income by Age Group:")
for group in stratified_sample['age_group'].unique():
    mean_income =
stratified_sample[stratified_sample['age_group'] == group]
['income'].mean()
    print(f"  {group}: ${mean_income:,.2f}")
```

Systematic Sampling

Definition: Select every k-th member from a numbered population list.

Method:

1. Calculate k = Population size / Sample size

2. Randomly select first member

3. Then select every k-th member after that

Example:

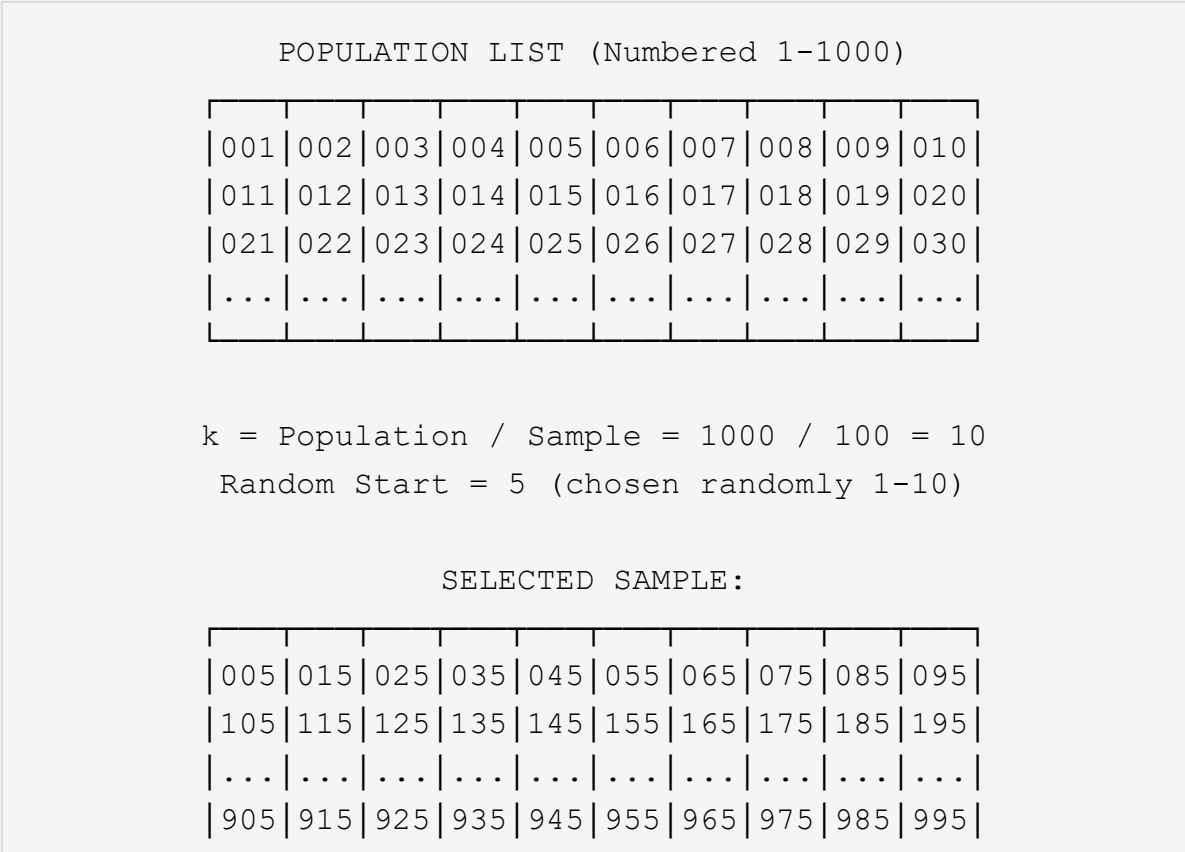
Population: 1,000 students | Desired sample: 100 students

k = 1,000 / 100 = 10

Randomly pick a number between 1-10 (say 5)

Select students 5, 15, 25, 35, 45, etc.

Systematic Sampling Visualization



```
└──┴──┴──┴──┴──┴──┴──┴──┴──┴──┘

Pattern: Start at 5, then every 10th item
Total selected: 100 items
```

🎯 Why is this important & what is the use?

Key Insight: Systematic sampling is easier to implement than random sampling while maintaining good representation.

What it tells about data: Results are generally unbiased if the population list is random.

How it helps in decision-making:

Application	Method	Benefit
Assembly Line Quality	**Every 50th product inspected**	**Regular quality checks without testing all**
Customer Service	**Every 10th customer surveyed**	**Continuous feedback collection**
Document Review	**Every 20th file audited**	**Efficient compliance checking**
Manufacturing	**Every 100th item tested**	**Consistent quality monitoring**

```
import numpy as np
import pandas as pd

# Create a production line dataset
np.random.seed(42)
production_size = 10000  # 10,000 products
products = pd.DataFrame({
    'product_id': range(1, production_size + 1),
    'production_hour': np.repeat(range(1, 101), 100),  # 100
hours of production
```

```
    'machine_id': np.random.choice(['M1', 'M2', 'M3', 'M4'],
production_size),
    'quality_score': np.random.normal(85, 10, production_size)
})

# Add some patterns
products.loc[products['machine_id'] == 'M1', 'quality_score']
+= 5
products.loc[products['machine_id'] == 'M4', 'quality_score'] -
= 3

print("=== PRODUCTION LINE DATA ===")
print(f"Total Products: {production_size:,}")
print(f"Production Hours: {products['production_hour'].min()}
to {products['production_hour'].max()}")
print(f"Machines: {products['machine_id'].unique()}")
print(f"Overall Quality Mean:
{products['quality_score'].mean():.2f}")

# Systematic Sampling
sample_size = 200
k = production_size // sample_size  # Sampling interval

# Choose random start
random_start = np.random.randint(1, k + 1)

# Select systematic sample
systematic_indices = []
current = random_start
while current <= production_size and len(systematic_indices) <
sample_size:
    systematic_indices.append(current - 1)  # Convert to 0-
based index
    current += k

systematic_sample = products.iloc[systematic_indices]

print(f"\n=== SYSTEMATIC SAMPLING ===")
print(f"Sample Size: {sample_size}")
print(f"Sampling Interval (k): {k}")
```

```
print(f"Random Start: {random_start}")
print(f"Selected Indices: {systematic_indices[:10]}...")  #
Show first 10

print(f"\n=== SAMPLE CHARACTERISTICS ===")
print(f"Sample Quality Mean:
{systematic_sample['quality_score'].mean():.2f}")
print(f"Sample Machine Distribution:")
for machine in products['machine_id'].unique():
    count = (systematic_sample['machine_id'] == machine).sum()
    proportion = count / sample_size * 100
    print(f"  {machine}: {count} items ({proportion:.1f}%)")

# Compare with simple random sample for validation
random_sample = products.sample(n=sample_size, random_state=42)

print(f"\n=== COMPARISON: SYSTEMATIC vs RANDOM SAMPLING ===")
print("Characteristic | Systematic | Random | Difference")
print("-" * 50)
print(f"Mean Quality  |
{systematic_sample['quality_score'].mean():.2f}    |
{random_sample['quality_score'].mean():.2f}  |
{abs(systematic_sample['quality_score'].mean() -
random_sample['quality_score'].mean()):.2f}")
print(f"Std Dev       |
{systematic_sample['quality_score'].std():.2f}    |
{random_sample['quality_score'].std():.2f}  |
{abs(systematic_sample['quality_score'].std() -
random_sample['quality_score'].std()):.2f}")

# Check if systematic sample covers all production hours
print(f"\n=== PRODUCTION HOUR COVERAGE ===")
hours_covered = systematic_sample['production_hour'].unique()
print(f"Hours covered in sample: {len(hours_covered)} out of
100")
print(f"Earliest hour:
{systematic_sample['production_hour'].min()}")
print(f"Latest hour:
{systematic_sample['production_hour'].max()}")
```

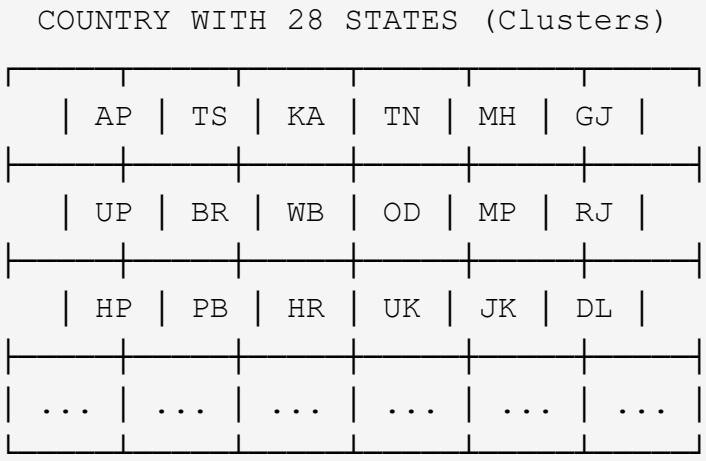
```
# Quality trend analysis
print(f"\n=== QUALITY TREND BY PRODUCTION HOUR ===")
hourly_quality = systematic_sample.groupby('production_hour')
['quality_score'].mean()
print(f"Quality range across hours: {hourly_quality.min():.2f}
to {hourly_quality.max():.2f}")
print(f"Average hourly variation: {hourly_quality.std():.2f}")
```

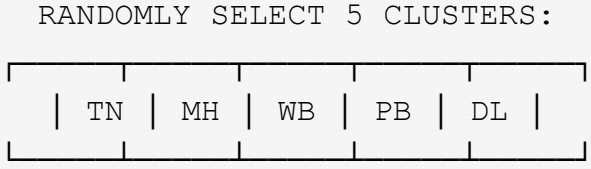
Cluster Sampling

Definition: Divide population into clusters (geographic groups, etc.), randomly select clusters, then sample all members within selected clusters.

Example:
Sample Indians: 1.4 billion people in 28 states
1. Divide into 28 clusters (states)
2. Randomly select 5 states
3. Survey all selected people from those 5 states

Cluster Sampling Visualization





SURVEY ALL PEOPLE IN SELECTED STATES:
TN: Survey all selected people in Tamil Nadu
MH: Survey all selected people in Maharashtra
WB: Survey all selected people in West Bengal
PB: Survey all selected people in Punjab

```
DL: Survey all selected people in Delhi

ADVANTAGE: Cost-effective, logistically easier
```

🎯 Why is this important & what is the use?

Key Insight: Cluster sampling is cost-effective when population is spread geographically or logically grouped.
What it tells about data: Good for large, geographically dispersed populations.
How it helps in decision-making:

Application	Clusters Used	Benefit
National Survey	Random cities	Survey all people in selected cities
School Research	Random schools	Study all students in selected schools
Hospital Study	Random hospitals	Collect data from all patients in selected hospitals
Agricultural Survey	Random farms	Measure all crops on selected farms

```
import numpy as np
import pandas as pd

# Create a national survey scenario
np.random.seed(42)

# Define 28 Indian states with different population sizes
states = [
    'Maharashtra', 'Uttar Pradesh', 'Bihar', 'West Bengal',
'Madhya Pradesh',
    'Tamil Nadu', 'Rajasthan', 'Karnataka', 'Gujarat', 'Andhra
Pradesh',
```

```
    'Odisha', 'Telangana', 'Kerala', 'Jharkhand', 'Assam',
'Punjab',
    'Chhattisgarh', 'Haryana', 'Delhi', 'Jammu & Kashmir',
'Uttarakhand',
    'Himachal Pradesh', 'Tripura', 'Meghalaya', 'Manipur',
'Nagaland',
    'Goa', 'Mizoram'
]

# Generate population data for each state
national_data = []
state_populations = {}

for i, state in enumerate(states):
    # Assign realistic population sizes (in thousands)
    population = np.random.randint(500, 50000) * 1000

    # Generate district data within state
    num_districts = np.random.randint(5, 50)

    for district in range(1, num_districts + 1):
        district_pop = population // num_districts
        avg_income = np.random.normal(30000, 10000) * (1 +
i*0.01)  # Slight state variation
        literacy_rate = np.random.normal(75, 10)

        national_data.append({
            'state': state,
            'district': f'District {district}',
            'population': district_pop,
            'avg_income': avg_income,
            'literacy_rate': literacy_rate
        })

    state_populations[state] = population

national_df = pd.DataFrame(national_data)

print("=== NATIONAL POPULATION DATA ===")
print(f"Total States: {len(states)}")
```

```
print(f"Total Districts: {len(national_df)}")
total_population = national_df['population'].sum()
print(f"Total Population: {total_population:,}")
print(f"National Avg Income:
${national_df['avg_income'].mean():,.2f}")
print(f"National Literacy Rate:
{national_df['literacy_rate'].mean():.1f}%")

# Cluster Sampling: Select random states
num_clusters = 5  # Select 5 states
selected_states = np.random.choice(states, num_clusters,
replace=False)

print(f"\n=== CLUSTER SAMPLING ===")
print(f"Selected {num_clusters} states (clusters):")
for state in selected_states:
    print(f"  • {state}")

# Get data from selected clusters
cluster_sample =
national_df[national_df['state'].isin(selected_states)]

print(f"\n=== CLUSTER SAMPLE CHARACTERISTICS ===")
print(f"Districts in sample: {len(cluster_sample)}")
print(f"Population in sample:
{cluster_sample['population'].sum():,}")
print(f"Sample Proportion:
{(cluster_sample['population'].sum()/total_population)*100:.1f}%"

# Calculate sample statistics
sample_income = cluster_sample['avg_income'].mean()
sample_literacy = cluster_sample['literacy_rate'].mean()

print(f"\n=== SAMPLE STATISTICS ===")
print(f"Sample Avg Income: ${sample_income:,.2f}")
print(f"Sample Literacy Rate: {sample_literacy:.1f}%")

# Compare with population statistics
pop_income = national_df['avg_income'].mean()
pop_literacy = national_df['literacy_rate'].mean()
```

```
print(f"\n=== COMPARISON WITH POPULATION ===")
print("Statistic | Population | Sample | Difference")
print("-" * 50)
print(f"Avg Income | ${pop_income:,.2f} | ${sample_income:,.2f}
| ${abs(pop_income - sample_income):,.2f}")
print(f"Literacy   | {pop_literacy:.1f}% |
{sample_literacy:.1f}% | {abs(pop_literacy -
sample_literacy):.1f}%")

# Cost analysis
print(f"\n=== COST EFFECTIVENESS ANALYSIS ===")
print("Method | Districts | Travel Cost | Time Required")
print("-" * 50)
print(f"Complete Survey | {len(national_df)} | High | Months")
print(f"Cluster Sampling | {len(cluster_sample)} | Low |
Weeks")

# Calculate design effect (DEFF) for cluster sampling
print(f"\n=== DESIGN EFFECT ANALYSIS ===")
# Within-cluster variance
within_var = cluster_sample.groupby('state')
['avg_income'].var().mean()
# Between-cluster variance
between_var = cluster_sample.groupby('state')
['avg_income'].mean().var()
# Design effect
deff = 1 + (between_var/within_var) *
(len(cluster_sample)/num_clusters - 1)
print(f"Within-cluster variance: {within_var:,.2f}")
print(f"Between-cluster variance: {between_var:,.2f}")
print(f"Design Effect (DEFF): {deff:.2f}")
print(f"Effective Sample Size: {len(cluster_sample)/deff:.0f}
districts")
```

4.3 Sampling Distribution

📊 Understanding Sampling Distribution

Definition: A sampling distribution is the distribution of a statistic (like mean or proportion) calculated from many different samples of the same population.

Simple Explanation:
If you take 100 different random samples of 1,000 people each and calculate the mean income in each sample, those 100 means will have their own distribution.

📊 Sampling Distribution Visualization

POPULATION DISTRIBUTION

(Original data pattern)

TAKE MANY SAMPLES:
Sample 1: Mean = 48,500
Sample 2: Mean = 49,200
Sample 3: Mean = 47,800
Sample 4: Mean = 50,100
... (100 samples total)

SAMPLING DISTRIBUTION OF MEANS:

(Bell-shaped, centered around population mean)

Key Properties:
• Centered at population parameter

• Shape is normal (bell curve)

• Spread decreases with larger samples

🎯 Why is this important & what is the use?

Key Insight: Sampling distribution helps you understand how sample statistics vary. It's the foundation for confidence intervals and hypothesis testing.

What it tells about data:

- **How much different samples vary**
- **What the expected value should be**
- **How confident you can be in sample estimates**

How it helps in decision-making:

Application	Sample Statistic	Sampling Distribution Tells Us
Election Polls	**50% for candidate A**	**Likely range is 45-55%**
Product Testing	**90% satisfaction**	**If this result is reliable**
Quality Control	**Defect rate 2%**	**If process is in control**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats

# Create population data
np.random.seed(42)
population_size = 1000000
population = np.random.exponential(scale=50000,
size=population_size)  # Right-skewed distribution

print("=== POPULATION CHARACTERISTICS ===")
print(f"Population Size: {population_size:,}")
```

```
print(f"Population Mean: ${population.mean():,.2f}")
print(f"Population Std Dev: ${population.std():,.2f}")
print(f"Population Skewness: {stats.skew(population):.3f}")

# Simulate sampling distribution
num_samples = 1000
sample_size = 100
sample_means = []

print(f"\n=== GENERATING SAMPLING DISTRIBUTION ===")
print(f"Number of samples: {num_samples}")
print(f"Sample size: {sample_size}")

for i in range(num_samples):
    sample = np.random.choice(population, size=sample_size,
replace=False)
    sample_means.append(sample.mean())

sample_means = np.array(sample_means)

print(f"\n=== SAMPLING DISTRIBUTION STATISTICS ===")
print(f"Mean of sample means: ${sample_means.mean():,.2f}")
print(f"Std Dev of sample means: ${sample_means.std():,.2f}")
print(f"Standard Error:
${population.std()/np.sqrt(sample_size):,.2f}")

# Theoretical properties
print(f"\n=== THEORETICAL PROPERTIES ===")
print(f"1. Mean should equal population mean:")
print(f"   Population mean: ${population.mean():,.2f}")
print(f"   Mean of sample means: ${sample_means.mean():,.2f}")
print(f"   Difference: ${abs(population.mean() -
sample_means.mean()):,.2f}")

print(f"\n2. Standard Error = Population SD / √n")
print(f"   Theoretical:
${population.std()/np.sqrt(sample_size):,.2f}")
print(f"   Observed: ${sample_means.std():,.2f}")

print(f"\n3. Shape should be approximately normal")
```

```
print(f"   Skewness of sampling distribution:
{stats.skew(sample_means):.3f}")
print(f"   (Should be close to 0 for normal distribution)")

# Confidence interval analysis
print(f"\n=== CONFIDENCE INTERVAL ANALYSIS ===")
confidence_level = 0.95
z_score = stats.norm.ppf((1 + confidence_level) / 2)

# For a single sample
single_sample = np.random.choice(population, size=sample_size,
replace=False)
sample_mean = single_sample.mean()
sample_std = single_sample.std()
margin_error = z_score * (sample_std / np.sqrt(sample_size))
ci_lower = sample_mean - margin_error
ci_upper = sample_mean + margin_error

print(f"Single sample results:")
print(f"  Sample mean: ${sample_mean:,.2f}")
print(f"  Sample std: ${sample_std:,.2f}")
print(f"  Margin of error: ${margin_error:,.2f}")
print(f"  95% CI: [${ci_lower:,.2f}, ${ci_upper:,.2f}]")
print(f"  Population mean (${population.mean():,.2f}) in CI:
{ci_lower <= population.mean() <= ci_upper}")

# Check how many CIs contain population mean
print(f"\n=== COVERAGE PROBABILITY ===")
# Generate confidence intervals for all samples
cis_contain_pop = 0
for mean in sample_means[:100]:  # Check first 100 for speed
    # In practice, we'd use each sample's std, but for
simplicity use population std
    margin = z_score * (population.std() /
np.sqrt(sample_size))
    ci_low = mean - margin
    ci_high = mean + margin
    if ci_low <= population.mean() <= ci_high:
        cis_contain_pop += 1
```

```
coverage_rate = cis_contain_pop / 100
print(f"Percentage of 95% CIs containing population mean:
{coverage_rate*100:.1f}%")
print(f"Expected: 95%")

# Effect of sample size
print(f"\n=== EFFECT OF SAMPLE SIZE ===")
sample_sizes = [30, 100, 500, 1000]
for size in sample_sizes:
    sample_means_size = []
    for _ in range(100):
        sample = np.random.choice(population, size=size,
replace=False)
        sample_means_size.append(sample.mean())

    std_error = np.array(sample_means_size).std()
    print(f"Sample size {size:4d}: Std Error =
${std_error:,.2f}")

print(f"\nRelationship: As sample size increases, standard
error decreases")
print(f"Standard Error ∝ 1/√n")
```

4.4 Central Limit Theorem (CLT)

🌟 The Magic of CLT

Definition: The Central Limit Theorem states that if you take enough random samples from any population and calculate their means, those sample means will be approximately normally distributed (bell curve shaped), regardless of the original population's shape.

Simple Explanation:
No matter how weird your original data looks, if you take many samples and plot their averages, you'll get a nice bell curve.

Key Requirements:
1. Random sampling
2. Sample size ≥ 30 (rule of thumb)
3. Independent samples

🌟 Central Limit Theorem Visualization

ANY POPULATION DISTRIBUTION:
Original: ____▂▄▆▇▅▃▂____ (Skewed Right)
Original: ▇▇▅▃▂____▂▃▅▇▇ (Skewed Left)
Original: ___▂▄▇▇▅▂____ (Bimodal)

TAKE MANY SAMPLES (n ≥ 30):
Sample 1 Mean: 48.5
Sample 2 Mean: 49.2

Sample 3 Mean: 50.1
... (1000 samples)

SAMPLING DISTRIBUTION OF MEANS:
Always: _▂▃▅▇▇▇▇▇▅▃▂_
(Normal/Bell-shaped)

The Magic: Means become normal regardless!
Center = Population Mean
Spread = Population SD / √n

🎯 Why is this important & what is the use?

Key Insight: CLT is the reason most statistical tests work. It allows us to use normal distribution for inference even when original data isn't normal.

What it tells about data:

- **Sample means cluster around the population mean**
- **Sample means follow a predictable distribution**
- **Larger samples produce more consistent estimates**

How it helps in decision-making:

Application	Individual Data	Sample Means (via CLT)	Benefit
Customer Spending	**Chaotic, unpredictable**	**Predictable average spending**	**Revenue forecasting**
Quality Control	**Wild individual variations**	**Consistent average measurements**	**Process control**
Opinion Polling	**Individual voters vary**	**Reliable average opinion polls**	**Election prediction**
Risk Management	**Individual stock volatility**	**Predictable portfolio returns**	**Investment planning**

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns

# Set random seed for reproducibility
np.random.seed(42)

print("=== DEMONSTRATING CENTRAL LIMIT THEOREM ===")
print("We'll show CLT works for different population
distributions")

# Define different population distributions
distributions = {
    "Exponential (Right-skewed)":
np.random.exponential(scale=50, size=100000),
    "Uniform (Flat)": np.random.uniform(0, 100, 100000),
    "Bimodal (Two peaks)": np.concatenate([
        np.random.normal(30, 10, 50000),
        np.random.normal(70, 10, 50000)
    ]),
    "Log-normal (Heavy right tail)":
np.random.lognormal(mean=3, sigma=0.5, size=100000)
}

# Parameters for sampling
sample_sizes = [5, 30, 100]  # Different sample sizes
num_samples = 1000  # Number of samples to take

results = {}

for dist_name, population in distributions.items():
    print(f"\n=== {dist_name.upper()} ===")
    print(f"Population mean: {population.mean():.2f}")
    print(f"Population std: {population.std():.2f}")
    print(f"Population skew: {stats.skew(population):.3f}")

    dist_results = {}
```

```
    for n in sample_sizes:
        sample_means = []

        # Take many samples of size n
        for _ in range(num_samples):
            sample = np.random.choice(population, size=n,
replace=False)
            sample_means.append(sample.mean())

        sample_means = np.array(sample_means)

        # Store results
        dist_results[n] = {
            'means': sample_means,
            'mean_of_means': sample_means.mean(),
            'std_of_means': sample_means.std(),
            'skew_of_means': stats.skew(sample_means),
            'theoretical_se': population.std() / np.sqrt(n)
        }

        print(f"\n  Sample size n = {n}:")
        print(f"    Mean of sample means:
{sample_means.mean():.2f}")
        print(f"    Std of sample means:
{sample_means.std():.2f}")
        print(f"    Theoretical SE:
{population.std()/np.sqrt(n):.2f}")
        print(f"    Skewness: {stats.skew(sample_means):.3f}")

    results[dist_name] = dist_results

# Normality tests
print(f"\n=== NORMALITY TESTS ===")
for dist_name in distributions.keys():
    print(f"\n{dist_name}:")
    for n in sample_sizes:
        sample_means = results[dist_name][n]['means']

        # Shapiro-Wilk test for normality
        if len(sample_means) <= 5000:  # Shapiro limit
```

```
            shapiro_stat, shapiro_p =
stats.shapiro(sample_means)
            normal = shapiro_p > 0.05
        else:
            # Use Kolmogorov-Smirnov for large samples
            ks_stat, ks_p = stats.kstest(sample_means, 'norm',
                                         args=
(sample_means.mean(), sample_means.std()))
            normal = ks_p > 0.05

        print(f"  n={n:3d}: {'Normal' if normal else 'Not
Normal'} (p-value: {shapiro_p if 'shapiro' in locals() else
ks_p:.4f})")

# Practical example: Customer spending
print(f"\n=== PRACTICAL EXAMPLE: CUSTOMER SPENDING ===")
print("Original customer spending data (right-skewed):")
customer_spending = np.random.exponential(scale=100,
size=10000)
print(f"Individual spending mean:
${customer_spending.mean():.2f}")
print(f"Individual spending std:
${customer_spending.std():.2f}")
print(f"Individual spending skew:
{stats.skew(customer_spending):.3f}")

# Take samples of 30 customers each
daily_samples = 365  # One year
customers_per_day = 30

daily_means = []
for day in range(daily_samples):
    daily_customers = np.random.choice(customer_spending,
size=customers_per_day, replace=False)
    daily_means.append(daily_customers.mean())

daily_means = np.array(daily_means)

print(f"\nDaily average spending (30 customers per day):")
print(f"Mean of daily means: ${daily_means.mean():.2f}")
```

```
print(f"Std of daily means: ${daily_means.std():.2f}")
print(f"Skew of daily means: {stats.skew(daily_means):.3f}")

# Calculate confidence interval for annual average
z_score = 1.96  # 95% confidence
margin_error = z_score * (daily_means.std() /
np.sqrt(len(daily_means)))
ci_lower = daily_means.mean() - margin_error
ci_upper = daily_means.mean() + margin_error

print(f"\n95% Confidence Interval for true average spending:")
print(f"${ci_lower:.2f} to ${ci_upper:.2f}")

# Predict next month's revenue
avg_daily_customers = 50
month_days = 30
predicted_monthly_revenue = daily_means.mean() *
avg_daily_customers * month_days
revenue_margin = margin_error * avg_daily_customers *
month_days

print(f"\nRevenue Prediction for next month:")
print(f"Expected: ${predicted_monthly_revenue:,.2f}")
print(f"Range (95% CI): ${predicted_monthly_revenue -
revenue_margin:,.2f} to ${predicted_monthly_revenue +
revenue_margin:,.2f}")

# Business decision based on CLT
print(f"\n=== BUSINESS DECISION SUPPORT ===")
print("Based on CLT, we can:")
print("1. Trust that daily averages are normally distributed")
print("2. Make revenue predictions with known confidence
intervals")
print("3. Set appropriate inventory levels")
print("4. Plan staffing based on expected customer volume")
print("5. Make data-driven decisions despite individual
variability")
```

📋 Summary & Key Takeaways

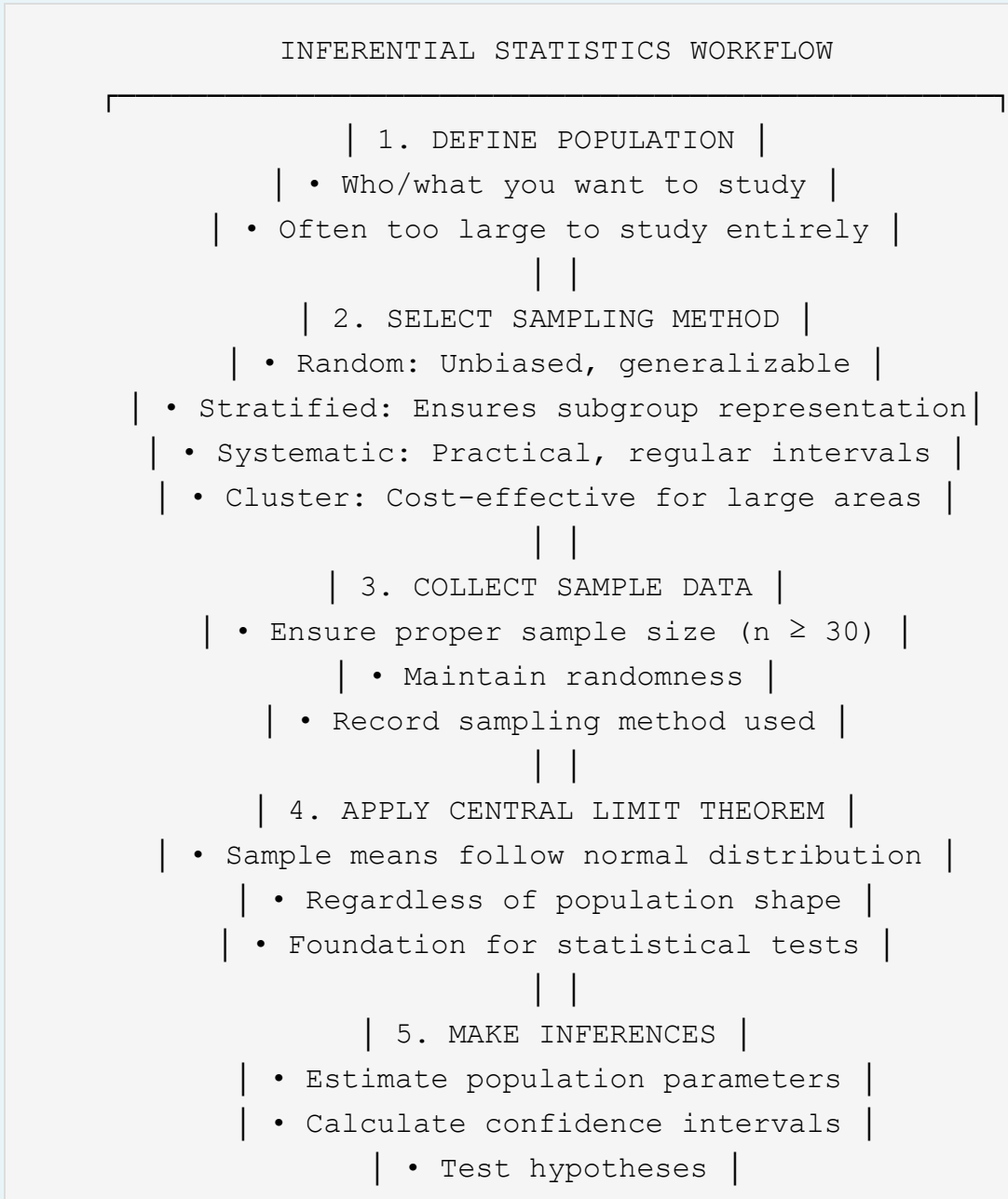
Concept	Key Idea	When to Use	Python Functions
Population vs Sample	Study sample, infer about population	When studying entire population is impractical	np.random.choice() df.sample()
Random Sampling	Equal chance for all members	When unbiased representation is critical	np.random.randint() random.sample()
Stratified Sampling	Sample proportionally from subgroups	When population has important subgroups	StratifiedShuffleSplit groupby().sample()
Systematic Sampling	Select every k-th member	When population list exists and is random	range(start, N, k) iloc[::k]
Cluster Sampling	Randomly select groups, study all within	When population is geographically dispersed	groupby().sample() np.random.choice(groups)
Sampling Distribution	Distribution of sample statistics	When assessing reliability of estimates	np.mean() in loop stats.sem()
Central Limit Theorem	Sample means become normal	When using statistical inference methods	stats.norm.fit() z-score calculations

🎯 Quick Decision Guide for Sampling:

1. **Start with your goal - What population parameter do you want to estimate?**

2. **Check population structure - Are there important subgroups? → Use Stratified**
3. **Consider logistics - Is population spread out? → Use Cluster**
4. **Ensure randomness - Always use random selection methods**
5. **Calculate sample size - Use power analysis for proper n**
6. **Apply CLT - With n ≥ 30, you can assume normality for inference**
7. **Report with error - Always include confidence intervals**

📊 The Inferential Statistics Pipeline



```
                    | • Make predictions |
                                 | |
                     | 6. REPORT RESULTS |
                  | • Include sampling method |
                   | • Report margin of error |
                   | • State confidence level |
                  | • Acknowledge limitations |
    └──────────────────────────────────────────────────┘
```

```
import numpy as np
from scipy import stats

print("=== QUICK INFERENCE CALCULATOR ===")

def quick_inference(population_mean=None, population_std=None,
                    sample_mean=None, sample_std=None,
sample_size=None,
                    confidence_level=0.95):
    """
    Quick function to demonstrate inferential calculations
    """
    if confidence_level == 0.95:
        z = 1.96
    elif confidence_level == 0.99:
        z = 2.576
    elif confidence_level == 0.90:
        z = 1.645
    else:
        z = stats.norm.ppf((1 + confidence_level) / 2)

    results = {}

    if sample_mean and sample_std and sample_size:
        # Calculate standard error
        se = sample_std / np.sqrt(sample_size)
        margin_error = z * se
```

```
        # Confidence interval
        ci_lower = sample_mean - margin_error
        ci_upper = sample_mean + margin_error

        results['standard_error'] = se
        results['margin_of_error'] = margin_error
        results['confidence_interval'] = (ci_lower, ci_upper)

        # Minimum sample size for desired margin
        if population_std:
            desired_margin = margin_error * 0.5  # Half the
current margin
            min_n = (z * population_std / desired_margin) ** 2
            results['min_sample_for_half_margin'] =
int(np.ceil(min_n))

    return results

# Example usage
print("Example: Product satisfaction survey")
print("Sample: 400 customers")
print("Sample satisfaction: 85% (0.85)")
print("Sample std: 0.35")

results = quick_inference(
    sample_mean=0.85,
    sample_std=0.35,
    sample_size=400,
    confidence_level=0.95
)

print(f"\nResults:")
print(f"Standard Error: {results['standard_error']:.4f}")
print(f"Margin of Error: ±{results['margin_of_error']:.4f}")
print(f"95% CI: [{results['confidence_interval'][0]:.3f},
{results['confidence_interval'][1]:.3f}]")
print(f"Interpretation: True satisfaction is between
{(results['confidence_interval'][0]*100):.1f}% and
{(results['confidence_interval'][1]*100):.1f}%")
```

```
# Sample size calculation
print(f"\n=== SAMPLE SIZE CALCULATION ===")
def calculate_sample_size(population_std, margin_error,
confidence_level=0.95):
    z = stats.norm.ppf((1 + confidence_level) / 2)
    n = (z * population_std / margin_error) ** 2
    return int(np.ceil(n))

# Example: Election poll
print("Election poll requirements:")
print("Desired margin of error: ±3% (0.03)")
print("Expected std (worst case): 0.5")
print("Confidence level: 95%")

required_n = calculate_sample_size(
    population_std=0.5,
    margin_error=0.03,
    confidence_level=0.95
)

print(f"Required sample size: {required_n:,} voters")
print(f"Common poll sizes: 1,000-2,000 (practical compromise)")
```

Section 4 of Statistics for Data Science - Inferential Statistics
Complete Guide to Making Conclusions from Samples

Data Science with Vamsi

Next: Section 5 - Probability Theory