Lecture 30: Math Functions (Part 2)

Continuing with built-in math functions.
Deep dive into pow() and sum().

Agenda

The pow() function
Two-argument pow() vs **
Three-argument pow() (Modular Exponentiation)
The sum() function
sum() start parameter
Floating point precision with sum
Summary

The pow() Function

Returns base to the power of exp.
Signature: ‘pow(base, exp, mod=None)‘.

pow() vs ** Operator

For two arguments, ‘pow(a, b)‘ is equivalent to ‘a ** b‘.
However, ‘pow()‘ is a function and can be passed as an argument to higher-order functions like ‘map()‘.

Three-Argument pow()

‘pow(base, exp, mod)‘ computes ‘(base ** exp) % mod‘.
It is FAR more efficient than doing the exponentiation and then modulo, especially for large numbers.
Essential in cryptography (e.g., RSA).

The sum() Function

Sums the items of an iterable from left to right and returns the total.
Signature: ‘sum(iterable, start=0)‘.

The start Parameter in sum()

The ‘start‘ value is added to the total of the items.
Useful if you want to start counting from a specific baseline.

sum() with Non-Numbers

‘sum()‘ is optimized for numbers. Using it to concatenate strings will raise a TypeError.
For strings, use ‘”.join(strings)‘ instead.
You CAN use it to flatten a list of lists, but ‘itertools.chain‘ is better.

Precision Issues with sum()

When summing many floating-point numbers, precision can be lost.
For exact floating-point summation, use ‘math.fsum()‘ instead of built-in ‘sum()‘.

Performance Characteristics

Both ‘pow()‘ and ‘sum()‘ are implemented in C and are highly optimized. Always prefer ‘sum(list)‘ over a manual ‘for‘ loop accumulating a total.

Summary

‘pow()‘ with three arguments is crucial for modular arithmetic.
‘sum()‘ is efficient but watch out for float precision and don’t use it for strings.

Next Lecture

In the final lecture of this unit, we will explore external Modules: math, random, and statistics.