🐍 21-Day Python Puzzle Challenge - Complete Documentation

Master Python's quirky behaviors and hidden features through mind-bending puzzles!

🚨 Day 1: Boolean Arithmetic Magic

🎯 Problem Statement

What happens when you perform arithmetic operations on boolean values in Python?

💻 Code Solution

```python
a = True
b = False
print(a + b + a)
```

📤 Expected Output

```
2
```

🔍 Detailed Explanation

In Python, boolean values are actually subclasses of integers:

- `True` has a numeric value of `1`
- `False` has a numeric value of `0`

When arithmetic operations are performed:

- `a + b + a` becomes `1 + 0 + 1 = 2`

This behavior exists because `bool` inherits from `int` in Python's type hierarchy.

💡 Pro Tips & Interview Tips

- **Interview Insight**: This demonstrates Python's duck typing and type coercion
- **Best Practice**: Avoid mixing booleans with arithmetic unless intentional
- **Memory Trick**: True = 1, False = 0 (think binary!)

🚨 Day 2: Division Operators Demystified

🎯 Problem Statement

Understanding the difference between floor division and true division operators.

💻 Code Solution

```python
print(7 // 2)  # Floor division
print(7 / 2)   # True division
```

📤 Expected Output

```
3
3.5
```

🔍 Detailed Explanation

Python has two division operators:

- `//` (Floor Division): Rounds down to the nearest integer
- `/` (True Division): Returns exact decimal result

Floor division always returns an integer (or float with .0), while true division returns a float.

💡 Pro Tips & Interview Tips

- **Interview Insight**: Floor division behavior differs between Python 2 and 3
- **Edge Case**: `7 // 2.0` returns `3.0` (float), not `3` (int)
- **Use Case**: Floor division is perfect for pagination calculations

---

🚨 Day 3: String Comparison Gotcha

🎯 Problem Statement

How does Python compare strings lexicographically?

💻 Code Solution

```python
result = "10" > "9"
print(result)
```

📤 Expected Output

```
False
```

🔍 Detailed Explanation

String comparison in Python is lexicographic (dictionary order), not numeric:

- Compares character by character using ASCII values
- `"1"` (ASCII 49) vs `"9"` (ASCII 57)
- Since 49 < 57, `"10" < "9"` returns `True`, so `"10" > "9"` is `False`

💡 Pro Tips & Interview Tips

- **Interview Trap**: Classic misconception between string and numeric comparison
- **Solution**: Use `int("10") > int("9")` for numeric comparison
- **Real World**: Always validate input types when comparing user data

---

🚨 Day 4: Object Identity vs Equality

🎯 Problem Statement

Understanding the difference between `is` and `==` operators with string interning.

💻 Code Solution

```python
a = "hello"
b = "hello"
print(a is b)
```

📤 Expected Output

```
True
```

🔍 Detailed Explanation

Python interns small strings for memory optimization:

- String literals that look like identifiers are automatically interned
- `is` checks object identity (same memory location)
- `==` checks value equality
- Since "hello" is interned, both variables point to the same object

💡 Pro Tips & Interview Tips

- **Interview Gold**: Demonstrates understanding of Python's memory management

- **Gotcha**: `sys.intern()` can force interning of any string
- **Best Practice**: Use `==` for value comparison, `is` only for `None`, `True`, `False`

🚨 Day 5: List Operations and Operator Precedence

🎯 Problem Statement

Understanding list concatenation and multiplication operator precedence.

💻 Code Solution

```python
list1 = [1, 2, 3]
list2 = [4, 5]
print(list1 + list2 * 2)
```

📤 Expected Output

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

🔍 Detailed Explanation

Operator precedence matters:

1. `*` has higher precedence than `+`
2. `list2 * 2` is evaluated first: `[4, 5, 4, 5]`
3. Then concatenation: `[1, 2, 3] + [4, 5, 4, 5]`

List multiplication creates repeated references to the same elements.

💡 Pro Tips & Interview Tips

- **Memory Trap**: `[[]] * 3` creates three references to the same list!
- **Solution**: Use list comprehension: `[[] for _ in range(3)]`
- **Precedence Rule**: PEMDAS applies to Python operators too

---

🚨 Day 6: Truthiness in Python

🎯 Problem Statement

Understanding which values are considered "falsy" in Python.

💻 Code Solution

```python
print(bool(0))      # Numeric zero
print(bool(""))     # Empty string
print(bool([]))     # Empty list
print(bool("False")) # Non-empty string
```

📤 Expected Output

```
False
False
False
True
```

🔍 Detailed Explanation

Python's falsy values:

- Numeric zeros: `0`, `0.0`, `0j`
- Empty collections: `""`, `[]`, `{}`, `set()`
- `None` and `False`

Everything else is truthy, including the string `"False"`!

💡 Pro Tips & Interview Tips

- **Interview Favorite**: "What values are falsy in Python?"
- **Gotcha**: `"0"` is truthy (non-empty string)
- **Best Practice**: Use explicit comparisons: `if len(my_list) > 0:` instead of `if my_list:`

🚨 Day 7: Walrus Operator (Assignment Expression)

🎯 Problem Statement

Understanding Python 3.8's walrus operator `:=` for assignment expressions.

💻 Code Solution

```python
if (n := 5) > 3:
    print(n)
```

📤 Expected Output

```
5
```

🔍 Detailed Explanation

The walrus operator `:=` assigns and returns a value in one expression:

- `(n := 5)` assigns `5` to `n` and returns `5`
- The condition `5 > 3` is `True`
- `n` is now available in the scope and equals `5`

💡 Pro Tips & Interview Tips

- **Version Note**: Only available in Python 3.8+
- **Use Case**: Perfect for avoiding repeated expensive operations
- **Style Tip**: Always use parentheses for clarity: `(var := expression)`

🚨 Day 8: Integer Caching and Identity

🎯 Problem Statement

Python's integer caching mechanism for small numbers.

💻 Code Solution

```python
a = 256
b = 256
print(a is b)  # True - cached

c = 257
d = 257
print(c is d)  # False - not cached
```

📤 Expected Output

```
True
False
```

🔍 Detailed Explanation

Python caches integers from -5 to 256 for performance:

- Small integers are pre-created and reused

- `256` and below: same object identity
- `257` and above: new objects created each time
- This is an implementation detail, not part of the language spec

💡 Pro Tips & Interview Tips

- **Interview Deep Dive**: Shows understanding of Python's memory optimization
- **Platform Dependent**: Range might vary between Python implementations
- **Never Rely**: Don't write code dependent on this behavior

---

🚨 Day 9: Dictionary Keys and Hash Collision

🎯 Problem Statement

Understanding how Python handles hash collisions with boolean and integer keys.

💻 Code Solution

```python
d = {True: "yes", 1: "one", False: "no", 0: "zero"}
print(list(d.keys()))
print(list(d.values()))
```

📤 Expected Output

```
[True, False]
['one', 'zero']
```

🔍 Detailed Explanation

Hash collision behavior:

- `True` and `1` have the same hash value
- `False` and `0` have the same hash value
- Later keys with same hash overwrite earlier values
- Original keys are preserved, but values are updated

💡 Pro Tips & Interview Tips

- **Hash Function**: `hash(True) == hash(1)` returns `True`
- **Dictionary Rule**: Keys must be unique by hash AND equality
- **Real World**: Be careful mixing numeric types as dictionary keys

---

🚨 Day 10: Generator Exhaustion

🎯 Problem Statement

Understanding how generators can only be consumed once.

💻 Code Solution

```python
gen = (x for x in range(3))
print(list(gen))  # Consumes generator
print(list(gen))  # Generator is exhausted
```

📤 Expected Output

```
[0, 1, 2]
[]
```

🔍 Detailed Explanation

Generators are iterators that produce values lazily:

- First `list(gen)` consumes all values: `[0, 1, 2]`
- Generator is now exhausted (internal pointer at end)
- Second `list(gen)` returns empty list
- To reuse, create a new generator or use `itertools.tee()`

💡 Pro Tips & Interview Tips

- **Memory Efficient**: Generators don't store all values in memory
- **One-Shot**: Unlike lists, generators can't be reset
- **Solution**: Store as list if you need multiple iterations

---

🚨 Day 11: List Comprehension Scope

🎯 Problem Statement

Understanding variable scope in list comprehensions.

💻 Code Solution

```python
```

```
x = 10
result = [x for x in range(3)]
print(x)
```

📤 Expected Output

```
10
```

🔍 Detailed Explanation

List comprehensions have their own scope in Python 3:

- The `x` inside the comprehension is local to that comprehension
- The outer `x = 10` is not modified
- In Python 2, this would print `2` (scope leakage bug)

💡 Pro Tips & Interview Tips

- **Version Difference**: Major improvement from Python 2 to 3
- **Scope Rule**: Comprehension variables don't leak to outer scope
- **Best Practice**: Use different variable names to avoid confusion

---

🚨 Day 12: Slice Assignment Magic

🎯 Problem Statement

Understanding how slice assignment can change list length.

💻 Code Solution

python

```python
numbers = [1, 2, 3, 4, 5]
numbers[1:4] = [10]  # Replace slice with single element
print(numbers)
```

📤 Expected Output

```
[1, 10, 5]
```

🔍 Detailed Explanation

Slice assignment replaces the entire slice with new values:

- `numbers[1:4]` selects elements `[2, 3, 4]` (indices 1, 2, 3)
- Assignment replaces these 3 elements with 1 element `[10]`
- List shrinks from 5 elements to 3 elements
- Final result: `[1, 10, 5]`

💡 Pro Tips & Interview Tips

- **Dynamic Length**: Slice assignment can grow or shrink lists
- **Efficiency**: More efficient than multiple `insert`/`delete` operations
- **Use Case**: Perfect for replacing multiple elements with different count

---

🚨 Day 13: Dictionary get() Method

🎯 Problem Statement

Understanding the dictionary `get()` method and default values.

💻 Code Solution

```python
data = {'name': 'Alice', 'age': 30}
print(data.get('city', 'Unknown'))  # Key doesn't exist
print(data.get('name'))           # Key exists
```

📤 Expected Output

```
Unknown
Alice
```

🔍 Detailed Explanation

The `get()` method safely accesses dictionary values:

- `get(key, default)` returns value if key exists, otherwise returns default
- `get(key)` returns value if key exists, otherwise returns `None`
- Prevents `KeyError` exceptions
- More elegant than using `try/except` blocks

💡 Pro Tips & Interview Tips

- **Safe Access**: Prevents crashes when key doesn't exist
- **Default Values**: Second parameter sets fallback value

- **Alternative**: `data.setdefault('city', 'Unknown')` also sets the key

🚨 Day 14: Mutable Default Arguments

🎯 Problem Statement

The classic "mutable default argument" trap in Python functions.

💻 Code Solution

```python
def append_item(item, a_list=[]):
    a_list.append(item)
    return a_list

print(append_item(1))  # [1]
print(append_item(2))  # [1, 2] - Same list object!
```

📤 Expected Output

```
[1]
[1, 2]
```

🔍 Detailed Explanation

Default arguments are evaluated once at function definition time:

- The empty list `[]` is created once and reused
- Each function call modifies the same list object
- First call appends `1`: `[1]`
- Second call appends `2` to existing list: `[1, 2]`

Correct Solution:

```python
def append_item(item, a_list=None):
    if a_list is None:
        a_list = []
    a_list.append(item)
    return a_list
```

💡 Pro Tips & Interview Tips

- **Classic Trap**: Most common Python interview question
- **Rule**: Never use mutable objects as default arguments
- **Solution**: Use `None` and create object inside function

🚨 Day 15: Generator Exhaustion (Repeat)

🎯 Problem Statement

Reinforcing the concept of generator exhaustion.

💻 Code Solution

```python
gen = (x for x in range(3))
print(list(gen))  # [0, 1, 2]
print(list(gen))  # []
```

📤 Expected Output

```
[0, 1, 2]
[]
```

🔍 Detailed Explanation

Same concept as Day 10 - generators are single-use iterators that become exhausted after consumption.

💡 Pro Tips & Interview Tips

- **Repetition Learning**: Key concepts deserve reinforcement
- **Memory Management**: Understanding iterators vs iterables is crucial
- **Interview Tip**: Be able to explain the difference between generators and lists

---

🚨 Day 16: String Concatenation and eval()

🎯 Problem Statement

Understanding string concatenation and the powerful (dangerous) `eval()` function.

💻 Code Solution

```python
x = "3" + "7"  # String concatenation
print(eval(x))  # Evaluates "37" as Python code
```

📤 Expected Output

```
37
```

🔍 Detailed Explanation

Two-step process:

1. `"3" + "7"` creates string `"37"` (concatenation, not addition)
2. `eval("37")` evaluates the string as Python code, returning integer `37`

Security Warning: `eval()` executes arbitrary code and is dangerous with user input!

💡 Pro Tips & Interview Tips

- **Security Risk**: Never use `eval()` with untrusted input
- **Safe Alternative**: Use `ast.literal_eval()` for safe evaluation
- **String vs Number**: Pay attention to data types in operations

---

🚨 Day 17: Class Variables and Instance Counting

🎯 Problem Statement

Understanding class variables vs instance variables and object counting.

💻 Code Solution

```python
class A:
    count = 0  # Class variable

    def __init__(self):
        A.count += 1  # Increment class variable

a = A()  # count becomes 1
b = A()  # count becomes 2
print(A.count)
```

📤 Expected Output

```
2
```

🔍 Detailed Explanation

Class variables are shared among all instances:

- `count` belongs to the class `A`, not individual instances
- Each `__init__` call increments the shared counter
- Accessible via class name: `A.count`
- Common pattern for counting instances

💡 Pro Tips & Interview Tips

- **Class vs Instance**: Class variables are shared, instance variables are unique
- **Access Methods**: `A.count` or `self.__class__.count`
- **Use Case**: Singleton pattern, instance counting, shared configuration

---

🚨 Day 18: Method Inheritance and super()

🎯 Problem Statement

Understanding method inheritance and the `super()` function.

💻 Code Solution

```python
class A:
    def greet(self):
        return "Hi"

class B(A):  # B inherits from A
    def greet(self):
        return super().greet() + " there"

print(B().greet())
```

📤 Expected Output

```
Hi there
```

🔍 Detailed Explanation

Method inheritance and extension:

- Class `B` inherits from class `A`
- `B.greet()` overrides `A.greet()` but extends it
- `super().greet()` calls parent class method: returns `"Hi"`

- String concatenation: `"Hi" + " there"` = `"Hi there"`

💡 Pro Tips & Interview Tips

- **Method Resolution Order (MRO)**: `super()` follows the MRO chain
- **Diamond Problem**: `super()` handles multiple inheritance correctly
- **Best Practice**: Use `super()` instead of calling parent class directly

---

🚨 Day 19: Closure and Late Binding

🎯 Problem Statement

Understanding closures and the "late binding" behavior in loops.

💻 Code Solution

```python
funcs = []
for i in range(3):
    funcs.append(lambda: i)  # Closure captures variable reference

print([f() for f in funcs])  # All functions return the final value of i
```

📤 Expected Output

```
[2, 2, 2]
```

🔍 Detailed Explanation

Late binding closure trap:

- Lambda functions capture the variable `i` by reference, not value
- After loop completes, `i` equals `2`
- All lambda functions reference the same `i` variable
- When called, they all return the final value: `2`

Correct Solution:

```python
funcs = [lambda x=i: x for i in range(3)]
# or
funcs = [lambda i=i: i for i in range(3)]
```

💡 Pro Tips & Interview Tips

- **Classic Trap**: Very common in JavaScript too
- **Closure Rule**: Captures variables by reference, not value
- **Solution**: Use default parameters to capture values

🚨 Day 20: Tuple Unpacking Variants

🎯 Problem Statement

Understanding different forms of sequence unpacking with single elements.

💻 Code Solution

```python
a, = (1,)    # Tuple unpacking
print(a)     # 1

b, = [2]     # List unpacking
print(b)     # 2

c, = 3       # This will cause an error!
print(c)
```

📤 Expected Output

```
1
2
TypeError: cannot unpack non-sequence int
```

🔍 Detailed Explanation

Sequence unpacking rules:

- `a, = (1,)`: Unpacks single-element tuple ✅
- `b, = [2]`: Unpacks single-element list ✅
- `c, = 3`: Tries to unpack integer (not iterable) ❌

The comma after the variable name indicates unpacking assignment.

💡 Pro Tips & Interview Tips

- **Syntax Meaning**: Comma makes it unpacking, not regular assignment
- **Iterable Requirement**: Right side must be iterable

- **Common Use**: `first, *rest = my_list` for splitting sequences

🚨 Day 21: Advanced Generators with send()

🎯 Problem Statement

Understanding advanced generator features with `yield` expressions and `send()` method.

💻 Code Solution

```python
def gen():
    x = yield "Start"  # yield expression receives sent value
    yield x * 2


g = gen()
print(next(g))      # Get first yielded value
print(g.send(10))   # Send value to generator
```

📤 Expected Output

```
Start
20
```

🔍 Detailed Explanation

Advanced generator communication:

1. `next(g)` starts generator, executes until first `yield "Start"`
2. Generator pauses at `x = yield "Start"`, waiting for input
3. `g.send(10)` sends `10` to the generator, which becomes the value of `x`
4. Generator continues: `yield x * 2` becomes `yield 10 * 2` = `20`

💡 Pro Tips & Interview Tips

- **Two-way Communication**: Generators can receive values via `send()`
- **First Call**: Must use `next()` or `send(None)` for first advance
- **Advanced Pattern**: Used in coroutines and async programming
- **Real World**: Foundation for async/await syntax

🎯 Challenge Complete!

Congratulations on completing the 21-day Python Puzzle Challenge! You've mastered:

- ✅ Boolean arithmetic and type coercion
- ✅ String vs numeric operations
- ✅ Memory management and object identity
- ✅ Generator behavior and exhaustion
- ✅ Scope rules and variable binding
- ✅ Class vs instance variables
- ✅ Inheritance and method resolution
- ✅ Closure traps and late binding
- ✅ Sequence unpacking patterns
- ✅ Advanced generator features

🏆 Final Pro Tips for Interviews

1. **Understand the Why**: Don't just memorize outputs, understand the underlying mechanisms
2. **Practice Edge Cases**: These puzzles represent common gotchas in real code
3. **Explain Your Thinking**: Walk through your reasoning step by step
4. **Know Python Versions**: Some behaviors differ between Python 2 and 3
5. **Memory Management**: Understanding object identity vs equality shows deep knowledge

Keep practicing and happy coding! 🐍✨