

Lesson 04: Functions

Data Science with Python – BSc Course

Data Science Program

45 Minutes

After this lesson, you will be able to:

- Define and call functions with parameters
- Use return statements to output values
- Understand variable scope (local vs global)
- Write docstrings for documentation

Finance Application: Create reusable functions for return calculations and risk metrics.

Functions are building blocks of modular code

Function Anatomy

Function Anatomy

```
def calculate_return(price_old, price_new):
    """Calculate percentage return."""
    return (price_new - price_old) / price_old * 100
```

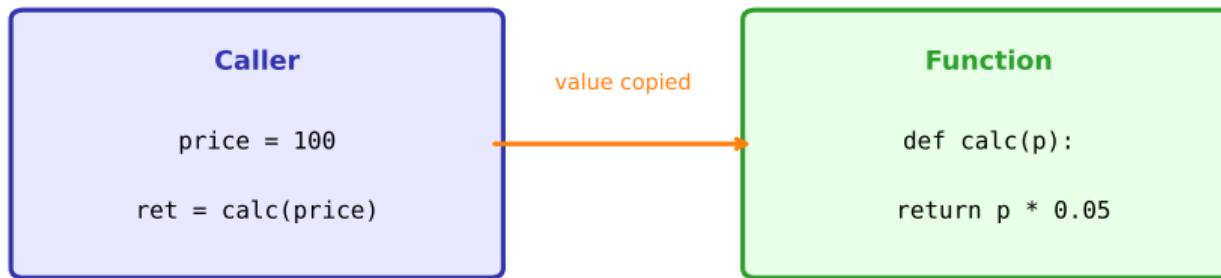
The diagram shows a Python function definition within a blue-bordered box. Annotations with arrows point to specific parts of the code:

- An orange arrow labeled "keyword" points to the `def` keyword.
- A blue arrow labeled "parameters" points to the parameters `price_old` and `price_new`.
- A green arrow labeled "return value" points to the `return` statement.
- A grey arrow labeled "docstring" points to the multi-line string starting with `"""`.

Functions encapsulate reusable logic

def keyword, name, parameters, colon, indented body, return

Parameter Passing



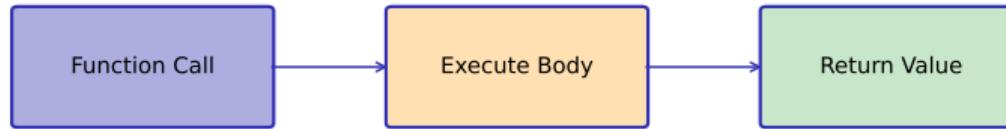
Positional: `func(a, b)`

Keyword: `func(x=1, y=2)`

Default: `def f(x=10)`

`*args, **kwargs`

Return Value Flow



Single return:

```
return price * 1.05
```

Multiple returns:

```
return mean, std
```

No return (None):

```
print("Hello")
```

Early return:

```
if x < 0: return 0
```

Functions without return statement return None

Variable Scope: Local vs Global

Global Scope

```
tax_rate = 0.15
```

Local Scope (inside function)

```
def calc_tax(income):  
    tax = income * tax_rate
```

Local variables exist only during function execution

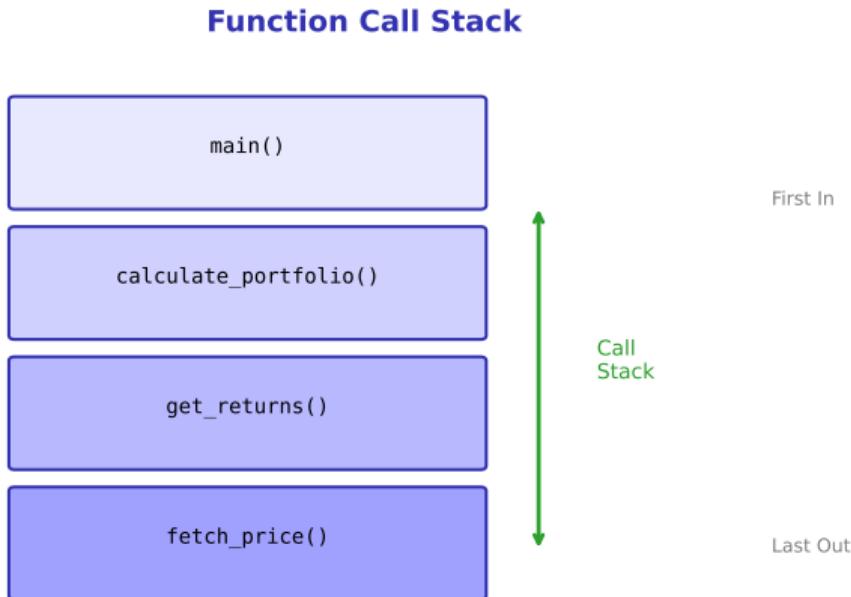
Docstring Best Practices

```
def calculate_sharpe(returns, rf_rate=0.02):
    """
    Calculate the Sharpe ratio for a series of returns.

    Parameters:
        returns (array): Daily return values
        rf_rate (float): Risk-free rate (default: 0.02)

    Returns:
        float: Annualized Sharpe ratio
    """
    excess = returns.mean() - rf_rate/252
    return excess / returns.std() * np.sqrt(252)
```

Function Call Stack



Stack grows with nested calls, shrinks as functions return

Pure vs Impure Functions

Pure Function

Same input -> Same output

No side effects

```
def add(a, b):  
    return a + b
```

Impure Function

Modifies external state

May have side effects

```
def update(lst, x):  
    lst.append(x)
```

Prefer pure functions for predictable, testable code

Essential Finance Functions

```
calculate_return(p1, p2)
```

Price change %

```
annualize_return(daily_ret)
```

Convert to yearly

```
calculate_volatility(returns)
```

Standard deviation

```
sharpe_ratio(ret, rf)
```

Risk-adjusted return

```
max_drawdown(prices)
```

Largest peak-to-trough

```
beta(stock, market)
```

Market sensitivity

Hands-on Exercise (25 min)

Build a finance functions library:

- ① `calculate_return(buy, sell)` – percentage return
- ② `annualize_return(daily_ret, days=252)` – annualized
- ③ `calculate_volatility(returns)` – standard deviation
- ④ `sharpe_ratio(returns, rf=0.02)` – risk-adjusted return
- ⑤ Test each function with sample data
- ⑥ Add docstrings to all functions

These functions will be used throughout the course

Key Takeaways:

- Functions encapsulate reusable logic
- Parameters pass data in, return sends data out
- Local scope: variables exist only inside function
- Docstrings document function purpose and usage
- Pure functions are predictable and testable

Next Lesson: DataFrames Introduction

Functions + pandas = powerful financial analysis