PRACTICAL PYTHON SERIES

TUPLES, NAMED-TUPLES, CLASSES & DATA-CLASSES IN PYTHON

BIMALKA THALAGALA
@bimalka98

If you have a background in languages like C or C++, you've likely searched for "structs in Python" at some point,

Only to find out that Python does not include a built-in data structure known as "struct."

Nonetheless, Python offers several data structures that are quite similar to structs, allowing you to group objects effectively.

They include *tuples, namedtuples, classes,* and *dataclasses.*

By comprehending the advantages and disadvantages of each option, you can choose the most appropriate data structure for your specific requirements.

This knowledge helps you achieve a balance between simplicity, readability, and functionality.

This slide set is organized as below:

TUPLE

```python
# Tuple example
person_tuple = ("Alice", 30, "alice@example.com")

print(person_tuple)
# Output: ('Alice', 30, 'alice@example.com')

print(person_tuple[0])  # Output: Alice
print(person_tuple[1])  # Output: 30
print(person_tuple[2])  # Output: alice@example.com
```

NAMEDTUPLE

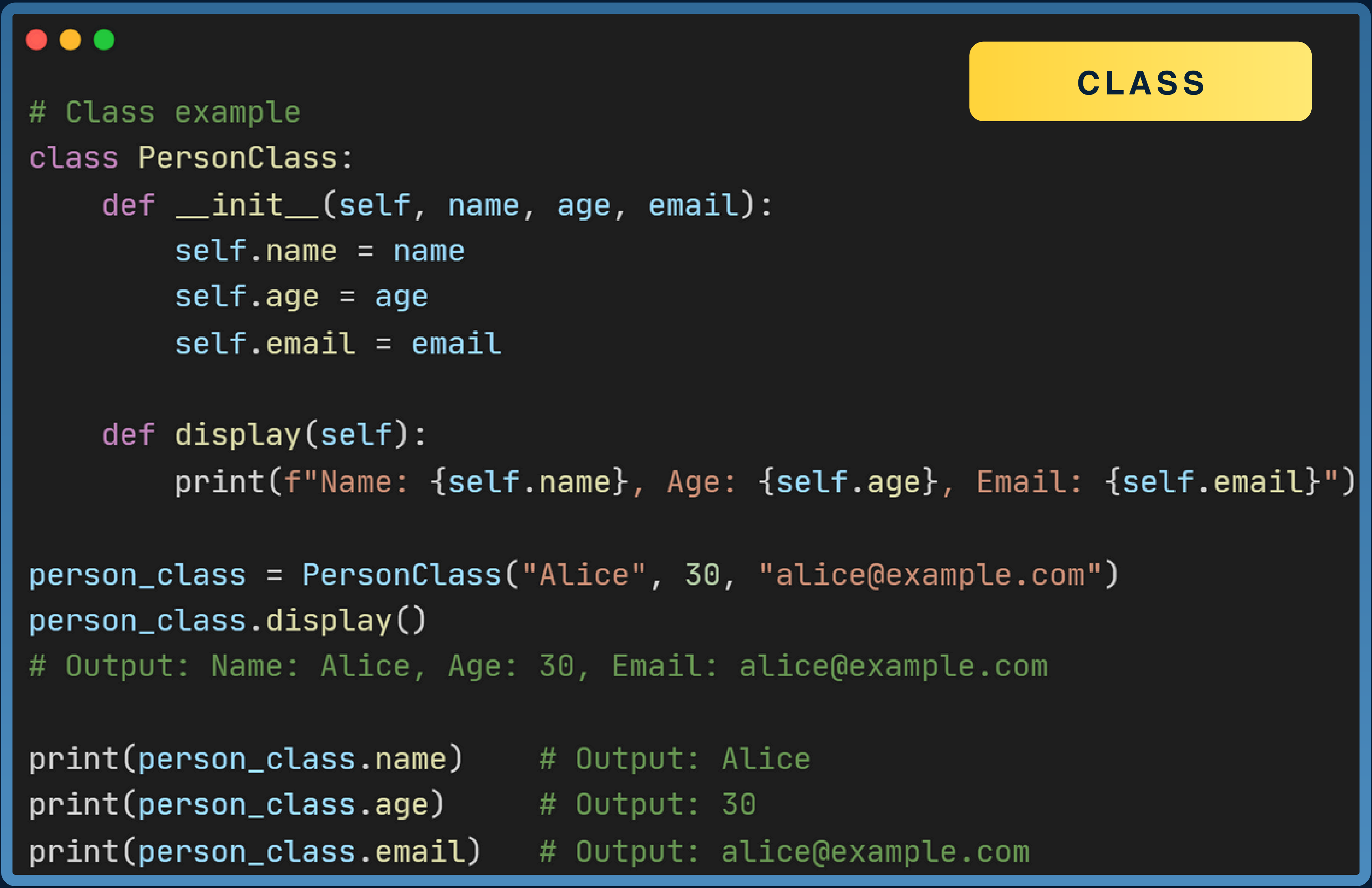
```python
# NamedTuple example
from collections import namedtuple

PersonNamedTuple = namedtuple('Person',
                    ['name', 'age', 'email'])
person_namedtuple = PersonNamedTuple(
                    "Alice", 30, "alice@example.com")
print(person_namedtuple)
# Output: Person(name='Alice', age=30, email='alice@example.com')

print(person_namedtuple.name)   # Output: Alice
print(person_namedtuple.age)    # Output: 30
print(person_namedtuple.email)  # Output: alice@example.com
```

TUPLES

- The simplest, most lightweight structure
- Immutable (cannot be changed after creation)
- Accessed by numeric indexes
- Best for small, unchanging collections of data
- Lowest memory overhead

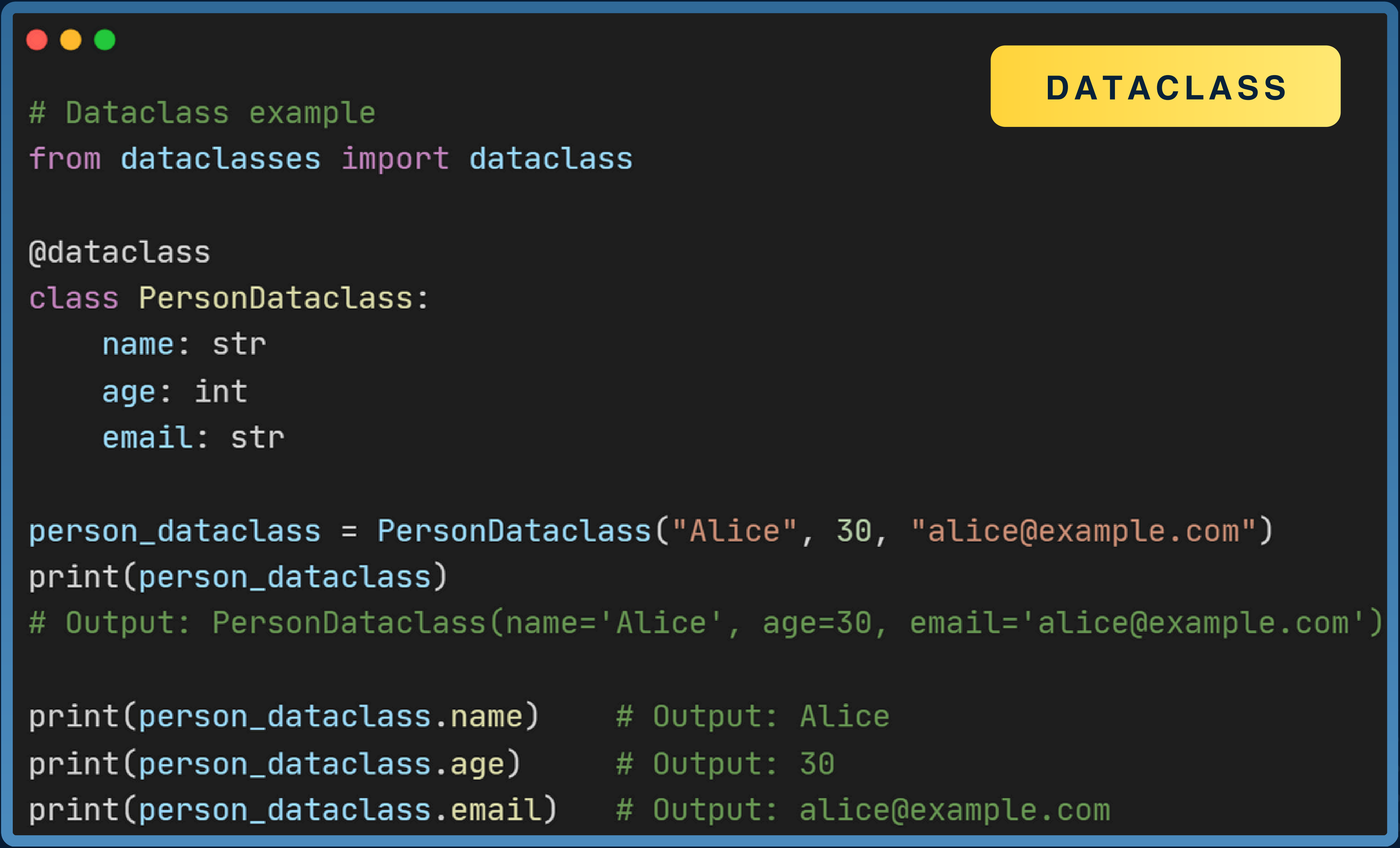
NAMED-TUPLES

- Provides named attributes instead of numeric indexes
- Still immutable like regular tuples
- More readable and self-documenting
- Slightly more memory overhead than tuples
- Easy to convert to dictionaries
- Good for intermediate scenarios where you want more clarity



CLASS

```python
# Class example
class PersonClass:
    def __init__(self, name, age, email):
        self.name = name
        self.age = age
        self.email = email

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}, Email: {self.email}")

person_class = PersonClass("Alice", 30, "alice@example.com")
person_class.display()
# Output: Name: Alice, Age: 30, Email: alice@example.com

print(person_class.name)    # Output: Alice
print(person_class.age)     # Output: 30
print(person_class.email)   # Output: alice@example.com
```



DATACLASS

```python
# Dataclass example
from dataclasses import dataclass

@dataclass
class PersonDataclass:
    name: str
    age: int
    email: str

person_dataclass = PersonDataclass("Alice", 30, "alice@example.com")
print(person_dataclass)
# Output: PersonDataclass(name='Alice', age=30, email='alice@example.com')

print(person_dataclass.name)    # Output: Alice
print(person_dataclass.age)     # Output: 30
print(person_dataclass.email)   # Output: alice@example.com
```

CLASSES

- Most flexible and powerful
- Mutable attributes
- Can add methods and complex behavior
- Supports inheritance and polymorphism
- Highest memory and computational overhead
- Best for complex objects with multiple behaviors and state

DATA-CLASSES

- Combines the simplicity of tuples with the readability of named tuples and the flexibility of classes
- Automatically generates special methods like __init__, __repr__, and __eq__
- Mutable by default, but can be made immutable with *frozen=True*
- Supports type annotations for fields
- Allows default values and field customization
- Suitable for data-centric classes with minimal boilerplate code

CHOOSING THE RIGHT STRUCTURE

- Have only less than 4 attributes?
    - → Consider Tuple
- Need attribute names?
    - → Consider NamedTuple
- Need minimal boilerplate with type annotations?
    - → Consider Dataclass
- Need methods, complex logic & scalability?
    - → Use Class

PERFORMANCE CONSIDERATION

- Tuple:
  - Fastest, lowest memory
- NamedTuple:
  - Slightly more overhead, but
  - Very readable
- Dataclass:
  - More overhead than NamedTuple, but
  - Very flexible and readable
- Class:
  - Most flexible, highest overhead
  - Scalable as the project grows

PROGRESSION PATH

Start with tuples for the simplest data grouping

↓

Move to NamedTuples for improved readability

↓

Use Dataclasses for structured data with minimal boilerplate

↓

Use Classes for complex, behavior-rich objects

Phew. That’s it!

By understanding these differences, you can choose the most appropriate data structure for your specific use case, balancing simplicity, readability, and functionality.

Happy & effective coding! 😉

BIMALKA THALAGALA
@bimalka98

If you found this content helpful,
please consider reacting or commenting, and
reposting it to help others too!

LIKE SAVE REPOST