

Python

Understanding Python Namespaces

A Deep Dive into Python's Name Resolution

May 2, 2025



Source Code

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

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



```
1 # --- 01_basic_namespaces.py ---
2 """
3 Basic Namespaces in Python
4
5 This file demonstrates the fundamental concept of
6 namespaces in Python.
7 """
8
9 # The built-in namespace contains functions like print,
10 # len, etc.
11 print("Built-in namespace example:")
12 print(f"Type of len function: {type(len)}")
13 print(f"ID of len function: {id(len)}")
14
15 # The global namespace of a module
16 x = 10
17 y = "hello"
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

www.asanchezyali.com

```
17
18 print("\nGlobal namespace example:")
19 print(f"x = {x}, type: {type(x)}, id: {id(x)}")
20 print(f"y = {y}, type: {type(y)}, id: {id(y)}")
21
22 # Local namespace in a function with proper global
23 # variable handling
24 def example_function():
25     # Declare x as global before using it
26     global x
27     z = 20 # Local variable
28     print("\nLocal namespace inside function:")
29     print(f"z = {z}, type: {type(z)}, id: {id(z)}")
30
31     # Now we can safely access and modify the global x
32     print(f"x from global namespace: {x}")
33     x = 30 # This modifies the global x
34     print(f"Modified global x: {x}")
35
36 # Call the function
37 print("\nBefore function call:")
38 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

```
1 # --- 02_nested_scopes.py ---
2 """
3 Nested Scopes and the nonlocal Keyword
4
5 This file demonstrates how Python handles nested function
6 scopes
7 and the usage of the 'nonlocal' keyword for modifying
8 variables
9
10 def scope_test():
11     def do_local():
12         spam = "local spam"    # Creates a new local variable
13
14     def do_nonlocal():
15         nonlocal spam        # Refers to spam in scope_test
16         spam = "nonlocal spam"
17
18     def do_global():
19         global spam
20         spam = "global spam"
21
22     do_local()
23     do_nonlocal()
24     do_global()
25
26     print("scope_test() local:", spam)
27
28     do_local()
29     do_nonlocal()
30     do_global()
31
32     print("scope_test() nonlocal:", spam)
33
34     do_local()
35     do_nonlocal()
36     do_global()
37
38     print("scope_test() global:", spam)
39
40     return spam
41
42 def main():
43     print("main() global:", spam)
44
45     scope_test()
46
47     print("main() global:", spam)
48
49     do_local()
50     do_nonlocal()
51     do_global()
52
53     print("main() global:", spam)
54
55     print("main() global:", spam)
56
57     do_local()
58     do_nonlocal()
59     do_global()
60
61     print("main() global:", spam)
62
63     print("main() global:", spam)
64
65     do_local()
66     do_nonlocal()
67     do_global()
68
69     print("main() global:", spam)
70
71     do_local()
72     do_nonlocal()
73     do_global()
74
75     print("main() global:", spam)
76
77     do_local()
78     do_nonlocal()
79     do_global()
80
81     print("main() global:", spam)
82
83     do_local()
84     do_nonlocal()
85     do_global()
86
87     print("main() global:", spam)
88
89     do_local()
90     do_nonlocal()
91     do_global()
92
93     print("main() global:", spam)
94
95     do_local()
96     do_nonlocal()
97     do_global()
98
99     print("main() global:", spam)
100
101     do_local()
102     do_nonlocal()
103     do_global()
104
105     print("main() global:", spam)
106
107     do_local()
108     do_nonlocal()
109     do_global()
110
111     print("main() global:", spam)
112
113     do_local()
114     do_nonlocal()
115     do_global()
116
117     print("main() global:", spam)
118
119     do_local()
120     do_nonlocal()
121     do_global()
122
123     print("main() global:", spam)
124
125     do_local()
126     do_nonlocal()
127     do_global()
128
129     print("main() global:", spam)
130
131     do_local()
132     do_nonlocal()
133     do_global()
134
135     print("main() global:", spam)
136
137     do_local()
138     do_nonlocal()
139     do_global()
140
141     print("main() global:", spam)
142
143     do_local()
144     do_nonlocal()
145     do_global()
146
147     print("main() global:", spam)
148
149     do_local()
150     do_nonlocal()
151     do_global()
152
153     print("main() global:", spam)
154
155     do_local()
156     do_nonlocal()
157     do_global()
158
159     print("main() global:", spam)
160
161     do_local()
162     do_nonlocal()
163     do_global()
164
165     print("main() global:", spam)
166
167     do_local()
168     do_nonlocal()
169     do_global()
170
171     print("main() global:", spam)
172
173     do_local()
174     do_nonlocal()
175     do_global()
176
177     print("main() global:", spam)
178
179     do_local()
180     do_nonlocal()
181     do_global()
182
183     print("main() global:", spam)
184
185     do_local()
186     do_nonlocal()
187     do_global()
188
189     print("main() global:", spam)
190
191     do_local()
192     do_nonlocal()
193     do_global()
194
195     print("main() global:", spam)
196
197     do_local()
198     do_nonlocal()
199     do_global()
200
201     print("main() global:", spam)
202
203     do_local()
204     do_nonlocal()
205     do_global()
206
207     print("main() global:", spam)
208
209     do_local()
210     do_nonlocal()
211     do_global()
212
213     print("main() global:", spam)
214
215     do_local()
216     do_nonlocal()
217     do_global()
218
219     print("main() global:", spam)
220
221     do_local()
222     do_nonlocal()
223     do_global()
224
225     print("main() global:", spam)
226
227     do_local()
228     do_nonlocal()
229     do_global()
230
231     print("main() global:", spam)
232
233     do_local()
234     do_nonlocal()
235     do_global()
236
237     print("main() global:", spam)
238
239     do_local()
240     do_nonlocal()
241     do_global()
242
243     print("main() global:", spam)
244
245     do_local()
246     do_nonlocal()
247     do_global()
248
249     print("main() global:", spam)
250
251     do_local()
252     do_nonlocal()
253     do_global()
254
255     print("main() global:", spam)
256
257     do_local()
258     do_nonlocal()
259     do_global()
260
261     print("main() global:", spam)
262
263     do_local()
264     do_nonlocal()
265     do_global()
266
267     print("main() global:", spam)
268
269     do_local()
270     do_nonlocal()
271     do_global()
272
273     print("main() global:", spam)
274
275     do_local()
276     do_nonlocal()
277     do_global()
278
279     print("main() global:", spam)
280
281     do_local()
282     do_nonlocal()
283     do_global()
284
285     print("main() global:", spam)
286
287     do_local()
288     do_nonlocal()
289     do_global()
290
291     print("main() global:", spam)
292
293     do_local()
294     do_nonlocal()
295     do_global()
296
297     print("main() global:", spam)
298
299     do_local()
300     do_nonlocal()
301     do_global()
302
303     print("main() global:", spam)
304
305     do_local()
306     do_nonlocal()
307     do_global()
308
309     print("main() global:", spam)
310
311     do_local()
312     do_nonlocal()
313     do_global()
314
315     print("main() global:", spam)
316
317     do_local()
318     do_nonlocal()
319     do_global()
320
321     print("main() global:", spam)
322
323     do_local()
324     do_nonlocal()
325     do_global()
326
327     print("main() global:", spam)
328
329     do_local()
330     do_nonlocal()
331     do_global()
332
333     print("main() global:", spam)
334
335     do_local()
336     do_nonlocal()
337     do_global()
338
339     print("main() global:", spam)
340
341     do_local()
342     do_nonlocal()
343     do_global()
344
345     print("main() global:", spam)
346
347     do_local()
348     do_nonlocal()
349     do_global()
350
351     print("main() global:", spam)
352
353     do_local()
354     do_nonlocal()
355     do_global()
356
357     print("main() global:", spam)
358
359     do_local()
360     do_nonlocal()
361     do_global()
362
363     print("main() global:", spam)
364
365     do_local()
366     do_nonlocal()
367     do_global()
368
369     print("main() global:", spam)
370
371     do_local()
372     do_nonlocal()
373     do_global()
374
375     print("main() global:", spam)
376
377     do_local()
378     do_nonlocal()
379     do_global()
380
381     print("main() global:", spam)
382
383     do_local()
384     do_nonlocal()
385     do_global()
386
387     print("main() global:", spam)
388
389     do_local()
390     do_nonlocal()
391     do_global()
392
393     print("main() global:", spam)
394
395     do_local()
396     do_nonlocal()
397     do_global()
398
399     print("main() global:", spam)
400
401     do_local()
402     do_nonlocal()
403     do_global()
404
405     print("main() global:", spam)
406
407     do_local()
408     do_nonlocal()
409     do_global()
410
411     print("main() global:", spam)
412
413     do_local()
414     do_nonlocal()
415     do_global()
416
417     print("main() global:", spam)
418
419     do_local()
420     do_nonlocal()
421     do_global()
422
423     print("main() global:", spam)
424
425     do_local()
426     do_nonlocal()
427     do_global()
428
429     print("main() global:", spam)
430
431     do_local()
432     do_nonlocal()
433     do_global()
434
435     print("main() global:", spam)
436
437     do_local()
438     do_nonlocal()
439     do_global()
440
441     print("main() global:", spam)
442
443     do_local()
444     do_nonlocal()
445     do_global()
446
447     print("main() global:", spam)
448
449     do_local()
450     do_nonlocal()
451     do_global()
452
453     print("main() global:", spam)
454
455     do_local()
456     do_nonlocal()
457     do_global()
458
459     print("main() global:", spam)
460
461     do_local()
462     do_nonlocal()
463     do_global()
464
465     print("main() global:", spam)
466
467     do_local()
468     do_nonlocal()
469     do_global()
470
471     print("main() global:", spam)
472
473     do_local()
474     do_nonlocal()
475     do_global()
476
477     print("main() global:", spam)
478
479     do_local()
480     do_nonlocal()
481     do_global()
482
483     print("main() global:", spam)
484
485     do_local()
486     do_nonlocal()
487     do_global()
488
489     print("main() global:", spam)
490
491     do_local()
492     do_nonlocal()
493     do_global()
494
495     print("main() global:", spam)
496
497     do_local()
498     do_nonlocal()
499     do_global()
499
500     print("main() global:", spam)
```

**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

www.asanchezyali.com

```
19     global spam          # Refers to global spam
20     spam = "global spam"
21
22     # Initialize spam in scope_test's scope
23     spam = "test spam"
24     print("Initial value:", spam)
25
26     do_local()
27     print("After local assignment:", spam)
28
29     do_nonlocal()
30     print("After nonlocal assignment:", spam)
31
32     do_global()
33     print("After global assignment:", spam)
34
35 print("Starting scope test\n")
36 scope_test()
37 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`



```
1 # --- 03_class_namespaces.py ---
2 """
3 Class and Instance Namespaces
4
5 This file demonstrates how Python handles namespaces in
6 classes,
7 showing the difference between class variables (shared by
8 all instances)
9 and instance variables (unique to each instance).
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

www.asanchezyali.com

PYTHON | NAMESPACES

```
8 """
9
10 class Dog:
11     # Class variable - shared by all instances
12     species = "Canis familiaris"
13
14     # Class variable containing a mutable object (for
15     # demonstration)
15     tricks = [] # Shared by all dogs!
16
17     def __init__(self, name):
18         # Instance variables - unique to each instance
19         self.name = name
20         self.individual_tricks = [] # Better - each dog
has its own list
21
22     def add_shared_trick(self, trick):
23         # Modifies the class variable (affects all dogs)
24         Dog.tricks.append(trick)
25
26     def add_individual_trick(self, trick):
27         # 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

```
49 print(f"Buddy's name: {buddy.name}")  
50  
51 # Demonstrate the difference with mutable objects  
52 print("\nShared tricks list (class variable):")  
53 fido.add_shared_trick("roll over")  
54 print(f"Fido's tricks: {Dog.tricks}")  
55 print(f"Buddy's tricks: {buddy.tricks} # Same list!")  
56  
57 print("\nIndividual tricks lists (instance variables):")  
58 fido.add_individual_trick("play dead")  
59 buddy.add_individual_trick("fetch")  
60 print(f"Fido's individual tricks:  
      {fido.individual_tricks}")  
61 print(f"Buddy's individual tricks:  
      {buddy.individual_tricks}")  
62  
63 # Demonstrate attribute lookup order  
64 print("\nAttribute lookup demonstration:")  
65 buddy.species = "Changed for buddy" # Creates a new  
    instance variable  
66 print(f"Fido's species (from class): {fido.species}")  
67 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

```
● ● ●

1 # --- 04_closures_annotations.py ---
2 """
3 Function Closures and Annotation Scopes
4
5 This file demonstrates Python's closure mechanism and how
6 annotations
7 are handled in different scopes. It shows how free
8 variables are
9 captured and how annotations are evaluated.
10
11 def make_counter(start: int = 0) -> Callable[[], int]:
12     """Creates a counter function with its own private
13     state."""
14
15     def counter() -> int:
16         nonlocal count
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

🌐 www.asanchezyali.com

```
17     current = count
18     count += 1
19     return current
20
21 return counter
22
23 def make_multiplier(factor: float) -> Callable[[float], float]:
24     """Creates a function that multiplies its input by a
25     stored factor."""
26     # 'factor' is a free variable captured by the closure
27     def multiplier(x: float) -> float:
28         return x * factor
29
30     return multiplier
31
32 # Demonstrate closure behavior
33 print("Closure demonstration:")
34 counter1 = make_counter(5)
35 counter2 = make_counter(10)
36 print(f"Counter 1: {counter1()}, {counter1()},
```

**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

www.asanchezyali.com

```
    {counter1()}"')
37 print(f"Counter 2: {counter2()}, {counter2()},
    {counter2()}")
38
39 # Demonstrate multiple closures with the same free variable
40 print("\nMultiplier demonstration:")
41 double = make_multiplier(2)
42 triple = make_multiplier(3)
43
44 print(f"Double 5: {double(5)}")
45 print(f"Triple 5: {triple(5)}")
46
47 # Demonstrate late binding behavior
48 def create_functions() -> List[Callable[[], int]]:
49     """Demonstrates late binding behavior in closures."""
50     functions = []
51     for i in range(3):
52         def func(captured_i=i): # Use default argument
53             for early binding
54                 return captured_i
55         functions.append(func)
56     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