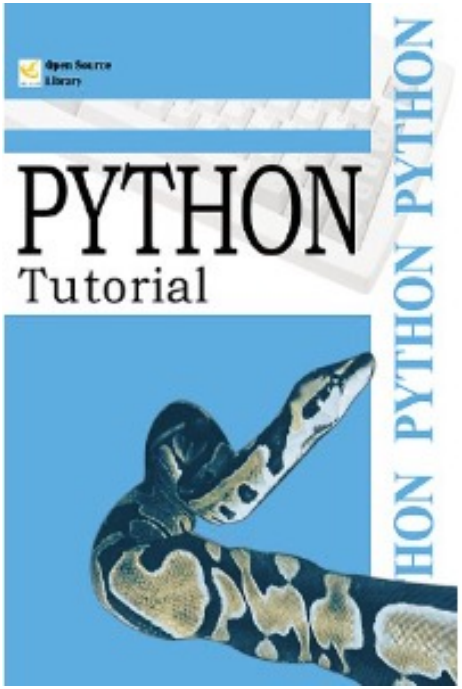
Introduction to Python



Dra. Mª Dolores Rodríguez Moreno

Objectives

Specific Objectives

- Understand the main Python features
- Overview of the language components

Source

- https://docs.python.org/3.10/tutorial/appetite.html
- https://python-textbok.readthedocs.io/en/1.0/index.html
- Python Tutorial - Tapa blanda. GuidoVan Rossum (2012)

Outline

- **Introduction**
- Why Study Python?
- Python Interpreter
- An informal introduction
- Numbers
- Strings
- Lists
- Functions
- Variable Scope

Introduction (I)

- Python was created by Guido van Rossum (TN)
  - Python 2.0: released on 2000
  - Python 3.0: released on 2008. Backwards-incompatible
- Python is:
  - General-purpose: many applications
  - High-level: abstract data structures doing more with less
  - Interpreted:
    - No compilation needed → directly run the code
    - Interactive mode for testing and debugging
- Emphasizes code *readability* and programmer's *productivity*

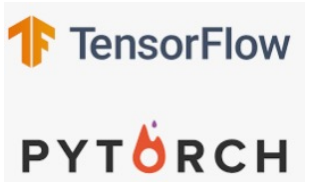
Introduction (II)

- Named for the BBC show
- Several paradigms: procedural, OOP, functional
- Extensive Standard Library with modules for various tasks like file I/O, system calls, sockets and more
- Strong Community Support: abundant resources tutorials & forums

Introduction (III)

- Web Development: with frameworks like Django and Flask

- Data Science and Machine Learning: with libraries like
  Pandas NumPy SciPy TensorFlow Pytorch



- Game Development: with libraries like Pygame

- Embedded Systems: as MicroPython and CircuitPython

Outline

- Introduction
- **Why Study Python?**
- Python Interpreter
- An informal introduction
- Numbers
- Strings
- Lists
- Functions

Why Study Python?

- Automate Tasks:
  - Perform search-and-replace over large text files
  - Rename and rearrange photo files
  - Write custom databases GUI applications or simple games
- For Developers:
  - Faster development cycle compared to C/C++/Java
  - Write test suites efficiently
  - Use Python as an extension language
- Advantages over Other Languages:
  - Simpler than C/C++/Java
  - Available on Windows macOS and Unix
  - Ideal for both small scripts and large programs

Why Study Python?

- Ease of Use
  - Simple syntax, easy to learn and use
  - High-level data types (arrays & dictionaries)
- Modular and Reusable
  - Split programs into reusable modules
  - Large collection of standard modules (fie I/O system calls GUI toolkits)
- Readable and Compact Code
  - Shorter programs than C/C++/Java
  - Indentation for statement grouping
  - No variable or argument declarations needed
- Extensible
  - Add new functions or modules in C
  - Link Python to binary libraries

Why Study Python?

Python

```
#!/usr/bin/python

print("Hello, world!")
```

Java

```
public class HelloWorld {
  public static void main(String[]
      args) {
    System.out.println("Hello, world
        !");
  }
}
```

C

```
#include <stdio.h>

int main()
{
  printf("Hello, world!\n");
}
```

C++

```
#include <iostream>

int main()
{
  std::cout << "Hello, world!\n"
      ;
}
```

Popularity

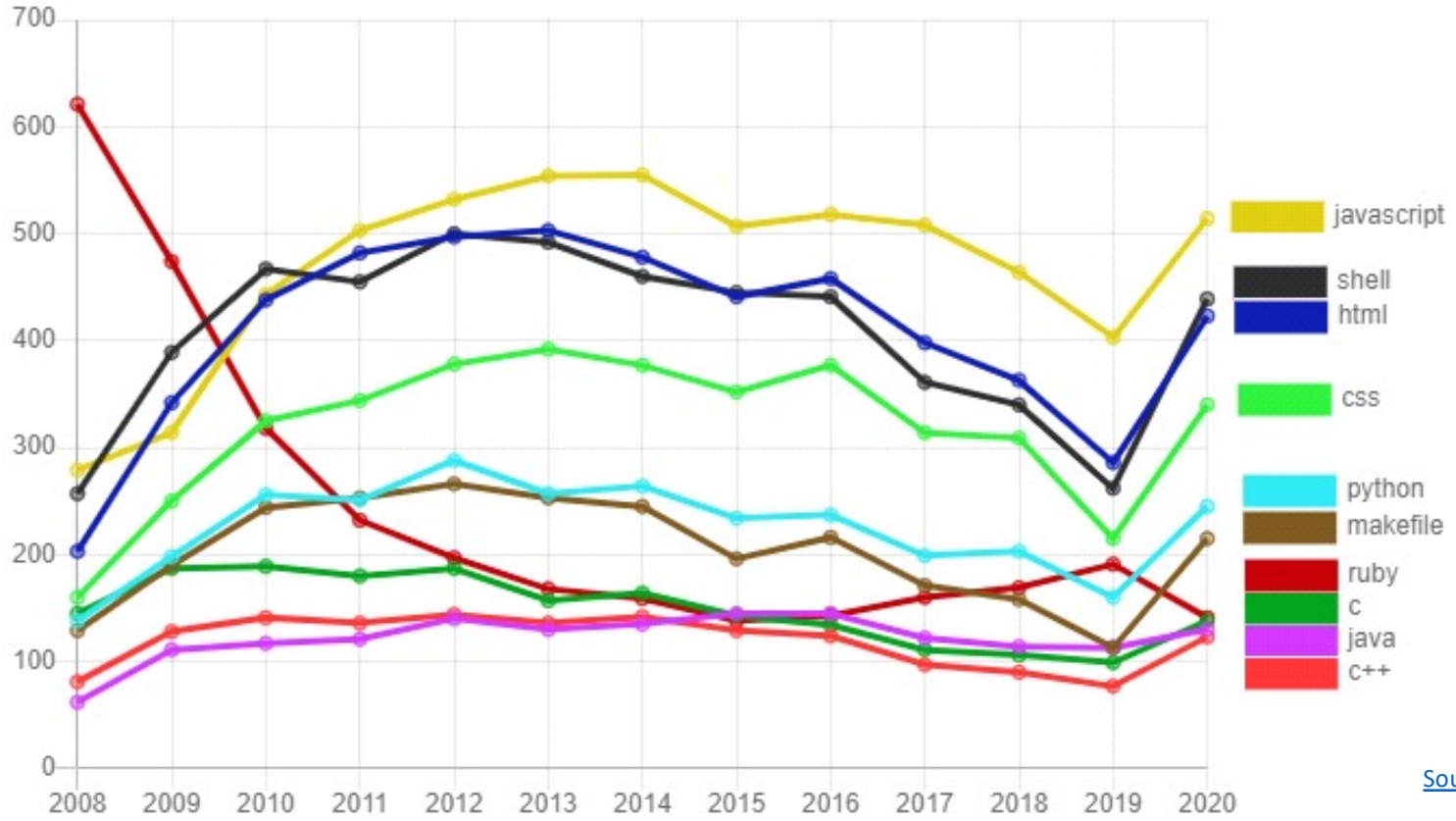
The PYPL PopularitY of Programming Language Index is created by analyzing how often language tutorials are searched on Google.

The more a language tutorial is searched, the more popular the language is assumed to be. It is a leading indicator. The raw data comes from Google Trends.

Worldwide, Jun 2024 :

Rank	Change	Language	Share	1-year trend
1		Python	29.06 %	+1.4 %
2		Java	15.97 %	+0.2 %
3		JavaScript	8.7 %	-0.6 %
4		C#	6.73 %	-0.0 %
5		C/C++	6.4 %	-0.0 %
6	↑	R	4.75 %	+0.3 %
7	↓	PHP	4.57 %	-0.5 %
8		TypeScript	3.0 %	-0.1 %
9		Swift	2.76 %	+0.3 %
10		Rust	2.5 %	+0.4 %
11		Objective-C	2.39 %	+0.3 %
12		Go	2.25 %	+0.3 %
13		Kotlin	1.98 %	+0.1 %
14		Matlab	1.47 %	-0.2 %
15	↑↑↑	Dart	1.02 %	+0.1 %

Popularity



Programming Languages popularity over time (GitHub)

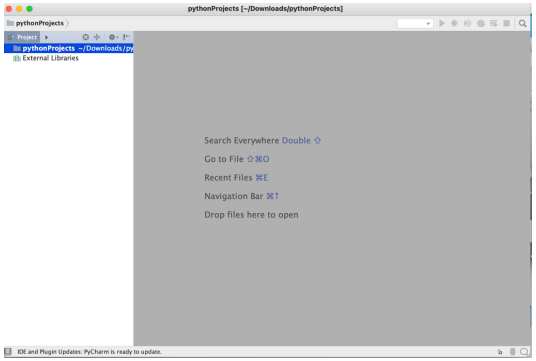
Source: Orlowska et al (2021)

Outline

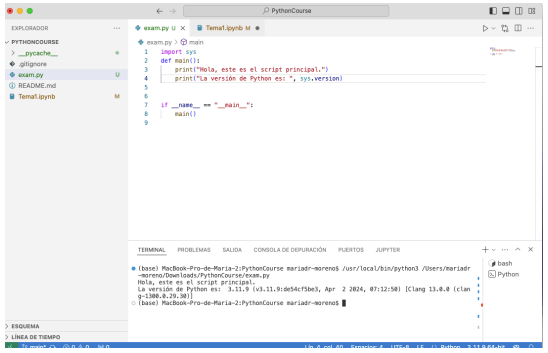
- Introduction
- Why Study Python?
- **Python Interpreter**
- An informal introduction
- Numbers
- Strings
- Lists
- Functions

Python Interpreter (I)

- If you have a Linux or Mac you already have Python!
- If you have Windows you have to install it
- There is no standard IDE



PyCharm



Visual Studio Code

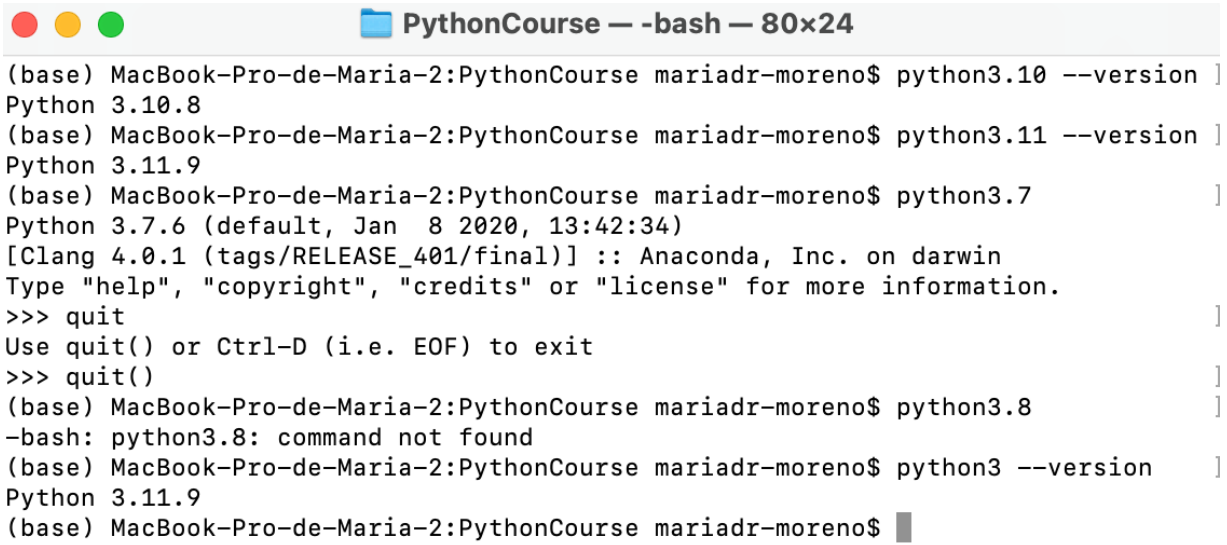
Python Interpreter (II)

- Python is an interpreted language, i.e. , it needs an interpreter
  - Interpreted = it is not complied = it needs no compilation
  - Faster development, slower execution
- Three operation modes:
  - Interactive: the interpreter reads the program from the stdin . From a terminal: usually the keyboard
  - Non-interactive: the interpreter reads the program from a file (.py) → Script. A python script is a sequence of python instructions stored in a .py
  - Mixed: from a Jupiter Notebook (.ipynb)

Interactive (I)

- Just run Python
- Different names for different versions to avoid conflicts
- python python3.7 ...

> > >



- The programmer executes as s/he writes code down

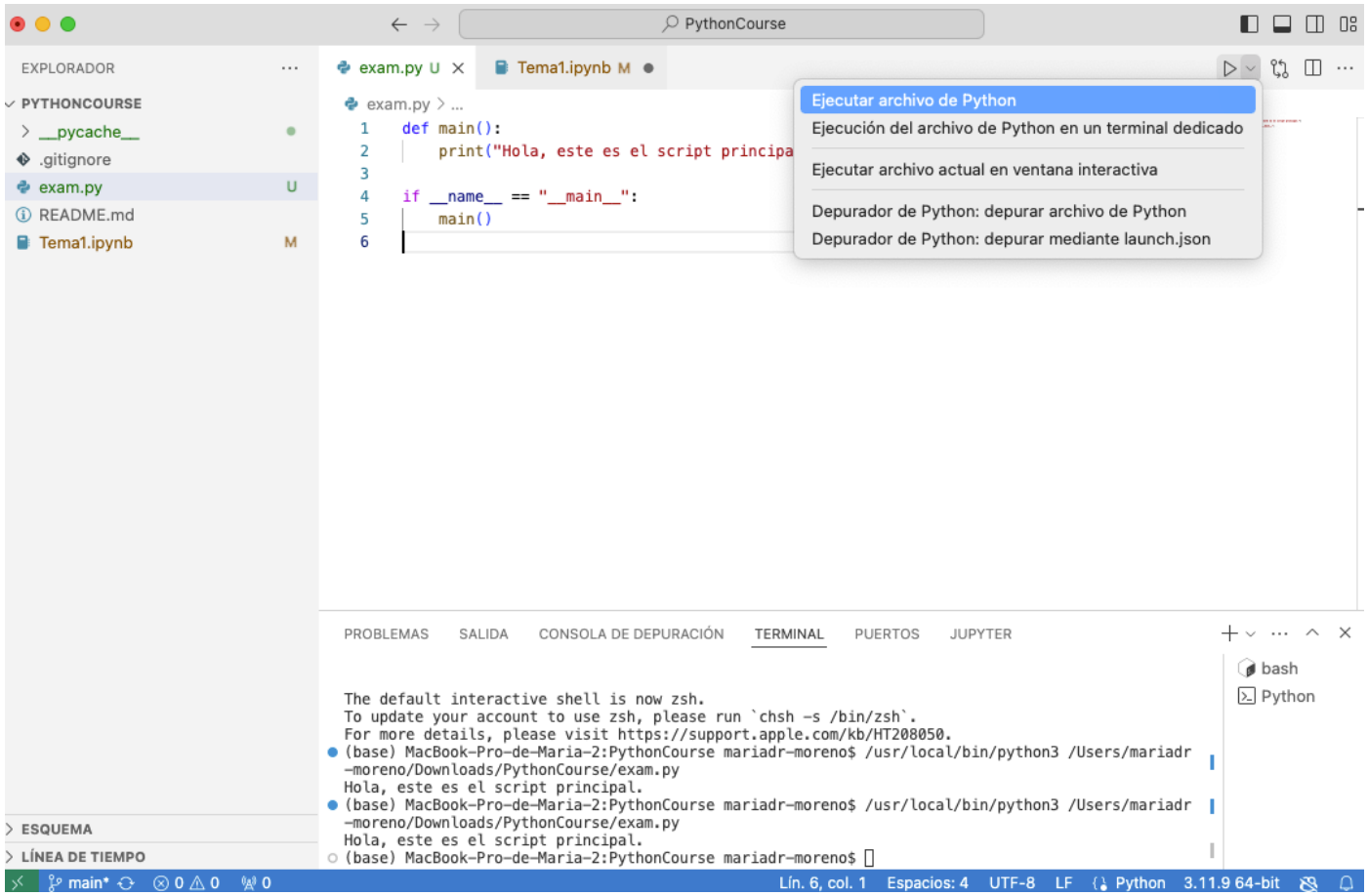
Non Interactive (I)

- The program is in a plain text file
- It can be edited with any text editor
- Extension ".py"
- By default UTF-8 encoding

exam.py > ...

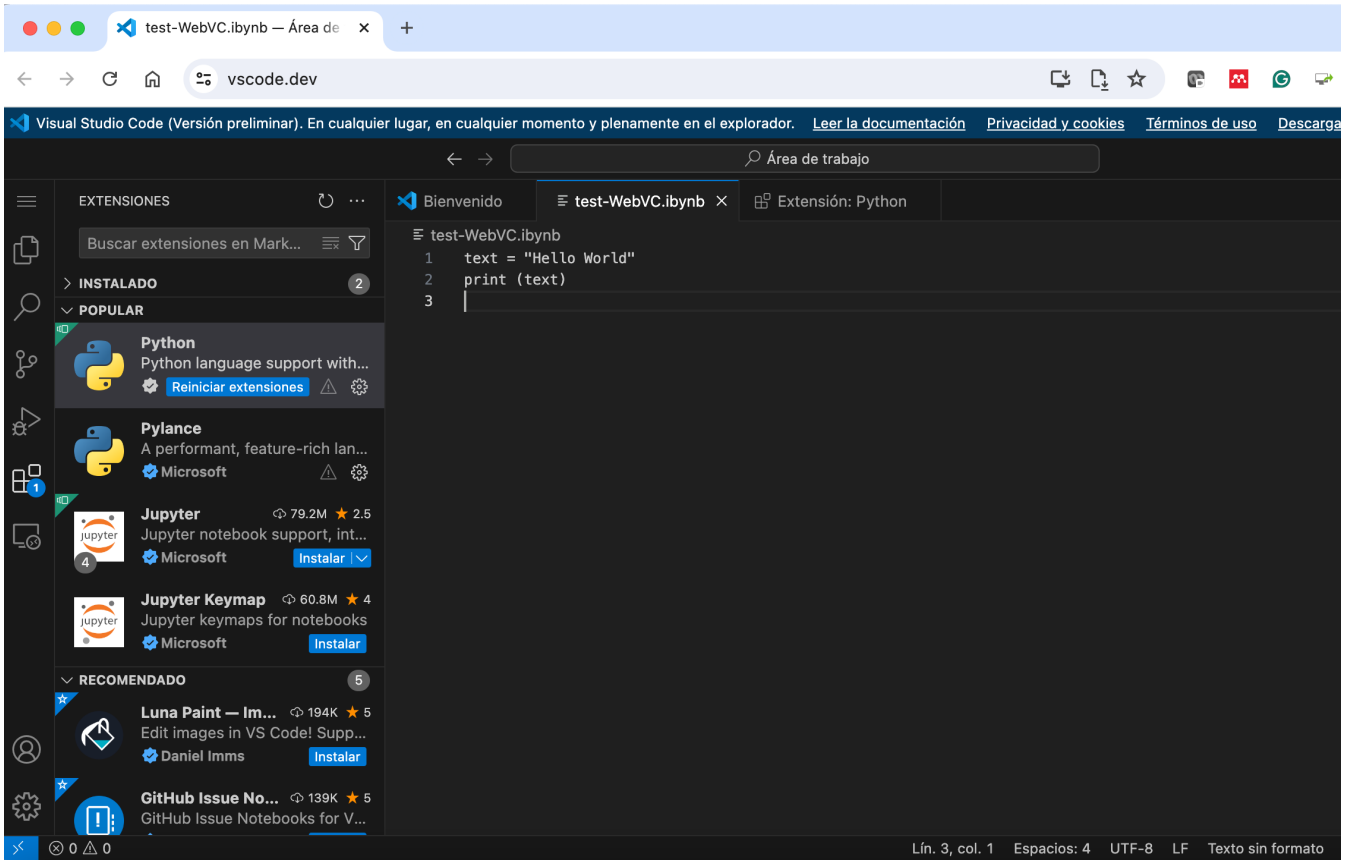
```
1   def main():
2       print("Hola, este es el script principal.")
3
4   if __name__ == "__main__":
5       main()
```

Non Interactive (II)



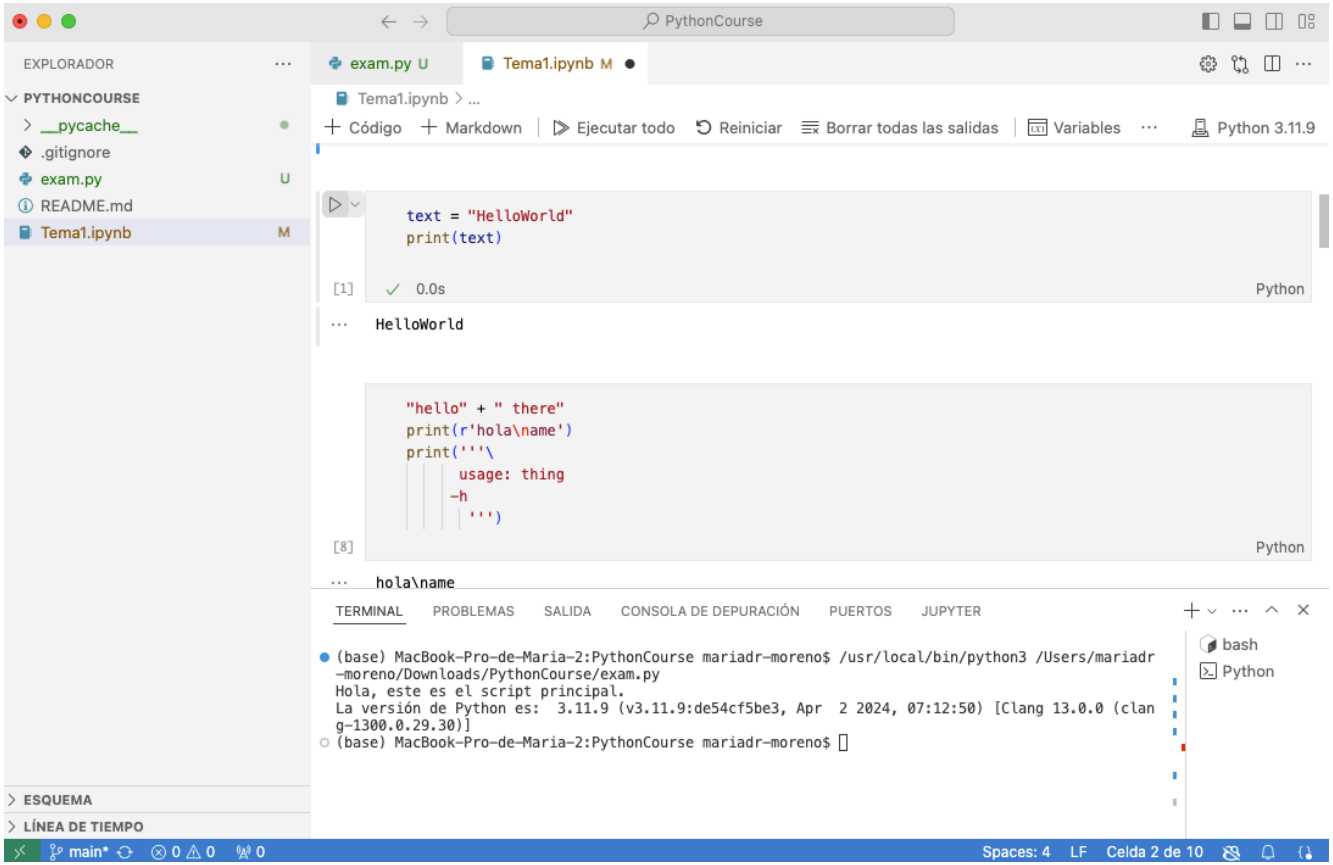
Non Interactive (III)

- Visual Code Web
- https://vscode.dev/



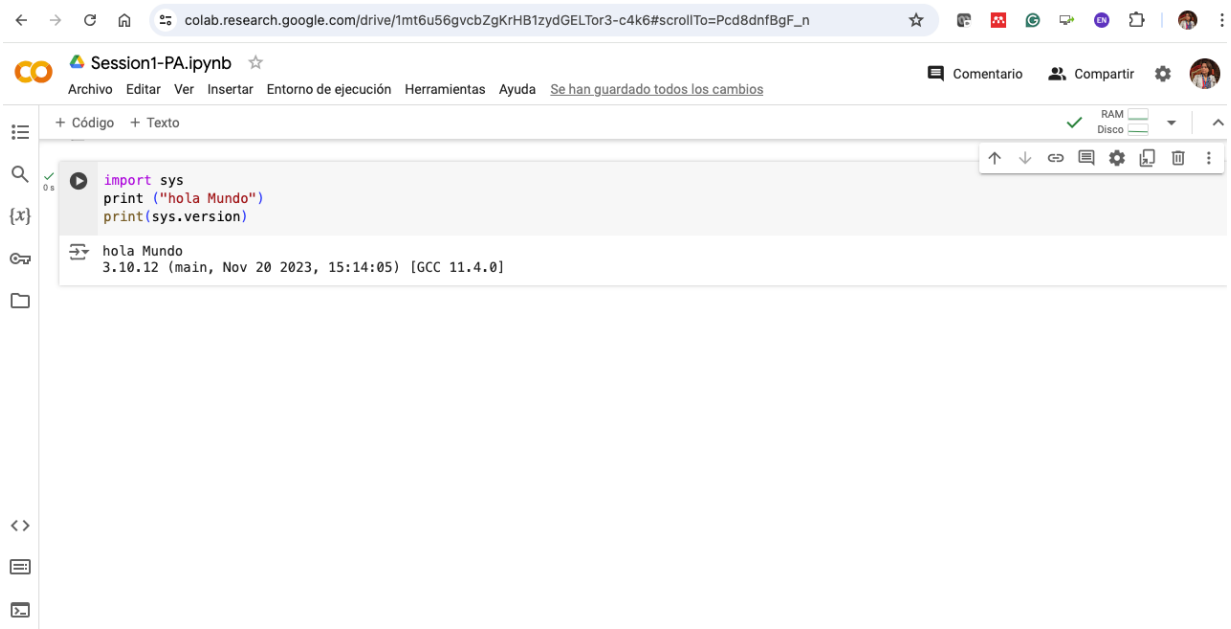
Mixed(I)

- Using a Jupiter Notebook
- .ipynb



Mixed(II)

- Google Collab
- https://colab.research.google.com/



Outline

- Introduction
- Why Study Python?
- Python Interpreter
- **An informal introduction**
- Numbers
- Strings
- Lists
- Functions

Keywords

- Reserved words with specific purposes and cannot be used for other purposes
- Examples of Python keywords:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	break
except	in	raise		

Identifier Names

- Names given to entities like variables functions and classes
- Rules for forming identifiers:
  - May contain letters numbers or underscore (_)
  - Cannot start with a number
  - Cannot be a keyword
- Guidelines for naming:
  - Be descriptive
  - Avoid unnecessary abbreviations
  - Follow naming conventions

Naming conventions(I)

The Python community has these recommended naming conventions

- joined_lower for functions, methods and, attributes
- joined_lower or ALL_CAPS for constants
- StudlyCaps for classes
- camelCase only to conform to pre-existing conventions
- Attributes: interface, _internal, __private

Naming conventions (II)

Variable Names:

- Use lowercase letters
- Separate words with underscores (_)
- Example: my_variable, total_sum, count

Function Names:

- Use lowercase letters
- Separate words with underscores (_)
- Example: my_function(), calculate_total(), get_user_input()

Class Names:

- Use CamelCase (capitalize the first letter of each word)
- Do not use underscores (_)
- Example: MyClass, UserProfile, DataAnalyzer

Naming conventions (III)

Constant Names:

- Use all uppercase letters
- Separate words with underscores (_)
- Example: MAX_VALUE, PI, DEFAULT_TIMEOUT

Module and Package Names:

- Use lowercase letters
- Can use underscores (_) if necessary to improve readability
- Example: my_module.py, user_profile.py, data_analyzer.py

Method Names:

- Follow the same convention as function names
- Use lowercase letters and separate words with underscores (_)
- Example: get_user_name(), calculate_total(), save_data()

Naming conventions (IV)

Private Variable and Method Names:

- Prefix with a single underscore (_) to indicate they are intended for internal use
- Example: _my_private_variable, _my_private_method()

Special Method Names:

- Use double underscores (__) before and after the name
- These are also known as "dunder" methods or magic methods
- Example: __init__(), __str__(), __repr__()

Example of identifiers

Invalid	**Reason**
Person Record	Identifier contains a space
DEFAULT-HEIGHT	Identifier contains a dash
class	Identifier is a keyword
2totalweight	Identifier starts with a number

Indentation and Semicolons

- Indentation to delimit blocks of code
- No need of {}
- No need for semicolons ";" to mark the end of instructions
- Can be used to put multiple instructions on a single line (not recommended)

```
## Individual instructions -- no semicolons
print("Hello!")

print("Here's a new instruction")

a = 2

# This instruction spans more than one line

b = [1        2        3

     4        5        6]

# This is legal but we should NOT do it

c = 1; d = 5

print("Here's another"  c  d)
```

Whitespace & Colons

- White space matters:
  - Indentation must be consistent
  - Use tabs or white spaces (don't mix them)
  - IDE can handle for you
- 'pass' is an empty command used for empty indentation block
- Colons (":") start of a new block in many constructs e.g. function definitions then clauses

Comment

- Comments start with # and continue until the end of the line
- Used to describe what the program does and how it works
- More than 1 line use """ ... """

```
# This is a multiline comment

# Each line starts with a hash (#)

# and continues until the end of the line.

print("Hello        World!")

"""

This is a multiline comment.

It spans multiple lines.

However   this is typically used for docstrings.

"""

print("Hello        World!")
```

Docstrings

- Type of multiline comment used to document modules classes functions and methods
- They are written using triple quotes (""" or ''') and can span multiple lines
- Different from regular comments because they are stored as an attribute of the object they document and can be accessed programmatically using tools like help() or __doc__

```
def greet(name):
    """
    This function greets the person whose name is passed as an argument.
    Parameters:
    name (str): The name of the person to greet.
        Returns:
    None
    """
    print("Hola " name "!")
# Using the help function to access the docstring
help(greet)
print(greet.__doc__)
```

Flow of control

- The order in which the computer executes instructions
- Example of flow of control

```
# this function definition starts a new block

def print_numbers(a b):

    # this instruction is inside the block

    print(“Numbers are:” a b)

# this if statement starts a new block

if it_is_tuesday:

    # this is inside the block

    print("It's Tuesday!")

# this is outside the block!

print("Print this no matter what")
```

Exercise

- The following Python program is not indented correctly
- Re-write it so that it is correct

```
def happy_day(day):

if day == "monday":

return ":("

if day != "monday":

return ":D"



print(happy_day("sunday"))

print(happy_day("monday"))
```

Reading and Writing

- For reading use “input”
- It reads a string

- For writing use “print”

```
first_number = input("Enter the first number: ")


print("The number is"   first_number)
```

Assignment(I)

- *Binding a variable* in Python means setting a *name* to hold a *reference* to some *object*

  Assignment creates references not copies

- Names in Python do not have an intrinsic type; objects have types
  - Python determines the type of the reference automatically based on what data is assigned to it
  - Basic types: numbers, strings, Booleans, complex
- You create a name the first time it appears on the left side of an assignment expression: `y = 5`
- A reference is deleted via garbage collection after any names bound to it have passed out of scope
- Python uses *reference semantics* (more later)

Assignment (II)

- You can assign multiple names at the same time
- This makes it easy to swap values

```
x, y = 2, 3
```

- Assignments can be chained

```
a = b = x = 2
```

Built-in Types

- Types of information Python can handle:
  - integers
  - floating numbers
  - strings
  - boolean
  - complex
  - None

- char is not a data-type instead is a string

```
print(type(1))      # <class 'int'>

print(type("a"))    # <class 'str'>

print(type(2.45))   # <class 'float'>

print(type(True))   # <class bool'>

print(type(4 + 4j))#<class 'complex'>

print(type(x))      #<class 'NoneType'>

print(type(´c´))    # <class 'str'>
```

Built-in Types

- Types of information Python can handle:
  - integers
  - floating numbers
  - strings
  - boolean
  - complex
  - None

- char is not a data-type instead is a string

```
print(type(1))      # <class 'int'>

print(type("a"))    # <class 'str'>

print(type(2.45))   # <class 'float'>

print(type(True))   # <class bool'>

print(type(4 + 4j))#<class 'complex'>

print(type(x))      #<class 'NoneType'>

print(type(´c´))    # <class 'str'>
```

Cast Types

- Cannot do arithmetic operations on variables of different types
- Casting: the operation of converting a variable to a different type
  - int()
  - float()
  - str()

Outline

- Introduction
- Why Study Python?
- Python Interpreter
- An informal introduction
- **Numbers**
- Strings
- Lists
- Functions
- Variable Scope

Numbers

- Integer numbers: e.g. 1 5 -34
- Operators:

Sign	Operator	Sign	Operator
=	Assignment	//	Floor division
+	Add	**	Exponent
-	Substration	+=	Assign +
*	Multiplication	-=	Assign -
/	Division	*=	Assign *
%	Modulus	/=	Assign /

Operator Precedence

- Similar rules as other languages

```
() (Parentheses)

** (Exponentiation)

+x   -x   ~x (Unary plus   Unary minus   Bitwise NOT)

*   /   //   % (Multiplication   Division   Floor Division
Modulus)

+   - (Addition   Subtraction)

<<   >> (Bitwise Shift Operators)

& (Bitwise AND)
```

```
^ (Bitwise XOR)

| (Bitwise OR)

Comparison operators: ==   !=   >   >=   <   <=   is   is
not   in   not in

not (Logical NOT)

and (Logical AND)

or (Logical OR)
```

Examples

```
# Exponentiation has the highest precedence

result = 2 ** 3 ** 2

# Equivalent to: 2 ** (3 ** 2) = 2 ** 9 = 512

# Multiplication and division have higher precedence than
addition and subtraction

result = 2 + 3 * 4 # Equivalent to: 2 + (3 * 4) = 2 + 12 =
14

# Parentheses can be used to override the default precedence

result = (2 + 3) * 4 # Equivalent to: 5 * 4 = 20
```

```
a = int(input("Number: "))

b = float(input("Number: "))

d = a * b /2

d += 1

C = d **2

print("Result c y d"   c  d)
```

Exercise

- Which of the following numbers are valid Python integers?
  - 110 1.0 17.5 -39 -2.3
- Explain the results of the following operations:
  - 15 + 20 * 3
  - 13 // 2 + 3
  - 31 + 10 // 3
  - 20 % 7 // 3
  - 3 ** 3 ** 2

Outline

- Introduction
- Why Study Python?
- Python Interpreter
- An informal introduction
- Numbers
- **Strings**
- Lists
- Functions
- Variable Scope

Strings (I)

- Can enclose in double or single quote
  - ‘ ’ # a string with a single quote
  - ” “ # a string with a double quote
- Triple quotes can cross end of line boundaries

```
>>>''' a two line
    string'''
```

- Strings are **immutable**
- Strings are objects of the “str” class

String Operations

- Concatenation: combining strings using the + operator
- Repetition: repeating a string using the * operator
- Indexing: accessing individual characters using indices []
- Slicing: extracting substrings using slice notation [:]

Examples

```
## Concatenation

hello = "Hello"

world = "World"

greet = hello + " " + world  # "Hello World"

# Repetition

laugh = "Ha"

repeated_laugh = laugh * 3  # "HaHaHa"
```

Indexing Operations

- Index starts with CERO (0)
- Index can be negative
- If -1 starts from the end until the length is reached
- Control you don´t overpass the limit → error

Examples

```
greet = "hola" + " " + "Spain"

# Indexing

char = greet[1]  # 'o'

char1 = greet[-1] # 'n'

char2 = greet[-11] # Error

greet[0] = "H" # Error
```

Slicing Operations

- Strings can be used as a sequence of characters

```
greet = "hola" + " " + ”Spain"

# Slicing

greet[2:] #'la Spain'

greet[:2] #'ho'

greet[2:] + greet[:2] #'la Spainho'

greet[2:4] #'la'
```

String Methods

- Strings have built-in methods for various operations
- Function: len()
- Common methods:
  - upper()
  - lower()
  - strip()
  - replace()
  - split()
  - join()
  - find()
  - count()

Examples of methods

```
s = "  Hello, World!  "

# Convert to uppercase

upper_s = s.upper()  # "  HELLO, WORLD!  "

# Convert to lowercase

lower_s = s.lower()  # "  hello, world!  "

# Strip whitespace

stripped_s = s.strip()  # "Hello, World!”

# Replace substring

replaced_s = s.replace("World", "Python")#” Hello, Python!  "
```

```
# Split into a list

split_s = s.split(",")  # ["  Hello", " World!  "]

# Join list into a string

joined_s = " ".join(["Hello", "World"])  # "Hello World"

# Find substring

index = s.find("World")  # 8

# Count occurrences

count = s.count("l")  # 3
```

Examples of len

```
# Length of an empty string

empty_string = ""

print(len(empty_string))  # Output: 0

# Length of a string with spaces

string_with_spaces = "Hello, World! "

print(len(string_with_spaces))  # Output: 14

# Using len() with other data types

my_list = [1, 2, 3, 4, 5]

print(len(my_list))  # Output: 5
```

String Formatting

- Formatting strings use *placeholders* or *f-strings*
- Placeholders:
  - % operator
  - str.format() method
- f-strings: introduced in Python 3.6, use {} to embed expressions inside string literals

Examples

```
name = "Alice"

age = 30

# Using % operator

formatted_str = "My name is %s and I am %d years old." % (name, age)

# Using str.format() method

formatted_str = "My name is {} and I am {} years old.".format(name, age)

# Using f-strings

formatted_str = f"My name is {name} and I am {age} years old."
```

Escape Sequences

- Special sequences in strings to represent certain characters
- Common escape sequences:

Sequence	Meaning
\\	literal backslash
\'	single quote
\"	double quote
\n	newline
\t	tab

Examples

```
# Literal backslash

backslash_str = "This is a backslash: \\"

print(backslash_str)  # Output: This is a backslash: \

# Single quote

single_quote_str = 'It\'s a single quote.'

print(single_quote_str)  # Output: It's a single quote.

# Double quote

double_quote_str = "He said, \"Hello!\""

print(double_quote_str)  # Output: He said, "Hello!"
```

```
# Newline

newline_str = "First line\nSecond line"

print(newline_str)

# Output:

# First line

# Second line

# Tab

tab_str = "Column1\tColumn2"

print(tab_str)  # Output: Column1   Column2
```

Exercise (I)

```
# I want to visualize the following text. Create 1 string and use += operator

File path: C:\Users\Username\Documents

Quote: "Stay hungry, stay foolish." - Steve Jobs

List:

    1. Item One

    2. Item Two

    3. Item Three
```

Exercise (II)

Write a program that takes a sentence from the user, converts the entire sentence to lowercase, removes leading and trailing whitespace, counts the number of words, and replaces a specific word with another

Outline

- Introduction
- Why Study Python?
- Python Interpreter
- An informal introduction
- Numbers
- Strings
- **Lists**
- Functions
- Variable Scope

Lists (I)

- An ordered collection of **mutable** data
  - Ordered: data in the list have a location
  - Mutable: data can be modified
  - Data types can be different
- Concatenation & Repetition
- Slice & Indexing notation

```
a = ['spam', 'eggs', 123]

print(a)

print(a[2])          # 123

print(a[1:])         # ['eggs', 123]

print(a + a[2:])     # ['spam','eggs',123,123]

a[0] = "jam"         # ['jam','eggs',123, 123]

print(a*2)           # ['jam', 'eggs', 123, 123, 'jam',

                     # 'eggs', 123, 123]
```

List Methods

- List have built-in methods for various operations
- Function: len()
- Common methods:
  - append()
  - extend()
  - insert()
  - remove()
  - pop()
  - clear()
  - index()
  - count()
  - sort()
  - reverse()

Examples of methods

```
fruits = ["apple", "banana", "cherry"]

# Append an element

fruits.append("date")  # ["apple", "banana", "cherry", "date"]

# Extend list with another list

fruits.extend(["elderberry", "fig"])  # ["apple", "banana", "cherry", "date", "elderberry", "fig"]

# Insert an element at a specific position

fruits.insert(1, "blueberry")  # ["apple", "blueberry", "banana", "cherry", "date", "elderberry", "fig"]

# Remove an element

fruits.remove("banana")  # ["apple", "blueberry", "cherry", "date", "elderberry", "fig"]
```

Examples of methods

```
# Pop an element (remove and return it)

popped_fruit = fruits.pop()  # "fig", ["apple", "blueberry", "cherry", "date", "elderberry"]

# Clear the list

fruits.clear()  # []

# Index of an element

index_of_cherry = fruits.index("cherry")  # 2

# Count occurrences of an element

count_of_apple = fruits.count("apple")  # 1

# Sort the list          fruits.sort()  # ["apple", "banana", "cherry", "date"]

# Reverse the list       fruits.reverse()  # ["date", "cherry", "banana", "apple"]
```

Outline

- Introduction
- Why Study Python?
- Python Interpreter
- An informal introduction
- Numbers
- Strings
- Lists
- **Functions**
- Variable Scope

Functions

- Functions are reusable blocks of code that perform a specific task
- Functions help in organizing code and avoiding repetition
- Define using the **def** followed by the function name and ()
  - **Parameters**: variables listed inside the parentheses in the function definition
  - **Arguments**: values passed to the function when it is called
  - Functions can have **default parameters**, allowing some arguments to be optional
- Return a value using the ***return*** statement
  - You can return 1 or more values separated by “,”
  - If no return statement is used, the function returns **None** by default

Example

```
def square(number=3):
    """
     This function returns the square of the given number.
    """
    return number ** 2

# Function without return statement
def say_hello():
    print("Hello, World!")
```

```
print(square(4))  # Output: 16

print(square())  # Output: 9

result = say_hello()

print(result)  # Prints the string and print None

print(square.__doc__)

help(square)
```

Main Function

- The main() function serves as the entry point for a Python program
- Helps in organizing code and makes it easier to understand and maintain
- Useful for defining a clear starting point for the program's execution
- Why Use main()?
  - Readability: makes the program structure clear and logical
  - Modularity: encapsulates the main logic of the program in a single function
  - Reusability: allows for parts of the code to be reused or tested separately
  - Best Practices: aligns with common programming conventions and prepares for larger projects

Example

```
def main():

    # Main logic of the program

    # Example: Calling functions, performing tasks, etc.

    pass



if __name__ == "__main__":

    main()
```

Exercise

Write a function called *process_text* that takes a string and a list of words as arguments. The function should:

- Convert the string to lowercase
- Remove leading and trailing whitespace
- Replace any word in the string that matches an element in the list of words with asterisks (*)
- Provide default values for the text and words to replace
- Return the processed string and the number of words replaced
- Call the *process_text* from a main function

Exercise: Output

```
# Function output with default values

Processed text: "this is a default text with ****** and *******
words."

Number of words replaced: 2

# User input and function output with custom values

Processed text: "python is an amazing programming language.
python is popular."

Number of words replaced: 3
```

Outline

- Introduction
- Why Study Python?
- Python Interpreter
- An informal introduction
- Numbers
- Strings
- Lists
- Functions
- **Variable Scope**

Variable Scope

- Region of the code a variable is accessible
- Variables declared in one part of the code may not be accessible in another
- 4 main types:
  1. **Local Scope:** variables defined inside a function, only accessible within that function
  2. **Enclosing Scope:** variables in the local scope of enclosing functions, often used in nested functions (**nonlocal** keyword)
  3. **Global Scope:** variables defined at the top level of a script or module, or explicitly declared as global using the **global** keyword. Accessed from any part of the code
  4. **Built-in Scope:** special reserved keywords and functions that are part of built-in namespace. E.g. len("Hello")

Example

```
#Local Scope

def my_function():

    x = 10

    print(x)  # This will print 10

my_function()

print(x)  # This will raise an error
```

```
#Enclosing Scope

def outer_function():

    x = 10  # Variable in the enclosing scope

    def inner_function():

        nonlocal x

        x += 5

        print(f"Inside inner_function: {x}")  # 15

    inner_function()

    print(f"Inside outer_function: {x}")  # 15

outer_function()
```

Example

```
#Global Scope (I)

x = 30

def my_function():

    print(x)  # This will print 30

my_function()

print(x)  # This will print 30
```

```
#Global Scope (II)

x = 20

def my_function():

    global x

    x+=1

    print(x)  # This will print 21

my_function()

x+=1

print(x)  # This will print 22
```

Exercise...

Create a Python program that manages a to-do list using global, local, and nonlocal variables, and demonstrates the use of built-in functions. Include a main() function

Global Variable: named **tasks** which will be a *list* to store the tasks. Functions:

- add_task(task): add a new task to the global tasks list
- remove_task(task): remove a task from the global tasks list if it exists
- list_tasks(): list all tasks stored in the global tasks list
- task_manager(): manage the operations of adding, modifying, and listing tasks
  - Modifies the first task in the tasks list
  - Uses the nonlocal keyword to refer to the tasks variable defined in the task_manager scope

...Exercise

main() Function: define a main() function that calls task_manager() to execute the main logic of the program

Use of Built-in Functions: use the built-in enumerate function in list_tasks() to number the tasks

Points to Consider:

- Global variables (tasks) are accessible and modifiable from any function in the script
- Local variables are defined within functions and are only accessible within those functions
- The nonlocal keyword is used within modify_task() to refer to the tasks variable from the task_manager scope

Example of the output...

```
Task 'Buy milk' added.

Task 'Call the doctor' added.

Task 'Exercise' added.

Task list:

1. Buy milk

2. Call the doctor

3. Exercise

Task 'Buy milk' modified to 'Buy milk (modi)'
```

```
Task list:

1. Buy milk (modified)

2. Call the doctor

3. Exercise

Task 'Exercise' removed.

Task list:

1. Buy milk (modified)

2. Call the doctor
```