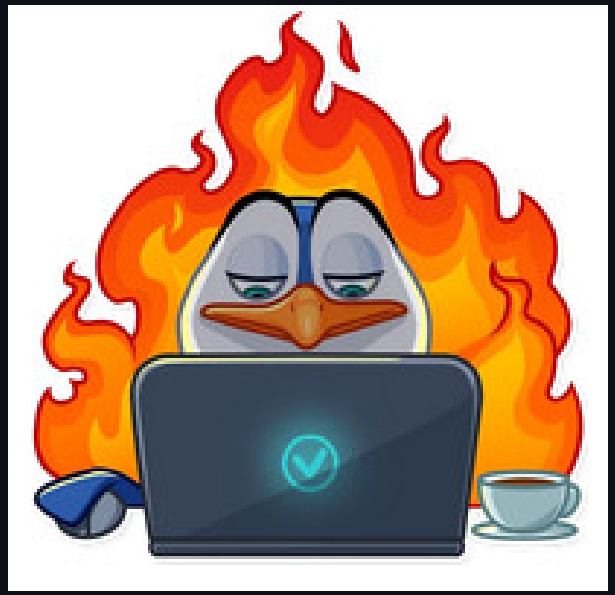
Day 1, Session 1: Python Basics for Data Analysis

Three-Day Data Analysis with Python Course

about: Alberto Cámara

- Mathematician, Data Freelancer
- Experienced Data Team leader
- Part-time teaching at **Esade** and **UB**
- Organiser at **Python Barcelona**
- I like hiking in mountains and growing vegetables



Course Schedule

Each day follows the same structure:

Time	Activity	Duration
Session 1	Theory + Live Coding	2 hours
☕ **Coffee Break**		15-30 min
Session 2	Theory + Live Coding	2 hours
🍽️ **Lunch Break**		1 hour
Practical Session	Hands-on Exercises	2 hours

Total: 6 hours of instruction + 3 hours of breaks per day

Know Your Audience

Before we start, I'd like to learn about YOU! 🙋

Let's do a quick poll:

1. **Previous experience with Python?** (None / Beginner / Intermediate / Advanced)
2. **Data analysis tools you've used?** (Excel / SQL / R / Tableau / Power BI / Other)
3. **Programming experience?** (None / Some scripting / Professional developer)
4. **Current tech stack at work?** (What tools do you use daily?)
5. **What do you hope to learn from this course?**

Goal of the course

What would you say is the goal of your job?

- The role of a **Data Analyst** in an organization is to provide **data-backed** evidence to support **decision-making**
- Our goal: to get a basic command of Python and some of its useful libraries in order to fulfill this role

Course Tools & Setup

We'll be using marimo notebooks for all hands-on work:

- Interactive Python notebooks
- Reactive execution (cells update automatically)
- Works right in your browser

👉 **Please log in now: molab.marimo.io**

All course materials and notebook links will be shared via:

- **Zoom chat** during sessions
- **Course repository** (link in chat)

Overview of session 1

1. **Getting Started with Python**
2. **Operators and Expressions**
3. **Control Flow**
4. **Functions Basics**

Part 1: Getting Started with Python

What is Python?

- **High-level**, **interpreted** programming language
- Designed for readability and simplicity
- Extremely popular for data analysis
- Rich ecosystem of libraries (pandas, numpy, plotly, etc.)

The Python Interpreter

Python code is executed line by line by the interpreter.

You can run Python in several ways:

- In the terminal (REPL)
- By asking the interpreter to run a script file (.py)
- **Interactive Notebooks**

We'll use marimo notebooks for hands-on practice

Variables and Assignment

Variables are containers for storing data values.

Key concepts:

- No need to declare type beforehand
- Use `=` for assignment
- Variable names should be descriptive
- Follow naming conventions (lowercase, underscores for spaces)

Syntax:

```
variable_name = value
```

Variables and Assignment (Example)

```
# Creating variables
sales_amount = 1500
product_name = "Laptop"
is_in_stock = True
discount_rate = 0.15

print(f"Product: {product_name}")
print(f"Sales: ${sales_amount}")
print(f"In stock: {is_in_stock}")
print(f"Discount: {discount_rate * 100}%")
```

Basic Data Types

Python has several fundamental data types:

Type	Example	Usage
`int`	`42`	Whole numbers
`float`	`3.14`	Decimal numbers
`str`	`"Hello"`	Text
`bool`	`True`, `False`	Logical values

You can check a variable's type with `type()`:

```
type(variable_name)
```

Basic Data Types (Example)

```
revenue = 50000
growth_rate = 0.23
company_name = "DataCorp"
profitable = True

print(f"revenue is type: {type(revenue)}")
print(f"growth_rate is type: {type(growth_rate)}")
print(f"company_name is type: {type(company_name)}")
print(f"profitable is type: {type(profitable)}")
```

Comments and Code Readability

Comments explain your code to others (and your future self!):

- Single-line comments use `#`
- Multi-line comments use triple quotes `'''` or `"""`
- Good code is self-documenting, but comments clarify **why**

Example:

```
# This is a single-line comment
x = 5  # Comments can also go after code

'''
This is a multi-line comment.
It can span multiple lines.
'''
```

Part 2: Operators and Expressions

Arithmetic Operators

Operator	Operation	Example
`+`	Addition	`5 + 3 = 8`
`-`	Subtraction	`5 - 3 = 2`
`*`	Multiplication	`5 * 3 = 15`
`/`	Division	`5 / 2 = 2.5`
`//`	Floor division	`5 // 2 = 2`
`%`	Modulo (remainder)	`5 % 2 = 1`
`**`	Exponentiation	`5 ** 2 = 25`

Arithmetic Operators (Example)

```
# Data analysis example: calculating metrics
total_sales = 45000
num_transactions = 150

average_transaction = total_sales / num_transactions
print(f"Average transaction value: ${average_transaction:.2f}")

# Growth calculation
previous_sales = 40000
growth = ((total_sales - previous_sales) / previous_sales) * 100
print(f"Sales growth: {growth:.1f}%")
```

Comparison Operators

Comparison of values yields boolean results:

Operator	Meaning	Example
`==`	Equal to	`5 == 5` → `True`
`!=`	Not equal to	`5 != 3` → `True`
`>`	Greater than	`5 > 3` → `True`
`<`	Less than	`5 < 3` → `False`
`>=`	Greater or equal	`5 >= 5` → `True`
`<=`	Less or equal	`5 <= 3` → `False`

Comparison Operators (Example)

```
# Business logic examples
target_revenue = 50000
actual_revenue = 48000

met_target = actual_revenue >= target_revenue
print(f"Met revenue target: {met_target}")

customer_age = 25
is_adult = customer_age >= 18
print(f"Customer is adult: {is_adult}")
```

Logical Operators

Combine multiple conditions:

- `and` - Both conditions must be True
- `or` - At least one condition must be True
- `not` - Inverts the boolean value

Example:

```
(age >= 18) and (has_id == True)
(is_member or purchase > 100)
not is_closed
```

Logical Operators (Example)

```
# Example: Customer segmentation
age = 28
premium_member = True
purchase_amount = 150

# Complex condition
eligible_for_bonus = (age >= 18) and (premium_member or purchase_amount > 100)
print(f"Eligible for bonus: {eligible_for_bonus}")

# Using not
is_minor = not (age >= 18)
print(f"Is minor: {is_minor}")
```

String Operations

Strings are sequences of characters, and we can manipulate them:

- **Concatenation**: `"Hello" + " " + "World"`
- **Repetition**: `"=" * 10`
- **F-strings** (formatted strings): `f"Hello {name}"`
- **Methods**: `.upper()`, `.lower()`, `.strip()`, etc.

String Operations (Example)

```python
first_name = "Alice"
last_name = "Johnson"

# Concatenation
full_name = first_name + " " + last_name
print(f"Full name: {full_name}")

# Repetition
separator = "-" * 20
print(separator)

# String formatting (f-strings)
customer_id = 12345
message = f"Welcome, {full_name}! Your ID is {customer_id}"
print(message)

# Useful string methods
print(f"Uppercase: {full_name.upper()}")
print(f"Lowercase: {full_name.lower()}")
print(f"Length: {len(full_name)} characters")
```

Part 3: Control Flow

Control Flow

Control flow determines the order in which code executes.

Essential for:

- Making decisions based on data
- Processing multiple records
- Implementing business logic

If/Else Statements

Execute code conditionally based on boolean expressions:

```
if condition:
    # code if condition is True
elif another_condition:
    # code if another_condition is True
else:
    # code if all conditions are False
```

If/Else Statements (Example)

```
# Example: Sales performance evaluation
monthly_sales = 75000

if monthly_sales >= 100000:
    performance = "Excellent"
    bonus = monthly_sales * 0.10
elif monthly_sales >= 75000:
    performance = "Good"
    bonus = monthly_sales * 0.05
elif monthly_sales >= 50000:
    performance = "Fair"
    bonus = monthly_sales * 0.02
else:
    performance = "Needs Improvement"
    bonus = 0

print(f"Performance: {performance}")
print(f"Bonus: ${bonus:,.2f}")
```

For Loops

Iterate over sequences (ranges, lists, strings, etc.):

```
for item in sequence:
    # code to execute for each item
```

Most common pattern in data analysis!

For Loops (Example 1)

```
# Example 1: Iterating over a range
print("Daily sales totals:")
for day in range(1, 8):  # Days 1 through 7
    print(f"  Day {day}: Processing...")
```

For Loops (Example 2)

```
# Example 2: Processing a list of values
daily_sales = [1200, 1450, 1100, 1800, 2100, 950, 1350]

total = 0
for sale in daily_sales:
    total += sale  # Same as: total = total + sale

average = total / len(daily_sales)
print(f"Total weekly sales: ${total:,.2f}")
print(f"Average daily sales: ${average:,.2f}")
```

Data Processing with For Loops

Real-world scenario: Processing customer data

```
# Customer purchase amounts
purchases = [45.99, 120.50, 89.99, 200.00, 15.75]

# Apply discount to purchases over $100
discounted_purchases = []

for purchase in purchases:
    if purchase > 100:
        discounted_price = purchase * 0.90  # 10% discount
        discounted_purchases.append(discounted_price)
        print(f"${purchase:.2f} → ${discounted_price:.2f} (10% off)")
    else:
        discounted_purchases.append(purchase)
        print(f"${purchase:.2f} (no discount)")
```

For Loops (Continued)

```
print(f"\nOriginal total: ${sum(purchases):.2f}")
print(f"Discounted total: ${sum(discounted_purchases):.2f}")
```

While Loops

Execute code while a condition remains True:

```
while condition:
    # code to execute
    # (make sure to eventually make condition False!)
```

Use when you don't know how many iterations you need in advance.

While Loops (Example)

```
# Example: Compound interest calculation
initial_investment = 10000
target_amount = 15000
annual_rate = 0.07

current_amount = initial_investment
years = 0

while current_amount < target_amount:
    current_amount = current_amount * (1 + annual_rate)
    years += 1
    print(f"Year {years}: ${current_amount:,.2f}")

print(f"\nReached target in {years} years!")
```

Break and Continue

Control loop execution:

- `break` - Exit the loop immediately
- `continue` - Skip to the next iteration

Break Example

```
# Example with break: Find first high-value transaction
transactions = [45, 120, 67, 89, 500, 234, 156]
threshold = 400

print(f"Looking for transaction over ${threshold}...")
for i, transaction in enumerate(transactions, 1):
    if transaction > threshold:
        print(f"Found at position {i}: ${transaction}")
        break
    print(f"  Transaction {i}: ${transaction} (not high enough)")
```

Continue Example

```
# Example with continue: Process only valid entries
data_entries = [100, -50, 200, 0, 150, -20, 300]

print("Processing valid (positive) entries only:")
valid_sum = 0

for entry in data_entries:
    if entry <= 0:
        print(f"  Skipping invalid entry: {entry}")
        continue

    valid_sum += entry
    print(f"  Added: {entry}")

print(f"\nSum of valid entries: {valid_sum}")
```

Part 4: Functions Basics

Functions

Functions are reusable blocks of code that perform specific tasks.

Benefits:

- Avoid repetition (DRY: Don't Repeat Yourself)
- Make code more organized and readable
- Easier to test and debug

Defining Functions

```
def function_name(parameters):
    # code to execute
    return result
```

Key components:

- `def` keyword
- Function name (descriptive, lowercase with underscores)
- Parameters (optional inputs)
- Return value (optional output)

Simple Function Example

```
# Simple function with no parameters
def greet():
    return "Hello, Data Analyst!"

message = greet()
print(message)
```

Functions with Parameters

Parameters allow functions to accept inputs:

```
def calculate_discount(price, discount_rate):
    """Calculate the discounted price."""
    discount_amount = price * discount_rate
    final_price = price - discount_amount
    return final_price

# Using the function
original = 100
result = calculate_discount(original, 0.20)
print(f"Original: ${original}")
print(f"After 20% discount: ${result}")
```

Functions with Parameters (Continued)

```
# Call with different values
result2 = calculate_discount(250, 0.15)
print(f"Original: $250")
print(f"After 15% discount: ${result2}")
```

Return Values

Functions can return multiple values:

```
def calculate_metrics(sales_list):
    """Calculate total, average, and maximum from a list of sales."""
    total_sales = sum(sales_list)
    avg_sales = total_sales / len(sales_list)
    max_sales = max(sales_list)

    # Return multiple values as a tuple
    return total_sales, avg_sales, max_sales
```

Return Values (Continued)

```
# Using the function
weekly_sales = [1200, 1450, 1100, 1800, 2100, 950, 1350]
total, average, maximum = calculate_metrics(weekly_sales)

print(f"Total: ${total:,.2f}")
print(f"Average: ${average:,.2f}")
print(f"Maximum: ${maximum:,.2f}")
```

Default Parameters

You can provide default values for parameters:

```
def apply_tax(amount, tax_rate=0.08):
    """Apply tax to an amount. Default tax rate is 8%."""
    return amount * (1 + tax_rate)

# Use default tax rate
price1 = apply_tax(100)
print(f"With default tax (8%): ${price1:.2f}")

# Override with custom tax rate
price2 = apply_tax(100, 0.10)
print(f"With custom tax (10%): ${price2:.2f}")
```

Scope: Local vs Global Variables

- **Local variables**: Created inside a function, only accessible within that function
- **Global variables**: Created outside functions, accessible everywhere

Best practice: Minimize use of global variables, pass data through parameters instead

Scope Example

```
# Global variable
company_name = "DataCorp"

def generate_report(revenue):
    # Local variable
    report_title = f"{company_name} Revenue Report"
    summary = f"{report_title}: ${revenue:,.2f}"
    return summary

report = generate_report(50000)
print(report)

# This would cause an error (report_title is not accessible here):
# print(report_title)
```

Practical Example: Data Validation Function

Complete example combining control flow, loops, and functions:

```
def validate_and_clean_sales(sales_data):
    """
    Validate and clean a list of sales values.

    - Remove negative values (data errors)
    - Remove zeros (cancelled transactions)
    - Return cleaned list and count of removed items
    """
    cleaned_data = []
    removed_count = 0

    for value in sales_data:
        if value > 0:
            cleaned_data.append(value)
        else:
            removed_count += 1
```

Practical Example (Continued)

```
# Test the function
raw_data = [100, 250, -50, 0, 180, 320, 0, -10, 290]
clean_data, errors = validate_and_clean_sales(raw_data)

print(f"Original data: {raw_data}")
print(f"Cleaned data: {clean_data}")
print(f"Removed {errors} invalid entries")
print(f"Clean total: ${sum(clean_data):,.2f}")
```

Session 1 Summary

- ✓ **Python Basics**: Variables, data types, comments
- ✓ **Operators**: Arithmetic, comparison, logical, string operations
- ✓ **Control Flow**: if/else, for loops, while loops, break/continue
- ✓ **Functions**: Definition, parameters, return values, scope

Questions?

Break time! ☕

See you in Session 2!