STAT 328 (Statistical Packages) (Python)

Course Overview This course introduces Python as a statistical tool: data entry, manipulation, matrix operations, statistical measures, hypothesis testing, probability distributions, basic regression, ANOVA, and practical workflows using `numpy`, `pandas`, `scipy`, and `statsmodels`.

Course Objectives

- Teach students to use Python for statistical analysis and reproducible research.
- Cover data entry, transformation, visualization basics (brief), and key statistical tests and models.
- Provide hands-on examples that students can run and adapt.

Learning Outcomes

By the end of the course, students will be able to:

1. Enter and clean datasets in Python using `pandas`.
2. Perform matrix calculations with `numpy`.
3. Compute common statistical summaries and special functions (gamma, logs, combinations).
4. Numerically integrate functions and do simulations.
5. Carry out 1-sample, 2-sample (independent), paired t-tests, correlation and simple linear regression.
6. Fit and interpret a one-way ANOVA.
7. Use probability distribution functions (pdf/pmf, cdf, ppf, rvs) from `scipy.stats`.

Python Introduction

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Python & Jupyter Installation Guide

1. Download Python

Download Python from the official website:
https://www.python.org/downloads/

---

2. Install Jupyter Notebook

Open Command Prompt (CMD) and run:

```
python -m pip install jupyter
```

Verify installation:

```
python -m jupyter --version
```

▶️ Open Jupyter Notebook

You can open Jupyter Notebook using one of the following commands:

```
python -m notebook
python -m jupyter
```

- ❖ Note: Both commands will launch Jupyter Notebook in your web browser.

Alternative way to Install Jupyter Notebook (in Lab)

```
C:\Users\Exam> py  -m pip install jupyter
```

▶️ Open Jupyter:

```
C:\Users\Exam> py  -m notebook
```

---

3. Install packages via Jupyter, CMD, or IDLE

- **Install Packages Inside Jupyter**

Run this inside a Jupyter cell:

```
import sys
!{sys.executable} -m pip install numpy pandas scipy statsmodels scikit-learn
```

Import Libraries:

```
import numpy as np
import pandas as pd
import math
from math import comb  # Python 3.8+
import scipy.stats as stats
import scipy.special as sps
import scipy.integrate as integrate
import statsmodels.api as sm
import statsmodels.formula.api as smf
from sklearn.linear_model import LinearRegression
```

-

- **Alternative way Install Packages in (CMD)**

After opening your CMD, type the command python -m pip install package_name and hit Enter to start the installation process.

```
python -m pip install numpy pandas scipy statsmodels scikit-learn
```

- **Alternative way Install Packages in IDLE (Shell)**

```
import os
os.system("python -m pip install numpy")
```

Verify installation:

```
import numpy
print(numpy.__version__)
```

Python Libraries and Their Uses:

NumPy: A fundamental library for numerical computing in Python.
It provides support for arrays, matrices, and mathematical operations.

Pandas: A powerful library for data analysis and manipulation.
It is used to work with structured data such as tables (DataFrames).

SciPy: A scientific computing library built on NumPy.
It provides advanced functions for statistics, optimization, integration, and more.

Statsmodels: A library for statistical modeling and hypothesis testing. It is used for regression analysis, time series analysis, and statistical tests**.**

Scikit-learn: A machine learning library.
It provides tools for classification, regression, clustering, and data preprocessing.

Saving and Exporting Files in Jupyter Notebook

After completing your work in a Jupyter Notebook, you can save or export your file in various formats depending on your specific requirements:

• **Saving the Source File**: Always save your work in the original format (.ipynb) first; this ensures you can easily edit or continue your work later.

• **Exporting as PDF:** You can export the notebook directly to PDF for submission by selecting "**File → Save and Export Notebook As → PDF**." (Note: This option may not always be available by default and may require additional tools like nbconvert or Pandoc).

• **Exporting as HTML:** As a reliable alternative, save the notebook as an HTML file, which can then be converted to PDF using the "Print" function in a web browser.
File → Save and Export Notebook As → HTML

• **Exporting as a Python Script:** If you only need the code without documentation or output cells, you can export the notebook as a Python script (.py).
File → Save and Export Notebook As → Executable Script

1) How to enter data in Python

a) Manually (lists, arrays, DataFrame)

```
# Lists
import numpy as np

ages = [21, 22, 20, 23]
scores = [77, 85, 90, 69]

# convert to numpy array
ages_arr = np.array(ages)

# Create a pandas DataFrame
df = pd.DataFrame({'age': ages, 'score': scores})
print(df)
```

b) Creating synthetic data (useful for examples and testing)

```
# create 100 observations from a normal distribution
data = np.random.normal(loc=50, scale=10, size=100)
df = pd.DataFrame({'x': data})
print(df)
```

2) How to multiply matrices in Python

```
#Use numpy for matrix multiplication. Remember: @
operator or np.dot or np.matmul.
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# elementwise multiply
elementwise = A * B
# matrix multiply
matmul1 = A @ B
matmul2 = np.dot(A, B)
matmu13 = np.matmul(A,B)

print('Elementwise:\n', elementwise)
print('Matrix product (A @ B):\n', matmul1)
print('Matrix product (np.dot(A, B)):\n', matmul2)
print('Matrix product (np.matmul(A,B)):\n', matmu13)
```

Matrix inverse and solving linear systems

```
# inverse
A_inv = np.linalg.inv(A)
# solve Ax = b
b = np.array([1, 2])
x = np.linalg.solve(A, b)
```

3) Statistical measures + Gamma function + logs + combinations

a) Common statistical measures

```
arr = np.array([2, 4, 7, 3, 9])
mean = arr.mean()
median = np.median(arr)
var = arr.var(ddof=1) # sample variance
std = arr.std(ddof=1) # sample std
minimum = arr.min()
maximum = arr.max()
quantiles = np.quantile(arr, [0.25, 0.5, 0.75])

print('mean', mean)
print('median', median)
print('sample variance', var)
print('sample std', std)
print('minimum:',minimum,'\t maximum:',maximum)
print('quantiles', quantiles)
```

Note:
ddof in NumPy stands for Delta Degrees of Freedom, It controls what value you divide by when calculating variance

- **ddof = 0**
  → Population variance, divides by (n)
- **ddof = 1**
  → Sample variance, divides by (n – 1)

b) Gamma function and log/ln

```
# gamma function (Gamma(x)) using scipy.special
from scipy.special import gamma, gammaln

# Example: Gamma(5) = 4! = 24
print('Gamma(5) =', gamma(5))

# Log gamma (natural log of Gamma) - use gammaln for
numerical stability
print('log Gamma(5) =', gammaln(5))
```

```
# Natural log and log base 10
x = 10.0
print('natural log ln(x):', math.log(x)) # ln
print('log base 10:', math.log10(x))

# If you need log with base e on numpy arrays
print('numpy log', np.log(np.array([1, np.e, np.e**2])))
```

c) Combinations: 5 choose 2

```
# Using math.comb (Python 3.8+)
print('5 choose 2 =', comb(5, 2))

# Or using scipy.special.comb
from scipy.special import comb as spcomb
print('5 choose 2 (scipy) =', spcomb(5, 2, exact=True))
```

4) Identify a function and numerical integration

a) Define a function and integrate using `scipy.integrate.quad`

We wish to define $f(x) = x^2 + 2x + 1$

```
# Define the function
def f(x):
    return x**2 + x*2 + 1

# Integrate from 0 to 1
Int = integrate.quad(f, 0, 1)
print(Int)
```

5) How to use loop functions in Python

a) for loop

```
nums = [1, 2, 3, 4]
for i in nums:
    print(i**2)
```

#b) while loop

```
i = 0
while i < 5:
    print('i =', i)
    i += 1
```

6) Probability distributions in Python (p, q, d, r explanation)

We use `scipy.stats` objects. The typical shorthand mapping from R to scipy:

- **d**: density / pmf → `.pdf(x)` for continuous, `.pmf(k)` for discrete.
- **p**: cumulative distribution function (CDF) → `.cdf(x)`.
- **q**: quantile function (inverse CDF / percent point function) → `.ppf(q)`.
- **r**: random generation → `.rvs(size=n)`.

Examples with Normal and Binomial

```
# Normal distribution example
mu, sigma = 0, 1
x = 1.5
# d (pdf)
pdf_val = stats.norm.pdf(x, loc=mu, scale=sigma)
# p (cdf)
cdf_val = stats.norm.cdf(x, loc=mu, scale=sigma)
# q (quantile for probability 0.975)
quantile = stats.norm.ppf(0.975, loc=mu, scale=sigma)
# r (simulate 10 random values)
samples = stats.norm.rvs(loc=mu, scale=sigma, size=10)

print('pdf', pdf_val)
print('cdf', cdf_val)
print('0.975 quantile', quantile)
print('samples', samples)

# Binomial (discrete)
n, p = 10, 0.3
k = 3
pmf = stats.binom.pmf(k, n=n, p=p) # d
print('pmf:',pmf)
cdf = stats.binom.cdf(k, n=n, p=p) # p
print('cdf:',cdf)
quant = stats.binom.ppf(0.95, n=n, p=p) # q
print('0.95 quantile:',quant)
samps = stats.binom.rvs(n=n, p=p, size=10) # r
print('samples: \n',samps)
```

Interpretation notes

- `pdf` (or `pmf`) gives the density/probability at a point.
- `cdf` yields $P(X \leq x)$.
- `ppf` is the inverse: given probability α, returns smallest x with CDF $\geq \alpha$.
- `rvs` produces random draws from that distribution.

7) one-sample t-test in Python (all alternatives)

Use scipy.stats.ttest_1samp for two-sided test by default. For one-sided, adjust the p-value accordingly. A one-sample t-test checks whether the mean of a sample is significantly different from a known or hypothesized population mean μ_0. Suppose we measure exam scores of 10 students, and we want to test whether their average score is different from 70.

```
import numpy as np
from scipy import stats

# Sample data
scores = np.array([72, 68, 75, 70, 74, 69, 71, 73, 76,
70])

# Hypothesized population mean
mu0 = 70
#Two sided
t_stat, p_value = stats.ttest_1samp(scores, mu0,
alternative='two-sided')

print("t-statistic:", t_stat)
print("p-value:", p_value)
# Right Tailed Greater
t_stat, p_value = stats.ttest_1samp(scores, mu0,
alternative='greater')

print("t-statistic:", t_stat)
print("p-value:", p_value)
# Left Tailed Less
t_stat, p_value = stats.ttest_1samp(scores, mu0,
alternative='less')

print("t-statistic:", t_stat)
print("p-value:", p_value)
```

8) two-sample t-test in Python (independent samples)

Use scipy.stats.ttest_ind. You can choose equal_var=True/False (Welch's test when False). A two-sample t-test compares the means of two independent groups to determine whether they are statistically different. Suppose we want to compare exam scores of students taught using Method A and Method B.

```
import numpy as np
from scipy import stats

# Two independent samples
method_A = np.array([78, 75, 80, 77, 79, 76, 81, 74])
method_B = np.array([72, 70, 74, 71, 73, 69, 75, 68])
```

```
# Two Sided μ_A ≠ μ_B
t_stat, p_value =
stats.ttest_ind(method_A,method_B,alternative='two-
sided')
print("t-statistic:", t_stat)
print("p-value:", p_value)

# Greater μ_A > μ_B
t_stat, p_value =
stats.ttest_ind(method_A,method_B,alternative='greater')
print("t-statistic:", t_stat)
print("p-value:", p_value)

# Less μ_A < μ_B
t_stat, p_value =
stats.ttest_ind(method_A,method_B,alternative='less')
print("t-statistic:", t_stat)
print("p-value:", p_value)
```

9) Paired sample t-test in Python

Use scipy.stats.ttest_rel for paired samples (dependent samples). A paired t-test compares the means of two related measurements taken from the same subjects. A clinic measures patients' blood pressure before and after a new medication. For example, we want to know whether the medication changed blood pressure.

```
import numpy as np
from scipy import stats

# Blood pressure readings (mmHg)
before = np.array([150, 145, 160, 155, 148, 152, 158,
149])
after  = np.array([140, 138, 150, 147, 142, 145, 148,
141])

t_stat, p_value =
stats.ttest_rel(before,after,alternative='two-sided')
print("t-statistic:", t_stat)
print("p-value:", p_value)
```

10) Correlation in Python

Use `scipy.stats.pearsonr` for Pearson correlation and `spearmanr` for Spearman rank.

```
x = np.array([1,2,3,4,5])
y = np.array([2,4,5,4,5])
```

```
pearson_r, pearson_p = stats.pearsonr(x, y)
spearman_r, spearman_p = stats.spearmanr(x, y)

print('Pearson r, p', pearson_r, pearson_p)
print('Spearman rho, p', spearman_r, spearman_p)
```

11) Simple regression (OLS) in Python

Use `statsmodels.api.OLS` or formula interface `statsmodels.formula.api.ols`.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Independent variable (hours studied)
X = np.array([2, 3, 5, 7, 9, 10, 12]).reshape(-1, 1)
# Dependent variable (exam scores)
y = np.array([50, 55, 65, 70, 75, 80, 85])

# Create the model
model = LinearRegression()
# Fit the model
model.fit(X, y)
# Get slope and intercept
print("Intercept (β0):", model.intercept_)
print("Slope (β1):", model.coef_[0])
```

12) F test

The F-test is used to compare variances of two populations. We have test scores from two classes and want to see if the variability of scores differs.

var1=3.6999 , var2= 5.7 >> F=0.65 ??

```
import numpy as np
from scipy import stats

# Sample data
class1 = np.array([85, 88, 90, 87, 86])
class2 = np.array([78, 75, 80, 74, 77])

# Sample sizes and degrees of freedom
n1, n2 = len(class1), len(class2)
dof1, dof2 = n1 - 1, n2 - 1

# Sample variances
var1 = np.var(class1, ddof=1)
var2 = np.var(class2, ddof=1)
```

```
# F-statistic (ratio of variances)
F = var1 / var2
print("F-statistic:", F)

# Two-sided p-value
p_two_sided = 2 * min(stats.f.cdf(F, dof1, dof2), 1 -
stats.f.cdf(F, dof1, dof2))
print("Two-sided p-value:", p_two_sided)

# Right-tailed p-value (σ1² > σ2²)
p_right = 1 - stats.f.cdf(F, dof1, dof2)
print("Right-tailed p-value (σ1² > σ2²):", p_right)

# Left-tailed p-value (σ1² < σ2²)
p_left = stats.f.cdf(F, dof1, dof2)
print("Left-tailed p-value (σ1² < σ2²):", p_left)
```

13) One-way ANOVA (single factor) in Python

13.1. Introduction

- **ANOVA (Analysis of Variance)** compares the means of **three or more groups**.
- **Null hypothesis (H_0):** all group means are equal.
- **Alternative hypothesis (H_1):** at least one group mean is different.
- In Python, we typically use the `statsmodels` package for ANOVA.

Suppose we have exam scores of students taught using three different teaching methods. We want to see if the method affects scores.

```
import numpy as np
from scipy import stats

# Scores for 3 groups
group1 = np.array([85, 88, 90, 87, 86])   # Method A
group2 = np.array([78, 75, 80, 74, 77])   # Method B
group3 = np.array([92, 95, 91, 93, 94])   # Method C

f_stat, p_value = stats.f_oneway(group1, group2, group3)

print("F-statistic:", f_stat)
print("p-value:", p_value)
```