Python Advanced Concepts and Performance Optimization - Day 3

Welcome to the third day of our advanced Python training. Today we'll dive deep into gradual typing, the buffer protocol, and PySpark for distributed computing.

Day 3 Agenda

Gradual Typing

Optional static typing in Python, mypy, type hints, and advanced typing techniques

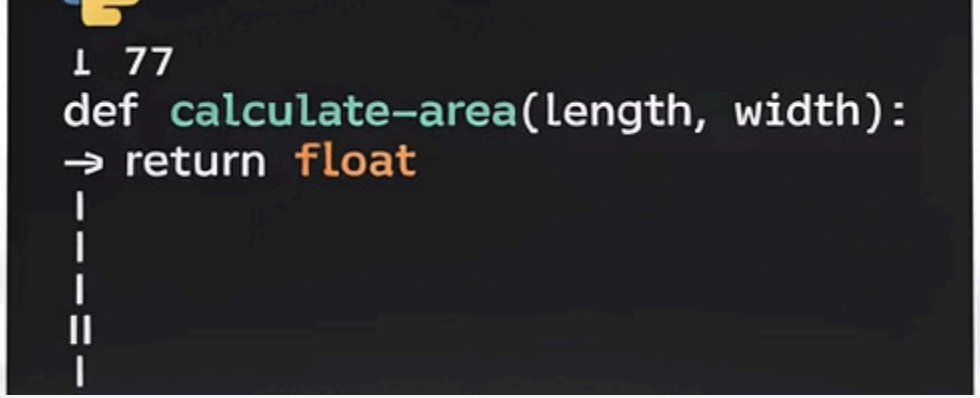
Buffer Protocol

Memory views, efficient data handling, and implementing custom buffer protocols

PySpark

Distributed computing, RDDs, DataFrames, and practical applications

By the end of today's session, you'll have gained practical knowledge of Python's type system, efficient memory management, and distributed computing techniques that will enhance both your code quality and performance.



Gradual Typing in Python

Python's journey from a dynamically typed language to supporting optional static typing

What is Gradual Typing?

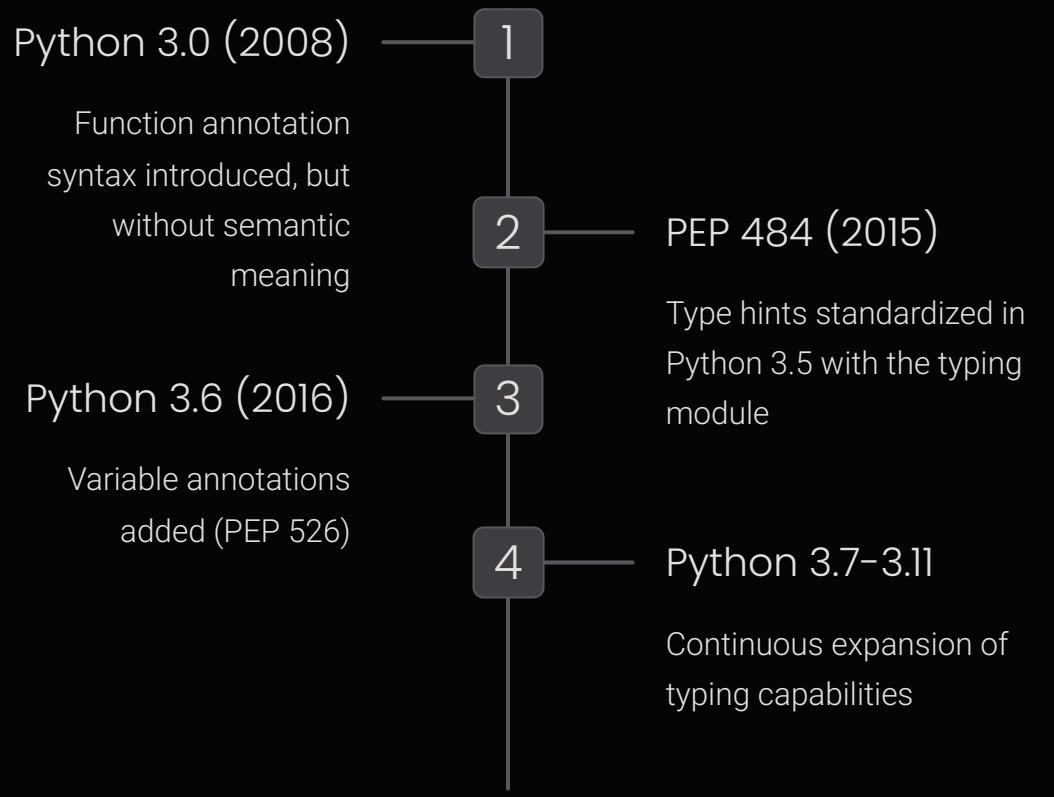
Gradual typing is a type system that allows some parts of a program to be dynamically typed and other parts to be statically typed. It bridges the gap between:

- Dynamic typing: Types checked at runtime
- Static typing: Types checked at compile time

Python implements gradual typing through type hints, which are optional annotations that specify the types of variables, function parameters, and return values.

These annotations do not affect runtime behavior but can be used by external tools to detect type errors before execution.

The journey to gradual typing in Python:



Benefits of Type Annotations

Better IDE Support

Enhanced code completion, documentation, and refactoring capabilities in editors like PyCharm, VS Code, etc.

Static Error Detection

Catch type-related errors before runtime using tools like mypy, pyright, or pyre

Improved Documentation

Self-documenting code that explicitly shows expected types for functions and variables

Code Maintainability

Easier to understand and maintain large codebases, especially in team environments

Type hints provide these benefits while preserving Python's flexibility—you can add types incrementally to existing code and choose which parts of your codebase benefit most from static typing.

Basic Type Annotations

Python's type hints allow you to annotate variables, function parameters, and return values with their expected types.

```
# Variable annotations
name: str = "John"
age: int = 30
active: bool = True

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

# Collection types
from typing import List, Dict, Tuple, Set

numbers: List[int] = [1, 2, 3]
user: Dict[str, str] = {"name": "John", "email": "j@example.com"}
point: Tuple[int, int] = (10, 20)
unique_tags: Set[str] = {"python", "typing"}
```

Modern syntax with Python 3.9+:

```
# Python 3.9+ allows using built-in collections as generic types
numbers: list[int] = [1, 2, 3]
user: dict[str, str] = {"name": "John", "email": "j@example.com"}
point: tuple[int, int] = (10, 20)
unique_tags: set[str] = {"python", "typing"}

# Optional types for nullable values
from typing import Optional

def get_user(user_id: int) -> Optional[dict[str, str]]:
    # Might return a user dict or None if not found
    return {"name": "John"} if user_id > 0 else None
```

Remember: Type annotations are optional and do not affect the runtime behavior of your code. They're purely for static analysis tools and documentation.

The mypy Type Checker

mypy is the original static type checker for Python, developed by Jukka Lehtosalo and now maintained by the Python core team. It analyzes your Python code with type annotations and reports potential type errors without running the code.

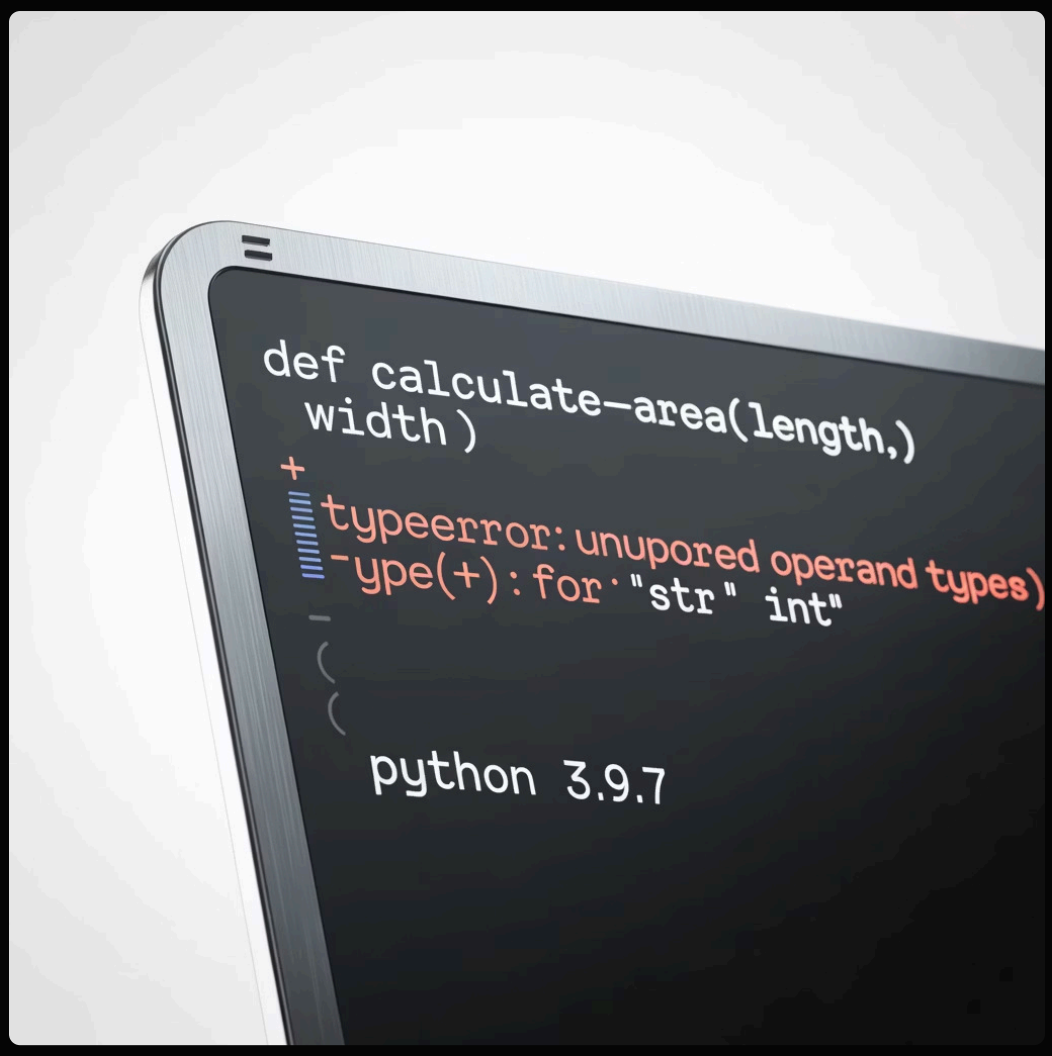
- Reads PEP 484 type annotations and performs static analysis
- Can be integrated into CI/CD pipelines to catch type errors early
- Configurable strictness levels to suit different project needs

Installation:

```
pip install mypy
```

Basic usage:

```
mypy your_script.py
```



mypy configuration options in mypy.ini or setup.cfg:

```
[mypy]
python_version = 3.10
warn_return_any = True
warn_unused_configs = True
disallow_untyped_defs = False
disallow_incomplete_defs = False

[mypy.plugins.numpy.ndarray]
plugin_is_available = True
```

Incremental adoption is key: You can start adding type hints to critical parts of your codebase first and gradually expand coverage as you become more comfortable with the type system.

Hands-on: Adding Type Hints and Checking Code with mypy

```
# Before: Untyped code
def calculate_total(prices, quantities, discount=0):
    total = sum(p * q for p, q in zip(prices, quantities))
    return total * (1 - discount)

# After: With type hints
from typing import List, Union, Optional

def calculate_total(
    prices: List[float],
    quantities: List[int],
    discount: Optional[float] = 0
) -> float:
    if len(prices) != len(quantities):
        raise ValueError("Price and quantity lists must be the same length")
    total = sum(p * q for p, q in zip(prices, quantities))
    return total * (1 - discount)
```

Now run mypy to check for type errors:

```
$ mypy example.py
```

With proper type hints, mypy can catch potential issues like passing a string where a number is expected, or using incompatible types in operations.

Working with Complex Types

Union Types

Represent values that could be one of several types:

```
from typing import Union

def process_id(id_value: Union[int, str]) -> str:
    return str(id_value)

# Python 3.10+ syntax
def process_id(id_value: int | str) -> str:
    return str(id_value)
```

Type Aliases

Create readable names for complex types:

```
from typing import Dict, List, Union

# Type alias
JSONValue = Union[str, int, float, bool, None,
                  Dict[str, 'JSONValue'],
                  List['JSONValue']]

def parse_json(data: str) -> JSONValue:
    import json
    return json.loads(data)
```

Callable Types

Represent functions and their signatures:

```
from typing import Callable

# A function that takes a function and applies it
def apply_twice(func: Callable[[int], int],
                value: int) -> int:
    return func(func(value))

# Usage
def double(x: int) -> int:
    return x * 2

result = apply_twice(double, 3)  # 12
```

Literal Types

Represent specific literal values:

```
from typing import Literal

# Only accept specific string values
def align_text(
 text: str,
 alignment: Literal["left", "center", "right"]
) -> str:
 # Align text based on the alignment parameter
 if alignment == "left":
 return text.ljust(50)
 elif alignment == "center":
 return text.center(50)
 else: # right
 return text.rjust(50)
```

Advanced Typing Features: @override

The `@override` decorator (introduced in Python 3.12) explicitly indicates that a method is intended to override a method in a superclass. This helps catch bugs where:

- You misspelled the method name
- The method signature doesn't match the parent class
- The parent class method was removed or renamed

Without this decorator, these issues might go unnoticed until runtime.

```
from typing import override

class Base:
    def process(self, value: str) -> int:
        return len(value)

class Derived(Base):
    @override
    def process(self, value: str) -> int:
        # This will be checked by type checkers
        # to ensure it actually overrides a method
        # in the parent class
        return len(value) * 2

    @override
    def proces(self, value: str) -> int:  # Error!
        # Type checker will catch this misspelling
        return len(value) * 2
```

This decorator is especially useful in large codebases with deep inheritance hierarchies, helping to maintain consistency as the codebase evolves.

Advanced Typing Features: @final

The `@final` decorator allows you to indicate that a class or method shouldn't be subclassed or overridden. This is useful for:

- Enforcing API design decisions
- Preventing accidental subclassing when it would break assumptions
- Making it clear which parts of your API are stable and which might change

Type checkers will report an error if someone tries to subclass a class or override a method marked as `@final`.

```
from typing import final

@final
class SecurityManager:
    """This class handles critical security operations
    and shouldn't be subclassed to prevent security bypasses."""

    def authenticate(self, username: str, password: str) -> bool:
        # Authentication logic
        return True

    @final
    def validate_token(self, token: str) -> bool:
        # Token validation logic that shouldn't be overridden
        return len(token) > 10

# This would trigger a type error:
class EnhancedSecurityManager(SecurityManager):  # Error!
    pass
```

Note: `@final` is enforced by type checkers, not at runtime. It serves as documentation and helps catch design violations during development.

Advanced Typing Features: @dataclass

The `@dataclass` decorator automatically generates special methods like `__init__`, `__repr__`, and `__eq__` for your class, based on the class variables you define.

This is particularly well-integrated with type hints, as the type annotations for class variables become both documentation and functional parts of the generated code.

```
from dataclasses import dataclass
from typing import List, Optional

@dataclass
class Person:
    name: str
    age: int
    email: Optional[str] = None
    skills: List[str] = None

    def __post_init__(self):
        if self.skills is None:
            self.skills = []

        if self.age < 0:
            raise ValueError("Age cannot be negative")

# Usage:
john = Person("John Doe", 30, "john@example.com")
jane = Person("Jane Smith", 25)

# Automatic equality comparison:
john == Person("John Doe", 30, "john@example.com")  # True
```

Dataclasses are perfect for domain models, configuration objects, and other data container classes where you need consistent behavior and less boilerplate code.

Introduction to Generics in Python

Generics allow you to write code that works with multiple types while still maintaining type safety. This is particularly useful for containers, collections, and algorithms that should work with any type of data.

TypeVar: The Foundation of Generics

`TypeVar` is the building block for generic types in Python. It represents a type variable that can be different for each use of a function or class.

```
from typing import TypeVar, List

T = TypeVar('T')  # Define a type variable

def first(items: List[T]) -> T:
    """Return the first item from a list."""
    if not items:
        raise ValueError("Empty list")
    return items[0]

# Usage with different types:
first([1, 2, 3])  # Returns int
first(["a", "b", "c"])  # Returns str
first([True, False])  # Returns bool
```

The type checker understands that the return type depends on the input type, preserving type information throughout your code.

You can constrain `TypeVar` to only accept certain types:

```
from typing import TypeVar, List, Union

# Type variable constrained to numeric types
Number = TypeVar('Number', int, float, complex)

def add_all(numbers: List[Number]) -> Number:
    """Sum all numbers in the list."""
    result: Number = numbers[0]
    for n in numbers[1:]:
        result += n
    return result

# Valid usage:
add_all([1, 2, 3])  # OK, returns int
add_all([1.0, 2.5, 3.7])  # OK, returns float

# Invalid usage (would be caught by type checker):
add_all(["a", "b", "c"])  # Error: strings aren't allowed
```

This helps catch type errors while still allowing flexibility across compatible types.

Generic Classes with the Generic Base Class

The `Generic` base class allows you to create your own generic classes that work with various types while maintaining type safety.

```
from typing import TypeVar, Generic, List, Optional

T = TypeVar('T')

class Stack(Generic[T]):
    def __init__(self) -> None:
        self.items: List[T] = []

    def push(self, item: T) -> None:
        self.items.append(item)

    def pop(self) -> Optional[T]:
        if not self.items:
            return None
        return self.items.pop()

    def peek(self) -> Optional[T]:
        if not self.items:
            return None
        return self.items[-1]
```

Usage of the generic Stack class:

```
# Integer stack
int_stack = Stack[int]()
int_stack.push(1)
int_stack.push(2)
x = int_stack.pop()  # Type is int

# String stack
str_stack = Stack[str]()
str_stack.push("hello")
str_stack.push("world")
s = str_stack.pop()  # Type is str

# The type checker will catch this error:
int_stack.push("oops")  # Error: expected int, got str
```

Generic classes provide flexibility while maintaining strong typing. They're useful for containers, data structures, and algorithms that should work with multiple types.

Many standard library collections in Python's typing module are implemented as generic classes, such as `List[T]`, `Dict[K, V]`, and `Set[T]`.

Protocols and Structural Typing

Protocols (introduced in PEP 544) enable structural typing in Python, which is based on what an object can do rather than what it inherits from.

This is similar to interfaces in languages like Go or TypeScript, and enables more flexible typing than traditional inheritance-based approaches.

```
from typing import Protocol, List, Iterator

class Sized(Protocol):
    def __len__(self) -> int: ...

class Iterable(Protocol[T]):
    def __iter__(self) -> Iterator[T]: ...

def get_first_and_size(obj: Iterable[T]) -> tuple[T, int]:
    iterator = iter(obj)
    first_item = next(iterator)

    if isinstance(obj, Sized):
        size = len(obj)
    else:
        # Count remaining items
        size = 1 + sum(1 for _ in iterator)

    return first_item, size
```

The power of protocols is that classes don't need to explicitly inherit or implement them—they just need to have the required methods with compatible signatures.

```
# These all work with get_first_and_size even though
# they don't explicitly inherit from Iterable or Sized

# Lists
result1 = get_first_and_size([1, 2, 3])  # (1, 3)

# Strings
result2 = get_first_and_size("hello")  # ('h', 5)

# Custom class
class MyCollection:
    def __init__(self, items):
        self._items = list(items)

    def __iter__(self):
        return iter(self._items)

    def __len__(self):
        return len(self._items)

result3 = get_first_and_size(MyCollection([4, 5, 6]))  # (4, 3)
```

Protocols enable "duck typing" at the static analysis level, making Python's type system more expressive while maintaining its flexibility.

Abstract Base Classes (ABCs) and Interfaces

Abstract Base Classes provide a way to define interfaces that derived classes must implement. Unlike protocols, ABCs use inheritance to enforce interface compliance.

```
from abc import ABC, abstractmethod
from typing import List, Optional

class Animal(ABC):
    @abstractmethod
    def make_sound(self) -> str:
        """Return the sound this animal makes."""
        pass

    @abstractmethod
    def move(self, distance: float) -> None:
        """Move the animal by the given distance."""
        pass

    def sleep(self, hours: int = 8) -> None:
        """Default implementation that can be overridden."""
        print(f"Sleeping for {hours} hours")
```

Implementing an ABC:

```
class Dog(Animal):
    def make_sound(self) -> str:
        return "Woof!"

    def move(self, distance: float) -> None:
        print(f"Running {distance} meters")

# This would fail:
# animal = Animal()  # TypeError: Can't instantiate abstract class

# This works:
dog = Dog()
dog.make_sound()  # "Woof!"
dog.move(10)      # "Running 10 meters"
dog.sleep()       # "Sleeping for 8 hours"

# This would fail at type-checking time:
class Cat(Animal):
    # Missing implementation of move()
    def make_sound(self) -> str:
        return "Meow!"
```

ABCs are useful when you want to ensure certain methods are implemented by subclasses. They combine runtime checking with static type checking.

ABCs vs. Protocols: When to Use Each

Abstract Base Classes

- **Inheritance-based**: Classes must explicitly inherit from the ABC
- **Runtime checking**: Prevents instantiation of incomplete implementations
- **Default implementations**: Can provide base functionality for subclasses
- **Registration**: External classes can be registered with ABC
- **isinstance/issubclass**: Support for runtime type checking

Best for: Framework design, enforcing contracts, when you control all implementation classes

Protocols

- **Structural typing**: Classes just need compatible methods
- **Static checking only**: No runtime enforcement
- **No inheritance required**: Works with existing classes
- **More flexible**: Supports duck typing philosophy
- **Type checking only**: Doesn't affect runtime behavior

Best for: Working with existing code, libraries you don't control, maintaining flexibility, or when explicit inheritance would create coupling

In practice, ABCs are often used for framework code where you want to enforce contracts, while Protocols are ideal for more flexible interfaces, especially when working with third-party code or following duck-typing principles.

Hands-on: Implementing a Generic Data Structure

Let's implement a priority queue as a generic data structure:

```
from typing import TypeVar, Generic, List, Optional, Protocol, cast
from dataclasses import dataclass
import heapq

T = TypeVar('T')

class Comparable(Protocol):
 def __lt__(self, other: object) -> bool: ...

U = TypeVar('U', bound=Comparable)

@dataclass(order=True)
class PrioritizedItem(Generic[T]):
 priority: int
 item: T

class PriorityQueue(Generic[T]):
 def __init__(self) -> None:
 self._queue: List[PrioritizedItem[T]] = []

 def add(self, item: T, priority: int) -> None:
 heapq.heappush(self._queue, PrioritizedItem(priority, item))

 def get(self) -> Optional[T]:
 if not self._queue:
 return None
 return heapq.heappop(self._queue).item

 def peek(self) -> Optional[T]:
 if not self._queue:
 return None
 return self._queue[0].item

 def is_empty(self) -> bool:
 return len(self._queue) == 0
```

Using Our Generic Priority Queue

```
# Usage with strings
task_queue = PriorityQueue[str]()
task_queue.add("Send email", 2)
task_queue.add("Fix critical bug", 1)
task_queue.add("Update documentation", 3)

# Items come out in priority order
next_task = task_queue.get()  # "Fix critical bug"
next_task = task_queue.get()  # "Send email"
next_task = task_queue.get()  # "Update documentation"

# Usage with custom class
@dataclass
class Task:
    name: str
    description: str
    estimated_hours: float

job_queue = PriorityQueue[Task]()
job_queue.add(Task("UI redesign", "Update the navigation menu", 4.5), 2)
job_queue.add(Task("Database migration", "Upgrade to latest version", 8.0), 1)

next_job = job_queue.get()  # Database migration task
```

This implementation demonstrates how generics allow us to create flexible data structures that work with any type while maintaining type safety. The type checker will ensure that if you create a PriorityQueue[str], you can only add strings to it and will only get strings from it.

Buffer Protocol in Python

Understanding memory-efficient data handling

What is the Buffer Protocol?

The buffer protocol is a low-level interface in Python that enables direct access to an object's internal data buffers without requiring intermediate copies. It's a cornerstone of Python's performance optimization capabilities.

Key characteristics:

- Allows different Python objects to share memory
- Enables zero-copy operations between compatible data structures
- Provides direct access to raw memory
- Crucial for scientific computing and data processing

The buffer protocol is what makes libraries like NumPy, Pandas, and PyTorch so efficient when handling large datasets.

Python objects supporting the buffer protocol:

- `bytes`, `bytearray`, `memoryview`
- `array.array` from the standard library
- NumPy arrays (`numpy.ndarray`)
- Many image processing libraries (PIL/Pillow)
- Data processing libraries (Pandas, PyTorch, TensorFlow)

At its core, the buffer protocol enables:

- Exposing internal data buffers
- Describing the memory layout (shape, strides, format)
- Controlling access (read-only vs. read-write)
- Sharing memory between objects efficiently

Memory Views: Python's Window into Memory

The `memoryview` object is Python's primary interface to the buffer protocol. It provides a way to access the internal data of an object that supports the buffer protocol without copying the data.

```
# Creating a memory view from a bytes object
data = bytes(range(10))  #
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09'
view = memoryview(data)

# Basic properties
print(len(view))        # 10
print(view.nbytes)      # 10 (total bytes)
print(view.readonly)    # True (bytes objects are immutable)
print(view.format)      # 'B' (unsigned char)
print(view.itemsize)    # 1 (byte)
print(view.ndim)        # 1 (one-dimensional)
print(view.shape)       # (10,) (like NumPy shape)

# Accessing data without copying
first_byte = view[0]    # 0
subset = view[1:4]      # memoryview of bytes 1, 2, 3
```

Memory views can be modified if the underlying buffer is mutable:

```
# With mutable underlying buffer
buffer = bytearray(range(10))
view = memoryview(buffer)

# Modify through the memory view
view[0] = 42
print(buffer)  # bytearray(b'*\x01\x02\x03\x04\x05\x06\x07\x08\t')

# Slicing creates new memory views without copying
first_half = view[:5]
second_half = view[5:]

# Changing the view changes the original buffer
first_half[1] = 99
print(buffer)  # bytearray(b'*c\x02\x03\x04\x05\x06\x07\x08\t')
```

Memory views are particularly useful when working with large data structures, as they allow you to operate on sections of data without making copies, saving both time and memory.

Advanced Memory View Operations

Memory views support multidimensional data and different data formats:

```
import array

# 2D array (3x4 matrix) of integers
matrix = array.array('i', [
    1, 2, 3, 4,    # Row 1
    5, 6, 7, 8,    # Row 2
    9, 10, 11, 12  # Row 3
])

# Create a memory view
view = memoryview(matrix)

# Reshape to 2D (this doesn't copy data)
view_2d = view.cast('i', shape=(3, 4))

# Access by row and column
print(view_2d[1, 2])  # 7 (row 1, column 2)

# Slice entire rows
row_1 = view_2d[0]    # First row: 1, 2, 3, 4
row_2 = view_2d[1]    # Second row: 5, 6, 7, 8

# Modifying a slice modifies the original data
row_1[0] = 99
print(matrix)         # array('i', [99, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

Converting between data formats:

```
# Create a bytes object with 8 bytes
double_bytes = bytes([
    0x55, 0x55, 0x55, 0x55, 0x55, 0x55, 0xd5, 0x3f
])  # IEEE 754 representation of 1/3

# View as a double (8 bytes)
view = memoryview(double_bytes)
float_view = view.cast('d')  # 'd' is double precision float
print(float_view[0])  # Approximately 0.3333333333333333

# Using struct for more complex formats
import struct
from datetime import datetime

# Create a buffer with a timestamp and values
buffer = bytearray(16)
struct.pack_into('QdQ', buffer, 0,
                 int(datetime.now().timestamp()),
                 3.14159,
                 1000000000)

# Access through memory view
view = memoryview(buffer)
timestamp = struct.unpack('Q', view[:8])[0]
value = struct.unpack('d', view[8:16])[0]
```

These capabilities make memory views powerful for working with binary data structures and interoperating with C libraries and hardware interfaces.

Use Cases: Efficient Data Handling with NumPy

NumPy leverages the buffer protocol to provide highly efficient array operations, making it ideal for numerical computing. Here's how the buffer protocol enhances NumPy's performance:

```
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5], dtype=np.int32)

# Create a memory view of the array
view = memoryview(arr)

# Properties of the view
print(view.format)  # 'i' (32-bit integer)
print(view.itemsize)  # 4 (bytes per integer)
print(view.shape)  # (5,)
print(view.strides)  # (4,) (bytes to step in each dimension)

# Convert between NumPy array and bytes without copying
bytes_data = arr.tobytes()  # Makes a copy
bytes_view = arr.view(np.uint8)  # No copy, view as bytes

# Create array from existing buffer
buffer = bytearray(20)  # 5 integers * 4 bytes
arr2 = np.frombuffer(buffer, dtype=np.int32)
arr2[0] = 42  # Modifies the original buffer
```

Zero-copy operations between NumPy and other libraries:

```
import numpy as np
from PIL import Image
import io

# Create a simple 3x3 RGB image array
img_array = np.zeros((3, 3, 3), dtype=np.uint8)
img_array[0, 0] = [255, 0, 0]  # Red pixel
img_array[1, 1] = [0, 255, 0]  # Green pixel
img_array[2, 2] = [0, 0, 255]  # Blue pixel

# Convert to PIL Image without copying data
img = Image.fromarray(img_array)

# Save to in-memory file
buffer = io.BytesIO()
img.save(buffer, format='PNG')

# Load back to NumPy without extra copies
buffer.seek(0)
loaded_img = Image.open(buffer)
new_array = np.array(loaded_img)

# Efficient data sharing with memory-mapped files
mmap_array = np.memmap('data.bin', dtype=np.float64,
                       mode='w+', shape=(1000, 1000))
# Use mmap_array like a regular ndarray, but it's stored on disk
```

These capabilities are crucial for data science and machine learning applications where efficient memory usage and high-performance data processing are essential.

Use Cases: Efficient Data Handling with Pandas

Pandas builds on NumPy's buffer protocol capabilities to provide efficient data analysis for tabular data. Here's how Pandas leverages the buffer protocol:

```
import pandas as pd
import numpy as np

# Create a DataFrame
df = pd.DataFrame({
    'A': np.random.rand(1000000),
    'B': np.random.rand(1000000),
    'C': np.random.rand(1000000)
})

# Access the underlying NumPy array without copying
arr_a = df['A'].values  # No copy, direct reference
arr_a[0] = 999  # Modifies the original DataFrame

# Efficient operations using vectorized NumPy functions
result = np.sqrt(df['A'] ** 2 + df['B'] ** 2)  # Fast, no loops

# Memory usage before optimization
print(df.memory_usage(deep=True).sum() / 1e6, "MB")

# Convert to more efficient data types using buffer protocol
df_optimized = df.copy()
df_optimized['A'] = df_optimized['A'].astype(np.float32)
df_optimized['B'] = df_optimized['B'].astype(np.float32)
df_optimized['C'] = df_optimized['C'].astype(np.float32)

# Memory usage after optimization
print(df_optimized.memory_usage(deep=True).sum() / 1e6, "MB")
# About 50% reduction
```

Zero-copy operations with Pandas:

```
import pandas as pd
import numpy as np

# Create a large DataFrame
df = pd.DataFrame({
    'id': range(1000000),
    'value': np.random.randn(1000000)
})

# Memory-efficient operations
# 1. Use inplace where possible
df.sort_values('id', inplace=True)

# 2. Leverage NumPy's view operations
values = df['value'].values  # No copy
mask = values > 0
positive_count = mask.sum()

# 3. Use categorical data for strings
sample_data = {
    'name': np.random.choice(
        ['Alice', 'Bob', 'Charlie', 'David'], 1000000
    ),
    'city': np.random.choice(
        ['New York', 'London', 'Tokyo', 'Paris'], 1000000
    )
}
df2 = pd.DataFrame(sample_data)

# Convert to categorical (uses integer codes under the hood)
df2['name'] = df2['name'].astype('category')
df2['city'] = df2['city'].astype('category')

# Massive memory savings for string columns
print(df2.memory_usage(deep=True))
```

These techniques are essential for data scientists working with large datasets where memory efficiency directly impacts processing speed and the ability to work with larger-than-memory datasets.

Hands-on: Implementing a Custom Buffer Protocol

Implementing the buffer protocol in custom classes allows them to interoperate efficiently with NumPy, Pandas, and other libraries. Let's implement a simple matrix class with buffer protocol support:

```
import ctypes
import struct

class Matrix:
 def __init__(self, rows, cols, data=None):
 self.rows = rows
 self.cols = cols
 self.format = 'd' # double precision
 self.itemsize = struct.calcsize(self.format)

 # Allocate memory
 buffer_size = rows * cols * self.itemsize
 self.buffer = bytearray(buffer_size)

 # Initialize data if provided
 if data is not None:
 if len(data) != rows * cols:
 raise ValueError("Data size doesn't match dimensions")
 for i, val in enumerate(data):
 struct.pack_into(self.format, self.buffer,
 i * self.itemsize, val)

 def __getitem__(self, key):
 row, col = key
 if not (0 <= row < self.rows and 0 <= col < self.cols):
 raise IndexError("Matrix indices out of range")

 offset = (row * self.cols + col) * self.itemsize
 return struct.unpack_from(self.format, self.buffer, offset)[0]

 def __setitem__(self, key, value):
 row, col = key
 if not (0 <= row < self.rows and 0 <= col < self.cols):
 raise IndexError("Matrix indices out of range")

 offset = (row * self.cols + col) * self.itemsize
 struct.pack_into(self.format, self.buffer, offset, value)
```

Adding Buffer Protocol Support

To support the buffer protocol, we need to implement the __buffer__ interface in Python 3.0+ or the new buffer protocol in Python 3.3+ with the memoryview object.

```
# Continuing from the previous Matrix class
 def __buffer__(self, flags):
 # Python 3.0+ buffer protocol
 return memoryview(self.buffer)

 # For Python 3.3+ with the new buffer protocol
 def __getbuffer__(self, view, flags):
 view.buf = ctypes.addressof(self.buffer)
 view.obj = self
 view.len = len(self.buffer)
 view.readonly = False
 view.format = self.format
 view.ndim = 2
 view.shape = (self.rows, self.cols)
 view.strides = (self.cols * self.itemsize, self.itemsize)
 view.suboffsets = None
 view.itemsize = self.itemsize

 def to_numpy(self):
 import numpy as np
 # Use the buffer protocol to create a NumPy array without copying
 return np.frombuffer(self.buffer, dtype=np.float64).reshape(self.rows, self.cols)

# Usage
matrix = Matrix(3, 3, [1, 2, 3, 4, 5, 6, 7, 8, 9])
print(matrix[0, 0]) # 1.0
print(matrix[1, 1]) # 5.0

# Direct access through memoryview
view = memoryview(matrix)
import numpy as np
np_array = np.asarray(view)
print(np_array) # 3x3 matrix without copying data
```

Real-world Example: Image Processing with Buffer Protocol

Let's see how the buffer protocol enables efficient image processing by avoiding unnecessary copies:

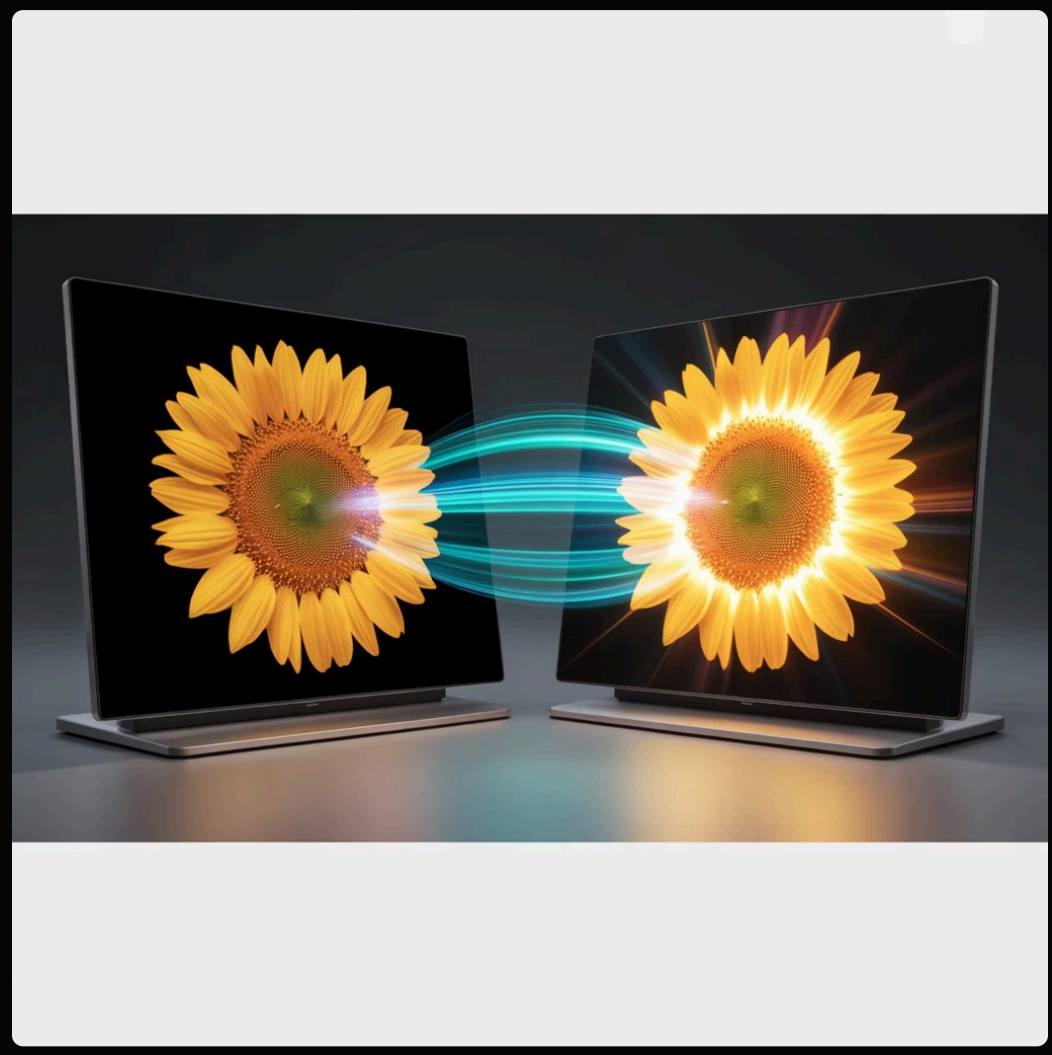
```
import numpy as np
from PIL import Image
import time

# Load an image
img = Image.open('large_image.jpg')

# Time the traditional approach (with copying)
start = time.time()
# Convert to NumPy array (makes a copy)
np_img = np.array(img)
# Apply a simple operation (brighten the image)
brightened = np_img * 1.5
# Clip values to valid range
brightened = np.clip(brightened, 0, 255).astype(np.uint8)
# Convert back to PIL Image (another copy)
result_img = Image.fromarray(brightened)
traditional_time = time.time() - start

# Time the buffer protocol approach
start = time.time()
# Get a memory view directly
buffer = memoryview(img.tobytes())
# Create NumPy array from the buffer without copying
np_img = np.frombuffer(buffer, dtype=np.uint8)
np_img = np_img.reshape(img.height, img.width,
len(img.getbands()))
# Process in-place when possible
np.multiply(np_img, 1.5, out=np_img)
np.clip(np_img, 0, 255, out=np_img)
# Create new image sharing the same buffer
result_img = Image.frombuffer(
    img.mode, img.size, np_img, 'raw', img.mode, 0, 1
)
buffer_time = time.time() - start

print(f"Traditional approach: {traditional_time:.4f} seconds")
print(f"Buffer protocol approach: {buffer_time:.4f} seconds")
print(f"Speedup: {traditional_time / buffer_time:.2f}x")
```



The buffer protocol approach is significantly faster, especially for large images, because it avoids unnecessary memory copies. This becomes even more important when processing video frames or multiple images in sequence.

Similar techniques are used in:

- Computer vision libraries like OpenCV
- Scientific visualization tools
- Machine learning frameworks for efficient data preprocessing
- Audio processing applications

The buffer protocol is one of Python's hidden performance features that makes it viable for high-performance computing applications despite being an interpreted language.

PySpark for Distributed Computing

Scaling Python to handle big data processing

Introduction to PySpark

PySpark is the Python API for Apache Spark, a fast and general-purpose cluster computing system. It provides:

- A unified analytics engine for large-scale data processing
- In-memory computation capabilities
- Fault tolerance and data parallelism
- Support for batch processing, interactive queries, streaming, and machine learning

PySpark bridges the gap between Python's ease of use and Spark's powerful distributed computing capabilities, enabling you to process data at scale without switching to a different language.

Key components of PySpark:

Spark Core

The foundation that provides distributed task scheduling, memory management, and fault recovery

Spark SQL

Module for structured data processing with SQL queries

Spark Streaming

Real-time data processing from various sources

MLlib

Machine learning library with common algorithms and utilities

When to Use PySpark

Ideal Use Cases:

- **Big Data Processing**: When your dataset is too large for a single machine's memory
- **Distributed ETL**: Extract, transform, and load operations on large datasets
- **Complex Analytics**: When you need to perform complex operations like joins and aggregations on large datasets
- **Machine Learning at Scale**: Training models on data too large for scikit-learn or similar libraries
- **Stream Processing**: Processing real-time data from sources like Kafka or Flume
- **Graph Processing**: Analyzing relationships in large networks

When to Consider Alternatives:

- **Small Datasets**: For data that fits in memory, pandas or vanilla Python may be faster due to lower overhead
- **Simple Processing**: For straightforward operations, simpler tools may be more appropriate
- **Single-Machine Performance**: When you need to squeeze maximum performance from a single machine, specialized libraries may be better
- **Low Latency Requirements**: For applications requiring millisecond responses, Spark's batch nature may be too slow

PySpark shines when you need to process data that's too large for a single machine or when you need to leverage distributed computing resources efficiently.

Setting Up PySpark (Local Mode)

To get started with PySpark in local mode (no cluster required), you'll need to:

1. Install Java (JDK 8 or higher required)
2. Install Python (3.6 or higher recommended)
3. Install PySpark using pip:

```
pip install pyspark
```

Additional dependencies for specific components:

```
# For Spark SQL
pip install pyspark[sql]

# For machine learning
pip install pyspark[ml]

# For pandas integration
pip install pyspark[pandas_on_spark]
```

Environment variables that may need to be set:

```
# For Windows:
set JAVA_HOME=C:\path\to\your\java
set SPARK_HOME=C:\path\to\pyspark\directory
set HADOOP_HOME=C:\path\to\hadoop\directory
set PYTHONPATH=%SPARK_HOME%\python;%PYTHONPATH%

# For Linux/Mac:
export JAVA_HOME=/path/to/your/java
export SPARK_HOME=/path/to/pyspark/directory
export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH
```

Verifying your installation:

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder \
    .appName("PySpark Test") \
    .master("local[*]") \
    .getOrCreate()

# Test with a simple operation
df = spark.createDataFrame([("Hello", 1), ("World", 2)],
                           ["word", "count"])
df.show()

# Stop the session
spark.stop()
```

If the code runs successfully and displays a table with "Hello" and "World", your PySpark installation is working correctly.

The SparkContext and SparkSession

SparkContext (sc)

The SparkContext is the entry point to any Spark functionality. It represents the connection to a Spark cluster and can be used to create RDDs, accumulators, and broadcast variables.

```
from pyspark import SparkContext, SparkConf

# Create a SparkConf object to configure the app
conf = SparkConf().setAppName("MyApp").setMaster("local[4]")

# Create a SparkContext
sc = SparkContext(conf=conf)

# Create an RDD
rdd = sc.parallelize([1, 2, 3, 4, 5])

# Process the RDD
squared = rdd.map(lambda x: x * x)
print(squared.collect())  # [1, 4, 9, 16, 25]

# Stop the context when done
sc.stop()
```

SparkSession (recommended for most use cases)

The SparkSession, introduced in Spark 2.0, provides a unified entry point for working with Spark DataFrames and Spark SQL. It encapsulates a SparkContext and provides more convenient methods for working with structured data.

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder \
    .appName("MyApp") \
    .master("local[4]") \
    .config("spark.executor.memory", "2g") \
    .getOrCreate()

# Get the underlying SparkContext if needed
sc = spark.sparkContext

# Create a DataFrame
df = spark.createDataFrame([
    (1, "John", 25),
    (2, "Alice", 30),
    (3, "Bob", 35)
], ["id", "name", "age"])

# Show the DataFrame
df.show()

# Access Spark SQL functionality
df.createOrReplaceTempView("people")
results = spark.sql("SELECT * FROM people WHERE age > 25")
results.show()

# Stop the session when done
spark.stop()
```

In modern Spark applications, SparkSession is the preferred entry point as it provides a unified interface to all Spark functionality. You should use SparkSession for new applications, especially those that work with structured data.

Resilient Distributed Datasets (RDDs)

RDDs (Resilient Distributed Datasets) are the fundamental data structure in Spark. They are:

- **Resilient**: Fault-tolerant with the ability to recompute missing or damaged partitions
- **Distributed**: Data is distributed across multiple nodes in a cluster
- **Dataset**: A collection of partitioned data with primitive values or values of user-defined classes

RDDs support two types of operations:

1. **Transformations**: Create a new RDD from an existing one (e.g., map, filter, join)
2. **Actions**: Return a value to the driver program after running a computation (e.g., count, collect, save)

RDDs are **lazily evaluated**: transformations are only computed when an action requires a result.

Creating RDDs:

```
# From a collection
rdd1 = sc.parallelize([1, 2, 3, 4, 5])

# From a file or directory
rdd2 = sc.textFile("data.txt")
rdd3 = sc.wholeTextFiles("data_dir/")

# From another RDD
rdd4 = rdd1.map(lambda x: x * x)
```

Common RDD transformations:

```
# Basic transformations
filtered = rdd.filter(lambda x: x % 2 == 0)  # Even numbers
mapped = rdd.map(lambda x: (x, 1))  # Create key-value pairs
flattened = rdd.flatMap(lambda x: range(x))  # Flatten results

# Key-value pair operations (for RDDs of tuples)
grouped = mapped.groupByKey()
reduced = mapped.reduceByKey(lambda a, b: a + b)
joined = rdd1.join(rdd2)  # Join two key-value RDDs
```

Common RDD actions:

```
# Retrieve results
result = rdd.collect() # Return all elements
first_item = rdd.first() # Return first element
sample = rdd.take(5) # Return first 5 elements

# Aggregations
count = rdd.count() # Count elements
total = rdd.reduce(lambda a, b: a + b) # Sum elements
stats = rdd.stats() # Summary statistics for numeric RDDs

# Save results
rdd.saveAsTextFile("output_dir/")
```

RDD Example: Word Count

```
from pyspark import SparkContext, SparkConf

# Configure and initialize SparkContext
conf = SparkConf().setAppName("WordCount").setMaster("local[*]")
sc = SparkContext(conf=conf)

# Load text file as RDD
lines = sc.textFile("shakespeare.txt")

# Transform the data: split into words, create (word, 1) pairs, count by key
words = lines.flatMap(lambda line: line.split(" "))
word_counts = words.map(lambda word: (word.lower(), 1)) \
                .reduceByKey(lambda a, b: a + b)

# Sort by count (descending)
sorted_counts = word_counts.map(lambda x: (x[1], x[0])) \
                .sortByKey(False) \
                .map(lambda x: (x[1], x[0]))

# Take the top 20 words
top_words = sorted_counts.take(20)

# Print the results
for word, count in top_words:
    print(f"{word}: {count}")

# Stop the SparkContext
sc.stop()
```

This example demonstrates a classic word count problem using RDDs. The process follows a few steps: reading text data, splitting into words, counting occurrences of each word, sorting by frequency, and retrieving the top results. This pattern is common in many Spark applications, especially those dealing with text analysis.

Understanding DataFrames in PySpark

DataFrames are a higher-level abstraction built on top of RDDs. They provide:

- A distributed collection of data organized into named columns
- Conceptually similar to tables in a relational database or pandas DataFrames
- Optimized execution through Spark SQL's Catalyst optimizer
- Rich set of functions and a more intuitive API than RDDs

DataFrames were introduced to make Spark more accessible to data scientists familiar with pandas and SQL, while providing better performance through optimization.

Advantages of DataFrames over RDDs:

- **Optimization**: Catalyst optimizer can significantly improve performance
- **Schema awareness**: DataFrames understand the structure of your data
- **SQL support**: Query data using SQL syntax
- **Structured API**: More intuitive for data manipulation
- **Integration**: Better integration with data sources like Hive, Parquet, JSON, etc.
- **Language compatibility**: Write once, run anywhere (Java, Scala, Python, R)

For most new Spark applications, especially those involving structured data, DataFrames are the recommended approach over RDDs.

Creating DataFrames

There are multiple ways to create DataFrames in PySpark:

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, \
                              StringType, IntegerType

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("DataFrameExamples") \
    .master("local[*]") \
    .getOrCreate()

# 1. From a list of tuples
data = [("Alice", 34), ("Bob", 45), ("Charlie", 29)]
df1 = spark.createDataFrame(data, ["name", "age"])

# 2. From a pandas DataFrame
import pandas as pd
pandas_df = pd.DataFrame({
    "name": ["Alice", "Bob", "Charlie"],
    "age": [34, 45, 29]
})
df2 = spark.createDataFrame(pandas_df)
```

```
# 3. From an RDD
rdd = spark.sparkContext.parallelize(data)
df3 = spark.createDataFrame(rdd, ["name", "age"])

# 4. From a file
df4 = spark.read.csv("people.csv", header=True,
                     inferSchema=True)
df5 = spark.read.json("people.json")
df6 = spark.read.parquet("people.parquet")

# 5. With explicit schema
schema = StructType([
    StructField("name", StringType(), nullable=False),
    StructField("age", IntegerType(), nullable=True)
])
df7 = spark.createDataFrame(data, schema)
df8 = spark.read.csv("people.csv", header=True, schema=schema)

# Display the DataFrame
df1.show()
```

Specifying the schema explicitly is recommended for production applications as it provides better control over data types and constraints, and can improve performance by avoiding schema inference.

Working with DataFrames: Basic Operations

Basic operations to explore and manipulate DataFrames:

```
# Explore DataFrame structure
df.printSchema()  # Print the schema
df.show(5, truncate=False)  # Display 5 rows without truncating
df.head(3)  # Return the first 3 rows as a list
df.describe().show()  # Summary statistics for numeric columns
df.count()  # Count rows
df.columns  # List column names
df.dtypes  # List column names and data types

# Selection operations
df.select("name", "age").show()  # Select specific columns
df.select(df["name"], df["age"] + 1).show()  # Expressions
df.selectExpr("name", "age + 1 as increased_age").show()
df.filter(df["age"] > 30).show()  # Filter rows (WHERE)
df.filter("age > 30").show()  # SQL expression filter
df.distinct().show()  # Remove duplicates
df.limit(2).show()  # Limit results
```

Data manipulation operations:

```
# Add, rename, and drop columns
df.withColumn("age_doubled", df["age"] * 2).show()
df.withColumnRenamed("age", "years").show()
df.drop("age").show()

# Sorting
df.sort("age").show()  # Ascending
df.sort(df["age"].desc()).show()  # Descending
df.orderBy(["name", "age"]).show()  # Multiple columns

# Aggregations
from pyspark.sql.functions import count, avg, sum, min, max

df.groupBy("department").count().show()
df.groupBy("department").agg(
    count("*").alias("employee_count"),
    avg("salary").alias("avg_salary"),
    sum("salary").alias("total_salary")
).show()

# Joins
employees.join(
    departments,
    employees["dept_id"] == departments["id"],
    "inner"
).show()
```

These operations form the foundation of data manipulation with PySpark DataFrames, allowing you to perform transformations similar to SQL operations but with the benefits of distributed computing.

Working with DataFrames: SQL Queries

One of the powerful features of PySpark is the ability to use SQL queries directly on DataFrames. This is especially useful for data analysts and scientists who are already familiar with SQL.

```
# Register DataFrame as a temporary view
df.createOrReplaceTempView("people")

# Run SQL queries
results = spark.sql("""
    SELECT name, age
    FROM people
    WHERE age > 30
    ORDER BY age DESC
""")

results.show()

# More complex SQL
analytics_results = spark.sql("""
    SELECT
        department,
        COUNT(*) as employee_count,
        AVG(salary) as avg_salary,
        MIN(age) as youngest,
        MAX(age) as oldest
    FROM people
    GROUP BY department
    HAVING COUNT(*) > 5
    ORDER BY avg_salary DESC
""")

analytics_results.show()
```

SQL functions and window operations:

```
# Register multiple DataFrames
employees.createOrReplaceTempView("employees")
departments.createOrReplaceTempView("departments")

# Join in SQL
joined_data = spark.sql("""
    SELECT e.name, e.salary, d.dept_name
    FROM employees e
    JOIN departments d ON e.dept_id = d.id
""")

# Window functions in SQL
rank_data = spark.sql("""
    SELECT
        name,
        department,
        salary,
        RANK() OVER (
            PARTITION BY department
            ORDER BY salary DESC
        ) as salary_rank
    FROM people
""")

# Using CASE statements
case_example = spark.sql("""
    SELECT
        name,
        age,
        CASE
            WHEN age < 30 THEN 'Young'
            WHEN age BETWEEN 30 AND 50 THEN 'Middle-aged'
            ELSE 'Senior'
        END as age_group
    FROM people
""")
```

SQL queries provide a familiar interface for data manipulation and analysis, making PySpark accessible to users with SQL backgrounds. The queries are translated to the same optimized execution plan as the DataFrame API operations.

DataFrame Transformations and Actions

Transformations

Like RDDs, DataFrames use lazy evaluation with transformations and actions. Transformations create a new DataFrame without modifying the original:

- `select()`, `filter()`, `groupBy()`, `orderBy()`
- `withColumn()`, `drop()`, `withColumnRenamed()`
- `join()`, `union()`, `intersect()`, `except()`
- `repartition()`, `coalesce()`
- `cache()`, `persist()`

Transformations are lazy, meaning they are not executed until an action is called. This allows Spark to optimize the execution plan.

Actions

Actions trigger computation and return results:

- `show()`: Display DataFrame contents
- `collect()`: Return all rows as a list
- `count()`: Return the number of rows
- `first()`, `head()`, `take()`: Return rows
- `write.save()`: Save DataFrame to storage
- `foreach()`: Apply a function to each row
- `describe()`: Generate summary statistics

When an action is called, Spark builds an optimized execution plan for all transformations needed to produce the result, then executes this plan. This is different from eager evaluation, where each operation would be executed immediately.

Understanding the distinction between transformations and actions is crucial for writing efficient Spark code. By chaining transformations together and only calling actions when necessary, you can take advantage of Spark's optimization capabilities.

DataFrame Example: Customer Analysis

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum, avg, count, desc, year, month, datediff

# Initialize SparkSession
spark = SparkSession.builder.appName("CustomerAnalysis").getOrCreate()

# Load data
customers = spark.read.csv("customers.csv", header=True, inferSchema=True)
orders = spark.read.csv("orders.csv", header=True, inferSchema=True)
products = spark.read.csv("products.csv", header=True, inferSchema=True)

# Register DataFrames as views for SQL
customers.createOrReplaceTempView("customers")
orders.createOrReplaceTempView("orders")
products.createOrReplaceTempView("products")

# Calculate customer lifetime value
customer_value = spark.sql("""
 SELECT
 c.customer_id,
 c.name,
 c.signup_date,
 COUNT(DISTINCT o.order_id) as order_count,
 SUM(o.total_amount) as total_spent,
 AVG(o.total_amount) as avg_order_value,
 MAX(o.order_date) as last_order_date,
 datediff(current_date(), MAX(o.order_date)) as days_since_last_order
 FROM customers c
 LEFT JOIN orders o ON c.customer_id = o.customer_id
 GROUP BY c.customer_id, c.name, c.signup_date
 ORDER BY total_spent DESC
""")

# Identify top-selling products
top_products = spark.sql("""
 SELECT
 p.product_id,
 p.product_name,
 p.category,
 COUNT(o.order_id) as times_ordered,
 SUM(o.quantity) as total_quantity_sold,
 SUM(o.quantity * p.price) as total_revenue
 FROM products p
 JOIN orders o ON p.product_id = o.product_id
 GROUP BY p.product_id, p.product_name, p.category
 ORDER BY total_revenue DESC
 LIMIT 10
""")

# Show results
customer_value.show(10)
top_products.show(10)

# Save results
customer_value.write.mode("overwrite").parquet("customer_value.parquet")
top_products.write.mode("overwrite").parquet("top_products.parquet")

# Stop SparkSession
spark.stop()
```

Working with CSV, JSON, and Parquet Files

Reading and Writing Different File Formats

```
# CSV
df_csv = spark.read.csv("data.csv",
                        header=True,
                        inferSchema=True)
df.write.csv("output/csv_data",
             header=True,
             mode="overwrite")

# JSON
df_json = spark.read.json("data.json")
df.write.json("output/json_data",
              mode="overwrite")

# Parquet (columnar storage format)
df_parquet = spark.read.parquet("data.parquet")
df.write.parquet("output/parquet_data",
                 mode="overwrite")

# ORC (another columnar format)
df_orc = spark.read.orc("data.orc")
df.write.orc("output/orc_data",
             mode="overwrite")

# JDBC (databases)
df_jdbc = spark.read.jdbc(
    url="jdbc:postgresql://localhost/database",
    table="table_name",
    properties={"user": "username", "password": "password"}
)
df.write.jdbc(
    url="jdbc:postgresql://localhost/database",
    table="output_table",
    mode="overwrite",
    properties={"user": "username", "password": "password"}
)
```

Parquet: The Recommended Format

Parquet is a columnar storage format that provides several advantages for big data processing:

- **Efficient compression**: Typically 2-4x smaller than CSV or JSON
- **Column pruning**: Only read the columns you need
- **Predicate pushdown**: Filter data during read instead of after loading
- **Schema enforcement**: Built-in schema information
- **Better performance**: Optimized for analytical queries

```
# Writing with partitioning (important for large datasets)
df.write.partitionBy("year", "month") \
      .parquet("data_partitioned.parquet")

# Reading with column selection and filtering
df_subset = spark.read.parquet("data.parquet") \
                 .select("id", "name", "value") \
                 .filter("value > 100")

# Saving with compression
df.write.option("compression", "snappy") \
      .parquet("data_compressed.parquet")
```

For most production Spark applications, Parquet is the recommended file format due to its performance and space efficiency.

Performance Optimization in PySpark

Partitioning

Control the distribution of data across the cluster:

- Use `repartition(n)` to increase partitions for more parallelism
- Use `coalesce(n)` to reduce partitions without shuffling
- Partition data files by frequently filtered columns

Caching and Persistence

Store intermediate results in memory or disk:

- Use `cache()` or `persist()` for reused DataFrames
- Choose appropriate storage level based on memory availability
- Use `unpersist()` when data is no longer needed

Broadcast Joins

Optimize joins with small tables:

- Use `broadcast()` to avoid shuffling
- Ideal when one table is small enough to fit in memory
- Can dramatically improve join performance

Predicate Pushdown

Filter data early in the process:

- Apply filters before joins or aggregations
- Use file formats that support predicate pushdown (Parquet, ORC)
- Let Spark optimize query execution

Performance Optimization Examples

Caching and Persistence

```
# Basic caching (MEMORY_AND_DISK storage level)
df.cache()

# More control with persist
from pyspark.storagelevel import StorageLevel
df.persist(StorageLevel.MEMORY_ONLY)  # Memory only
df.persist(StorageLevel.DISK_ONLY)    # Disk only
df.persist(StorageLevel.OFF_HEAP)     # Off-heap memory

# Remove from cache when done
df.unpersist()
```

Broadcast Joins

```
from pyspark.sql.functions import broadcast

# Regular join (might trigger shuffling)
result = large_df.join(small_df, "key")

# Broadcast join (avoids shuffling)
result = large_df.join(broadcast(small_df), "key")
```

Partitioning

```
# Increase partitions for more parallelism
df_repartitioned = df.repartition(100)

# Repartition by specific column for join optimization
df_repartitioned = df.repartition("join_key")

# Reduce partitions without full shuffle
df_coalesced = df.coalesce(10)

# Partitioning during write
df.write.partitionBy("year", "month") \
      .parquet("partitioned_data")
```

Query Optimization

```
# Explain the execution plan
df.filter(df.age > 30).select("name").explain()

# Detailed explain
df.filter(df.age > 30).select("name").explain(True)

# Set configuration for optimization
spark.conf.set("spark.sql.shuffle.partitions", 200)
spark.conf.set("spark.sql.autoBroadcastJoinThreshold",
 10 * 1024 * 1024) # 10MB
```

Hands-on: Writing and Running a Simple PySpark Job

Let's create a complete PySpark application that analyzes a dataset of e-commerce transactions:

```
# save as ecommerce_analysis.py
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum, avg, count, desc, month, year, expr

# Initialize SparkSession
spark = SparkSession.builder \
 .appName("E-commerce Analysis") \
 .master("local[*]") \
 .getOrCreate()

# Load the dataset
df = spark.read.csv("transactions.csv", header=True, inferSchema=True)

# Clean and transform data
transactions = df.filter(col("amount") > 0) \
 .withColumn("purchase_month", month("transaction_date")) \
 .withColumn("purchase_year", year("transaction_date"))

# Create a temporary view for SQL queries
transactions.createOrReplaceTempView("transactions")

# 1. Monthly sales analysis
monthly_sales = spark.sql("""
 SELECT
 purchase_year,
 purchase_month,
 COUNT(*) as transaction_count,
 SUM(amount) as total_sales,
 AVG(amount) as average_sale
 FROM transactions
 GROUP BY purchase_year, purchase_month
 ORDER BY purchase_year, purchase_month
""")

# 2. Top customers
top_customers = spark.sql("""
 SELECT
 customer_id,
 COUNT(*) as transaction_count,
 SUM(amount) as total_spent,
 AVG(amount) as average_purchase
 FROM transactions
 GROUP BY customer_id
 ORDER BY total_spent DESC
 LIMIT 10
""")

# 3. Product category analysis
category_analysis = spark.sql("""
 SELECT
 product_category,
 COUNT(*) as transaction_count,
 SUM(amount) as total_sales,
 AVG(amount) as average_sale
 FROM transactions
 GROUP BY product_category
 ORDER BY total_sales DESC
""")

# Save results
monthly_sales.write.mode("overwrite").parquet("monthly_sales.parquet")
top_customers.write.mode("overwrite").parquet("top_customers.parquet")
category_analysis.write.mode("overwrite").parquet("category_analysis.parquet")

# Show sample results
print("Monthly Sales Sample:")
monthly_sales.show(5)

print("Top Customers Sample:")
top_customers.show(5)

print("Category Analysis Sample:")
category_analysis.show(5)

# Stop SparkSession
spark.stop()
```

Running the PySpark Job

Local Mode

Run the job using spark-submit:

```
spark-submit \
  --master local[*] \
  --executor-memory 4g \
  ecommerce_analysis.py
```

Or use the Python interpreter with PySpark installed:

```
python ecommerce_analysis.py
```

Cluster Mode

Submit to a YARN cluster:

```
spark-submit \
  --master yarn \
  --deploy-mode cluster \
  --executor-memory 4g \
  --num-executors 10 \
  --executor-cores 2 \
  ecommerce_analysis.py
```

Monitoring the Job

Access the Spark UI to monitor the job (typically at http://localhost:4040 in local mode or through the YARN ResourceManager in cluster mode).

The Spark UI provides information about:

- Job progress and status
- Stage details and task execution
- Storage usage for cached data
- Executor information
- Environment configuration
- SQL query details

The Spark UI is a powerful tool for debugging and optimizing Spark applications, helping you identify bottlenecks and inefficiencies in your code.

PySpark for Machine Learning with MLlib

MLlib is Spark's machine learning library, providing scalable implementations of common machine learning algorithms and utilities. It's designed to scale to large datasets and leverage distributed computing.

Key features of MLlib:

- Algorithms for classification, regression, clustering, etc.
- Feature extraction, transformation, and selection
- Pipeline API for building and evaluating workflows
- Utilities for model evaluation and hyperparameter tuning
- Distributed implementation for big data

Simple example: Training a classification model

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Initialize SparkSession
spark =
SparkSession.builder.appName("MLExample").getOrCreate()

# Load data
data = spark.read.csv("customer_churn.csv", header=True,
 inferSchema=True)

# Prepare features
feature_cols = ["tenure", "monthlyCharges", "totalCharges"]
assembler = VectorAssembler(
 inputCols=feature_cols,
 outputCol="features"
)
data = assembler.transform(data)

# Split data
train, test = data.randomSplit([0.8, 0.2], seed=42)

# Train a model
lr = LogisticRegression(
 featuresCol="features",
 labelCol="churn",
 maxIter=10
)
model = lr.fit(train)

# Make predictions
predictions = model.transform(test)

# Evaluate model
evaluator = BinaryClassificationEvaluator(
 rawPredictionCol="rawPrediction",
 labelCol="churn"
)
auc = evaluator.evaluate(predictions)
print(f"AUC: {auc}")
```

PySpark Streaming

Spark Streaming allows you to process data in real-time from sources like Kafka, Flume, or TCP sockets. It processes data in micro-batches, treating each batch as a small RDD or DataFrame.

There are two APIs for streaming in PySpark:

1. **Structured Streaming**: The newer, recommended API based on DataFrames
2. **DStream**: The older RDD-based API

Structured Streaming provides:

- End-to-end exactly-once semantics
- Event-time processing
- Windowed operations
- Integration with DataFrame and SQL APIs

Simple Structured Streaming example:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split, window

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("StreamingWordCount") \
    .getOrCreate()

# Create a streaming DataFrame from a socket source
lines = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Process the stream
words = lines.select(
    explode(split(lines.value, " ")).alias("word")
)
word_counts = words.groupBy("word").count()

# Start the query
query = word_counts.writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

# Wait for the query to terminate
query.awaitTermination()
```

This example reads text from a socket, splits it into words, counts them, and outputs the results to the console in real-time as new data arrives.

PySpark vs. pandas: When to Use Each

pandas

Best for:

- Small to medium datasets (fits in memory)
- Single-machine processing
- Interactive data exploration
- Complex, pandas-specific operations
- Lower latency requirements
- When you don't need distributed computing

Advantages:

- Faster for small datasets
- More comprehensive API
- Better integration with other Python libraries
- Lower overhead
- Simpler setup and use

PySpark

Best for:

- Large datasets (beyond single machine memory)
- Distributed processing across clusters
- Batch processing of massive data
- Integration with existing Spark ecosystem
- When fault tolerance is critical

Advantages:

- Scales to petabytes of data
- Distributed processing
- Fault tolerance
- Integrated streaming and ML capabilities
- SQL support
- Optimized execution engine

Many workflows combine both: use pandas for initial exploration and development on sample data, then scale up to PySpark for production processing of full datasets. The pandas-on-Spark API (previously known as Koalas) helps bridge this gap by providing a pandas-like API on top of Spark.

Bridging the Gap: pandas API on Spark

The pandas API on Spark (previously known as Koalas) provides a pandas-compatible API on top of PySpark, allowing you to use familiar pandas code with Spark's distributed computing capabilities.

```
# Import pandas API on Spark
import pyspark.pandas as ps

# Create a DataFrame
pdf = ps.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [5, 6, 7, 8]
})

# Use pandas-like operations
pdf['c'] = pdf['a'] + pdf['b']
result = pdf.groupby('a').sum()

# Convert between pandas API on Spark and pandas
import pandas as pd
regular_pandas_df = pdf.to_pandas()
spark_pandas_df = ps.from_pandas(regular_pandas_df)

# Convert to/from Spark DataFrame
spark_df = pdf.to_spark()
pdf2 = ps.DataFrame(spark_df)

# SQL operations
pdf.createOrReplaceTempView("my_table")
result = ps.sql("SELECT * FROM my_table WHERE a > 2")
```

Benefits of pandas API on Spark:

- Familiar pandas API for those already familiar with pandas
- Easy transition between single-machine and distributed computing
- Ability to scale pandas code to large datasets
- Access to Spark's distributed computing capabilities
- Simplified migration path from pandas to Spark

Limitations:

- Not 100% compatible with all pandas features
- Some operations may be less efficient than native Spark
- Higher overhead than regular pandas for small datasets
- Some pandas-specific optimizations aren't available

The pandas API on Spark is an excellent tool for data scientists who are familiar with pandas but need to scale their work to larger datasets. It provides a smooth transition path from single-machine to distributed computing.

Summary: Gradual Typing in Python

Key Concepts

- Gradual typing allows mixing dynamic and static typing
- Type hints provide documentation and enable static checking
- mypy is the primary tool for static type checking
- Generics enable type-safe container types
- Protocols support structural typing (duck typing)
- ABCs enforce interfaces through inheritance

Benefits

- Catch errors before runtime
- Improved code documentation
- Better IDE support
- Gradual adoption in existing codebases
- Preserves Python's flexibility

Best Practices

- Start with critical components
- Use appropriate level of type specificity
- Combine with good test coverage
- Leverage tools like mypy in CI/CD pipelines

Summary: Buffer Protocol

Key Concepts

- Low-level interface for sharing memory between objects
- Enables zero-copy operations
- memoryview provides a Python interface to the buffer protocol
- Crucial for efficient data handling in scientific computing
- Powers libraries like NumPy, Pandas, and image processing tools

Benefits

- Dramatic performance improvements for large data
- Reduced memory consumption
- Efficient interoperability between libraries
- Direct access to memory
- Support for multidimensional data and various data formats

Use Cases

- Data science and machine learning
- Image and video processing
- Interfacing with C libraries
- Working with binary data formats
- Memory-mapped files

Summary: PySpark

Key Concepts

- Distributed computing framework for big data
- RDDs: Low-level distributed data collections
- DataFrames: Structured data with schema
- SQL: Query data using familiar SQL syntax
- Transformations and actions for data processing
- Support for machine learning, streaming, and graph processing

Benefits

- Scales to petabytes of data
- Fault tolerance and data resilience
- Optimized execution through Catalyst optimizer
- Unified platform for batch, streaming, and ML
- Python interface to Spark's power
- Rich ecosystem of tools and libraries

Best Practices

- Use DataFrames over RDDs for structured data
- Optimize with partitioning, caching, and broadcast joins
- Choose appropriate file formats (Parquet recommended)
- Monitor execution with Spark UI
- Use pandas API on Spark for pandas-like interface

Integration of Today's Topics

Gradual Typing

Type safety for PySpark and buffer protocol implementations enhances code quality and helps catch errors early.

Python Ecosystem

All three topics enhance Python's capabilities for enterprise-grade applications with type safety, performance, and scalability.

Buffer Protocol

Powers efficient data handling in NumPy and Pandas, which can be used alongside PySpark for optimal performance across scales.

PySpark

Scales Python to big data, leveraging type-checked code for reliable distributed processing and buffer protocol for efficient data exchange.

The combination of these advanced features enables Python to serve as a complete platform for everything from type-safe application development to high-performance computing and big data analytics.

Real-world Application Example

A data processing pipeline that combines all three technologies:

```
 from typing import List, Dict, Optional, TypeVar, Generic
 import numpy as np
 import pandas as pd
 from pyspark.sql import SparkSession
 from pyspark.sql.functions import udf
 from pyspark.sql.types import ArrayType, DoubleType

 T = TypeVar('T')

 class EfficientBuffer(Generic[T]):
  """A type-safe buffer that efficiently handles data exchange."""

  def __init__(self, capacity: int) -> None:
  self.buffer = np.zeros(capacity, dtype=np.float64)
  self.view = memoryview(self.buffer)

  def get_data(self) -> np.ndarray:
  """Return the numpy array without copying."""
  return self.buffer

  def process(self, data: List[T]) -> np.ndarray:
  """Process data and return results efficiently."""
  # Implementation depends on type T
  pass

# Initialize Spark
spark = SparkSession.builder.appName("IntegratedPipeline").getOrCreate()

# Load and process large dataset
raw_data = spark.read.parquet("large_dataset.parquet")

# Define a UDF that uses buffer protocol for efficiency
def process_vector(values: List[float]) -> List[float]:
  # Convert to numpy without copying using buffer protocol
  np_array = np.array(values)
  # Process using numpy's efficient operations
  result = np.sqrt(np_array) * 2
  # Return as list (Spark will handle conversion)
  return result.tolist()

# Register UDF with proper return type
process_udf = udf(process_vector, ArrayType(DoubleType()))

# Apply the UDF to the DataFrame
processed = raw_data.withColumn("processed_values",
  process_udf("values"))

# Perform distributed aggregation
aggregated = processed.groupBy("category").agg(...)

# Save results
aggregated.write.parquet("results.parquet")
```

Further Learning Resources

Gradual Typing

- **Documentation**: Python's typing module documentation
- **mypy**: The mypy documentation and cheat sheets
- **Book**: "Robust Python" by Patrick Viafore
- **PEPs**: PEP 484, 526, 544, 585, 586, 589, 591, 612

Buffer Protocol

- **Documentation**: Python's Buffer Protocol documentation
- **Article**: "Understanding the Python Buffer Protocol"
- **Tutorial**: "Memory Management in Python" by Jake VanderPlas
- **NumPy Documentation**: "Array Interface Protocol"

PySpark

- **Official Documentation**: Apache Spark and PySpark docs
- **Book**: "Spark: The Definitive Guide" by Chambers and Zaharia
- **Book**: "Learning PySpark" by Drabas and Lee
- **Course**: "Big Data Analysis with Spark and Python" on Coursera
- **GitHub**: PySpark example repositories and notebooks

Community Resources

- **Stack Overflow**: Tags for python-typing, buffer-protocol, pyspark
- **GitHub**: typeshed project for Python library stubs
- **Conferences**: PyCon, PyData, and Spark Summit talks
- **Blogs**: Real Python, Towards Data Science

Practical Exercises to Try

Gradual Typing Exercise

Add type hints to an existing Python project and run mypy to check for type errors. Implement a generic data structure like a priority queue or sorted dictionary with proper type annotations.

Buffer Protocol Exercise

Create a custom class that implements the buffer protocol and integrates with NumPy. Benchmark the performance difference between using the buffer protocol and regular copying operations for large arrays.

PySpark Exercise

Implement a data processing pipeline using PySpark that reads from various data sources, performs transformations and aggregations, and writes the results in an optimized format. Compare performance with and without optimization techniques.

Integration Exercise

Combine all three topics by creating a type-annotated PySpark application that processes large datasets efficiently using the buffer protocol. Use mypy to check your code and benchmark the performance gains from your optimizations.

Questions?

Feel free to ask about any of the topics we've covered today:

- Gradual typing and type hints in Python
- The buffer protocol and memory views
- PySpark for distributed computing
- Integration of these technologies in real-world applications

You can also reach out via email at **training@chandrashekar.info** for follow-up questions or to discuss specific applications in your projects.

Thank You!

Thank you for participating in Day 3 of our Python Advanced Concepts and Performance Optimization course. We've covered:

- Gradual typing for more reliable code
- Buffer protocol for efficient memory usage
- PySpark for scaling Python to big data

Tomorrow we'll continue with advanced topics including:

- Python's C extensions
- Just-in-time compilation with Numba
- Parallel processing techniques

Contact Information

Instructor: Chandrashekar Babu
Email: training@chandrashekar.info
Course Materials: materials.python-advanced.com

Feedback

Please take a few minutes to complete the feedback form for today's session. Your input helps us improve the course content and delivery.

Feedback link: feedback.python-advanced.com/day3