KendraGraph Learning Guide

Space Science + Python for Orbital Risk Intelligence

Author's Note: This guide teaches you exactly what you need to build and understand orbital collision risk systems. No PhD required, no wasted theory. Learn by doing.

Timeline: 8 weeks to solid foundation
Daily Commitment: 1-2 hours
Philosophy: Learn what you need, when you need it

Table of Contents

PART 1: Space & Orbit Foundations

Chapter 1: Orbital Mechanics Basics

The Core Concept

Satellites don't fight gravity - they fall around Earth.

Imagine throwing a ball:

- Throw softly → falls to ground nearby

- Throw harder → lands farther away
- Throw at 7.8 km/s → curves around Earth before landing
- Throw at 7.8 km/s at right angle to gravity → never lands (orbit!)

Why This Matters for Collision Risk

```
Lower orbit = Faster speed = More satellites = More risk
Upper orbit = Slower speed = Fewer satellites = Less risk

LEO (Low Earth Orbit):
- Altitude: 200-2,000 km
- Speed: 7-8 km/s
- Period: 90-120 minutes
- Crowded! (Starlink, ISS, most satellites)

MEO (Medium Earth Orbit):
- Altitude: 2,000-35,786 km
- Speed: 3-7 km/s
- GPS satellites live here

GEO (Geostationary):
- Altitude: 35,786 km
- Speed: 3.07 km/s
- Period: 24 hours (stays over same spot)
- Communication satellites
```

The Only Equation You Really Need

```
Orbital Period: T = 2π √(a³/μ)

Where:
- T = time for one complete orbit (seconds)
- a = semi-major axis (orbit radius in km)
- μ = 398,600 km³/s² (Earth's gravitational parameter)
- π = 3.14159...

Example: ISS at 400km altitude
a = Earth radius (6,371) + altitude (400) = 6,771 km
T = 2π √(6771³/398600) = 5,561 seconds = 92.7 minutes
```

Practical Python Implementation

```
python
```

```python
import numpy as np

def orbital_period_minutes(altitude_km):
    """Calculate orbital period given altitude above Earth"""
    earth_radius = 6371  # km
    mu = 398600  # km³/s²

    a = earth_radius + altitude_km
    period_seconds = 2 * np.pi * np.sqrt(a**3 / mu)
    period_minutes = period_seconds / 60

    return period_minutes

# Test it
print(f"ISS (400km): {orbital_period_minutes(400):.1f} minutes")
print(f"Starlink (550km): {orbital_period_minutes(550):.1f} minutes")
print(f"GPS (20,200km): {orbital_period_minutes(20200):.1f} minutes")
```

Output:

```
ISS (400km): 92.7 minutes
Starlink (550km): 95.8 minutes
GPS (20,200km): 718.3 minutes (11.97 hours)
```

Key Insights for Your Project

1. **Similar altitudes = collision risk zone**
    - Starlink at 550km passes near other 550km satellites constantly
    - Natural separation between LEO and GEO
2. **Orbital decay matters**
    - Atmospheric drag at <600km
    - Satellites slowly lose altitude
    - Lower altitude = faster speed = phasing changes
3. **Don't need perfect physics**
    - SGP4 library handles complicated orbital mechanics
    - You focus on geometry and risk scoring

What to Skip

- ❌ Deriving Kepler's laws

- ❌ Hohmann transfer orbits
- ❌ Lagrange points
- ❌ Three-body problem
- ❌ Orbital perturbation math

Chapter 2: Two-Line Elements (TLE)

What is a TLE?

A standardized text format for describing satellite orbits. Every satellite tracked by NORAD has one.

Real Example (ISS):

```
ISS (ZARYA)
1 25544U 98067A   23365.50000000  .00016717  00000-0  10270-3 0  9005
2 25544  51.6400 247.4627 0006703 130.5360 325.0288 15.72125391414289
```

The Fields That Matter

Line 0: Name

```
ISS (ZARYA)
```

Human-readable name (not used by computers)

Line 1: Identification & Epoch

```
1 25544U 98067A   23365.50000000  .00016717  00000-0  10270-3 0  9005
  ^^^^^           ^^^^^^^^^^^^^^
  NORAD ID        Epoch (when measured)
```

- **25544**: Unique satellite ID (NORAD catalog number)
- **23365.50000000**: Year 2023, day 365.5 (Dec 31, noon UTC)

Line 2: Orbital Elements

```
2 25544  51.6400 247.4627 0006703 130.5360 325.0288 15.72125391414289
         ^^^^^^^^ ^^^^^^^^ ^^^^^^^ ^^^^^^^^ ^^^^^^^^ ^^^^^^^^
         Incl     RAAN     Ecc     ArgPer   MeanAnom MeanMotion
```

- **51.6400°**: Inclination (orbit tilt)

- **0006703**: Eccentricity × 10^7 (0.0006703 - nearly circular)
- **15.72125391**: Mean motion (orbits per day)

Python: Parse TLE

Never parse manually. Use libraries:

```python
from skyfield.api import EarthSatellite, load

# TLE text (get from Space-Track.org or CelesTrak)
name = "ISS (ZARYA)"
line1 = "1 25544U 98067A   23365.50000000  .00016717  00000-0  10270-3 0  9005"
line2 = "2 25544  51.6400 247.4627 0006703 130.5360 325.0288 15.72125391414289"

# Create satellite object
satellite = EarthSatellite(line1, line2, name)

# Extract useful info
print(f"Name: {satellite.name}")
print(f"NORAD ID: {satellite.model.satnum}")
print(f"Epoch: {satellite.epoch.utc_iso()}")
print(f"Inclination: {satellite.model.inclo * 57.2958:.2f}°")
print(f"Mean Motion: {satellite.model.no_kozai / (2*3.14159) * 1440:.2f} orbits/day")
```

TLE Freshness - CRITICAL

TLEs expire because:

- Atmospheric drag changes orbit
- Satellites maneuver
- Measurement errors accumulate

Check TLE age:

```python
```

```python
from datetime import datetime, timezone

def tle_age_hours(satellite):
    """How old is this TLE?"""
    epoch = satellite.epoch.utc_datetime()
    now = datetime.now(timezone.utc)
    age_hours = (now - epoch).total_seconds() / 3600
    return age_hours

def tle_confidence(age_hours):
    """Confidence in TLE accuracy"""
    if age_hours < 24:
        return "HIGH (< 1 day old)"
    elif age_hours < 72:
        return "MEDIUM (1-3 days old)"
    elif age_hours < 168:
        return "LOW (3-7 days old)"
    else:
        return "VERY LOW (> 1 week old)"

age = tle_age_hours(satellite)
print(f"TLE age: {age:.1f} hours")
print(f"Confidence: {tle_confidence(age)}")
```

Rule of thumb:

- < 24 hours: Good for collision prediction
- 24-72 hours: Acceptable with increased uncertainty
- > 72 hours: High uncertainty, refresh TLE
- > 1 week: Don't use for collision analysis

Getting TLEs

Free sources:

1. **Space-Track.org** (Best, requires free account)

python

```python
```

```
import requests

# Login and get session
login_url = "https://www.space-track.org/ajaxauth/login"
query_url = "https://www.space-track.org/basicspacedata/query/class/gp/NORAD_CAT_ID/25544/orderby/EPOCH%20des

session = requests.Session()
session.post(login_url, data={'identity': 'YOUR_EMAIL', 'password': 'YOUR_PASSWORD'})

response = session.get(query_url)
tle_text = response.text
```

2. **CelesTrak** (Public, no login, but less frequent updates)

```python
import requests

url = "https://celestrak.org/NORAD/elements/gp.php?GROUP=stations&FORMAT=tle"
response = requests.get(url)
tles = response.text.split('\n')
```

What to Skip

- ❌ TLE checksum validation (library does it)
- ❌ Manual coordinate conversions
- ❌ Understanding every field in detail
- ❌ Historical TLE formats (pre-2000)

Chapter 3: Orbital Elements Deep Dive

The 6 Classical Orbital Elements

Every orbit is defined by exactly 6 numbers. Think of them as GPS coordinates for space.

1. Semi-Major Axis (a) - Size

What it is: Average distance from Earth's center

```
Small a = low orbit = fast satellite
Large a = high orbit = slow satellite

Examples:
ISS: a = 6,771 km (400 km altitude)
Starlink: a = 6,921 km (550 km altitude)
GPS: a = 26,560 km (20,189 km altitude)
GEO: a = 42,164 km (35,786 km altitude)
```

Why it matters:

- Groups satellites by altitude
- Similar `a` = potential collision zone
- Different `a` = natural separation

Python:

```python
def altitude_from_period(period_minutes):
    """Calculate altitude given orbital period"""
    mu = 398600  # km³/s²
    period_sec = period_minutes * 60
    a = (mu * (period_sec / (2 * np.pi))**2)**(1/3)
    altitude = a - 6371  # subtract Earth radius
    return altitude

# ISS orbits every 92 minutes
print(f"Altitude: {altitude_from_period(92):.0f} km")
```

2. Eccentricity (e) - Shape

What it is: How "stretched" the orbit is

```
e = 0: Perfect circle
e = 0.01: Slightly elliptical (most satellites)
e = 0.5: Very elliptical (Molniya orbit)
e = 1.0: Parabola (escape trajectory)

Most LEO satellites: e < 0.01 (treat as circular)
```

Why it matters:

- Low e (circular) = predictable paths

- High e (elliptical) = varying speed and altitude
- For collision risk: usually ignore unless e > 0.1

Practical implication:

```python
def is_circular(eccentricity, threshold=0.01):
    """Check if orbit is approximately circular"""
    return eccentricity < threshold

# ISS eccentricity = 0.0006703
if is_circular(0.0006703):
    print("Treat as circular orbit for collision analysis")
```

3. Inclination (i) - Tilt

What it is: Angle between orbit plane and equator

```
i = 0°: Equatorial (along equator)
i = 28.5°: Cape Canaveral launches (Florida latitude)
i = 51.6°: ISS (Baikonur, Kazakhstan)
i = 90°: Polar (passes over both poles)
i = 98°: Sun-synchronous (always same local time)
i > 90°: Retrograde (orbits "backwards")
```

Why it REALLY matters:

Collision risk is highest when inclinations match:

```
Satellite A: i = 51.6°
Satellite B: i = 51.5°
→ They cross paths twice per orbit!

Satellite A: i = 51.6°
Satellite C: i = 98.0°
→ Paths cross at steep angles, brief encounters
```

Python: Inclination similarity check

```python

```

```python
def inclination_risk_group(inclination):
    """Group satellites by inclination for risk analysis"""
    if inclination < 30:
        return "EQUATORIAL"
    elif 45 <= inclination <= 55:
        return "ISS_ZONE"  # Very crowded!
    elif inclination > 85:
        return "POLAR"
    else:
        return "MID_INCLINATION"

def collision_risk_from_inclination(i1, i2):
    """Higher risk if inclinations are similar"""
    diff = abs(i1 - i2)

    if diff < 5:
        return "HIGH (similar orbital planes)"
    elif diff < 30:
        return "MEDIUM (crossing orbits)"
    else:
        return "LOW (different planes)"

# Example
print(collision_risk_from_inclination(51.6, 52.1))  # HIGH
print(collision_risk_from_inclination(51.6, 98.0))  # MEDIUM
```

4. RAAN (Ω) - Right Ascension of Ascending Node

What it is: Longitude where orbit crosses equator going northward

```
Think: "Where does the orbit cross the equator?"

RAAN = 0°: Crosses at prime meridian
RAAN = 90°: Crosses at 90°E longitude
RAAN = 180°: Crosses at 180°E
RAAN = 270°: Crosses at 90°W
```

Why it matters:

- Changes slowly over time (precession from Earth's bulge)
- Satellites with similar RAAN are in same "longitude slice"
- Sun-synchronous orbits: RAAN precesses 1°/day to track sun

Key insight for collision risk:

```
Similar inclination + similar RAAN = HIGH RISK
(orbits nearly overlap)

Similar inclination + different RAAN = MEDIUM RISK
(orbits in same plane but shifted)
```

Python:

```python
def raan_difference(raan1, raan2):
    """Calculate angular difference between RAANs"""
    diff = abs(raan1 - raan2)
    # Handle wrap-around (350° and 10° are 20° apart)
    if diff > 180:
        diff = 360 - diff
    return diff

def combined_orbital_similarity(i1, i2, raan1, raan2):
    """Combined risk score from inclination and RAAN"""
    inc_diff = abs(i1 - i2)
    raan_diff = raan_difference(raan1, raan2)

    # Both must be similar for high risk
    if inc_diff < 5 and raan_diff < 30:
        return "VERY HIGH RISK (nearly identical orbits)"
    elif inc_diff < 10 and raan_diff < 60:
        return "HIGH RISK (similar orbital plane)"
    else:
        return "MODERATE/LOW RISK"

# Starlink constellation example
print(combined_orbital_similarity(53.0, 53.2, 100, 110))
```

5. Argument of Perigee (ω) - Orientation of Ellipse

What it is: Where in the orbit the satellite is closest to Earth

For circular orbits (e ≈ 0): IGNORE THIS

- Most collision analysis ignores ω because orbits are nearly circular

For elliptical orbits:

When it matters:

- Highly elliptical orbits (e > 0.1)
- Molniya orbits (Russian communication satellites)
- Transfer orbits

Skip for now unless working with HEO (Highly Elliptical Orbits)

6. Mean Anomaly (M) - Position in Orbit

What it is: Where the satellite is right now in its orbit

```
M = 0°: At perigee (closest to Earth)
M = 90°: 1/4 through orbit
M = 180°: At apogee (farthest from Earth)
M = 270°: 3/4 through orbit
M = 360°/0°: Back at perigee
```

Why it matters:

- Combined with epoch, tells you exact satellite position
- Changes quickly (completes 0-360° each orbit)
- For collision analysis: SGP4 handles this automatically

Practical use:

```
python
```

```
# You never manually use mean anomaly
# SGP4 library converts M → true position

from skyfield.api import EarthSatellite, load

ts = load.timescale()
t = ts.now()

# This internally uses mean anomaly + orbital elements
position = satellite.at(t)
x, y, z = position.position.km

print(f"Position: ({x:.0f}, {y:.0f}, {z:.0f}) km")
```

Summary Table

Element	Symbol	What It Controls	Collision Relevance
Semi-Major Axis	a	Orbit size & speed	**CRITICAL** - groups satellites
Eccentricity	e	Orbit shape	Low (most circular)
Inclination	i	Orbit tilt	**CRITICAL** - crossing geometry
RAAN	Ω	Equator crossing longitude	**HIGH** - orbital plane location
Argument of Perigee	ω	Ellipse orientation	Low (for circular orbits)
Mean Anomaly	M	Current position	Handled by SGP4

Practical Risk Grouping

python

```python
def orbital_risk_group(a, e, i, raan):
    """Create risk group ID for similar orbits"""
    # Round to bins
    altitude_bin = round(a - 6371, -1)  # nearest 10km
    inclination_bin = round(i, 0)       # nearest degree
    raan_bin = round(raan / 30) * 30    # 30° bins

    group_id = f"ALT{altitude_bin}_INC{inclination_bin}_RAAN{raan_bin}"
    return group_id

# Examples
print(orbital_risk_group(6771, 0.001, 51.6, 247))
# Output: ALT400_INC52_RAAN240

print(orbital_risk_group(6921, 0.001, 53.0, 105))
# Output: ALT550_INC53_RAAN120

# Satellites in same group have higher collision risk
```

What to Skip

- ❌ Converting between different anomaly types (mean, true, eccentric)
- ❌ Orbital element perturbation equations
- ❌ Coordinate system transformations (TEME, J2000, etc.)
- ❌ Historical epoch systems

Chapter 4: Coordinate Frames (ECI vs ECEF)

Why Multiple Coordinate Systems?

Problem: Earth rotates, but satellites orbit in (mostly) fixed planes.

Two perspectives:

1. **Inertial (space-fixed):** Ignores Earth's rotation - good for orbital mechanics
2. **Earth-fixed:** Rotates with Earth - good for ground locations

ECI - Earth-Centered Inertial

What it is: Origin at Earth's center, axes fixed relative to stars

```
Imagine freezing Earth's rotation:
- X-axis points toward vernal equinox (fixed star direction)
- Z-axis points to North Pole
- Y-axis completes right-hand system

Satellites orbit in (nearly) fixed ECI planes
Earth rotates underneath them
```

When to use:

- Calculating satellite positions
- Computing relative distances between satellites
- Orbital mechanics calculations

Python example:

```python
from skyfield.api import EarthSatellite, load

ts = load.timescale()
t = ts.now()

# Get position in ECI (TEME frame, similar to ECI)
position = satellite.at(t)
x, y, z = position.position.km

print(f"ECI Position: ({x:.0f}, {y:.0f}, {z:.0f}) km")
# Example: (3421, -5102, 2814) km
```

ECEF - Earth-Centered Earth-Fixed

What it is: Origin at Earth's center, axes rotate with Earth

```
Axes fixed to Earth's surface:
- X-axis points to 0° longitude (prime meridian) on equator
- Z-axis points to North Pole
- Y-axis points to 90°E on equator

Ground locations have fixed ECEF coordinates
Satellites move in ECEF as Earth rotates
```

When to use:

- Ground station locations

- Sub-satellite points (lat/lon below satellite)
- Visualizing on map/globe

Python example:

```python
# Get geographic location (converts ECI → ECEF → Lat/Lon)
geographic = position.subpoint()

print(f"Latitude: {geographic.latitude.degrees:.2f}°")
print(f"Longitude: {geographic.longitude.degrees:.2f}°")
print(f"Altitude: {geographic.elevation.km:.0f} km")
# Example: Lat: 28.5°, Lon: -80.1°, Alt: 420 km
```

For Collision Analysis: Use ECI

Critical rule: Always compute distances in ECI, never mix frames

```python
# CORRECT: Both positions in same frame (ECI)
pos1_eci = sat1.at(t).position.km
pos2_eci = sat2.at(t).position.km
distance = np.linalg.norm(pos1_eci - pos2_eci)

# WRONG: Mixing frames
pos1_eci = sat1.at(t).position.km
pos2_latlon = sat2.at(t).subpoint()  # Different frame!
# distance = ???  # This makes no sense!
```

Practical Code for Your Project

```python
```

```python
import numpy as np
from skyfield.api import EarthSatellite, load

def compute_distance_km(sat1, sat2, time):
    """Compute 3D distance between two satellites"""
    # Both in ECI frame automatically
    pos1 = sat1.at(time).position.km
    pos2 = sat2.at(time).position.km

    # Euclidean distance in 3D
    distance = np.linalg.norm(pos1 - pos2)
    return distance

def get_ground_track(satellite, time):
    """Get lat/lon for visualization"""
    pos = satellite.at(time)
    geo = pos.subpoint()

    return {
        'lat': geo.latitude.degrees,
        'lon': geo.longitude.degrees,
        'alt': geo.elevation.km
    }

# Usage
ts = load.timescale()
t = ts.now()

# Collision analysis: ECI distances
dist = compute_distance_km(sat1, sat2, t)
print(f"Separation: {dist:.2f} km")

# Visualization: ground tracks
track1 = get_ground_track(sat1, t)
track2 = get_ground_track(sat2, t)
print(f"Sat1 over: {track1['lat']:.1f}°, {track1['lon']:.1f}°")
```

Common Mistake to Avoid

```
python
```

```
# BAD: Computing distance from lat/lon
lat1, lon1 = 40.7, -74.0  # New York
lat2, lon2 = 34.0, -118.2  # Los Angeles

# This is ground distance, NOT satellite distance!
# Satellites at same lat/lon can be 1000 km apart vertically!

# GOOD: Always use 3D ECI positions
pos1_eci = [x1, y1, z1]
pos2_eci = [x2, y2, z2]
distance_3d = sqrt((x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2)
```

What to Skip

- ❌ TEME vs J2000 vs GCRF frame differences
- ❌ Precession and nutation corrections
- ❌ Manual frame transformation matrices
- ❌ Geodetic vs geocentric coordinates

Chapter 5: SGP4 Propagation Model

What is SGP4?

Simplified General Perturbations 4 - The standard algorithm for predicting satellite positions from TLEs.

Think of it as: GPS navigation for satellites

- Input: TLE + time
- Output: Position & velocity

Why "Simplified"?

- Doesn't model every tiny force
- Good enough for most collision analysis
- Fast to compute (thousands of satellites/second)

What SGP4 Accounts For

```
✅ Earth's gravity (main force)
✅ Earth's oblateness (J2 perturbation - equatorial bulge)
✅ Atmospheric drag (approximate model)
✅ Solar radiation pressure (approximate)
```

✅ Third-body effects (sun/moon gravity) (approximate)

❌ NOT modeled in detail:
- Solar weather variations
- Satellite maneuvers
- High-precision drag
- Relativistic effects

When SGP4 Breaks Down

Good for:
- LEO satellites (200-2000 km)
- Time spans < 7 days from TLE epoch
- Circular/near-circular orbits

Degraded accuracy:
- Very low altitude (< 200 km) - drag dominates
- Highly elliptical orbits
- > 7 days from TLE epoch
- After satellite maneuvers

Python: Using SGP4

Method 1: Direct sgp4 library

python

```python
from sgp4.api import Satellite, jday
from datetime import datetime

# Parse TLE
line1 = "1 25544U 98067A   23365.50000000 ..."
line2 = "2 25544  51.6400 247.4627 ..."

satellite = Satellite(line1, line2)

# Propagate to specific time
year = 2023
month = 12
day = 31
hour = 12
minute = 0
second = 0

jd, fr = jday(year, month, day, hour, minute, second)
error_code, position, velocity = satellite.sgp4(jd, fr)

if error_code == 0:
    print(f"Position: {position} km")
    print(f"Velocity: {velocity} km/s")
else:
    print(f"Error: {error_code}")
```

Method 2: Skyfield (easier, recommended)

```python
```

```python
from skyfield.api import EarthSatellite, load

ts = load.timescale()

# Create satellite
satellite = EarthSatellite(line1, line2, "ISS")

# Propagate to now
t = ts.now()
position = satellite.at(t)

# Get position (ECI frame)
x, y, z = position.position.km
vx, vy, vz = position.velocity.km_per_s

print(f"Position: ({x:.0f}, {y:.0f}, {z:.0f}) km")
print(f"Velocity: ({vx:.2f}, {vy:.2f}, {vz:.2f}) km/s")
```

Propagation Over Time Window

For collision analysis, check multiple time steps:

```python
```

```python
import numpy as np

def propagate_window(satellite, start_time, duration_hours, step_minutes=10):
    """
    Propagate satellite over time window

    Returns:
        times: array of time objects
        positions: array of (x,y,z) positions
        velocities: array of (vx,vy,vz) velocities
    """
    ts = load.timescale()

    # Generate time array
    num_steps = int(duration_hours * 60 / step_minutes)
    times = [start_time + i * step_minutes / (24 * 60) for i in range(num_steps)]

    positions = []
    velocities = []

    for t in times:
        pos = satellite.at(ts.from_datetime(t))
        positions.append(pos.position.km)
        velocities.append(pos.velocity.km_per_s)

    return times, np.array(positions), np.array(velocities)

# Example: Propagate ISS for 24 hours in 10-minute steps
from datetime import datetime, timezone

start = datetime.now(timezone.utc)
times, positions, velocities = propagate_window(satellite, start, 24, 10)

print(f"Propagated {len(times)} time steps")
print(f"First position: {positions[0]}")
print(f"Last position: {positions[-1]}")
```

Checking Propagation Errors

```
python
```

```python
def check_sgp4_health(satellite, time):
    """Check if SGP4 propagation succeeded"""
    try:
        pos = satellite.at(time)
        position = pos.position.km

        # Check for NaN or unrealistic values
        if np.any(np.isnan(position)):
            return False, "NaN in position"

        distance_from_
```