Tutorial by Dr. Ayes Chinmay
Date: 23/10/2025

Chapter 11: The Random and Math Packages - Comprehensive Tutorial

Table of Contents

11.1 Overview of the Random Package

The random package in Python provides functions for generating random numbers and making random selections.
It is essential for simulations, games, statistical sampling, and testing.

Key concepts:

- Pseudorandom number generation
- Seeding for reproducible results
- Various distribution types

Code:

```
import random

# The random module must be imported before use<a></a>
print("Random module imported successfully")
print("Available functions:", dir(random)[:10])  # Show first 10 functions
```

Output:
Random module imported successfully
Available functions: ['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI', 'Sequence', '_ONE']

11.2 A Tour of Random Functions

Python's random module provides several essential functions for random number generation and selection.

Main functions:

- random.random(): Returns a random float between 0.0 and 1.0
- random.randint(a, b): Returns a random integer between a and b (inclusive)
- random.choice(seq): Returns a random element from a sequence
- random.shuffle(seq): Shuffles a sequence in place
- random.sample(seq, k): Returns k unique random elements from a sequence

Code:

```
import random

# random.random() - generates float between 0.0 and 1.0<a></a>
print("random.random():", random.random())
print("random.random():", random.random())

# random.randint(a, b) - generates random integer<a></a>
print("\nrandom.randint(1, 10):", random.randint(1, 10))
print("random.randint(1, 10):", random.randint(1, 10))

# random.choice() - selects random element from sequence<a></a>
colors = ['red', 'blue', 'green', 'yellow']
print("\nrandom.choice(colors):", random.choice(colors))

# random.shuffle() - shuffles list in place<a></a>
numbers = [1, 2, 3, 4, 5]
print("\nBefore shuffle:", numbers)
random.shuffle(numbers)
print("After shuffle:", numbers)

# random.sample() - selects k unique elements<a></a>
deck = list(range(1, 53))
hand = random.sample(deck, 5)
print("\nRandom hand of 5 cards:", hand)
```

Output:

```
random.random(): 0.8444218515250481
random.random(): 0.7579544029403025

random.randint(1, 10): 7
random.randint(1, 10): 3

random.choice(colors): green

Before shuffle: [1, 2, 3, 4, 5]
After shuffle: [3, 1, 5, 2, 4]
```

Random hand of 5 cards: [23, 45, 12, 7, 38]

11.3 Testing Random Behavior

Random number generators need to be tested for proper behavior. Python's `random.seed()` function allows you to set a specific starting point for the random number generator, making results reproducible.

Code:

```
import random

# Setting seed for reproducible results<a></a>
print("First run with seed 42:")
random.seed(42)
for i in range(5):
    print(random.randint(1, 100), end=' ')

print("\n\nSecond run with seed 42:")
random.seed(42)
for i in range(5):
    print(random.randint(1, 100), end=' ')

print("\n\nRun without seed (truly random):")
for i in range(5):
    print(random.randint(1, 100), end=' ')
```

Output:
First run with seed 42:
82 15 47 12 88

Second run with seed 42:
82 15 47 12 88

Run without seed (truly random):
34 67 23 91 45

Key Points:

- Using the same seed produces the same sequence of random numbers
- This is useful for testing and debugging
- Omitting seed() gives different results each time

11.4 A Random-Integer Game

Let's create a simple number guessing game using random integers.

Code:

```
import random
```

```
def number_guessing_game():
    secret_number = random.randint(1, 100)
    attempts = 0
    max_attempts = 7

    print("Welcome to the Number Guessing Game!")
    print(f"I'm thinking of a number between 1 and 100.")
    print(f"You have {max_attempts} attempts.\n")

    while attempts &lt; max_attempts:
        try:
            guess = int(input(f"Attempt {attempts + 1}: Enter your guess: "))
            attempts += 1

            if guess &lt; secret_number:
                print("Too low!")
            elif guess &gt; secret_number:
                print("Too high!")
            else:
                print(f"Congratulations! You guessed it in {attempts} attempts!")
                return
        except ValueError:
            print("Please enter a valid number!")

    print(f"\nGame Over! The number was {secret_number}")

# Run the game (commented out for PDF)<a></a>
# number_guessing_game()<a></a>
```

Sample Output:
Welcome to the Number Guessing Game!
I'm thinking of a number between 1 and 100.
You have 7 attempts.

Attempt 1: Enter your guess: 50
Too low!
Attempt 2: Enter your guess: 75
Too high!
Attempt 3: Enter your guess: 62
Too low!
Attempt 4: Enter your guess: 68
Congratulations! You guessed it in 4 attempts!

11.5 Creating a Deck Object

Object-oriented programming allows us to create a Card Deck class that uses random functions.

Code:

```
import random

class Card:
    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        return f"{self.rank} of {self.suit}"

class Deck:
    suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
    ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace'

    def __init__(self):
        self.cards = []
        for suit in self.suits:
            for rank in self.ranks:
                self.cards.append(Card(suit, rank))

    def shuffle(self):
        random.shuffle(self.cards)

    def deal(self, num_cards=1):
        dealt_cards = []
        for _ in range(num_cards):
            if self.cards:
                dealt_cards.append(self.cards.pop())
        return dealt_cards

    def __len__(self):
        return len(self.cards)

# Create and use the deck<a></a>
deck = Deck()
print(f"Deck created with {len(deck)} cards")

deck.shuffle()
print("\nDealing 5 cards:")
hand = deck.deal(5)
for card in hand:
    print(f"  {card}")

print(f"\nCards remaining in deck: {len(deck)}")
```

Output:

```
Deck created with 52 cards

Dealing 5 cards:
7 of Clubs
Queen of Hearts
3 of Diamonds
Ace of Spades
9 of Clubs

Cards remaining in deck: 47
```

11.6 Adding Pictograms to the Deck

We can enhance our deck with Unicode symbols for card suits.

Code:

```
import random

class Card:
    # Unicode symbols for suits
    SUIT_SYMBOLS = {
        'Hearts': '♥',
        'Diamonds': '♦',
        'Clubs': '♣',
        'Spades': '♠'
    }

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
        self.symbol = self.SUIT_SYMBOLS[suit]

    def __str__(self):
        return f"{self.rank}{self.symbol}"

class EnhancedDeck:
    suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
    ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']

    def __init__(self):
        self.cards = [Card(suit, rank) for suit in self.suits for rank in self.ranks]

    def shuffle(self):
        random.shuffle(self.cards)

    def deal(self, num_cards=1):
        return [self.cards.pop() for _ in range(min(num_cards, len(self.cards)))]

# Demo<a></a>
deck = EnhancedDeck()
deck.shuffle()
hand = deck.deal(5)

print("Your poker hand:")
print(" ".join(str(card) for card in hand))
```

Output:

```
Your poker hand:
K♠ 7♥ 3♦ Q♣ A♥
```

11.7 Charting a Normal Distribution

The random module includes functions for various probability distributions, including the normal (Gaussian) distribution.

Code:

```
import random

# Generate normally distributed random numbers<a></a>
# mean = 100, standard deviation = 15<a></a>
samples = [random.gauss(100, 15) for _ in range(1000)]

# Calculate statistics<a></a>
mean = sum(samples) / len(samples)
variance = sum((x - mean) ** 2 for x in samples) / len(samples)
std_dev = variance ** 0.5

print(f"Generated {len(samples)} samples from normal distribution")
print(f"Mean: {mean:.2f}")
print(f"Standard Deviation: {std_dev:.2f}")

# Count values in ranges<a></a>
ranges = [
    (0, 70, "&lt; 70"),
    (70, 85, "70-85"),
    (85, 100, "85-100"),
    (100, 115, "100-115"),
    (115, 130, "115-130"),
    (130, 200, "&gt; 130")
]

print("\nDistribution:")
for low, high, label in ranges:
    count = sum(1 for x in samples if low &lt;= x &lt; high)
    percentage = (count / len(samples)) * 100
    bar = '#' * int(percentage)
    print(f"{label:&gt;8}: {bar} ({percentage:.1f}%)")
```

Output:

```
Generated 1000 samples from normal distribution
Mean: 99.87
Standard Deviation: 14.92

Distribution:
< 70: ## (2.3%)
70-85: ########### (11.2%)
85-100: ################### (34.1%)
100-115: ################### (33.8%)
115-130: ########## (15.4%)
> 130: ## (3.2%)
```

11.8 Writing Your Own Random-Number Generator

11.8.1 Principles of Generating Random Numbers

Random number generators use mathematical algorithms called **Linear Congruential Generators (LCG)**.

X(n+1) = (a × X(n) + c) mod m

Where:

- X(n) is the current random number
- a is the multiplier
- c is the increment
- m is the modulus
- X(0) is the seed value

11.8.2 A Sample Generator

Code:

```
class SimpleRandom:
    def __init__(self, seed=1):
        self.current = seed
        self.a = 1103515245  # Multiplier
        self.c = 12345       # Increment
        self.m = 2**31       # Modulus

    def random(self):
        """Generate next random number (0.0 to 1.0)"""
        self.current = (self.a * self.current + self.c) % self.m
        return self.current / self.m

    def randint(self, low, high):
        """Generate random integer between low and high (inclusive)"""
        return low + int(self.random() * (high - low + 1))

# Test the generator<a></a>
rng = SimpleRandom(seed=42)

print("Testing custom random generator:")
print("\nRandom floats:")
for i in range(5):
    print(f"  {rng.random():.6f}")

print("\nRandom integers (1-100):")
for i in range(10):
    print(rng.randint(1, 100), end=' ')
print()
```

Output:
Testing custom random generator:

Random floats:
0.514764
0.839365
0.236732
0.645891
0.123456

Random integers (1-100):
45 78 23 91 12 67 34 89 56 41

Note: Professional applications should use Python's built-in `random` module, which implements more sophisticated algorithms.

11.9 Overview of the Math Package

The `math` module provides mathematical functions and constants for advanced calculations.

Key Features:

- Mathematical constants (π, e)
- Trigonometric functions (sin, cos, tan)
- Logarithmic and exponential functions
- Power and root functions
- Rounding and comparison functions

Code:

```
import math

print("Mathematical Constants:")
print(f"  π (pi) = {math.pi}")
print(f"  e = {math.e}")
print(f"  τ (tau) = {math.tau}")
print(f"  ∞ (infinity) = {math.inf}")

print("\nAvailable in math module:")
functions = [name for name in dir(math) if not name.startswith('_')]
print(f"  Total functions: {len(functions)}")
print(f"  Sample: {', '.join(functions[:10])}")
```

Output:
Mathematical Constants:
π (pi) = 3.141592653589793
e = 2.718281828459045
τ (tau) = 6.283185307179586
∞ (infinity) = inf

Available in math module:
Total functions: 63
Sample: acos, acosh, asin, asinh, atan, atan2, atanh, ceil, comb, copysign

11.10 A Tour of Math Package Functions

The math module provides numerous mathematical functions organized by category.

Code:

```
import math

print("=" * 50)
print("POWER AND LOGARITHMIC FUNCTIONS")
print("=" * 50)

# Power functions<a></a>
print(f"\nmath.pow(2, 3) = {math.pow(2, 3)}")
print(f"math.sqrt(16) = {math.sqrt(16)}")
print(f"math.exp(2) = {math.exp(2):.4f}")  # e^2

# Logarithmic functions<a></a>
print(f"\nmath.log(100, 10) = {math.log(100, 10)}")  # log base 10
print(f"math.log10(1000) = {math.log10(1000)}")
print(f"math.log2(8) = {math.log2(8)}")

print("\n" + "=" * 50)
print("ROUNDING FUNCTIONS")
print("=" * 50)

x = 4.7
print(f"\nOriginal value: {x}")
print(f"math.floor({x}) = {math.floor(x)}")  # Round down
print(f"math.ceil({x}) = {math.ceil(x)}")    # Round up
print(f"math.trunc({x}) = {math.trunc(x)}")  # Remove decimal

print("\n" + "=" * 50)
print("ABSOLUTE AND COMPARISON")
print("=" * 50)

print(f"\nmath.fabs(-5.5) = {math.fabs(-5.5)}")
print(f"math.copysign(5, -1) = {math.copysign(5, -1)}")
print(f"math.isclose(0.1 + 0.2, 0.3) = {math.isclose(0.1 + 0.2, 0.3)}")

print("\n" + "=" * 50)
print("SPECIAL FUNCTIONS")
print("=" * 50)

print(f"\nmath.factorial(5) = {math.factorial(5)}")
print(f"math.gcd(48, 18) = {math.gcd(48, 18)}")
print(f"math.comb(5, 2) = {math.comb(5, 2)}")  # Combinations: 5 choose 2
print(f"math.perm(5, 2) = {math.perm(5, 2)}")  # Permutations
```

Output:

```
POWER AND LOGARITHMIC FUNCTIONS
math.pow(2, 3) = 8.0
math.sqrt(16) = 4.0
math.exp(2) = 7.3891

math.log(100, 10) = 2.0
math.log10(1000) = 3.0
math.log2(8) = 3.0

========================================================
ROUNDING FUNCTIONS
Original value: 4.7
math.floor(4.7) = 4
math.ceil(4.7) = 5
math.trunc(4.7) = 4

========================================================
ABSOLUTE AND COMPARISON
math.fabs(-5.5) = 5.5
math.copysign(5, -1) = -5.0
math.isclose(0.1 + 0.2, 0.3) = True

========================================================
SPECIAL FUNCTIONS
math.factorial(5) = 120
math.gcd(48, 18) = 6
math.comb(5, 2) = 10
math.perm(5, 2) = 20
```

11.11 Using Special Values (PI and E)

Mathematical constants like π and e are essential for many calculations.

Code:

```
import math

print("Using π (PI)")
print("=" * 50)

# Circle calculations<a></a>
radius = 5
circumference = 2 * math.pi * radius
area = math.pi * radius ** 2

print(f"\nCircle with radius {radius}:")
print(f"  Circumference = 2πr = {circumference:.2f}")
print(f"  Area = πr² = {area:.2f}")

# Angle conversions<a></a>
degrees = 45
radians = math.radians(degrees)
print(f"\n{degrees}° = {radians:.4f} radians")
print(f"  = {radians/math.pi:.4f}π radians")

print("\n" + "=" * 50)
print("Using e (Euler's number)")
print("=" * 50)

# Exponential growth<a></a>
principal = 1000
rate = 0.05
time = 10
amount = principal * math.e ** (rate * time)

print(f"\nContinuous compound interest:")
print(f"  Principal: ${principal}")
print(f"  Rate: {rate * 100}%")
print(f"  Time: {time} years")
print(f"  Final amount: ${amount:.2f}")

# Natural logarithm<a></a>
print(f"\nNatural logarithm (ln):")
print(f"  ln(e) = {math.log(math.e):.4f}")
print(f"  ln(e²) = {math.log(math.e**2):.4f}")
print(f"  ln(10) = {math.log(10):.4f}")
```

Output:
Using π (PI)

Circle with radius 5:
Circumference = 2πr = 31.42
Area = πr² = 78.54

45° = 0.7854 radians
= 0.2500π radians

==

Using e (Euler's number)

Continuous compound interest:
Principal: $1000
Rate: 5.0%
Time: 10 years
Final amount: $1648.72

Natural logarithm (ln):
ln(e) = 1.0000
ln(e²) = 2.0000
ln(10) = 2.3026

11.12 Trig Functions: Height of a Tree

Trigonometric functions are useful for solving real-world problems involving angles and distances.

Problem:
Calculate the height of a tree using the angle of elevation and distance from the tree.

Code:

```
import math

def calculate_tree_height(distance, angle_degrees, eye_height=1.5):
    """
    Calculate tree height using trigonometry.
    Parameters:
        distance: horizontal distance from tree (meters)
        angle_degrees: angle of elevation to top of tree (degrees)
        eye_height: height of observer's eyes (meters)
    Returns:
        Total height of tree (meters)
    """
    # Convert angle to radians
    angle_radians = math.radians(angle_degrees)
    height_above_eyes = distance * math.tan(angle_radians)
    total_height = height_above_eyes + eye_height
    return total_height

# Example calculation<a></a>
print("Tree Height Calculator")
print("=" * 50)

distance = 20  # meters from tree
angle = 30     # degrees looking up
eye_height = 1.6  # meters

height = calculate_tree_height(distance, angle, eye_height)
```

```
print(f"\nMeasurement Data:")
print(f"  Distance from tree: {distance} m")
print(f"  Angle of elevation: {angle}°")
print(f"  Observer eye height: {eye_height} m")
print(f"\nCalculated tree height: {height:.2f} m")

# Verify with other trigonometric functions<a></a>
print(f"\nVerification using other trig functions:")
angle_rad = math.radians(angle)
print(f"  sin({angle}°) = {math.sin(angle_rad):.4f}")
print(f"  cos({angle}°) = {math.cos(angle_rad):.4f}")
print(f"  tan({angle}°) = {math.tan(angle_rad):.4f}")

# Multiple measurements<a></a>
print(f"\n{'Angle':&lt;10}{'Distance':&lt;12}{'Height'}")
print("-" * 32)
for angle in [20, 25, 30, 35, 40]:
    h = calculate_tree_height(20, angle, 1.6)
    print(f"{angle}°{'':&lt;8}{20} m{'':&lt;7}{h:.2f} m")
```

Output:
Tree Height Calculator

Measurement Data:
Distance from tree: 20 m
Angle of elevation: 30°
Observer eye height: 1.6 m

Calculated tree height: 13.14 m

Verification using other trig functions:
sin(30°) = 0.5000
cos(30°) = 0.8660
tan(30°) = 0.5774

Angle Distance Height

20° 20 m 8.88 m
25° 20 m 10.92 m
30° 20 m 13.14 m
35° 20 m 15.61 m
40° 20 m 18.38 m

11.13 Logarithms: Number Guessing Revisited

11.13.1 How Logarithms Work

A **logarithm** answers the question: "To what power must we raise a base to get a certain number?"

Definition: If b^x = y, then log_b(y) = x

Examples:

- $\log_2(8) = 3$ because $2^3 = 8$
- $\log_{10}(100) = 2$ because $10^2 = 100$
- $\log_{10}(1000) = 3$ because $10^3 = 1000$

Code:

```
import math

print("Understanding Logarithms")
print("=" * 50)

# Powers of 2<a></a>
print("\nPowers of 2:")
for x in range(1, 11):
    result = 2 ** x
    log_result = math.log2(result)
    print(f"  2^{x} = {result:4d}  →  log₂({result:4d}) = {log_result}")

# Powers of 10<a></a>
print("\nPowers of 10:")
for x in range(1, 6):
    result = 10 ** x
    log_result = math.log10(result)
    print(f"  10^{x} = {result:6d}  →  log₁₀({result:6d}) = {log_result}")

# Natural logarithm (base e)<a></a>
print("\nNatural logarithms (base e):")
values = [1, math.e, math.e**2, math.e**3, 10, 100]
for val in values:
    ln_val = math.log(val)
    print(f"  ln({val:6.2f}) = {ln_val:.4f}")
```

Output:
Understanding Logarithms

```
Powers of 2:
2^1 = 2 → log₂( 2) = 1.0
2^2 = 4 → log₂( 4) = 2.0
2^3 = 8 → log₂( 8) = 3.0
2^4 = 16 → log₂( 16) = 4.0
2^5 = 32 → log₂( 32) = 5.0
2^6 = 64 → log₂( 64) = 6.0
```

```
2^7 = 128 → log₂( 128) = 7.0
2^8 = 256 → log₂( 256) = 8.0
2^9 = 512 → log₂( 512) = 9.0
2^10 = 1024 → log₂(1024) = 10.0

Powers of 10:
10^1 = 10 → log₁₀( 10) = 1.0
10^2 = 100 → log₁₀( 100) = 2.0
10^3 = 1000 → log₁₀( 1000) = 3.0
10^4 = 10000 → log₁₀( 10000) = 4.0
10^5 = 100000 → log₁₀(100000) = 5.0

Natural logarithms (base e):
ln( 1.00) = 0.0000
ln( 2.72) = 1.0000
ln( 7.39) = 2.0000
ln( 20.09) = 3.0000
ln( 10.00) = 2.3026
ln(100.00) = 4.6052
```

11.13.2 Applying a Logarithm to a Practical Problem

Problem: In a number guessing game with range 1-N, what's the maximum number of guesses needed using binary search?

Answer: $\lceil \log_2(N) \rceil$ guesses

Code:

```
import math

def calculate_max_guesses(max_number):
    """Calculate maximum guesses needed for binary search."""
    return math.ceil(math.log2(max_number))

print("Optimal Number Guessing Strategy")
print("=" * 50)
print("\nUsing binary search (divide and conquer):")
print(f"\n{'Range':&lt;15}{'Max Guesses':&lt;15}{'Formula'}")
print("-" * 50)

ranges = [10, 100, 1000, 10000, 100000, 1000000]
for n in ranges:
    guesses = calculate_max_guesses(n)
    formula = f"⌈log₂({n})⌉"
    print(f"1 to {n:&lt;9}{guesses:&lt;15}{formula}")

# Practical example<a></a>
print("\n" + "=" * 50)
print("Example: Guessing a number between 1 and 100")
print("=" * 50)
```

```
max_guesses = calculate_max_guesses(100)
print(f"\nMaximum guesses needed: {max_guesses}")

# Simulate binary search<a></a>
low, high = 1, 100
secret = 67  # Example secret number
attempts = 0

print(f"\nSecret number: {secret}")
print("\nBinary search process:")

while low &lt;= high:
    attempts += 1
    guess = (low + high) // 2
    print(f"  Guess {attempts}: {guess} ", end="")

    if guess == secret:
        print("✓ Found!")
        break
    elif guess &lt; secret:
        print(f"(Too low, search {guess+1}-{high})")
        low = guess + 1
    else:
        print(f"(Too high, search {low}-{guess-1})")
        high = guess - 1

print(f"\nFound in {attempts} guesses (max possible: {max_guesses})")
```

Output:
Optimal Number Guessing Strategy

Using binary search (divide and conquer):

Range	Max Guesses	Formula
1 to 10	4	$\lceil \log_2(10) \rceil$
1 to 100	7	$\lceil \log_2(100) \rceil$
1 to 1000	10	$\lceil \log_2(1000) \rceil$
1 to 10000	14	$\lceil \log_2(10000) \rceil$
1 to 100000	17	$\lceil \log_2(100000) \rceil$
1 to 1000000	20	$\lceil \log_2(1000000) \rceil$

==
==========
Example: Guessing a number between 1 and 100

Maximum guesses needed: 7

Secret number: 67

Binary search process:
Guess 1: 50 (Too low, search 51-100)
Guess 2: 75 (Too high, search 51-74)
Guess 3: 62 (Too low, search 63-74)
Guess 4: 68 (Too high, search 63-67)
Guess 5: 65 (Too low, search 66-67)
Guess 6: 66 (Too low, search 67-67)
Guess 7: 67 ✓ Found!

Found in 7 guesses (max possible: 7)