# Advanced Python Functions for Data Science

Say OL

say_ol@yahoo.com

October 01, 2024

# Overview of Advanced Topics I

- First-Class Functions
- Higher-Order Functions
- Lambda Functions
- Decorators
- Function Annotations
- Closures
- Generators
- Coroutines
- Context Managers
- Partial Functions
- Using `*args` and `**kwargs`
- Type Hinting
- Docstrings and Documentation

# Overview of Advanced Topics II

- Error Handling in Functions
- Performance Considerations
- Memoization
- Function Caching
- Functional Programming Concepts
- Composing Functions
- Chaining Functions
- Using `functools` Module
- The `inspect` Module
- Dynamic Function Creation
- Using Functions with DataFrames
- Applying Functions to Data Structures
- Custom Function Libraries

# Overview of Advanced Topics III

- Testing Functions
- Debugging Function Issues
- Using Jupyter Notebooks for Functions
- Best Practices in Function Design
- Common Pitfalls in Python Functions
- Real-World Examples in Data Science
- Case Studies
- Interactive Q&A Session
- Resources for Further Learning
- Conclusion

# Recap: Basic Function Syntax

## Syntax

```
def function_name(parameters):
    # Function body
    return value
```

# First-Class Functions

- Functions are first-class citizens in Python.
- They can be assigned to variables, passed as arguments, and returned from other functions.

## Example

```
def add(x, y):
    return x + y

def operate(func, x, y):
    return func(x, y)

result = operate(add, 5, 10)  # Returns 15
```

# Higher-Order Functions

- Functions that take other functions as arguments or return them.
- Useful in functional programming paradigms.

## Example

```
def square(x):
    return x * x

def apply_function(func, value):
    return func(value)

result = apply_function(square, 5)  # Returns 25
```

# Lambda Functions

- Anonymous functions defined with the `lambda` keyword.
- Useful for short, throwaway functions.

## Example

```
add = lambda x, y: x + y
result = add(5, 10)  # Returns 15
```

# Decorators

- Modify or enhance functions without changing their code.
- Commonly used for logging, authentication, etc.

## Example

```
def decorator_function(original_function):
    def wrapper_function():
      print("Wrapper executed before {}".format(original_function._
         return original_function()
    return wrapper_function

@decorator_function
def display():
    return "Display function executed"

display()  # Calls the wrapper first
```

# Function Annotations

- Provide a way to attach metadata to function parameters and return values.
- Useful for documentation and type hinting.

## Example

```
def multiply(x: int, y: int) -> int:
    return x * y
```

# Closures

- Functions that capture the lexical scope in which they are defined.
- Allow access to free variables from the enclosing scope.

## Example

```
def outer_function(msg):
    def inner_function():
        print(msg)
    return inner_function

my_func = outer_function("Hello, World!")
my_func()  # Prints "Hello, World!"
```

# Generators

- Functions that use the `yield` statement to produce a series of values.
- Memory efficient and allow iteration over large data sets.

## Example

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for number in countdown(5):
    print(number)  # Prints 5, 4, 3, 2, 1
```

# Coroutines

- A special type of generator that can consume values.
- Useful for asynchronous programming and cooperative multitasking.

## Example

```
def coroutine_example():
    while True:
        value = yield
        print("Received:", value)

coro = coroutine_example()
next(coro)  # Initialize coroutine
coro.send(10)  # Prints "Received: 10"
```

# Context Managers

- Allow resource management (e.g., file handling) with the `with` statement.
- Ensure proper acquisition and release of resources.

## Example

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

# Partial Functions

- Create a new function with some arguments fixed.
- Useful for function customization.

## Example

```
from functools import partial

def power(base, exp):
    return base ** exp

square = partial(power, exp=2)
result = square(5)  # Returns 25
```

# Using *args and **kwargs

- Allow functions to accept a variable number of positional and keyword arguments.
- Useful for flexible function signatures.

## Example

```
def variable_arguments(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)

variable_arguments(1, 2, a=3, b=4)
```

# Type Hinting

- Helps with code readability and static analysis.
- Allows specifying the expected data types of parameters and return values.

## Example

```
def add_numbers(a: int, b: int) -> int:
    return a + b
```

# Docstrings and Documentation

- Provides a way to document functions.
- Accessible through the `__doc__` attribute.

## Example

```
def example_function():
    """This is an example function docstring."""
    pass

print(example_function.__doc__)  # Prints the docstring
```

# Error Handling in Functions

- Use try-except blocks to handle exceptions.
- Enhance robustness and reliability of code.

## Example

```
def divide(x, y):
    try:
        return x / y
    except ZeroDivisionError:
        return "Cannot divide by zero"

result = divide(5, 0)  # Returns "Cannot divide by zero"
```

# Performance Considerations

- Function call overhead and recursion depth can impact performance.
- Optimize by reducing function calls where possible.

# Memoization

- Cache the results of expensive function calls.
- Improve performance by avoiding repeated calculations.

## Example

```
def memoize(func):
    cache = {}
    def memoized_func(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return memoized_func

@memoize
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

result = fibonacci(10)  # Efficiently computes Fibonacci
```

# Function Caching

- Similar to memoization but usually implemented using built-in libraries.
- Use `functools.lru_cache` for easy caching.

## Example

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

result = fibonacci(10)  # Efficiently computes Fibonacci
```

# Functional Programming Concepts

- Map, filter, and reduce functions.
- Promote immutability and stateless functions.

## Example

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

# Using map to square numbers
squared = list(map(lambda x: x**2, numbers))

# Using filter to get even numbers
evens = list(filter(lambda x: x % 2 == 0, numbers))

# Using reduce to sum numbers
from functools import reduce
total = reduce(lambda x, y: x + y, numbers)
```

# Composing Functions

- Combine multiple functions into a single function.
- Useful for creating pipelines of data transformations.

## Example

```
def compose(f, g):
    return lambda x: f(g(x))

def square(x):
    return x * x

def add_one(x):
    return x + 1

composed_function = compose(square, add_one)
result = composed_function(4)  # Returns 25
```

# Chaining Functions

- Allow functions to be chained for cleaner code.
- Improves readability and reduces intermediate variables.

## Example

```
def chain_functions(x):
    return add_one(square(x))

result = chain_functions(4)  # Returns 25
```

# Using `functools` Module

- Provides higher-order functions for functional programming.
- Functions like `reduce`, `partial`, and `lru_cache`.

# The `inspect` Module

- Introspection capabilities for live objects.
- Helps to retrieve function signatures and documentation.

## Example

```
import inspect

def example_func(a, b):
    pass

print(inspect.signature(example_func))  # Prints the signature
```

# Dynamic Function Creation

- Create functions at runtime using the `exec()` function.
- Useful for metaprogramming and dynamic behavior.

## Example

```
def create_function(name):
    exec(f"def {name}(x): return x * 2", globals())

create_function('dynamic_func')
print(dynamic_func(5))  # Returns 10
```

# Using Functions with DataFrames

- Apply custom functions to Pandas DataFrames.
- Leverage vectorized operations for performance.

## Example

```
import pandas as pd

def add_five(x):
    return x + 5

df = pd.DataFrame({'A': [1, 2, 3]})
df['B'] = df['A'].apply(add_five)
```

# Applying Functions to Data Structures

- Use built-in functions like `map()` and `filter()`.
- Promote functional programming paradigms.

## Example

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

# Using map to create a new list of squared numbers
squared_numbers = list(map(lambda x: x**2, numbers))  # [1, 4, 9, 16,

# Using filter to extract even numbers
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))  # [2, 4]
```

# Custom Function Libraries

- Organize functions into reusable libraries.
- Promote code reusability and maintainability.

## Example

```
# my_functions.py
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

# main.py
from my_functions import add, subtract

result_add = add(5, 3)        # Returns 8
result_subtract = subtract(5, 3)  # Returns 2
```

# Testing Functions

- Write unit tests to validate function behavior.
- Use testing frameworks like `unittest` or `pytest`.

## Example

```
import unittest

def multiply(a, b):
    return a * b

class TestMathFunctions(unittest.TestCase):
    def test_multiply(self):
        self.assertEqual(multiply(2, 3), 6)
        self.assertEqual(multiply(-1, 5), -5)

if __name__ == '__main__':
    unittest.main()
```

# Debugging Function Issues

- Use debugging tools like pdb.
- Trace function calls and inspect variables.

## Example

```
def faulty_function(x):
    return x / 0  # Will raise ZeroDivisionError

import pdb

pdb.set_trace()  # Set a breakpoint
faulty_function(5)
```

# Using Jupyter Notebooks for Functions

- Interactive environment for testing and debugging functions.
- Visualize data and function outputs easily.

## Example

```
# In a Jupyter Notebook cell
def plot_data(data):
    import matplotlib.pyplot as plt
    plt.plot(data)
    plt.show()

data = [1, 2, 3, 4, 5]
plot_data(data)  # Displays a line plot of the data
```

# Best Practices in Function Design

- Keep functions small and focused.
- Use descriptive names and docstrings.

## Example

```
def calculate_area(radius: float) -> float:
    """Calculate the area of a circle given its radius."""
    import math
    return math.pi * radius ** 2
```

# Common Pitfalls in Python Functions

- Be aware of mutable default arguments.
- Understand variable scope and closures.

## Example

```
def append_to_list(value, list=[]):
    list.append(value)
    return list

result1 = append_to_list(1)  # Returns [1]
result2 = append_to_list(2)  # Returns [1, 2] (unexpected)
```

# Real-World Examples in Data Science

- Showcase how advanced functions are used in data processing.
- Discuss practical applications in machine learning and analytics.

## Example

```
import pandas as pd

# Load a dataset
df = pd.read_csv('data.csv')

# Define a function for preprocessing
def preprocess_data(df):
    df.fillna(0, inplace=True)  # Fill missing values
    return df

# Apply preprocessing
cleaned_df = preprocess_data(df)
```

# Questions and Discussion