Python

Understanding Python Namespaces

A Deep Dive into Python's Name Resolution

May 2, 2025

Source Code

PYTHON | NAMESPACES

1. Understanding Python Namespaces

In Python, a namespace is a container that holds a mapping of names to objects. Think of it as a dictionary where variable names are the keys and the objects they refer to are the values. Understanding namespaces is crucial because they:

- Prevent naming conflicts by organizing names into hierarchical spaces
- Define the scope and lifetime of variables
- Help manage the visibility and accessibility of variables

2. Types of Namespaces

Python has several types of namespaces with different lifetimes:

- **Built-in Namespace:** Contains built-in functions and exceptions
- **Global Namespace:** Module-level variables and functions
- **Local Namespace:** Names inside functions
- **Enclosing Namespace:** Names in outer functions (for nested functions)

Let's explore each type with practical examples:

2.1. Basic Namespace Example

How to Run:

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | **NAMESPACES**

- Save the code as `01_basic_namespaces.py`
- Run: `python3 01_basic_namespaces.py`

```python
# --- 01_basic_namespaces.py ---
"""
Basic Namespaces in Python

This file demonstrates the fundamental concept of namespaces in Python.
A namespace is a mapping from names to objects.
"""

# The built-in namespace contains functions like print, len, etc.
print("Built-in namespace example:")
print(f"Type of len function: {type(len)}")
print(f"ID of len function: {id(len)}")

# The global namespace of a module
x = 10
y = "hello"
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | **NAMESPACES**

```python

print("\nGlobal namespace example:")
print(f"x = {x}, type: {type(x)}, id: {id(x)}")
print(f"y = {y}, type: {type(y)}, id: {id(y)}")

# Local namespace in a function with proper global variable handling
def example_function():
    # Declare x as global before using it
    global x
    z = 20  # Local variable
    print("\nLocal namespace inside function:")
    print(f"z = {z}, type: {type(z)}, id: {id(z)}")

    # Now we can safely access and modify the global x
    print(f"x from global namespace: {x}")
    x = 30  # This modifies the global x
    print(f"Modified global x: {x}")

# Call the function
print("\nBefore function call:")
print(f"Global x is: {x}")
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
38
39 example_function()
40
41 print("\nAfter function call:")
42 print(f"Global x is now: {x}")  # Will show the modified
      value
```

Key points about this example:

- Built-in functions like `len` live in the built-in namespace
- Variables defined at module level (`x` and `y`) are in the global namespace
- Variables defined inside functions (`z`) are in the local namespace
- The `global` keyword allows modifying global variables from inside functions

2.2. Nested Scopes and the nonlocal Keyword

Let's explore how Python handles nested function scopes and the usage of the `nonlocal` keyword:

How to Run:

- Save the code as `02_nested_scopes.py`
- Run: `python3 02_nested_scopes.py`

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | **NAMESPACES**

```python
# --- 02_nested_scopes.py ---
"""
Nested Scopes and the nonlocal Keyword

This file demonstrates how Python handles nested function scopes
and the usage of the 'nonlocal' keyword for modifying variables
in outer (but not global) scopes.
"""

def scope_test():
    def do_local():
        spam = "local spam"  # Creates a new local variable

    def do_nonlocal():
        nonlocal spam        # Refers to spam in scope_test
        spam = "nonlocal spam"

    def do_global():
```


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
        global spam         # Refers to global spam
        spam = "global spam"

    # Initialize spam in scope_test's scope
    spam = "test spam"
    print("Initial value:", spam)

    do_local()
    print("After local assignment:", spam)

    do_nonlocal()
    print("After nonlocal assignment:", spam)

    do_global()
    print("After global assignment:", spam)

print("Starting scope test\n")
scope_test()
print("\nIn global scope:", spam)  # Will print the global
    spam value
```

This example demonstrates:

- Local assignments do not affect variables in outer scopes

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

- `nonlocal` allows modifying variables in the nearest enclosing scope
- `global` allows modifying variables in the global scope
- Each function has its own local namespace

2.3. Class and Instance Namespaces

Python also has special namespaces for classes and instances. Let's explore how they work:

How to Run:

- Save the code as `03_class_namespaces.py`
- Run: `python3 03_class_namespaces.py`

```
# --- 03_class_namespaces.py ---
"""
Class and Instance Namespaces

This file demonstrates how Python handles namespaces in classes,
showing the difference between class variables (shared by all instances)
and instance variables (unique to each instance).
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | **NAMESPACES**

```
"""

class Dog:
    # Class variable - shared by all instances
    species = "Canis familiaris"

    # Class variable containing a mutable object (for demonstration)
    tricks = []  # Shared by all dogs!

    def __init__(self, name):
        # Instance variables - unique to each instance
        self.name = name
        self.individual_tricks = []  # Better - each dog has its own list

    def add_shared_trick(self, trick):
        # Modifies the class variable (affects all dogs)
        Dog.tricks.append(trick)

    def add_individual_trick(self, trick):
        # Modifies the instance variable (affects only
```


Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
    this dog)
28          self.individual_tricks.append(trick)
29
30 # Create two dogs
31 print("Creating two dogs...\n")
32 fido = Dog("Fido")
33 buddy = Dog("Buddy")
34
35 # Demonstrate class variable sharing
36 print("Class variable demonstration:")
37 print(f"Fido's species: {fido.species}")
38 print(f"Buddy's species: {buddy.species}")
39
40 # Change the class variable
41 Dog.species = "Canis lupus familiaris"
42 print("\nAfter changing class variable:")
43 print(f"Fido's species: {fido.species}")
44 print(f"Buddy's species: {buddy.species}")
45
46 # Demonstrate instance variable independence
47 print("\nInstance variable demonstration:")
48 print(f"Fido's name: {fido.name}")
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
print(f"Buddy's name: {buddy.name}")

# Demonstrate the difference with mutable objects
print("\nShared tricks list (class variable):")
fido.add_shared_trick("roll over")
print(f"Fido's tricks: {Dog.tricks}")
print(f"Buddy's tricks: {buddy.tricks}  # Same list!")

print("\nIndividual tricks lists (instance variables):")
fido.add_individual_trick("play dead")
buddy.add_individual_trick("fetch")
print(f"Fido's individual tricks:
   {fido.individual_tricks}")
print(f"Buddy's individual tricks:
   {buddy.individual_tricks}")

# Demonstrate attribute lookup order
print("\nAttribute lookup demonstration:")
buddy.species = "Changed for buddy"  # Creates a new
   instance variable
print(f"Fido's species (from class): {fido.species}")
print(f"Buddy's species (from instance): {buddy.species}")
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
68 print(f"Class species attribute: {Dog.species}")
```

Key points about class and instance namespaces:

- Class variables are shared among all instances
- Instance variables are unique to each instance
- Be cautious with mutable class variables
- Python looks up attributes first in the instance namespace, then in the class namespace

3. Advanced Namespace Concepts: Closures and Annotations

Python's namespace system becomes even more powerful when working with closures and type annotations. These advanced features demonstrate the flexibility and sophistication of Python's scoping rules.

3.1. Function Closures and Free Variables

A closure occurs when a nested function references variables from its enclosing scope. The inner function "closes over" the variables it needs, preserving them even after the outer function returns.

How to Run:

- Save the code as `04_closures_annotations.py`

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | **NAMESPACES**

- Run: `python3 04_closures_annotations.py`

```python
# --- 04_closures_annotations.py ---
"""
Function Closures and Annotation Scopes

This file demonstrates Python's closure mechanism and how annotations
are handled in different scopes. It shows how free variables are
captured and how annotations are evaluated.
"""
from typing import Callable, List

def make_counter(start: int = 0) -> Callable[[], int]:
    """Creates a counter function with its own private state."""
    count = start

    def counter() -> int:
        nonlocal count
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
        current = count
        count += 1
        return current

    return counter

def make_multiplier(factor: float) -> Callable[[float], float]:
    """Creates a function that multiplies its input by a stored factor."""
    # 'factor' is a free variable captured by the closure
    def multiplier(x: float) -> float:
        return x * factor

    return multiplier

# Demonstrate closure behavior
print("Closure demonstration:")
counter1 = make_counter(5)
counter2 = make_counter(10)

print(f"Counter 1: {counter1()}, {counter1()},
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
    {counter1()}")
print(f"Counter 2: {counter2()}, {counter2()},
    {counter2()}")

# Demonstrate multiple closures with the same free variable
print("\nMultiplier demonstration:")
double = make_multiplier(2)
triple = make_multiplier(3)

print(f"Double 5: {double(5)}")
print(f"Triple 5: {triple(5)}")

# Demonstrate late binding behavior
def create_functions() -> List[Callable[[], int]]:
    """Demonstrates late binding behavior in closures."""
    functions = []
    for i in range(3):
        def func(captured_i=i):  # Use default argument
    for early binding
            return captured_i
        functions.append(func)
    return functions
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
56
57 print("\nLate binding demonstration:")
58 funcs = create_functions()
59 for f in funcs:
60     print(f"Function returns: {f()}")
61
62 # Demonstrate annotation scopes
63 def outer(x: 'TypeVar') -> None:
64     y: 'TypeVar'  # Forward reference in annotation
65
66     class TypeVar:
67         pass
68
69     y = TypeVar()  # This refers to the local TypeVar class
70     print(f"Type of y: {type(y).__name__}")
71
72 print("\nAnnotation scope demonstration:")
73 outer(None)  # The 'TypeVar' in the annotation is treated
      as a string
```

This example demonstrates several advanced concepts:

- Each function call creates a new instance of the local variables
- Closures can "remember" values from the enclosing scope

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

- The `nonlocal` keyword is needed to modify enclosed variables
- Type annotations in closures follow special scoping rules

3.2. Key Insights About Closures

When working with closures in Python:

- Each function call creates a new instance of the local variables
- Closures can "remember" values from the enclosing scope
- The `nonlocal` keyword is needed to modify enclosed variables
- Type annotations in closures follow special scoping rules

3.3. Best Practices with Closures

When using closures in your code:

- Use closures to create functions with private state
- Be careful with mutable free variables
- Consider using default arguments for early binding when needed
- Document the closure's behavior and any captured variables

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

4. Name Resolution Rules: LEGB

Python follows the LEGB rule when looking up names:

- **Local**: Names declared inside the current function
- **Enclosing**: Names in enclosing functions
- **Global**: Names declared at module level
- **Built-in**: Names in the built-in module

Python searches these namespaces in this order until it finds the name or raises a `NameError`.

5. Best Practices

When working with namespaces in Python:

- Use `global` sparingly - it can make code harder to understand
- Prefer passing values as arguments and returning results
- Be careful with mutable class variables
- Use clear and descriptive names to avoid confusion
- Document when you need to use `global` or `nonlocal`

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

6. References

- Python Documentation. (2025). *Python Scopes and Namespaces*. Link
- Python Documentation. (2025). *The global statement*. Link
- Python Documentation. (2025). *The nonlocal statement*. Link
- Lutz, M. (2023). *Learning Python, 6th Edition*. O'Reilly Media.

- This article was edited and written in collaboration with AI. If you find any inconsistencies or have suggestions for improvement, please don't hesitate to open an issue in our GitHub repository or reach out directly.

7. Explore More Python Concepts

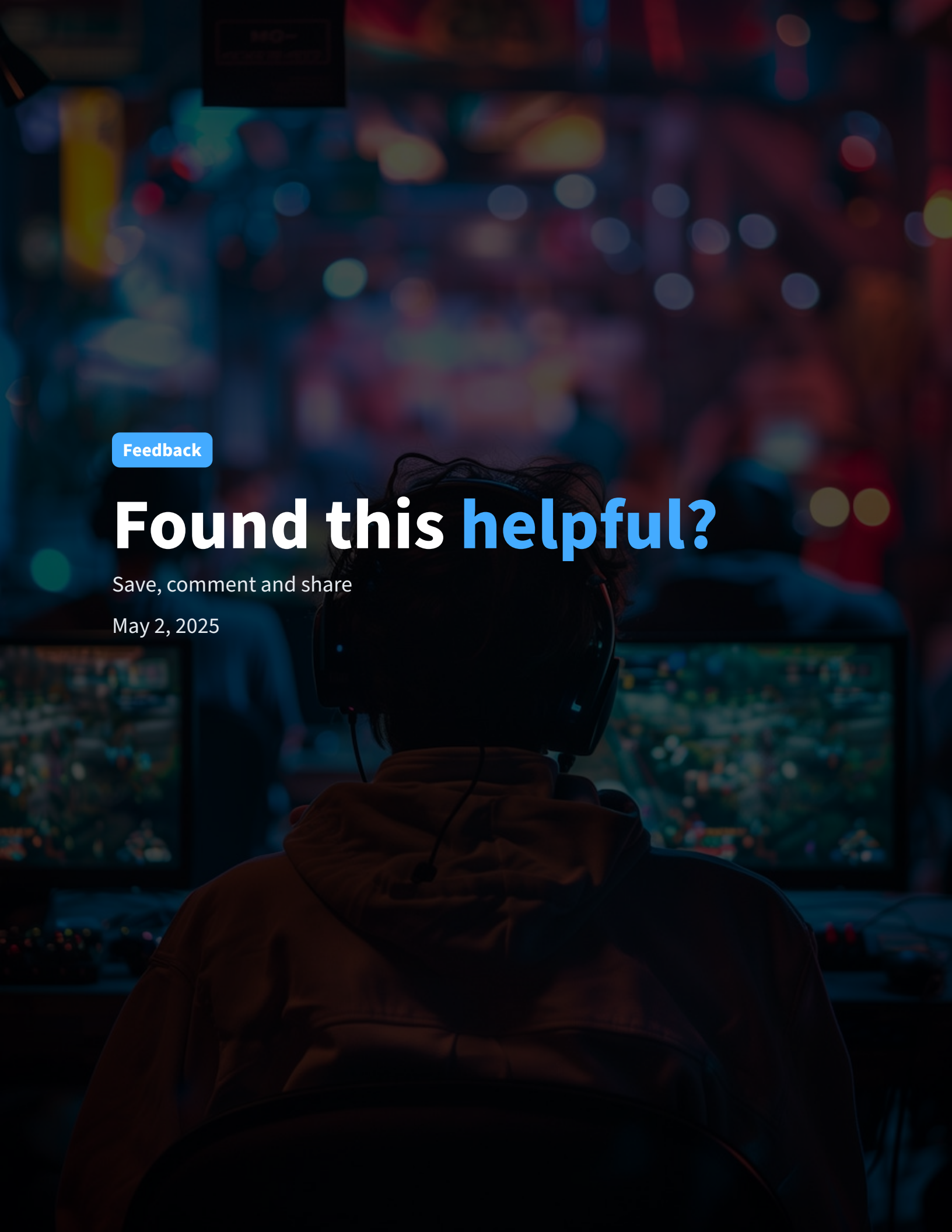
Enjoyed This Content?

Don't miss my other Python articles:

Python Decorators: A Comprehensive Guide

Learn how to write and use decorators in Python to enhance your functions and classes with additional functionality.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com



Feedback

Found this helpful?

Save, comment and share

May 2, 2025