1. Positional Arguments

- These are the most common; arguments are matched to parameters by their position (order).
- The number and order of arguments must match the function definition.

python

```
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")

greet("Aayush", 23)
```

Output:
Hello Aayush, you are 23 years old.

---

2. Keyword Arguments

- Arguments are passed by explicitly naming each parameter, so order doesn't matter.
- This enhances readability and avoids confusion.

python

```
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")

greet(age=23, name="Aayush")
```

Output:
Hello Aayush, you are 23 years old.

---

3. Default Arguments

- Parameters can have default values, making them optional in function calls.

python

```
def calculate_area(length, width=5):
    area = length * width
    print(f"Area: {area}")

calculate_area(10)          # Uses default width=5
calculate_area(10, 8)       # Overrides width
```

Output:
Area: 50
Area: 80

---

4. Arbitrary Positional Arguments (*args)

- Use an asterisk (*) to accept a variable number of non-keyword arguments (packed as a tuple).

python

```
def add_numbers(*args):
    print(sum(args))

add_numbers(1, 2, 3)        # Output: 6
add_numbers(3, 10, 5, 88)   # Output: 106
```

---

5. Arbitrary Keyword Arguments (**kwargs)

- Use two asterisks (**) to accept any number of keyword arguments (packed as a dictionary).

python

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

print_details(name="Aayush", age=23, course="CSE")
```

Output:
name: Aayush
age: 23
course: CSE

Anonymous Functions :

Anonymous functions in Python—commonly referred to as lambda functions—are small, unnamed functions defined with the `lambda` keyword. They are especially useful when used with built-in functions such as `map()`, `filter()`, and `reduce()`. Here's a concise overview with practical examples:

Lambda (Anonymous) Functions

- Syntax: `lambda arguments: expression`
- Lambda functions can have any number of arguments but only one expression (no statements or multiple lines).

python

```python
# Example: Squaring a number
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

---

1. map()

- Purpose: Applies a function to all items in an iterable (like a list) and returns a map object (iterable).

python

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # Output: [1, 4, 9, 16, 25]
```

---

2. filter()

- Purpose: Filters elements from an iterable for which the function returns True.

python

```
numbers = [1, 2, 3, 4, 5]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2, 4]
```

---

3. reduce()

- Purpose: Repeatedly applies a function to the items of an iterable, reducing it to a single value.
- Note: `reduce()` is available in the `functools` module.

python

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 120
```

Key Points

- Lambda functions are best for quick, throwaway uses, usually when passing as arguments.
- They make code concise, ideal for operations like mapping, filtering, or reducing collections.
- For clarity and maintainability, use named functions (`def`) for complex logic or reuse.