Lesson 04: Functions

Data Science with Python – BSc Course

Data Science Program

BSc Course

45 Minutes

Where We Are

Last time: Control Flow gave your code decision-making power

You can write conditionals and loops. But what happens when you copy the same 10 lines of calculation code into five different cells?

Functions eliminate copy-paste – write once, call anywhere

Learning Objectives

After this lesson, you will be able to:

- **Define** functions with parameters and return values
- **Explain** the difference between print() and return
- **Apply** default parameters and keyword arguments
- **Distinguish** local vs global variable scope
- **Create** a reusable finance functions library

Finance Application: Build reusable Sharpe ratio, return, and volatility calculators that you will use for the rest of the course.

Bloom's levels: define (remember), explain (understand), apply, distinguish (analyze), create

The Copy-Paste Problem

You have written the same calculation five times. There is a better way.

Without functions:

```
# Cell 1: AAPL return
ret_aapl = (sell_aapl - buy_aapl) / buy_aapl
# Cell 2: MSFT return
ret_msft = (sell_msft - buy_msft) / buy_msft
# Cell 3: GOOGL return
ret_googl = (sell_googl - buy_googl) / buy_googl
# ...same formula, repeated
```

With a function:

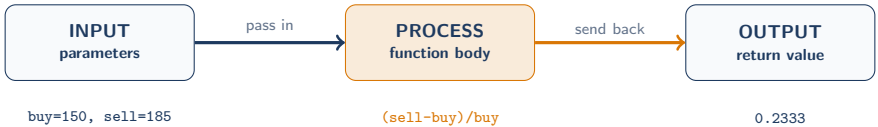
```
def calc_return(buy, sell):
    return (sell - buy) / buy
ret_aapl = calc_return(150, 185)
ret_msft = calc_return(320, 340)
ret_googl = calc_return(125, 142)
# One formula, used everywhere
```

Fix a bug in one place → fixed everywhere. That is the power of functions.

DRY principle: Don't Repeat Yourself

What Is a Function?

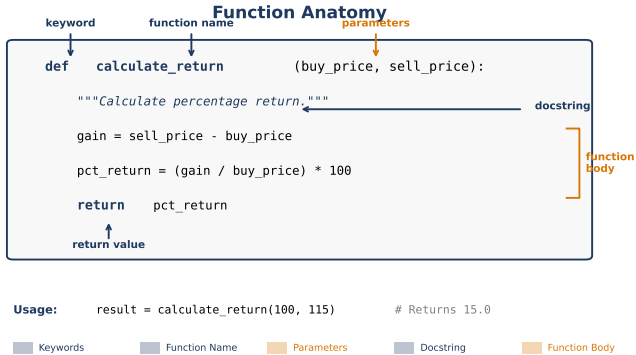
A function is a named, reusable block of code



Three parts: name it, give it inputs (parameters), get outputs (return value).

Functions transform inputs into outputs – like formulas in a spreadsheet

Function Anatomy



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

Defining and Calling Functions

Definition (blueprint):

```
def greet(name):
    """Say hello."""
    print(f"Hello, {name}!")
```

Call (execution):

```
greet("Alice")
# Output:  Hello, Alice!
```

A function does nothing until you **call** it.

With return value:

```
def double(x):
    return x * 2
result = double(5)
# result = 10
```

Naming rules:

- Lowercase with underscores
- Verb-first: `calc_return`
- Descriptive: `get_price`
- Avoid: `f`, `do_stuff`

Convention: function names use snake_case and start with a verb

Parameters vs Arguments

Parameters (in definition):

```
def calc(price, shares):
    return price * shares
```

price and shares are **parameters** – placeholders.

Positional arguments:

```
calc(185.50, 100)
# price=185.50, shares=100
```

Arguments (at call site):

```
calc(185.50, 100)
```

185.50 and 100 are **arguments** – actual values.

Keyword arguments:

```
calc(shares=100, price=185.50)
# Order doesn't matter!
```

Parameters are variables in the definition. Arguments are values in the call.

Keyword arguments make calls self-documenting and order-independent

Return Values

Single return:

```
def calc_return(buy, sell):
    return (sell - buy) / buy
r = calc_return(150, 185)
# r = 0.2333
```

Multiple returns (tuple):

```
def stats(prices):
    return mean(prices), std(prices)
m, s = stats(my_prices)
```

No return (returns None):

```
def show(name):
    print(f"Stock:  {name}")
result = show("AAPL")
# result is None
```

Early return (guard clause):

```
def safe_div(a, b):
    if b == 0:
        return 0 # exit early
    return a / b
```

Functions without return statement return None

Default Parameters

Give parameters sensible defaults so callers can skip them

```
def annualize_return(daily_ret, trading_days=252):
    """Annualize a daily return."""
    return daily_ret * trading_days
# Use default (252 trading days)
annualize_return(0.0004)  # → 0.1008
# Override for crypto (365 days)
annualize_return(0.0004, trading_days=365)  # → 0.146
```

Rules for defaults:

- Defaults must come **after** required parameters
- Never use mutable defaults (`def f(x=[])`) – use `None` instead
- Choose defaults that cover the most common use case

Default parameters reduce boilerplate while keeping flexibility

Checkpoint: Test Your Understanding

Q1: What is the difference between `print()` and `return`?

Q2: What does this return? `def f(x, y=10): return x + y` called as `f(5)`

Q3: Why is `def f(x=[])` dangerous?

Think for 30 seconds before we discuss. Answers: Q1: print displays, return sends value back to caller. Q2: 15. Q3: The list is shared across calls.

If you got all three, you understand function basics

Flexible Arguments: *args and **kwargs

***args – Variable positional:**

```
def portfolio_value(*prices):
    """Sum any number of prices."""
    return sum(prices)

portfolio_value(150, 340, 125)
# → 615
```

*args collects extra positional arguments into a tuple.

****kwargs – Variable keyword:**

```
def stock_info(**data):
    for key, val in data.items():
        print(f"{key}:  {val}")

stock_info(ticker="AAPL",
           price=185.50)
```

**kwargs collects keyword arguments into a dict.

Use sparingly. Explicit parameters are clearer for most functions.

---

*args = tuple of extras; **kwargs = dict of extras

Lambda Functions

One-line anonymous functions for simple transformations

Regular function:

```
def pct(x):
    return x * 100
```

Lambda equivalent:

```
pct = lambda x:  x * 100
```

Common use – sorting:

```
stocks = [("AAPL", 185), ("MSFT", 340)]
sorted(stocks, key=lambda s:  s[1])
# Sorts by price
```

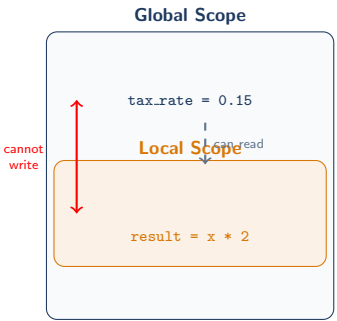
With pandas later:

```
df["Pct"] = df["Return"].apply(
    lambda x:  x * 100)
```

Rule of thumb: if you need more than one line, use `def`.

Lambda: quick throwaway functions. def: anything reusable.

Variable Scope: Local vs Global



Local variables:

```
def process(x):
    result = x * 2  # Local
    return result

# result not accessible here!
```

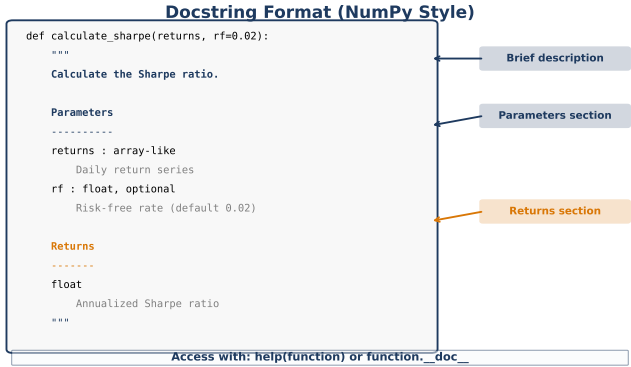
Global variables:

```
tax_rate = 0.15  # Global
def calc_tax(income):
    return income * tax_rate
    # Can READ global
```

Rule: Avoid `global` keyword. Pass values as parameters instead.

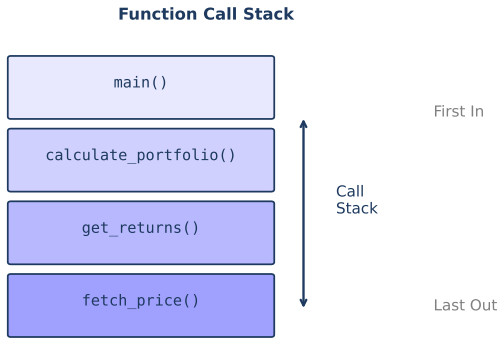
Local variables are created when function starts and destroyed when it ends

Docstrings: Documenting Your Functions



Access docstring with help(function) or function.__doc__

Function Call Stack



Stack grows with nested calls, shrinks as functions return

Finance: Sharpe Ratio Function

The Sharpe ratio measures risk-adjusted return: Sharpe $= \frac{\bar{R} - R_f}{\sigma_R}$

```
def sharpe_ratio(returns, rf_rate=0.02):
    """Calculate annualized Sharpe ratio."""
    import numpy as np
    excess = np.mean(returns) - rf_rate / 252
    vol = np.std(returns)
    return (excess / vol) * np.sqrt(252)
```

`sharpe_ratio(my_returns)` – uses 2% risk-free rate
`sharpe_ratio(my_returns, rf_rate=0.05)` – overrides to 5%

Sharpe ¿ 1 is good, ¿ 2 is excellent, ¿ 3 is outstanding

Finance: Portfolio Return Calculator

Combine functions to build a portfolio analyzer

```
def portfolio_return(weights, returns):
    """Weighted average return of a portfolio."""
    total = 0
    for w, r in zip(weights, returns):
        total += w * r
    return total
# 60% AAPL, 40% MSFT
weights = [0.6, 0.4]
returns = [0.12, 0.08]  # annual returns
port_ret = portfolio_return(weights, returns)
# → 0.104 (10.4%)
```

Key insight: Each function does one job. Compose them for complex analysis.

Small, focused functions are easier to test and debug

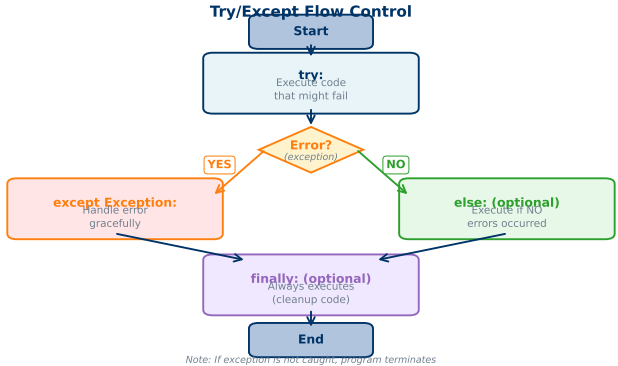
Finance Functions Library

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

Build your own library of financial calculation functions

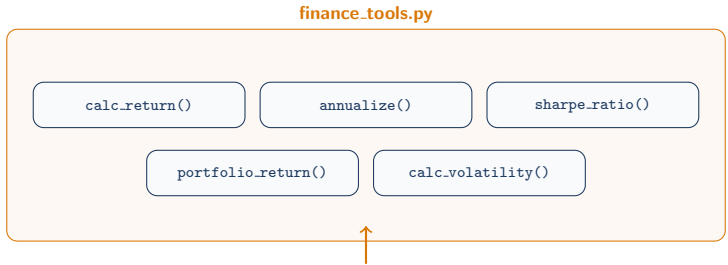
Error Handling with try/except



Defensive programming: handle errors gracefully instead of crashing

Building a Function Library

Organize related functions into a reusable module



Workflow: Write functions → save as `.py` file → import anywhere.

Reusable modules save hours of work across projects

Hands-on Exercise (25 min)

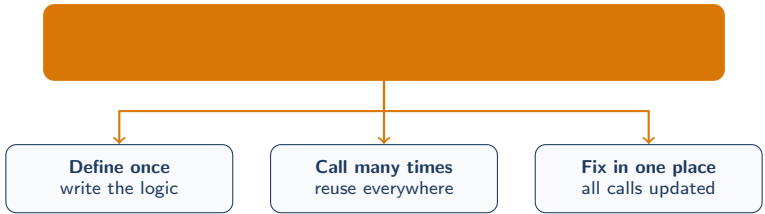
Build a finance functions library:

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

Bonus: Add try/except to handle edge cases (division by zero, empty lists).

These functions will be used throughout the course

The Big Idea



Functions are the foundation of every professional codebase.

Functions = reusability + readability + testability

Lesson Summary

Key Takeaways:

- `def` creates a function; `return` sends a value back
- Parameters are placeholders; arguments are actual values
- Default parameters make common calls shorter
- Local variables exist only inside the function
- Lambda for one-liners; `def` for everything else

Next Lesson: L05 – DataFrames Introduction
The most important data structure in data science. Functions + pandas = powerful financial analysis.

Functions + pandas = powerful financial analysis