Python level 2

NATIXIS

PEA
Porto
Executive
Academy

Intermediate course
Class 1

Fábio Neves & Luis Dias

September 23

AGENDA

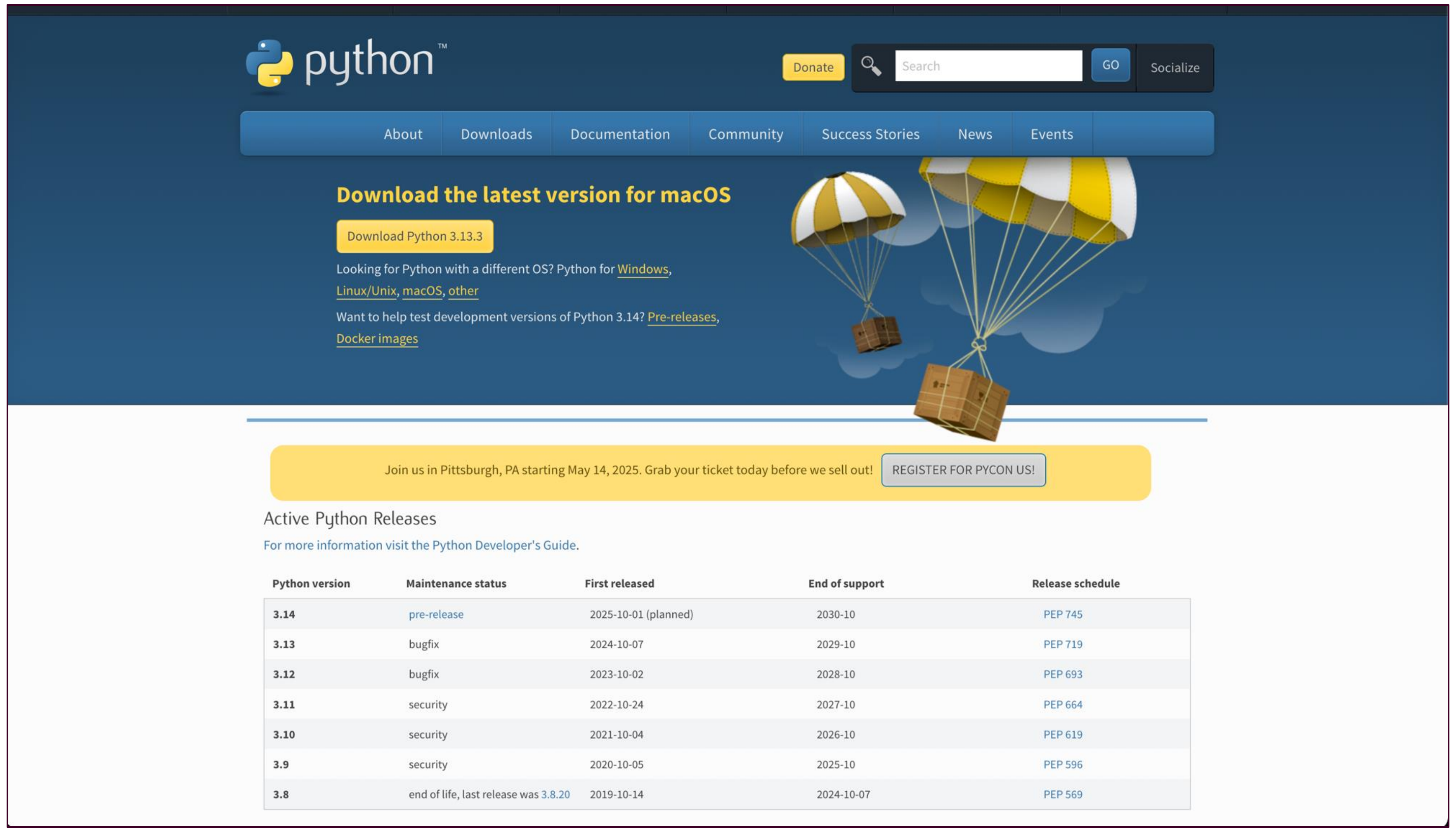
1. **Python functions**
    a. Defining basic functions
    b. Functions with return
    c. Functions with parameters
    d. Functions with standard parameters
2. **Comprehensions in Python**
    a. List comprehension
    b. Dictionary comprehension
    c. Set comprehension
    d. Generator comprehension

The setup

The basis – installing Python

Download a Python:

1. Visit the official Python website: Python Downloads (https://www.python.org/downloads/);
2. Click on the "Downloads" tab and select the version suitable for your operating system (Windows, macOS, or Linux);
3. Install the latest Python version.



Download the latest version for macOS

Looking for Python with a different OS? Python for Windows, Linux/Unix, macOS, other

Want to help test development versions of Python 3.14? Pre-releases, Docker images

Join us in Pittsburgh, PA starting May 14, 2025. Grab your ticket today before we sell out!

Active Python Releases

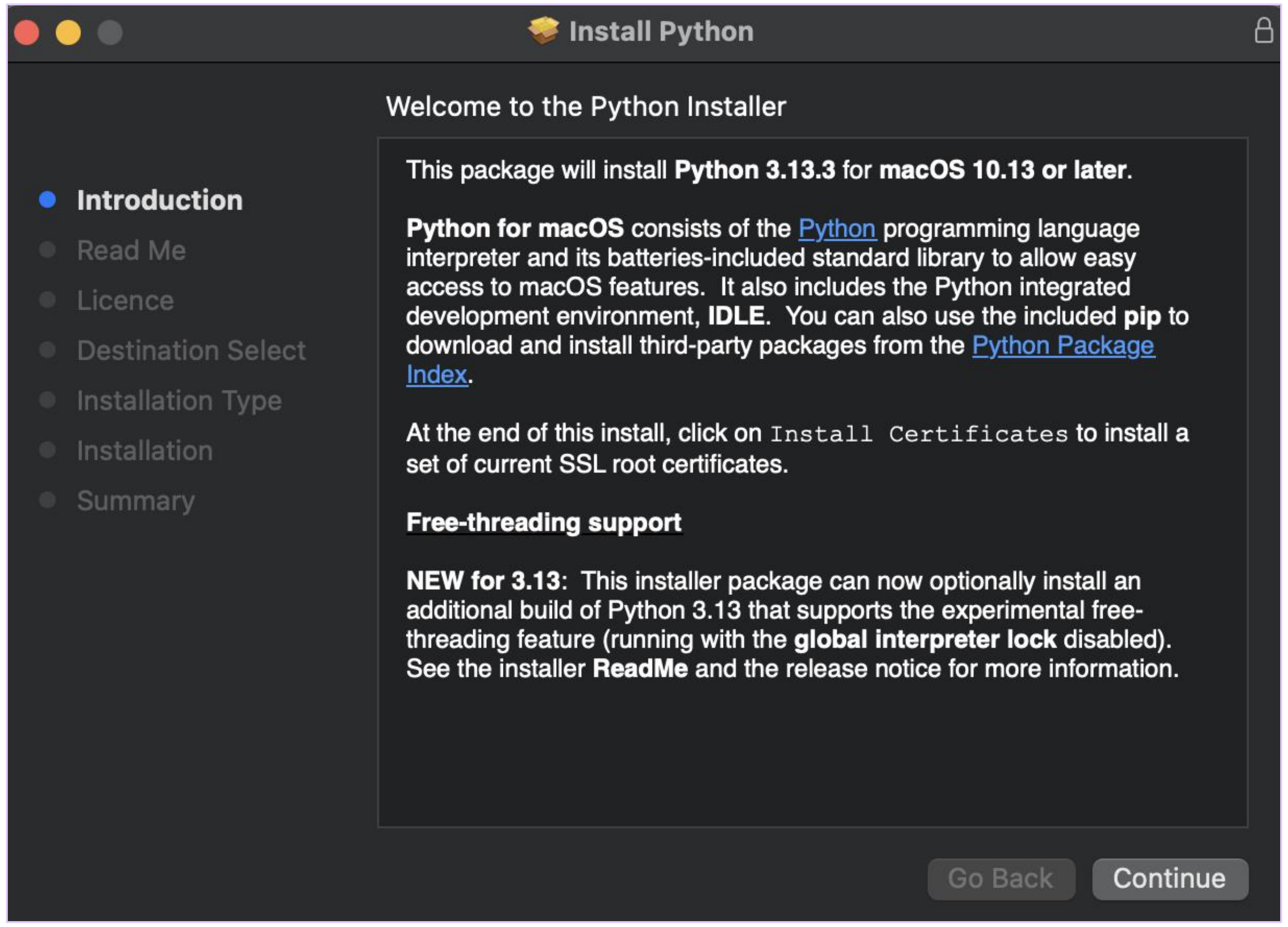
For more information visit the Python Developer's Guide.

Python version	Maintenance status	First released	End of support	Release schedule
3.14	pre-release	2025-10-01 (planned)	2030-10	PEP 745
3.13	bugfix	2024-10-07	2029-10	PEP 719
3.12	bugfix	2023-10-02	2028-10	PEP 693
3.11	security	2022-10-24	2027-10	PEP 664
3.10	security	2021-10-04	2026-10	PEP 619
3.9	security	2020-10-05	2025-10	PEP 596
3.8	end of life, last release was 3.8.20	2019-10-14	2024-10-07	PEP 569

The basis – installing Python

Run the Installer:

1. For **Windows**: Double-click the downloaded installer (.exe) and follow the installation wizard;
2. For **macOS**: Double-click the downloaded installer (.pkg) and follow the installation instructions;
3. For **Linux**: Open a terminal and navigate to the directory where you downloaded the installer. Run the command: sudo dpkg -i <installer_filename>.



Welcome to the Python Installer

This package will install **Python 3.13.3** for **macOS 10.13 or later**.

Python for macOS consists of the Python programming language interpreter and its batteries-included standard library to allow easy access to macOS features. It also includes the Python integrated development environment, **IDLE**. You can also use the included **pip** to download and install third-party packages from the Python Package Index.

At the end of this install, click on `Install Certificates` to install a set of current SSL root certificates.

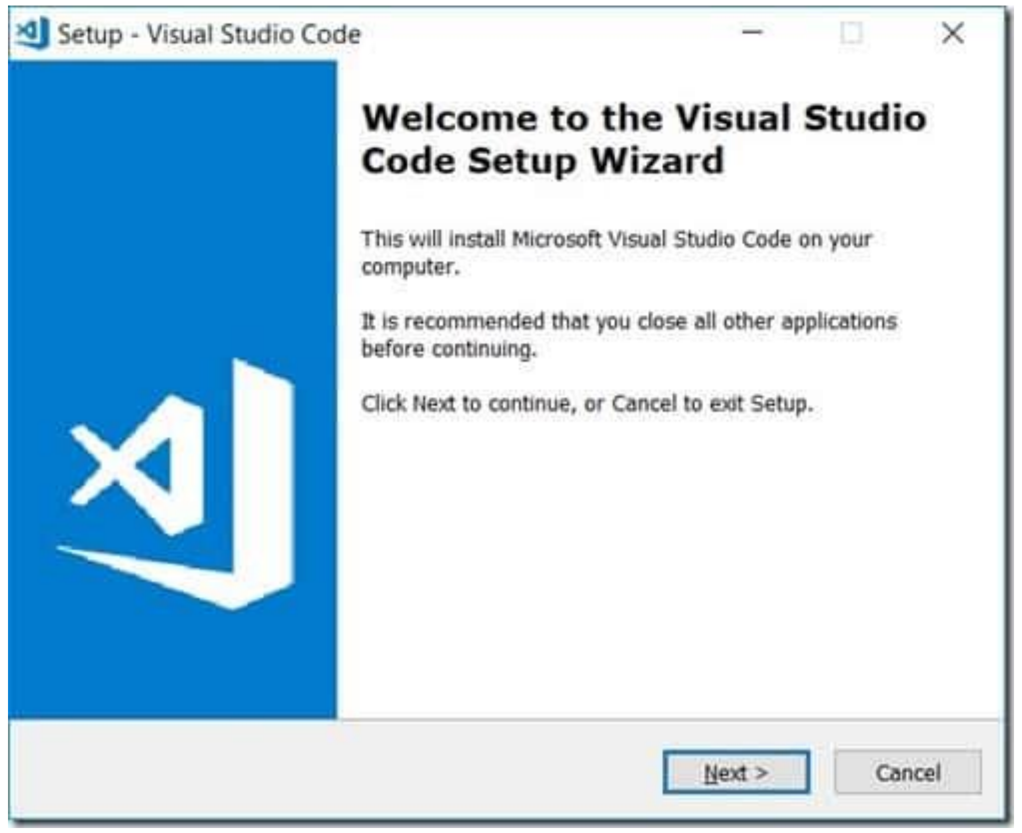
Free-threading support

NEW for 3.13: This installer package can now optionally install an additional build of Python 3.13 that supports the experimental free-threading feature (running with the **global interpreter lock** disabled). See the installer **ReadMe** and the release notice for more information.

The basis – installing VS Code on Windows

Install VS Code

- **For windows**
  - Run the downloaded .exe installer.
  - Accept the license agreement.
  - Choose the installation location (default is fine for most users).
  - **Recommended:** Check the following options during setup:
    - Add "Open with Code" to context menu
    - Register Code as editor for supported file types
  - Click Install.



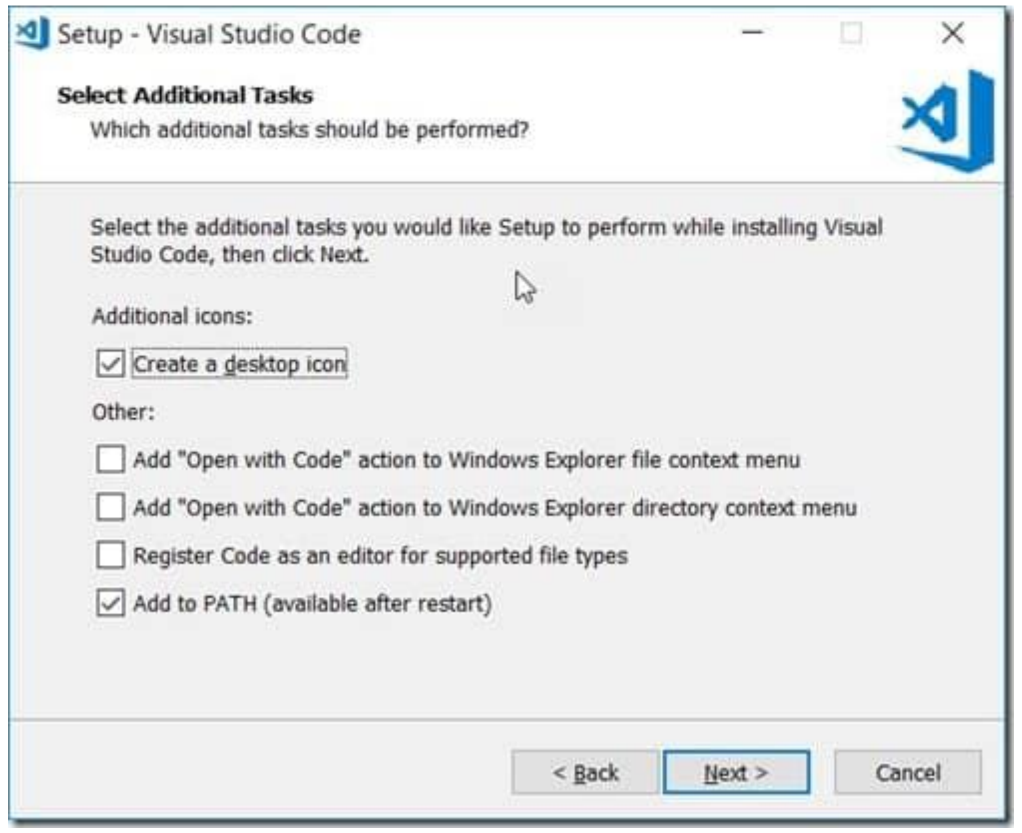
Setup - Visual Studio Code

Welcome to the Visual Studio Code Setup Wizard

This will install Microsoft Visual Studio Code on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.



Setup - Visual Studio Code

Select Additional Tasks

Which additional tasks should be performed?

Select the additional tasks you would like Setup to perform while installing Visual Studio Code, then click Next.

Additional icons:

- [✓] Create a desktop icon

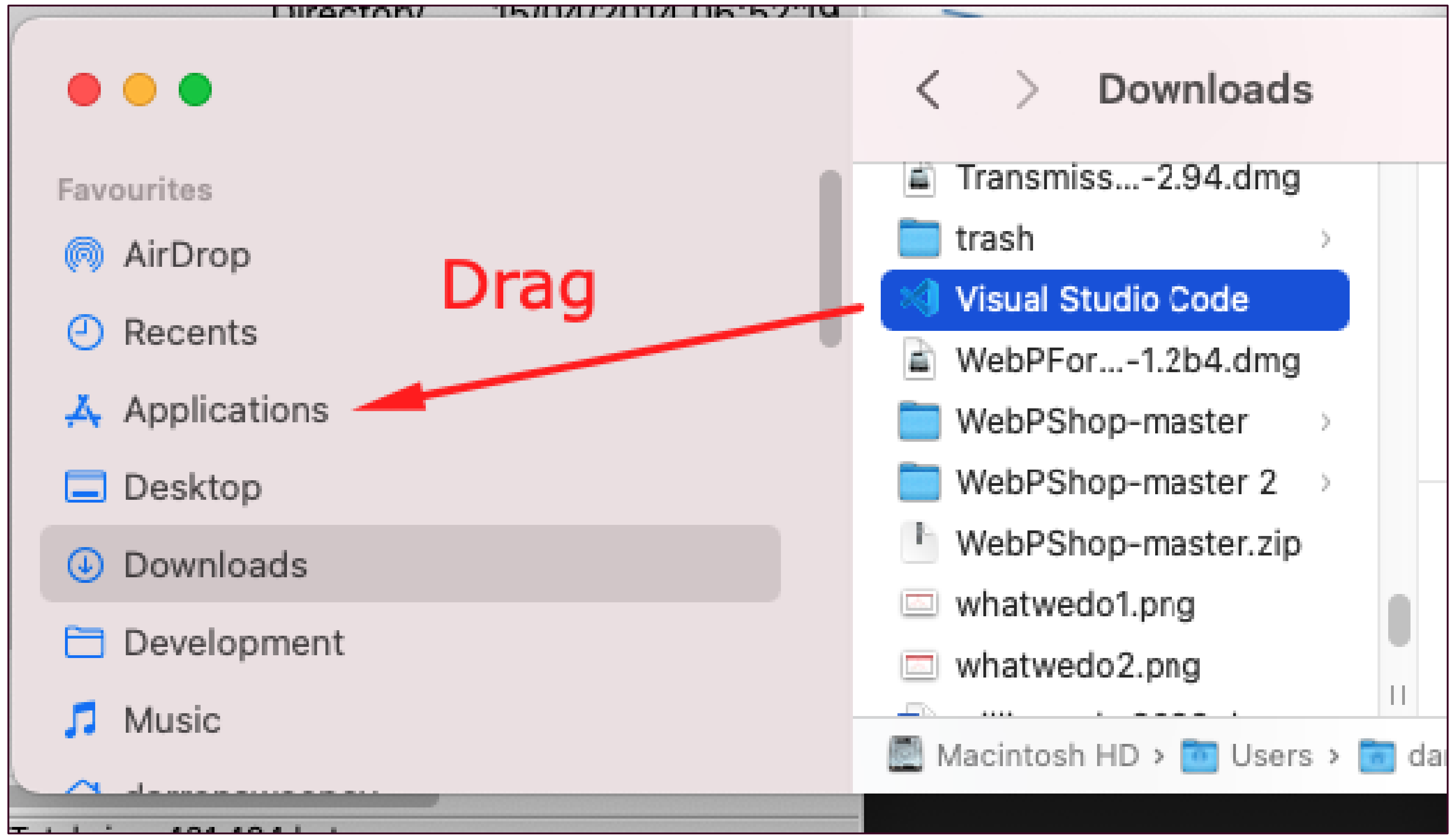
Other:

- [ ] Add "Open with Code" action to Windows Explorer file context menu
- [ ] Add "Open with Code" action to Windows Explorer directory context menu
- [ ] Register Code as an editor for supported file types
- [✓] Add to PATH (available after restart)

The basis – installing VS Code on MAC

Install VS Code

- **For MAC**
  - Open the downloaded .zip file.
  - Drag the Visual Studio Code app into your Applications folder.
  - You can optionally add VS Code to your Dock for easy access.



The basis – VS Code essentials

Set up a virtual environment

- **Create** the virtual environment: In the terminal and insider your project folder type.

```
terminal
cd path-to-your-project-folder python -m venv venv
```

- **Activate** the virtual environment.

Windows `.\venv\Scripts\Activate.ps1` **MAC/Linux** `source venv/bin/activate`

- After activation, your **terminal** will show the environment name at the beginning, like.

```
terminal
(venv) your-computer-name:your-project-folder username$
```

The basis – Virtual Environments

A **virtual environment** is like a **special, isolated space** on your computer where you can install Python packages **just for one project**, without messing with other projects or your system's Python.

Think of it like:

📦 A "project box" that keeps everything you need inside.

🛡️ It protects your project from problems like version conflicts (different projects needing different versions of the same package).

Example:

Project A needs **Pandas 3.2**.

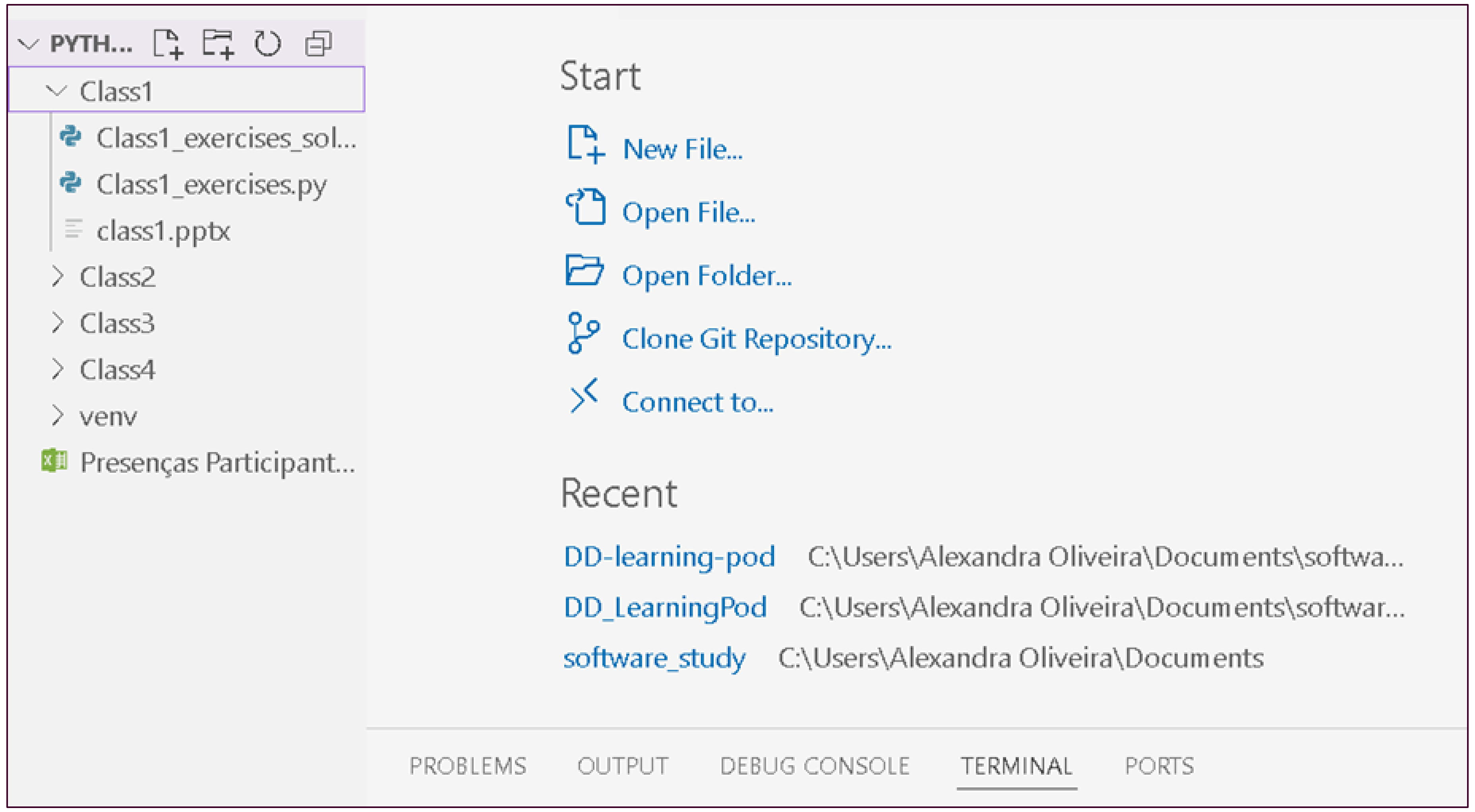
Project B needs **Pandas 4.**

If you use a virtual environment, **each project can have its own Pandas version**, no problems!

Without a virtual environment, everything would install globally, and projects could easily break each other.

The basis – Virtual Environments

An integrated development environment (IDE) is a software application that helps programmers develop software code efficiently. It increases developer productivity by combining capabilities such as software editing, building, testing, and packaging in an easy-to-use application. Just as writers use text editors and accountants use spreadsheets, software developers use IDEs to make their job easier.



VS Code is one of the best IDEs for programming in Python

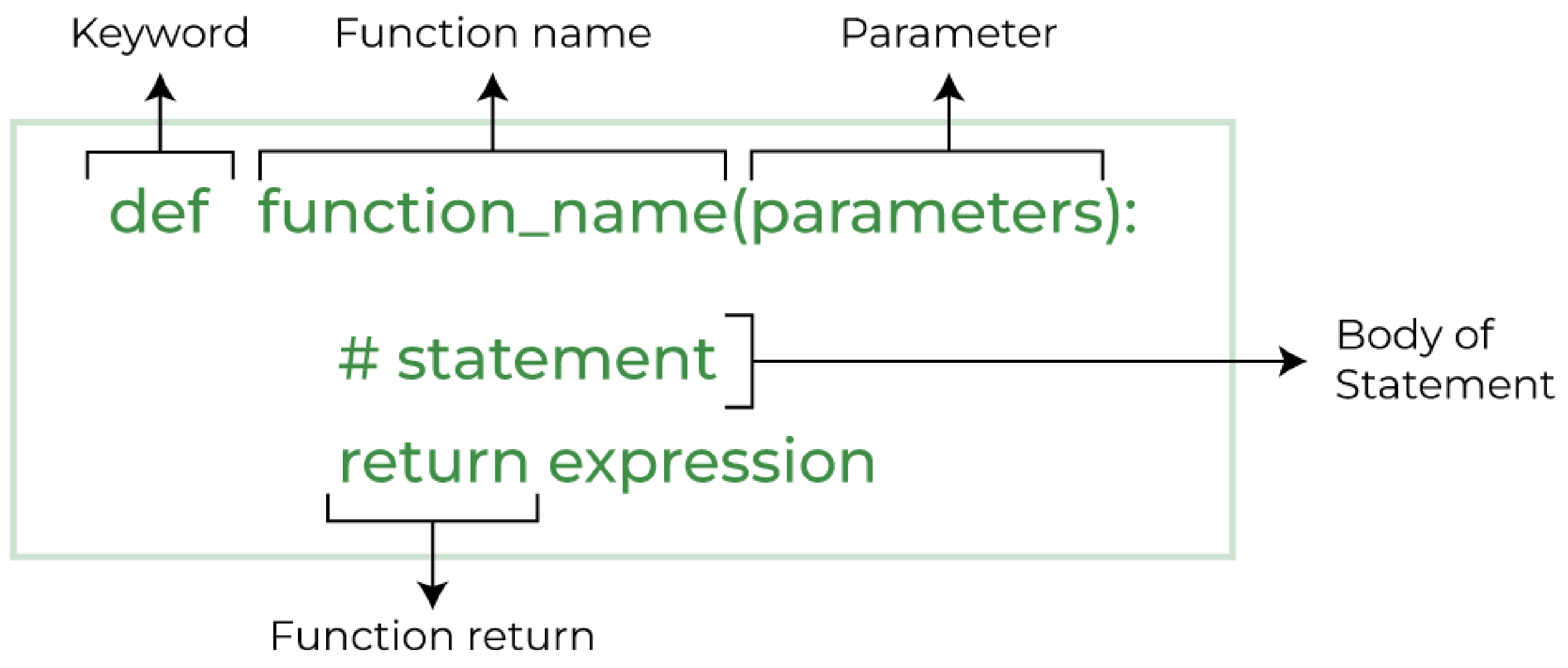
Python Functions

What is a Python Function?

- In Python a function is some reusable code that takes arguments(s) as input, does some computation, and then returns a result or results

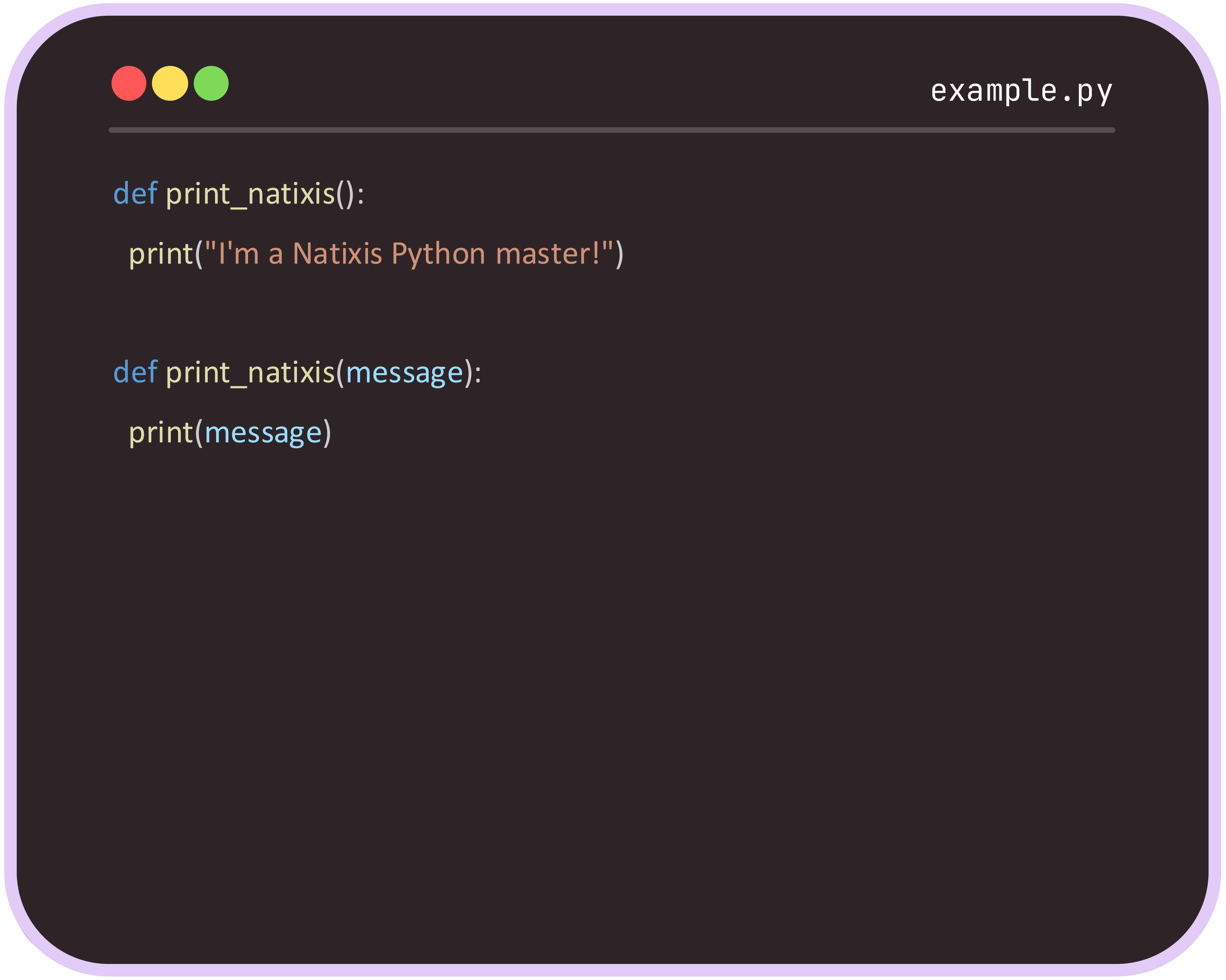
- We define a function using the def reserved word

- We call/invoke the function by using the function name, parentheses, and arguments in an expression



What is a Python Function?

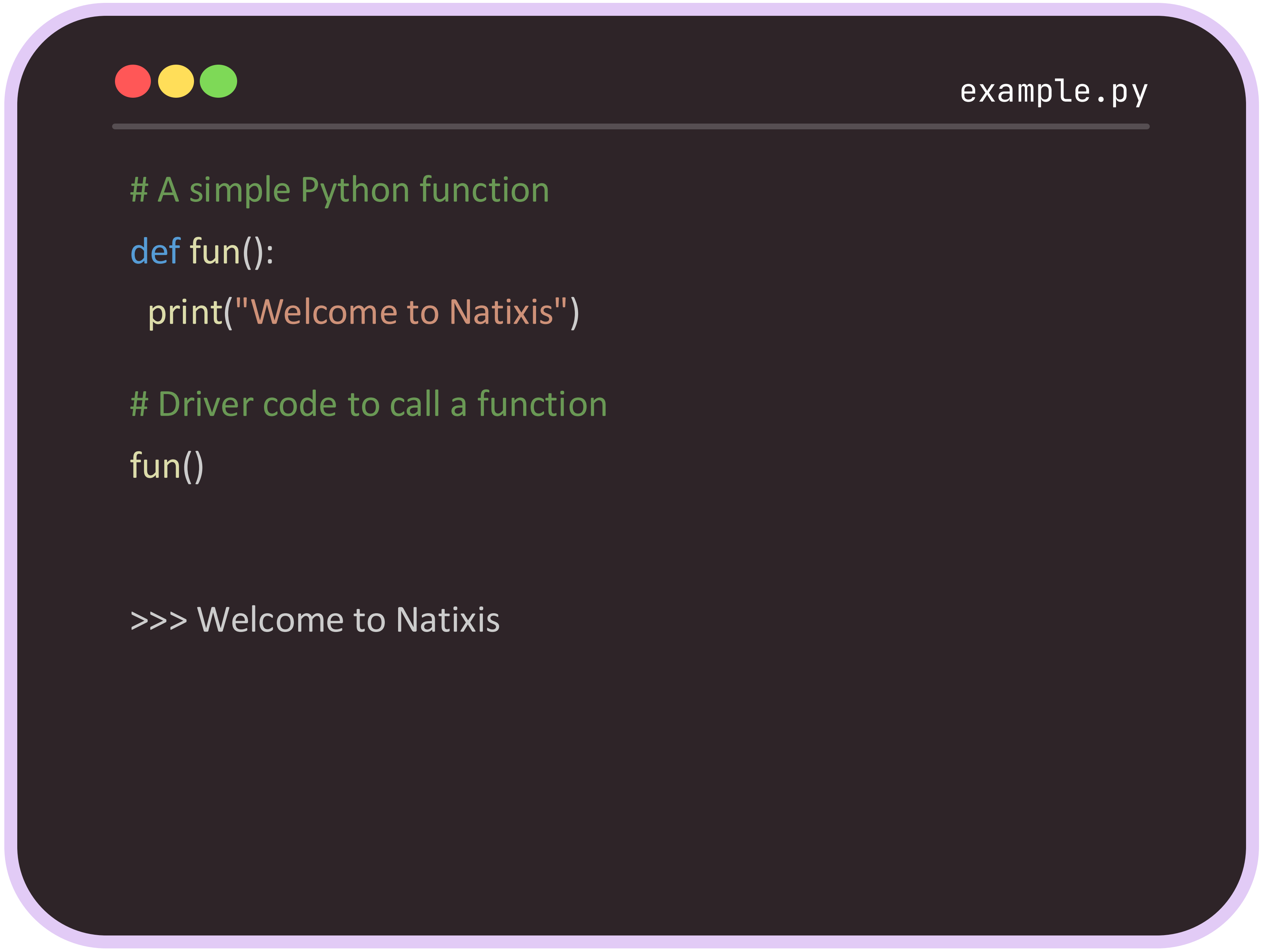
- We create a new function using the def keyword followed by optional parameters in parentheses
- We indent the body of the function
- This defines the function but does not execute the body of the function



```
def print_natixis():
 print("I'm a Natixis Python master!")


def print_natixis(message):
 print(message)
```

How to call a Python Function?

After creating a function, we can call it by using the name of the function followed by parenthesis containing parameters of that function.

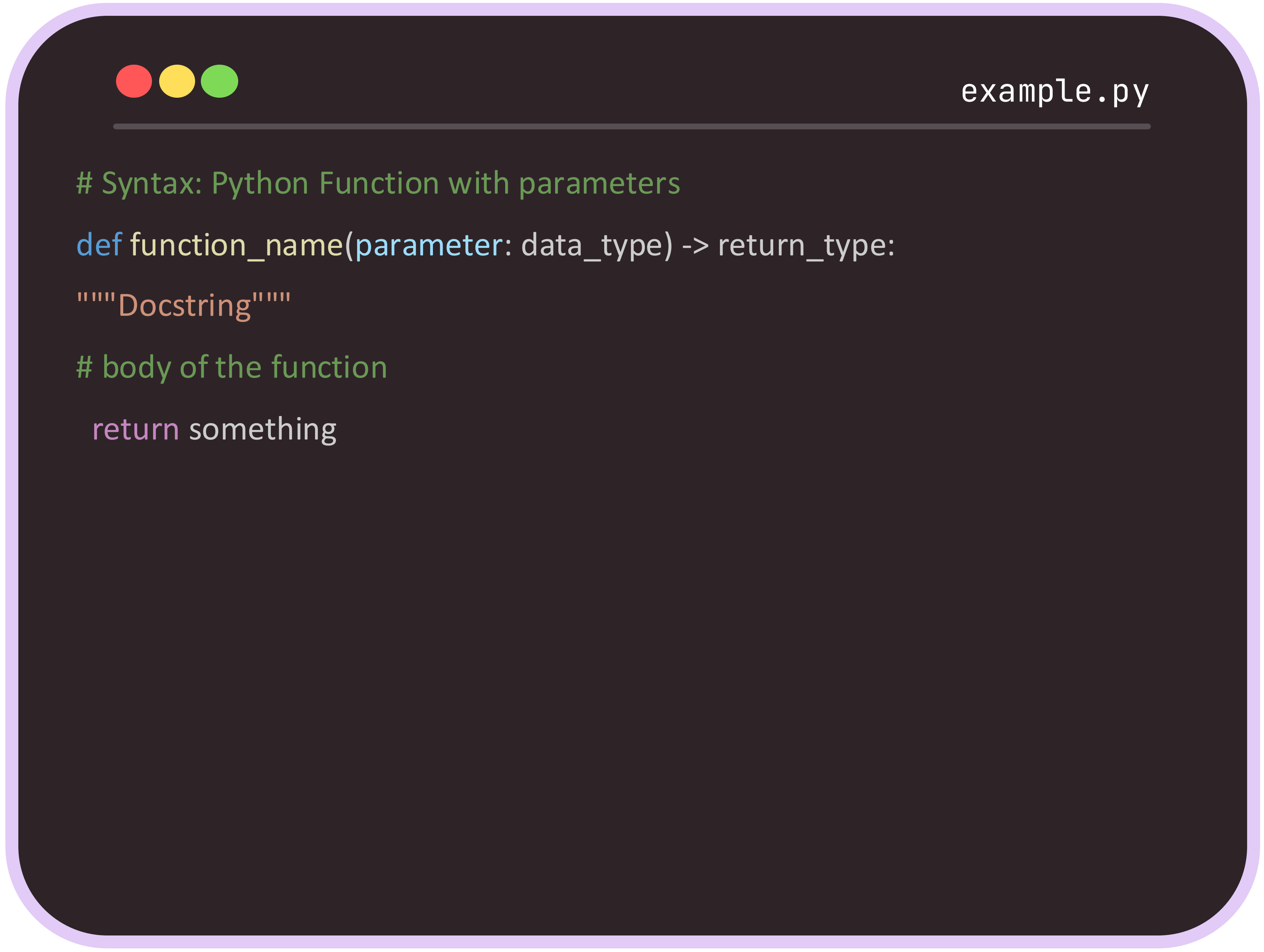


example.py

```
# A simple Python function
def fun():
  print("Welcome to Natixis")

# Driver code to call a function
fun()


>>> Welcome to Natixis
```

Function with parameters

If you have experience in C/C++ or Java then you must be thinking about the return type of the function and data type of arguments. That is possible in Python as well (specifically for Python 3.5 and above).

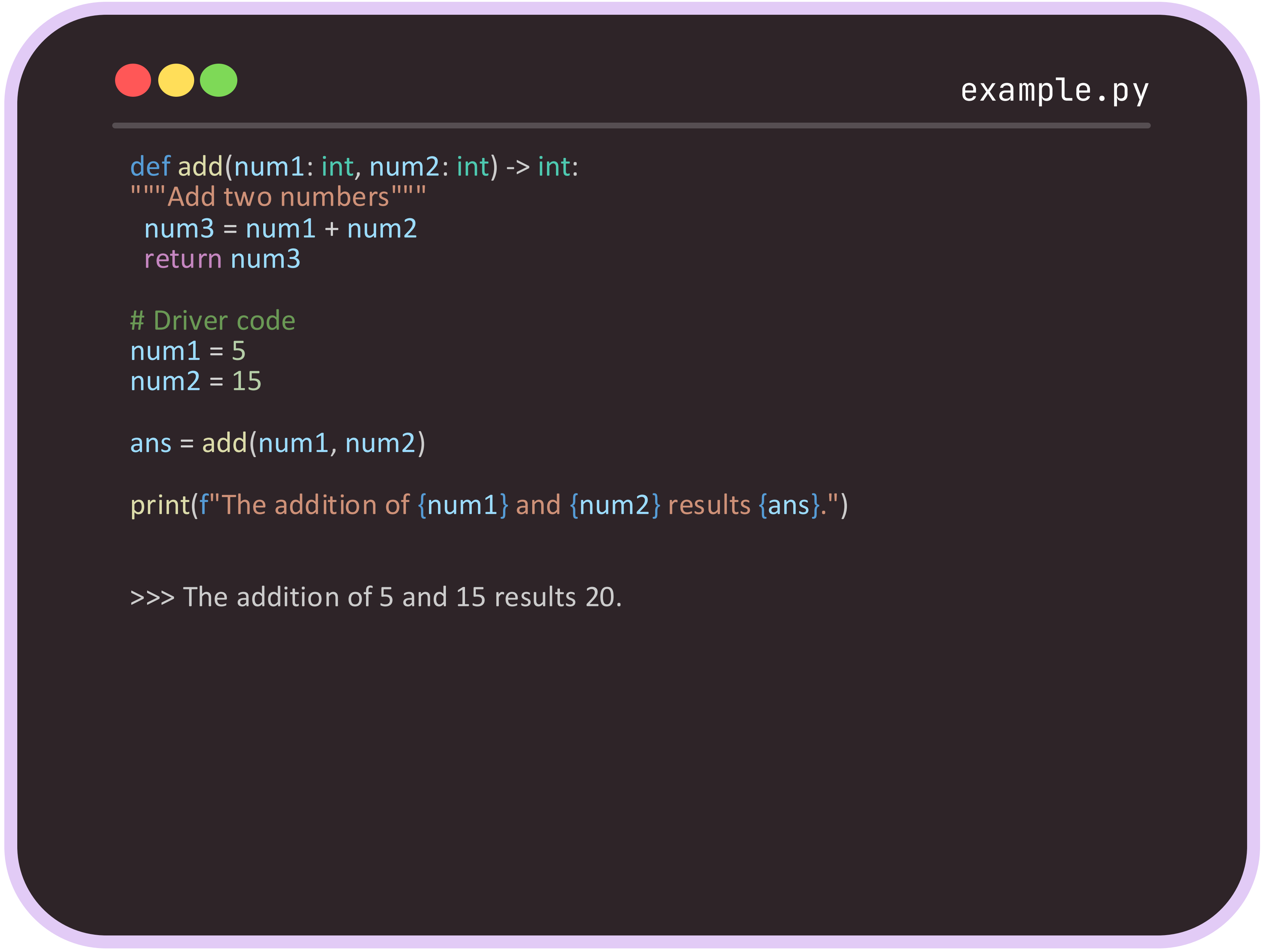


example.py

```
# Syntax: Python Function with parameters
def function_name(parameter: data_type) -> return_type:
"""Docstring"""
# body of the function
 return something
```

Function with parameters

The following example uses arguments that you will learn later in this session so you can come back on it again if not understood. It is defined here for people with prior experience in languages like C/C++ or Java.

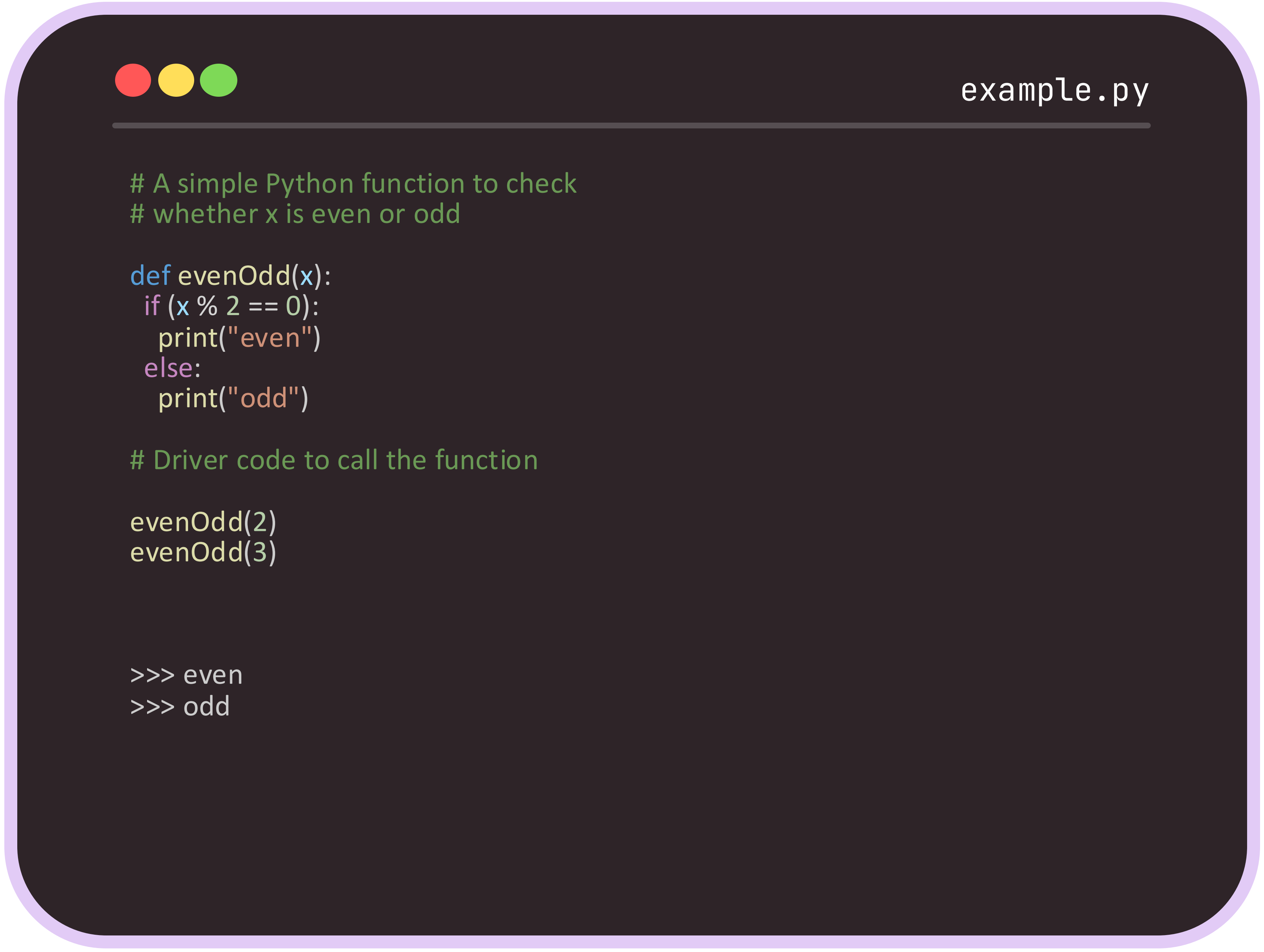


example.py

```python
def add(num1: int, num2: int) -> int:
"""Add two numbers"""
  num3 = num1 + num2
  return num3

# Driver code
num1 = 5
num2 = 15

ans = add(num1, num2)

print(f"The addition of {num1} and {num2} results {ans}.")


>>> The addition of 5 and 15 results 20.
```

Function with parameters

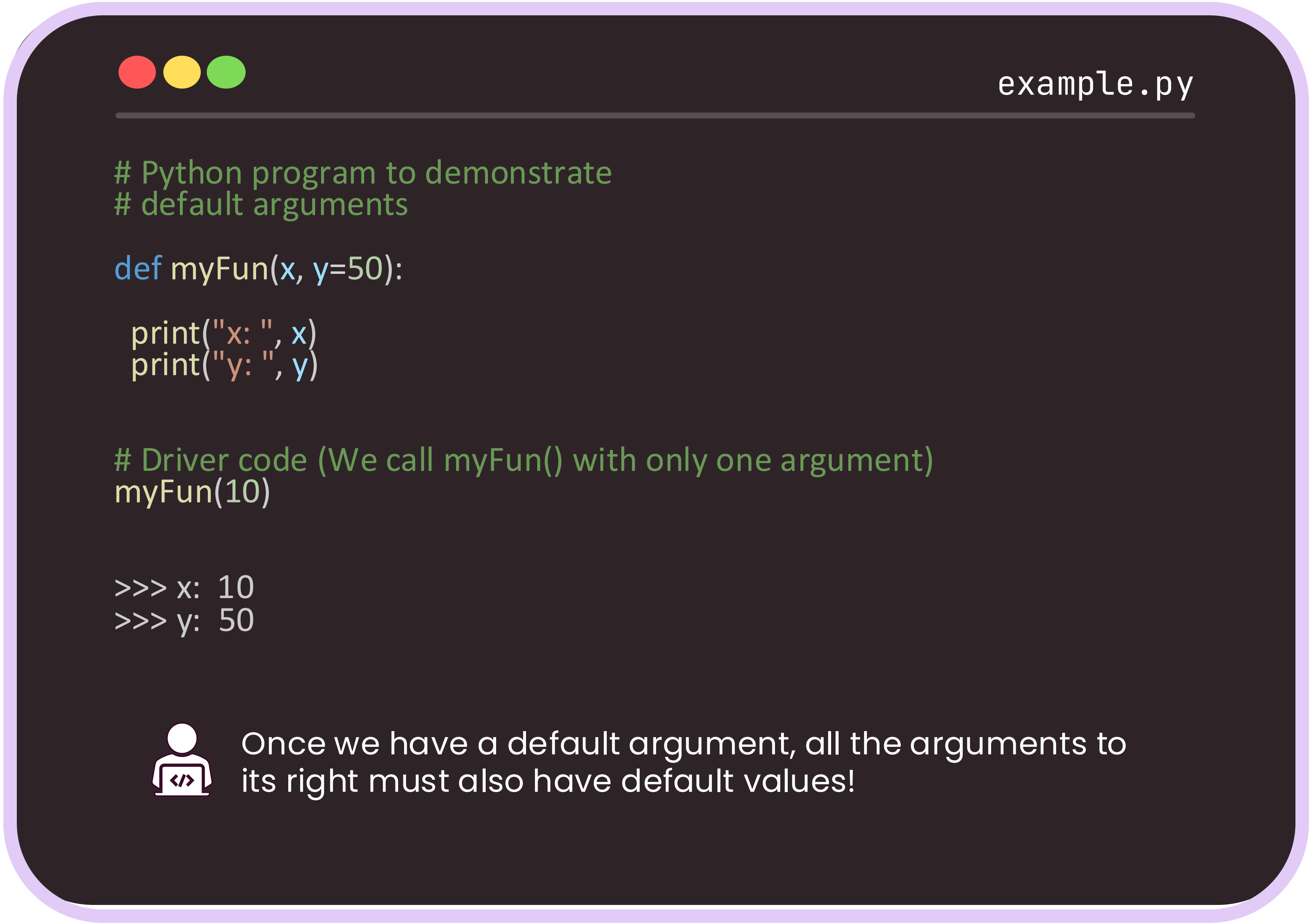
Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.



```python
# A simple Python function to check
# whether x is even or odd

def evenOdd(x):
  if (x % 2 == 0):
    print("even")
  else:
    print("odd")

# Driver code to call the function

evenOdd(2)
evenOdd(3)



>>> even
>>> odd
```

How to set default arguments values?

Python supports various types of arguments that can be passed at the time of the function call. A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.

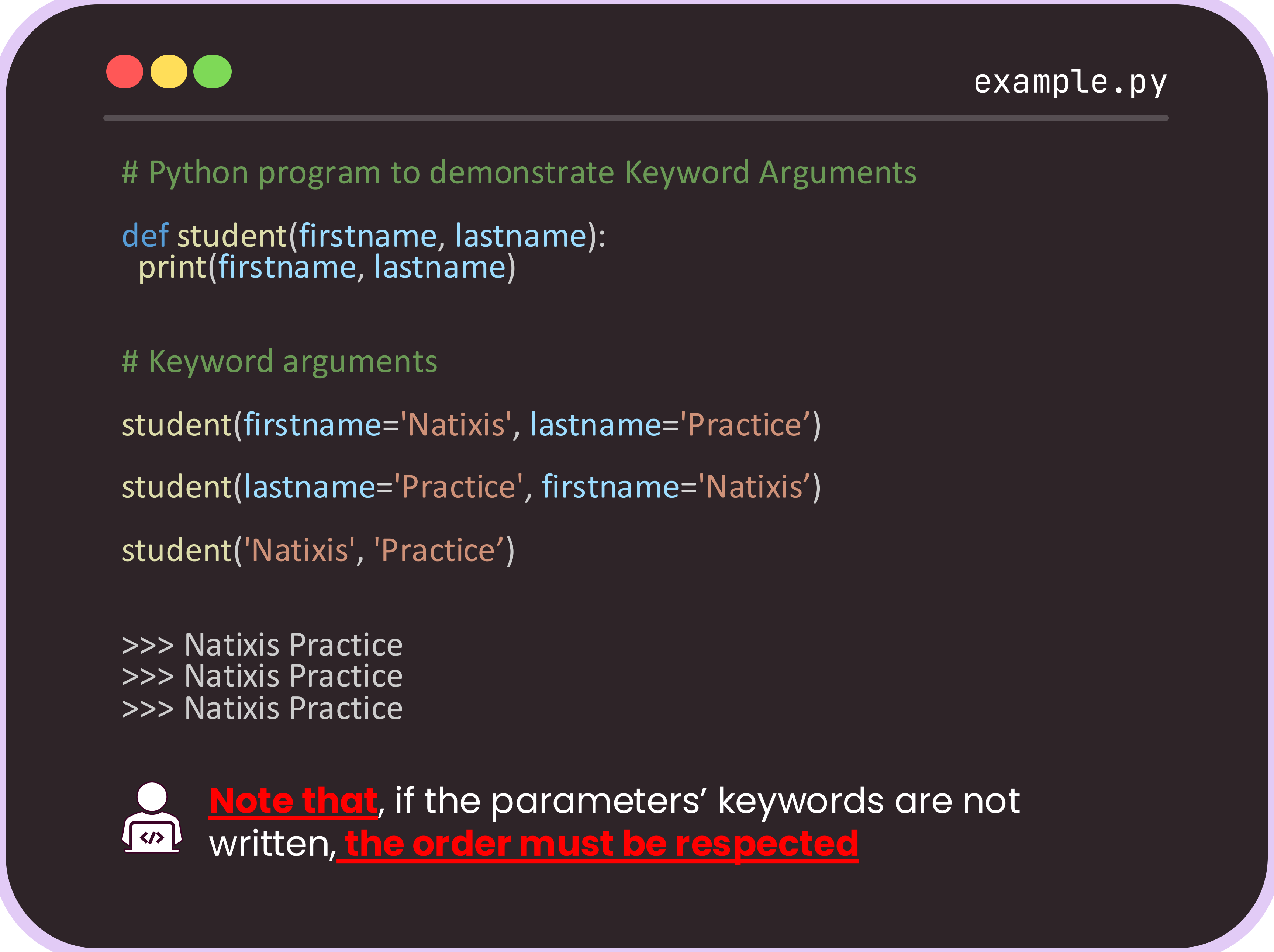

example.py

```
# Python program to demonstrate
# default arguments

def myFun(x, y=50):

  print("x: ", x)
  print("y: ", y)


# Driver code (We call myFun() with only one argument)
myFun(10)


>>> x:  10
>>> y:  50
```

Once we have a default argument, all the arguments to its right must also have default values!

How to use Keyword arguments?

The idea is to allow the caller to specify the argument name with values so that caller does not need to remember the order of parameters.



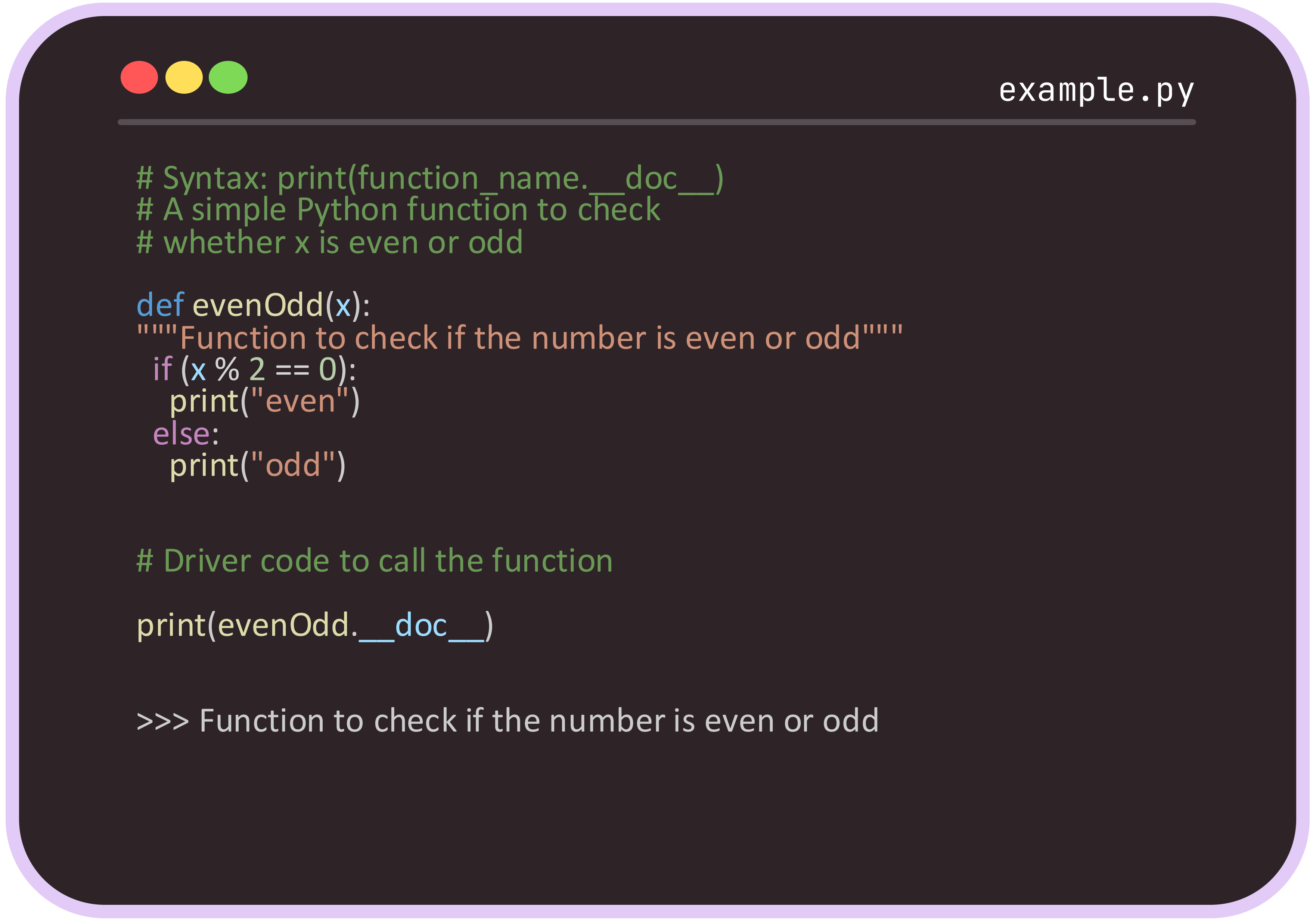
example.py

```
# Python program to demonstrate Keyword Arguments

def student(firstname, lastname):
  print(firstname, lastname)


# Keyword arguments

student(firstname='Natixis', lastname='Practice’)

student(lastname='Practice', firstname='Natixis’)

student('Natixis', 'Practice’)


>>> Natixis Practice
>>> Natixis Practice
>>> Natixis Practice
```

Note that, if the parameters’ keywords are not written, **the order must be respected**

Use Docstring everywhere

The first string after the function is called the Document string or **Docstring** in short. This is used to describe the functionality of the function. The use of docstring in functions is optional but it is considered a good practice.

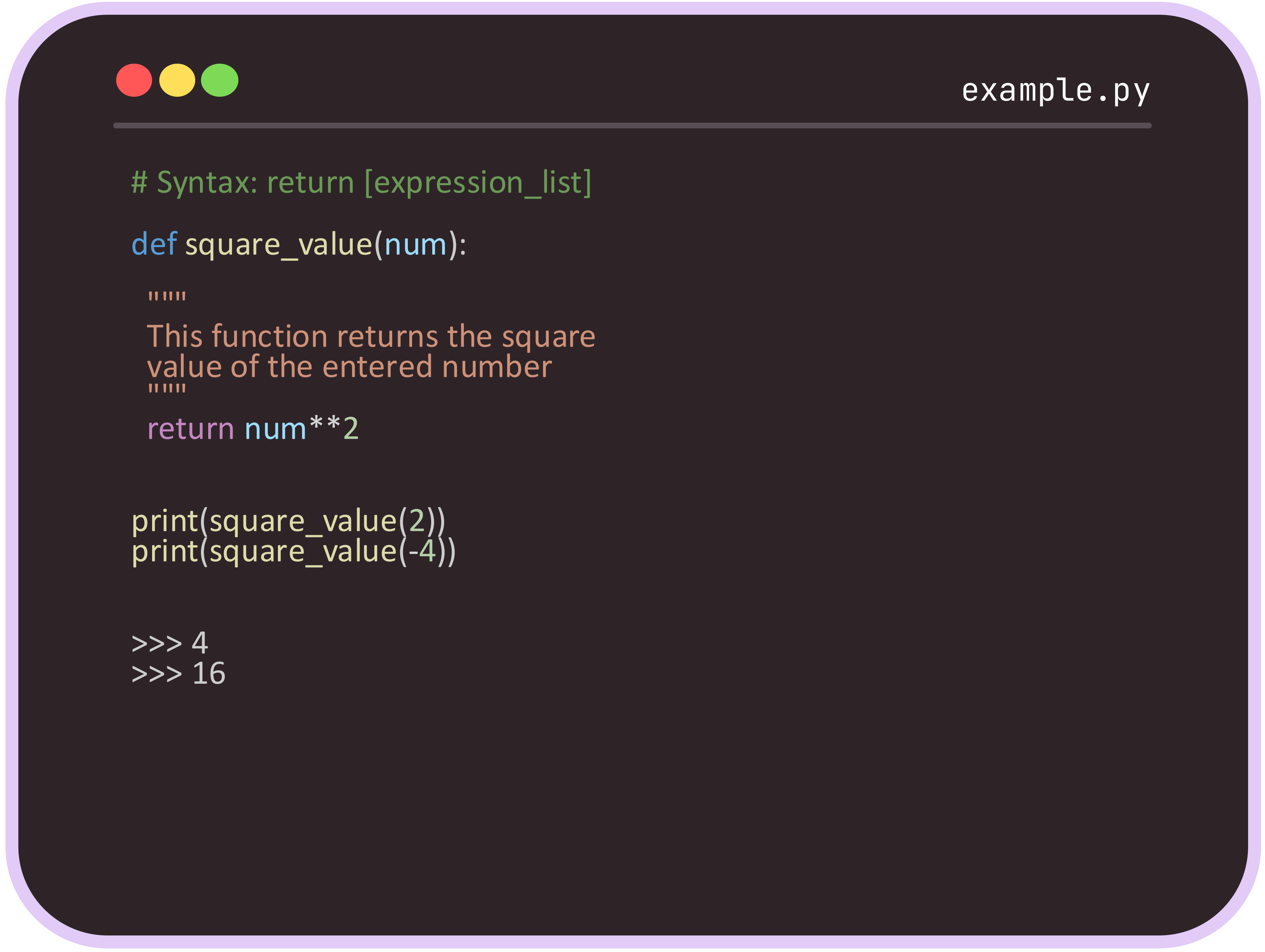


example.py

```
# Syntax: print(function_name.__doc__)
# A simple Python function to check
# whether x is even or odd

def evenOdd(x):
"""Function to check if the number is even or odd"""
  if (x % 2 == 0):
    print("even")
  else:
    print("odd")


# Driver code to call the function

print(evenOdd.__doc__)


>>> Function to check if the number is even or odd
```

How to use the “return” statement?

The function return statement is used to exit from a function and go back to the function caller and return the specified value or data item to the caller.

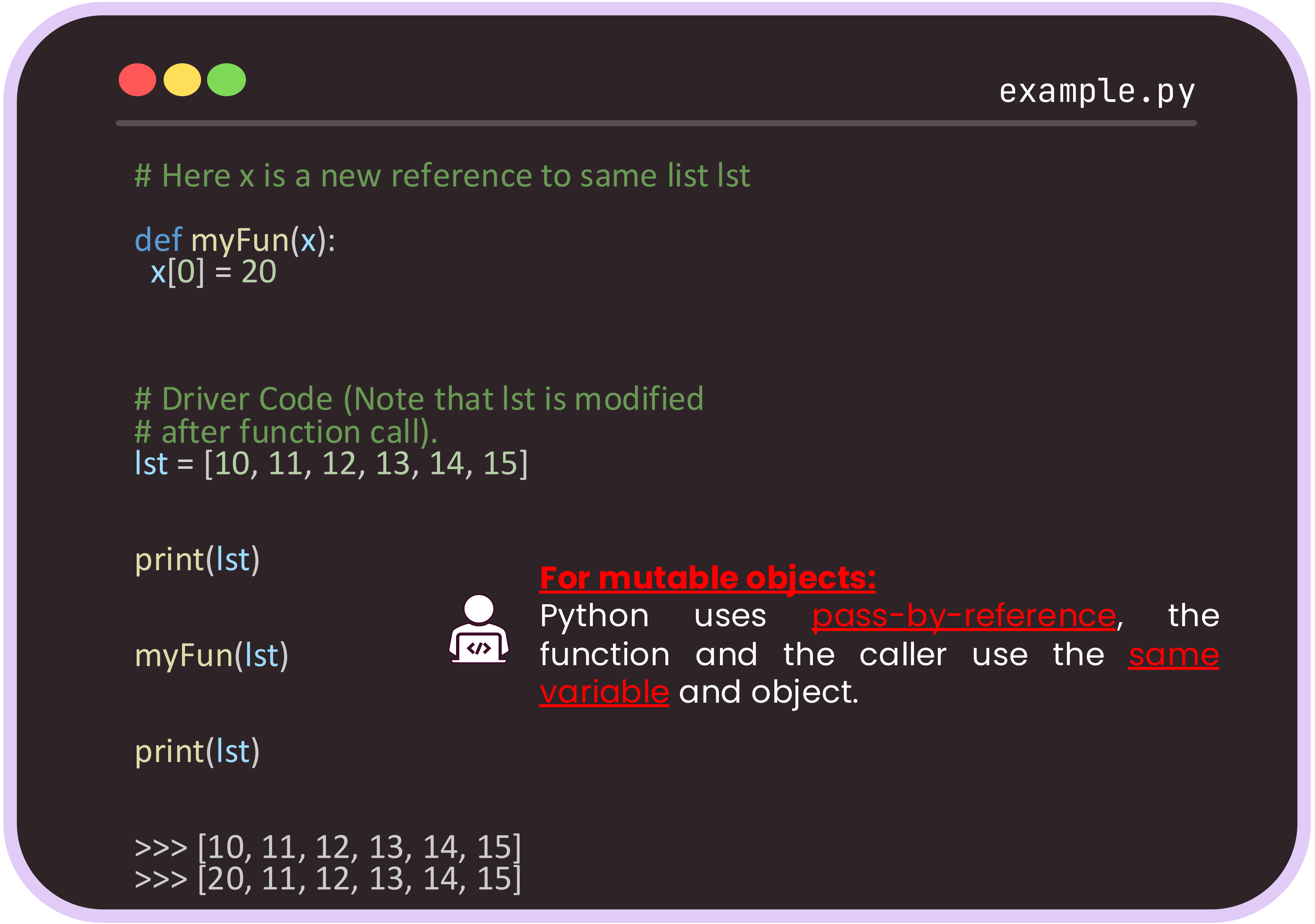


example.py

```
# Syntax: return [expression_list]

def square_value(num):
  """
  This function returns the square
  value of the entered number
  """
  return num**2


print(square_value(2))
print(square_value(-4))


>>> 4
>>> 16
```

How to pass by reference or pass by value?

One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function, a new reference to the object is created.



example.py

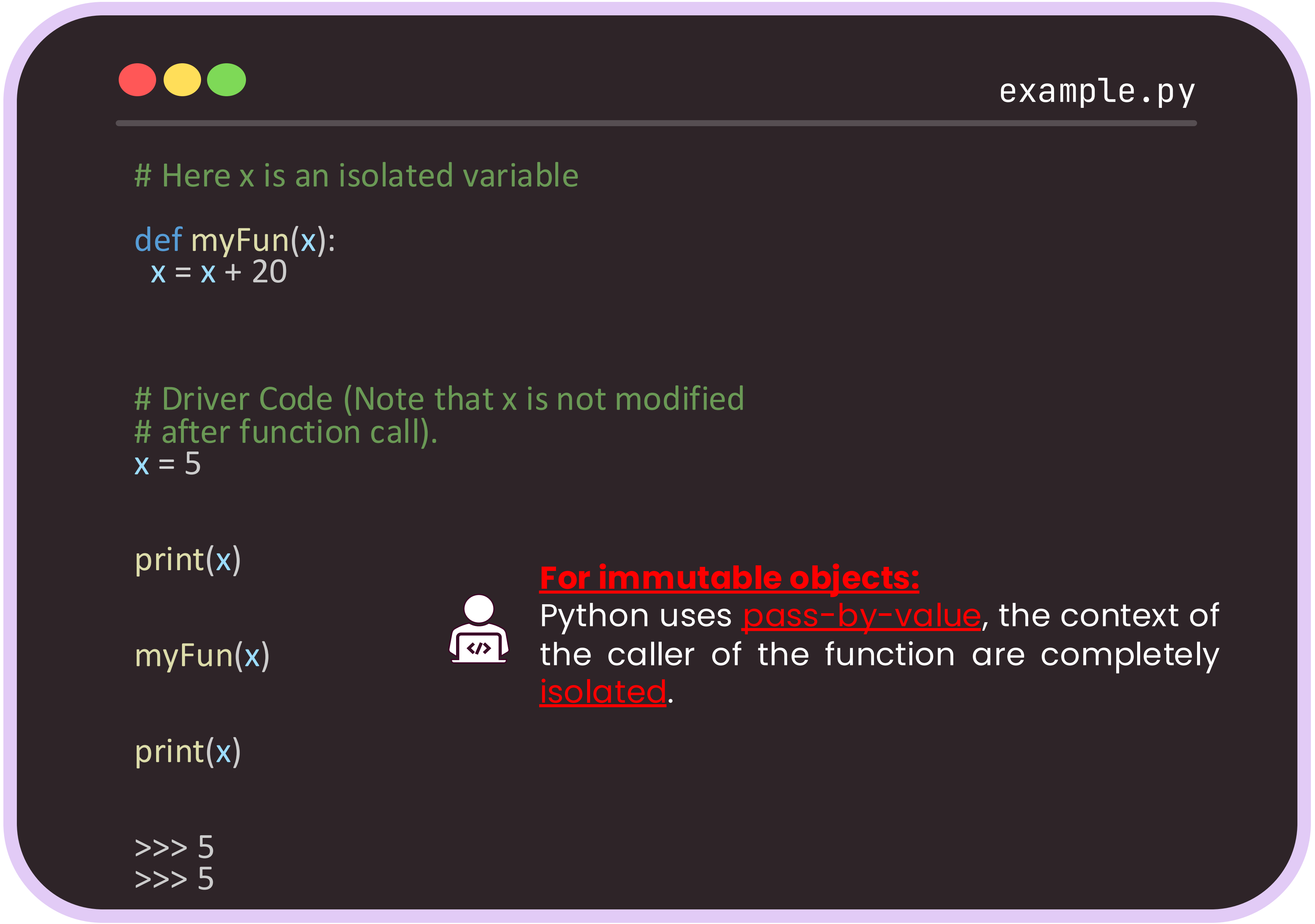
```
# Here x is a new reference to same list lst

def myFun(x):
  x[0] = 20



# Driver Code (Note that lst is modified
# after function call).
lst = [10, 11, 12, 13, 14, 15]


print(lst)


myFun(lst)


print(lst)


>>> [10, 11, 12, 13, 14, 15]
>>> [20, 11, 12, 13, 14, 15]
```

For mutable objects:
Python uses pass-by-reference, the function and the caller use the same variable and object.

Source: https://www.geeksforgeeks.org/is-python-call-by-reference-or-call-by-value/

How to pass by reference or pass by value?

One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function, a new reference to the object is created.



```
example.py

# Here x is an isolated variable

def myFun(x):
 x = x + 20



# Driver Code (Note that x is not modified
# after function call).
x = 5


print(x)


myFun(x)


print(x)


>>> 5
>>> 5
```

For immutable objects:
Python uses pass-by-value, the context of the caller of the function are completely isolated.

Source: https://www.geeksforgeeks.org/is-python-call-by-reference-or-call-by-value/

Which are the mutable objects?

Data type	Built-in Class	Mutable
Numbers	int, float, complex	**No**
Strings	str	**No**
Tuples	tuple	**No**
Bytes	bytes	**No**
Booleans	bool	**No**
Frozen sets	frozenset	**No**
Lists	list	**Yes**
Dictionaries	Dict	**Yes**
Sets	Set	**Yes**
Data Frames	DataFrame	**Yes**

Source: https://realpython.com/python-mutable-vs-immutable-types/

Why to use a Python Function?

Reuse:

```
#collect input from user
celsius = float(input("Enter Celsius value: "))
#calculate value in Fahrenheit
Fahrenheit = (celsius*1.8) + 32
print("Fahrenheit value is ",fahrenheit)
```

Program to calculate *Fahrenheit*

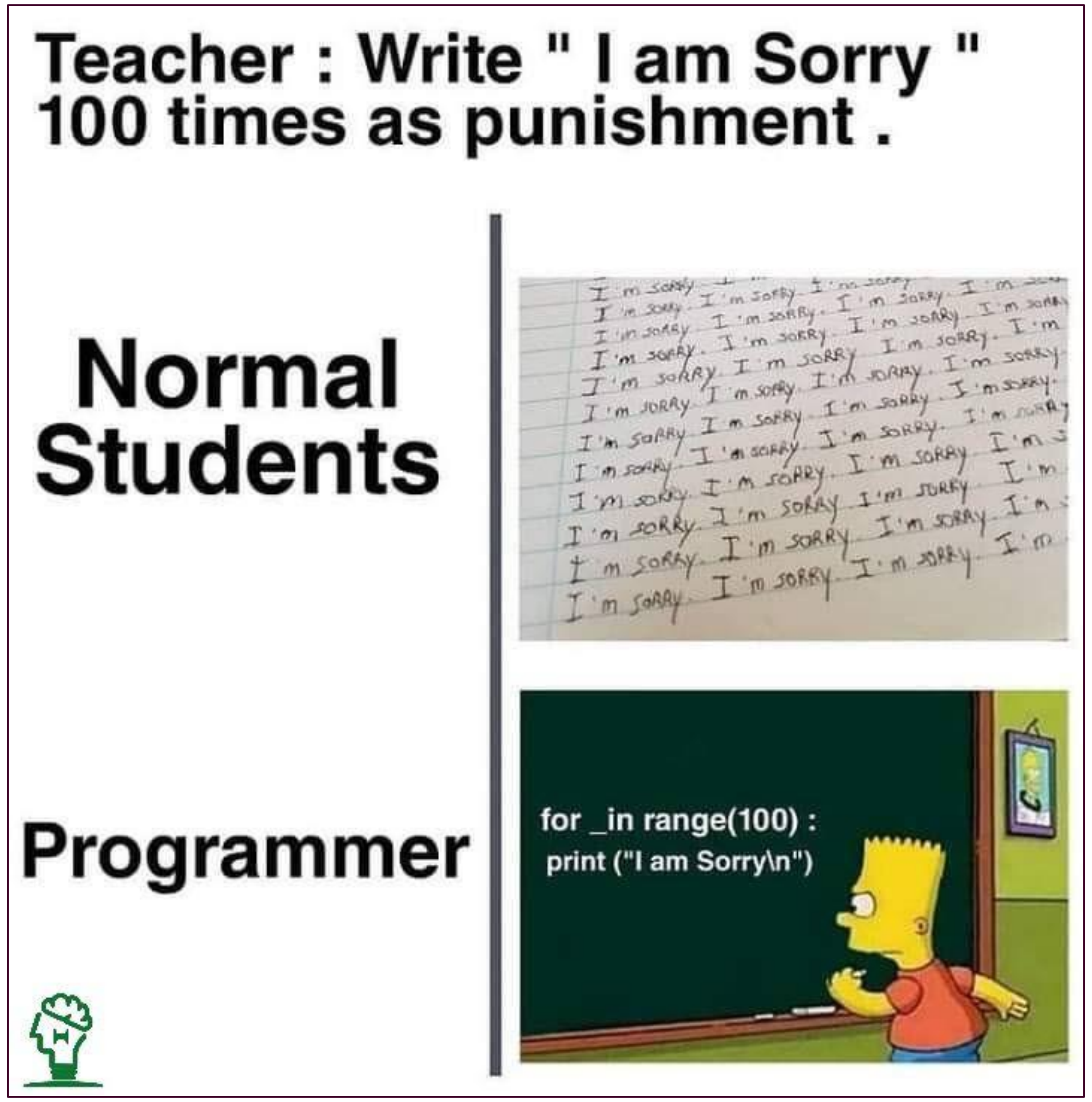
Fahrenheit = (9/5)Celsius + 32

Logic to calculate *Fahrenheit*

```
#collect input from user
celsius = float(input("Enter Celsius value: "))
#calculate value in Fahrenheit
Fahrenheit = (celsius*1.8) + 32
print("Fahrenheit value is ",fahrenheit)
```

You wouldn't want to **repeat** those **same lines of code** every time a value needed conversion

To function or not to function....

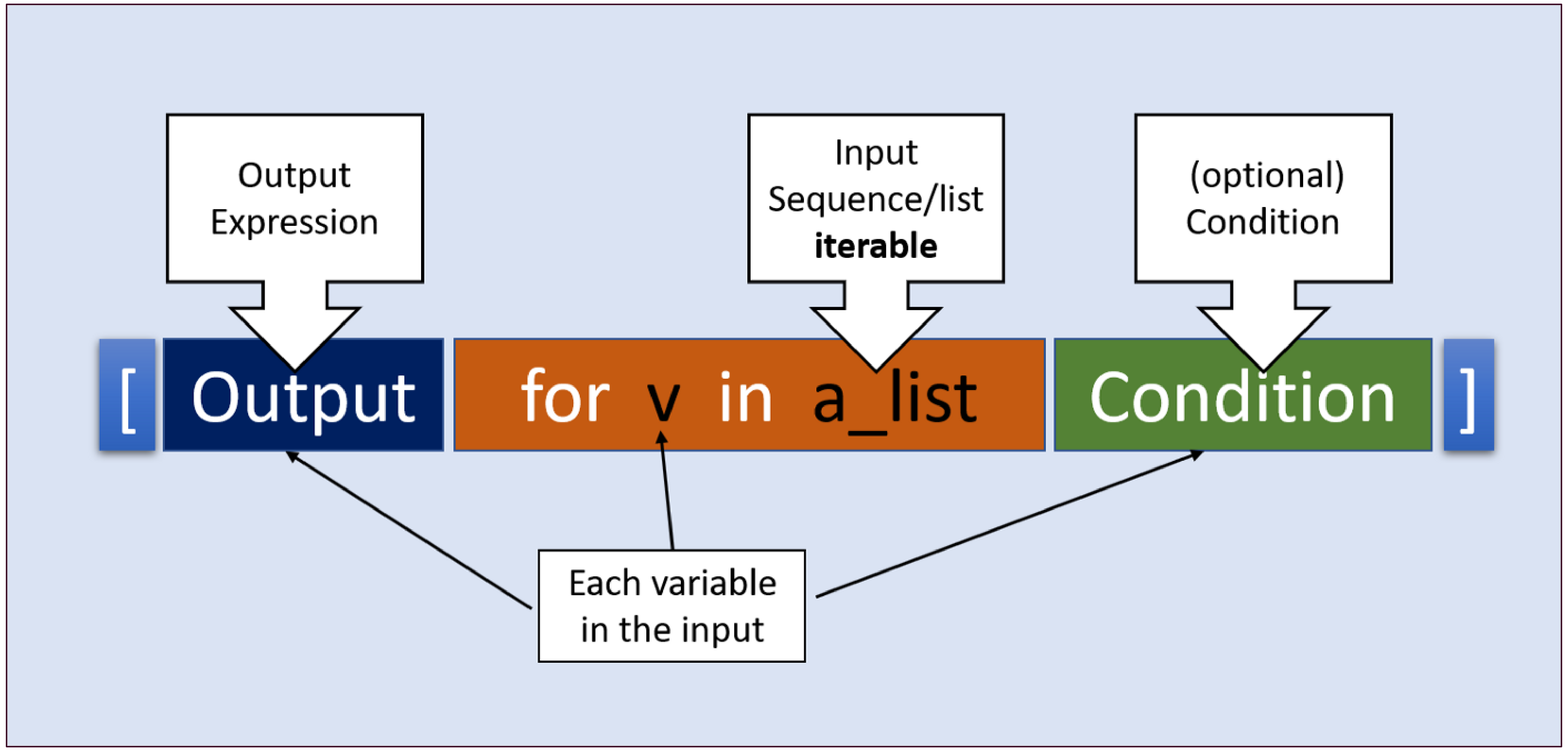

Teacher : Write " I am Sorry " 100 times as punishment .
Normal Students
Programmer
I'm sorry. I'm sorry. I'm sorry.
for _in range(100) :
print ("I am Sorry\n")

- Organize your code into "paragraphs" - capture a complete thought and "name it"
- Don't repeat yourself - make it work once and then reuse it
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions
- Make a library of common stuff that you do over and over

Comprehensions in Python

What is a comprehension?

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined. Python supports the following 4 types of comprehensions:


Output Expression
Input Sequence/list **iterable**
(optional) Condition
[Output for v in a_list Condition]
Each variable in the input

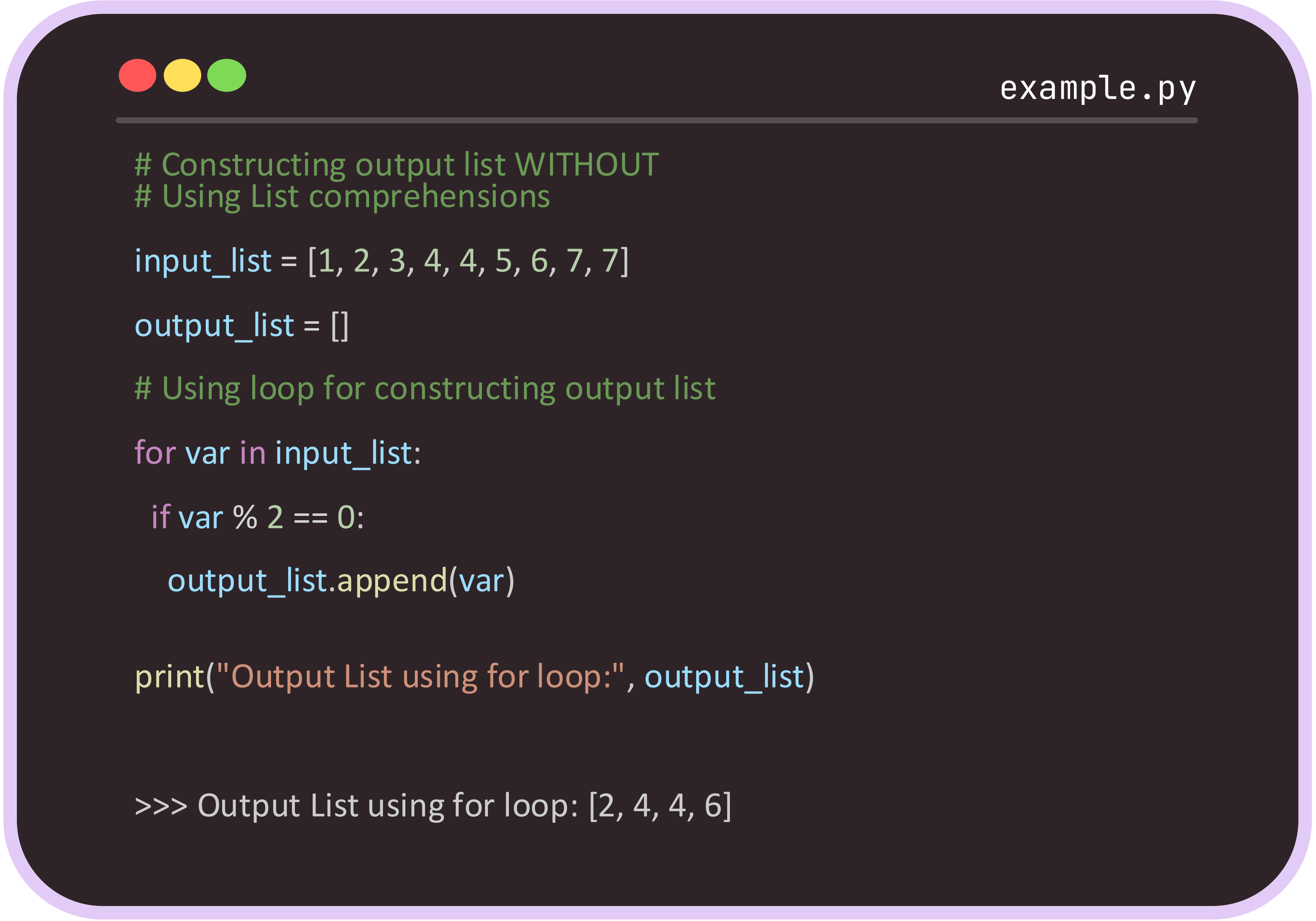
What is a comprehension?

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined. Python supports the following 4 types of comprehensions:

Comprehension type	Syntax
List	output_list = [output_exp for var in input_list if (var satisfies this condition)]
Dictionary	output_dict = {key:value for (key, value) in iterable if (key, value satisfy this condition)}
Set	output_set = {output_exp for var in input_list if (var satisfies this condition)}
Generator	output_generator = (output_exp for var in input_list if (var satisfies this condition))

List Comprehension

List Comprehensions provide an elegant way to create new lists. The following is the basic structure of a list comprehension. Note that list comprehension may or may not contain an if condition. List comprehensions can contain multiple for (nested list comprehensions).

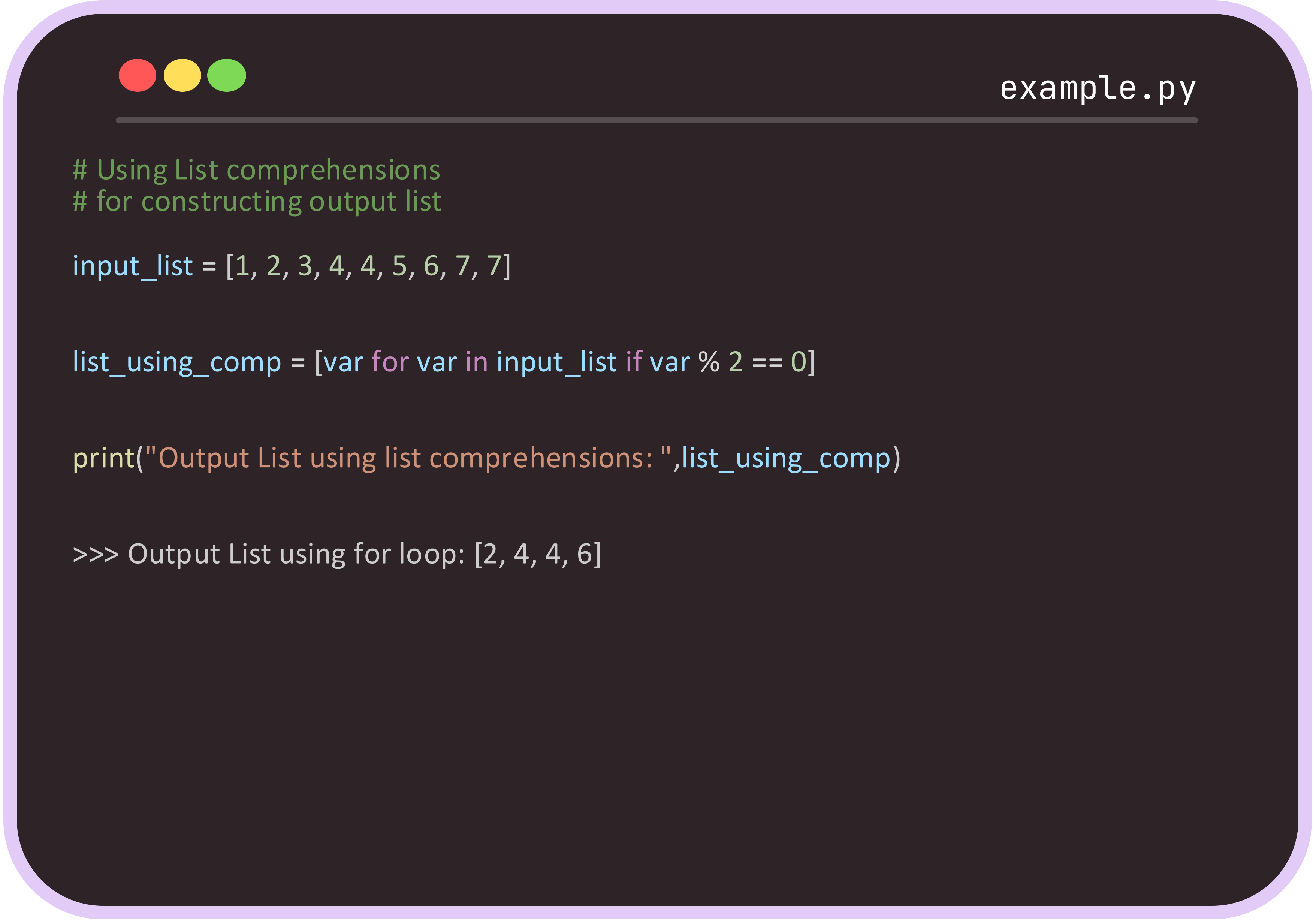


example.py

```
# Constructing output list WITHOUT
# Using List comprehensions

input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]

output_list = []

# Using loop for constructing output list

for var in input_list:

  if var % 2 == 0:

    output_list.append(var)


print("Output List using for loop:", output_list)


>>> Output List using for loop: [2, 4, 4, 6]
```

List Comprehension

List Comprehensions provide an elegant way to create new lists. The following is the basic structure of a list comprehension. Note that list comprehension may or may not contain an if condition. List comprehensions can contain multiple for (nested list comprehensions).

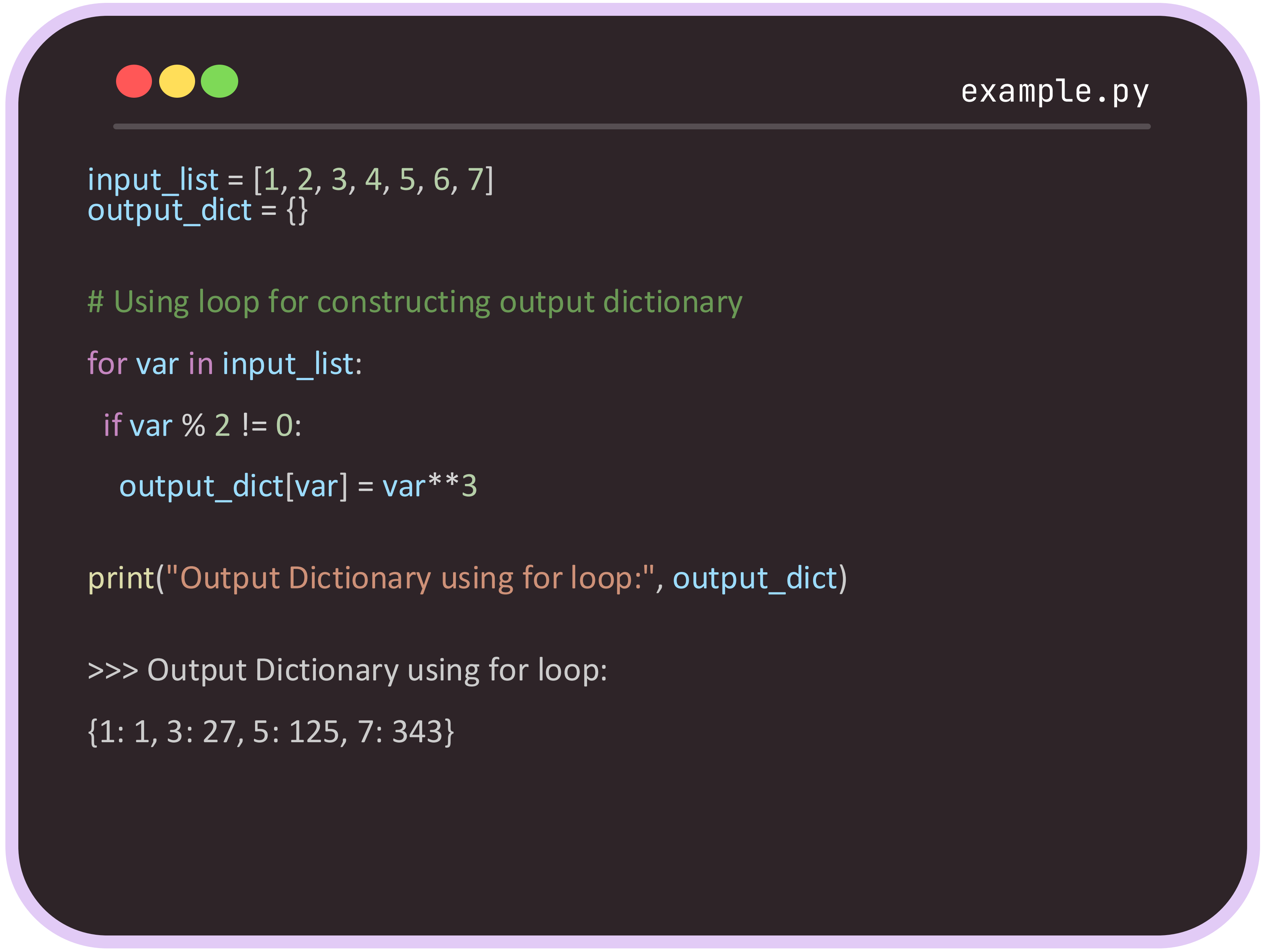


example.py

```
# Using List comprehensions
# for constructing output list

input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]


list_using_comp = [var for var in input_list if var % 2 == 0]


print("Output List using list comprehensions: ",list_using_comp)


>>> Output List using for loop: [2, 4, 4, 6]
```

Dictionary Comprehension

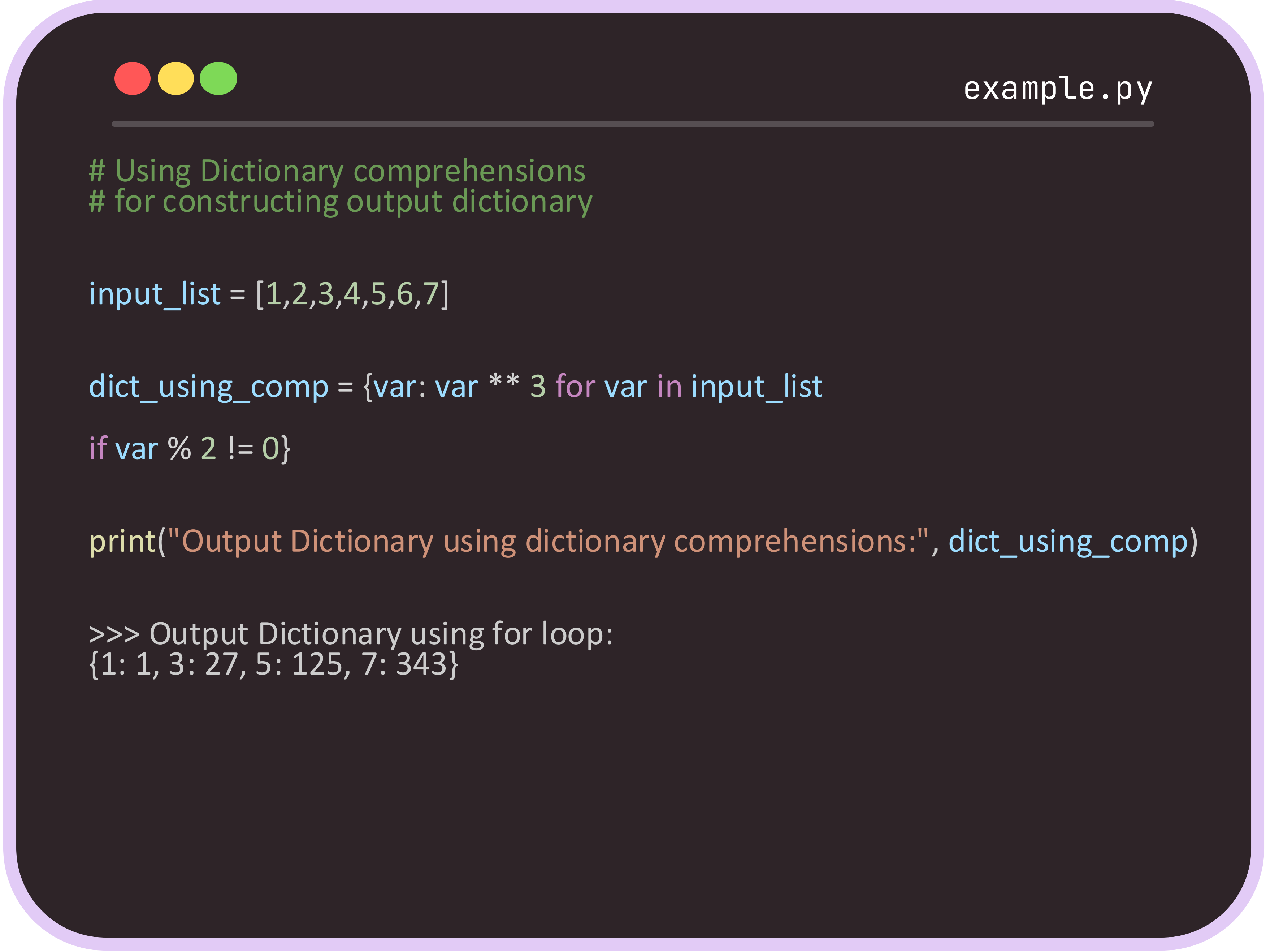
Extending the idea of list comprehensions, we can also create a dictionary using dictionary comprehensions. The basic structure of a dictionary comprehension looks like below.



```
input_list = [1, 2, 3, 4, 5, 6, 7]
output_dict = {}

# Using loop for constructing output dictionary
for var in input_list:
  if var % 2 != 0:
    output_dict[var] = var**3

print("Output Dictionary using for loop:", output_dict)

>>> Output Dictionary using for loop:
{1: 1, 3: 27, 5: 125, 7: 343}
```

Dictionary Comprehension

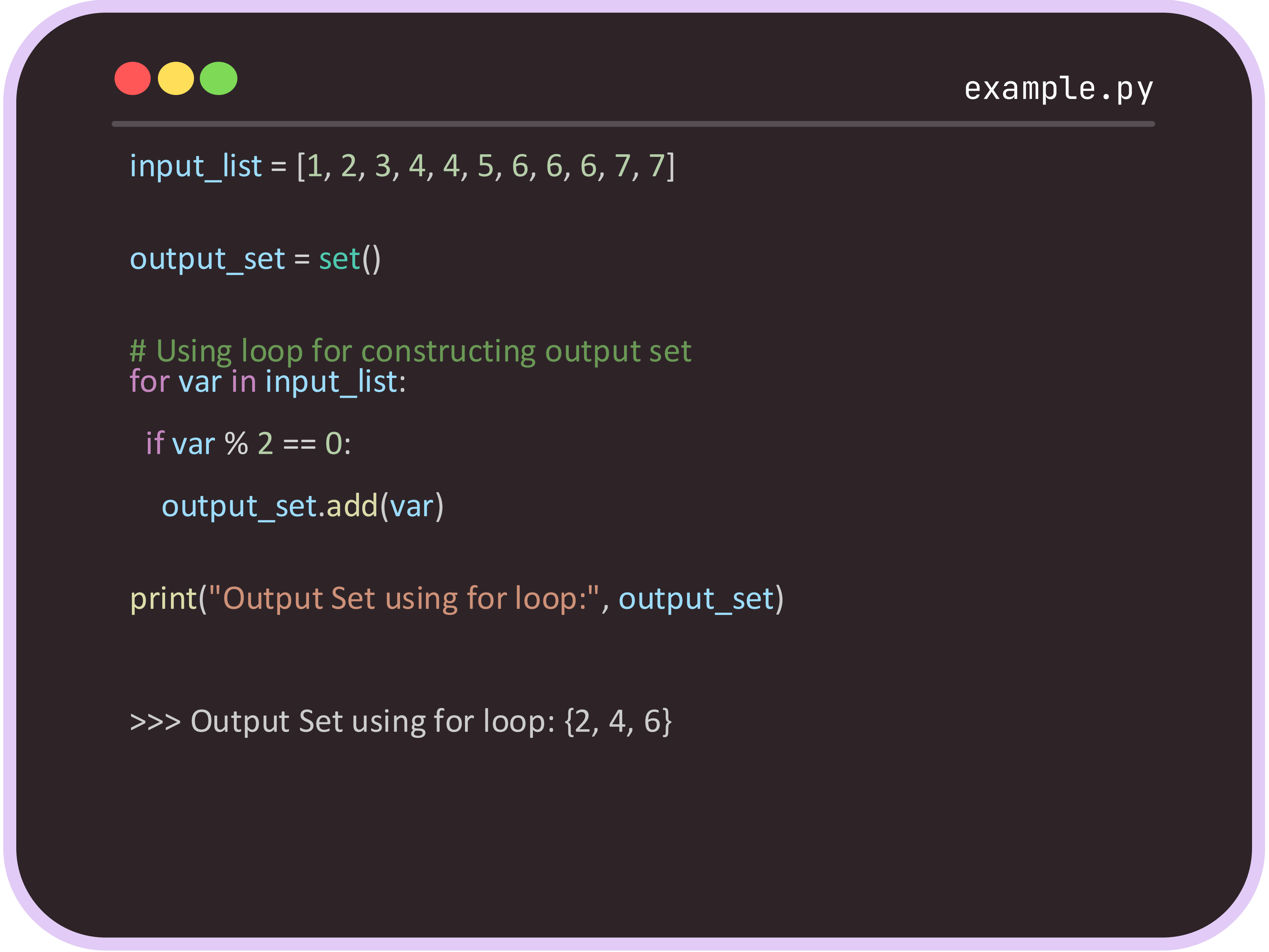
Extending the idea of list comprehensions, we can also create a dictionary using dictionary comprehensions. The basic structure of a dictionary comprehension looks like below.



```
# Using Dictionary comprehensions
# for constructing output dictionary

input_list = [1,2,3,4,5,6,7]

dict_using_comp = {var: var ** 3 for var in input_list

if var % 2 != 0}

print("Output Dictionary using dictionary comprehensions:", dict_using_comp)

>>> Output Dictionary using for loop:
{1: 1, 3: 27, 5: 125, 7: 343}
```

Set Comprehension

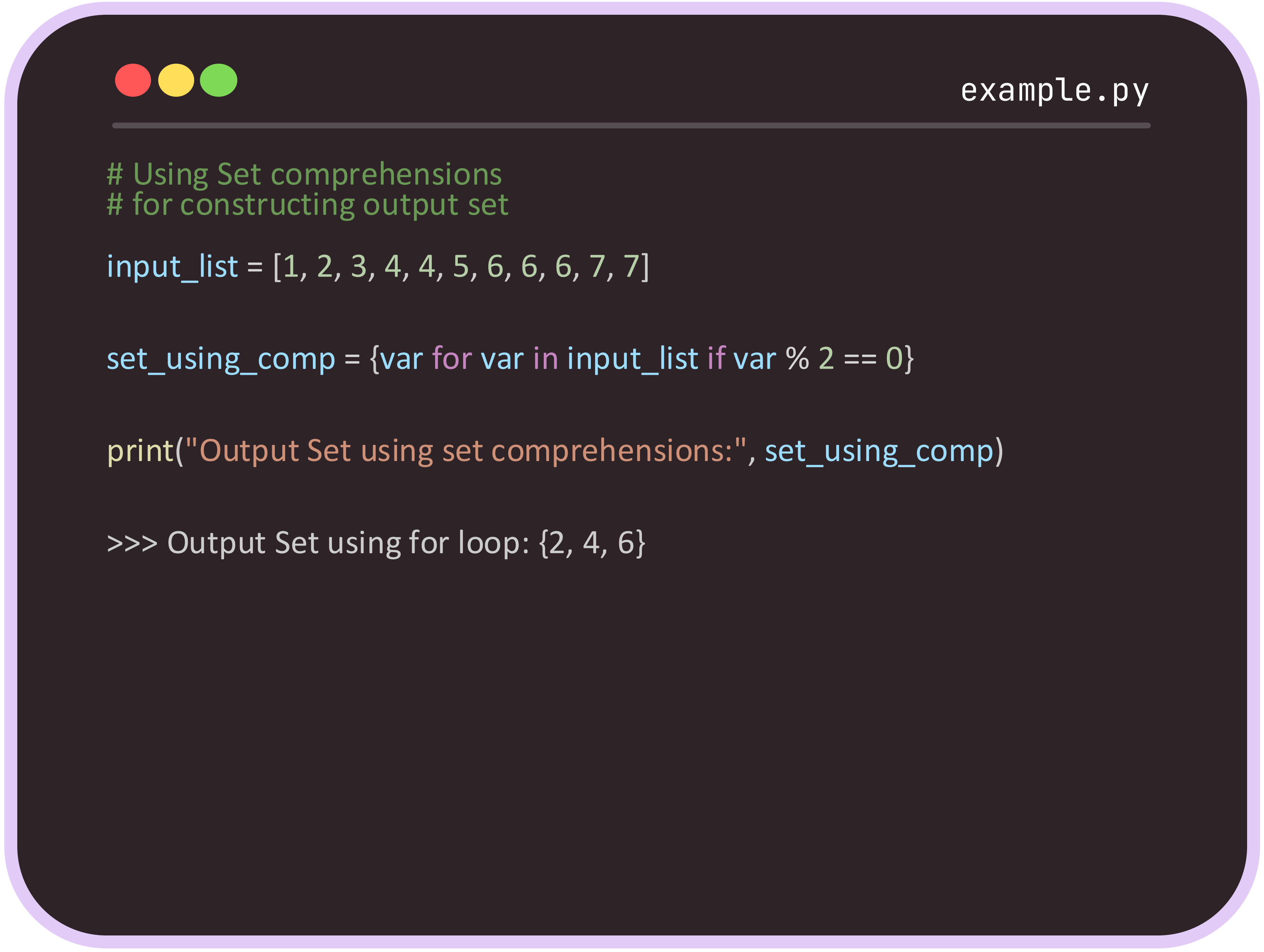
Set comprehensions are pretty similar to list comprehensions. The only difference between them is that set comprehensions use curly brackets { }. Let's look at the following example to understand set comprehensions.



```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]

output_set = set()

# Using loop for constructing output set
for var in input_list:
  if var % 2 == 0:
    output_set.add(var)

print("Output Set using for loop:", output_set)


>>> Output Set using for loop: {2, 4, 6}
```

Set Comprehension

Set comprehensions are pretty similar to list comprehensions. The only difference between them is that set comprehensions use curly brackets { }. Let’s look at the following example to understand set comprehensions.



```
# Using Set comprehensions
# for constructing output set

input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]

set_using_comp = {var for var in input_list if var % 2 == 0}

print("Output Set using set comprehensions:", set_using_comp)

>>> Output Set using for loop: {2, 4, 6}
```

Generator Comprehension

The major difference between them is that generators don’t allocate memory for the whole list. Instead, they generate each value one by one which is why they are memory efficient. Let’s look at the following example to understand generator comprehension:

example.py

```python
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]

output_gen = (var for var in input_list if var % 2 == 0)

print(output_gen)

print("Output values using generator comprehensions:", end = ' ')

for var in output_gen:
  print(var, end = ' ‘)
```

```
>>> <generator object <genexpr> at 0x104d2aa80>
>>> Output values using generator comprehensions: 2 4 4 6
```

To comprehension or not to comprehension...

- An **elegant** way to define and create lists based on existing lists.
- Generally, **more compact and faster** than normal functions and loops for creating list.
- However, we should **avoid** writing very **long list comprehensions** in one line to ensure that code is user-friendly.
- **Remember, every list comprehension can be rewritten in for loop, but every for loop can't be rewritten in the form of list comprehension.**

Class Wrap-up

What have we learned today?

1. **Defining Basic Functions:**
   - We began by understanding the syntax and structure of basic functions in Python.
   - Explored how to create and call functions to encapsulate and reuse code.
2. **Functions with Return:**
   - Discussed the concept of return statements within functions, enabling the functions to yield results.
   - Explored examples showcasing the use of return to convey values back to the calling code.
3. **Functions with Parameters:**
   - Extended our knowledge to functions that accept parameters, allowing for dynamic and flexible behaviour.
   - Examined how to pass arguments to functions, enhancing their versatility.
4. **Functions with Standard Parameters:**
   - Dived into standard parameters, understanding how default values can be set for function parameters.
   - Demonstrated how this feature provides flexibility and simplifies function calls.
5. **Comprehensions in Python:**
   - Shifted our focus to comprehensions, a concise and expressive way to create data structures in Python.
   - Explored the syntax and usage of comprehensions, enabling compact creation of either list, dictionary, set or a generator.

PRACTICE
PRACTICE
PRACTICE

You won't master a skill if you don't practice!

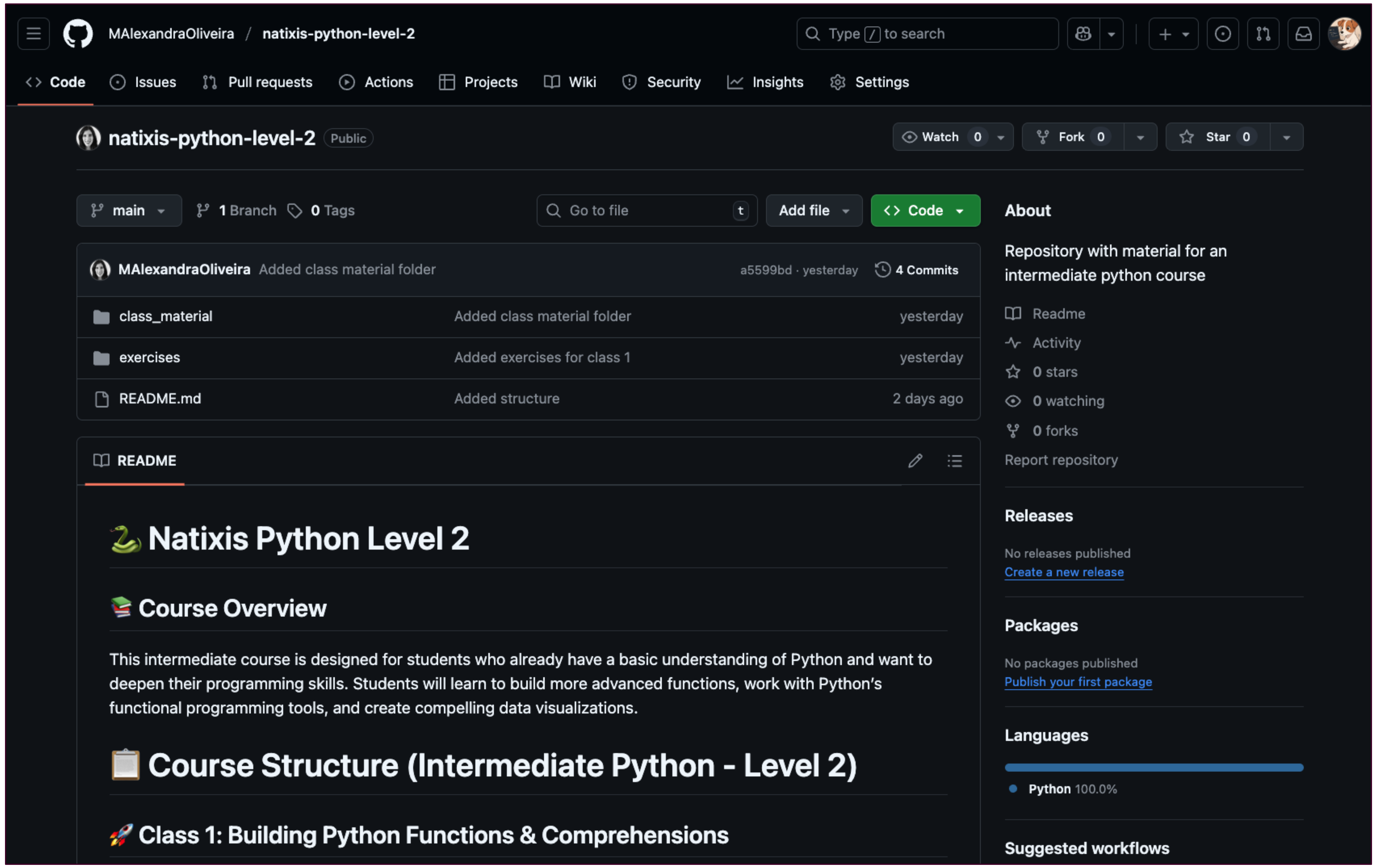
Exercises – Learn by doing!

In order to facilitate the learning process of Python **we have prepared for each session a python file** where you can find **exercises** that will help you to grasp the introduced Python concepts.

Visual Studio Code

We will use **VS CODE** as our Python program IDE

Exercises for today



MAlexandraOliveira Added class material folder a5599bd · yesterday 4 Commits

class_material	Added class material folder	yesterday
exercises	Added exercises for class 1	yesterday
README.md	Added structure	2 days ago

README

🐍 Natixis Python Level 2

📚 Course Overview

This intermediate course is designed for students who already have a basic understanding of Python and want to deepen their programming skills. Students will learn to build more advanced functions, work with Python's functional programming tools, and create compelling data visualizations.

📋 Course Structure (Intermediate Python - Level 2)

🚀 Class 1: Building Python Functions & Comprehensions

About

Repository with material for an intermediate python course

Readme
Activity
0 stars
0 watching
0 forks
Report repository

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

Python 100.0%

Suggested workflows

Link to exercises: https://github.com/MAlexandraOliveira/natixis-python-level-2/blob/main/exercises/Class1_exercises.py

Why should you deactivate Copilot? (for now)

As **beginners in Python programming**, it’s crucial to focus on truly understanding how code works, rather than just seeing it appear. Tools like GitHub Copilot can be tempting, but they **often offer solutions without explanation**, making it easy to skip the learning process. While these tools are designed to assist, **not replace your thinking**, they can encourage you to rely on solutions you don't fully grasp—and they’re not always correct. To truly learn, you need to write, debug, and explore code on your own. **By turning off Copilot** during the early stages of your learning, you give yourself the opportunity to develop real problem-solving skills, build confidence, and create a strong foundation. Later, when you have a solid grasp of the basics, Copilot can serve as a useful support tool, but always approach its suggestions with a critical mindset, not blind trust.

Steps to turn-off GitHub Copilot:

1. Go to Settings (File > Preferences > Settings or press Ctrl+,).
2. In the search bar, type: Copilot.
3. Find the setting GitHub Copilot: Enable.
4. Uncheck it to disable Copilot globally.

THANK YOU 😊

Questions?

Fábio Neves fnssm26@gmail.com

Luis Dias luis.f.s.m.dias@gmail.com