20 Python Concepts I Wish I Knew Way Earlier

Shivam Kumar · Follow
Published in Stackademic · 8 min read · Aug 30

Photo by Shahadat Rahman on Unsplash

Have you ever stumbled upon a Python feature and thought, “I wish I knew this earlier!”? You’re not alone! In this blog, I’am share 20 Python concepts that would’ve made my coding journey smoother. Dive in, and maybe you’ll discover a few gems you’ve missed!

1. List Comprehensions

Importance:
List comprehensions provide a concise way to create lists, which is beneficial for both readability and, in many cases, performance. It reduces the need for multi-line loops.

When to Use:
Use list comprehensions when you want to transform or filter data, particularly when the logic is simple. They’re suitable for small operations on data sets.

Example:
If we need to find squares of all even numbers in a range, we can use:

```
squares = [x**2 for x in range(10) if x % 2 == 0]
```

Potential Pitfalls:
Avoid using nested list comprehensions as they can reduce readability. Moreover, if the logic becomes complex, it’s better to use a for loop.

Real-world Scenario:
Imagine processing user input from a website, where you need to extract only the numeric inputs:

```
inputs = ["John", "23", "Doe", "45"]
ages = [int(x) for x in inputs if x.isdigit()]
```

2. Lambda Functions

Importance:
Lambda functions are useful for writing small, throwaway functions without the need for a formal function definition.

When to Use:
They're particularly handy when you need a simple function for a short period, and you won't reuse it. Commonly used with `map()`, `filter()`, and `sorted()`.

***Example**:*
Sorting a list of strings based on their length:

```
words = ["apple", "banana", "cherry", "date"]
sorted_words = sorted(words, key=lambda x: len(x))
```

Potential Pitfalls:
Lambda functions can reduce readability when overused or when the logic becomes complex. In such cases, it's better to define a proper function.

***Real-world Scenario**:*
Imagine filtering out products from an inventory based on a minimum price:

```
products = [{"name": "A", "price": 50}, {"name": "B", "price": 30}]
filtered_products = filter(lambda x: x['price'] > 40, products)
```

3. Map, Filter, and Reduce

Importance:
These functions offer a functional approach to processing collections. They reduce the need for explicit loops, resulting in cleaner code.

When to Use:
- `map()`: When you want to apply a function to every item of a collection.
- `filter()`: When you need to select items based on a predicate.
- `reduce()`: When you want to cumulatively apply a function to items, reducing the sequence to a single value.

***Example**:*
Using `map()` to convert strings to upper case:

```
names = ["alice", "bob", "charlie"]
upper_names = list(map(str.upper, names))
```

Potential Pitfalls:
Remember that `map()` and `filter()` return iterators in Python 3.x. To get a list, you need to convert them using `list()`.

Real-world Scenario:
Calculating the total price of items in a shopping cart:

```
from functools import reduce
cart = [{"name": "item1", "price": 50}, {"name": "item2", "price": 100}]
total = reduce(lambda x, y: x + y['price'], cart, 0)
```

4. Decorators

Importance:
Decorators allow you to extend and modify the behavior of callable objects like functions and methods without permanently modifying the callable itself.

When to Use:
When you want to add functionalities to existing code or when you want to modify the behavior of a function without changing its source code.

***Example**:*
A simple decorator to measure the time taken by a function to execute:

```
import time
def timer_decorator(func):
 def wrapper(*args, **kwargs):
 start_time = time.time()
 result = func(*args, **kwargs)
 end_time = time.time()
 print(f"{func.__name__} executed in {end_time - start_time} seconds")
 return result
 return wrapper
@timer_decorator
def sample_function():
 time.sleep(2)
```

Potential Pitfalls:
Overusing decorators or stacking many of them can make code harder to understand.

Real-world Scenario:
Using decorators in web frameworks like Flask to manage routes or permissions.

5. Generators

Importance:
Generators provide a way to iterate over large datasets without loading everything into memory. They produce items on-the-fly and can be more memory-efficient.

When to Use:
For large datasets, streams, or when you need to represent infinite sequences.

***Example**:*
A generator to produce Fibonacci sequence:

```
def fibonacci(n):
 a, b = 0, 1
 for _ in range(n):
 yield a
 a, b = b, a + b
```

Potential Pitfalls:
Generators are iterators; once consumed, they can't be reused.

Real-world Scenario:
Streaming data from a large log file without loading the entire file into memory.

6. f-Strings

Importance:
Introduced in Python 3.6, f-strings provide a concise and convenient way to embed expressions inside string literals.

When to Use:
Whenever you want to embed variable values inside strings or when formatting strings.

Example:

```
name = "Alice"
greeting = f"Hello, {name}!"
```

Potential Pitfalls:
Watch out for potential string injection attacks if you're including user input inside f-strings.

Real-world Scenario:
Dynamically generating SQL queries, though be cautious about SQL injection attacks.

7. *args and **kwargs

Importance:
Allows you to pass a variable number of arguments to a function, offering flexibility.

When to Use:
When you're not sure about the number of arguments, or when designing functions/methods for a broad range of use cases.

Example:
A function that multiplies all given arguments:

```
def multiply(*args):
 result = 1
 for num in args:
 result *= num
 return result
```

Potential Pitfalls:
Overloading a function with too many responsibilities can make it hard to understand or maintain.

Real-world Scenario:
Building wrappers around APIs where you might need to pass different parameters based on endpoint requirements.

8. Type Hinting

Importance:
Introduced in Python 3.5, type hinting helps in making the code more readable and allows for better IDE support and static type checking.

When to Use:
For enhancing code clarity, especially in larger projects or libraries meant for public consumption.

Example

```
def greet(name: str) -> str:
 return f"Hello, {name}!"
```

Potential Pitfalls:
Python remains a dynamically typed language. Type hints are just hints and won't enforce type checking unless you use tools like `mypy`.

Real-world Scenario:
In codebases where multiple developers work and you need to ensure clarity regarding function expectations.

9. Context Managers (with statement)

Importance:
Context managers ensure resources are efficiently managed and properly closed after usage, making code cleaner and resource management more foolproof.

When to Use:
When working with resources like files, databases, or network connections that require proper setup and teardown.

Example:
Opening and reading a file:

```
with open('file.txt', 'r') as f:
 content = f.read()
```

Potential Pitfalls:
Forgetting to use the `with` statement when it's beneficial can lead to resources not being released, potentially causing memory leaks or other issues.

Real-world Scenario:
Handling database connections to ensure they're properly closed, even if exceptions occur.

10. Walrus Operator (:=)

Importance:
Introduced in Python 3.8, the walrus operator helps assign values to variables as part of an expression.

When to Use:
Useful when you need both a value from an expression and want to retain that value for later use.

Example:
Reading lines from a file until a blank line is found:

```
with open('file.txt', 'r') as f:
 while (line := f.readline().strip()):
 print(line)
```

Potential Pitfalls:
Overusing it can make code harder to read for those not familiar with the operator.

Real-world Scenario:
Parsing through logs and breaking when a certain pattern is identified.

11. Namedtuples

Importance:
Namedtuples create simple classes for storing data, making code more self-documenting.

When to Use:
When you need a lightweight, immutable data structure.

Example:

```
from collections import namedtuple
Person = namedtuple('Person', ['name', 'age'])
alice = Person(name="Alice", age=30)
```

Potential Pitfalls:
Since they're immutable, you can't modify them after creation. For mutable structures, consider using data classes (Python 3.7+).

Real-world Scenario:
Representing a data point, like coordinates or RGB values.

12. Enumeration (enumerate)

Importance:
`enumerate()` lets you loop over an iterable and have an automatic counter, making code clearer.

When to Use:
Whenever you need both the index and value during iterations.

Example:

```
names = ["Alice", "Bob", "Charlie"]
for index, name in enumerate(names):
 print(f"{index}: {name}")
```

Potential Pitfalls:
None, really. It's a neat utility to keep code clear.

Real-world Scenario:
Displaying rankings or serial numbers alongside items in a list.

13. Zipping and Unzipping Lists

Importance:
`zip()` allows combining multiple iterables, making it easier to loop through multiple lists in parallel.

When to Use:
When you need to iterate simultaneously through multiple sequences.

Example:

```
names = ["Alice", "Bob"]
scores = [85, 92]
for name, score in zip(names, scores):
 print(f"{name}: {score}")
```

Potential Pitfalls:
`zip()` stops at the shortest input list. For different-sized iterables, consider using `itertools.zip_longest()`.

Real-world Scenario:
Matching user inputs with corresponding answers in a quiz.

14. Dictionaries — get() and setdefault()

Importance:
These methods enhance dictionary manipulation, aiding in handling missing keys gracefully.

When to Use:
- `get()`: When you want to retrieve a key's value but aren't sure it exists.
- `setdefault()`: When you want to set a default value if the key doesn't exist.

Example:

```
data = {"name": "Alice"}
age = data.get("age", 30)
data.setdefault("country", "USA")
```

Potential Pitfalls:
Overlooking these can lead to redundant code to check key existence.

Real-world Scenario:
Fetching configuration values with fallback defaults.

15. The __main__ Guard

Importance:
It ensures that certain code only runs when a script is executed directly, not when imported.

When to Use:
In scripts where certain code (like tests or demonstrations) should only run when executed as the main program.

Example :

```
if __name__ == "__main__":
```

```
    print("This script is being run directly!")
```

Potential Pitfalls:
Forgetting to use this guard can lead to unexpected behavior when the module is imported.

Real-world Scenario:
Creating utility scripts that can both be imported for functions or run directly for tasks.

16. Virtual Environments

Importance:
They help manage project-specific dependencies, ensuring there's no conflict with system-wide packages.

When to Use:
For every Python project, to keep dependencies isolated.

Example:

```
python -m venv my_project_env
source my_project_env/bin/activate
```

Potential Pitfalls:
Not using virtual environments can lead to package conflicts and hard-to-debug issues.

Real-world Scenario:
Maintaining separate projects with different library versions.

17. The Asterisk (*) Operator

Importance:
Beyond multiplication, the asterisk is versatile: for packing and unpacking, keyword argument unpacking, and repetition.

When to Use:
When needing to unpack collections into separate elements.

Example:

```
def func(a, b, c):
    return a + b + c
values = [1, 2, 3]
print(func(*values))
```

Potential Pitfalls:
Overuse can reduce readability, especially with multiple unpackings in a row.

Real-world Scenario:
Passing a dynamic list of values to a function expecting separate arguments.

18. The `else` Clause in Loops

Importance:
Allows you to execute code when a loop wasn't interrupted by a `break` statement.

When to Use:
When you have a block of code that should run only if the loop completed naturally.

Example:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            break
    else:
        print(n, "is a prime number.")
```

Potential Pitfalls:
It's often overlooked or misunderstood, leading to potential logic errors.

Real-world Scenario:
Searching for items in a structure and performing an action if none are found.

19. Deepcopy vs. Shallow Copy

Importance:
Understanding these is crucial when working with mutable objects and wanting to duplicate their content.

When to Use:
- Shallow Copy: When you only want a new collection with references to the same objects.
- Deepcopy: When you want a completely independent clone of the original object and all its contents.

Example:

```
import copy
original = [[1, 2, 3], [4, 5, 6]]
shallow = copy.copy(original)
deep = copy.deepcopy(original)
```

Potential Pitfalls:
Using a shallow copy when a deepcopy is needed can lead to unintended modifications of the original data.

Real-world Scenario:
Duplicating complex data structures like nested lists or dictionaries without affecting the original.

20. Python's Underscore (_) Uses

Importance:
It's versatile: denotes private variables, holds the result of the last executed statement in REPL, or acts as a throwaway variable.

When to Use:
- Naming: For "protected" variables.
- REPL: To reuse the last result.
- Looping: When you don't need the loop variable.

Example:

```
for _ in range(5):
    print("Hello, World!")
```

Potential Pitfalls:
Its varied uses can be confusing, especially for newcomers.

Real-world Scenario:
Iterating a specific number of times without needing the loop counter or marking a method as internal.