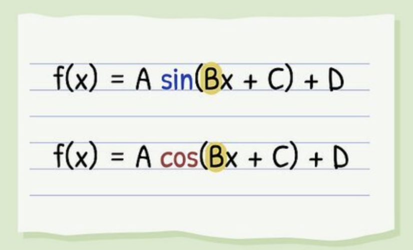
**Lesson 3 – Functions: Building Blocks for Optimization**

# Why Functions?

In math, a function is a rule that turns an *input* into an *output*. For example, $f(x) = x^2$ takes $x = 3$ and gives 9. In Python, functions work the same way: we give them information, they perform steps, and they return a result. However, in Python a function doesn't always have to return an output. Some functions only perform actions (like printing or modifying data) and implicitly return None. Returning a value is optional.



$f(x) = A\sin(Bx + C) + D$

$f(x) = A\cos(Bx + C) + D$

**Optimization Connection**

Optimization often requires evaluating the same formula for many different options (routes, costs, scores). By writing that formula as a function, we can reuse it quickly and reliably instead of rewriting the same code each time.

**Everyday analogy:**

- A **calculator button** (like $\sqrt{\ }$) always does the same job on different numbers.
- A **phone contact** saves a number once so you can call it anytime without retyping.

# Defining a Function in Python

In Python, we create a function with the keyword `def`. A function has a **name**, may take **inputs**, and can give back an **output**.
**Example: Greeting someone**

```python
def greet(name):
    print("Hello,", name)

greet("Jordan")
greet("Maya")
```

*Output*

```
Hello, Jordan
Hello, Maya
```

### Key points

- `def greet(name):` defines the function and its input (`name`).
- Indentation (spaces at the start of the line) is required in Python.
- You can call the function as many times as you want with different inputs.

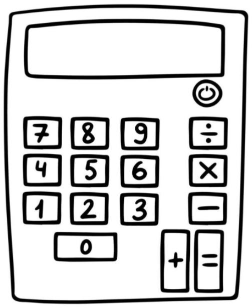
**Example: A simple math function**

```
def square(x):
    return x * x

print(square(3))
print(square(5))
```

*Output*

```
9
25
```



### Optimization Connection

By writing a formula as a function (like cost, time, or score), we can test many options quickly. Instead of rewriting the same code, we just call the function again and again.

## Functions with Multiple Inputs

Functions can take more than one input. This is useful when we want to *compare* or *combine* information.

**Example: Checking affordability**

```
def affordable(price, budget):
    if price <= budget:
        return True
    else:
        return False

print(affordable(5, 10)) # True
print(affordable(12, 10)) # False
```

*Output*

```
True
False
```

**Example: Calculating a total cost**

```python
def total_cost(quantity, unit_price):
    return quantity * unit_price

print(total_cost(4, 3)) # 12
print(total_cost(2, 7)) # 14
```

*Output*

```
12
14
```

**Optimization Connection**

By wrapping formulas in functions, we can test many options quickly. For example, if we know the budget, a function can tell us if a choice (price, quantity) is feasible. This is the same idea as applying **constraints** in optimization.

## Functions + Loops: Testing Many Options

By combining functions with loops, we can **evaluate many choices** and select the best one. This is at the heart of optimization.

**Example: Finding the best affordable item**

```python
def affordable(price, budget):
    return price <= budget

prices = [5, 3, 7, 2, 9]
budget = 6

for p in prices:
    if affordable(p, budget):
        print("You can buy item that costs:", p)
```

*Output*

```
You can buy item that costs: 5
You can buy item that costs: 3
You can buy item that costs: 2
```

**Example: From fastest route to best route**

**Step 1: Choosing by time only**

Suppose each route has a travel time (in minutes). We want to know *which route* is the fastest and how long it takes.

```
times = [12, 15, 9] # travel times in minutes
routes = ["Route A", "Route B", "Route C"]

best_time = float('inf') # start impossibly large
best_route = None # we will store the NAME of the best route

for i, t in enumerate(times):
    if t < best_time: # keep the smallest time seen so far
        best_time = t
        best_route = routes[i] # remember which route had that time

print("Fastest route is", best_route, "with time", best_time, "minutes")
```

*Output*

```
Fastest route is Route C with time 9 minutes
```

### Step 2: Adding more information

Now each route has two values: travel time and fun level.

```
routes = [(12, 8), (15, 10), (9, 6)] # (time, fun)
```

- Route 1: 12 minutes, fun = 8
- Route 2: 15 minutes, fun = 10
- Route 3: 9 minutes, fun = 6

Which one is *best*? Fastest does not always mean most fun.

### Step 3: Creating a scoring rule

We design a function that balances both. Here, each point of fun counts as +1, but every 2 minutes of time subtracts 1 (penalty).

```
def score(time, fun):
    return fun - 0.5 * time
```

### Step 4: Looping to find the best

We now use a loop to evaluate all routes with the same rule.

```
best_score = -float('inf')
best_route = None
```

```python
for time, fun in routes:
    s = score(time, fun)
    print("Route:", (time, fun), "score =", s)
    if s > best_score:
        best_score = s
        best_route = (time, fun)

print("Best route:", best_route, "with score =", best_score)
```

*Output*

```
Route: (12, 8) score = 2.0
Route: (15, 10) score = 2.5
Route: (9, 6) score = 1.5
Best route: (15, 10) with score = 2.5
```

### Optimization Lens

This is the heart of optimization:

1. Write a function to evaluate each option.
2. Loop through all possibilities.
3. Keep the one with the best score.

Even though Route 3 was the fastest, Route 2 wins overall because its higher fun level offsets the extra time.

## Practice: Writing Your Own Functions

Now it is your turn to practice writing and using functions. Remember, a function should:

- Have a clear purpose,
- Take inputs (parameters),
- Do some work (computation or decision),
- Return or print an output.

### Lesson 3 Activities

1. Write a function `square(x)` that returns $x^2$. Test it with at least 3 numbers.
2. Write a function `max_of_two(a, b)` that returns the larger of two numbers. (Do not use Python's built-in `max`.)
3. Write a function `affordable(price, budget)` that returns `True` if the item is within budget, else `False`. Use a loop to test it on `prices = [4, 12, 7, 3]` with a budget of 8.

4. Write a function `route_score(time, fun)` that uses the formula:

   $$\text{score} = \text{fun} - 0.5 \times \text{time}$$

   Then use a loop to find the best route from: `routes = [(10, 6), (14, 9), (8, 5)]`.

5. **Challenge:** Write a function `best_item(items, budget)` that takes a list of pairs `(price, fun)` and returns the most fun item you can afford.