Built-In Functions, Function Definition and Call in Python

Write once, use many times with functions!

Built-in functions are ready to use. Functions help you organize code and avoid repetition.
Define a function, then call it whenever you need it.

1. Built-In Functions in Python

- Python provides many built-in functions (predefined).
- They are ready to use, no need to define them.
- Used for common tasks like input, output, type conversion, maths, etc.

Category	Built-in Functions (Examples)	Purpose
Input/Output	`print()`, `input()`	Display output / take input
Type Conversion	`int()`; `float()`; `str()`; `bool()`	Convert data type
Mathematical	`len()`, `abs()`, `round()`; `max()`, `min()`, `sum()`	Perform mathematical operations
String	`upper()`, `lower()`, `split()`, `strip()`	Work with strings
List	`list()`, `sorted()`, `append()`, `remove()`	Work with lists
Others	`range()`, `open()`, `type()`	Other useful operations

Example: Some Built-in Functions

```
print("Hello, Python!")                  # Output
a = int(input("Enter a number: "))       # Input & type conversion
print(len("Python"))                     # Output: 6
print(max(5, 8, 2))                      # Output: 8
```

Output

```
Hello, Python!
(user input)
6
8
```

2. Function Definition

- A function is a block of code that performs a specific task.
- You define a function using the keyword `def`, a name, parameters (optional) and a return statement (optional).

Syntax

```
def function_name(parameters):
    # function body (statements)
    return value    # optional
```

Example: Define a Function

```
1 def add(a, b):
2     result = a + b
3     return result
```

Function name · Parameters · Function body · Return statement

What does it do?

- This function takes two numbers (a and b), adds them, and returns the result.
- The code inside the function body runs when the function is called.

3. Function Call

- To use a function, you call it by using its name followed by brackets ().
- You can pass values (arguments) to the function.
- The function executes and returns a result (if any).

Syntax

```
function_name(arguments)
```

Example: Call the Function

```
1 def add(a, b):
2     return a + b
3
4
4 x = add(5, 3)   # call the function
5 print(x)
```

Output

```
8
```

Function with No Parameters

```
def greet():
    print("Hello!")

greet()    # call
```

Output

```
Hello!
```

Function with Multiple Parameters

```
def multiply(a, b):
    return a * b

print(multiply(4, 6))
```

Output

```
24
```

4. Key Points to Remember

1. Built-in functions are ready to use (e.g., print(), len(), int().
2. Use `def` to define your own function.
3. Place code inside the function body (indented).
4. Use parentheses () to call a function and pass arguments.
5. Functions can return a value using `return` (optional).
6. A function can have parameters and/or no parameters.

Flow: Define and Call a Function

Write function with `def` → (Optional) pass arguments → Call the function by name → Get output (if any)

Functions make your code clean, reusable and easier to maintain!

Scope and Lifetime of Variables in Python

Same variable, different scope... different lifetime!

Scope tells us where a variable can be accessed. Lifetime tells us how long a variable exists in memory.

1. Scope of Variables

- Scope defines the region of a program where a variable can be accessed or used.
- In Python, there are mainly three types of scopes:

Local Scope
Variables inside a function (only accessible within the function).

Enclosing Scope
Variables in an outer function (accessible to inner functions).

Global Scope
Variables defined outside all functions (accessible throughout the program).

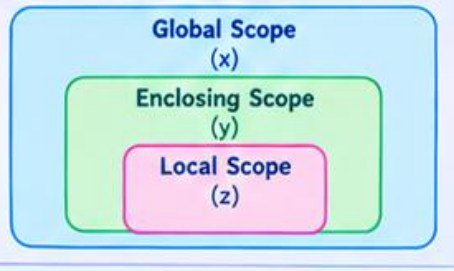
Types of Scope with Example

```
1   x = "global"            # global scope
2
3   def outer():            # enclosing scope
4       y = "enclosing"
5
6       def inner():        # local scope
7           z = "local"
8           print(x)        # can access global
9           print(y)        # can access enclosing
10          print(z)        # local variable
11
12      inner()
13
14  outer()
```

Output

```
global
enclosing
local
```

Scope Hierarchy



Key Points

- A variable is accessible in its current scope and inner scopes (if applicable).
- Not accessible outside its scope.
- You can use the same name in different scopes (shadowing).

2. Lifetime of Variables

- Lifetime refers to the period during which a variable exists in memory.
- In Python, variables are created when a value is assigned and are destroyed when they are no longer needed.
- For local variables, the lifetime is limited to the function call.
- For global variables, the lifetime is the entire program execution.

Lifetime in Different Scopes

Local Variable Lifetime

Exists only during the function call.

```
1  def greet():
2      msg = "Hello" # local
3      print(msg)
4
5  greet()
6  print(msg)   # Error!
```

msg is not defined outside the function.

Global Variable Lifetime

Exists throughout the program.

```
1  msg = "Python"  # global
2  def show():
3      print(msg)
4
5  show()
6  print(msg)   # Works!
```

msg is accessible everywhere in the program.

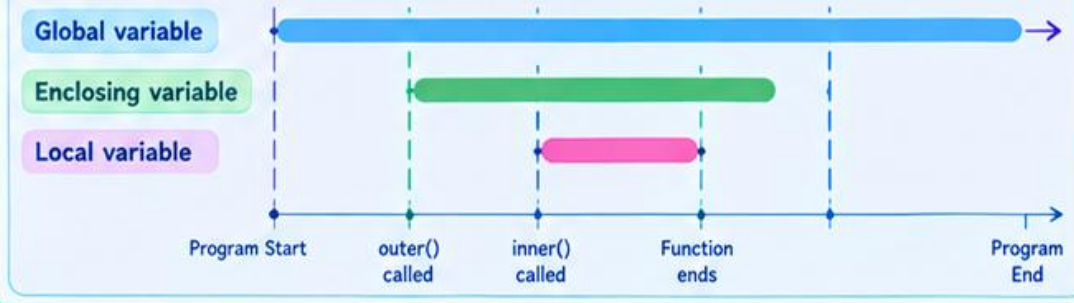
Nested Function Lifetime

Exists while the outer function is running (and inner function is called).

```
1  def outer():
2      x = 10    # enclosing
3      def inner():
4          print(x)  # local (inner)
5      inner()
6
7  outer()
```

x is not accessible outside outer().

Visual Representation of Lifetime



Quick Examples

1. Local scope

```
def add(a, b):
    result = a + b
    return result

print(add(5, 3))
```

result is local to add().

2. Enclosing scope

```
def outer():
    x = 10
    def inner():
        print(x)
    inner()
outer()
```

x is accessible in inner().

3. Global scope

```
x = 5
def show():
    print(x)

show()
```

x is accessible everywhere.

Key Takeaways

1. Scope tells where a variable can be accessed.
2. Lifetime tells how long a variable exists in memory.
3. Local variables → function only.
4. Enclosing variables → inner functions.
5. Global variables → entire program.
6. Use scope wisely to avoid bugs and unexpected behavior!

Remember

Scope = Where can I use it?

Lifetime = How long does it exist?

Right scope...
Right lifetime...
Better code!

Default Parameters, Command Line Arguments in Python

Make your functions flexible with default values • Get inputs from the command line for dynamic programs!

Same Program... Different Inputs... More Flexibility!

1. Default Parameters

- Default parameters are values assigned to function parameters in the function definition.
- If the caller does not provide a value, the default value is used.
- They make functions more flexible and easier to use.

Syntax

```
def function_name(param1, param2 = default_value, ...):
    # function body
```

Example: Function with Default Parameters

```
1  def greet(name, msg="Hello"):
2      print(f"{msg}, {name}!")
3
4  greet("Alice")                # Uses default message
5  greet("Bob", "Good morning")  # Uses custom message
```

Output

```
Hello, Alice!

Good morning, Bob!
```

Example: Multiple Default Parameters

```
1  def calculate(a, b, op="+"):
2      if op == "+":
3          return a + b
4      elif op == "-":
5          return a - b
6      else:
7          return "Invalid operation"
8
9  print(calculate(5, 3))        # 8
10 print(calculate(5, 3, "-"))   # 2
```

Output

```
8
2
```

Key Points

- ✓ Default values are used when an argument is not passed.
- ✓ They are set in the function definition.
- ✓ Useful when some parameters are optional.
- ✓ Be careful with mutable default values (like lists, dicts)!

2. Command Line Arguments

- Command line arguments are values passed to a Python script when it is run from the terminal or command prompt.
- They are accessed using the sys module.
- It helps to make programs interactive and flexible.

Syntax

```
import sys

# Access arguments
sys.argv[0]  # script name
sys.argv[1]  # first argument
sys.argv[2]  # second argument ...
```

Run from terminal

```
$ python script.py arg1 arg2
```

↑ Arguments passed from command line

Example: Sum of Two Numbers (using command line arguments)

```
1  import sys
2
3  if len(sys.argv) != 3:
4      print("Usage: python script.py <num1> <num2>")
5  else:
6      num1 = int(sys.argv[1])
7      num2 = int(sys.argv[2])
8      print(f"Sum: {num1 + num2}")
```

Output

Run with arguments:

```
$ python script.py 5 7
```

Output:

```
Sum: 12
```

Run without enough arguments:

```
$ python script.py 5
```

Output:

```
Usage: python script.py
       <num1> <num2>
```

Accessing Arguments as a List

```
args = sys.argv
print(args)
```

Output

```
['script.py', '5', '7']
```

Key Points

- ✓ Use **import sys** to work with command line arguments.
- ✓ sys.argv is a list of arguments (including the script name).
- ✓ Convert arguments to the required type (e.g., int, float).
- ✓ Always check the number of arguments to avoid errors.

Quick Comparison

Feature	Default Parameters	Command Line Arguments
Purpose	Provides default values for function parameters	Gets input when script is run from terminal
Where used	In function definition	In main program (using sys.argv)
Access	By calling the function	From sys.argv list
Example	def func(a=10)	python script.py 10

Remember

- ✓ Default parameters = Optional values for functions.
- ✓ Command line arguments = Input from terminal when running the script.
- ✓ Both make your Python programs more flexible and user-friendly!

Write flexible code... use Default Parameters and Command Line Arguments!

More Inputs = More Possibilities!

Lambda Functions, Assert Statement, Importing User Defined Module in Python

Learn Apply Build Better Code!

Small features... Big impact! Write concise code, validate your data, and reuse your own modules!

1. Lambda Functions

- Lambda functions are small, anonymous functions.
- They are defined using the lambda keyword.
- They can have any number of arguments but only one expression.
- They are often used for short operations and in functions like map(), filter() and sorted().

Syntax

```
lambda arguments: expression
```

Example

```
# Lambda function to add 10 to a number
add_ten = lambda x: x + 10
print(add_ten(5))    # Output: 15

# Using lambda with map()
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x*x, numbers))
print(squared)    # Output: [1, 4, 9, 16]
```

Flow

Arguments (input) → Expression (computation) → Return value

Output:

```
15
[1, 4, 9, 16]
```

Lambda is a single-expression function, not a full function block.

2. Assert Statement

- The assert statement is used for debugging.
- It checks whether a condition is True.
- If the condition is False, it raises an AssertionError.
- It helps catch errors early in the program.

Syntax

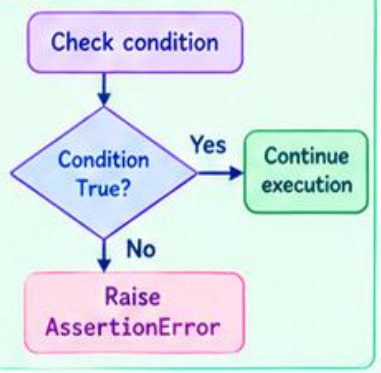
```
assert condition  # optional message
```

Example

```
age = 18
assert age >= 18, "Age must be at least 18"
print("Valid age!")

# If age = 16, it will raise AssertionError
# AssertionError: Age must be at least 18
```

How it works



Output (when age = 18):

```
Valid age!
```

Error (when age = 16):

```
AssertionError:
Age must be at least 18
```

Use assert for conditions that should always be true.

3. Importing User Defined Module in Python

- A module is a Python file containing functions, variables, or classes.
- You can create your own module and import it into another file.
- This helps in code organization and reusability.

Steps

1. Create a Python file (e.g., mymodule.py) with functions or variables.
2. Use import statement in your main program to access them.
3. Call the functions or use the variables.

Example: 1. Create a module (mymodule.py)

```
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

def add(a, b):
    return a + b
```

This file is saved as **mymodule.py**

Example: 2. Use the module in another file

```
# main.py
import mymodule      ← import the module

print(mymodule.greet("Alice"))
print(mymodule.add(5,3))
```

Output:

```
Hello, Alice!
8
```

Other Import Options

- **import mymodule** (Use module name)
- **from mymodule import greet** (Import specific function)
- **from mymodule import *** (Import all names - not recommended)
- **import mymodule as mm** (Use alias)

Key Takeaways

- **Lambda** → create small, anonymous functions.
- **Assert** → validate conditions, raise errors if needed.
- **Import** → reuse your own modules and keep code organized.

Write better code... with Python!

Remember

- ✔ **Lambda** = short + simple
- ✔ **Assert** = catch bugs early
- ✔ **Import** = reuse + organize

Lists, Commonly Used Functions on Lists, Passing Lists in Python

Lists are ordered, mutable collections of items. They help us store multiple values in a single variable and work with them easily using built-in functions and methods.

Lists make your code more organized and flexible!

1. Lists

- A list is an ordered collection of items.
- It is mutable (can be changed).
- Items are separated by commas and enclosed in square brackets [].
- A list can contain different data types.

Syntax

```
list_name = [item1, item2, item3, ...]
```

Example

```
fruits = ["Apple", "Banana", "Cherry"]
print(fruits)        # ['Apple', 'Banana', 'Cherry']
print(type(fruits))  # <class 'list'>
```

Output

```
['Apple', 'Banana', 'Cherry']
<class 'list'>
```

Accessing Elements

- **Indexing:** `fruits[0]` → first element
- **Negative indexing:** `fruits[-1]` → last element
- **Slicing:** `fruits[1:3]` → elements from index 1 to 2

Example

```
print(fruits[0])      # Apple
print(fruits[-1])     # Cherry
print(fruits[1:3])    # ['Banana', 'Cherry']
```

Output

```
Apple
Cherry
['Banana', 'Cherry']
```

2. Commonly Used Functions on Lists

Python provides many built-in functions and methods to work with lists. Here are some commonly used ones:

Function / Method	Description
`len(list)`	Returns the number of elements in the list.
`append(x)`	Adds an element at the end.
`insert(i, x)`	Inserts element x at index i.
`extend(iterable)`	Adds elements of an iterable (to the end).
`remove(x)`	Removes the first occurrence of x.
`pop([i])`	Removes and returns element at index i (last element if no index).
`index(x)`	Returns the index of the first occurrence of x.
`count(x)`	Returns the number of occurrences of x.
`sort()`	Sorts the list in ascending order.
`reverse()`	Reverses the list in place.
`clear()`	Removes all elements from the list.

Example

```
numbers = [10, 20, 30, 40, 50]
print(len(numbers))     # 5
numbers.append(60)
print(numbers)          # [10, 20, 30, 40, 50, 60]
numbers.remove(30)
print(numbers!])        # [10, 20, 40, 50, 60]
numbers.sort()
print(numbers)          # [10, 20, 40, 50, 60]
```

Output

```
5
[10, 20, 30, 40, 50, 60]
[10, 20, 40, 50, 60]
```

3. Passing Lists

Lists can be passed to functions in two ways:

1 Passing List as an Argument

- The function receives the list and works on it.
- Changes made inside the function affect the original list (because lists are mutable).

Example

```
def add_item(lst, item):
    lst.append(item)

nums = [1, 2, 3]
add_item(nums, 4)
print(nums)     # [1, 2, 3, 4]
```

Output

```
[1, 2, 3, 4]
```

Note:
If you don't want the original list to change, pass a copy using slice or `list()`:

```
add_item(nums[:], 4)
# or
add_item(list(nums), 4)
```

2 Returning a List from a Function

- The function can create and return a list.
- The returned list can be stored in a variable and used in the calling program.

Example

```
def get_squares(nums):
    return [n*n for n in nums]

result = get_squares([1, 2, 3, 4])
print(result)       # [1, 4, 9, 16]
```

Output

```
[1, 4, 9, 16]
```

Use case:

- To create a new list from existing data.
- To return processed data from a function.

Key Takeaways

- ✓ Lists are ordered, mutable collections.
- ✓ Use built-in functions and methods to manipulate lists.
- ✓ Passing a list to a function passes the reference (changes affect original).
- ✓ Use slicing or `list()` if you want to avoid changes.

Quick Tips

- Use `append()` to add one item.
- Use `extend()` to add multiple items.
- Use `sort()` to sort the list.
- Use copy (`list[:]`) to create a new list.
- Use list comprehension for short and clean code.

Remember

"Lists are powerful!! They help you store, organize and process multiple values with ease."

Tuples and Dictionaries
and Tuples and Dictionaries as Arguments to Functions

Tuples store ordered, immutable collections. Dictionaries store key-value pairs.
Both can be passed to functions to make your code more modular and reusable!

Organize data... Pass it... Get things done!

1. Tuples in Python

What is a Tuple?

- A tuple is an ordered collection of items.
- It is immutable (cannot be changed).
- Can contain different data types.
- Represented using parentheses ( ) and items are separated by commas.

Syntax

```
tuple_name = (item1, item2, item3, ...)
```

Example

```
fruits = ("Apple", "Banana", "Cherry")
print(fruits)
print(type(fruits))
```

Output:
```
('Apple', 'Banana', 'Cherry')
<class 'tuple'>
```

Tuple Operations

- **Indexing:** `t[0]` → first element
- **Slicing:** `t[1:3]` → elements 1 to 2
- **Concatenation:** `t1 + t2`
- **Repetition:** `t * 3`
- **Length:** `len(t)`

Key Points

- ✓ Ordered and immutable.
- ✓ Can be used as function arguments.
- ✓ Useful for fixed data (e.g., coordinates, records).

2. Dictionaries in Python

What is a Dictionary?

- A dictionary stores data in key-value pairs.
- It is mutable (can be changed).
- Keys must be unique and immutable (e.g., strings, numbers, tuples).
- Represented using curly braces { } and key-value pairs separated by colons (:).

Syntax

```
dict_name = {key1: value1, key2: value2, ...}
```

Example

```
student = {
    "name": "Alice",
    "age": 20,
    "course": "Python"
}
print(student)
print(type(student))
```

Output:
```
{'name': 'Alice', 'age': 20,
'course': 'Python'}
<class 'dict'>
```

Dictionary Operations

- **Access value:** `d[key]`
- **Add/Update:** `d[key] = value`
- **Remove:** `del d[key]`
- **Check key:** `key in d`
- **Length:** `len(d)`

Key Points

- ✓ Key-value pairs, unordered (in Python 3.7+ preserves insertion order).
- ✓ Mutable.
- ✓ Used to store structured data.
- ✓ Very useful for lookups and mapping data.

3. Tuples and Dictionaries as Arguments to Functions

- You can pass tuples and dictionaries to functions. Functions can receive, process and use them just like any other type of argument.

1. Passing a Tuple to a Function

- Tuples are passed as a single argument (positional or keyword).

Example (positional argument)

```
def display_info(data):
    print("Tuple received:", data)
    for item in data:
        print(item, end=" ")
    print()

colors = ("Red", "Green", "Blue")
display_info(colors)
```

Output:
```
Tuple received:
Red Green Blue
```

Example (unpacking with *)

```
def sum_values(a, b, c):
    return a + b + c

nums = (10, 20, 30)
print(sum_values(*nums))
```

Output:
```
60
```

2. Passing a Dictionary to a Function

- Dictionaries can be passed as a single argument or using keyword arguments with **.

Example (as a single argument)

```
def show_details(info):
    print("User details:")
    for key, value in info.items():
        print(f"{key}: {value}")

student = {"name": "Alice", "age": 20,
           "course": "Python"}
show_details(student)
```

Output:
```
User details:
name: Alice
age: 20
course: Python
```

3. Passing Both (Tuple and Dictionary)

- You can pass a tuple and a dictionary together as arguments to a function.

Example

```
def person_info(name, details):
    print(f"Name: {name}")
    print("Details:")
    for key, value in details.items():
        print(f"{key}: {value}")

name = "Bob"
details = {"age": 25, "city": "Delhi", "course": "DSA"}
person_info(name, details)
```

Output:
```
Name: Bob
Details:
age: 25
city: Delhi
course: DSA
```

Quick Recap

- ✓ Tuple = ordered + immutable + ( )
- ✓ Dictionary = key-value + mutable + { }
- ✓ Both can be passed to functions as arguments.

Key Takeaways

- ✓ Use tuples for fixed, read-only data.
- ✓ Use dictionaries for flexible, key-based data.
- ✓ Pass them to functions to make your code clean, modular and reusable.

Remember

"Tuples keep your data in order,
Dictionaries give it a name,
Functions make it work together!"

Using Math and NumPy Module for List of Integers and Arrays in Python

Math for single values... NumPy for arrays!

The math module provides mathematical functions for individual numbers.
The NumPy module provides powerful tools for working with arrays (lists of numbers).

1. Math Module – List of Integers

- The math module provides common mathematical functions like square root, power, trigonometric functions, etc.
- It works with single numbers. We can apply these functions to each element of a list using a loop or list comprehension.

```
import math
```

Common Math Functions

- sqrt()
- pow()
- ceil()
- floor()
- sin()
- cos()
- tan()
- log()

Example: Using math functions with a list of integers

```
import math
nums = [1, 4, 9, 16]

# Square root of each number
sqrt_list = [math.sqrt(x) for x in nums]
print("Square roots:", sqrt_list)

# Power (square) of each number
square_list = [math.pow(x, 2) for x in nums]
print("Squares:", square_list)
```

Output

```
Square roots:
[1.0, 2.0, 3.0, 4.0]

Squares:
[1.0, 16.0, 81.0, 256.0]
```

Another Example: Trigonometric Function (on list)

```
import math

angles = [0, 30, 60, 90]

sin_values = [math.sin(math.radians(a)) for a in angles]
print("Sine values:", sin_values)
```

Output

```
Sine values:
[0.0, 0.5, 0.866,
 1.0]
```

2. NumPy Module – Arrays

- NumPy is a powerful library for numerical computing in Python.
- It provides an array object (ndarray) for efficient operations on large collections of numbers.
- It supports vectorized operations (no need for loops).

```
import numpy as np
```

Creating an Array from a List

```
arr = np.array([1, 4, 9, 16])
print(arr)
print(type(arr))
```

Output

```
[ 1  4  9  16]
<class 'numpy.ndarray'>
```

Example: Basic Operations on Arrays

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Arithmetic operations
print("Add 5:", arr + 5)       # adds 5 to each element
print("Multiply by 2:", arr * 2)  # multiplies each element

# Mathematical functions
print("Square root:", np.sqrt(arr))
print("Sum of array:", np.sum(arr))
print("Mean of array:", np.mean(arr))
```

Output

```
Add 5: [  6  7  8  9 10]
Multiply by 2:
[ 2  4  6  6  8 10]
Square root:
[1.          1.41421356
 1.73205081 2.          2.23606798]
Sum of array: 15
Mean of array: 3.0
```

Example: Working with 2D Arrays

```
import numpy as np

matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])

print("Matrix:\n", matrix)

# Matrix operations
print("Transpose:\n", matrix.T)
print("Sum along axis 0 (columns):", np.sum(matrix, axis=0))
print("Sum along axis 1 (rows):", np.sum(matrix, axis=1))
```

Output

```
Matrix:
[[1 2  3]
 [4 5  6]]
Transpose:
[[1 4]
 [2 5]
 [3 6]]
Sum along axis 0 (columns):
 [5 7 9]
Sum along axis 1 (rows):
 [ 6 15]
```

Key Differences

Feature	math Module	NumPy Module
Main use	Mathematical functions (for single values)	Array operations (for lists/arrays)
Data type	Works with numbers (one at a time)	Works with ndarray (multiple values)
Need for loops	Usually required	Not needed (vectorized)
Examples	sqrt(), sin(), log()	array(), sum(), mean(), linspace(), etc.

Working with List of Integers (Math)

We can use math functions on a list by using a loop or list comprehension.

```
import math
nums = [2, 3, 5, 7]
sqrt_nums = [math.sqrt(x) for x in nums]
print(sqrt_nums)
```

Output: [1.41421356, 1.73205081, 2.23606798, 2.64575131]

Quick Recap

- ✓ Use math module for single number calculations.
- ✓ Use NumPy for efficient array operations.
- ✓ Convert list to array with np.array().
- ✓ Perform element-wise operations, statistical functions, and matrix operations with NumPy.

Small numbers or big data... Math and NumPy make it easier!