

Python: Advanced Concepts

This section covers several advanced concepts in Python, including decorators, getters and setters, static and class methods, magic methods, exception handling, map/filter/reduce, the walrus operator, and `*args/**kwargs`.

Decorators in Python

Introduction

Decorators in Python are a powerful and expressive feature that allows you to modify or enhance functions and methods in a clean and readable way. They provide a way to wrap additional functionality around an existing function without permanently modifying it. This is often referred to as *metaprogramming*, where one part of the program tries to modify another part of the program at compile time.

Decorators use Python's higher-order function capability, meaning functions can accept other functions as arguments and return new functions.

Understanding Decorators

A decorator is simply a callable (usually a function) that takes another function as an argument and returns a replacement function. The replacement function typically *extends* or *alters* the behavior of the original function.

Basic Example of a Decorator

```
def my_decorator(func):  
    def wrapper():  
        print("Something is happening before the function is call  
        func()  
        print("Something is happening after the function is calle  
    return wrapper
```

```
@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

Here, `@my_decorator` is syntactic sugar for `say_hello = my_decorator(say_hello)`. It modifies the behavior of `say_hello()` by wrapping it inside `wrapper()`. The `wrapper` function adds behavior before and after the original function call.

Using Decorators with Arguments

Decorators themselves can also accept arguments. This requires another level of nesting: an outer function that takes the decorator's arguments and returns the actual decorator function.

```
def repeat(n):
    def decorator(func):
        def wrapper(a):
            for _ in range(n):
                func(a)
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")
```

```
greet("world")
```

Output:

```
Hello, world!  
Hello, world!  
Hello, world!
```

In this example, `repeat(3)` *returns* the decorator function. The `@` syntax then applies that returned decorator to `greet`. The argument in the `wrapper` function ensures that the decorator can be used with functions that take any number of positional and keyword arguments.

Chaining Multiple Decorators

You can apply multiple decorators to a single function. Decorators are applied from bottom to top (or, equivalently, from the innermost to the outermost).

```
def uppercase(func):  
    def wrapper():  
        return func().upper()  
    return wrapper  
  
def exclaim(func):  
    def wrapper():  
        return func() + "!!!"  
    return wrapper  
  
@uppercase  
@exclaim  
def greet():  
    return "hello"  
  
print(greet())
```

Output:

```
HELLO!!!
```

Here, `greet` is first decorated by `exclaim`, and then the result of *that* is decorated by `uppercase`. It's equivalent to `greet = uppercase(exclaim(greet))`.

Recap

Decorators are a key feature in Python that enable code reusability and cleaner function modifications. They are commonly used for:

- **Logging:** Recording when a function is called and its arguments.
- **Timing:** Measuring how long a function takes to execute.
- **Authentication and Authorization:** Checking if a user has permission to access a function.
- **Caching:** Storing the results of a function call so that subsequent calls with the same arguments can be returned quickly.
- **Rate Limiting:** Controlling how often a function can be called.
- **Input Validation:** Checking if the arguments to a function meet certain criteria.
- **Instrumentation:** Adding monitoring and profiling to functions.

Frameworks like Flask and Django use decorators extensively for routing, authentication, and defining middleware.

Getters and Setters in Python

Introduction

In object-oriented programming, **getters** and **setters** are methods used to control access to an object's attributes (also known as properties or instance variables). They provide a way to *encapsulate* the internal representation of an object, allowing you to validate data, enforce constraints, and perform other operations when an attribute is accessed or modified. While Python doesn't have private

variables in the same way as languages like Java, the convention is to use a leading underscore (`_`) to indicate that an attribute is intended for internal use.

Using getters and setters helps:

- **Encapsulate data and enforce validation:** You can check if the new value meets certain criteria before assigning it.
- **Control access to “private” attributes:** By convention, attributes starting with an underscore are considered private, and external code should use getters/setters instead of direct access.
- **Make the code more maintainable:** Changes to the internal representation of an object don't necessarily require changes to code that uses the object.
- **Add additional logic:** Logic can be added when getting or setting attributes.

Using Getters and Setters

Traditional Approach (Using Methods)

A basic approach is to use explicit getter and setter methods:

```
class Person:
    def __init__(self, name):
        self._name = name # Convention: underscore (_) denotes a private attribute

    def get_name(self):
        return self._name

    def set_name(self, new_name):
        self._name = new_name

p = Person("Alice")
print(p.get_name()) # Alice
p.set_name("Bob")
print(p.get_name()) # Bob
```

Using `@property` (Pythonic Approach)

Python provides a more elegant and concise way to implement getters and setters using the `@property` decorator. This allows you to access and modify attributes using the usual dot notation (e.g., `p.name`) while still having the benefits of getter and setter methods.

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self): # Getter
        return self._name

    @name.setter
    def name(self, new_name): # Setter
        self._name = new_name

p = Person("Alice")
print(p.name) # Alice (calls the getter)

p.name = "Bob" # Calls the setter
print(p.name) # Bob
```

Benefits of `@property` :

- **Attribute-like access:** You can use `obj.name` instead of `obj.get_name()` and `obj.set_name()`, making the code cleaner and more readable.
- **Consistent interface:** The external interface of your class remains consistent even if you later decide to add validation or other logic to the getter or setter.
- **Read-only properties:** You can create read-only properties by simply omitting the `@property.setter` method (see the next section).
- `@property.deleter` : deletes a property. Here is an example:

```

class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self): # Getter
        return self._name

    @name.setter
    def name(self, new_name): # Setter
        self._name = new_name

    @name.deleter
    def name(self):
        del self._name

p = Person("Alice")
print(p.name) # Alice
del p.name
print(p.name) # AttributeError: 'Person' object has no attril

```

Read-Only Properties

If you want an attribute to be **read-only**, define only the `@property` decorator (the getter) and omit the `@name.setter` method. Attempting to set the attribute will then raise an `AttributeError`.

```

class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @property

```

```
def area(self): # Read-only computed property
    return 3.1416 * self._radius * self._radius

c = Circle(5)
print(c.radius) # 5
print(c.area) # 78.54

# c.radius = 10 # Raises AttributeError: can't set attribute
# c.area = 20 # Raises AttributeError: can't set attribute
```

Recap

- **Getters and Setters** provide controlled access to an object's attributes, promoting encapsulation and data validation.
- The `@property` decorator offers a cleaner and more Pythonic way to implement getters and setters, allowing attribute-like access.
- You can create **read-only properties** by defining only a getter (using `@property` without a corresponding `@<attribute>.setter`).
- Using `@property`, you can dynamically compute values (like the `area` in the `Circle` example) while maintaining an attribute-like syntax.

Static and Class Methods in Python

Introduction

In Python, methods within a class can be of three main types:

- **Instance Methods:** These are the most common type of method. They operate on *instances* of the class (objects) and have access to the instance's data through the `self` parameter.
- **Class Methods:** These methods are bound to the *class* itself, not to any particular instance. They have access to class-level attributes and can be used to modify the class state. They receive the class itself (conventionally named `cls`) as the first argument.

- **Static Methods:** These methods are associated with the class, but they don't have access to either the instance (`self`) or the class (`cls`). They are essentially regular functions that are logically grouped within a class for organizational purposes.
-

Instance Methods (Default Behavior)

Instance methods are the default type of method in Python classes. They require an instance of the class to be called, and they automatically receive the instance as the first argument (`self`).

```
class Dog:
    def __init__(self, name):
        self.name = name # Instance attribute

    def speak(self):
        return f"{self.name} says Woof!"

dog = Dog("Buddy")
print(dog.speak()) # Buddy says Woof!
```

Class Methods (`@classmethod`)

A class method is marked with the `@classmethod` decorator. It takes the class itself (`cls`) as its first parameter, rather than the instance (`self`). Class methods are often used for:

- **Modifying class attributes:** They can change the state of the class, which affects all instances of the class.
- **Factory methods:** They can be used as alternative constructors to create instances of the class in different ways.

```
class Animal:
    species = "Mammal" # Class attribute
```

```

@classmethod
def set_species(cls, new_species):
    cls.species = new_species # Modifies class attribute

@classmethod
def get_species(cls):
    return cls.species

print(Animal.get_species()) # Mammal
Animal.set_species("Reptile")
print(Animal.get_species()) # Reptile

# You can also call class methods on instances, but it's less common
a = Animal()
print(a.get_species()) # Reptile

```

Example: Alternative Constructor

```

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def from_string(cls, data):
        name, age = data.split("-")
        return cls(name, int(age)) # Creates a new Person instance

p = Person.from_string("Alice-30")
print(p.name, p.age) # Alice 30

```

In this example, `from_string` acts as a factory method, providing an alternative way to create `Person` objects from a string.

Static Methods (@staticmethod)

Static methods are marked with the `@staticmethod` decorator. They are similar to regular functions, except they are defined within the scope of a class.

- They **don't** take `self` or `cls` as parameters.
- They are useful when a method is logically related to a class but doesn't need to access or modify the instance or class state.
- Often used for utility functions that are related to the class

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

print(MathUtils.add(3, 5)) # 8

#Can also be called on an instance
m = MathUtils()
print(m.add(4,5)) # 9
```

When to Use Static Methods?

- When a method is logically related to a class but doesn't require access to instance-specific or class-specific data.
- For utility functions that perform operations related to the class's purpose (e.g., mathematical calculations, string formatting, validation checks).

Key Differences Between Method Types

Method Type	Requires <code>self</code> ?	Requires <code>cls</code> ?	Can Access Instance Attributes?	Can Modify Class Attributes?
Instance Method	☑ Yes	✗ No	☑ Yes	☑ Yes (indirectly)

Method Type	Requires <code>self</code> ?	Requires <code>cls</code> ?	Can Access Instance Attributes?	Can Modify Class Attributes?
Class Method	✗ No	☑ Yes	✗ No (directly)	☑ Yes
Static Method	✗ No	✗ No	✗ No	✗ No

Recap

- **Instance methods** are the most common type and operate on individual objects (`self`).
- **Class methods** operate on the class itself (`cls`) and are often used for factory methods or modifying class-level attributes.
- **Static methods** are utility functions within a class that don't depend on the instance or class state. They're like regular functions that are logically grouped with a class.

Magic (Dunder) Methods in Python

Introduction

Magic methods, also called **dunder (double underscore) methods**, are special methods in Python that have double underscores at the beginning and end of their names (e.g., `__init__` , `__str__` , `__add__`). These methods allow you to define how your objects interact with built-in Python operators, functions, and language constructs. They provide a way to implement *operator overloading* and customize the behavior of your classes in a Pythonic way.

They are used to:

- Customize object creation and initialization (`__init__` , `__new__`).
- Enable operator overloading (e.g., `+` , `-` , `*` , `==` , `<` , `>`).

- Provide string representations of objects (`__str__` , `__repr__`).
 - Control attribute access (`__getattr__` , `__setattr__` , `__delattr__`).
 - Make objects callable (`__call__`).
 - Implement container-like behavior (`__len__` , `__getitem__` , `__setitem__` , `__delitem__` , `__contains__`).
 - Support with context managers (`__enter__` , `__exit__`)
-

Common Magic Methods

1. `__init__` – Object Initialization

The `__init__` method is the constructor. It's called automatically when a new instance of a class is created. It's used to initialize the object's attributes.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p = Person("Alice", 30)
print(p.name, p.age)  # Alice 30
```

2. `__str__` and `__repr__` – String Representation

- `__str__` : This method should return a human-readable, informal string representation of the object. It's used by the `str()` function and by `print()` .
- `__repr__` : This method should return an unambiguous, official string representation of the object. Ideally, this string should be a valid Python expression that could be used to recreate the object. It's used by the `repr()` function and in the interactive interpreter when you just type the object's name and press Enter.

```

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"Person({self.name}, {self.age})" # User-friendly

    def __repr__(self):
        return f"Person(name='{self.name}', age={self.age})" # U

p = Person("Alice", 30)
print(str(p))      # Person(Alice, 30)
print(repr(p))     # Person(name='Alice', age=30)
print(p)           # Person(Alice, 30) # print() uses __str__ if

```

If `__str__` is not defined, Python will use `__repr__` as a fallback for `str()` and `print()`. It's good practice to define *at least* `__repr__` for every class you create.

3. `__len__` – Define Behavior for `len()`

This method allows objects of your class to work with the built-in `len()` function. It should return the "length" of the object (however you define that).

```

class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __len__(self):
        return self.pages

b = Book("Python 101", 250)
print(len(b)) # 250

```

4. `__add__`, `__sub__`, `__mul__`, etc. – Operator Overloading

These methods allow you to define how your objects behave with standard arithmetic and comparison operators.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2  # Calls __add__
print(v3)     # Vector(6, 8)
v4 = v3 - v1
print(v4)     # Vector(4, 5)
v5 = v1 * 5
print(v5)    # Vector(10, 15)
```

Other common operator overloading methods include:

- `__eq__` (`==`)
- `__ne__` (`!=`)
- `__lt__` (`<`)
- `__gt__` (`>`)
- `__le__` (`<=`)

- `__ge__` (`>=`)
 - `__truediv__` (`/`)
 - `__floordiv__` (`//`)
 - `__mod__` (`%`)
 - `__pow__` (`**`)
-

Recap

Magic (dunder) methods are a powerful feature of Python that allows you to:

- Customize how your objects interact with built-in operators and functions.
 - Make your code more intuitive and readable by using familiar Python syntax.
 - Implement operator overloading, container-like behavior, and other advanced features.
 - Define string representation.
-

Exception Handling and Custom Errors in Python

Introduction

Exceptions are events that occur during the execution of a program that disrupt the normal flow of instructions. Python provides a robust mechanism for handling exceptions using `try-except` blocks. This allows your program to gracefully recover from errors or unexpected situations, preventing crashes and providing informative error messages. You can also define your own custom exceptions to represent specific error conditions in your application.

Basic Exception Handling

The `try-except` block is the fundamental construct for handling exceptions:

- The `try` block contains the code that might raise an exception.

- The `except` block contains the code that will be executed if a specific exception occurs within the `try` block.

```
try:
    x = 10 / 0 # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Output:

```
Cannot divide by zero!
```

Handling Multiple Exceptions

You can handle multiple types of exceptions using multiple `except` blocks or by specifying a tuple of exception types in a single `except` block.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
    print("Invalid input! Please enter a number.")

# Alternative using a tuple:
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ZeroDivisionError, ValueError) as e:
    print(f"An error occurred: {e}")
```

Using `else` and `finally`

- **`else`** : The `else` block is optional and is executed only if *no* exception occurs within the `try` block. It's useful for code that should run only when the `try` block succeeds.
- **`finally`** : The `finally` block is also optional and is *always* executed, regardless of whether an exception occurred or not. It's typically used for cleanup operations, such as closing files or releasing resources.

```
try:
    file = open("test.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
else:
    print("File read successfully.")
    print(f"File contents:\n{content}")
finally:
    file.close() # Ensures the file is closed no matter what
```

Raising Exceptions (`raise`)

You can manually raise exceptions using the `raise` keyword. This is useful for signaling error conditions in your own code.

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or older!")
    return "Access granted."

try:
    print(check_age(20)) # Access granted.
    print(check_age(16)) # Raises ValueError
except ValueError as e:
    print(f"Error: {e}")
```

Custom Exceptions

Python allows you to define your own custom exception classes by creating a new class that inherits (directly or indirectly) from the built-in `Exception` class (or one of its subclasses). This makes your error handling more specific and informative.

```
class InvalidAgeError(Exception):
    """Custom exception for invalid age."""
    def __init__(self, message="Age must be 18 or older!"):
        self.message = message
        super().__init__(self.message)

def verify_age(age):
    if age < 18:
        raise InvalidAgeError() # Raise your custom exception
    return "Welcome!"

try:
    print(verify_age(16))
except InvalidAgeError as e:
    print(f"Error: {e}")
```

By defining custom exceptions, you can:

- Create a hierarchy of exceptions that reflect the specific error conditions in your application.
- Provide more informative error messages tailored to your application's needs.
- Make it easier for other parts of your code (or other developers) to handle specific errors appropriately.

Conclusion

- `try-except` blocks are essential for handling errors and preventing program crashes.
- Multiple `except` blocks or a tuple of exception types can be used to handle different kinds of errors.

- The `else` block executes only if no exception occurs in the `try` block.
 - The `finally` block *always* executes, making it suitable for cleanup tasks.
 - The `raise` keyword allows you to manually trigger exceptions.
 - Custom exceptions (subclasses of `Exception`) provide a way to represent application-specific errors and improve error handling clarity.
-

Map, Filter, and Reduce

Introduction

`map`, `filter`, and `reduce` are higher-order functions in Python (and many other programming languages) that operate on iterables (lists, tuples, etc.). They provide a concise and functional way to perform common operations on sequences of data without using explicit loops. While they were more central to Python's functional programming style in earlier versions, list comprehensions and generator expressions often provide a more readable alternative in modern Python.

Map

The `map()` function applies a given function to each item of an iterable and returns an iterator that yields the results.

Syntax: `map(function, iterable, ...)`

- `function` : The function to apply to each item.
- `iterable` : The iterable (e.g., list, tuple) whose items will be processed.
- `...` : `map` can take multiple iterables. The function must take the same number of arguments

```
numbers = [1, 2, 3, 4, 5]

# Square each number using map
squared_numbers = map(lambda x: x**2, numbers)
print(list(squared_numbers)) # Output: [1, 4, 9, 16, 25]
```

```
#Example with multiple iterables
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]
summed = map(lambda x, y: x + y, numbers1, numbers2)
print(list(summed)) # Output: [5, 7, 9]

# Equivalent list comprehension:
squared_numbers_lc = [x**2 for x in numbers]
print(squared_numbers_lc) # Output: [1, 4, 9, 16, 25]
```

Filter

The `filter()` function constructs an iterator from elements of an iterable for which a function returns `True`. In other words, it filters the iterable based on a condition.

Syntax: `filter(function, iterable)`

- `function` : A function that returns `True` or `False` for each item. If `None` is passed, it defaults to checking if the element is `True` (truthy value).
- `iterable` : The iterable to be filtered.

```
numbers = [1, 2, 3, 4, 5, 6]

# Get even numbers using filter
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers)) # Output: [2, 4, 6]

# Equivalent list comprehension:
even_numbers_lc = [x for x in numbers if x % 2 == 0]
print(even_numbers_lc) # Output: [2, 4, 6]

# Example with None as function
values = [0, 1, [], "hello", "", None, True, False]
truthy_values = filter(None, values)
print(list(truthy_values)) # Output: [1, 'hello', True]
```

Reduce

The `reduce()` function applies a function of two arguments cumulatively to the items of an iterable, from left to right, so as to reduce the iterable to a single value.

`reduce` is not a built-in function; it must be imported from the `functools` module.

Syntax: `reduce(function, iterable[, initializer])`

- `function` : A function that takes two arguments.
- `iterable` : The iterable to be reduced.
- `initializer` (optional): If provided, it's placed before the items of the iterable in the calculation and serves as a default when the iterable is empty.

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]

# Calculate the sum of all numbers using reduce
sum_of_numbers = reduce(lambda x, y: x + y, numbers)
print(sum_of_numbers) # Output: 15

# Calculate the product of all numbers using reduce
product_of_numbers = reduce(lambda x, y: x * y, numbers)
print(product_of_numbers) # Output: 120

#reduce with initializer
empty_list_sum = reduce(lambda x,y: x+y, [], 0)
print(empty_list_sum) # 0

# Without the initializer:
# empty_list_sum = reduce(lambda x,y: x+y, []) # raises TypeError

# Equivalent using a loop (for sum):
total = 0
for x in numbers:
    total += x
print(total) # 15
```

When to use map, filter, reduce vs. list comprehensions/generator expressions:

- **Readability:** List comprehensions and generator expressions are often more readable and easier to understand, especially for simple operations.
- **Performance:** In many cases, list comprehensions/generator expressions can be slightly faster than `map` and `filter`.
- **Complex Operations:** `reduce` can be useful for more complex aggregations where
- **Complex Operations:** `reduce` can be useful for more complex aggregations where the logic is not easily expressed in a list comprehension. `map` and `filter` may also be preferable when you already have a named function that you want to apply.
- **Functional Programming Style:** If you're working in a more functional programming style, `map`, `filter`, and `reduce` can fit naturally into your code.

Walrus Operator (:=)

Introduction

The walrus operator (`:=`), introduced in Python 3.8, is an *assignment expression* operator. It allows you to assign a value to a variable *within an expression*. This can make your code more concise and, in some cases, more efficient by avoiding repeated calculations or function calls. The name "walrus operator" comes from the operator's resemblance to the eyes and tusks of a walrus.

Use Cases

1. **Conditional Expressions:** The most common use case is within `if` statements, `while` loops, and list comprehensions, where you need to both test a condition *and* use the value that was tested.

```
# Without walrus operator
data = input("Enter a value (or 'quit' to exit): ")
```

```
while data != "quit":
    print(f"You entered: {data}")
    data = input("Enter a value (or 'quit' to exit): ")

# With walrus operator
while (data := input("Enter a value (or 'quit' to exit): "))
    print(f"You entered: {data}")
```

In the “with walrus” example, the input is assigned to `data` *and* compared to “quit” in a single expression.

2. **List Comprehensions:** You can avoid repeated calculations or function calls within a list comprehension.

```
numbers = [1, 2, 3, 4, 5]

# Without walrus operator: calculate x * 2 twice
results = [x * 2 for x in numbers if x * 2 > 5]

# With walrus operator: calculate x * 2 only once
results = [y for x in numbers if (y := x * 2) > 5]
```

3. **Reading Files:** You can read lines from a file and process them within a loop.

```
# Without Walrus
with open("my_file.txt", "r") as f:
    line = f.readline()
    while line:
        print(line.strip())
        line = f.readline()

# With Walrus
with open("my_file.txt", "r") as f:
    while (line := f.readline()):
        print(line.strip())
```


Considerations

- **Readability:** While the walrus operator can make code more concise, it can also make it harder to read if overused. Use it judiciously where it improves clarity.
 - **Scope:** The variable assigned using `:=` is scoped to the surrounding block (e.g., the `if` statement, `while` loop, or list comprehension).
 - **Precedence:** The walrus operator has lower precedence than most other operators. Parentheses are often needed to ensure the expression is evaluated as intended.
-

Args and Kwargs

Introduction

`*args` and `**kwargs` are special syntaxes in Python function definitions that allow you to pass a variable number of arguments to a function. They are used when you don't know in advance how many arguments a function might need to accept.

- `*args` : Allows you to pass a variable number of *positional* arguments.
- `**kwargs` : Allows you to pass a variable number of *keyword* arguments.

`*args` (Positional Arguments)

`*args` collects any extra positional arguments passed to a function into a *tuple*. The name `args` is just a convention; you could use any valid variable name preceded by a single asterisk (e.g., `*values`, `*numbers`).

```
def my_function(*args):  
    print(type(args)) # <class 'tuple'>  
    for arg in args:  
        print(arg)  
  
my_function(1, 2, 3, "hello") # Output: 1 2 3 hello
```

```
my_function() # No output (empty tuple)
my_function("a", "b") # Output: a b
```

In this example, `*args` collects all positional arguments passed to `my_function` into the `args` tuple.

`kwargs`** (Keyword Arguments)

`**kwargs` collects any extra *keyword* arguments passed to a function into a *dictionary*. Again, `kwargs` is the conventional name, but you could use any valid variable name preceded by two asterisks (e.g., `**data`, `**options`).

```
def my_function(**kwargs):
    print(type(kwargs)) # <class 'dict'>
    for key, value in kwargs.items():
        print(f"{key}: {value}")

my_function(name="Alice", age=30, city="New York")
# Output:
# name: Alice
# age: 30
# city: New York

my_function() # No output (empty dictionary)
my_function(a=1, b=2)
# Output:
# a: 1
# b: 2
```

In this example, `**kwargs` collects all keyword arguments into the `kwargs` dictionary.

Combining `*args` and `**kwargs`

You can use both `*args` and `**kwargs` in the same function definition. The order is important: `*args` must come *before* `**kwargs`. You can also include regular positional and keyword parameters.

```
def my_function(a, b, *args, c=10, **kwargs):
    print(f"a: {a}")
    print(f"b: {b}")
    print(f"args: {args}")
    print(f"c: {c}")
    print(f"kwargs: {kwargs}")

my_function(1, 2, 3, 4, 5, c=20, name="Bob", country="USA")
# Output:
# a: 1
# b: 2
# args: (3, 4, 5)
# c: 20
# kwargs: {'name': 'Bob', 'country': 'USA'}

my_function(1, 2)
# Output:
# a: 1
# b: 2
# args: ()
# c: 10
# kwargs: {}
```

Use Cases

- **Flexible Function Design:** `*args` and `**kwargs` make your functions more flexible, allowing them to handle a varying number of inputs without needing to define a specific number of parameters.
- **Decorator Implementation:** Decorators often use `*args` and `**kwargs` to wrap functions that might have different signatures.
- **Function Composition:** You can use `*args` and `**kwargs` to pass arguments through multiple layers of function calls.
- **Inheritance:** Subclasses can accept extra parameters to those defined by parent classes.

```
# Example showing use in inheritance
class Animal:
```

```
def __init__(self, name):
    self.name = name

class Dog(Animal):
    def __init__(self, name, breed, *args, **kwargs):
        super().__init__(name)
        self.breed = breed
        # Process any additional arguments or keyword arguments here
        print(f"args: {args}")
        print(f"kwargs: {kwargs}")

dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Lucy", "Labrador", 1, 2, 3, color="Black", age = 5)
```