Introduction to Python

Workshop 1
Thomas Zilliox

Agenda

- What is python?
- Pythonic style — PEP 8
- Variables, data types and data structures
- Control structures (if, for, while, comprehension syntax)
- String and file manipulation
- Functions and modular programming
- Error handling and debugging techniques
- Introduction to virtual environments and packages

What is Python ?

Python is a high-level, interpreted programming language known for its clear syntax and readability. It is open-source and multi OS.

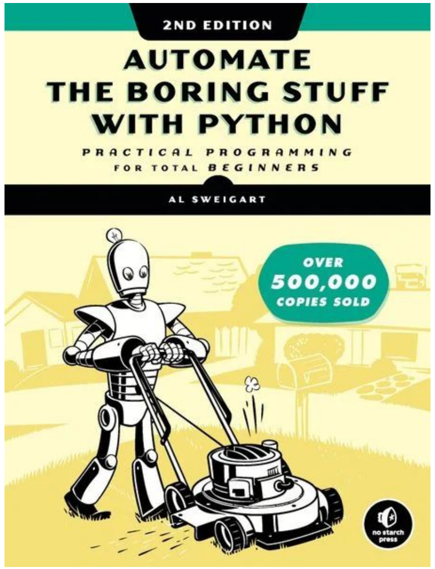
To use it locally you will need :

- A Python version (to make it work)
- An IDE (to write script), I would suggest PyCharm or Visual Studio Code.

For this workshop we will use Google Colab and this GitHub repository.

Why using Python ?

- Free
- Simple
- Large community
- You can do a lot!
  - Data analysis
  - Deep Learning and Machine Learning
  - Website (Google, Youtube, Instagram, Reddit)
  - Automate many things


2ND EDITION
AUTOMATE THE BORING STUFF WITH PYTHON
PRACTICAL PROGRAMMING FOR TOTAL BEGINNERS
AL SWEIGART
OVER 500,000 COPIES SOLD
no starch press

Agenda

- What is python?
- Pythonic style — PEP 8
- Variables, data types and data structures
- Control structures (if, for, while, comprehension syntax)
- String and file manipulation
- Functions and modular programming
- Error handling and debugging techniques
- Introduction to virtual environments and packages

Pythonic style - PEP 8

PEP 8 is the guide style for Python: https://peps.python.org/pep-0008/

Summary:

- Consistency is key. If a project does not follow PEP8, follow its guideline
- Make your code clean and readable: put comments, comprehensive naming.
- Naming convention:
  - Understandable variable/function names
  - For variables and functions: lower_case_with_underscores
  - For constants: UPPER_CASE_WITH_UNDERSCORE
  - For classes: CapWords

Agenda

- What is python?
- Pythonic style — PEP 8
- Variables, data types and data structures
- Control structures (if, for, while, comprehension syntax)
- String and file manipulation
- Functions and modular programming
- Error handling and debugging techniques
- Introduction to virtual environments and packages

Variable and data types

A variable in Python is a named reference used to store data in memory.

It can be either a data type:

- Boolean (True or False)
- Number (Integer or Float)
- Text (String - Chain of characters)

Or a data structure

- Dictionary, List, Set or Tuple

Variable and data types - Booleans

A boolean is a built-in data type that represents one of two possible values: True or False.

```
flag = True
other_flag = False
```

Variable and data types - Numbers

A Python integer is a whole number without a decimal point, able to represent both positive and negative values with unlimited length. A Python float is a real number that contains a decimal point or is written in exponential notation (like 3.14, -0.001, or 2.5e2).

```
# Integer (int)
x = 42

# Floating point (float)
pi = 3.14

# Complex numbers
z = 2 + 3j
```

Variable and data types - Numbers

```
## Common operation
x = 3
y = 2

# Addition
print("Addition:", x + y)

# Subtraction
print("Subtraction:", x - y)

# Multiplication
print("Multiplication:", x * y)
```

```
# Division
print("Division:", x / y)

# Floor Division
print("Floor Division:", x // y)

# Modulus (remainder)
print("Modulus:", x % y)

# Exponentiation (x raised to the
power of y)
print("Exponentiation:", x ** y)
```

Variable and data types - Text

A Python string is a built-in data type that represents a sequence of characters enclosed in single quotes (' '), double quotes (" "), or triple quotes (''' ''' or """ """). Strings are immutable, meaning once created, their contents cannot be changed; operations on strings always produce new string objects.

```python
text = "Hello world"

# Convert to uppercase
print("Uppercase:", text.upper())

# Convert to lowercase
print("Lowercase:", text.lower())

# Split the text into a list of words
print("Split:", text.split())
```

Variable and data structures - List

A Python list is an ordered, mutable collection that can store elements of any type (e.g., numbers, strings, objects).
Mutable -> Items can be changed, added, or removed after the list is created.

```
my_list = [1, "apple", 3.14, True]

a = [1,2,3,4]
print("Last element: ", a[-1])
print("Elements from index 1 to 3: ", a[1:4])

# Add an element at the end
a.append(5)

# Update an element
a[1] = 100

# Removes the first occurrence of an element.
a.remove(100)
```

Variable and data structures - Dictionary

A Python dictionary is a built-in data structure that stores data in key-value pairs.

```
students = {"Alice": 20, "Bob": 21, "Clara": 19}

print("Age of Alice: ", students["Alice"])

# Add or update
students["Dan"] = 22          # add new
students["Alice"] = 21        # update existing

# Remove element
age = students.pop("Bob")    # removes "Bob" and returns his age

# Show all names or ages
print("Names:", list(students.keys()))
print("Ages:", list(students.values()))
print("Pairs:", list(students.items()))
```

Variable and data structures - Set

A set is an unordered collection of unique elements.

```
students = {"Alice", "Bob", "Cara"}

# Add one element
students.add("Dan")

# Remove an element
students.remove("Bob")   # Removes Bob (error if not found)

# Safer removal (no error if missing)
students.discard("Zoe")
```

Variable and data structures - Tuple

A tuple is an ordered, immutable collection. It’s like a list, but you can’t change it after creation.
Immutable -> “cannot be changed after creation.”

```python
temperatures = (18, 20, 22, 19, 21)

# Access elements
print("First temperature:", temperatures[0])
print("Last temperature:", temperatures[-1])
print("Middle three:", temperatures[1:4])

# Count and find position
print("Number of times 21°C appears:", temperatures.count(21))
print("Index of 22°C:", temperatures.index(22))

# Concatenate tuples
next_week = (23, 24, 22, 21, 20)
combined = temperatures + next_week
```

Variable and data types - Summary

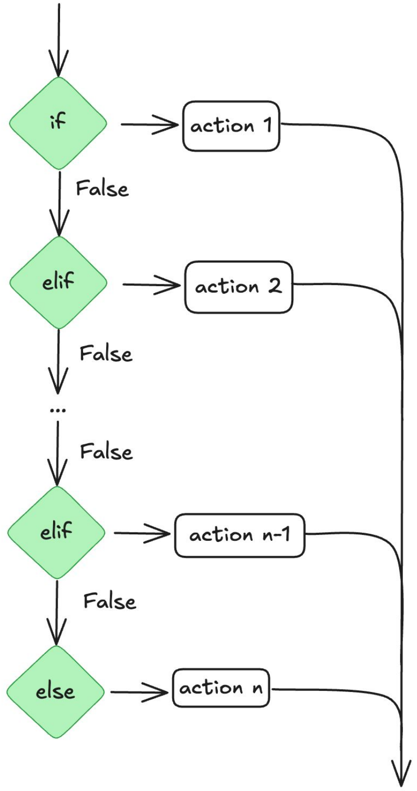
Feature	List	Dictionary	Set	Tuple
Definition	Ordered collection of elements	Collection of key → value pairs	Unordered collection of unique elements	Ordered, unchangeable collection
Syntax	[ ]	{key: value}	{ } or set()	( )
Example	[1, 2, 3]	{"a": 1, "b": 2}	{1, 2, 3}	(1, 2, 3)
Order preserved?	Yes ✅	Yes ✅ (Python 3.7+)	No ❌	Yes ✅
Allows duplicates?	Yes ✅	No ❌ (keys unique)	No ❌	Yes ✅
Mutable?	Yes ✅	Yes ✅	Yes ✅	No ❌
Indexed access?	Yes ✅ by position (a[0])	No ❌ — by key only (a["key"])	No ❌ — not indexed	Yes ✅ by position (a[0])
Can change elements?	Yes ✅	Yes ✅ (change values)	Yes ✅ (add/remove items)	No ❌
Iteration	Over elements	Over keys (by default)	Over elements	Over elements
Typical use	Store ordered data	Map names to values	Store unique items, test membership	Fixed sequence of data

Agenda

- What is python?
- Pythonic style — PEP 8
- Variables, data types and data structures
- Control structures (if, for, while, comprehension syntax)
- String and file manipulation
- Functions and modular programming
- Error handling and debugging techniques
- Introduction to virtual environments and packages

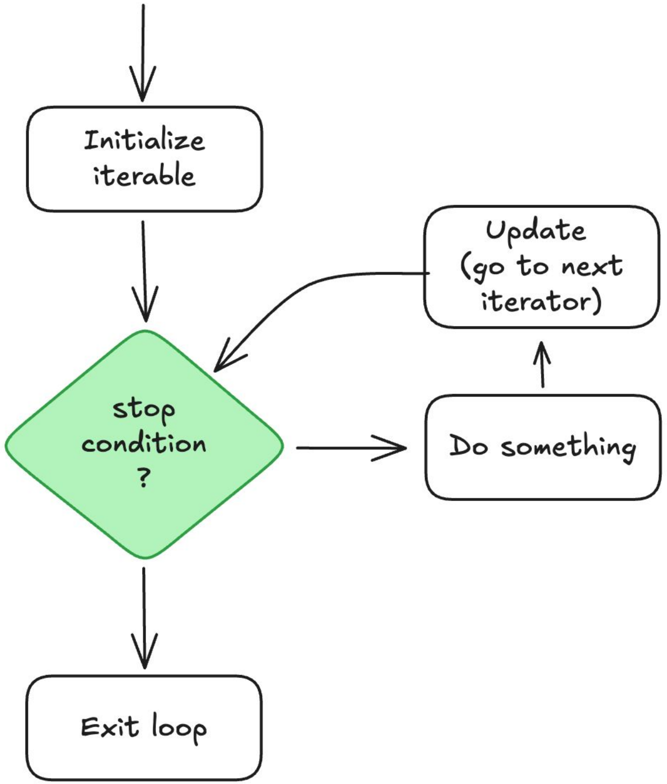
If structure

```
if (condition_1) {
    # code that runs when condition_1 is true
}
elif (condition_2) {
    # code that runs when condition_2 is true
}
else {
    # code that runs when all conditions are false (optional)
}

if(language == "FR"):
    print("Bonjour")
else:
    print("Hello")
```



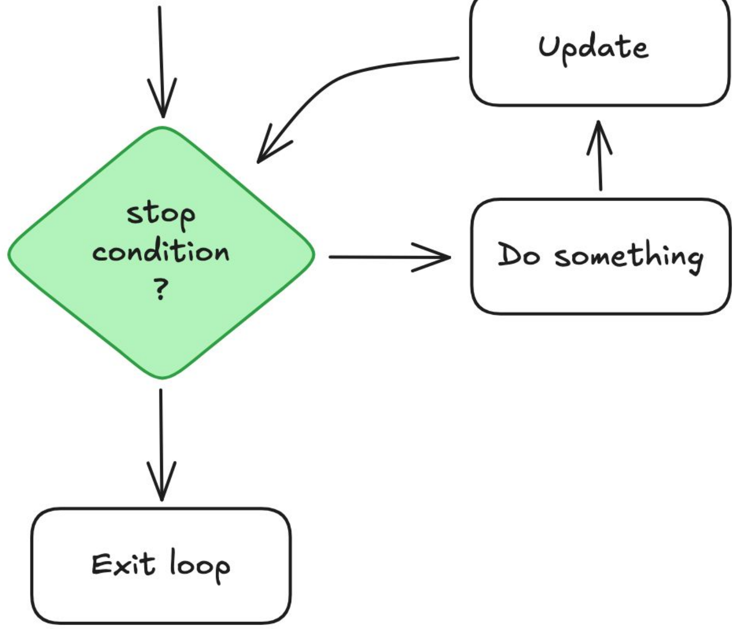
For loop

```
for variable in iterable:
    # code that runs each iteration

for letter in ["a","b","c"]:
    print(letter)

for number in range(3):
    print(number)
    # Expected output:
    # 0
    # 1
    # 2
```



While loop

```
while condition:
    # code that runs each iteration


i = 0           # initialization
while i < 5:    # condition
    print(i)    # loop body
    i += 1      # increment/update step
```



Loop - Keywords

- `break` (exits a loop early),
- `continue` (skips to the next iteration),
- `pass` (a null operation placeholder)

```python
for i in range(5):
 if i == 4:
   break
 if i == 3:
   pass
 if i == 2:
   continue
 print(i)
```

<u>Expected output:</u>

```
0
1
3
```

List comprehension

A list comprehension is a compact syntax for building a list from another iterable.

```
[expression for item in iterable if condition]
```

```
# Standard for-loop way
squares = []
for x in range(5):
    squares.append(x * 2)

# Same result using a list comprehension
squares = [x * 2 for x in range(5)]
```

Agenda

- What is python?
- Pythonic style — PEP 8
- Variables, data types and data structures
- Control structures (if, for, while, comprehension syntax)
- String and file manipulation
- Functions and modular programming
- Error handling and debugging techniques
- Introduction to virtual environments and packages

String and file manipulation - print

```
print("Hello" + " " + "World")

temperature = 20


# Print the temperature using a regular print statement with multiple arguments

print("The temperature is:",temperature)


# Print the temperature using an f-string (formatted string)

print(f"The temperature is {temperature} °C")
```

String and file manipulation - file handling

```python
# Writing to a file (this creates or overwrites the file)
with open("example.txt", "w") as file:
    file.write("Hello, this is a test file!\n")
    file.write("This is an other line.\n")

# Appending more text to the same file
with open("example.txt", "a") as file:
    file.write("This line was appended later.\n")

# Reading from the file
with open("example.txt", "r") as file:
    content = file.read()
    print("File content:")
    print(content)
```

Agenda

- What is python?
- Pythonic style — PEP 8
- Variables, data types and data structures
- Control structures (if, for, while, comprehension syntax)
- String and file manipulation
- Functions and modular programming
- Error handling and debugging techniques
- Introduction to virtual environments and packages

Functions and modular programming

A function is a reusable block of code that performs a specific task. It helps reduce repetition and makes code easier to read.

```
def my_function(my_argument):
  # do something
```

From https://peps.python.org/pep-0008/ -> function naming should be lower_case_with_underscores

Functions and modular programming

```
# Function with no argument and no return

def greet():

  print("hello")


greet()


# Function with argument and no return
def greet_someone(someone):
  print("hello",someone)


greet_someone("Bob")
```

Functions and modular programming

```
# Function with argument and return
def reverse_string(text):
  return text[::-1]

my_text = "hello"
reversed_text = reverse_string(my_text)
print(reversed_text)

# Function with a default parameter
def power(base, exponent=2):
    return base ** exponent

print(power(3))
print(power(3,3))
print(power(exponent=4,base=2)) # After a named argument you NEED to have named argument
```

Modular programming

Modular programming means splitting a large program into multiple files (modules) to keep code organized.

For example with the project structure:

```
my_project/
├── math_utils.py
└── main.py
```

Modular programming

```
#math_utils.py
def add(a, b):
    return a + b

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

def statistics(values):
    return {
        "min": min(values),
        "max": max(values),
        "sum": sum(values),
    }
```

Modular programming

```
#main.py

import math_utils


print(math_utils.add(3, 4))

print(math_utils.statistics([1, 4, 2, 9]))
```

Agenda

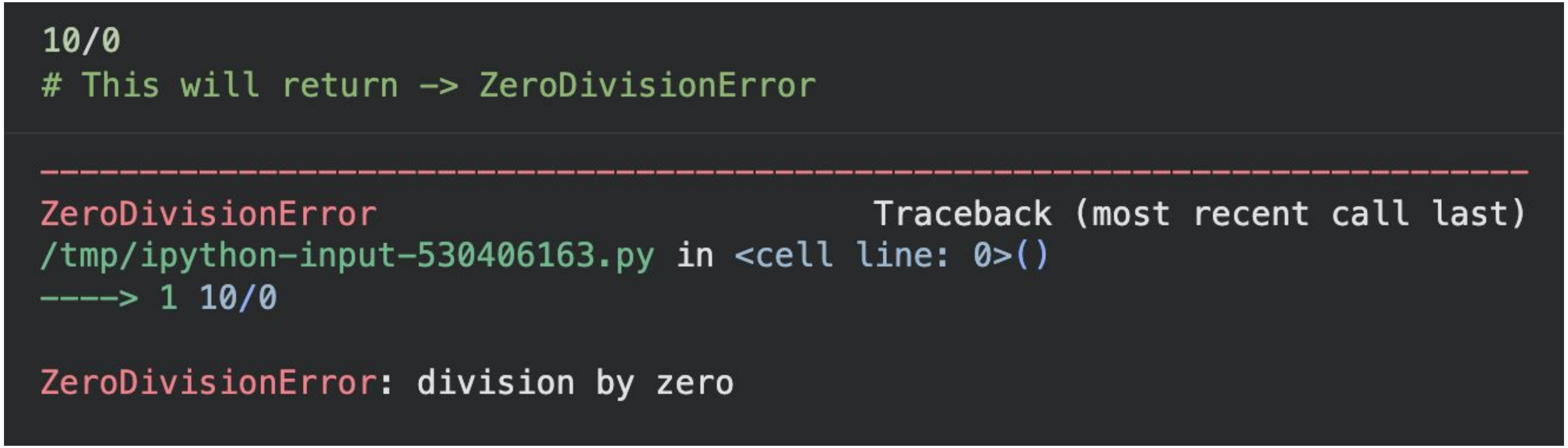
- What is python?
- Pythonic style — PEP 8
- Variables, data types and data structures
- Control structures (if, for, while, comprehension syntax)
- String and file manipulation
- Functions and modular programming
- Error handling and debugging techniques
- Introduction to virtual environments and packages

Error handling and debugging techniques

Generic pattern

```
try:
    y = float(input("Enter a decimal number: "))
except ValueError:
    print("Invalid input. Please enter a decimal value.")
else:
    print("Your number squared is", y * y)
finally:
    print("Execution finished.")
```

Error handling and debugging techniques



```
10/0
# This will return -> ZeroDivisionError
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipython-input-530406163.py in <cell line: 0>()
----> 1 10/0

ZeroDivisionError: division by zero
```

Error handling and debugging techniques

```
# General error handling:

try:

  10/0

except Exception  as e:

  print(f"The following error
occured: {e}")
```

```
# Specific error handling:

try:

  10/0

except ValueError:

    print("That wasn't a number.")

except ZeroDivisionError:

    print("Division by zero?")
```

Error handling and debugging techniques

```
# Bad error handling:
try:
  10/0
  "string"/10
except Exception  as e:
  print(f"The following error occured: {e}")
```

Agenda

- What is python?
- Pythonic style — PEP 8
- Variables, data types and data structures
- Control structures (if, for, while, comprehension syntax)
- String and file manipulation
- Functions and modular programming
- Error handling and debugging techniques
- Introduction to virtual environments and packages

Virtual environments

A virtual environment is an isolated Python workspace that contains its own interpreter and set of installed packages. Instead of installing everything globally on your system, each project gets its own environment.

This prevents version conflicts, for example, one project can use NumPy 1.23 while another uses NumPy 2.0, because each environment manages its dependencies independently.

Why Use Virtual Environments?

- They prevent dependency collisions between projects.
- They make your project reproducible and easier to share.
- They keep your system's Python installation clean and stable

Packages

Packages are reusable pieces of Python code (libraries or modules) that provide additional functionality.

To install them you will usually run a command such as

```
pip install pandas
```

And to call them in a script

```python
import pandas as pd
```

Example - open a csv

```
import pandas as pd

# Load the CSV file
df = pd.read_csv("data.csv")

# Show the first 5 rows
print(df.head())

# Access a specific column
print(df["column_name"])
```