

# Lecture 01 - Introduction to Python



# 1. Motivation



# What is Python?

Python is a high-level, interpreted programming language known for its simplicity and readability.

## Key Features of Python

- **Easy to Learn:** Python's syntax is straightforward.
- **Interpreted Language:** Python code is executed line by line, which makes debugging easier.
- **Dynamically Typed:** You don't need to declare variable types explicitly; Python handles it automatically.
- **Versatile:** Python is used in web development, data analysis, automation, and much more.
- **Huge Ecosystem:** Python has a large standard library and third-party modules for a wide variety of applications.



# Why Python for Finance?

- **Data Handling:** In finance, you often work with large datasets—Python's libraries like `pandas` and `NumPy` are designed to handle and analyze financial data efficiently.
- **Automation:** Python can automate repetitive tasks like data retrieval, report generation, and portfolio analysis.
- **Financial Modeling:** Python is a great tool for building complex models such as forecasting, risk management, and pricing.
- **Integration with Data Science:** Python is the most popular language for data science, offering extensive support for statistical analysis, machine learning, and data visualization.



## 2. Setting up the environment

## 2.1 Installing a Python environment



# Python

- Download Python from the [official website](#).
- Installation includes the Python interpreter and Integrated Development Environment (IDLE) for coding.



## Installing Anaconda

- Anaconda is a suite of useful tools and packages for Python development.
- Download Anaconda from the [official website](#)
- Once it is installed, confirm the following environments and packages are available:
  - [Spyder](#)
  - [Jupyter Notebook](#)





## 2.2 Running Python Code



There are multiple ways to run Python.

Consider the following code line which instructs to simply print out "Hello World!"

```
print ("Hello World!")
```



## Python Shell

- Use the Python IDLE (from terminal or from any IDE setting like Spyder)
- Type the code and press `Enter`
- Check the output



## Python Script

- Open an empty file (Spyder, [Sublime Text Editor](#), etc.)
- Write the code
- Save file as `helloworld.py`
- Run the script
  - From the terminal, run the script by typing  
`python helloworld.py`
  - From the IDE (like Spyder), launch the run
- Check the output



# Notebook

*This is a Notebook*

Platforms like [Jupyter Notebooks](#) are widely used in data science for documenting and running code interactively.

- Open Jupyter Notebook by typing in the terminal

## Jupyter Notebook

- Create a new notebook with a Python environment
- Write the code in the first cell
- Run the cell
- Check the output (see below)

```
In [ ]: print ('Hello World!')
```



# 3. Overview of the Python environment

1. Syntax
2. Control structures
3. Functions
4. Data structures
5. Libraries



## 3.1 Syntax





### 3.1.1 Python as a calculator

```
In [ ]: 5000 + 250
```

```
In [ ]: 10000 * 1.05
```

```
In [ ]: 10000 / 2
```

**Comments:** Use comments ( `#` ) to explain code, particularly useful for documenting underlying logic.

```
In [ ]: # Basic financial arithmetic  
print(5000 + 250)    # Adding investment returns  
print(10000 * 1.05)  # Calculating interest (5% growth)  
print(10000 / 2)     # Splitting an investment
```

```
In [ ]: # This is a comment  
print("Welcome to Python for Finance!")  # This prints a message
```

## 3.1.2 Variables and Data Types



Variables

### **Variables are store data for calculations**

The operation `=` assigns a value to a variable.

```
In [ ]: # Variable assignment in a financial scenario
stock_price = 150.25 # Price of a stock
investment_amount = 10000 # Amount invested
shares = investment_amount / stock_price # Number of shares
```

```
In [ ]: shares
```



Data types

**Variables can be of different types.**

- **Strings** ( `str` ): Text
- **Integers** ( `int` ): Integer value
- **Floats** ( `float` ): Real value
- **Booleans** ( `bool` ): True or False

Because Python is **dynamically typed**, there is no need to explicitly mention the type of the variable. Yet, in some cases, it may be important to **cast variables** from one type to another.

*More on this in the next lecture.*



## 3.2 Control structures



**Control structures** allow to condition the sequence of action of a code on the particular value a variable exhibits at the time of execution.

- Conditional statements
- Loops

**Note:** `tabs` are organizational pillars of the Python code structure



## 3.2.1 Conditional statements

**Conditional statements** consider the specific value of a variable at the time of execution and determine the outcome based on a logical operation.

### Structure

```
If CONDITON HOLDS:  
    OUTCOME 1  
Elif OTHER CONDITION HOLDS:  
    OUTCOME 2  
Else:  
    OUTCOME 3
```

**Note:** check the `tabs`





```
In [ ]: balance = 5000
        if balance >= 10000:
            print("You are eligible for premium services.")
        else:
            print("Standard services apply.")
```

## 3.2.2 Loops

**Loops** repeat a sequence of actions until a condition is satisfied. There are two types of loops:

- `while`
- `for`



While

### Structure

```
While CONDITION HOLDS:  
    ACTION(s)
```



```
In [ ]: # Use case: Simulating monthly deposit growth  
balance = 1000  
months = 0  
while balance < 2000:  
    balance += 100 # Monthly deposit  
    months += 1  
print(f"It took {months} months to double the balance.")
```

For

### Structure

```
For CONDITION HOLDS | Increment action:  
    ACTION(s)
```



```
In [ ]: # Use case: Summing up daily returns from a list  
daily_returns = [0.01, -0.02, 0.03, 0.02, -0.01]  
total_return = 0  
for r in daily_returns:  
    total_return += r  
print("Total return for the week:", total_return)
```

## 3.3 Functions

A **function** is a reusable block of code that is saved up and can be called at multiple places in the main script.

### Structure

```
def my_function (parameters):  
    ACTION(s)  
    return VALUE
```



```
In [ ]: # Function to calculate compound interest  
def calculate_compound_interest(principal, rate, time):  
    return principal * (1 + rate) ** time
```

```
In [ ]: # Example usage  
result = calculate_compound_interest(1000, 0.05, 5)  
print("Compound Interest:", result)
```



## 3.4 Data structures



## 3.4.1 Lists

**Lists** allow to store and treat multiple data points into one variable

```
In [ ]: # Example: List of daily stock prices  
stock_prices = [150.25, 153.50, 152.00, 155.00]  
print(stock_prices[0]) # Accessing the first day's price  
stock_prices.append(157.25) # Adding a new day's price  
print(stock_prices)
```



## 3.4.2 Dictionaries

**Dictionaries** allow store and treat multiple pairs of data point associating keys and values.

```
In [ ]: # Example: Dictionary to store portfolio allocation
portfolio = {
    "AAPL": 5000,
    "GOOGL": 3000,
    "AMZN": 2000
}
print(portfolio["AAPL"]) # Accessing allocation for AAPL
portfolio["GOOGL"] += 1000 # Updating allocation for GOOGL
print(portfolio)
```



## 3.5 Libraries

**Libraries** are pre-built packages of functions for tasks like data analysis and visualization.



### Key Libraries for Finance

- **NumPy** : For numerical computations matrix operations in portfolio analysis
- **pandas** : Used for data manipulation handling financial datasets
- **matplotlib** : For data visualization plotting stock prices



```
In [ ]: import numpy

print(numpy.sqrt(16)) # Square root
print(numpy.pi)      # Value of pi
```

## Documentation

Libraries come with **documentation**.

On Notebooks, they can be directly accessed from the cell pressing `maj` + `tab` after the function.



Example of documentation `numpy.sqrt()`

```
Call signature:  numpy.sqrt(*args, **kwargs)
Type:           ufunc
String form:     <ufunc 'sqrt'>
File:           ~/opt/anaconda3/lib/python3.9/site-
packages/numpy/__init__.py
Docstring:
sqrt(x, /, out=None, *, where=True, casting='same_kind', order='K',
dtype=None, subok=True[, signature, extobj])
```

Return the non-negative square-root of an array, element-wise.

#### Parameters

-----

`x` : array\_like

The values whose square-roots are required.

`out` : ndarray, None, or tuple of ndarray and None, optional

A location into which the result is stored. If provided, it must have

a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as

a

keyword argument) must have length equal to the number of outputs.

`where` : array\_like, optional

This condition is broadcast over the input. At locations where the

condition is True, the ``out`` array will be set to the ufunc result.





# 4. Organisation

The rest of the class is organized as follows:

Lecture	Topic	Content
Lecture 02	Data types and structures	int, float, list
Lecture 03	Control structures	if, for
Lecture 04	Functions	def
Lecture 05	OO Programming	class
Lecture 06	Libraries	
Lecture 07	Numerical computing	numpy
Lecture 08	Data manipulation	pandas
Lecture 09	Input & output	read, write
Lecture 10	Data visualization	matplotlib
Lecture 11	Time series	
Lecture 12	Network analysis	networkx
Lecture 13&14	Machine learning	scikit-learn

In [ ]:

