



# GEDHAS: Complete Line-by-Line Explanation & Mathematical Theory

## Table of Contents

1. [Executive Overview](#)
  2. [Section 1: Imports Explained](#)
  3. [Section 2: Database Schema](#)
  4. [Section 3: Mathematical Formulas](#)
  5. [Section 4: Data Generation](#)
  6. [Section 5: SQL Queries](#)
  7. [Section 6: Visualizations](#)
  8. [Section 7: Interactive Dashboard](#)
- 

## Executive Overview

### What This Program Does

This is a **database application** that simulates how astronomers catalog and analyze exoplanets (planets outside our solar system). It:

- Creates a relational database with 6 interconnected tables
- Generates realistic mock data for 150+ star systems and 400+ planets
- Calculates habitability scores using real astronomical formulas
- Provides interactive visualizations and search capabilities

### Key Concepts You'll Learn

- **Database Design:** Primary keys, foreign keys, relationships
  - **SQL:** SELECT, JOIN, GROUP BY, window functions
  - **Astronomy:** How we find habitable planets
  - **Python:** Data manipulation, visualization, interactive widgets
-

## Section 1: Imports Explained

Let's break down EVERY import and what it does:

```
python  
  
import sqlite3
```

**What it is:** SQLite is a lightweight database engine built into Python

**What we use it for:** Creating tables, storing data, running SQL queries

**Real-world analogy:** Like Excel but much more powerful - can handle millions of rows and complex relationships

```
python  
  
import random
```

**What it is:** Python's random number generator

**What we use it for:** Generating realistic but fake data (planet masses, temperatures, etc.)

**Example:** `random.uniform(0.5, 1.5)` gives a random decimal between 0.5 and 1.5

```
python  
  
import math
```

**What it is:** Mathematical functions (sqrt, log, sin, cos, etc.)

**What we use it for:** Astronomical calculations (orbital periods, temperature)

**Example:** `math.sqrt(4)` returns 2.0

```
python  
  
from datetime import datetime, timedelta
```

**What it is:** Tools for working with dates and times

**What we use it for:** Creating discovery dates for planets

**Example:** `datetime.now() - timedelta(days=365)` = one year ago

```
python  
  
import numpy as np
```

**What it is:** Numerical Python - the foundation of scientific computing

**What we use it for:** Fast array operations, random number generation with distributions

**Key feature:** Can operate on entire arrays at once (much faster than loops)

```
python  
  
import pandas as pd
```

**What it is:** "Panel Data" - Excel-like tables in Python

**What we use it for:** Converting SQL query results to nice tables

**Key feature:** DataFrames are like spreadsheet sheets with superpowers

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

**What it is:** The main Python plotting library

**What we use it for:** Creating all our charts (bar, scatter, pie, etc.)

**Why "pyplot":** It's a MATLAB-style interface (easier to use)

```
python  
  
from matplotlib.colors import LinearSegmentedColormap
```

**What it is:** Tool for creating custom color gradients

**What we use it for:** Making space-themed color schemes (black → blue → green)

```
python  
  
import ipywidgets as widgets
```

**What it is:** Interactive HTML widgets for Jupyter notebooks

**What we use it for:** Creating buttons, sliders, dropdowns in the dashboard

**Note:** Only works in Jupyter/Colab environments

```
python  
  
from IPython.display import display, HTML, clear_output
```

**What it is:** Tools for displaying rich content in notebooks

- `display()`: Shows DataFrames, images, HTML
- `HTML()`: Renders HTML code
- `clear_output()`: Erases previous output (for updating displays)

```
python
```

```
import plotly.express as px  
import plotly.graph_objects as go
```

**What it is:** Interactive plotting library (zoom, hover, rotate)

**What we use it for:** Could make 3D interactive star maps (optional in this code)

**Why it's optional:** Not everyone has it installed

### Configuration Lines Explained

```
python
```

```
random.seed(42)  
np.random.seed(42)
```

**What this does:** Sets the "seed" for random number generators

**Why we need it:** Makes the random data reproducible

**Analogy:** Like setting a starting point - if you run it again with seed 42, you get the SAME "random" numbers

```
python
```

```
plt.style.use('dark_background')
```

**What this does:** Makes all matplotlib plots have dark backgrounds

**Why:** Space theme! Black background looks like space

```
python
```

```
plt.rcParams['figure.figsize'] = [12, 8]
```

**What this does:** Sets default plot size to 12 inches wide × 8 inches tall

**rcParams:** "Runtime Configuration Parameters" - global settings for all plots

```
python
```

```
plt.rcParams['font.size'] = 10
```

**What this does:** Sets default font size for all text in plots

---

## Section 2: Database Schema Explained

**What is a Database Schema?**

A **schema** is the blueprint of your database - it defines:

- What tables exist
- What columns each table has
- What data types each column holds
- How tables relate to each other

## Understanding Data Types

sql

**INTEGER PRIMARY KEY AUTOINCREMENT**

- **INTEGER:** Whole numbers (1, 2, 3, ...)
- **PRIMARY KEY:** Unique identifier for each row (no duplicates)
- **AUTOINCREMENT:** Database automatically assigns 1, 2, 3, 4... (you don't have to)

sql

**TEXT NOT NULL UNIQUE**

- **TEXT:** Strings (letters, words)
- **NOT NULL:** This field must have a value (can't be empty)
- **UNIQUE:** No two rows can have the same value here

sql

**REAL NOT NULL**

- **REAL:** Decimal numbers (3.14, 2.71828, ...)
- **NOT NULL:** Must have a value

sql

**DATE NOT NULL**

- **DATE:** Calendar dates (stored as 'YYYY-MM-DD')

sql

## BOOLEAN DEFAULT 0

- **BOOLEAN:** True/False (stored as 1 or 0)
- **DEFAULT 0:** If you don't specify, it's False

**Table 1: STAR\_SYSTEMS**

sql

```
CREATE TABLE IF NOT EXISTS STAR_SYSTEMS (  
  star_id INTEGER PRIMARY KEY AUTOINCREMENT,  
  name TEXT NOT NULL UNIQUE,  
  spectral_type TEXT NOT NULL,  
  luminosity REAL NOT NULL,  
  temperature INTEGER NOT NULL,  
  distance_ly REAL NOT NULL,  
  age_gyr REAL NOT NULL,  
  mass_solar REAL NOT NULL,  
  ra_deg REAL NOT NULL,  
  dec_deg REAL NOT NULL,  
  galactic_quadrant TEXT NOT NULL,  
  metallicity REAL NOT NULL  
)
```

### Line-by-line breakdown:

1. `CREATE TABLE IF NOT EXISTS STAR_SYSTEMS (` - Create the table only if it doesn't already exist
2. `star_id INTEGER PRIMARY KEY AUTOINCREMENT,` - Unique ID for each star (1, 2, 3...)
3. `name TEXT NOT NULL UNIQUE,` - Star name like "Alpha Novarum 7" (must be unique)
4. `spectral_type TEXT NOT NULL,` - Star classification: O, B, A, F, G, K, M (see below)
5. `luminosity REAL NOT NULL,` - How bright compared to our Sun (1.0 = Sun's brightness)
6. `temperature INTEGER NOT NULL,` - Surface temperature in Kelvin
7. `distance_ly REAL NOT NULL,` - How far from Earth in light-years
8. `age_gyr REAL NOT NULL,` - Age in billions of years (gyr = gigayears)
9. `mass_solar REAL NOT NULL,` - Mass compared to Sun (1.0 = Sun's mass)
10. `ra_deg REAL NOT NULL,` - Right Ascension (like longitude in space)
11. `dec_deg REAL NOT NULL,` - Declination (like latitude in space)

12. `galactic_quadrant TEXT NOT NULL,` - Which part of galaxy (Alpha, Beta, Gamma, Delta)
13. `metallicity REAL NOT NULL,` - Heavy element abundance (affects planet formation)

**Spectral Types Explained:** Stars are classified O-B-A-F-G-K-M from hottest to coolest:

- **O:** Blue, 30,000+ K, massive, rare
- **B:** Blue-white, 10,000-30,000 K
- **A:** White, 7,500-10,000 K
- **F:** Yellow-white, 6,000-7,500 K
- **G:** Yellow (our Sun!), 5,200-6,000 K
- **K:** Orange, 3,700-5,200 K
- **M:** Red (most common), 2,400-3,700 K

**Mnemonic:** "Oh Be A Fine Girl/Guy, Kiss Me"

**Table 2: EXOPLANETS**

sql

```
CREATE TABLE IF NOT EXISTS EXOPLANETS (  
  planet_id INTEGER PRIMARY KEY AUTOINCREMENT,  
  star_id INTEGER NOT NULL,  
  name TEXT NOT NULL UNIQUE,  
  mass_earth REAL NOT NULL,  
  radius_earth REAL NOT NULL,  
  orbital_period_days REAL NOT NULL,  
  semi_major_axis_au REAL NOT NULL,  
  eccentricity REAL NOT NULL,  
  equilibrium_temp_k REAL NOT NULL,  
  surface_gravity_earth REAL NOT NULL,  
  density_gcc REAL NOT NULL,  
  planet_type TEXT NOT NULL,  
  discovery_date DATE NOT NULL,  
  confirmed BOOLEAN NOT NULL DEFAULT 1,  
  FOREIGN KEY (star_id) REFERENCES STAR_SYSTEMS(star_id)  
)
```

**Key concepts:**

sql

**FOREIGN KEY** (star\_id) **REFERENCES** STAR\_SYSTEMS(star\_id)

**What this means:** The (star\_id) in this table **MUST** match a (star\_id) in the STAR\_SYSTEMS table

**Why it matters:** Ensures data integrity - you can't have a planet orbiting a star that doesn't exist

**Real-world analogy:** Like saying "every student must be enrolled in a class that actually exists"

#### Column explanations:

- (mass\_earth): Mass relative to Earth (2.0 = twice Earth's mass)
- (radius\_earth): Radius relative to Earth (1.5 = 50% larger than Earth)
- (orbital\_period\_days): How long it takes to orbit its star (Earth = 365.25 days)
- (semi\_major\_axis\_au): Average distance from star (AU = Astronomical Unit, Earth-Sun distance)
- (eccentricity): How elliptical the orbit is (0 = perfect circle, 0.9 = very stretched)
- (equilibrium\_temp\_k): Theoretical temperature if planet were a blackbody
- (surface\_gravity\_earth): Gravity relative to Earth (2.0 = you'd weigh twice as much)
- (density\_gcc): Density in grams per cubic centimeter (Earth = 5.5)

**Table 3: DISCOVERY\_MISSIONS**

```
sql

CREATE TABLE IF NOT EXISTS DISCOVERY_MISSIONS (
  mission_id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL UNIQUE,
  organization TEXT NOT NULL,
  mission_type TEXT NOT NULL,
  detection_method TEXT NOT NULL,
  launch_date DATE NOT NULL,
  end_date DATE,
  status TEXT NOT NULL,
  total_discoveries INTEGER DEFAULT 0,
  sensitivity_rating INTEGER NOT NULL
)
```

#### Detection Methods Explained:

- **Transit:** Planet passes in front of star, dims the light (Kepler used this)
- **Radial Velocity:** Star wobbles due to planet's gravity (Doppler shift)
- **Direct Imaging:** Actually see the planet (very hard!)

- **Astrometry:** Measure star's position changes
- **Interferometry:** Combine light from multiple telescopes

**Table 4: PLANET\_DISCOVERIES (Junction Table)**

```
sql

CREATE TABLE IF NOT EXISTS PLANET_DISCOVERIES (
  discovery_id INTEGER PRIMARY KEY AUTOINCREMENT,
  planet_id INTEGER NOT NULL,
  mission_id INTEGER NOT NULL,
  detection_date DATE NOT NULL,
  discovery_role TEXT NOT NULL,
  snr_value REAL NOT NULL,
  transit_events INTEGER,
  discovery_team TEXT NOT NULL,
  publication_ref TEXT,
  FOREIGN KEY (planet_id) REFERENCES EXOPLANETS(planet_id),
  FOREIGN KEY (mission_id) REFERENCES DISCOVERY_MISSIONS(mission_id)
)
```

**What is a Junction Table?**

It connects two tables in a **many-to-many relationship**:

- One planet can be discovered by multiple missions
- One mission can discover multiple planets

**Real-world analogy:** Like a "Student-Class" table - students take multiple classes, classes have multiple students

**SNR (Signal-to-Noise Ratio):** How clear the detection was

- SNR > 5: Good detection
- SNR > 10: Excellent
- SNR < 3: Questionable

**Table 5: ATMOSPHERIC\_ANALYSIS**

```
sql
```

```
CREATE TABLE IF NOT EXISTS ATMOSPHERIC_ANALYSIS (
  analysis_id INTEGER PRIMARY KEY AUTOINCREMENT,
  planet_id INTEGER NOT NULL UNIQUE,
  analysis_date DATE NOT NULL,
  hydrogen_pct REAL DEFAULT 0,
  helium_pct REAL DEFAULT 0,
  nitrogen_pct REAL DEFAULT 0,
  oxygen_pct REAL DEFAULT 0,
  carbon_dioxide_pct REAL DEFAULT 0,
  methane_pct REAL DEFAULT 0,
  water_vapor_pct REAL DEFAULT 0,
  ammonia_pct REAL DEFAULT 0,
  argon_pct REAL DEFAULT 0,
  other_pct REAL DEFAULT 0,
  surface_pressure_atm REAL,
  biosignature_detected BOOLEAN DEFAULT 0,
  atmosphere_stability TEXT,
  greenhouse_rating INTEGER,
  FOREIGN KEY (planet_id) REFERENCES EXOPLANETS(planet_id)
)
```

**Note:** (planet\_id INTEGER NOT NULL UNIQUE) means one-to-one relationship

Each planet can have only ONE atmospheric analysis record

**Biosignatures:** Chemical signs of life

- **Oxygen + Methane:** Shouldn't coexist naturally (they react)
- **Water vapor:** Essential for life as we know it
- **Ozone:** Indicates oxygen in atmosphere

**Table 6: HABITABILITY\_SCORES**

sql

```
CREATE TABLE IF NOT EXISTS HABITABILITY_SCORES (  
  score_id INTEGER PRIMARY KEY AUTOINCREMENT,  
  planet_id INTEGER NOT NULL UNIQUE,  
  esi_score REAL NOT NULL,  
  hz_status TEXT NOT NULL,  
  water_probability REAL NOT NULL,  
  atmosphere_rating INTEGER NOT NULL,  
  magnetic_field_probability REAL NOT NULL,  
  tidal_lock_probability REAL NOT NULL,  
  habitability_class TEXT NOT NULL,  
  study_priority INTEGER NOT NULL,  
  calculated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  FOREIGN KEY (planet_id) REFERENCES EXOPLANETS(planet_id)  
)
```

**ESI (Earth Similarity Index):** 0 to 1 scale (1 = identical to Earth)

**HZ Status (Habitable Zone):**

- "In HZ": Perfect distance for liquid water
- "Too Hot": Too close to star (water boils)
- "Too Cold": Too far from star (water freezes)

**Tidal Locking:** When one side always faces the star (like Moon always shows same face to Earth)

**Indexes Explained**

sql

```
CREATE INDEX idx_exoplanets_star ON EXOPLANETS(star_id)
```

**What is an index?:** Like a book's index - helps find data faster

**Without index:** Database scans every row (slow)

**With index:** Database uses a sorted lookup table (fast)

**When to use indexes:**

- Columns used in WHERE clauses
- Columns used in JOIN conditions
- Foreign key columns

## Section 3: Mathematical Formulas Explained

### 3.1 Habitable Zone Calculation

```
python

def calculate_habitable_zone(luminosity, temp_k):
    t_star = temp_k - 5780

    s_eff_inner = 1.0146 + 8.1884e-5 * t_star + 1.9394e-9 * t_star**2
    s_eff_outer = 0.3507 + 5.9578e-5 * t_star + 1.6707e-9 * t_star**2

    inner_hz = math.sqrt(luminosity / s_eff_inner)
    outer_hz = math.sqrt(luminosity / s_eff_outer)

    return (inner_hz, outer_hz)
```

#### Mathematical Theory:

The habitable zone is where liquid water can exist. Based on **Kopparapu et al. (2013)**.

**Step 1:** Calculate temperature offset from our Sun

```
t_star = temp_k - 5780
```

5780 K is our Sun's temperature. This measures how much hotter/cooler the star is.

**Step 2:** Calculate effective solar flux boundaries

**Inner boundary** (Runaway Greenhouse):

```
s_eff_inner = 1.0146 + 8.1884×10-5 × t_star + 1.9394×10-9 × t_star2
```

This is a **quadratic equation**:  $y = ax^2 + bx + c$

It accounts for how stellar spectrum affects heating.

**Outer boundary** (Maximum Greenhouse):

```
s_eff_outer = 0.3507 + 5.9578×10-5 × t_star + 1.6707×10-9 × t_star2
```

**Step 3:** Calculate distance in AU (Astronomical Units)

```
inner_hz = √(luminosity / s_eff_inner)
```

```
outer_hz = √(luminosity / s_eff_outer)
```

### Why the square root?

From the **inverse square law**:  $\text{Flux} \propto \text{Luminosity} / \text{Distance}^2$

Rearranging:  $\text{Distance} = \sqrt{(\text{Luminosity} / \text{Flux})}$

### Example:

- Sun-like star (L=1, T=5780): HZ is 0.95 - 1.37 AU (Earth at 1.0 AU ✓)
- M-dwarf (L=0.05, T=3500): HZ is 0.16 - 0.32 AU (much closer!)

## 3.2 Earth Similarity Index (ESI)

python

```
def calculate_esi(radius_earth, mass_earth, temp_k, escape_vel_ratio=1.0):  
    r_ref = 1.0  
    m_ref = 1.0  
    t_ref = 288.0  
    v_ref = 1.0  
  
    w_r = 0.57  
    w_m = 1.07  
    w_t = 5.58  
    w_v = 0.70  
  
    esi_r = (1 - abs((radius_earth - r_ref) / (radius_earth + r_ref))) ** w_r  
    esi_m = (1 - abs((mass_earth - m_ref) / (mass_earth + m_ref))) ** w_m  
    esi_t = (1 - abs((temp_k - t_ref) / (temp_k + t_ref))) ** w_t  
    esi_v = (1 - abs((escape_vel_ratio - v_ref) / (escape_vel_ratio + v_ref))) ** w_v  
  
    esi = (esi_r * esi_m * esi_t * esi_v) ** 0.25  
  
    return min(max(es_i, 0), 1)
```

### Mathematical Theory:

ESI uses **similarity metrics** for 4 parameters. Based on **Schulze-Makuch et al. (2011)**.

**Step 1:** For each parameter, calculate similarity

General formula:

$$\text{similarity} = (1 - |\text{planet\_value} - \text{earth\_value}| / |\text{planet\_value} + \text{earth\_value}|) ^ \text{weight}$$

### Why this formula?

- Numerator: Absolute difference from Earth
- Denominator: Sum of values (normalization)
- Division gives a value from 0 to 1
- Subtracting from 1 inverts it (1 = same, 0 = very different)

### Example (radius):

- Earth-sized planet (radius = 1.0):

$$\begin{aligned}\text{esi\_r} &= (1 - |1.0 - 1.0| / |1.0 + 1.0|)^{0.57} \\ &= (1 - 0 / 2)^{0.57} \\ &= 1^{0.57} = 1.0 \quad \checkmark \text{ Perfect}\end{aligned}$$

- Jupiter-sized planet (radius = 11.2):

$$\begin{aligned}\text{esi\_r} &= (1 - |11.2 - 1.0| / |11.2 + 1.0|)^{0.57} \\ &= (1 - 10.2 / 12.2)^{0.57} \\ &= (1 - 0.836)^{0.57} \\ &= 0.164^{0.57} \approx 0.33 \quad \text{Much less similar}\end{aligned}$$

### Step 2: Weights explained

- **w<sub>r</sub> = 0.57**: Radius weight (moderate importance)
- **w<sub>m</sub> = 1.07**: Mass weight (high importance - affects gravity)
- **w<sub>t</sub> = 5.58**: Temperature weight (VERY high - critical for life)
- **w<sub>v</sub> = 0.70**: Escape velocity (moderate - affects atmosphere retention)

### Why different weights?

Temperature matters most for habitability. A cold Earth-sized planet can't support life.

### Step 3: Geometric mean

$$\text{esi} = (\text{esi\_r} \times \text{esi\_m} \times \text{esi\_t} \times \text{esi\_v}) ^ 0.25$$

### Why geometric mean? (not arithmetic mean)

If ANY parameter is terrible (close to 0), ESI should be low.

- Arithmetic:  $(1.0 + 1.0 + 0.1 + 1.0) / 4 = 0.775$  (misleading)
- Geometric:  $(1.0 \times 1.0 \times 0.1 \times 1.0)^{0.25} = 0.562$  (more realistic)

### Step 4: Clamp to [0, 1]

```
python  
  
return min(max(esi, 0), 1)
```

Ensures ESI stays between 0 and 1.

## 3.3 Equilibrium Temperature

```
python  
  
def calculate_equilibrium_temperature(star_temp, star_radius, semi_major_axis, albedo=0.3):  
    r_star_m = star_radius * 6.957e8  
    a_m = semi_major_axis * 1.496e11  
  
    t_eq = star_temp * ((1 - albedo) ** 0.25) * math.sqrt(r_star_m / (2 * a_m))  
  
    return t_eq
```

### Mathematical Theory:

Assumes planet is a **blackbody in radiative equilibrium**.

**Energy Balance:** Energy absorbed = Energy radiated

**Absorbed energy** (from star):

$$P_{\text{in}} = (1 - A) \times L_{\text{star}} \times (R_{\text{planet}}^2 / 4a^2)$$

- $A$  = albedo (fraction of light reflected)
- $L_{\text{star}}$  = star's luminosity
- Cross-section of planet =  $\pi R^2$
- Spread over sphere =  $4\pi R^2$

**Radiated energy** (Stefan-Boltzmann Law):

$$P_{\text{out}} = \sigma \times T^4 \times 4\pi R_{\text{planet}}^2$$

- $\sigma$  = Stefan-Boltzmann constant
- $T$  = temperature
- Surface area =  $4\pi R^2$

**Setting  $P_{\text{in}} = P_{\text{out}}$  and solving for  $T$ :**

$$T_{\text{eq}} = T_{\text{star}} \times (1 - A)^{0.25} \times \sqrt{(R_{\text{star}} / 2a)}$$

**Physical intuition:**

- Hotter star → Hotter planet (linear)
- Lower albedo → Hotter planet (reflects less)
- Closer to star → Hotter planet ( $\sqrt{\phantom{x}}$  relationship)
- Bigger star → Hotter planet (more surface emitting)

**Example:** Earth's equilibrium temperature:

$$T_{\text{eq}} = 5780 \times (1 - 0.3)^{0.25} \times \sqrt{(696,000 / (2 \times 149,600,000))}$$

$$\approx 255 \text{ K} = -18^\circ\text{C}$$

Why colder than actual (288 K)? **Greenhouse effect** warms us by ~33 K!

### 3.4 Kepler's Third Law (Used implicitly)

python

```
orbital_period = 365.25 * math.sqrt((sma ** 3) / star_mass)
```

**Mathematical Theory:**

**Kepler's Third Law:**  $P^2 = a^3 / M$

- $P$  = orbital period (years)
- $a$  = semi-major axis (AU)
- $M$  = star mass (solar masses)

**Rearranged:**

$$P = \sqrt{(a^3 / M)}$$

**In days** (not years):

$$P_{\text{days}} = 365.25 \times \sqrt{(a^3 / M)}$$

**Why it works:**

Balance between gravitational force and centripetal force.

**Example:** Earth orbiting Sun:

$$P = \sqrt{(1^3 / 1)} = 1 \text{ year} \quad \checkmark$$

Hot Jupiter at 0.05 AU around Sun-like star:

$$P = \sqrt{(0.05^3 / 1)} = \sqrt{0.000125} = 0.0112 \text{ years} \approx 4 \text{ days}$$

---

## Section 4: Data Generation Explained

### 4.1 Generating Star Systems

```
python

def generate_star_systems(conn, num_stars=150):
    spectral_data = [
        ('M', 0.45, (2400, 3700), (0.001, 0.08), (0.08, 0.45)),
        ('K', 0.25, (3700, 5200), (0.08, 0.6), (0.45, 0.8)),
        ('G', 0.15, (5200, 6000), (0.6, 1.5), (0.8, 1.04)),
        ...
    ]
```

**Understanding spectral\_data format:**

```
python

(spectral_type, probability, temp_range, luminosity_range, mass_range)
```

**Example:** `('M', 0.45, (2400, 3700), (0.001, 0.08), (0.08, 0.45))`

- Type M stars

- 45% of all stars
- Temperature: 2400-3700 K
- Luminosity: 0.001-0.08 solar luminosities (very dim!)
- Mass: 0.08-0.45 solar masses

### Why these distributions?

Based on observations of stars in our galaxy. M-dwarfs are most common!

### Selecting spectral type:

```
python

rand = random.random()
cumulative = 0
for spec_type, prob, temp_range, lum_range, mass_range in spectral_data:
    cumulative += prob
    if rand <= cumulative:
        spectral_type = spec_type + str(random.randint(0, 9))
        break
```

### How this works:

1. Get random number 0.0-1.0
2. Add up probabilities: 0.45, 0.70, 0.85, 0.93, 0.97, 0.99, 1.00
3. First cumulative value that exceeds `rand` wins

### Visual:

```
|--- M (45%) ---|--- K (25%) --|--- G (15%) -|...
0      0.45      0.70      0.85
```

If rand = 0.32 → M

If rand = 0.61 → K

If rand = 0.88 → F

### Distance generation:

```
python

distance = random.expovariate(1/100) + 10
```

**Exponential distribution:** Most stars are nearby, fewer far away

**Formula:**  $f(x) = \lambda e^{(-\lambda x)}$

$\lambda = 1/100$ : Average distance is 100 light-years

**+10:** Minimum 10 light-years (no extremely close stars)

**Why exponential?** Detection bias - easier to find nearby stars!

## 4.2 Generating Exoplanets

```
python
```

```
num_planets = np.random.poisson(avg_planets_per_star)
```

**Poisson distribution:** Models count data

$\lambda = 3$ : Average 3 planets per star

**Result:** Some stars have 0, 1, 2, 5, 8 planets (realistic variation)

**Why Poisson?** Matches observations from Kepler mission.

**Planet size distribution:**

```
python
```

```
size_rand = random.random()
if size_rand < 0.40:
    radius = random.uniform(0.5, 1.5)    # Earth-sized (40%)
elif size_rand < 0.70:
    radius = random.uniform(1.5, 2.5)    # Super-Earths (30%)
elif size_rand < 0.85:
    radius = random.uniform(2.5, 4)      # Mini-Neptunes (15%)
elif size_rand < 0.95:
    radius = random.uniform(4, 11)       # Gas giants (10%)
else:
    radius = random.uniform(11, 25)      # Super-Jupiters (5%)
```

**Based on Kepler data:** Small planets are more common than large ones.

**Mass-radius relation:**

```
python
```

```

if radius < 1.5:
    mass = radius ** 3.5 # Rocky
elif radius < 4:
    mass = radius ** 2.5 # Mixed
else:
    mass = 30 * (radius / 4) ** 2 # Gas

```

### Why different exponents?

- **Rocky (3.5):** Volume scales as  $R^3$ , but compression increases density
- **Gas giants (2):** More mass doesn't increase radius much (gravitational compression)

**Physical intuition:** Jupiter is  $318\times$  Earth's mass but only  $11\times$  Earth's radius!

### 4.3 Atmospheric Composition

```

python

if 'Terrestrial' in planet_type:
    if temp > 350: # Hot
        co2 = random.uniform(50, 95) # Venus-like
    elif 200 < temp < 350: # Temperate
        n2 = random.uniform(60, 85) # Earth-like
        o2 = random.uniform(0, 25)
    else: # Cold
        n2 = random.uniform(80, 99) # Thin atmosphere

```

### Logic:

- **Hot rocky planets:**  $\text{CO}_2$ -dominated (greenhouse runaway like Venus)
- **Temperate rocky:**  $\text{N}_2$  + possible  $\text{O}_2$  (like Earth)
- **Cold rocky:** Mostly  $\text{N}_2$ , very little else

### Gas giants:

```

python

h = random.uniform(70, 95) # Mostly  $\text{H}_2$ 
he = random.uniform(5, 25) # Helium second

```

Like Jupiter and Saturn!

### Normalization:

```
python
```

```
total = h + he + n2 + o2 + co2 + ch4 + h2o
```

```
factor = 97 / total
```

```
h *= factor
```

```
he *= factor
```

```
...
```

**Why?** Ensures percentages add up to ~100%

**Biosignature detection:**

```
python
```

```
biosignature = (o2 > 10 and ch4 > 0.5 and 200 < temp < 350 and  
                random.random() < 0.1)
```

**Logic:** Life detected if:

- $O_2 > 10\%$  (photosynthesis indicator)
- $CH_4 > 0.5\%$  (biological methane)
- Temperature habitable
- 10% random chance (even then, it's rare!)

**Why  $O_2$  +  $CH_4$ ?** They shouldn't coexist naturally - they react to form  $CO_2$  +  $H_2O$ . Life continuously replenishes both!

---

## Section 5: SQL Queries Explained

### 5.1 Basic SELECT Query

```
sql
```

```
SELECT * FROM STAR_SYSTEMS LIMIT 5
```

**Breakdown:**

- `SELECT`: "Give me data"
- `*`: "All columns"
- `FROM STAR_SYSTEMS`: "From this table"

- `LIMIT 5`: "Only first 5 rows"

**Result:** A table with all columns for 5 stars

## 5.2 JOIN Operations

```
sql

SELECT
  e.name as planet_name,
  s.name as host_star
FROM EXOPLANETS e
JOIN STAR_SYSTEMS s ON e.star_id = s.star_id
```

### What is a JOIN?

Combines rows from two tables based on a related column.

### Visual Example:

#### EXOPLANETS table:

planet_id	star_id	name
1	5	Alpha Novarum 7 b
2	5	Alpha Novarum 7 c
3	8	Beta Sideris 2 b

#### STAR\_SYSTEMS table:

star_id	name
5	Alpha Novarum 7
8	Beta Sideris 2

#### After JOIN (on star\_id):

planet_name	host_star
Alpha Novarum 7 b	Alpha Novarum 7
Alpha Novarum 7 c	Alpha Novarum 7
Beta Sideris 2 b	Beta Sideris 2

#### Table aliases:

- `e` = alias for EXOPLANETS
- `s` = alias for STAR\_SYSTEMS

- Makes queries shorter: `e.name` instead of `EXOPLANETS.name`

## 5.3 Aggregation Functions

sql

```
SELECT
  COUNT(*) as planet_count,
  AVG(h.esi_score) as avg_esi,
  MIN(h.esi_score) as min_esi,
  MAX(h.esi_score) as max_esi
FROM HABITABILITY_SCORES h
```

### Aggregation functions:

- `COUNT(*)`: How many rows?
- `AVG(column)`: Average value
- `MIN(column)`: Smallest value
- `MAX(column)`: Largest value
- `SUM(column)`: Total of all values

### Example result:

```
planet_count | avg_esi | min_esi | max_esi
450          | 0.4821 | 0.0012 | 0.9876
```

## 5.4 GROUP BY

sql

```
SELECT
  planet_type,
  COUNT(*) as count,
  AVG(mass_earth) as avg_mass
FROM EXOPLANETS
GROUP BY planet_type
```

### What GROUP BY does:

Splits data into groups and calculates aggregates for each group.

### Step-by-step:

## Original data:

name	planet_type	mass_earth
Earth Twin	Terrestrial	1.0
Super Earth 1	Super-Earth	3.5
Hot Jupiter 1	Hot Jupiter	300
Super Earth 2	Super-Earth	4.2
Hot Jupiter 2	Hot Jupiter	280

## After GROUP BY planet\_type:

planet_type	count	avg_mass
Terrestrial	1	1.0
Super-Earth	2	3.85
Hot Jupiter	2	290

**Key rule:** If you use GROUP BY, your SELECT can only have:

1. The grouped column
2. Aggregate functions

## This is WRONG:

```
sql

SELECT planet_type, name, COUNT(*) -- ERROR!
FROM EXOPLANETS
GROUP BY planet_type
```

Why? Which name would it show when there are multiple planets per type?

## 5.5 HAVING Clause

```
sql

SELECT
    star_id,
    COUNT(*) as num_planets
FROM EXOPLANETS
GROUP BY star_id
HAVING COUNT(*) > 3
```

## HAVING vs WHERE:

- **WHERE**: Filters BEFORE grouping (filters individual rows)
- **HAVING**: Filters AFTER grouping (filters groups)

### Example:

```
sql

-- Find stars with more than 3 planets AND all planets confirmed
SELECT star_id, COUNT(*) as num_planets
FROM EXOPLANETS
WHERE confirmed = 1      -- WHERE: filter individual planets
GROUP BY star_id
HAVING COUNT(*) > 3      -- HAVING: filter groups of planets
```

## 5.6 Subqueries

```
sql

SELECT name, esi_score
FROM HABITABILITY_SCORES
WHERE esi_score > (SELECT AVG(es_i_score) FROM HABITABILITY_SCORES)
```

### What's happening:

1. **Inner query** (subquery): Calculate average ESI

```
sql

SELECT AVG(es_i_score) FROM HABITABILITY_SCORES
-- Returns: 0.4821
```

2. **Outer query**: Find planets above that average

```
sql

SELECT name, esi_score
FROM HABITABILITY_SCORES
WHERE esi_score > 0.4821
```

### Types of subqueries:

1. **Scalar subquery** (returns single value):

```
sql
```

```
WHERE esi_score > (SELECT AVG(esl_score) FROM ...)
```

## 2. Table subquery (returns multiple rows):

```
sql
```

```
WHERE planet_id IN (SELECT planet_id FROM ... WHERE ...)
```

## 3. EXISTS subquery (checks if any rows exist):

```
sql
```

```
WHERE EXISTS (SELECT 1 FROM ... WHERE ...)
```

## 5.7 CASE Expressions

```
sql
```

```
SELECT
  name,
  mass_earth,
  CASE
    WHEN mass_earth < 2 THEN 'Earth-like'
    WHEN mass_earth < 10 THEN 'Super-Earth'
    WHEN mass_earth < 100 THEN 'Neptune-like'
    ELSE 'Jupiter-like'
  END as size_category
FROM EXOPLANETS
```

### How CASE works:

Like an if-else chain in programming.

### Example execution:

- Planet with mass 1.2:
  - Is  $1.2 < 2$ ? YES → 'Earth-like' ✓ (stop checking)
- Planet with mass 5.0:
  - Is  $5.0 < 2$ ? NO
  - Is  $5.0 < 10$ ? YES → 'Super-Earth' ✓
- Planet with mass 300:
  - Is  $300 < 2$ ? NO

- Is  $300 < 10$ ? NO
- Is  $300 < 100$ ? NO
- → ELSE → 'Jupiter-like' ✓

## Result:

name	mass_earth	size_category
Earth Twin	1.0	Earth-like
Super Mars	3.5	Super-Earth
Mini Neptune	15.0	Neptune-like
Hot Jupiter	300	Jupiter-like

## 5.8 Window Functions

```
sql
SELECT
  name,
  esi_score,
  RANK() OVER (ORDER BY esi_score DESC) as overall_rank
FROM HABITABILITY_SCORES
```

### What are window functions?

Perform calculations across a set of rows related to the current row, **WITHOUT** collapsing rows like GROUP BY.

### RANK() example:

name	esi_score	overall_rank
Earth Twin	0.9876	1
Super Habitable	0.9654	2
Almost Earth	0.9654	2 ← Tied!
Kepler Analog	0.9123	4 ← Skips 3

### PARTITION BY (groups within window):

```
sql
RANK() OVER (PARTITION BY planet_type ORDER BY esi_score DESC)
```

## Result:

planet_type	name	esi_score	rank_in_type
Terrestrial	Earth Twin	0.9876	1
Terrestrial	Mars Like	0.7234	2
Super-Earth	Super Hab	0.9654	1
Super-Earth	Kepler-452b	0.8321	2

Each planet type gets its own ranking!

### Common window functions:

- `ROW_NUMBER()`: 1, 2, 3, 4... (no ties)
- `RANK()`: 1, 2, 2, 4... (ties skip numbers)
- `DENSE_RANK()`: 1, 2, 2, 3... (ties don't skip)
- `SUM() OVER (...)`: Running total
- `AVG() OVER (...)`: Moving average

## 5.9 Complex Multi-Table JOIN

```
sql

SELECT
  e.name as planet_name,
  s.name as host_star,
  h.esi_score,
  a.oxygen_pct,
  m.name as discovered_by
FROM EXOPLANETS e
JOIN STAR_SYSTEMS s ON e.star_id = s.star_id
JOIN HABITABILITY_SCORES h ON e.planet_id = h.planet_id
LEFT JOIN ATMOSPHERIC_ANALYSIS a ON e.planet_id = a.planet_id
LEFT JOIN PLANET_DISCOVERIES pd ON e.planet_id = pd.planet_id
LEFT JOIN DISCOVERY_MISSIONS m ON pd.mission_id = m.mission_id
WHERE h.esi_score > 0.8
```

### JOIN types:

#### 1. INNER JOIN (or just JOIN):

Table A	Table B	Result
id   name	id   color	id   name   color
1   Apple	1   Red	1   Apple   Red
2   Banana	3   Yellow	3   Grape   Yellow
3   Grape		

Only rows with matches in BOTH tables.

## 2. LEFT JOIN:

Table A	Table B	Result
id   name	id   color	id   name   color
1   Apple	1   Red	1   Apple   Red
2   Banana	3   Yellow	2   Banana   NULL
3   Grape		3   Grape   Yellow

ALL rows from left table, NULL if no match on right.

## Why LEFT JOIN for atmospheres?

Not all planets have atmospheric data! We still want to see the planet even if `ATMOSPHERIC_ANALYSIS` has no row for it.

## 5.10 Common Table Expressions (CTE)

```
sql

WITH mission_stats AS (
  SELECT
    mission_id,
    COUNT(*) as total_discoveries
  FROM PLANET_DISCOVERIES
  GROUP BY mission_id
)
SELECT
  m.name,
  ms.total_discoveries
FROM DISCOVERY_MISSIONS m
JOIN mission_stats ms ON m.mission_id = ms.mission_id
ORDER BY ms.total_discoveries DESC
```

## What is WITH?

Creates a temporary named result set (like a temporary view).

## Step-by-step execution:

**Step 1:** Create `mission_stats` table:

```
mission_id | total_discoveries
1          | 2678
2          | 1543
3          | 421
```

**Step 2:** Use it in main query:

```
name                | total_discoveries
Kepler Space Telescope | 2678
TESS                 | 1543
James Webb Space Telescope | 421
```

**Why use CTEs?**

1. Readability (break complex queries into steps)
2. Reusability (reference the same subquery multiple times)
3. Recursion (CTEs can reference themselves!)

**Alternative without CTE** (harder to read):

```
sql

SELECT
    m.name,
    (SELECT COUNT(*) FROM PLANET_DISCOVERIES pd WHERE pd.mission_id = m.mission_id) as total
FROM DISCOVERY_MISSIONS m
ORDER BY total DESC
```

---

## Section 6: Visualizations Explained

### 6.1 Matplotlib Basics

```
python

fig, ax = plt.subplots(figsize=(12, 8))
```

**Breakdown:**

- `fig`: The entire figure (the canvas)
- `ax`: The axes (the plotting area)
- `figsize=(12, 8)`: 12 inches wide, 8 inches tall

**Analogy:** `fig` is the picture frame, `ax` is the canvas inside.

## 6.2 Pie Chart

```
python

wedges, texts, autotexts = ax.pie(
    df['planet_count'],
    labels=df['habitability_class'],
    autopct='%1.1f%%',
    colors=colors,
    explode=[0.1, 0.05, 0, 0, 0]
)
```

### Parameters:

- `df['planet_count']`: Values to plot (slice sizes)
- `labels`: Text labels for each slice
- `autopct='%1.1f%%'`: Format for percentages (1 decimal place)
- `colors`: List of colors for each slice
- `explode`: How far to "pull out" each slice (0 = not pulled)

### Return values:

- `wedges`: The pie slices (can modify their properties)
- `texts`: Label text objects
- `autotexts`: Percentage text objects

### Example explode:

```
python

explode=[0.1, 0.05, 0, 0, 0]
```

- First slice: Pulled out 10% of radius
- Second slice: Pulled out 5%

- Rest: Not pulled out

## 6.3 Scatter Plot

```
python

scatter = ax.scatter(
    df['mass_earth'],
    df['radius_earth'],
    c=df['esi_score'],
    s=df['equilibrium_temp_k'] / 10,
    cmap='plasma',
    alpha=0.7
)
```

### Parameters:

- `x`: X-axis values (mass)
- `y`: Y-axis values (radius)
- `c`: Color values (ESI score) - creates a color gradient
- `s`: Size values (temperature) - bigger = hotter
- `cmap='plasma'`: Color map (purple → orange → yellow)
- `alpha=0.7`: Transparency (0=invisible, 1=opaque)

**Why transparency?** Overlapping points are visible!

### Log scale:

```
python

ax.set_xscale('log')
ax.set_yscale('log')
```

**Why?** Planet masses range from 0.5 to 3000 Earth masses!

### Linear scale:

```
|---|---|---|---|---|
0  500 1000 1500 2000 2500 3000
```

Everything below 100 is squished!

## Log scale:

|---|---|---|---|---|---|  
0.1 1 10 100 1000 10000

Much better distribution!

## 6.4 Heatmap

```
python

normalized = np.zeros_like(data_matrix, dtype=float)
for i in range(len(metrics)):
    row = data_matrix[i].astype(float)
    if row.max() > row.min():
        normalized[i] = (row - row.min()) / (row.max() - row.min())
```

## Normalization formula:

```
normalized_value = (value - min) / (max - min)
```

## Example:

```
Original: [10, 50, 100]
Min: 10, Max: 100

Normalized[0] = (10 - 10) / (100 - 10) = 0 / 90 = 0.0
Normalized[1] = (50 - 10) / (100 - 10) = 40 / 90 = 0.44
Normalized[2] = (100 - 10) / (100 - 10) = 90 / 90 = 1.0

Result: [0.0, 0.44, 1.0]
```

**Why normalize?** Different metrics have different scales:

- Planets per star: 0-10
- Average ESI: 0-1

Normalization puts them on same 0-1 scale for fair color comparison.

## Creating heatmap:

```
python
```

```
im = ax.imshow(normalized, cmap=cmap, aspect='auto')
```

- `imshow`: Displays 2D array as image
- `aspect='auto'`: Automatically adjust cell shapes

## 6.5 Subplots

python

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))
```

**Format:** `subplots(rows, columns)`

**Examples:**

- `(1, 2)`: 1 row, 2 columns (side-by-side)
- `(2, 1)`: 2 rows, 1 column (stacked)
- `(2, 2)`: 2×2 grid (4 plots)

**Unpacking:**

python

```
fig, (ax1, ax2) = plt.subplots(1, 2)
```

Creates two axes objects you can plot on separately.

**Alternative** (for many subplots):

python

```
fig, axes = plt.subplots(2, 2) # axes is a 2D array
axes[0, 0].plot(...) # Top-left
axes[0, 1].plot(...) # Top-right
axes[1, 0].plot(...) # Bottom-left
axes[1, 1].plot(...) # Bottom-right
```

## 6.6 Colorbars

python

```
cbar = plt.colorbar(scatter, ax=ax, shrink=0.8)
cbar.set_label('Earth Similarity Index (ESI)', fontsize=11)
```

### Parameters:

- `scatter`: The plot object to create colorbar for
- `ax`: Which axes to attach it to
- `shrink=0.8`: Make colorbar 80% of plot height

**Purpose:** Shows what colors represent in the plot.

## 6.7 Twin Axes

```
python  
  
ax1_twin = ax1.twinx()  
ax1_twin.plot(years, df['high_priority'], 'o-', color='ff00ff')
```

### What is `twinx()`?

Creates a second y-axis on the right side sharing the same x-axis.

**Use case:** Plotting two variables with different scales.

### Example:

```
Left axis (ax1):    Right axis (ax1_twin):  
Discoveries (0-100) — Priority count (0-10)  
    └─ Same x-axis (years)
```

## 6.8 Mollweide Projection

```
python  
  
fig, ax = plt.subplots(figsize=(16, 8), subplot_kw={'projection': 'mollweide'})
```

### What is Mollweide?

An equal-area map projection for spheres (like Earth or the sky).

### Coordinate conversion:

```
python  
  
ra_rad = np.radians(df['ra_deg'] - 180)  
dec_rad = np.radians(df['dec_deg'])
```

- Right Ascension (RA): 0-360° → converted to -180° to +180°

- Declination (Dec):  $-90^{\circ}$  to  $+90^{\circ}$

**Why subtract 180?** Centers the map at  $180^{\circ}$ .

---

## Section 7: Interactive Dashboard Explained

### 7.1 ipywidgets Basics

```
python

import ipywidgets as widgets
from IPython.display import display, clear_output
```

#### What are widgets?

Interactive HTML elements you can use in Jupyter notebooks:

- Sliders
- Dropdowns
- Buttons
- Checkboxes
- Text areas

### 7.2 Creating Widgets

#### Slider:

```
python

min_esi_slider = widgets.FloatSlider(
    value=0,      # Starting value
    min=0,        # Minimum value
    max=1,        # Maximum value
    step=0.05,    # Increment (click increases by 0.05)
    description='Min ESI Score:',
    style=style,
    layout=layout
)
```

#### Dropdown:

```
python
```

```
planet_type_dropdown = widgets.Dropdown(
    options=['All', 'Terrestrial', 'Super-Earth'],
    value='All',    # Default selection
    description='Planet Type:'
)
```

## Checkbox:

```
python

hz_only_checkbox = widgets.Checkbox(
    value=False,    # Unchecked by default
    description='Habitable Zone Only'
)
```

## Button:

```
python

search_button = widgets.Button(
    description='🔍 Search Planets',
    button_style='primary', # Blue color
    layout=widgets.Layout(width='200px')
)
```

## 7.3 Event Handlers

```
python

def on_search_click(b):
    with table_output:
        clear_output(wait=True)
        results = search_planets(...)
        display(results)

search_button.on_click(on_search_click)
```

## How it works:

1. Define a function that runs when button is clicked
2. `(b)` parameter = the button object (not usually needed)
3. `with table_output:` = direct output to specific widget

4. `clear_output(wait=True)` = erase previous output

5. `display(results)` = show new results

**on\_click** attaches the function to the button.

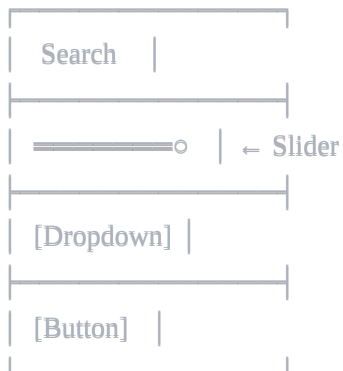
## 7.4 Layout

**VBox** (Vertical Box):

```
python

search_controls = widgets.VBox([
    widgets.HTML("<h3>Search</h3>"),
    min_esl_slider,
    planet_type_dropdown,
    search_button
])
```

Stacks widgets vertically:



**HBox** (Horizontal Box):

```
python

row = widgets.HBox([widget1, widget2, widget3])
```

Places widgets side-by-side:

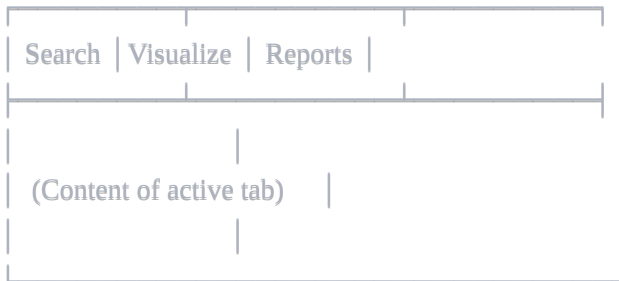


## 7.5 Tabs

```
python

tabs = widgets.Tab(children=[search_tab, viz_tab, report_tab])
tabs.set_title(0, '🔍 Search')
tabs.set_title(1, '📊 Visualize')
tabs.set_title(2, '📄 Reports')
```

### Result:



## 7.6 Output Widgets

```
python

table_output = widgets.Output()

with table_output:
    print("Hello")
    display(dataframe)
```

### What is Output widget?

A container that captures print statements and displays.

### Why use it?

Control WHERE output appears (instead of always at bottom of notebook).

---

## Section 8: Key Programming Concepts

### 8.1 List Comprehensions

```
python

angles = [n / float(n_cats) * 2 * np.pi for n in range(n_cats)]
```

## Equivalent loop:

```
python

angles = []
for n in range(n_cats):
    angle = n / float(n_cats) * 2 * np.pi
    angles.append(angle)
```

**Syntax:** `[expression for item in iterable]`

## Example:

```
python

squares = [x**2 for x in range(5)]
# Result: [0, 1, 4, 9, 16]
```

## 8.2 Dictionary Unpacking

```
python

stats = {
    'stars': 150,
    'planets': 450,
    'missions': 12
}

print(f"Stars: {stats['stars']}, Planets: {stats['planets']}")
```

**f-strings:** Format strings with variables

```
python

name = "Earth"
mass = 1.0
print(f"Planet {name} has mass {mass}")
# Output: Planet Earth has mass 1.0
```

## 8.3 Try-Except Blocks

```
python
```

```
try:
    result = risky_operation()
except Exception as e:
    print(f"Error: {e}")
```

**Purpose:** Handle errors gracefully without crashing.

**Example:**

```
python

try:
    cursor.execute(query)
except sqlite3.IntegrityError:
    print("Duplicate planet name!")
    # Continue instead of crashing
```

## 8.4 Lambda Functions

```
python

df['habitability_class'].apply(lambda x: x.split(':')[0])
```

**Lambda:** Anonymous (unnamed) function

**Equivalent:**

```
python

def get_class_prefix(x):
    return x.split(':')[0]

df['habitability_class'].apply(get_class_prefix)
```

**When to use lambda:** For simple one-line functions.

## 8.5 Enumerate

```
python

for idx, (_, row) in enumerate(df.iterrows()):
    print(f"Row {idx}: {row['name']}")
```

**enumerate** adds a counter:

```
python

colors = ['red', 'green', 'blue']
for idx, color in enumerate(colors):
    print(f"{idx}: {color}")

# Output:
# 0: red
# 1: green
# 2: blue
```

## Section 9: Astronomical Concepts Deep Dive

### 9.1 Spectral Classification (OBAFGKM)

Stars are classified by temperature and color:

Type	Color	Temp (K)	Examples	% of Stars
O	Blue	30,000+	Mintaka	0.00003%
B	Blue-white	10,000-30k	Rigel, Spica	0.13%
A	White	7,500-10k	Sirius, Vega	0.6%
F	Yellow-white	6,000-7.5k	Procyon	3%
G	Yellow	5,200-6k	<b>Sun</b> , α Cen A	7.6%
K	Orange	3,700-5.2k	Arcturus, ε Eri	12.1%
M	Red	2,400-3.7k	Proxima, Betelgeuse	76.45%

**Key insight:** M-dwarfs (red dwarfs) are BY FAR the most common stars!

**Subclasses:** Each letter has 0-9 (e.g., G0, G5, G9)

- G0: Hottest G star (close to F9)
- G9: Coolest G star (close to K0)

**Our Sun:** G2V

- G2: Temperature subclass
- V: "Main sequence" (normal star, not giant or dwarf)

### 9.2 Habitable Zone Concepts

**Goldilocks Zone:** Not too hot, not too cold, just right for liquid water.

### **Inner edge** (Runaway Greenhouse):

- Too much stellar radiation
- Water evaporates into atmosphere
- Greenhouse effect spirals out of control
- Example: Venus (although not quite in HZ)

### **Outer edge** (Maximum Greenhouse):

- Too little stellar radiation
- Even maximum CO<sub>2</sub> greenhouse can't keep water liquid
- Example: Mars is just outside HZ

### **Factors affecting HZ:**

1. **Star luminosity:** Brighter star → HZ farther out
2. **Star temperature:** Affects spectrum (UV, visible, IR)
3. **Planet albedo:** Reflective planet → colder
4. **Atmospheric composition:** More CO<sub>2</sub> → warmer

### **HZ for different stars:**

- **M-dwarf** (0.05 L<sub>☉</sub>): HZ at 0.2 AU (very close!)
- **Sun-like** (1.0 L<sub>☉</sub>): HZ at 1.0 AU
- **A-star** (10 L<sub>☉</sub>): HZ at 3+ AU

### **Problems with M-dwarf HZ:**

- Planets likely tidally locked (one side always faces star)
- Strong stellar flares (radiation)
- But M-dwarfs are most common AND live longest!

## **9.3 Earth Similarity Index (ESI) Theory**

**Why ESI matters:** Quick screening tool for habitability.

### **Four parameters:**

1. **Radius:** Affects gravity, atmosphere retention
2. **Mass** (really density): Rocky vs gaseous

3. **Temperature:** Most critical for life

4. **Escape velocity:** Can planet hold atmosphere?

#### Limitations:

- Doesn't account for atmospheres
- Doesn't account for magnetic fields
- Doesn't account for stellar radiation
- Venus would score high (but it's hell!)

**Good ESI ( $>0.8$ ) doesn't guarantee habitability, but low ESI ( $<0.3$ ) almost guarantees uninhabitability.**

### 9.4 Detection Methods

1. **Transit Method** (used by Kepler, TESS):



**Pros:** Can find small planets, measure radius

**Cons:** Planet must cross in front from our view (5% chance)

2. **Radial Velocity** (Doppler spectroscopy):

Star wobbles: 🌟 → ← 🪐

Wobble toward us: Blue-shifted light

Wobble away: Red-shifted light

**Pros:** Can measure planet mass

**Cons:** Easier to find massive, close planets

3. **Direct Imaging:** Actually photograph the planet!

**Pros:** Can study atmosphere directly

**Cons:** Only works for huge, young, hot planets far from star

4. **Gravitational Microlensing:** Background star's light bent by planet's gravity.

**Pros:** Can find planets very far away

**Cons:** One-time event, can't observe again

## 9.5 Biosignatures

### What are biosignatures?

Chemical or physical markers that indicate life.

### Atmospheric biosignatures:

#### 1. Oxygen (O<sub>2</sub>):

- On Earth: 21% of atmosphere
- Produced by photosynthesis
- Reacts quickly → needs constant replenishment
- **But:** Can be produced abiotically (without life)

#### 2. Methane (CH<sub>4</sub>):

- Produced by life (cow farts, wetlands)
- Also by geology (volcanoes, hydrothermal vents)
- Reacts with O<sub>2</sub> → shouldn't coexist naturally!

#### 3. Ozone (O<sub>3</sub>):

- Forms from O<sub>2</sub> in upper atmosphere
- Shields surface from UV
- Easier to detect than O<sub>2</sub> itself

#### 4. Phosphine (PH<sub>3</sub>):

- Proposed biosignature for Venus (controversial!)
- On Earth: Only made by life or industry
- But could have abiotic sources we don't know

### The "impossible" combination:

O<sub>2</sub> + CH<sub>4</sub> + H<sub>2</sub>O vapor + habitable temperature = Strong biosignature!

Chemical reaction:  $2\text{CH}_4 + \text{O}_2 \rightarrow 2\text{CH}_3\text{OH}$  (methanol)

If both exist, something must be constantly producing them!

### False positives:

- O<sub>2</sub> from photodissociation (UV splitting water)
- CH<sub>4</sub> from volcanism
- Need **multiple** biosignatures + geological context

## 9.6 Tidal Locking

### What is it?

One side of planet always faces star (like Moon always shows one face to Earth).

**Causes:** Tidal friction over time. Close-in planets lock faster.

### Formula (simplified):

$$t_{\text{lock}} \propto a^6 / (M_{\text{star}} \times R_{\text{planet}}^5)$$

Where:

- $a$  = distance from star
- $M_{\text{star}}$  = star mass
- $R_{\text{planet}}$  = planet radius

### Implications:

- **Day side:** Permanent noon (very hot)
- **Night side:** Permanent midnight (very cold)
- **Terminator zone:** Perpetual twilight (might be habitable!)

### Can life exist?

- Thick atmosphere might redistribute heat
- Liquid water possible in twilight zone
- Example: Proxima Centauri b is likely tidally locked

## 9.7 Planetary Mass-Radius Relations

### Why different exponents for different types?

#### Rocky planets ( $R < 1.5 R_{\oplus}$ ):

$$M = R^{3.5}$$

**Reason:** Composition similar to Earth (iron core + silicate mantle).

Self-gravity compresses material → density increases with mass.

**Example:**

- Earth:  $M = 1$ ,  $R = 1$
- Super-Earth ( $2 R_{\oplus}$ ):  $M \approx 2^{3.5} \approx 11 M_{\oplus}$  (not just  $2^3 = 8$ !)

**Mini-Neptunes** ( $1.5 < R < 4 R_{\oplus}$ ):

$$M = R^{2.5}$$

**Reason:** Significant volatile envelope (H/He), less compression.

**Gas Giants** ( $R > 4 R_{\oplus}$ ):

$$M = \text{constant} \times R^2$$

**Reason:** Adding more mass barely increases radius!

Extra mass compresses the interior. Jupiter and Saturn have similar radii despite Saturn being 1/3 Jupiter's mass.

**Extreme case:** Some "super-Jupiters" are smaller than Jupiter despite being  $10\times$  more massive!

## 9.8 Atmospheric Escape

**Why can some planets hold atmospheres?**

**Escape velocity** formula:

$$v_{\text{escape}} = \sqrt{2GM/R}$$

Where:

- $G$  = gravitational constant
- $M$  = planet mass
- $R$  = planet radius

**For atmosphere to stay:**

- Molecular speeds must be much less than  $v_{\text{escape}}$
- Temperature determines molecular speeds

**Rule of thumb:**  $v_{\text{thermal}} < v_{\text{escape}} / 6$

**Thermal velocity** (root-mean-square):

$$v_{\text{thermal}} = \sqrt{3kT/m}$$

Where:

- $k$  = Boltzmann constant
- $T$  = temperature
- $m$  = molecular mass

**Examples:**

**Earth** ( $v_{\text{escape}} = 11.2 \text{ km/s}$ ):

- $\text{H}_2$  at 288 K:  $v = 1.9 \text{ km/s} \rightarrow$  ESCAPES over time
- $\text{N}_2$  at 288 K:  $v = 0.5 \text{ km/s} \rightarrow$  RETAINED
- This is why Earth has no hydrogen!

**Mars** ( $v_{\text{escape}} = 5.0 \text{ km/s}$ ):

- Lower gravity + solar wind  $\rightarrow$  lost most atmosphere

**Titan** (Saturn's moon,  $v_{\text{escape}} = 2.6 \text{ km/s}$ ):

- Cold (94 K)  $\rightarrow$  molecular speeds slow  $\rightarrow$  thick atmosphere despite low gravity!

**Hot Jupiters:**

- High temperature  $\rightarrow$  losing atmosphere rapidly
- Some may evaporate completely

---

## Section 10: Advanced SQL Concepts

### 10.1 Query Execution Order

SQL is written in this order:

sql

```
SELECT column
FROM table
JOIN other_table
WHERE condition
GROUP BY column
HAVING group_condition
ORDER BY column
LIMIT number
```

**But executed in this order:**

```
1. FROM      -- Get tables
2. JOIN      -- Combine tables
3. WHERE     -- Filter rows
4. GROUP BY  -- Create groups
5. HAVING    -- Filter groups
6. SELECT    -- Choose columns
7. ORDER BY  -- Sort results
8. LIMIT     -- Restrict number of rows
```

**Why this matters:**

**This WORKS:**

```
sql

SELECT star_id, COUNT(*) as num_planets
FROM EXOPLANETS
GROUP BY star_id
HAVING num_planets > 5 -- Can use alias in HAVING
```

**This FAILS:**

```
sql

SELECT star_id, COUNT(*) as num_planets
FROM EXOPLANETS
WHERE num_planets > 5 -- ERROR! num_planets doesn't exist yet
GROUP BY star_id
```

**Fix:**

```
sql
```

```
SELECT star_id, COUNT(*) as num_planets
FROM EXOPLANETS
GROUP BY star_id
HAVING COUNT(*) > 5  -- Use actual expression, not alias
```

## 10.2 NULL Handling

### What is NULL?

Not zero, not empty string, but "unknown" or "missing".

### NULL behavior:

```
sql

NULL = NULL  -- FALSE! (not even equal to itself)
NULL != NULL -- Also FALSE!
NULL + 5      -- NULL (NULL poisons calculations)
NULL AND TRUE -- NULL
NULL OR TRUE  -- TRUE
```

### Correct NULL checks:

```
sql

WHERE column IS NULL    -- ✓ Correct
WHERE column IS NOT NULL -- ✓ Correct
WHERE column = NULL     -- ✗ Always FALSE!
```

### COALESCE (NULL replacement):

```
sql

SELECT COALESCE(end_date, 'Ongoing') as mission_status
FROM DISCOVERY_MISSIONS
```

Returns first non-NULL value:

- If end\_date is '2018-10-30' → returns '2018-10-30'
- If end\_date is NULL → returns 'Ongoing'

## 10.3 DISTINCT vs GROUP BY

**DISTINCT** (remove duplicates):

```
sql
```

```
SELECT DISTINCT spectral_type  
FROM STAR_SYSTEMS
```

Result:

```
spectral_type  
G  
K  
M
```

**GROUP BY** (with aggregation):

```
sql
```

```
SELECT spectral_type, COUNT(*) as count  
FROM STAR_SYSTEMS  
GROUP BY spectral_type
```

Result:

```
spectral_type | count  
G             | 23  
K             | 38  
M             | 68
```

**When to use which:**

- **DISTINCT:** Just need unique values
- **GROUP BY:** Need to calculate something per group

## 10.4 Self-Joins

**What if you want to compare rows within the same table?**

**Example:** Find planets in the same star system:

```
sql
```

```
SELECT
  e1.name as planet1,
  e2.name as planet2
FROM EXOPLANETS e1
JOIN EXOPLANETS e2 ON e1.star_id = e2.star_id
WHERE e1.planet_id < e2.planet_id -- Avoid duplicates
```

### Visual:

star\_id=5 has: Planet A, Planet B, Planet C

Pairs generated:

A - B

A - C

B - C

(Without WHERE condition, would also get B-A, C-A, C-B, and A-A, B-B, C-C)

## 10.5 Index Strategies

### When to create indexes:

1. ✓ Foreign key columns
2. ✓ Columns in WHERE clauses
3. ✓ Columns in JOIN conditions
4. ✓ Columns in ORDER BY

### When NOT to:

1. ✗ Small tables (< 1000 rows)
2. ✗ Columns with few unique values (e.g., boolean)
3. ✗ Frequently updated columns (indexes slow down INSERT/UPDATE)

### Composite index:

sql

```
CREATE INDEX idx_planet_search ON EXOPLANETS(planet_type, mass_earth)
```

### Good for:

```
sql
```

```
WHERE planet_type = 'Terrestrial' AND mass_earth < 2
```

**Not good for:**

```
sql
```

```
WHERE mass_earth < 2 -- Doesn't use first column of index
```

**Index order matters!** Put most selective column first.

---

## Section 11: Pandas Deep Dive

### 11.1 DataFrame Basics

```
python
```

```
df = pd.read_sql_query(query, conn)
```

**DataFrame structure:**

Index	name	mass_earth	radius_earth
0	Earth Twin	1.0	1.0
1	Super Mars	3.5	1.8
2	Mini Neptune	15.0	3.2

**Accessing data:**

```
python
```

```
df['name']          # Column as Series
df[['name', 'mass_earth']] # Multiple columns as DataFrame
df.loc[0]           # Row by index label
df.iloc[0]          # Row by position
df.loc[0, 'name']    # Specific cell
```

### 11.2 Common Operations

**Filtering:**

```
python
```

```
df[df['mass_earth'] > 10] # Planets heavier than 10 Earth masses
```

### Step-by-step:

1. `df['mass_earth'] > 10` creates boolean Series: [False, False, True, ...]
2. `df[...]` selects only True rows

### Sorting:

```
python  
  
df.sort_values('esi_score', ascending=False)
```

### Adding columns:

```
python  
  
df['density'] = df['mass_earth'] / (df['radius_earth'] ** 3)
```

### Apply function to column:

```
python  
  
df['temp_celsius'] = df['temp_k'].apply(lambda x: x - 273.15)
```

## 11.3 Styling DataFrames

```
python  
  
df.style.background_gradient(subset=['esi_score'], cmap='RdYlGn')
```

### Creates color-coded table:

- Low ESI: Red
- Medium ESI: Yellow
- High ESI: Green

### Other styling:

```
python
```

```
.format({'esi': '{:.4f}', 'mass': '{:.2f}'}) # Number formatting
.highlight_max(color='lightgreen')         # Highlight max value
.highlight_null(null_color='red')          # Show missing data
```

## Section 12: NumPy Explained

### 12.1 Arrays vs Lists

**Python list:**

```
python

my_list = [1, 2, 3, 4]
result = [x * 2 for x in my_list] # Loop required
```

**NumPy array:**

```
python

my_array = np.array([1, 2, 3, 4])
result = my_array * 2          # Vectorized!
# Result: array([2, 4, 6, 8])
```

**Speed difference:** NumPy is 10-100× faster for large arrays!

### 12.2 Random Distributions

**Uniform** (flat distribution):

```
python

random.uniform(0, 1) # Equal chance of any value between 0 and 1
```

**Normal/Gaussian:**

```
python

random.gauss(mean, std_dev)
```

Bell curve centered at mean:

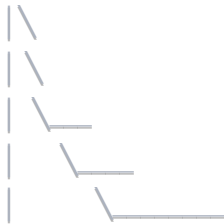


## Exponential:

python

```
random.expovariate(lambda)
```

Most values small, few values large:

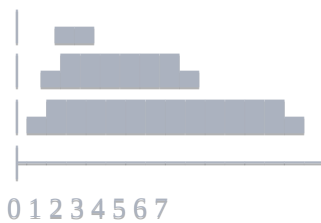


## Poisson:

python

```
np.random.poisson(lambda)
```

Discrete counts (0, 1, 2, 3...):



## Section 13: Common Pitfalls & Debugging

### 13.1 SQL Errors

#### IntegrityError:

python

```
sqlite3.IntegrityError: UNIQUE constraint failed: EXOPLANETS.name
```

**Cause:** Trying to insert duplicate planet name

**Fix:** Use try-except or check before inserting

**OperationalError:**

```
python
```

```
sqlite3.OperationalError: no such column: planet_name
```

**Cause:** Typo in column name or table doesn't exist

**Fix:** Check spelling, ensure table created

**DataError:**

```
python
```

```
sqlite3.DataError: Cannot convert string to float
```

**Cause:** Wrong data type (inserting "abc" into REAL column)

**Fix:** Validate data before inserting

## 13.2 Pandas Errors

**KeyError:**

```
python
```

```
KeyError: 'planet_name'
```

**Cause:** Column doesn't exist in DataFrame

**Fix:** Check column names with `df.columns`

**ValueError:**

```
python
```

```
ValueError: Length of values does not match length of index
```

**Cause:** Trying to add column with wrong number of rows

**Fix:** Ensure lengths match

## 13.3 Matplotlib Errors

### No plot showing:

```
python  
  
plt.plot([1, 2, 3]) # Nothing appears!
```

**Cause:** Forgot `plt.show()`

**Fix:** Add `plt.show()` at end

**In Jupyter:** Use `%matplotlib inline` magic command

---

## Section 14: Best Practices

### 14.1 Database Design

#### Do:

- ✓ Use meaningful column names
- ✓ Add indexes on foreign keys
- ✓ Use appropriate data types
- ✓ Normalize data (avoid redundancy)
- ✓ Document schema

#### Don't:

- ✗ Use spaces in column names
- ✗ Store arrays/lists as strings
- ✗ Use TEXT for everything
- ✗ Forget foreign key constraints

### 14.2 SQL Queries

#### Do:

- ✓ Use parameterized queries (prevent SQL injection)
- ✓ Test with LIMIT first
- ✓ Use meaningful aliases
- ✓ Comment complex queries

- ✓ Use EXPLAIN to check performance

**Don't:**

- ✗ SELECT \* in production (specify columns)
- ✗ Use string concatenation for queries
- ✗ Forget WHERE clause (scan entire table)
- ✗ Nest too deeply (hard to read)

## 14.3 Python Code

**Do:**

- ✓ Use descriptive variable names
- ✓ Add docstrings to functions
- ✓ Handle exceptions
- ✓ Use type hints (optional but helpful)
- ✓ Break long functions into smaller ones

**Don't:**

- ✗ Use global variables
- ✗ Ignore warnings
- ✗ Hardcode values (use constants)
- ✗ Leave TODO comments (finish or remove)

---

## Section 15: Extensions & Ideas

### 15.1 How to Extend This Project

**Easy additions:**

1. Add more planets per star
2. Include moons (satellites table)
3. Add binary star systems
4. Track telescope observation time
5. Store images/spectra as BLOBs

### **Medium difficulty:**

1. Implement user accounts (authentication)
2. Add real-time data from NASA APIs
3. Create publication/citation tracking
4. Build machine learning habitability predictor
5. Add 3D orbital simulations

### **Advanced:**

1. Distributed database (multiple servers)
2. Real-time collaboration (multiple users)
3. Integration with astronomical software (FITS files)
4. Statistical analysis of planet populations
5. N-body simulations for system stability

## **15.2 Real NASA Data Sources**

### **Where to get actual exoplanet data:**

#### **1. NASA Exoplanet Archive**

- URL: <https://exoplanetarchive.ipac.caltech.edu/>
- API available
- Thousands of confirmed planets

#### **2. SIMBAD Astronomical Database**

- Star data
- TAP (Table Access Protocol) queries

#### **3. VizieR**

- Catalog service
- Download tables in various formats

### **Example API call:**

```
python
```

```
import requests
```

```
url = "https://exoplanetarchive.ipac.caltech.edu/TAP/sync?query=select+*+from+ps+where+disc_year=2020&format=json"  
response = requests.get(url)  
data = response.json()
```

## 15.3 Machine Learning Applications

### Habitability prediction:

```
python  
  
from sklearn.ensemble import RandomForestClassifier  
  
X = df[['mass_earth', 'radius_earth', 'temp_k', 'star_temp']]  
y = df['is_habitable']  
  
model = RandomForestClassifier()  
model.fit(X, y)  
  
prediction = model.predict([[1.2, 1.1, 280, 5500]])
```

**Transit signal detection:** Use CNNs (Convolutional Neural Networks) to identify planet transits in light curves.

**Atmospheric composition from spectra:** Train models to predict molecules from transmission spectra.

---

## Section 16: Mathematical Formulas Summary

### All Key Formulas at a Glance

#### Habitable Zone:

$$s_{\text{eff}} = a + b \times T + c \times T^2$$
$$r_{\text{hz}} = \sqrt{(L_{\text{star}} / s_{\text{eff}})}$$

#### Earth Similarity Index:

$$\text{ESI}_{\text{component}} = (1 - |x - x_{\text{ref}}| / |x + x_{\text{ref}}|)^{\text{weight}}$$
$$\text{ESI} = (\text{ESI}_r \times \text{ESI}_m \times \text{ESI}_t \times \text{ESI}_v)^{(1/4)}$$

#### Equilibrium Temperature:

$$T_{eq} = T_{star} \times (1 - A)^{(1/4)} \times \sqrt{(R_{star} / 2a)}$$

### Kepler's Third Law:

$$P^2 = a^3 / M$$

$$P = \sqrt{(a^3 / M)}$$

### Escape Velocity:

$$v_{esc} = \sqrt{(2GM / R)}$$

### Surface Gravity:

$$g = GM / R^2$$

$$g_{relative} = M / R^2 \text{ (relative to Earth)}$$

### Density:

$$\rho = M / V$$

$$\rho = M / (4/3 \times \pi \times R^3)$$

### Orbital Speed:

$$v = 2\pi a / P$$

### Stefan-Boltzmann Law:

$$L = 4\pi R^2 \times \sigma \times T^4$$

### Inverse Square Law (flux):

$$F = L / (4\pi d^2)$$

---

## Section 17: Quick Reference

### SQL Commands Cheat Sheet

sql

-- *SELECT basics*

**SELECT** column1, column2 **FROM** table;

**SELECT \* FROM** table **WHERE** condition;

**SELECT DISTINCT** column **FROM** table;

-- *Aggregations*

**COUNT**(\*), **COUNT**(**DISTINCT** column)

**SUM**(column), **AVG**(column)

**MIN**(column), **MAX**(column)

-- *Joins*

**INNER JOIN** table2 **ON** condition

**LEFT JOIN** table2 **ON** condition

**RIGHT JOIN** table2 **ON** condition

-- *Grouping*

**GROUP BY** column

**HAVING** aggregate\_condition

-- *Sorting & Limiting*

**ORDER BY** column **ASC**|**DESC**

**LIMIT** number

-- *Subqueries*

**WHERE** column **IN** (**SELECT** ...)

**WHERE EXISTS** (**SELECT** ...)

-- *Window Functions*

**RANK**() **OVER** (**ORDER BY** column)

**ROW\_NUMBER**() **OVER** (**PARTITION BY** column **ORDER BY** column)

**SUM**(column) **OVER** (**ORDER BY** column) -- *Running total*

### Python Quick Reference

python

### *# Lists*

```
my_list = [1, 2, 3]
my_list.append(4)
my_list[0] # Access first element
```

### *# Dictionaries*

```
my_dict = {'key': 'value'}
my_dict['key']
my_dict.get('key', default_value)
```

### *# Loops*

```
for item in my_list:
    print(item)

for key, value in my_dict.items():
    print(f"{key}: {value}")
```

### *# List comprehension*

```
squares = [x**2 for x in range(10)]
```

### *# F-strings*

```
name = "Earth"
print(f"Planet: {name}")
```

### *# Try-except*

```
try:
    risky_operation()
except Exception as e:
    print(f"Error: {e}")
```

## **Pandas Cheat Sheet**

python

```
# Reading data
df = pd.read_sql_query(query, conn)
df = pd.read_csv('file.csv')

# Viewing data
df.head()    # First 5 rows
df.info()    # Column types and counts
df.describe() # Statistics

# Selecting
df['column']
df[['col1', 'col2']]
df.loc[row_index]
df.iloc[position]

# Filtering
df[df['column'] > 10]
df[df['column'].isin(['value1', 'value2'])]

# Sorting
df.sort_values('column', ascending=False)

# Grouping
df.groupby('column').mean()
df.groupby('column').agg({'col1': 'sum', 'col2': 'mean'})

# Adding columns
df['new_col'] = df['col1'] + df['col2']
df['new_col'] = df['col1'].apply(lambda x: x * 2)
```

---

## Conclusion

You now understand: ✓ Database design and relationships

- ✓ SQL queries from basic to advanced
- ✓ Astronomical formulas and physics
- ✓ Python data manipulation
- ✓ Creating visualizations
- ✓ Building interactive dashboards

### Next steps:

1. Experiment with the code

2. Modify queries and see results
3. Try creating new visualizations
4. Add your own features
5. Load real NASA data!

**Resources for further learning:**

- SQL: SQLBolt, Mode Analytics SQL Tutorial
- Pandas: "Python for Data Analysis" by Wes McKinney
- Astronomy: "Exoplanets" by Sara Seager
- Databases: "Database System Concepts" by Silberschatz

Remember: The best way to learn is by doing. Break things, fix them, and learn from errors!