Python **Best Practices**

by **Subhayu Kumar Bala**

Disclaimer

Please Note: This presentation is not an Introduction to Python.

It assumes a basic familiarity with Python programming concepts and focuses on best practices and advanced topics. If you are new to Python, we recommend starting with introductory resources before diving into the topics covered here.

This presentation is designed for those who have some experience with Python and want to explore more advanced techniques and best practices.

Although, if you have any questions, please feel free to stop me and ask.

Follow the PEP 8 Style Guide for Python Code

- **Indentation**: Use **4**-space indentation.

```python
def bad_function():
  if True:
      print("Poor indentation!!!")

def better_function():
    if True:
        print("Indented properly")
```

- **Maximum Line Length**: Lines of code should be no longer than **79** characters. For docstrings and comments, the limit is **72** characters.

...continued

- **Imports**: Imports should be grouped and placed at the top of the file. Standard library imports should come first, followed by third-party library imports, and finally, your own module imports.

```python
# bad import style
import os
import fastapi
import sys
from pydantic import model
```

```python
# better way to import
import os
import sys

import fastapi
from pydantic import model
```

...continued

- **Whitespace in Expressions and Statements**: PEP 8 specifies guidelines for using whitespace around operators and commas, making your code more readable.

```python
# Inconsistent
x=5
y = 10

# Consistent
x = 5
y = 10
```

- **Blank Lines**: PEP 8 suggests using blank lines to separate functions, classes, and logical sections of code to improve readability.

...continued

- **Naming Conventions**: PEP 8 provides rules for naming variables, constants, functions, classes, and modules.

```python
# Bad
x = int(input("number of heads: "))

class camelman:
    l = 4

    def __init__(self, h: int):
        self.h = h

# Good
num_heads = int(input("number of heads: "))

class CamelMan:
    LEGS = 4 # tells you this is a constant

    def __init__(self, head: int):
        self.head = head
```

...continued

- **Comments and Docstrings**: Document your code with comments and docstrings.

```python
def add(a, b):
    """Adds two numbers."""
    return a + b
```

- **Other Coding Recommendations**: PEP 8 covers many other coding recommendations, such as how to handle exceptions, how to format long expressions, and how to use whitespace inside parentheses and square brackets.

Visit for reference: https://peps.python.org/pep-0008/

...continued

List Comprehensions

- Use list comprehensions for concise code.
- They are more concise and often faster than equivalent for-loops.

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

# Without list comprehension
squared = []
for num in numbers:
    squared.append(num * num)

# With list comprehension
squared = [num * num for num in numbers]
```

...continued

Setting dictionary values

- Let's say we have a dictionary named `category_items_map`.
- Now if we want to an item to the `"flowers"` category in the dictionary. *Assuming we don't know the existing contents*

```
>>> # Most of us use the following syntax
>>> category_items_map["flowers"] = category_items_map.get("flowers", []) + ["lily"]
>>>
>>> # A better way would be to use
>>> category_items_map.setdefault("flowers", []).append("lotus")
>>>
>>> category_items_map
{"flowers": ["rose", "lily", "lily"]}
```

- The above `setdefault()` method functionality can be achieved using the **`defaultdict`** class from the `collections` module. The only difference is that it's not as flexible in terms of the type of the default values.

...continued

Getting values from dictionary

- Use `.get()` method instead of third bracket notation `[]`.

```
>>> number_names = {
...     1: "one",
...     2: "two",
...     3: "three",
... }
>>>
>>> number_names[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 4
>>>
>>> number_names.get(4, "not found")
'not found'
>>>
```

...continued

Asserting earlier is better than afterwards

- Check for values before sending them into a complex/time-consuming logic.
- Here's an example of a poorly executed logic:

```
>>> def get_orders_by_ids(provider: str, ids: list[int]):
...     raw_orders: list[dict] = scrape_website_orders(provider) # takes 30-40 seconds
...     return [
...         order for order in raw_orders
...         if order.get("id") in ids
...     ]
...
>>> get_orders_by_ids("grubhub", 123)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in get_orders_by_ids
TypeError: argument of type 'int' is not iterable
```

...continued

- A better approach

```
>>> def get_orders_by_ids(provider: str, ids: list[int]):
...     assert isinstance(ids, list), "Please pass a list of ids" # <---- makes a difference
...     raw_orders: list[dict] = scrape_website_orders(provider) # takes 30-40 seconds
...     return [
...         order for order in raw_orders
...         if order.get("id") in ids
...     ]
...
>>> get_orders_by_ids("grubhub", 123)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in get_orders_by_ids
AssertionError: Please pass a list of ids
```

- Both of these throws an error. The only difference is that the second one throws the error immediately without wasting time/resource in scraping the website.

Exception Handling

- Handle exceptions with try-except blocks.
- Be specific about the exceptions.

```python
def divide(x: float, y: float):
    try:
        result = x / y
    except ZeroDivisionError:
        result = None
    return result
```

Minimalism > Complexity

- Use minimal code. Here's an example

```python
# Why use try-except?
def divide(x: float, y: float):
    if y == 0:
        return None
    return x / y
```

- Use fewer lines of code if it doesn't hamper readability.

```python
# Why use an if statement?
def divide(x: float, y: float):
    return None if y == 0 else x / y

# Want others to scratch their heads? Try this
def divide(x: float, y: float):
    return (x / y, None)[y == 0]
```

Modularize Code

- Break code into reusable functions and modules.

```python
# Non-modular code
class Rectangle:
    def __init__(self, width: int, height: int):
        ...

class Square(Rectangle):
    def __init__(self, side: int):
        super().__init__(side, side)

r = Rectangle(10, 20)
s = Square(10)

# Modular code
from geometry import Rectangle, Square

r = Rectangle(10, 20)
s = Square(10)
```

Linting and Code Analysis

- Use static code analysis tools.

```
$ pylint my_module.py
```

- For linting directly from editor, use
  - black *# personal favorite*
  - Pylint
  - flake8
  - mypy

Check the version

- Check the python version when requirements are tight

```python
import sys

if not sys.version_info >= (3, 10):
    print("Need python 3.10 or higher")
    exit() # this is not required, but should be used when the codebase needs the mentioned version
```

Use virtual environments

- Use virtual environments to keep your project isolated from other projects.

```
$ python -m venv myenv
$ source myenv/bin/activate
```

Write Object-Oriented Code

- Not using OOP can lead to unstructured and hard-to-maintain code as your project grows. You may end up with functions and variables scattered across your codebase, making it challenging to manage. Here's an example without using OOP:

```python
def create_car(make: str, model: str):
    return {"make": make, "model": model, "speed": 0}

def accelerate(car, mph: float):
    car["speed"] += mph
    print(f"Accelerating {car['make']} {car['model']} to {car['speed']} mph.")

# Usage
my_car = create_car("Toyota", "Camry")
accelerate(my_car, 30)
```

...continued

- When working on larger projects, use object-oriented programming (OOP) principles to organize your code into classes and objects. Here's a better way to create a car using OOP:

```python
class Car:
    def __init__(self, make: str, model: str):
        self.make = make
        self.model = model
        self.speed = 0

    def accelerate(self, mph: float):
        self.speed += mph
        print(f"Accelerating {self.make} {self.model} to {self.speed} mph.")

# Usage
my_car = Car("Toyota", "Camry")
my_car.accelerate(30)
```

Use __dunder__ methods

```python
import time

class TimerContext:
    elapsed_time = 0.0

    def __str__(self) -> str:
        return f"Start: {self.start_time:.3f} | End: {self.end_time:.3f} | Elapsed: {self.elapsed_time:.3f}"

    def __enter__(self):
        self.start_time = time.perf_counter()
        return self  # Return an object that represents the context

    def __exit__(self, exc_type, exc_value, traceback):
        self.end_time = time.perf_counter()
        self.elapsed_time = self.end_time - self.start_time
        print(self) # This internally calls the `__str__` method

with TimerContext() as timer: # Call the __enter__ method
    for _ in range(1000000):
        pass  # Simulate some work
# After the block, the __exit__ method is automatically
print(timer.elapsed_time) # The context manager stays persistent
```

Use pathlib.Path for storing paths

- When dealing with file paths, use Python's pathlib module for better cross-platform compatibility and path manipulation.
- Not using pathlib.Path can lead to code that may break on different operating systems due to inconsistent path separators, making your code less portable.

```python
# Bad (using string paths)
import os

data_dir = "data"
file_path = data_dir + "/file.txt" # this won't work on windows
if os.path.exists(file_path):
    file_name = file_path.split("/")[-1] # does this look good?

# Good
from pathlib import Path

data_dir = Path("data")
file_path = data_dir / "file.txt" # this will work everywhere
if file_path.exists():
    file_name = file_path.name # isn't this more elegant?
```

Use relative paths instead of absolute paths

- Whenever possible, use relative file paths instead of absolute file paths to make your code more portable and independent of the file system structure.

```python
# Good (relative path)
aliens_dir = Path("aliens")
proof_path = aliens_dir / "proof.mp4"

# Bad (absolute path)
proof_path = Path("/home/subhayu/area51/aliens/proof.mp4")
```

- Using absolute paths makes your code less portable. If the project is moved to a different directory or another system, the absolute paths may not point to the correct locations, causing errors and making the code less flexible.

Use f-strings

When formatting strings in Python, use f-strings (formatted string literals) for improved readability and simplicity.

```
name, age = "Steve Rogers", 100

message = "Hello, my name is " + name + " and I am " + str(age) + " years old." # Bad
message = f"Hello, my name is {name} and I am {age} years old." # Good
```

```
>>> a, b, c = 1, "two", 3.0
>>> f"{a = }, {b = }, {c = }" # Using equals gives the values
"a = 1, b = 'two', c = 3.0"
>>> f"{25/200:.2%}" # Converting to percentage
'12.50%'
>>> f"${1234567890:,}" # Thousands separator
'$1,234,567,890'
>>> f"Decimal: {16}, Binary: {16:b}, Octal: {16:o}, Hex: {16:x}" # Convert to other bases
'Decimal: 16, Binary: 10000, Octal: 20, Hex: 10'
```

Use a config file for storing project settings

- Store project-specific settings, such as database URLs, API keys, and other configuration parameters, in a separate configuration file.
- There are lots of different ways to do this. You can use **.env**, **.yaml**, **.ini**, etc...
- Personally, I like to use .env files. Here's an example:

```
# .env
DATABASE_URL='postgres://tony:jarvis@localhost:5432/avengers'
HTTP_PORT=5000
```

```python
# config.py
from dotenv import load_dotenv

load_dotenv()

DATABASE_URL = os.getenv("DATABASE_URL")
HTTP_PORT = int(os.getenv("HTTP_PORT", 8080))
```

Always create an uniform interface

- Here, **interface** can be a class, function, or module that does the same task.
- The below example shows how easy it is to query a database using a class.

```python
class ComicDB:
    def __init__(self, url: str):
        ...
    def query(self, collection: str, query: dict = {}):
        ...
    def insert(self, collection: str, data: dict):
        ...

DB = ComicDB("mongodb://localhost:27017/")

mcu_heroes = DB.query("heroes", {"universe": "Marvel"})
mcu_villains = DB.query("villains", {"universe": "Marvel"})

dc_heroes = DB.query("heroes", {"universe": "DC"})
dc_villains = DB.query("villains", {"universe": "DC"})
```

Tips to improve your Python Game

- Unpacking iterables

```
>>> a, b, c = [1, 2, 3]
>>> a, c
(1, 3)
>>>
>>> a, b, *c, d = [1, 2, 3, 4, 5]
>>> a, b, c, d
(1, 2, [3, 4], 5)
>>>
>>> d = {1: "one", 2: "two", 3: "three"}
>>> keys, vals = zip(*d.items())
>>> keys, vals
((1, 2, 3), ('one', 'two', 'three'))
```

...continued

- Check if list is not empty

```
# Non-Pythonic way
if len(my_list) > 0:
    ...

# Pythonic way
if my_list:
    ...
```

- Get keys from a dictionary as a list

```
>>> list(my_dict.keys()) # Tedious way
['a', 'b', 'c']
>>>
>>> list(my_dict) # Better way
['a', 'b', 'c']
```

...continued

- Use `if __name__ == "__main__":`
  - When you have a python file that can be called as a module while also being executed as a script, you should use `if __name__ == "__main__":`.

```
if __name__ == "__main__":
    main()
```

- Use ternary operators

```
a = 1

# Ehhh! bad
if a > 0:
    print(a)
else:
    print(0)


print((0, a)[a > 0]) # Good, but tough to interpret

print(a if a > 0 else 0) # Better
```

...continued

- Use `try` / `except` / `finally`

```python
try:
    ...
except:
    ...
finally: # This always executes
    ...
```

- Use `namedtuples` for simple models

```python
>>> from collections import namedtuple
>>>
>>> Point = namedtuple("Point", ["x", "y"])
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
>>> list(p)
[1, 2]
```

…continued

- Use `in` / `not in`

```
>>> 1 in [1, 2, 3]
True
>>> 1 not in [1, 2, 3]
False
```

- Use `is` / `is not`

```
>>> x = [1, 2, 3] # x is a list
>>> y = x         # y is a reference to x
>>> z = [1, 2, 3] # z is a new list
>>>
>>> x is y
True
>>> x is not z
False
```

...continued

- Use `enumerate`

```
>>> for i, item in enumerate([2, 1, 0]):
...     print(i, item)
...
0 2
1 1
2 0
```

- Use walrus `:=` operator

```
>>> print(a := 1) # Can be used with any function/method call
1
>>> a
1
```

***Note**: Walrus `:=` operator can be used from **Python 3.8** onwards.*

...continued

- Use `isinstance`

```
>>> isinstance(1, int) # check if 1 is an int
True
>>> isinstance(1.0, (int, float)) # check if 1 is an int or float
True
>>> isinstance(1, str) # check if 1 is a string
False
```

- Use `all` / `any`

```
>>> all([1, 2, 3]) # check if all elements in an iterable are true
True
>>> any([1, 2, 3]) # check if any element in an iterable is true
True
>>> any([0, 2, 3]) # check if any element in an iterable is true
True
>>> all([0, 2, 3]) # check if all elements in an iterable are true
False
```

...continued

- Use `zip` and `zip_longest` to iterate over iterables parallelly

```
>>> list(zip([1, 2, 3], [4, 5, 6, 7]))
[(1, 4), (2, 5), (3, 6)]
>>>
>>> from itertools import zip_longest
>>>
>>> list(zip_longest([1, 2, 3], [4, 5, 6, 7]), fillvalue=0)
[(1, 4), (2, 5), (3, 6), (0, 7)]
```

- Use `chain` to concatenate iterables

```
>>> from itertools import chain
>>>
>>> list(chain([1, 2, 3], [4, 5, 6]))
[1, 2, 3, 4, 5, 6]
```

...continued

- Convert `list` of `tuple`s to `dict`ionary

```
>>> dict([("a", 1), ("b", 2)])
{'a': 1, 'b': 2}
```

- Use `dict` comprehension

```
>>> {k: v for k, v in [("a", 1), ("b", 2)]}
{'a': 1, 'b': 2}
```

- Merging Dictionaries with the `**` operator

```
>>> d1 = {"a": 1, "b": 5}
>>> d2 = {"b": 2, "c": 3, "d": 4}
>>> {**d1, **d2}
{'a': 1, 'b': 2, 'c': 3, 'd': 4} # merged and overwritten duplicates
```

...continued

- Use generator expressions

```
>>> # generators are lazy, i.e. they are evaluated only when needed
>>> (a for a in range(10) if a % 2 == 0)
<generator object <genexpr> at 0x7ff3c7a2c7c0>
```

- Use `Counter` to count items in an iterable

```
>>> from collections import Counter
>>>
>>> Counter([1, 1, 1, 2, 2, 3]) # afterall, this is an instance of dict
Counter({1: 3, 2: 2, 3: 1})
```

- Want a Queue, use `deque`

```
>>> from collections import deque
>>>
>>> queue = deque([1, 2, 3], maxlen=3)
>>> queue.append(4)
>>> list(queue)
[2, 3, 4]
```

...continued

- Use `lambda` to create anonymous functions

```
>>> sum = lambda a, b: a + b
>>> sum(1, 2)
3
```

- Use `map` and `filter` to apply a function to an iterable

```
>>> list(map(lambda x: x**2, [1, 2, 3, 4, 5]))
[1, 4, 9, 16, 25]
>>> list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4, 5]))
[2, 4]
```

- Use `reduce` to apply a function to an iterable

```
>>> from functools import reduce
>>>
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5]) # Sum of a list
15
```

...continued

- Use `traceback` module when debugging

```
>>> import traceback as tb
>>>
>>> try:
...     0 / 0
... except ZeroDivisionError as e:
...     print(e) # Gives only the error message. Not useful for debugging
...     print()
...     tb.print_exception(e, limit=3) # Prints the traceback upto 3 levels deep. Useful for debugging
...
division by zero

Traceback (most recent call last):
File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

  - `traceback` has many other useful functions for handling errors.

...continued

- Use `math.isclose` to compare floating-point numbers

```python
x = 0.1 + 0.2
print(x == 0.3)  # False 😱

# need a small tolerance
import math
print(math.isclose(x, 0.3))  # True
```

Tips to increase efficiency

Efficiency tips in Python focus on writing code that runs faster and uses fewer resources. Here are some efficiency tips to consider:

1. **Use Built-in Functions:** Python provides many built-in functions for common operations like sorting (`sorted()`), finding the maximum or minimum value (`max()` , `min()`), and summing elements (`sum()`). Using these functions is often more efficient than writing custom code.
2. **Avoid Global Imports:** Import only the functions or classes you need from libraries instead of using wildcard imports (`from module import *`). This reduces namespace pollution and can slightly improve import times.

...continued

3. **Avoid Unnecessary Loops:** Minimize the use of nested loops and try to reduce the complexity of your algorithms. Sometimes, a more efficient data structure can help.
4. **Generator Expressions:** Use generator expressions when you need to iterate over a large dataset or generate values lazily. Generators save memory.
5. **Set Operations:** When checking for membership or finding unique elements, use sets (`set`) or dictionaries (`dict`) instead of lists (`list`) because sets have faster lookup times.
6. **Avoid Repeated Appending:** Instead of repeatedly appending elements to a list in a loop, consider using a list comprehension or `join()` for strings. Appending in a loop can be slow for large lists.

...continued

7. **Use `collections`:** Python's `collections` module provides specialized data structures like `defaultdict`, `Counter`, and `deque` that can improve performance in certain scenarios.
8. **Avoid Global Variables:** Minimize the use of global variables as they can slow down your code's execution. Use local variables or function arguments whenever possible.
9. **Profile Your Code:** Use profiling tools like `cProfile` to identify performance bottlenecks in your code. Optimize only the parts of your code that need it.
10. **Vectorized Operations with NumPy:** If you are working with numerical data and need to perform operations on arrays, consider using the NumPy library. NumPy provides highly optimized functions for array operations.

...continued

11. **Concurrency and Parallelism:** When dealing with tasks that can be performed independently, consider using Python's `concurrent.futures` or `multiprocessing` libraries to take advantage of multi-core processors.
12. **Cython or Cythonize:** If you need to optimize performance-critical code further, consider using Cython, which allows you to write Python-like code that can be compiled to C for improved performance.
13. **Memory Management:** Be mindful of memory usage, especially when dealing with large datasets. Use generators to avoid loading everything into memory at once, and consider memory profiling tools like `memory_profiler`.
14. **Algorithm Choice:** The choice of algorithm can have a significant impact on efficiency. Be aware of different algorithms and their time complexities when solving problems.

Conclusion

As we conclude our journey through Python best practices, remember this: Python isn't just a programming language; it's a canvas for *digital masterpieces*. It empowers architects, engineers, and dreamers. It fuels innovation and creativity—from small scripts to digital marvels shaping our future. Python is your **gateway to extraordinary creations**. So, as you step into the coding world, armed with today's knowledge, keep pushing boundaries. The world eagerly awaits your Python-powered brilliance. **Happy coding**, and may your Pythonic adventures be *awe-inspiring*.

Python's Symphony

“

Python code, elegant and neat,
A language where brilliance takes its seat.
With simplicity, it does excel,
In the coder's heart, it casts a spell.

From web to AI, it conquers all,
In Python's world, we stand tall.
So code away, with joy and might,
In Python's realm, where dreams take flight.

”

Thank You