🤓 Learn All About Python Functions – Step-by-Step Practice! 🚀

Unlock the power of Python functions by solving these interactive problems 👇

▶ ◇ **1. Basic Function Syntax**

📝 **Problem:** Write a function to calculate and return the **square of a number**.

```
def square(num):
    return num ** 2

print(square(5))  # Output: 25
```

▶ ◇ **2. Function with Multiple Parameters**

📝 **Problem:** Create a function that takes **two numbers** as parameters and returns their **sum**.

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

print(add(3, 7))  # Output: 10
```

▶ 🔀 **3. Polymorphism in Functions**

📝 **Problem:** Write a function `multiply` that multiplies **two numbers**, but can also **accept and multiply strings**.

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

print(multiply(3, 4))      # Output: 12
print(multiply("Hi", 3))   # Output: HiHiHi
```

▶ ⚖️ **4. Function Returning Multiple Values**

📝 **Problem:** Create a function that returns both the **area** and **circumference** of a circle given its **radius**.

```
import math

def circle_stats(radius):
```

```
    area = math.pi * radius ** 2
    circumference = 2 * math.pi * radius
    return area, circumference

a, c = circle_stats(5)
print(f"Area: {a}, Circumference: {c}")
```

▶ 👤 **5. Default Parameter Value**

✏️ **Problem:** Write a function that **greets a user**. If no name is provided, it should greet with a **default name**.

```
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet("Darshan")  # Output: Hello, Darshan!
greet()           # Output: Hello, Guest!
```

▶ ⚡ **6. Lambda Function**

✏️ **Problem:** Create a **lambda function** to compute the **cube** of a number.

```
cube = lambda x: x ** 3
print(cube(3))  # Output: 27
```

▶ 📦 **7. Function with *args**

✏️ **Problem:** Write a function that takes a **variable number of arguments** and returns their **sum**.

```
def sum_all(*args):
    return sum(args)

print(sum_all(1, 2, 3, 4))  # Output: 10
```

▶ 🗃️ **8. Function with **kwargs**

✏️ **Problem:** Create a function that accepts any number of **keyword arguments** and prints them as **key: value**.

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_kwargs(name="Alice", age=25, city="Delhi")
```

▶ 🔁 **9. Generator Function with `yield`**

📝 **Problem:** Write a **generator function** that yields all **even numbers** up to a specified limit.

```
def even_generator(limit):
    for i in range(0, limit + 1, 2):
        yield i

for num in even_generator(10):
    print(num, end=" ")  # Output: 0 2 4 6 8 10
```

▶ 🧬 **10. Recursive Function**

📝 **Problem:** Create a **recursive function** to calculate the **factorial** of a number.

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

🔁 Python `yield` Keyword – Mastering Generators 🚀

🧠 What is `yield`?

- `yield` is used in **generator functions** to return a value **without ending** the function.
- It **pauses** the function, **saves the state**, and **resumes** from where it left off when called again.
- Think of `yield` as `return`, but **with memory**!

🔁 Difference between `return` and `yield`

Feature	`return`	`yield`
Stops function	☑ Yes	❌ No – pauses and can resume
Returns	Single value or object	A generator (iterator)
Memory usage	❌ Can be high	☑ Very efficient (lazy evaluation)
Use case	One-time result	Repeated or large sequence generation

⚙ How it works

When you call a generator function:

- It doesn't execute immediately.
- It returns a **generator object**.
- Use `next()` or loop (`for`) to get values one by one.

📦 Simple Example

```
def count_up_to(limit):
    num = 1
    while num <= limit:
        yield num
        num += 1

counter = count_up_to(5)
for i in counter:
    print(i)
```

📤 Output:

```
1
2
3
4
5
```

🔍 Real-Life Analogy

Imagine a **vending machine**:

- Each time you press a button (`next()`), it gives one snack (`yield`).
- It doesn’t shut down after one use like `return`.
- It stays on and remembers where it left off.

🔥 When to Use `yield`

- When working with **large datasets**.
- When you need **lazy evaluation**.
- When returning a **stream of data** instead of storing it all in memory.

⚡ Pro Tip:

You can combine `yield` with `yield from` to yield from another generator!

```python
def sub_gen():
    yield 1
    yield 2

def main_gen():
    yield from sub_gen()
    yield 3

for i in main_gen():
    print(i)
```
