Python.

python platforms

anaconda.

Anaconda is a platform for data science and artificial intelligence (AI) that uses the Python and R programming languages.

It's used to develop and manage projects for data science and AI

- Package management
- Development environment
- Pre-installed packages
- Open-source

steps to install Anaconda

- ➔ go to official website **download** link : https://www.anaconda.com/download/success
- ➔ follow the basic installation instructions
- ➔ select **Install for: just me**.
- ➔ don’t forget to select **add anaconda in your environment variables**.
- ➔ verify your conda version, conda --version.

open jupyter notebook

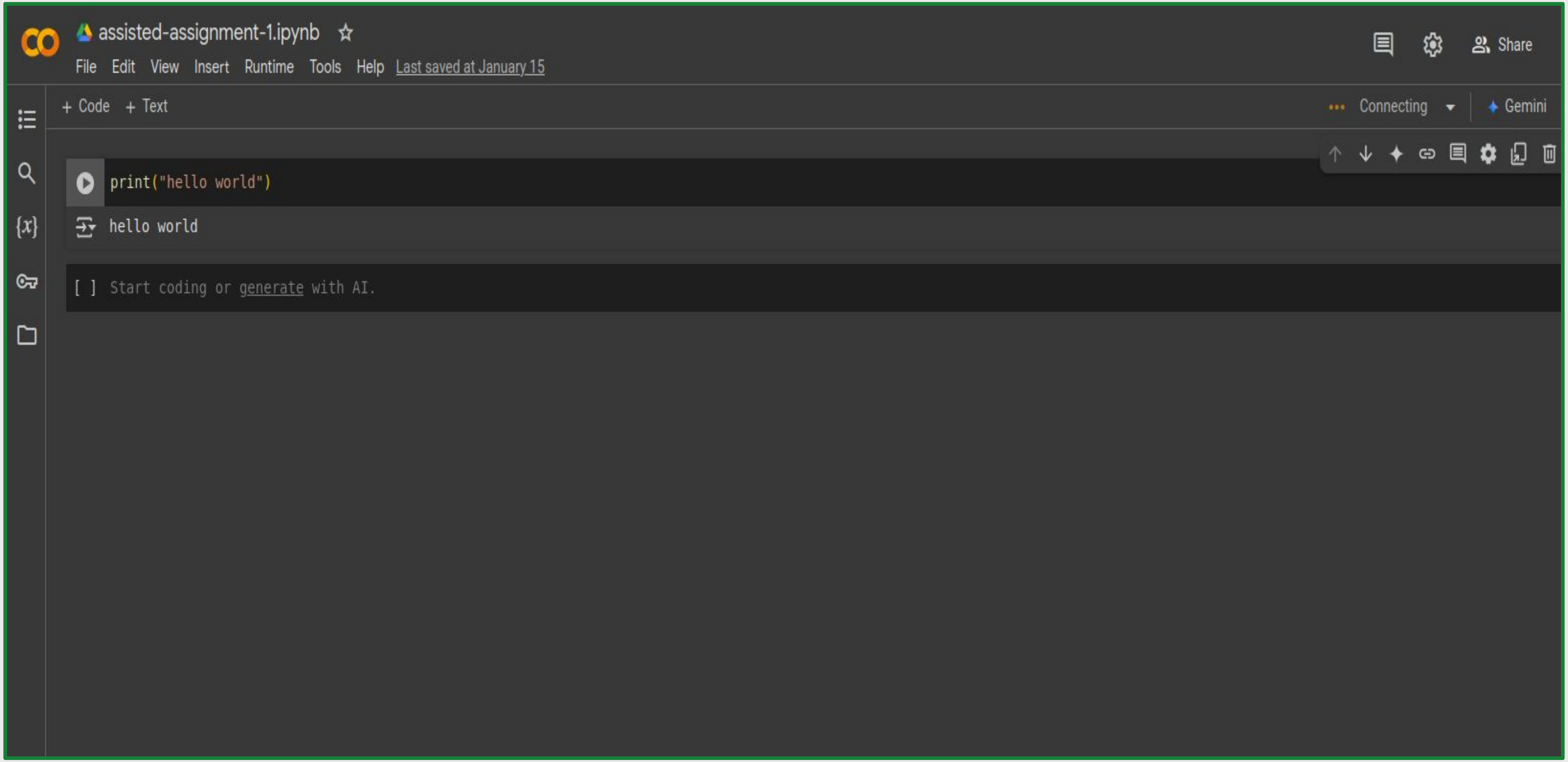
- ➔ open the Anaconda Navigator application.
- ➔ in navigator interface, click on "Jupyter Notebook."
- ➔ a browser window will open, displaying the Jupyter Notebook interface.

google collab.

Colab is a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources.

Just be log into your google account and use it.

As simple as that.



assisted-assignment-1.ipynb
File Edit View Insert Runtime Tools Help Last saved at January 15
Share
+ Code + Text
Connecting
Gemini

```
print("hello world")
```

```
hello world
```

```
[ ] Start coding or generate with AI.
```

basics of python.

introduction to basic python.

*install anaconda

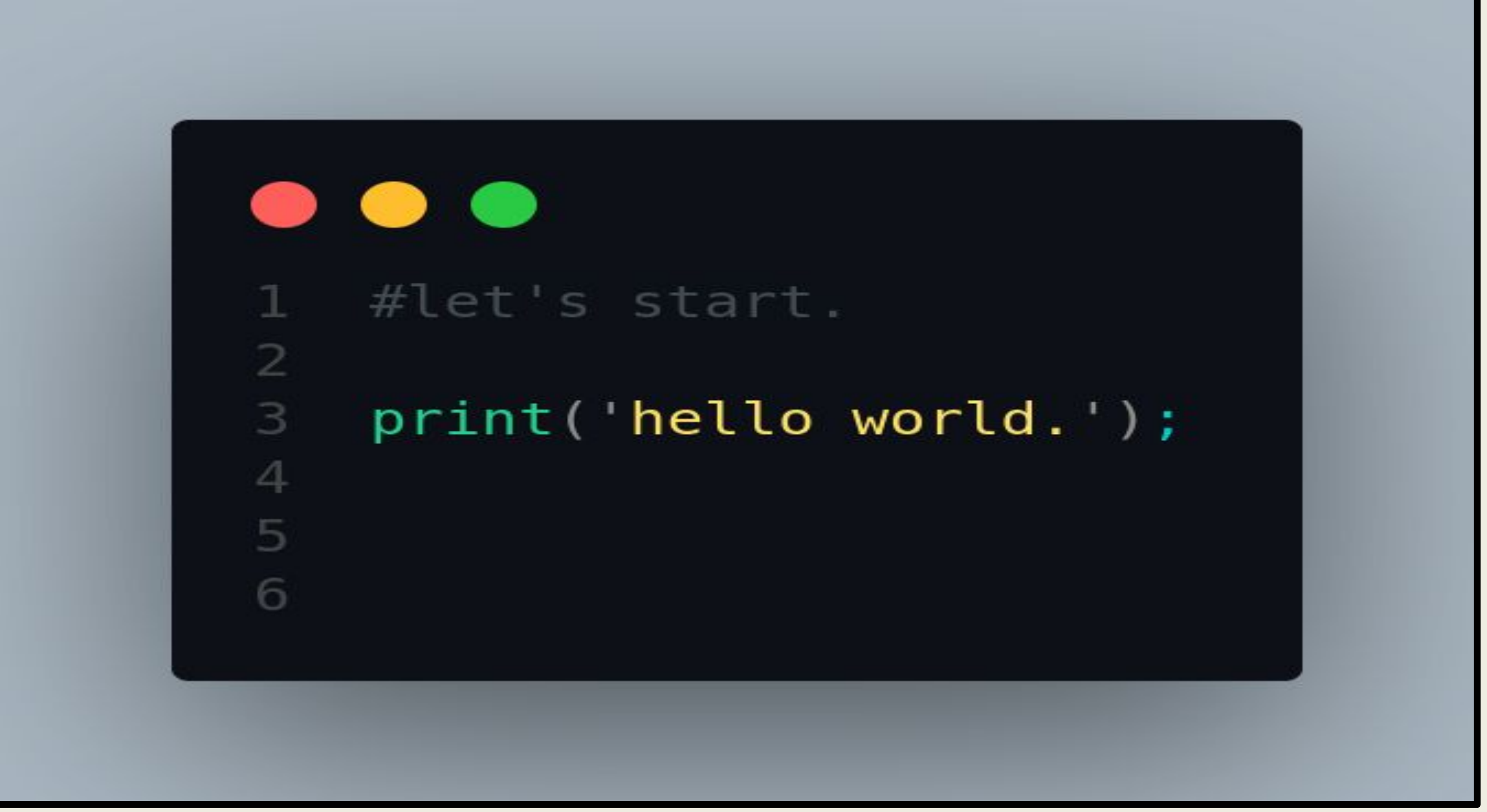
history.

created, 1989 by **Guido van Rossum**.

v1 by 1991.

used for.

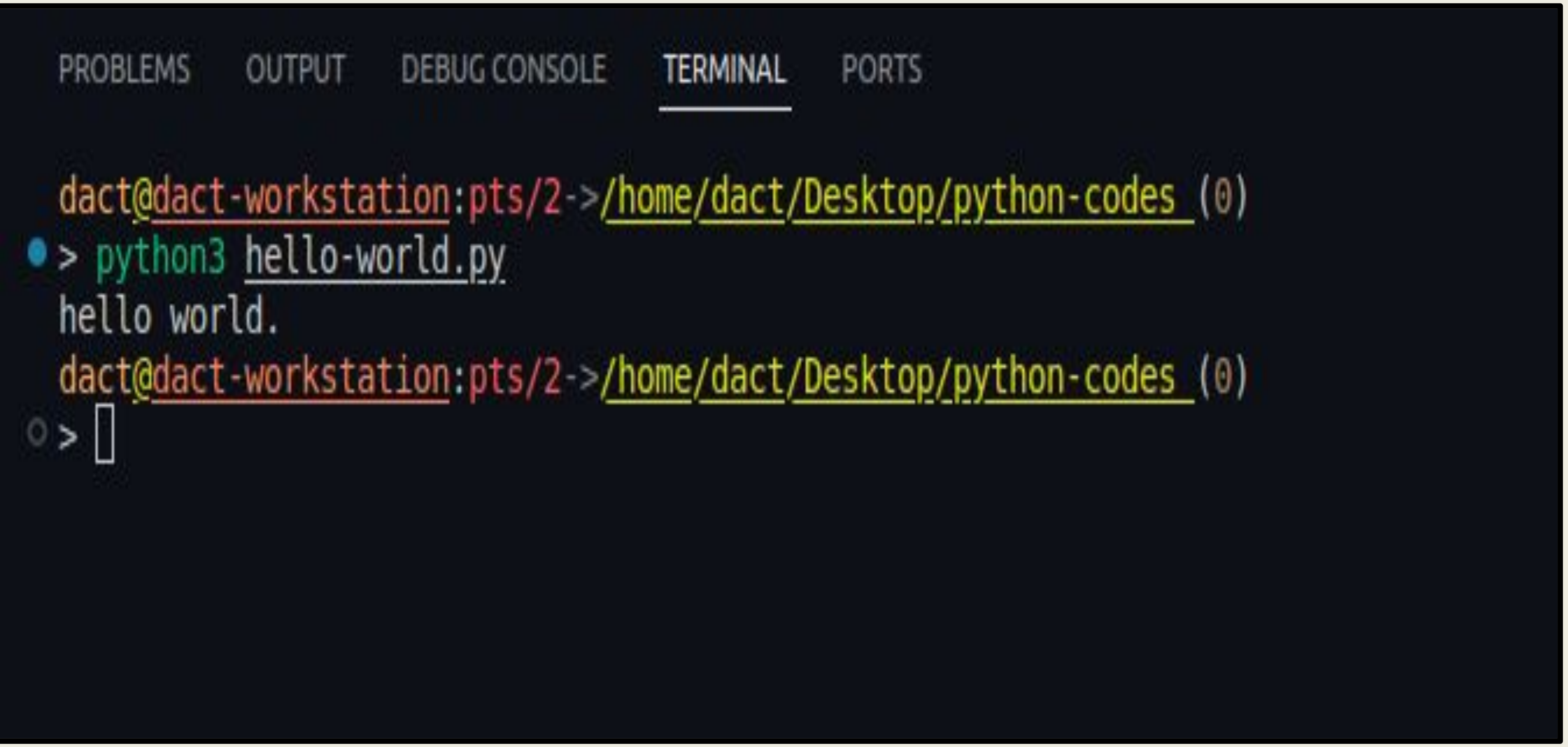
- ➔ **data-science.**
- ➔ **ai/ml**.
- ➔ **backend developer**.
- ➔ automation & scripting.
- ➔ cybersecurity.
- ➔ finTech.



```
#let's start.

print('hello world.');
```

➔ interpreted language.

command to execute your python file.
>**python3** ***<file-name>.py***



```
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
> python3 hello-world.py
hello world.
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
> 
```

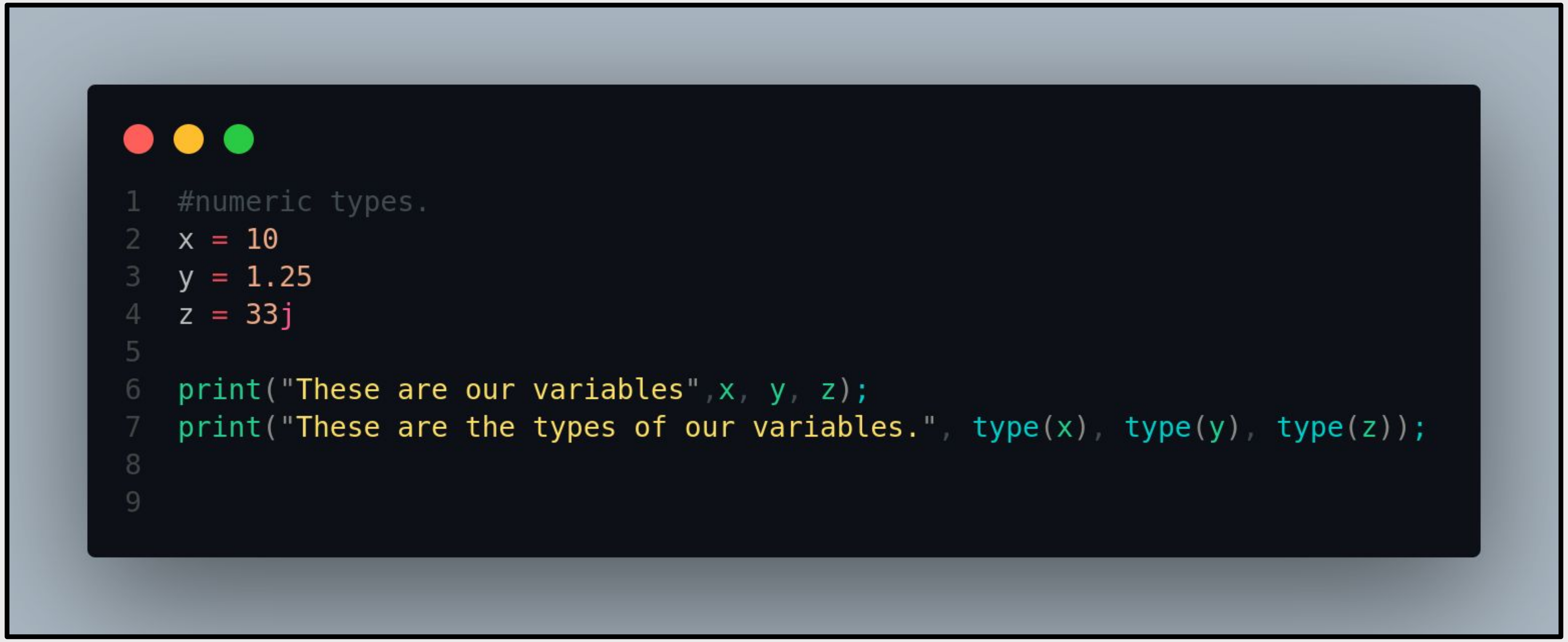
data-types.

data-types.

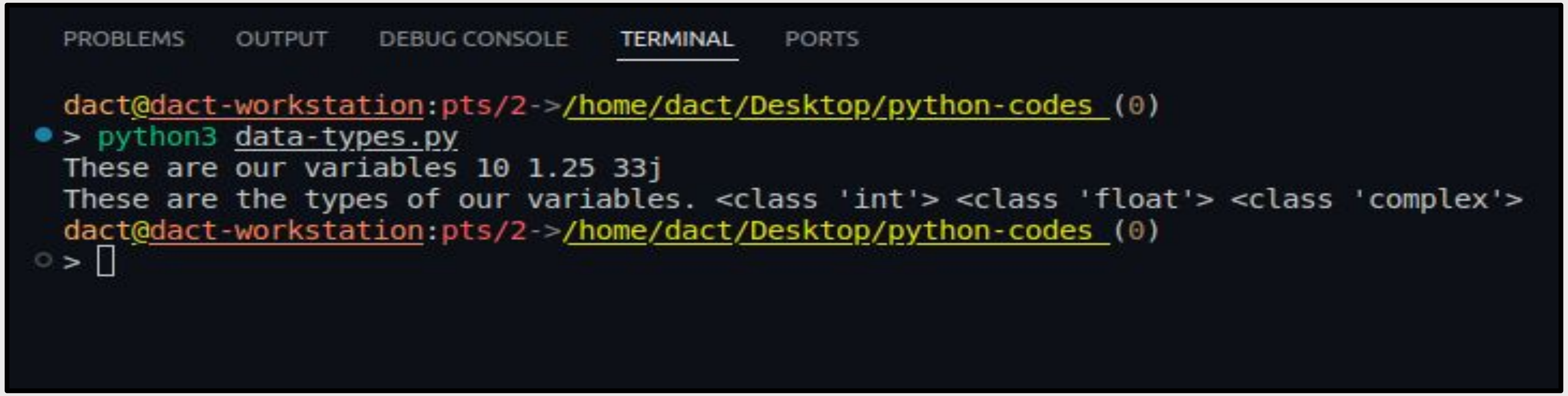
- **numeric**
  - int.
  - float.
  - complex.
- **text**
  - str.
- **boolean**
  - true / false.

- **sequence**
  - list.
  - tuple.
  - range.
- **mapping**
  - dict.

numeric data.



```python
#numeric types.
x = 10
y = 1.25
z = 33j

print("These are our variables",x, y, z);
print("These are the types of our variables.", type(x), type(y), type(z));
```



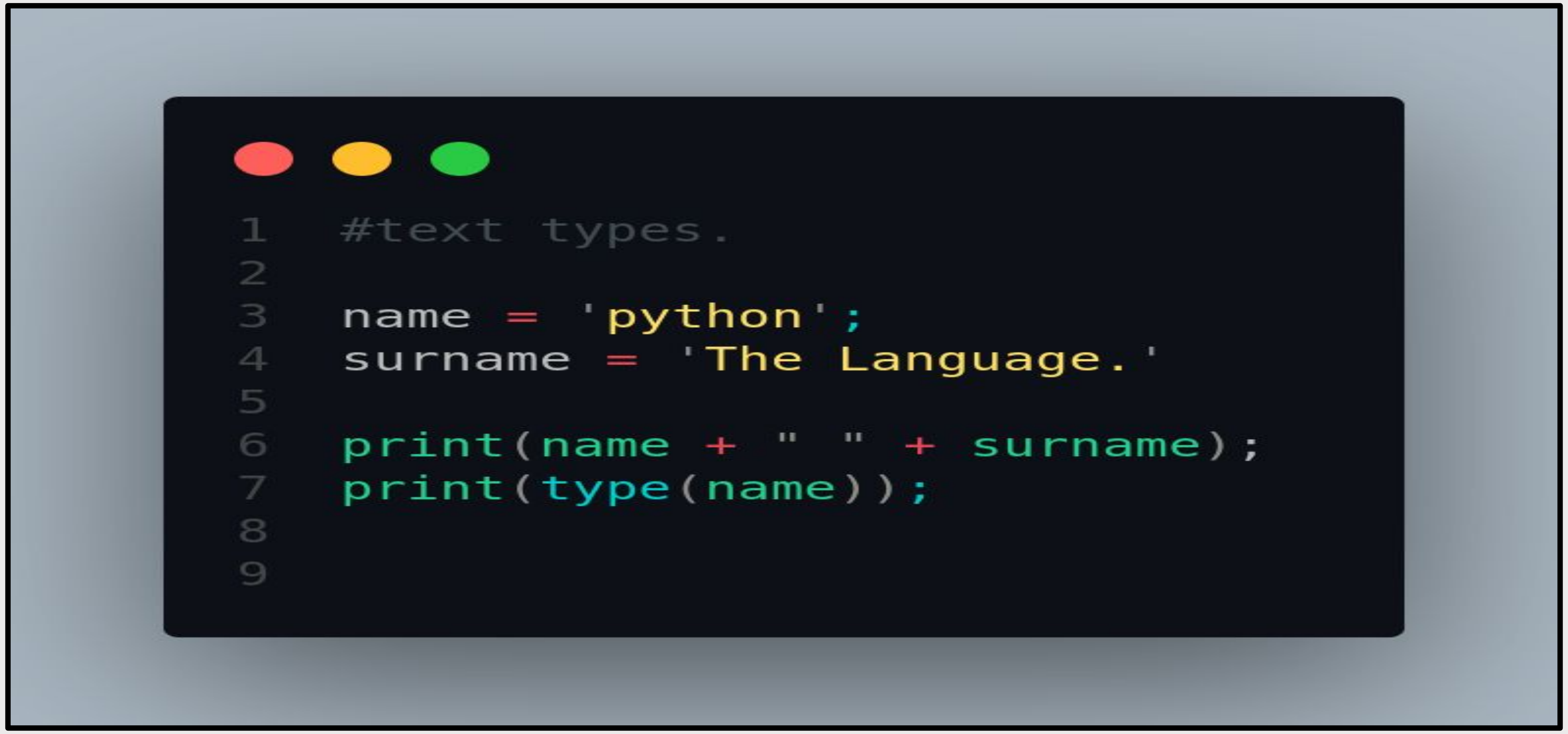
```
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
> python3 data-types.py
These are our variables 10 1.25 33j
These are the types of our variables. <class 'int'> <class 'float'> <class 'complex'>
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
>
```

int.

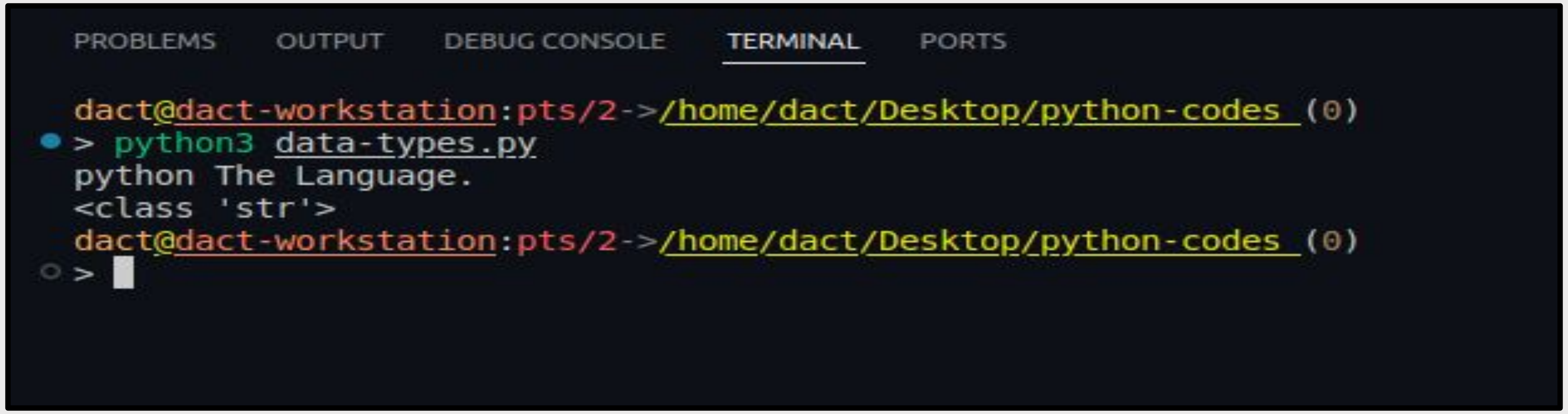
float.

complex.

text data.



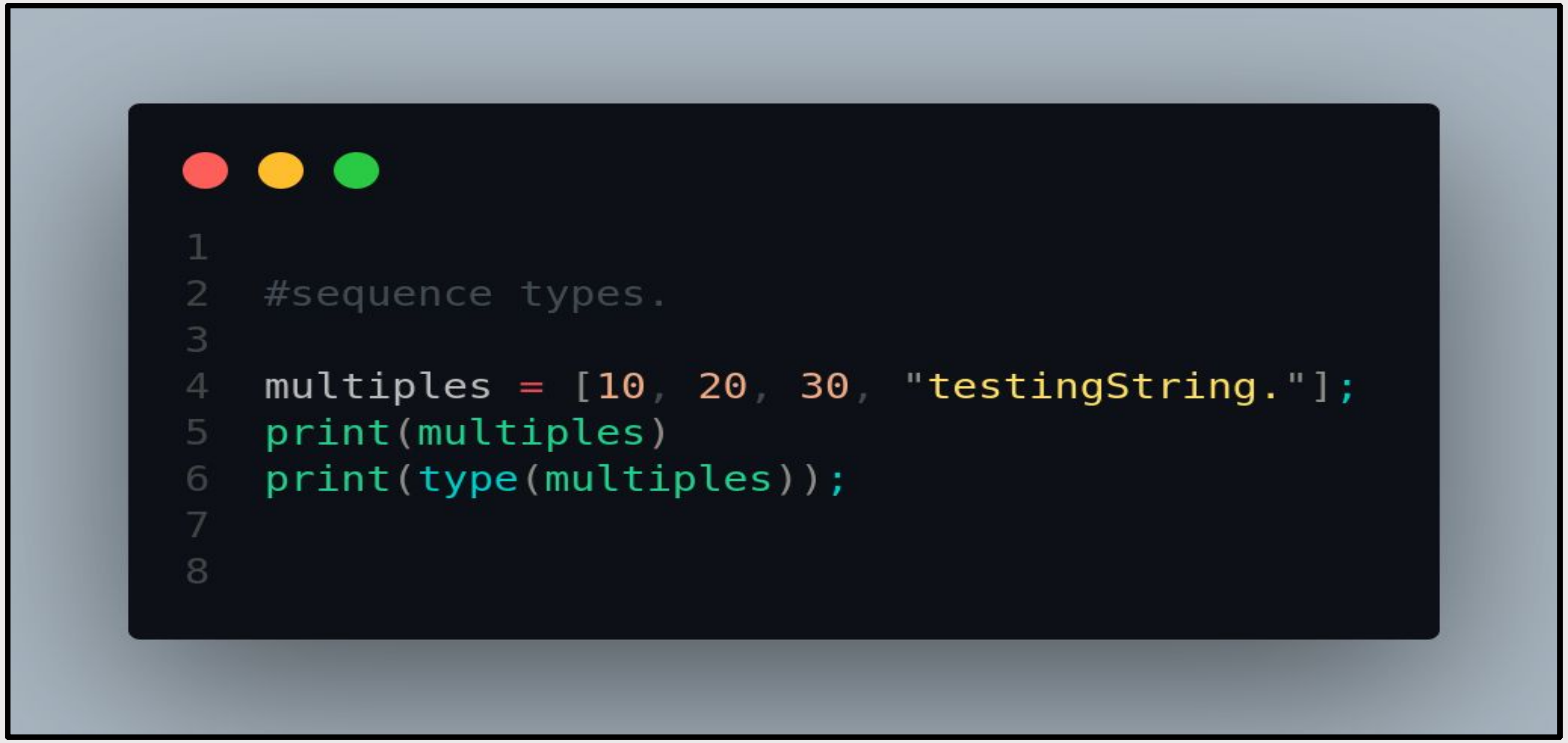
```python
#text types.

name = 'python';
surname = 'The Language.'

print(name + " " + surname);
print(type(name));
```



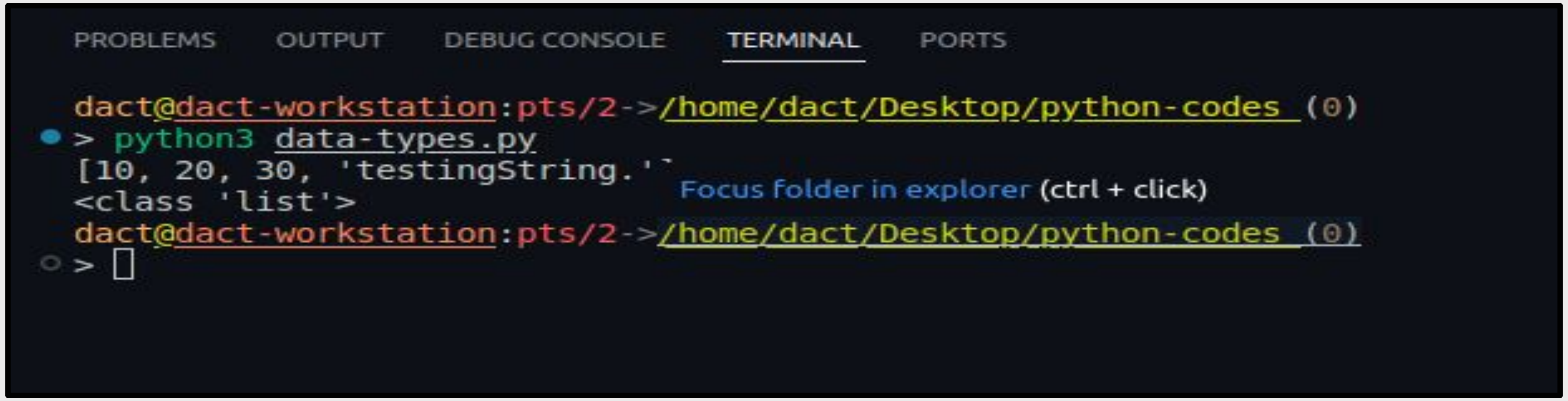
```
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
> python3 data-types.py
python The Language.
<class 'str'>
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
>
```

string.

sequence.



```python
#sequence types.

multiples = [10, 20, 30, "testingString."];
print(multiples)
print(type(multiples));
```

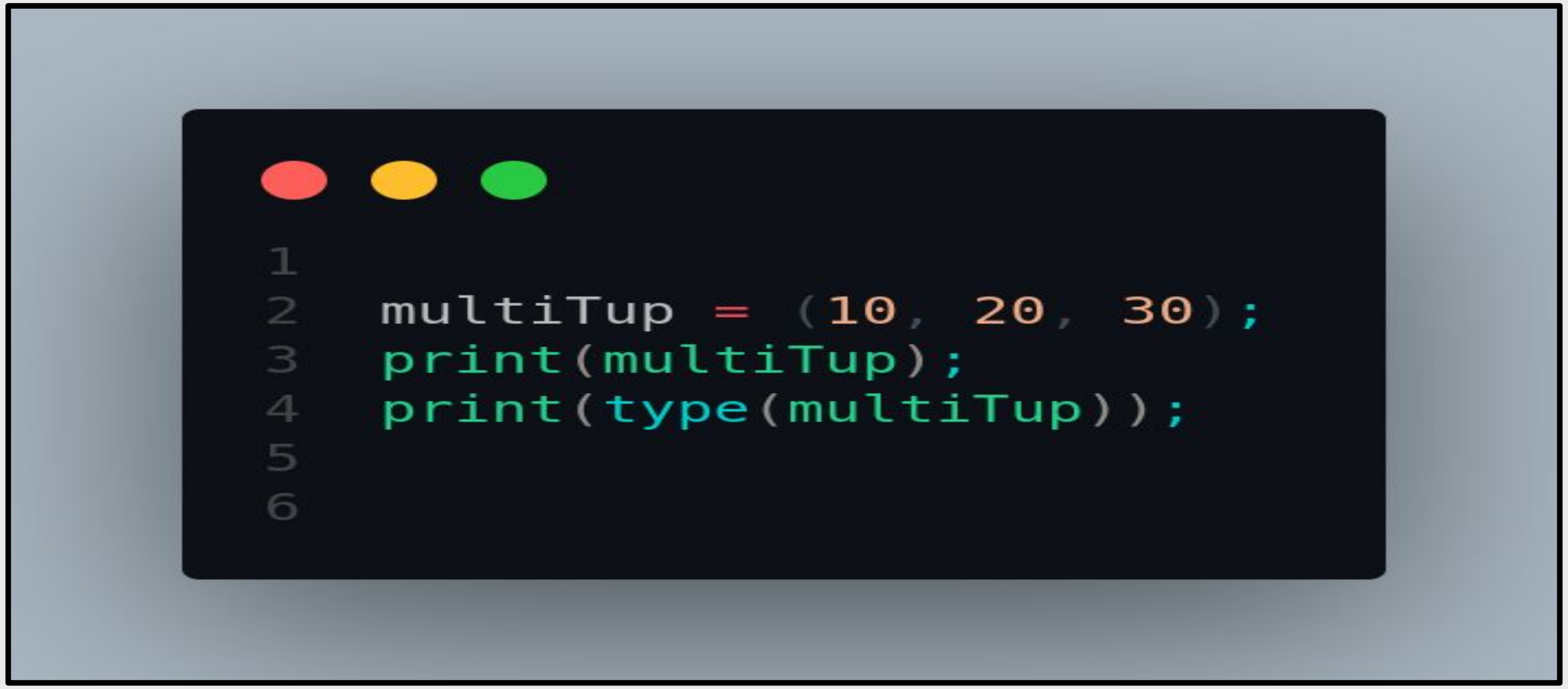


```
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
> python3 data-types.py
[10, 20, 30, 'testingString.']
<class 'list'>
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
> 
```

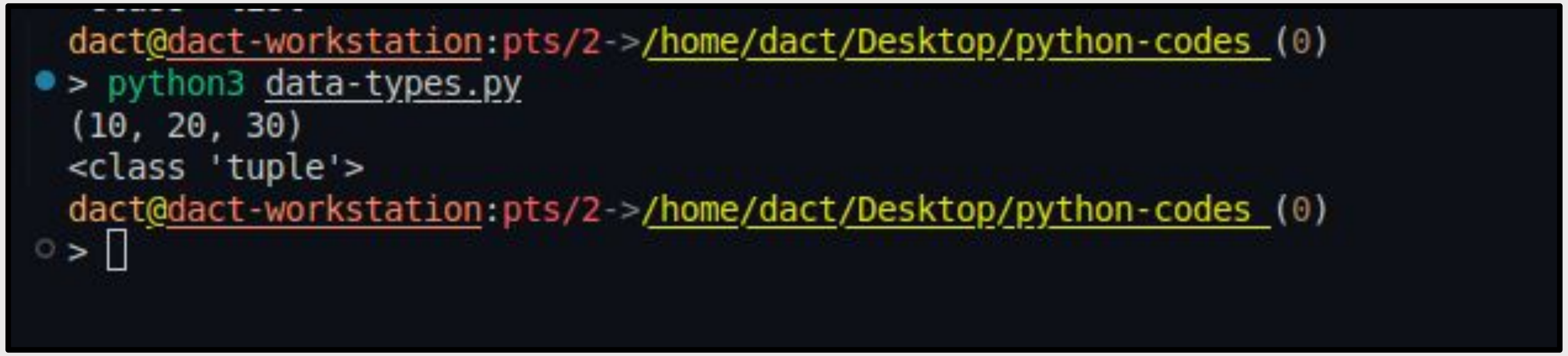
list.

ordered
mutable
allows duplicate
element

sequence.



```python

multiTup = (10, 20, 30);
print(multiTup);
print(type(multiTup));


```

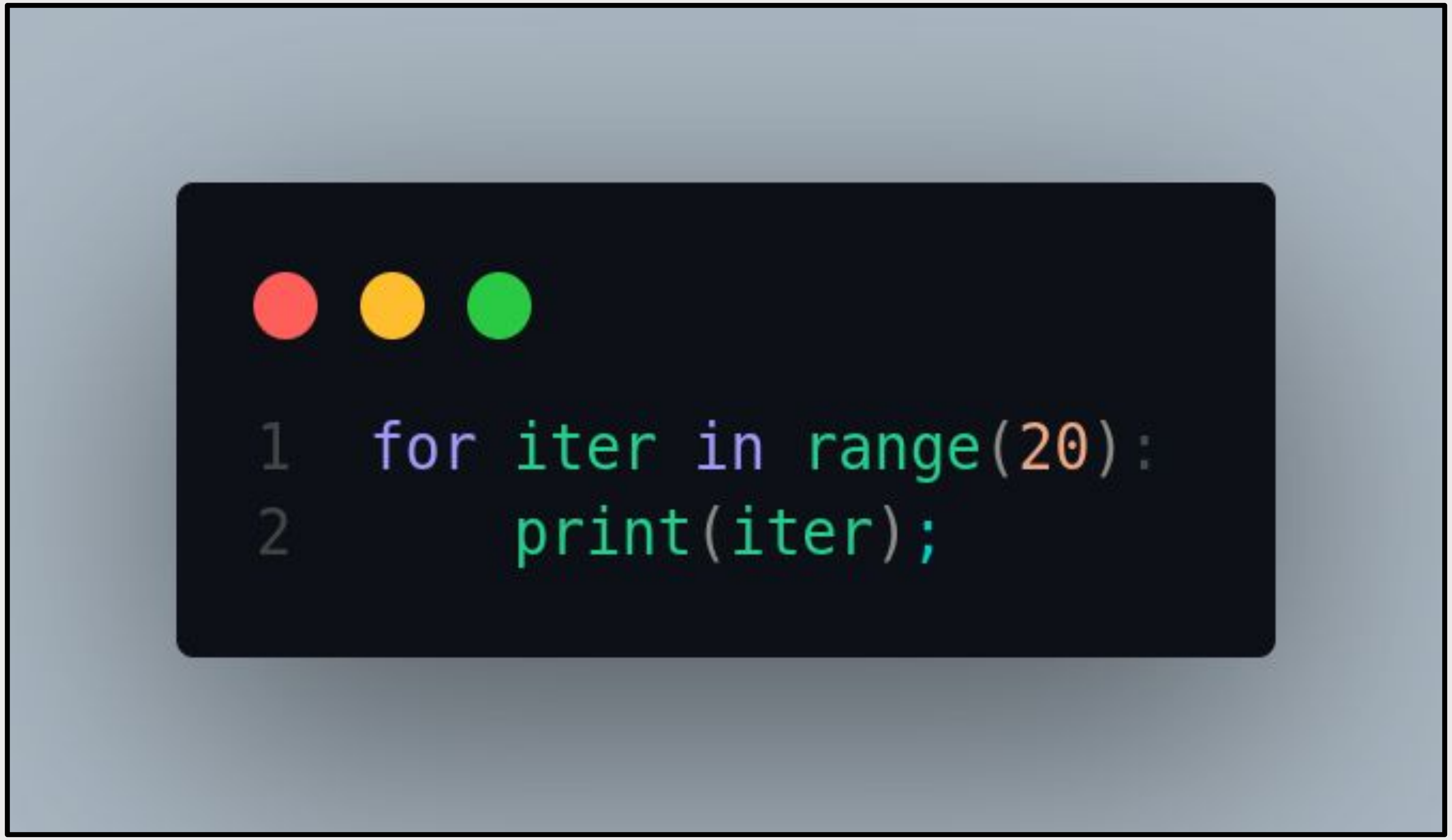


```
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
> python3 data-types.py
(10, 20, 30)
<class 'tuple'>
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
> 
```

tuple.

ordered
<u>immutable</u>
allows duplicate elements.

sequence.

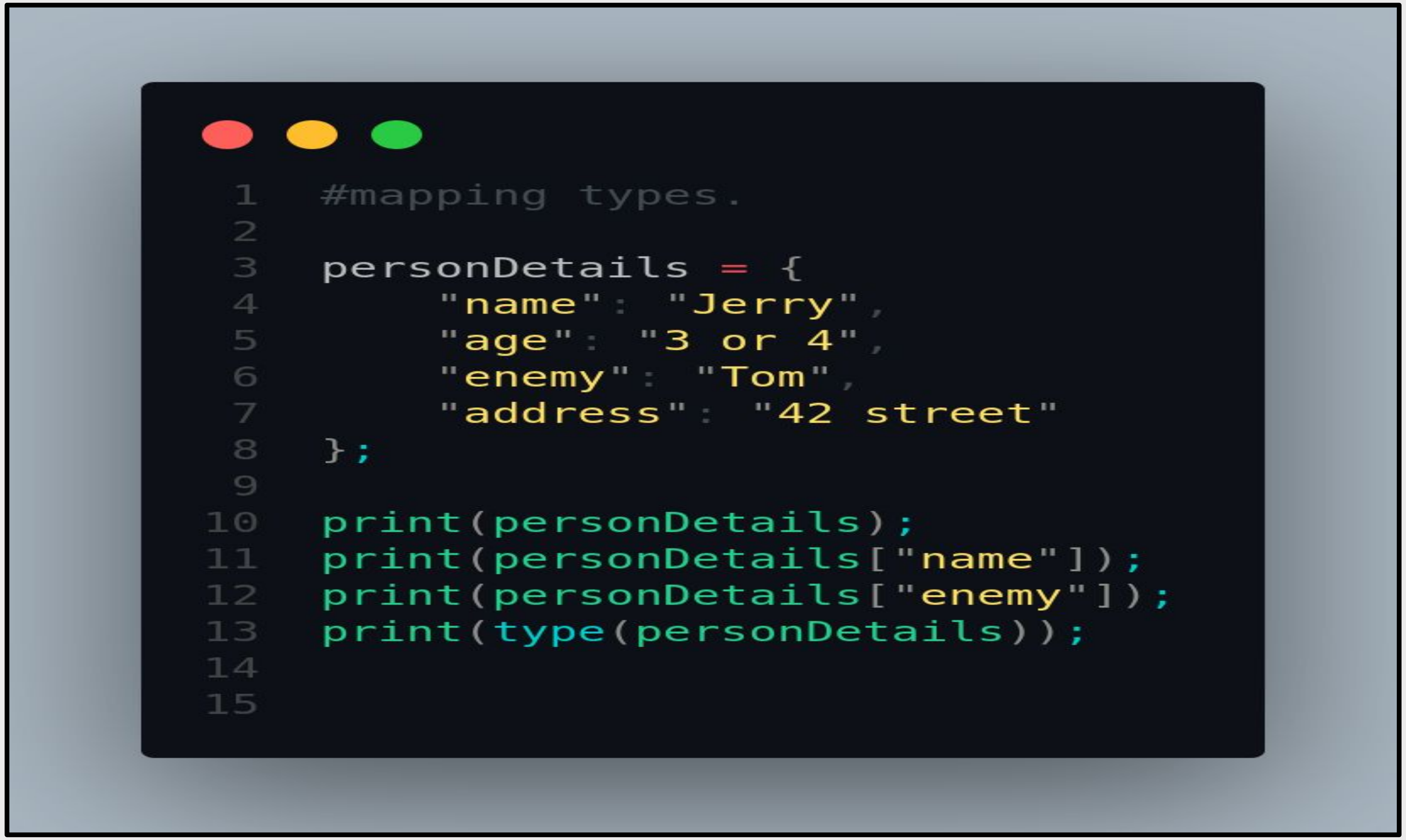


```
for iter in range(20):
    print(iter);
```

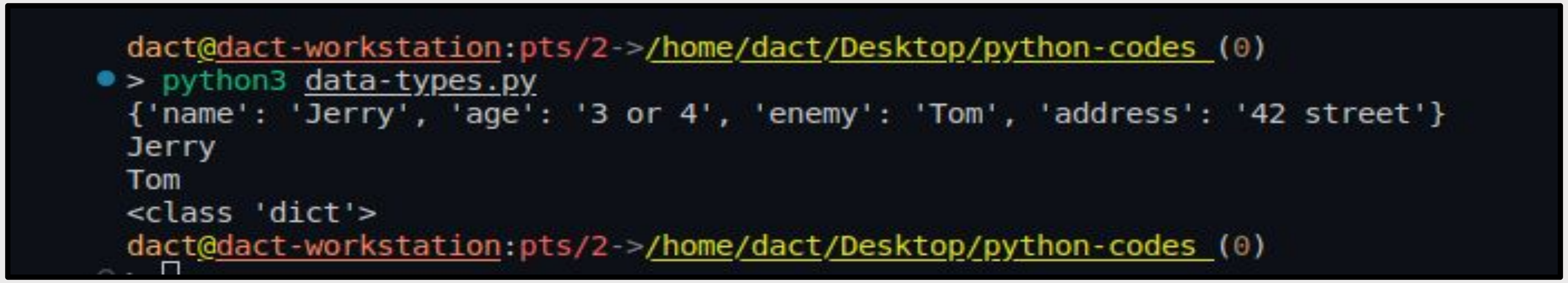
range.

represents sequence of numbers. predominantly used in loops.

mapping.



```
#mapping types.

personDetails = {
        "name": "Jerry",
        "age": "3 or 4",
        "enemy": "Tom",
        "address": "42 street"
};

print(personDetails);
print(personDetails["name"]);
print(personDetails["enemy"]);
print(type(personDetails));
```

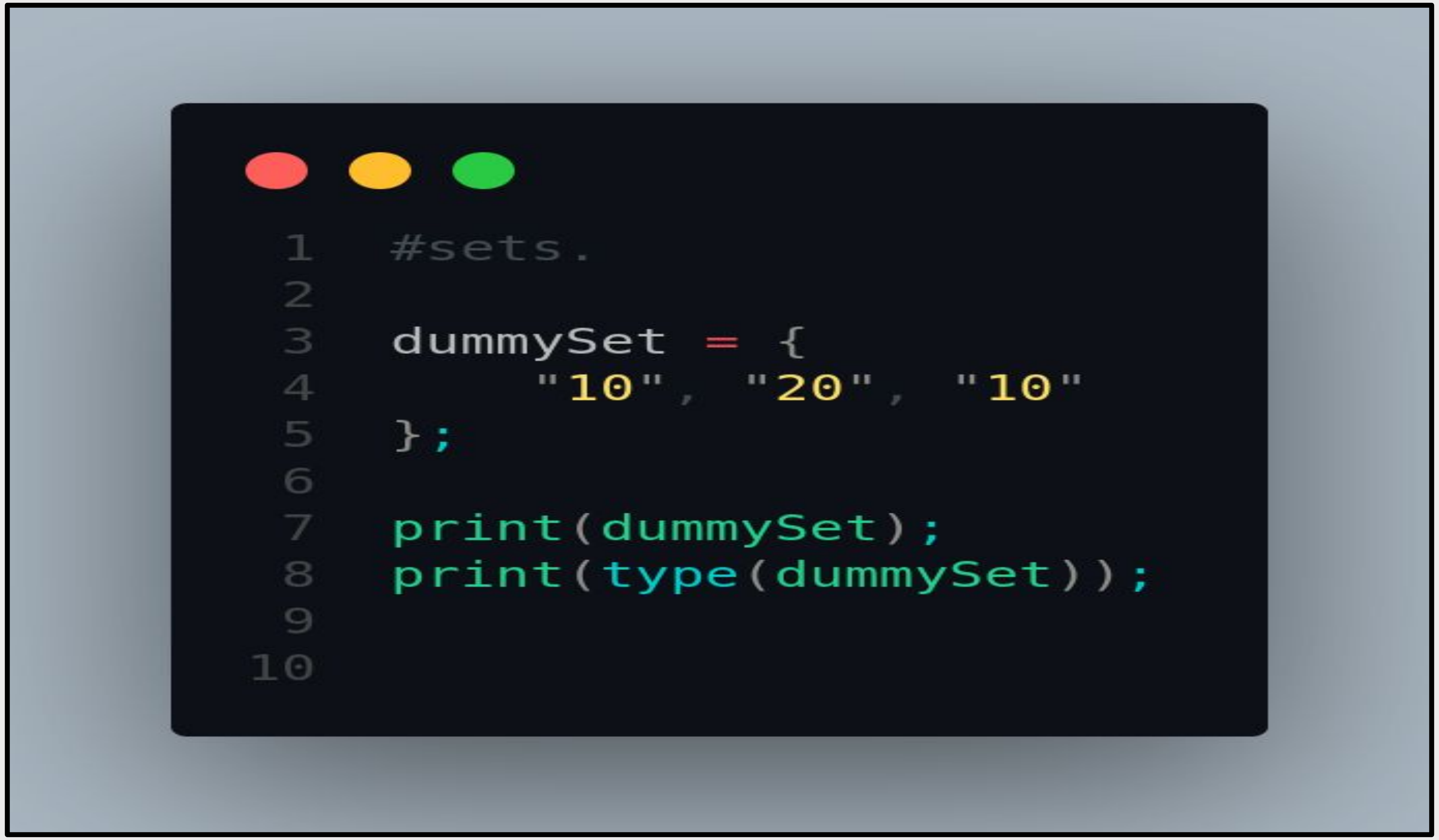


```
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
> python3 data-types.py
{'name': 'Jerry', 'age': '3 or 4', 'enemy': 'Tom', 'address': '42 street'}
Jerry
Tom
<class 'dict'>
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
```

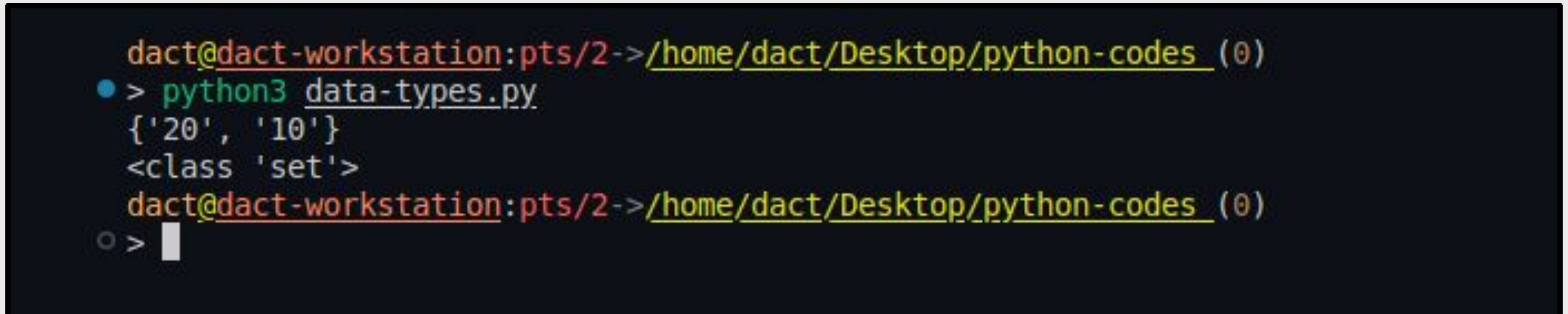
dictionary.

stores data in {key:value} pairs.
called as objects in javaScript.
keys must be unique.

sets.



```
#sets.

dummySet = {
        "10", "20", "10"
};

print(dummySet);
print(type(dummySet));
```



```
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
> python3 data-types.py
{'20', '10'}
<class 'set'>
dact@dact-workstation:pts/2->/home/dact/Desktop/python-codes (0)
> 
```

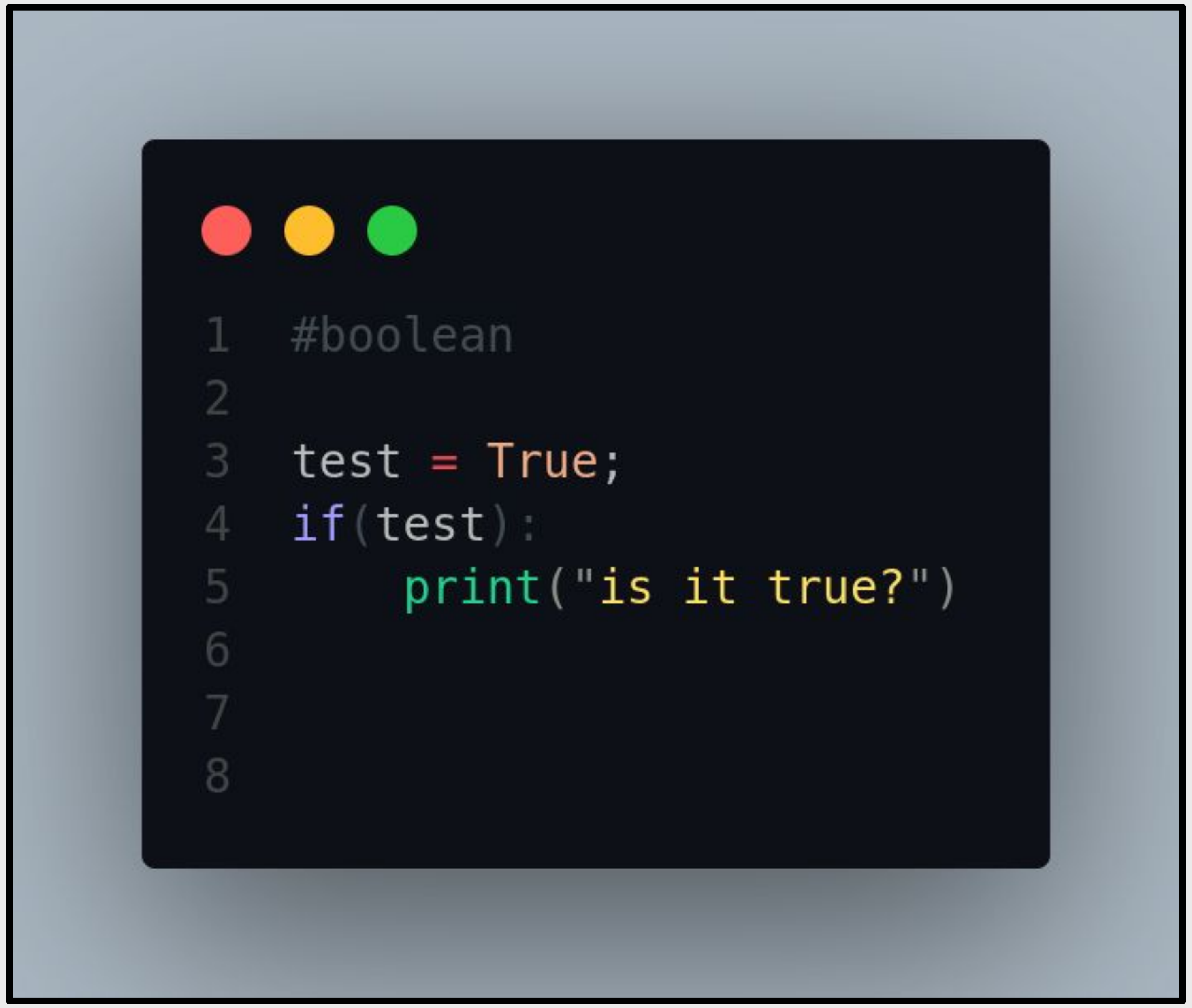
sets.

unordered
mutable
<u>no duplicate elements</u>

explore frozensets.

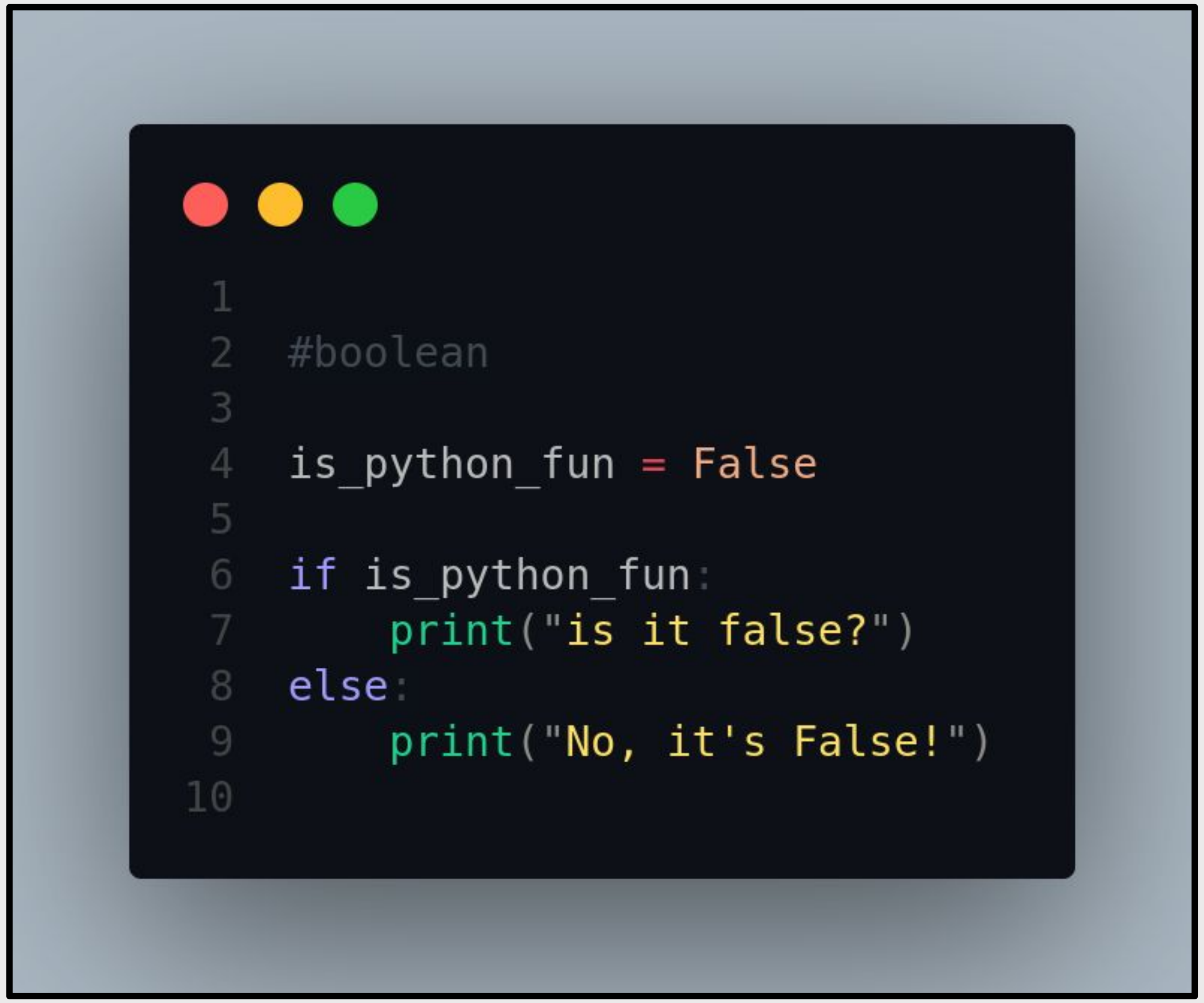
Feature/Aspect	List	Tuple	Set	Dictionary (dict)
Definition	Ordered collection of items.	Ordered, immutable collection.	Unordered collection of unique items.	Unordered collection of key-value pairs.
Syntax	[ ]	( )	{ }	{key: value}
Mutability	Mutable (can be changed).	Immutable (cannot be changed).	Mutable, but elements must be immutable.	Mutable (keys must be immutable).
Duplicates	Allows duplicates.	Allows duplicates.	Does not allow duplicates.	Keys must be unique; values can be duplicate.
Order	Maintains insertion order (Python 3.7+).	Maintains order.	Unordered.	Maintains insertion order (Python 3.7+).
Accessing Elements	Accessed via index (list[0]).	Accessed via index (tuple[0]).	Cannot access via index (no order).	Accessed via keys (dict['key']).
Changeability	Can add, remove, or modify items.	Cannot be changed after creation.	Can add or remove items, but no duplicates.	Can add, remove, or modify key-value pairs.
Common Methods	append(), pop(), remove(), extend()	count(), index()	add(), remove(), union(), intersection()	keys(), values(), items(), get()
Performance	Slower than tuple for iteration.	Faster for iteration (immutable).	Faster for membership testing.	Fast lookups and retrievals via keys.
Use Case	Use when data can change.	Use when data is fixed and needs protection.	Use for unique elements.	Use for key-value relationships.

boolean.



```python
#boolean

test = True;
if(test):
    print("is it true?")
```

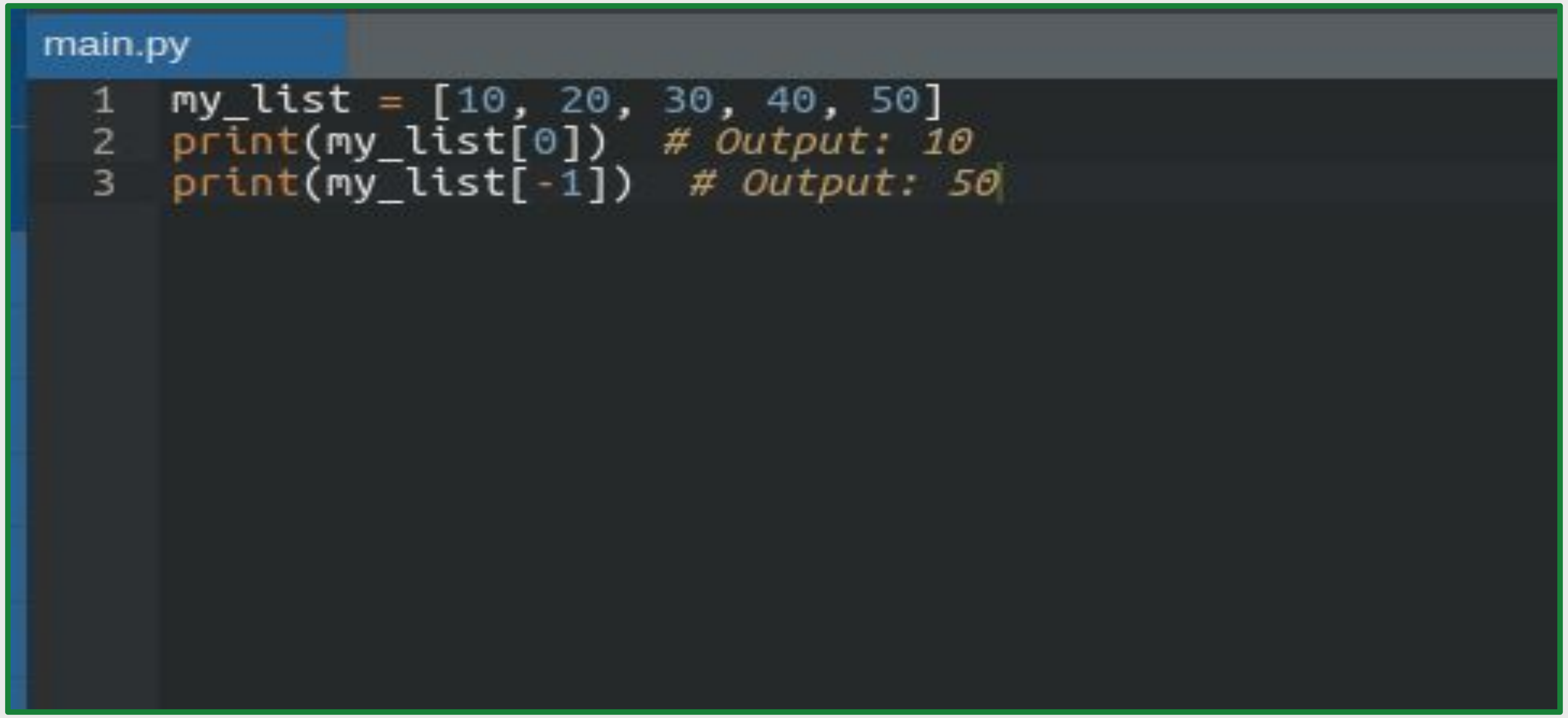
true or false.



```python

#boolean

is_python_fun = False

if is_python_fun:
    print("is it false?")
else:
    print("No, it's False!")
```

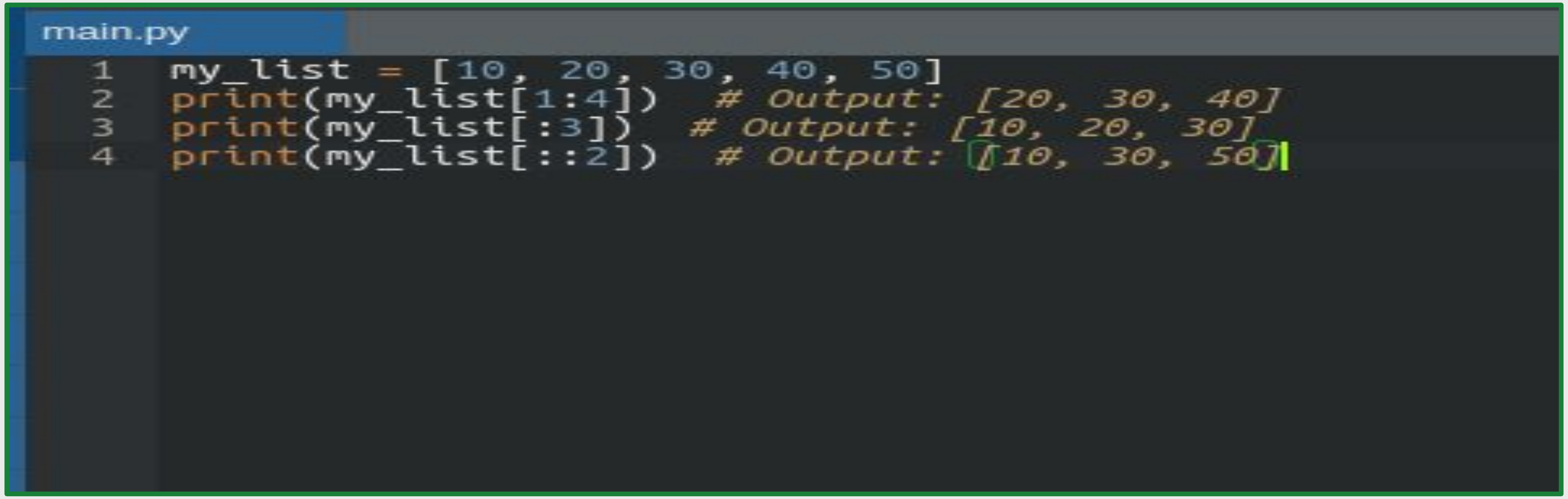
slicing & indexing

- Indexing is the process of accessing individual elements in a sequence (e.g., lists, strings, or tuples).
- Python uses zero-based indexing: the first element has index 0.
- Negative indices allow access from the end of the sequence.



```python
my_list = [10, 20, 30, 40, 50]
print(my_list[0])  # Output: 10
print(my_list[-1])  # Output: 50
```

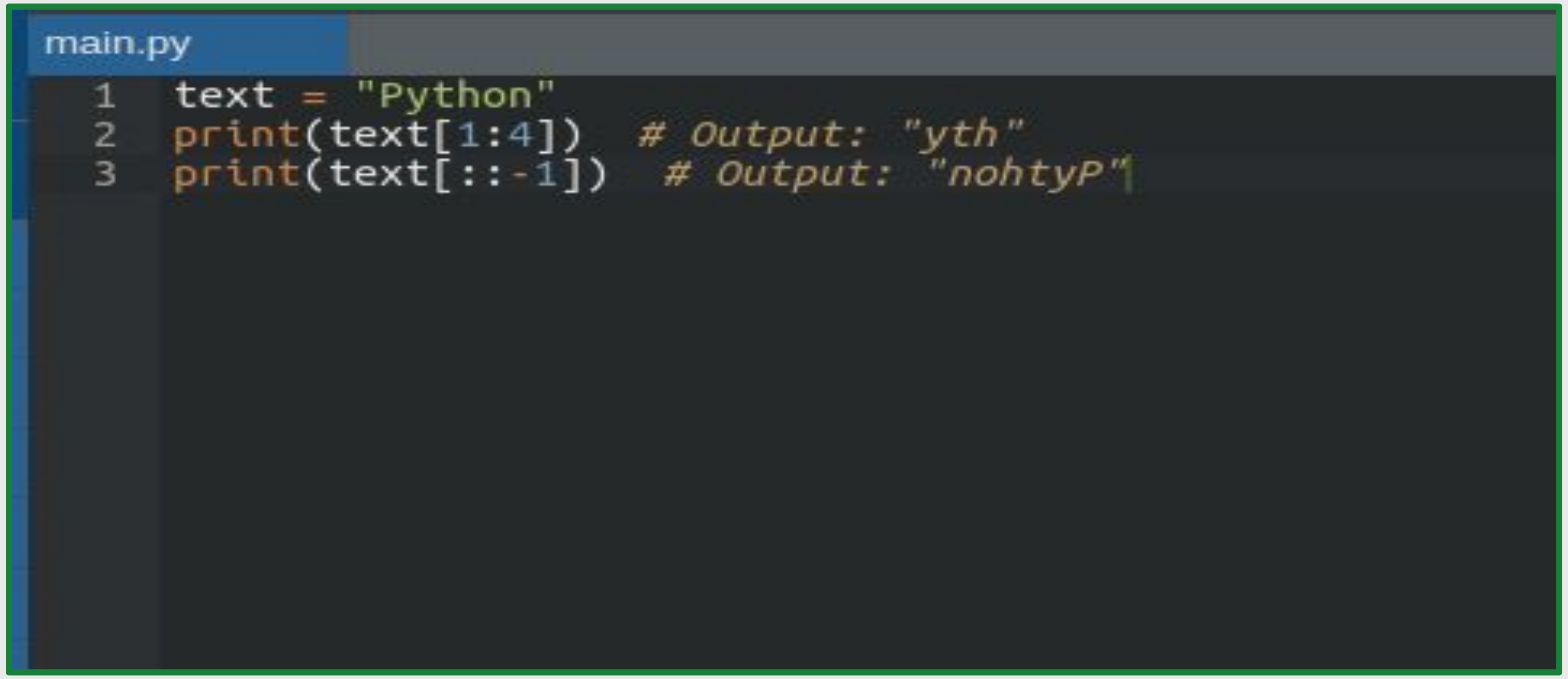
- Slicing extracts a portion of a sequence.
- Syntax: sequence[start:stop:step]
  - start: The starting index (inclusive).
  - stop: The ending index (exclusive).
  - step: The interval between indices (default is 1).



```python
my_list = [10, 20, 30, 40, 50]
print(my_list[1:4])  # Output: [20, 30, 40]
print(my_list[:3])  # Output: [10, 20, 30]
print(my_list[::2])  # Output: [10, 30, 50]
```

Key Points to Remember:

1. Omitting start begins at the first element.
2. Omitting stop continues to the end of the sequence.
3. Omitting step defaults to 1.
4. Negative step values reverse the sequence.



main.py

```python
text = "Python"
print(text[1:4])  # Output: "yth"
print(text[::-1])  # Output: "nohtyP"
```

operators.

operators.

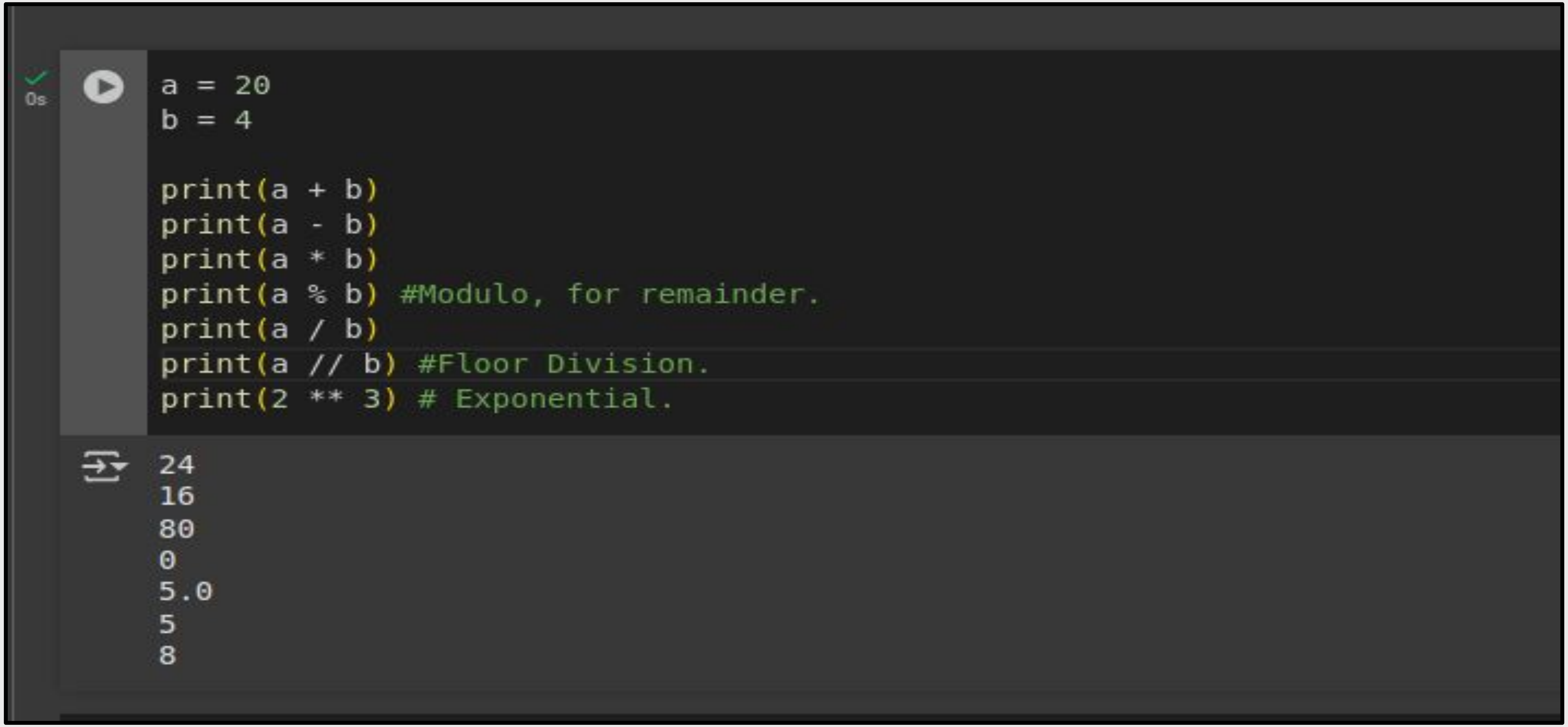
different types.

- ❑ arithmetic
- ❑ comparison
- ❑ logical
- ❑ assignment
- ❑ membership
- ❑ identity

operators.

operators	keywords / symbols	purpose.
arithmetic operators	**+, -, *, /, %, **, //**	perform mathematical calculations.
comparison operators	**==, !=, >, <, >=, <=**	compare values and return boolean.
logical operators	**and, or, not**	combine or invert boolean values.
assignment operators	**=, +=, -=, *=, /=, %=**	assign or modify variable values.
membership operators	**in, not in**	test for membership in sequences like lists.
identity operators	**is, is not**	check if two objects refer to the same memory location.

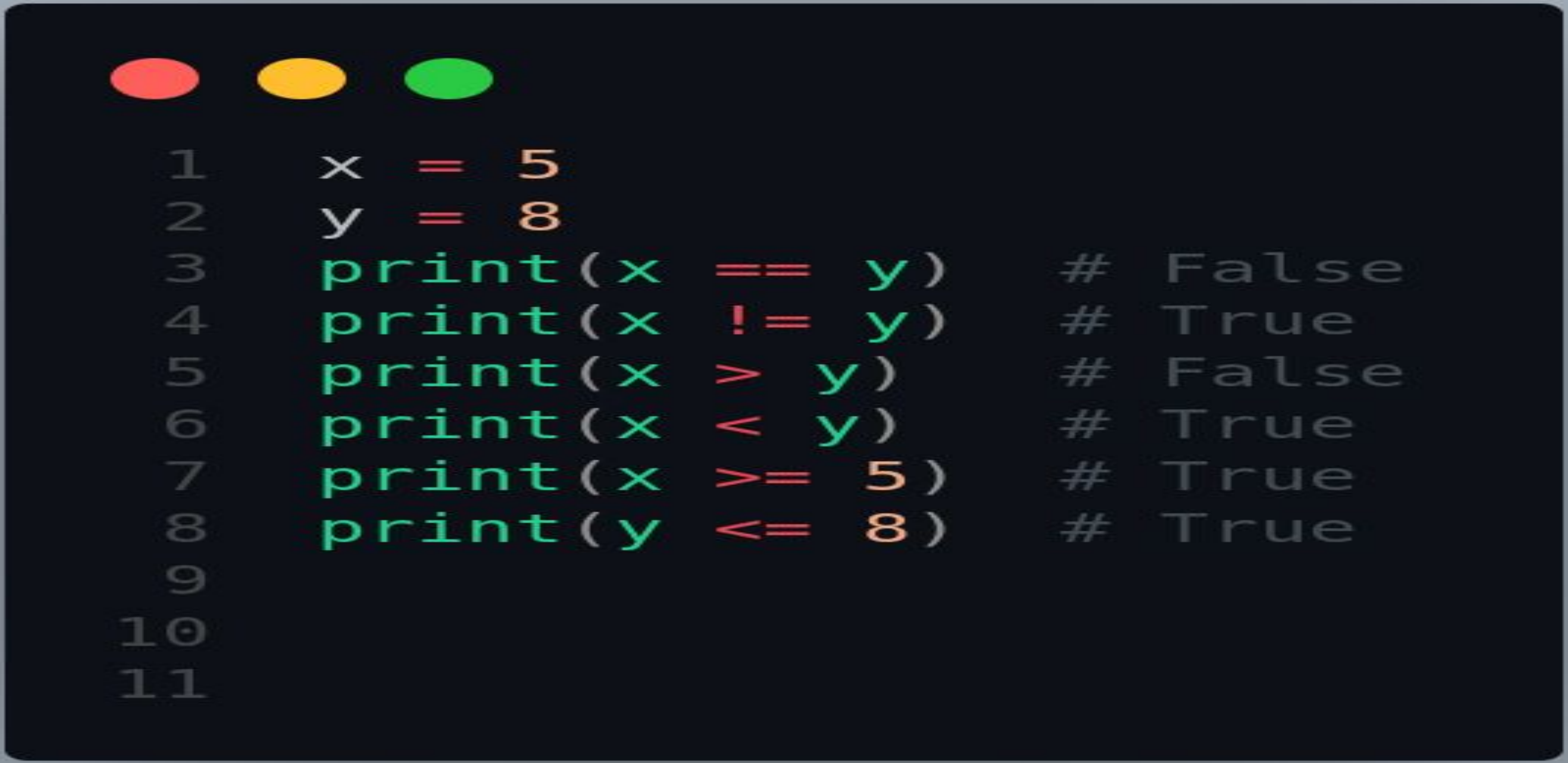
arithmetic.



```
a = 20
b = 4

print(a + b)
print(a - b)
print(a * b)
print(a % b) #Modulo, for remainder.
print(a / b)
print(a // b) #Floor Division.
print(2 ** 3) # Exponential.
```

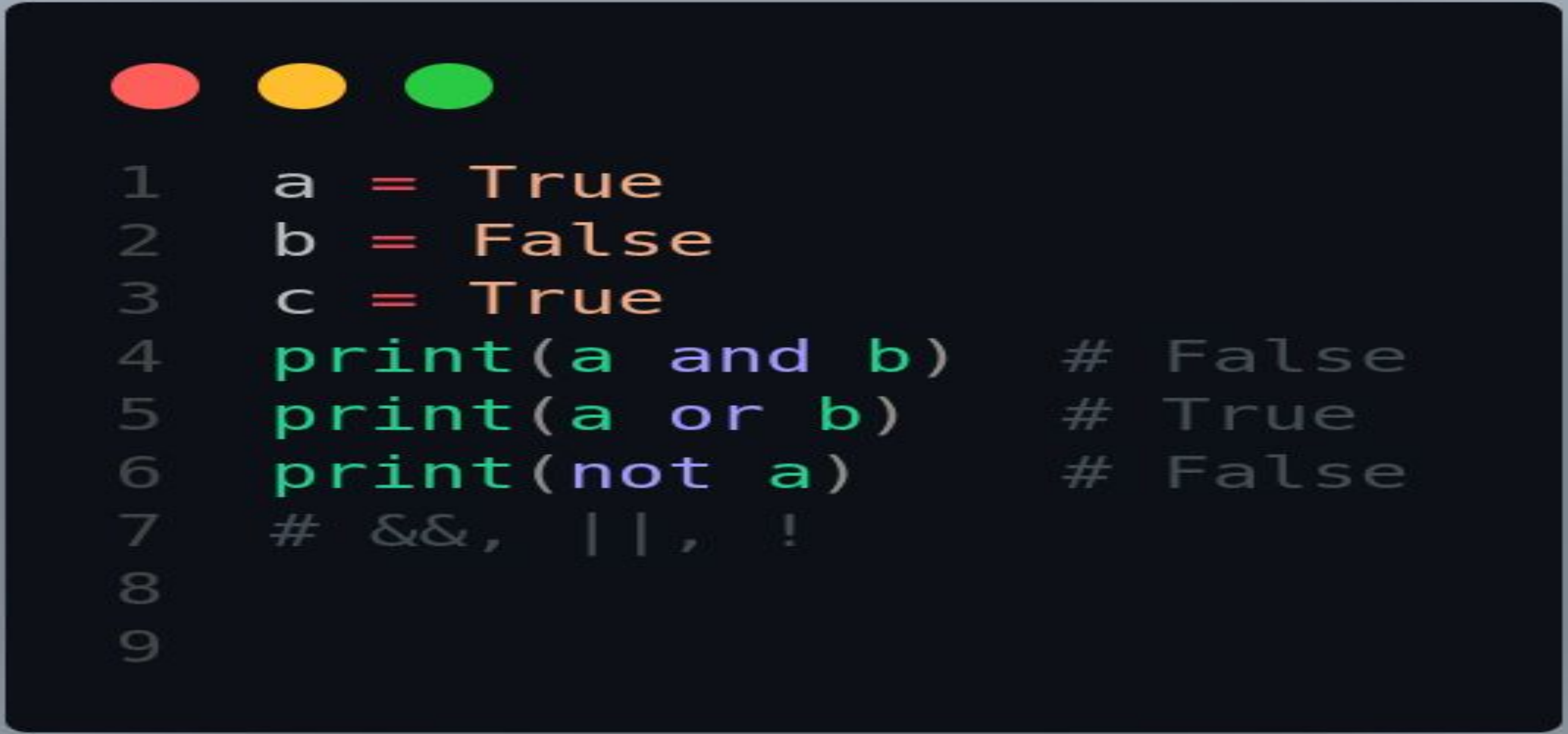
```
24
16
80
0
5.0
5
8
```

comparison.



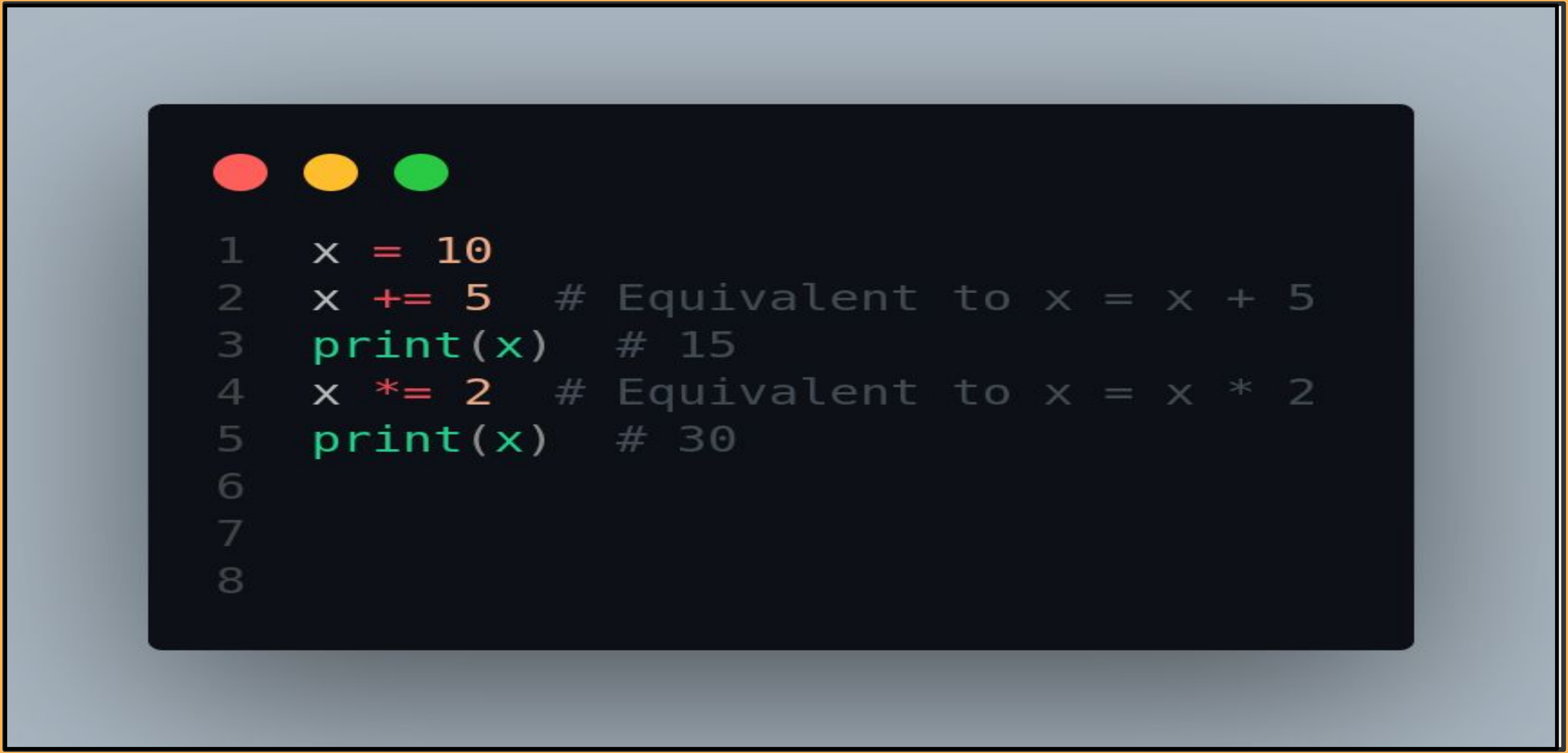
```
x = 5
y = 8
print(x == y)   # False
print(x != y)   # True
print(x > y)    # False
print(x < y)    # True
print(x >= 5)   # True
print(y <= 8)   # True
```

logical.



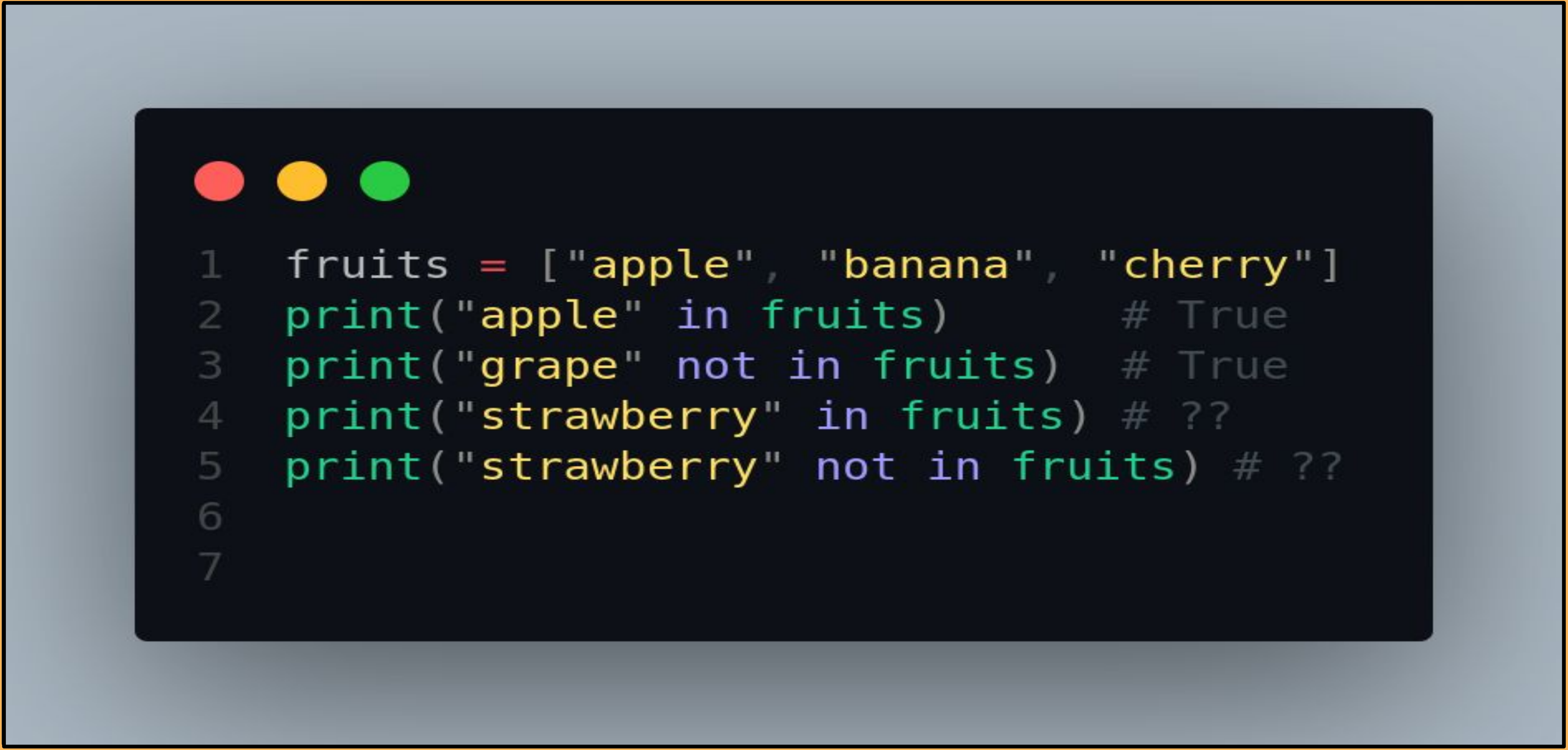
```python
a = True
b = False
c = True
print(a and b)  # False
print(a or b)   # True
print(not a)    # False
# &&, ||, !
```

assignment.



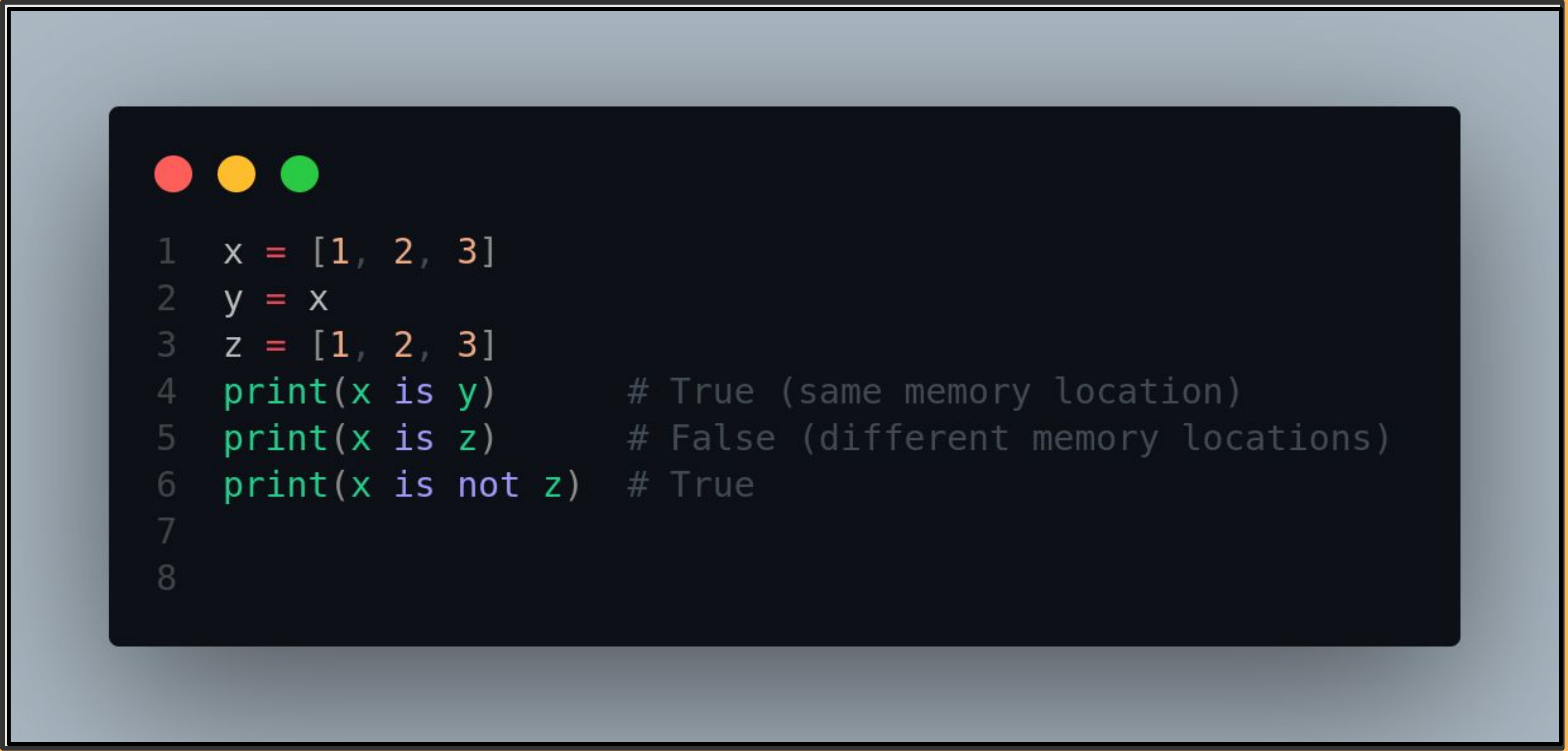
```python
x = 10
x += 5  # Equivalent to x = x + 5
print(x)  # 15
x *= 2  # Equivalent to x = x * 2
print(x)  # 30
```

membership.



```python
fruits = ["apple", "banana", "cherry"]
print("apple" in fruits)      # True
print("grape" not in fruits)  # True
print("strawberry" in fruits) # ??
print("strawberry" not in fruits) # ??


```

identity.



```
x = [1, 2, 3]
y = x
z = [1, 2, 3]
print(x is y)       # True (same memory location)
print(x is z)       # False (different memory locations)
print(x is not z)   # True
```

aspect	membership	identity
Operators	in, not in	is, is not
Purpose	To check if a value is present in a sequence (e.g., list, string, tuple, set, dictionary keys).	To check if two variables point to the same memory location.
Usage	Verifies membership in sequences or collections.	Compares object identities (not their values).
Use Cases	- Checking if an item exists in a list, tuple, dictionary keys, or sets. - Validating inputs.	- Determining if two variables reference the same object. - Used in optimizing memory usage.
Importance in Programming	- Efficiently validates presence/absence of elements in collections. - Often used in conditions and loops.	- Avoids confusion between identical objects and identical references. - Crucial for memory management in Python.
Performance	Fast for small collections but depends on the size of the sequence.	Faster since it directly compares object memory locations.
Limitations	- Works only for iterable data types. - Cannot check non-sequential data.	- Misinterpreted for value comparison - Only useful for identity checks.
Visual Analogy	"Does this item belong to this collection?"	"Are these two items the exact same object?"

control flow.

control-flow.

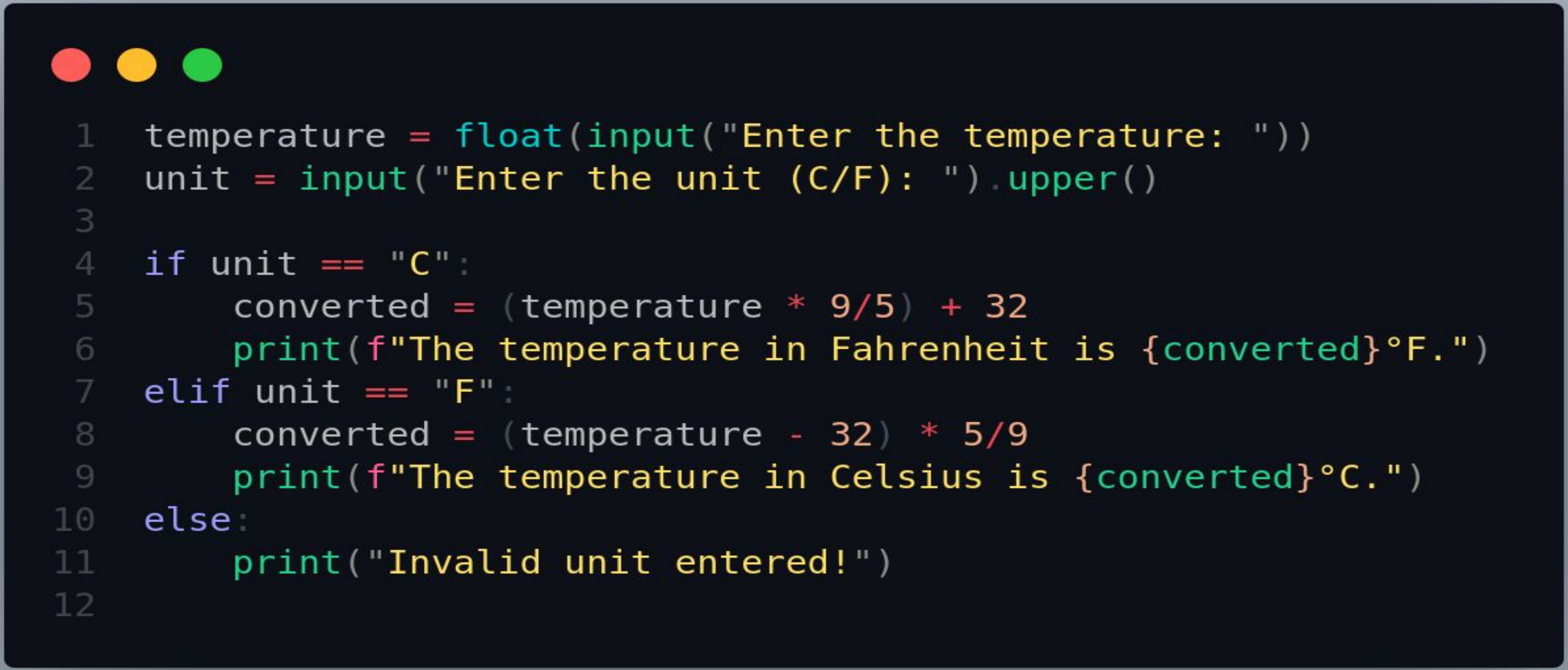
conditional statements.

if
elif
else

loops

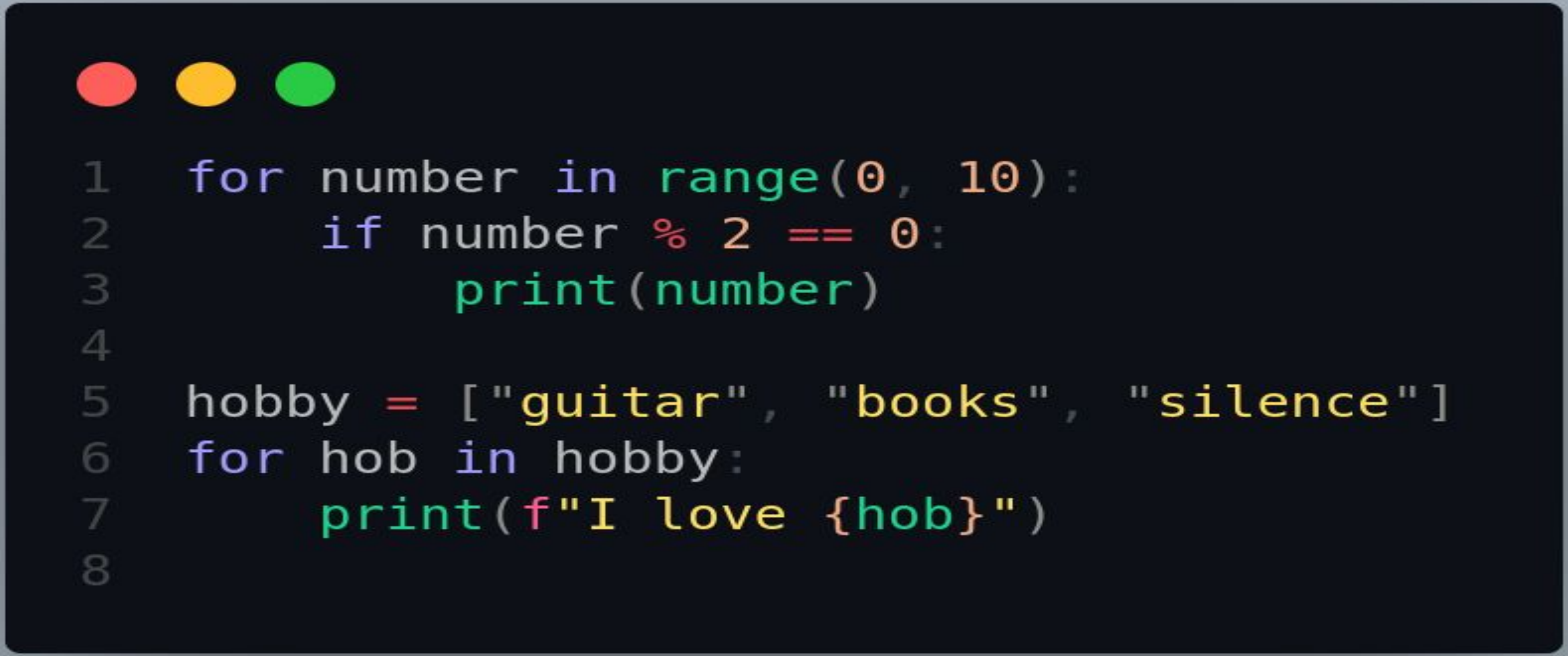
for
while

conditional flow.



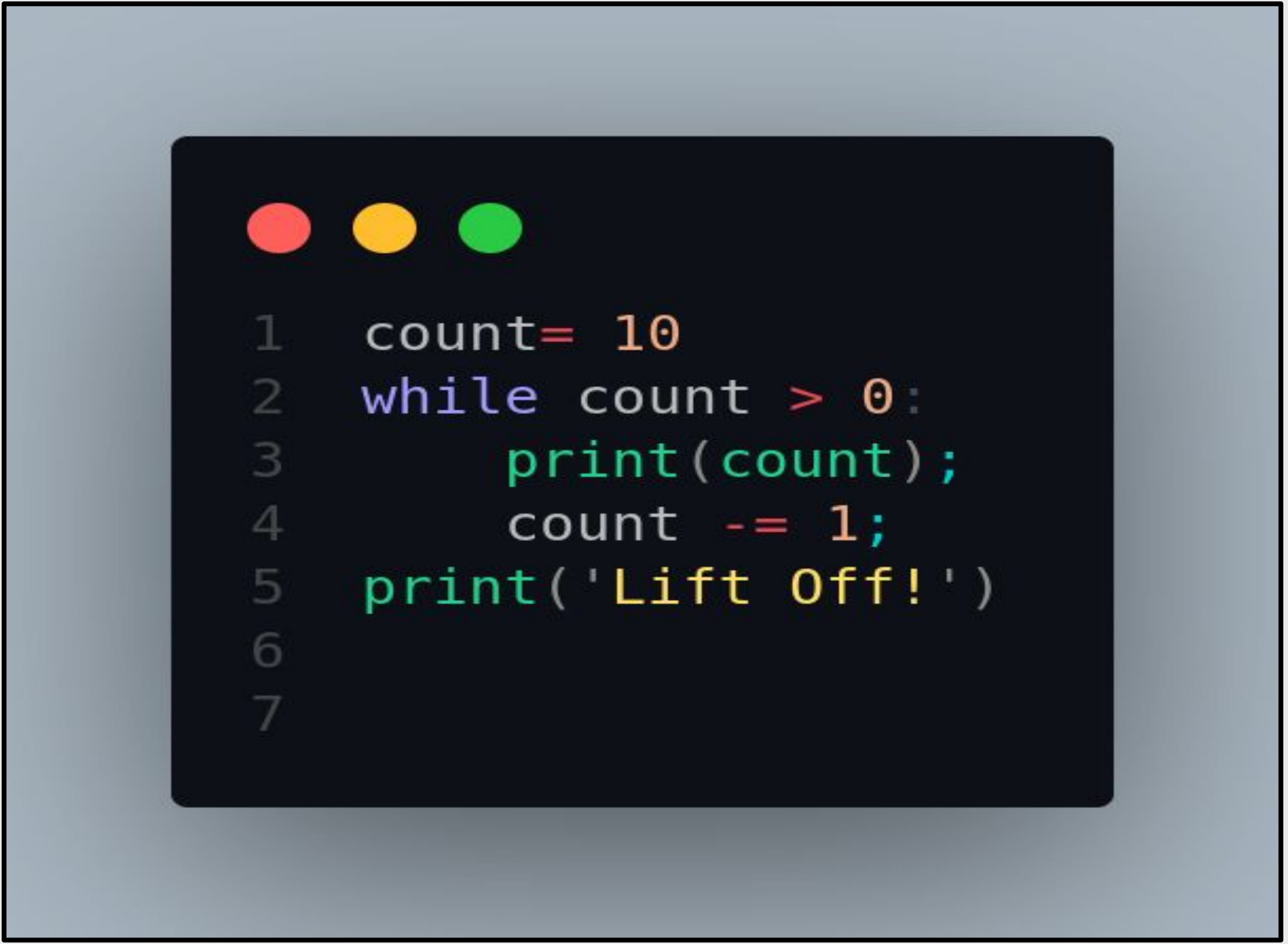
```python
temperature = float(input("Enter the temperature: "))
unit = input("Enter the unit (C/F): ").upper()

if unit == "C":
    converted = (temperature * 9/5) + 32
    print(f"The temperature in Fahrenheit is {converted}°F.")
elif unit == "F":
    converted = (temperature - 32) * 5/9
    print(f"The temperature in Celsius is {converted}°C.")
else:
    print("Invalid unit entered!")
```

loops.

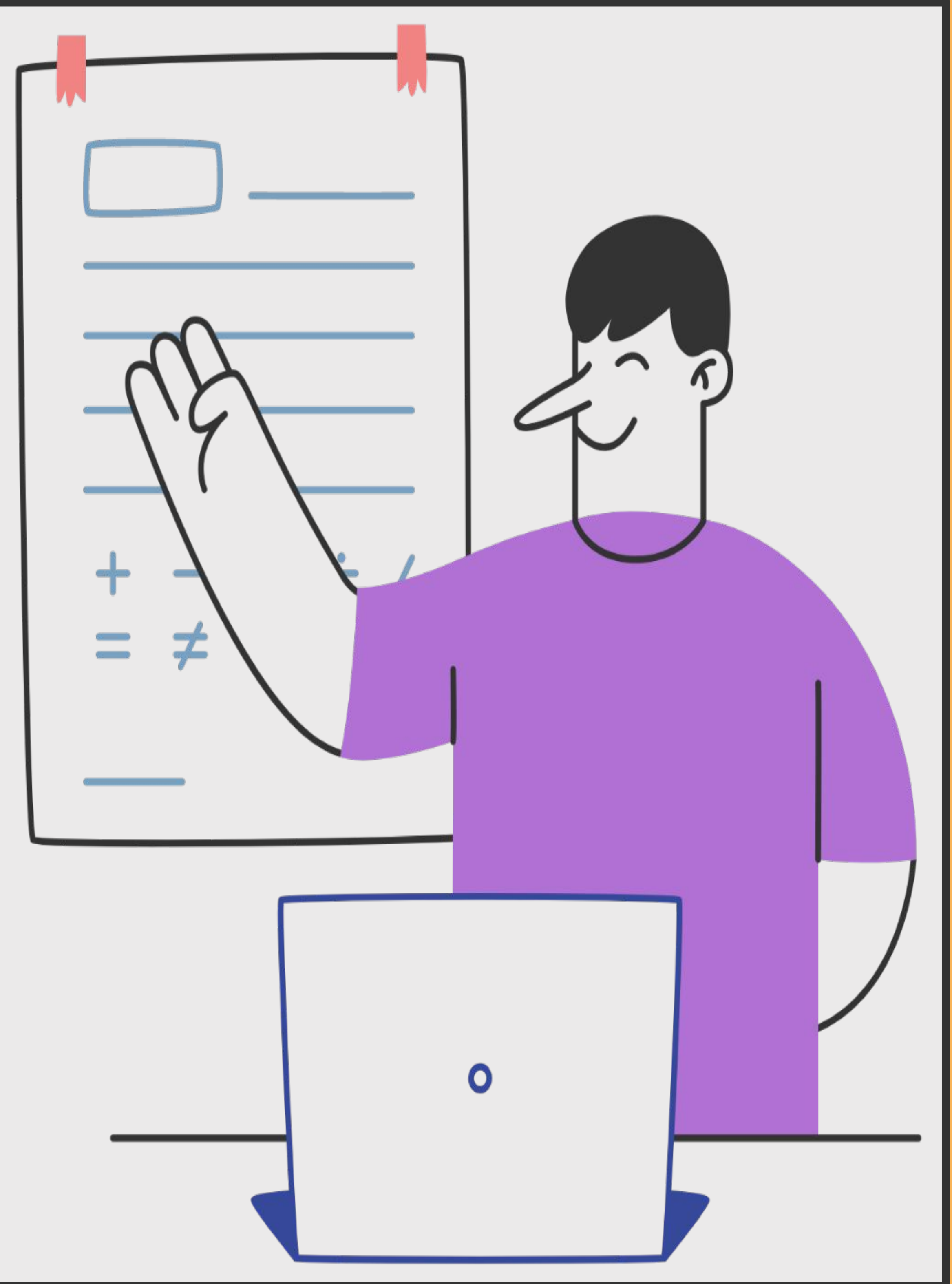


```python
for number in range(0, 10):
    if number % 2 == 0:
        print(number)

hobby = ["guitar", "books", "silence"]
for hob in hobby:
    print(f"I love {hob}")
```

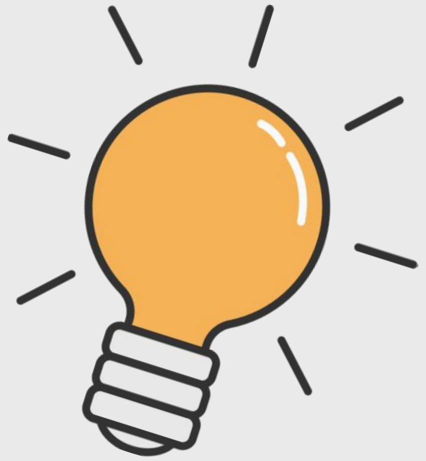
loops.



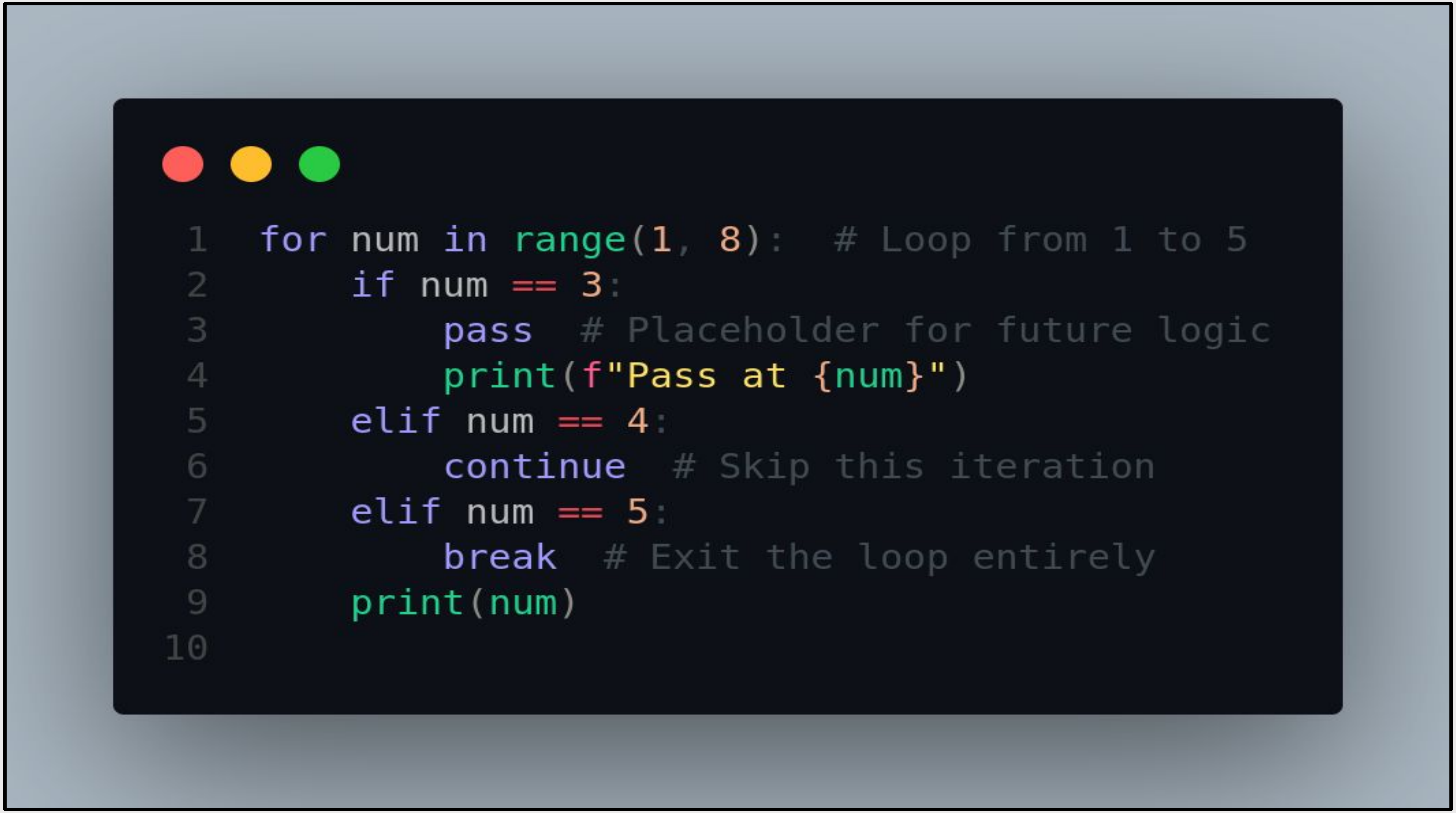
```
count= 10
while count > 0:
    print(count);
    count -= 1;
print('Lift Off!')
```



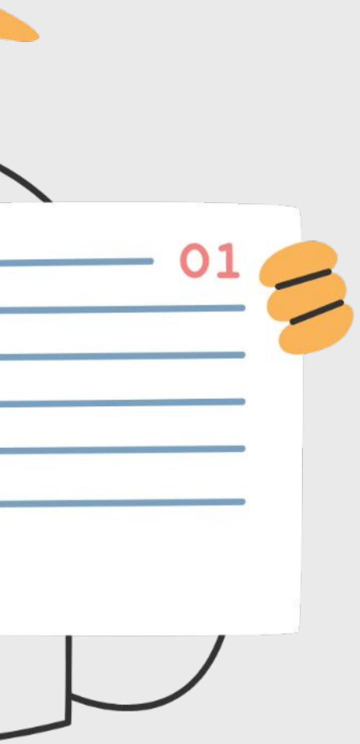
handling the loops.



break	stop then and there & exit the loop.
continue	skip the particular iteration & continues with next iteration. (if condition matches.)
pass	does nothing, placeholder.



```python
for num in range(1, 8):  # Loop from 1 to 5
    if num == 3:
        pass  # Placeholder for future logic
        print(f"Pass at {num}")
    elif num == 4:
        continue  # Skip this iteration
    elif num == 5:
        break  # Exit the loop entirely
    print(num)

```

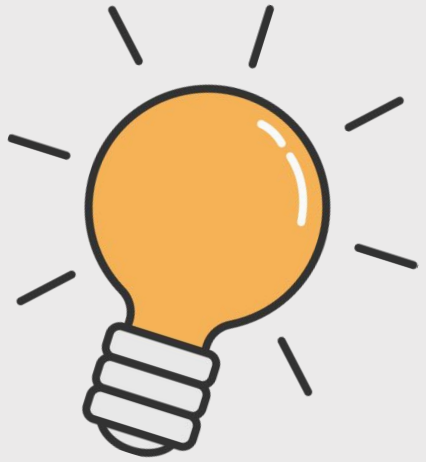


aspect	break	continue	pass
Definition	Exits the nearest enclosing loop immediately.	Skips the rest of the current loop iteration and moves to the next iteration.	Does nothing; it's a placeholder for future code.
Purpose	To terminate a loop prematurely.	To bypass part of a loop's body for specific conditions.	To provide syntactically valid code where no action is needed.
Effect on Loop	Ends the loop entirely.	Continues with the next iteration of the loop.	No effect; the loop continues as usual.
Common Usage	- Exit loops when a specific condition is met.	- Skip specific iterations in a loop.	- Placeholder for code not yet implemented.
Performance Impact	Stops further iterations, potentially saving resources.	Executes fewer instructions per skipped iteration.	Has no impact as it does nothing.
Use Case Example	Use break when: - You've found the target in a search loop. - An error or exit condition occurs in a loop.	Use continue when: - You need to skip processing certain items but want to proceed with others.	Use pass when: - You need to write incomplete code or placeholders in functions, classes, or loops.
Analogy	Think of it as a "**Stop!**" sign.	Think of it as a "**Skip!**" sign.	Think of it as a "**Do nothing for now.**" sign.

functions.

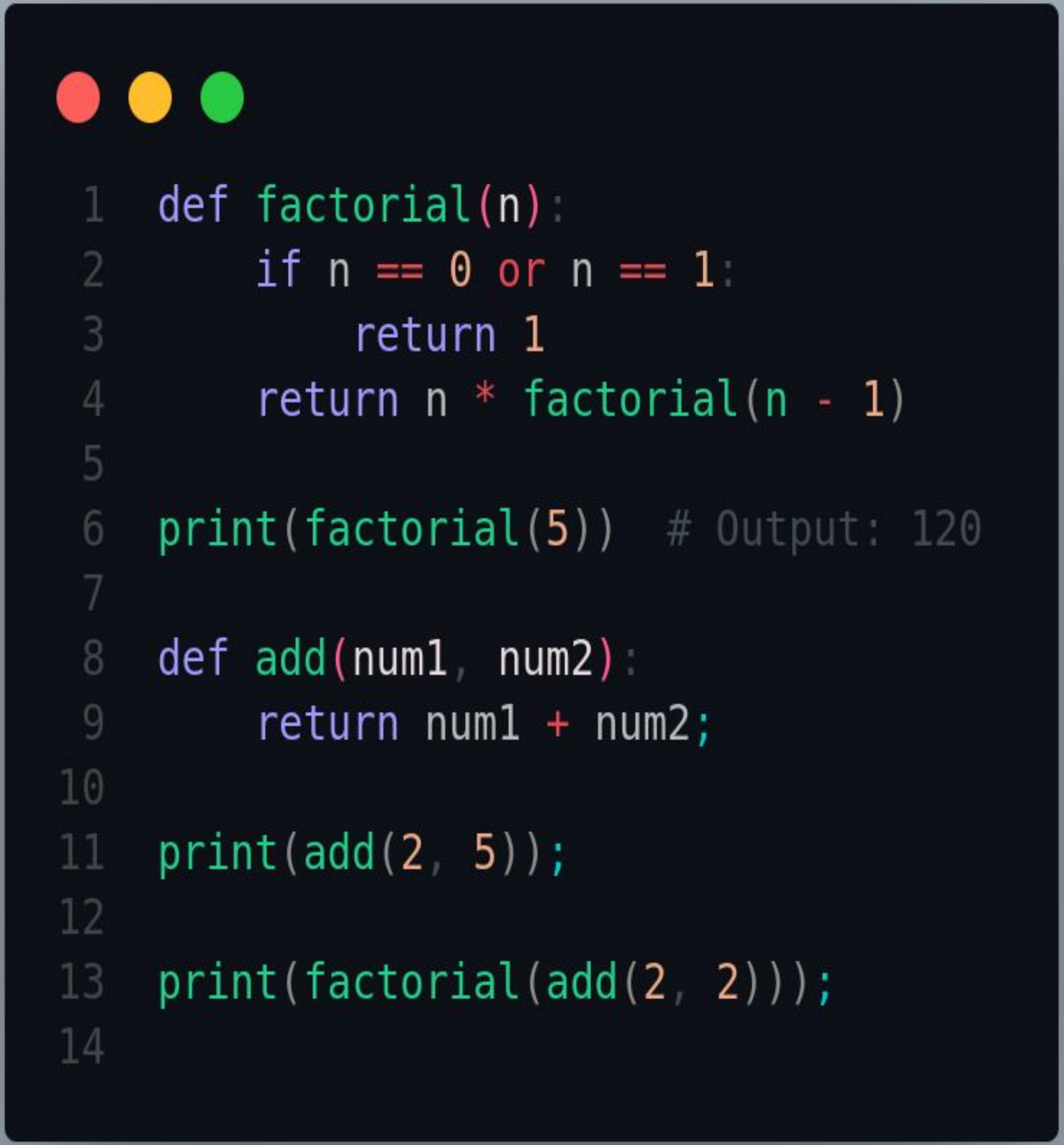
A function is a reusable block of code designed to perform a specific task.

```
def <function_name>(parameters):
    #re-usable code.
    return result;
```



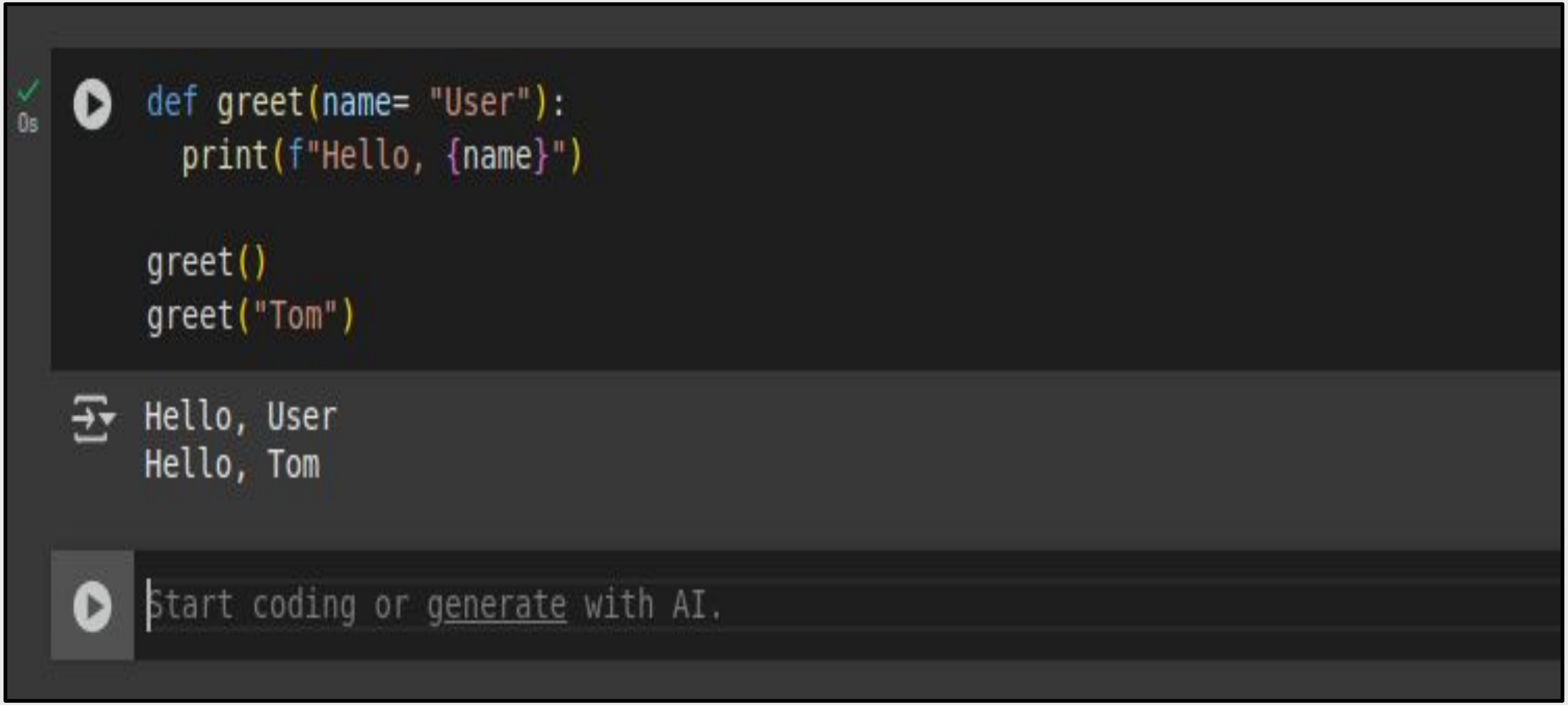
Benefits

- Code reusability.
- Modular design.
- Easier debugging and maintenance.



```python
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # Output: 120

def add(num1, num2):
    return num1 + num2;

print(add(2, 5));

print(factorial(add(2, 2)));

```

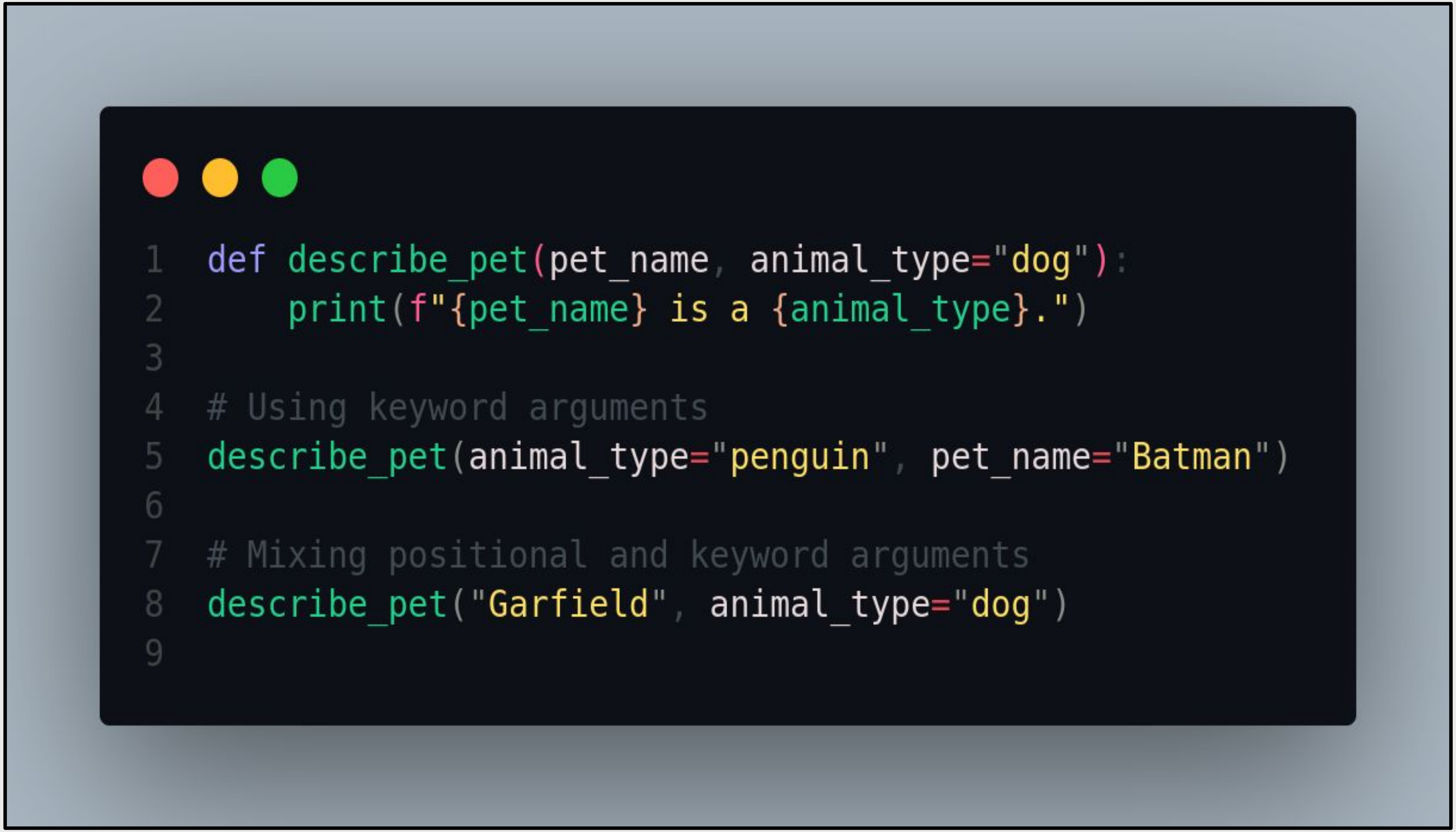
Feature	Use Case
default parameters	when certain arguments have common default values (e.g., default greeting name).
keyword arguments	to enhance readability and manage functions with many parameters.
*args	when the number of positional arguments is unknown (e.g., summing multiple numbers).
**kwargs	when the number of named arguments is unknown or optional (e.g., flexible configuration).



```
def greet(name= "User"):
  print(f"Hello, {name}")

greet()
greet("Tom")
```

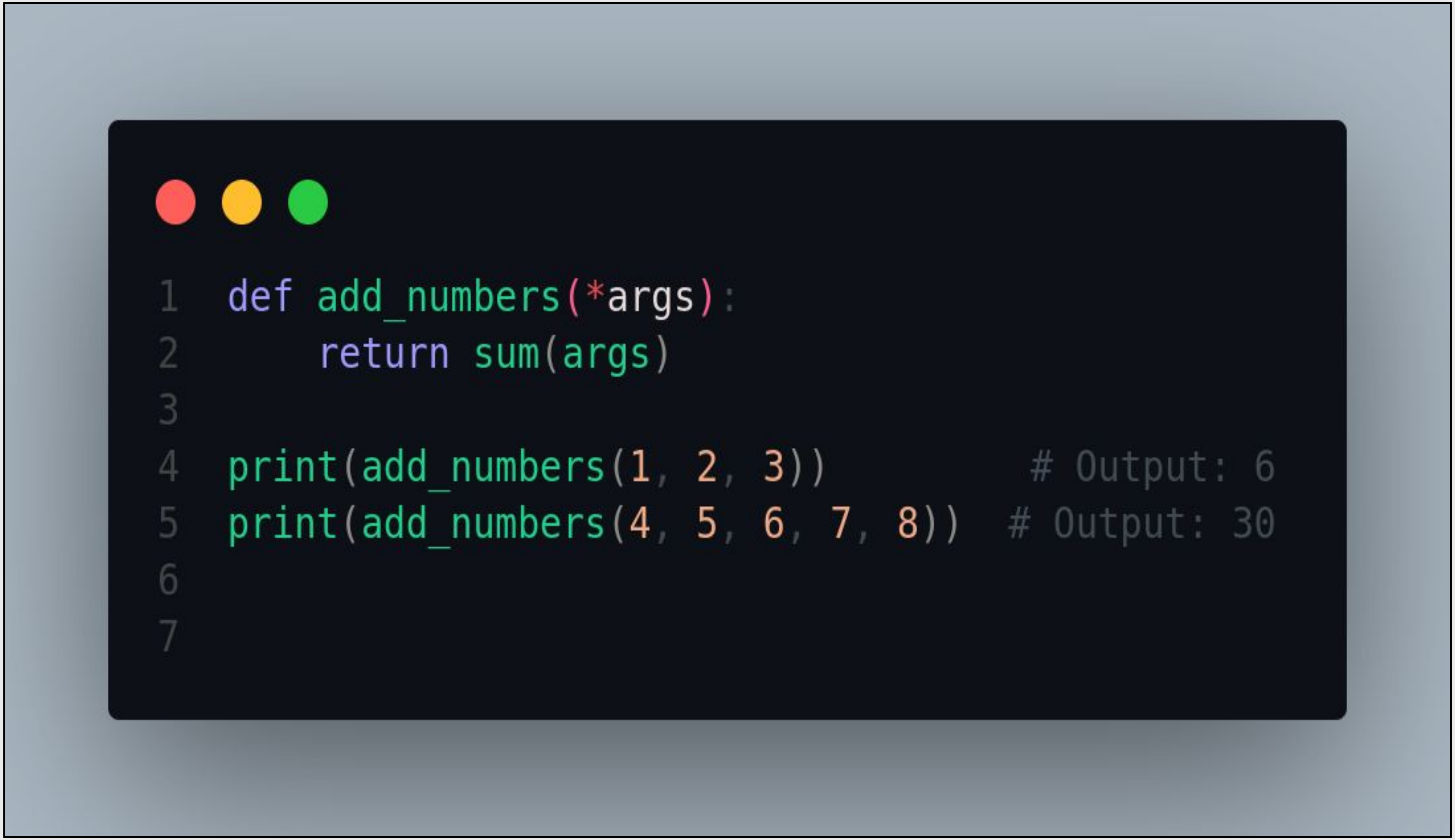
```
Hello, User
Hello, Tom
```

default arguments.



```python
def describe_pet(pet_name, animal_type="dog"):
    print(f"{pet_name} is a {animal_type}.")

# Using keyword arguments
describe_pet(animal_type="penguin", pet_name="Batman")

# Mixing positional and keyword arguments
describe_pet("Garfield", animal_type="dog")
```

keywords arguments.



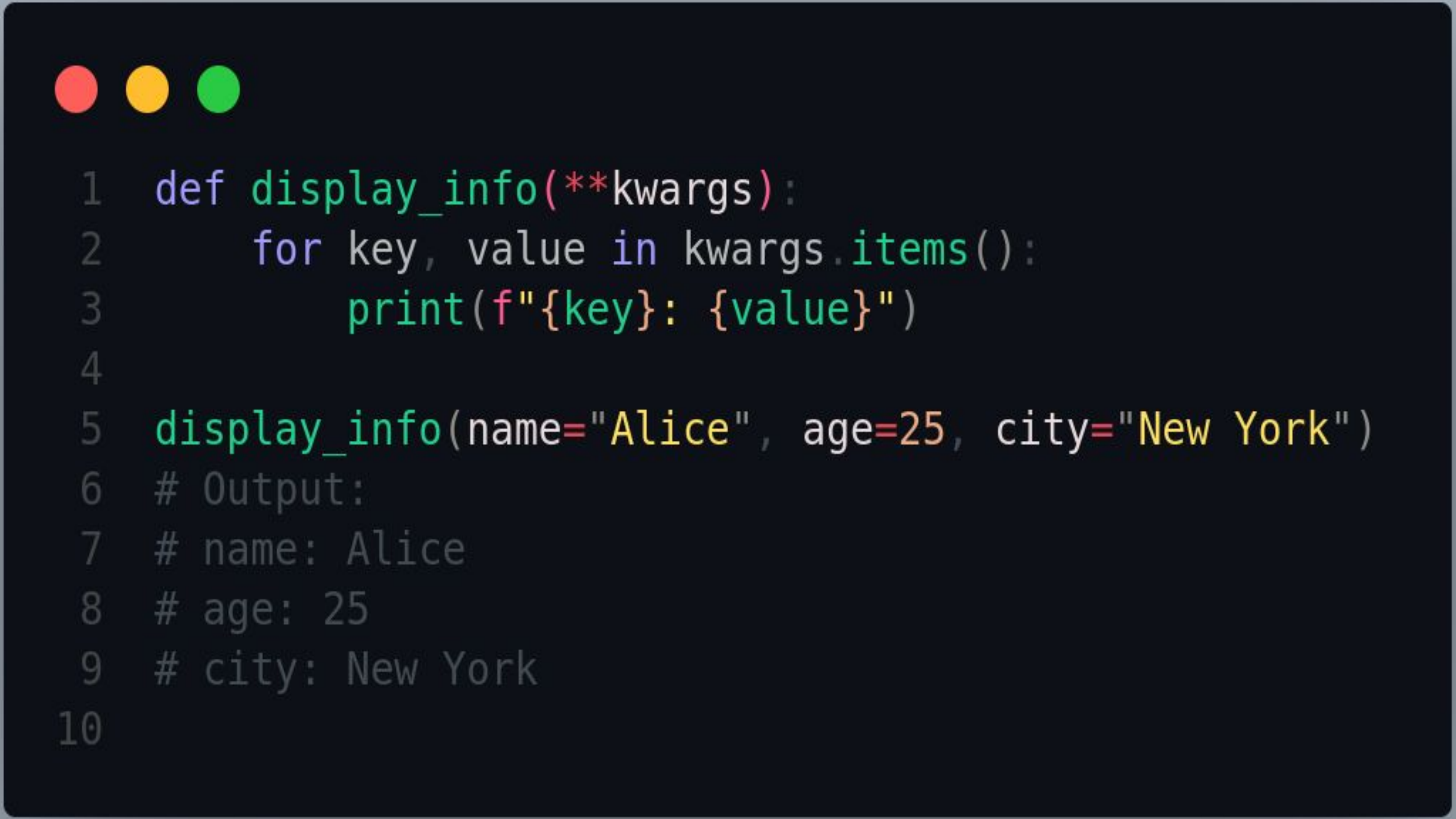
```python
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3))         # Output: 6
print(add_numbers(4, 5, 6, 7, 8))  # Output: 30


```

*args

Allows passing a variable number of positional arguments to a function. The arguments are captured as a tuple.

Key Points:

- Useful when the exact number of arguments is not known in advance.
- The parameter name *args is a convention, but any name preceded by * works.

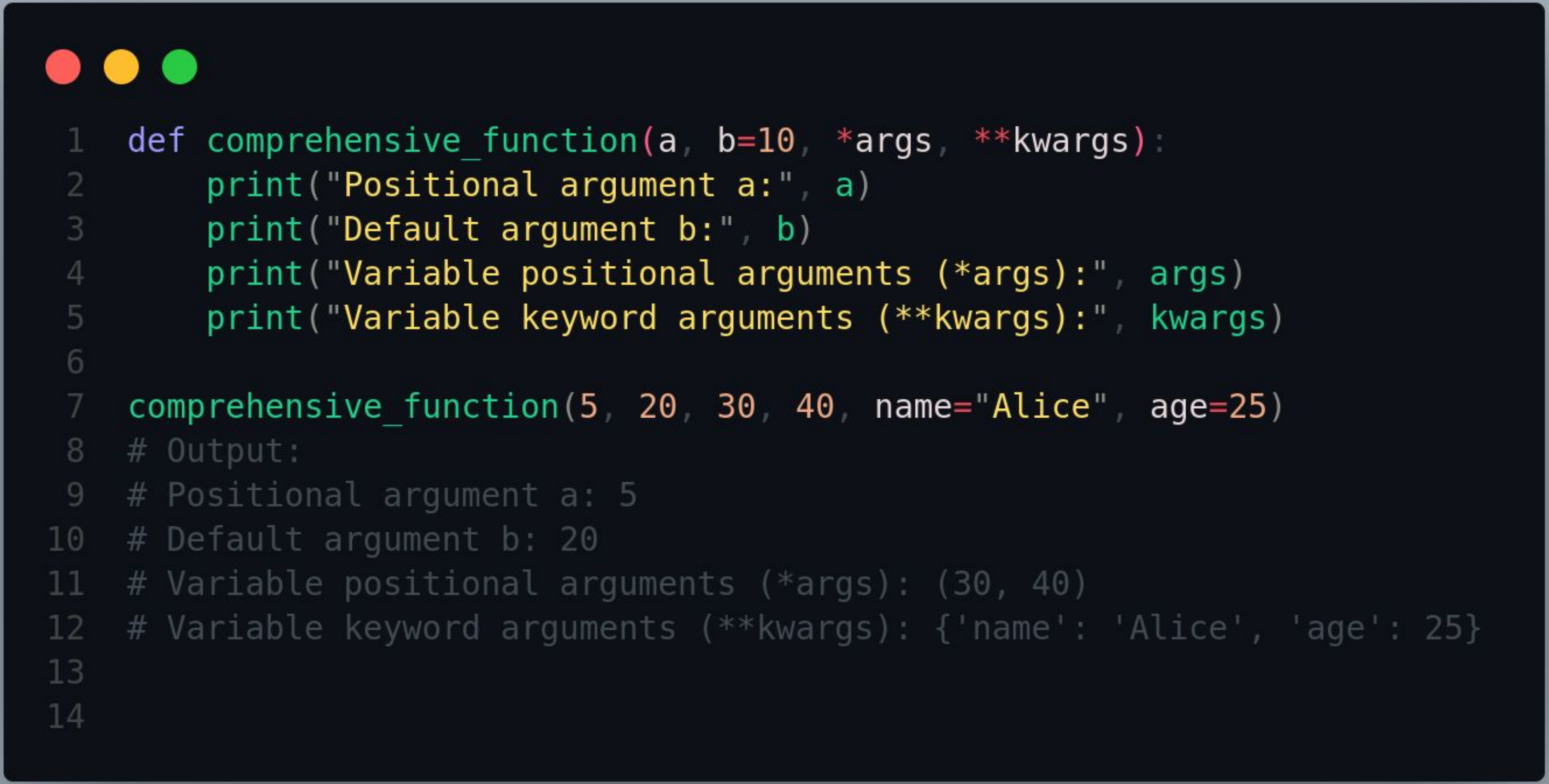


```python
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

display_info(name="Alice", age=25, city="New York")
# Output:
# name: Alice
# age: 25
# city: New York
```

*kwargs

Allows passing a variable number of keyword arguments to a function.
The arguments are captured as a dictionary.
Key Points:

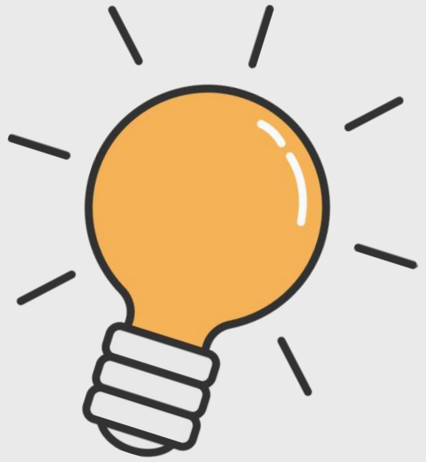
- Useful for functions that need flexible keyword arguments.
- The parameter name **kwargs is a convention, but any name preceded by ** works.

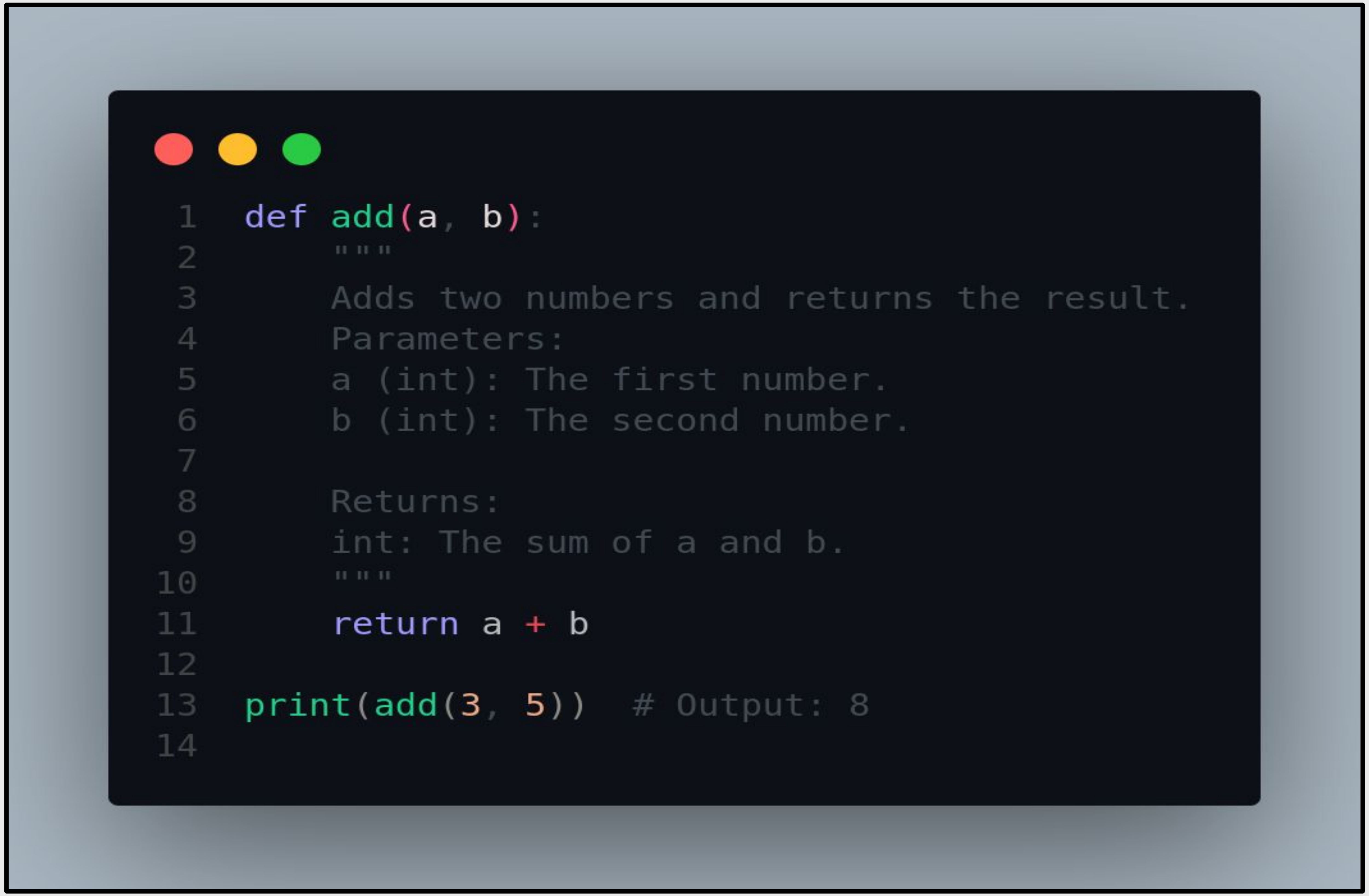


```python
def comprehensive_function(a, b=10, *args, **kwargs):
    print("Positional argument a:", a)
    print("Default argument b:", b)
    print("Variable positional arguments (*args):", args)
    print("Variable keyword arguments (**kwargs):", kwargs)

comprehensive_function(5, 20, 30, 40, name="Alice", age=25)
# Output:
# Positional argument a: 5
# Default argument b: 20
# Variable positional arguments (*args): (30, 40)
# Variable keyword arguments (**kwargs): {'name': 'Alice', 'age': 25}


```

Feature	Use Case
Default Parameters	When certain arguments have common default values (e.g., default greeting name).
Keyword Arguments	To enhance readability and manage functions with many parameters.
***args**	When the number of positional arguments is unknown (e.g., summing multiple numbers).
****kwargs**	When the number of named arguments is unknown or optional (e.g., flexible configuration).

docstrings in functions.



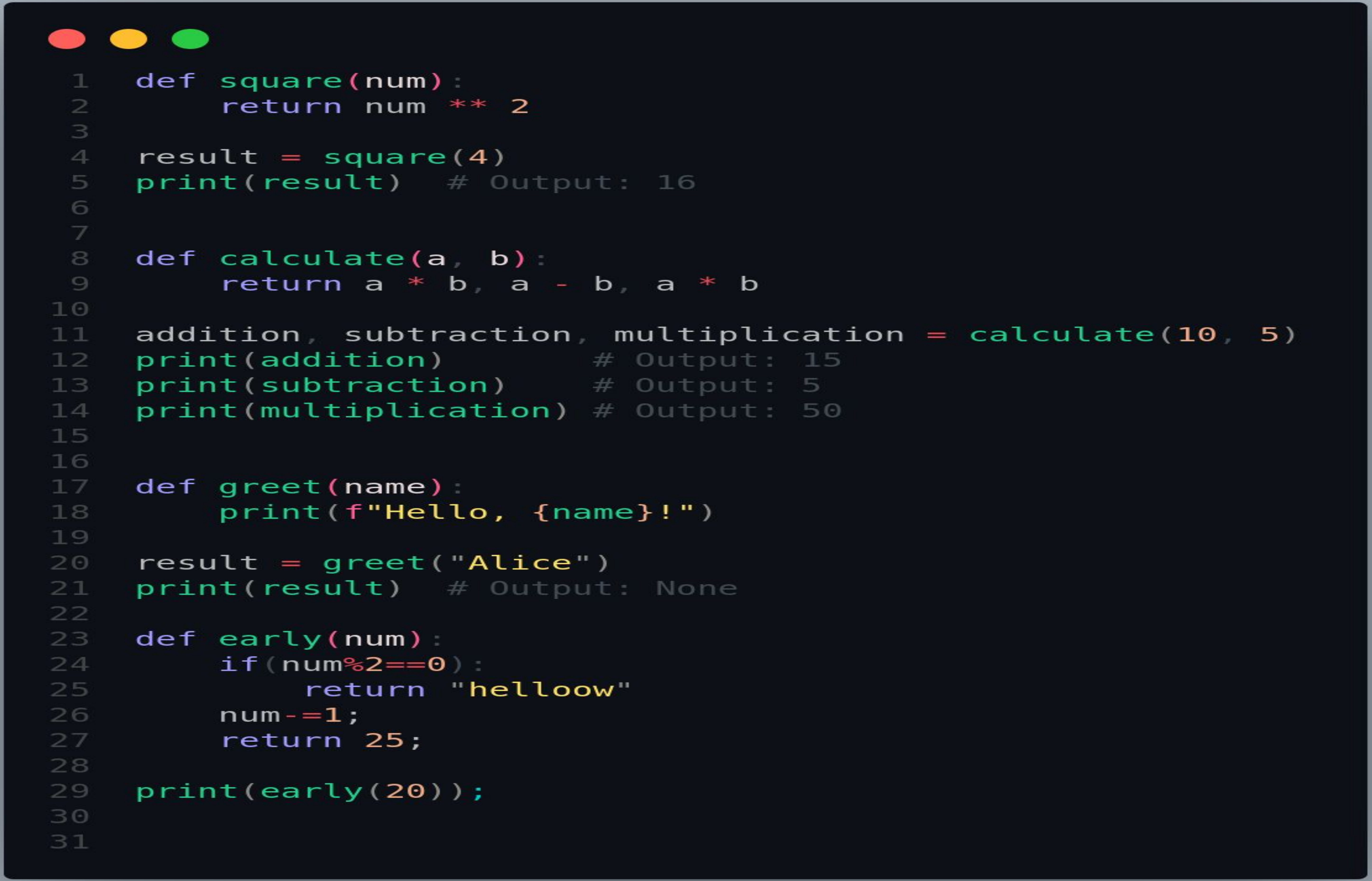


```python
def add(a, b):
    """
    Adds two numbers and returns the result.
    Parameters:
    a (int): The first number.
    b (int): The second number.

    Returns:
    int: The sum of a and b.
    """
    return a + b

print(add(3, 5))  # Output: 8
```

return.

- used in functions to send a value or multiple values back to the caller.
- it is one of the core concepts that makes functions useful for reusability and modularity.

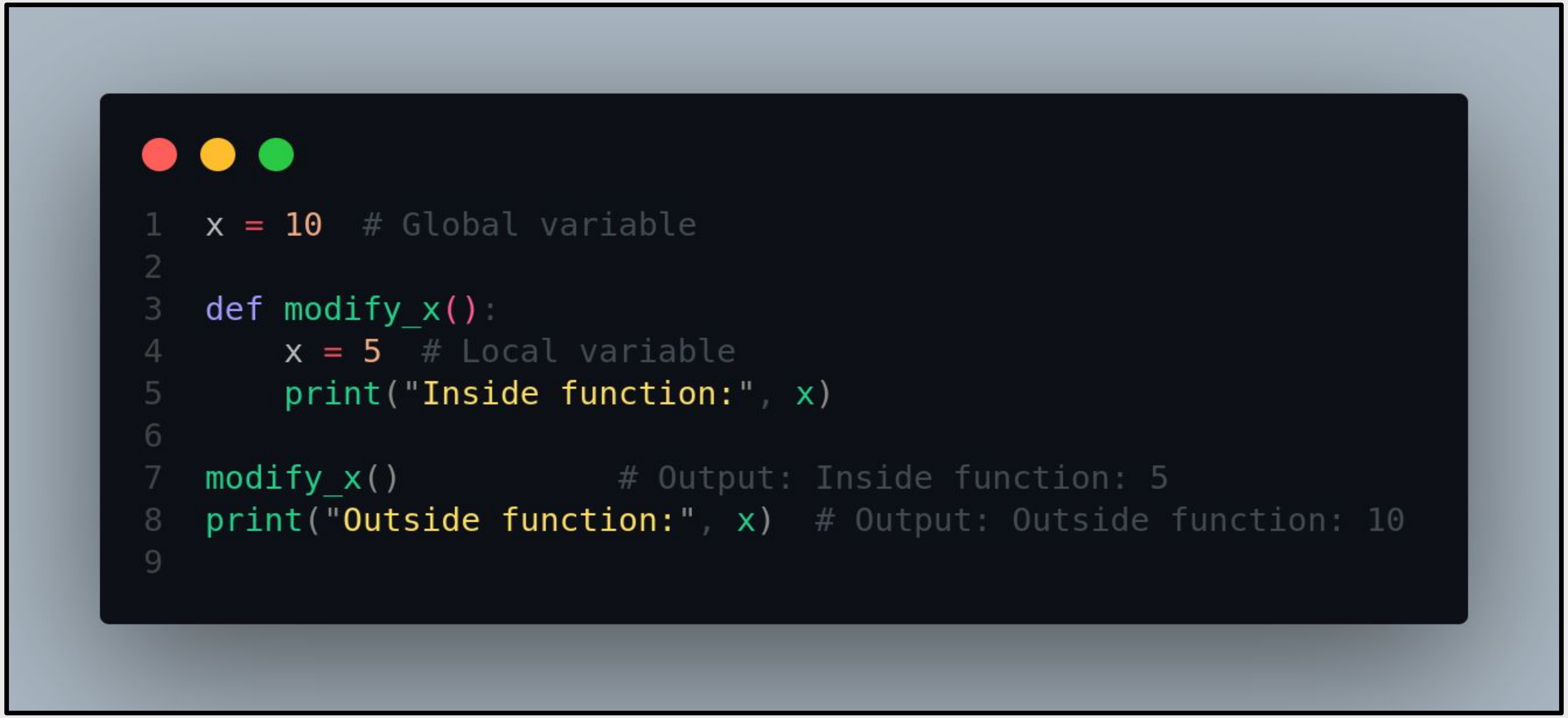
aspect	details
purpose	to exit a function and send a result back to the part of the program where the function was called.
default behavior	if return is omitted, the function returns None.
multiple values.	you can return multiple values as a tuple.



```python
def square(num):
    return num ** 2

result = square(4)
print(result)  # Output: 16


def calculate(a, b):
    return a * b, a - b, a * b

addition, subtraction, multiplication = calculate(10, 5)
print(addition)       # Output: 15
print(subtraction)    # Output: 5
print(multiplication) # Output: 50


def greet(name):
    print(f"Hello, {name}!")

result = greet("Alice")
print(result)  # Output: None

def early(num):
    if(num%2==0):
        return "helloow"
    num-=1;
    return 25;

print(early(20));
```

local & global

- local variables: defined inside a function, accessible only within that function. (block scope.)
- global variables: defined outside any function, accessible throughout the program.
- global keyword: used to modify global variables inside a function.



```python
x = 10  # Global variable

def modify_x():
    x = 5  # Local variable
    print("Inside function:", x)

modify_x()           # Output: Inside function: 5
print("Outside function:", x)  # Output: Outside function: 10

```

modules.

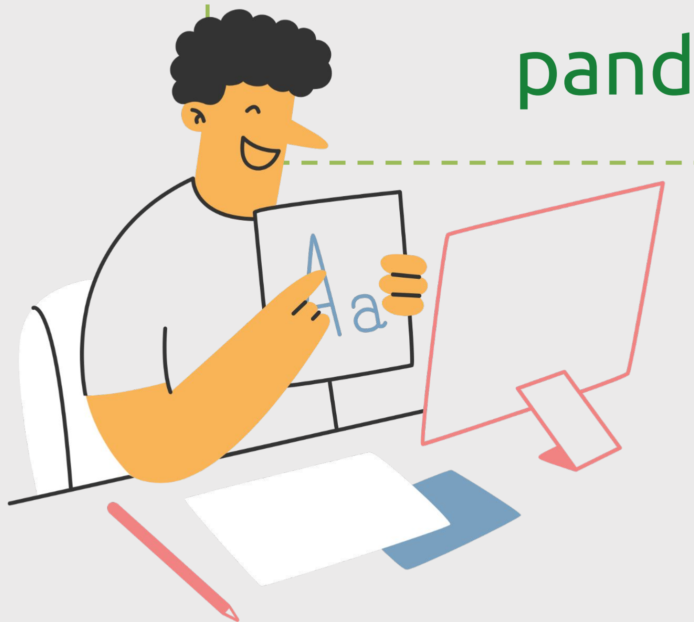
module is a file containing python definitions, functions, or classes that can be reused in other programs.

types of modules;

- **built-in modules**
  - pre-installed with Python.
    - math, os, random.
- **custom modules**
  - created by you, utilised in different files.
- **external modules**
  - curated for special tasks.
    - pandas, tensorflow.

advantages;

- makes code more organized and reusable.
- provides access to pre-written functionalities.



how to work with module.

ways to import modules.

import module_name

imports the entire module.

from module_name import specific_function

imports specific items from a module.

import module_name as alias

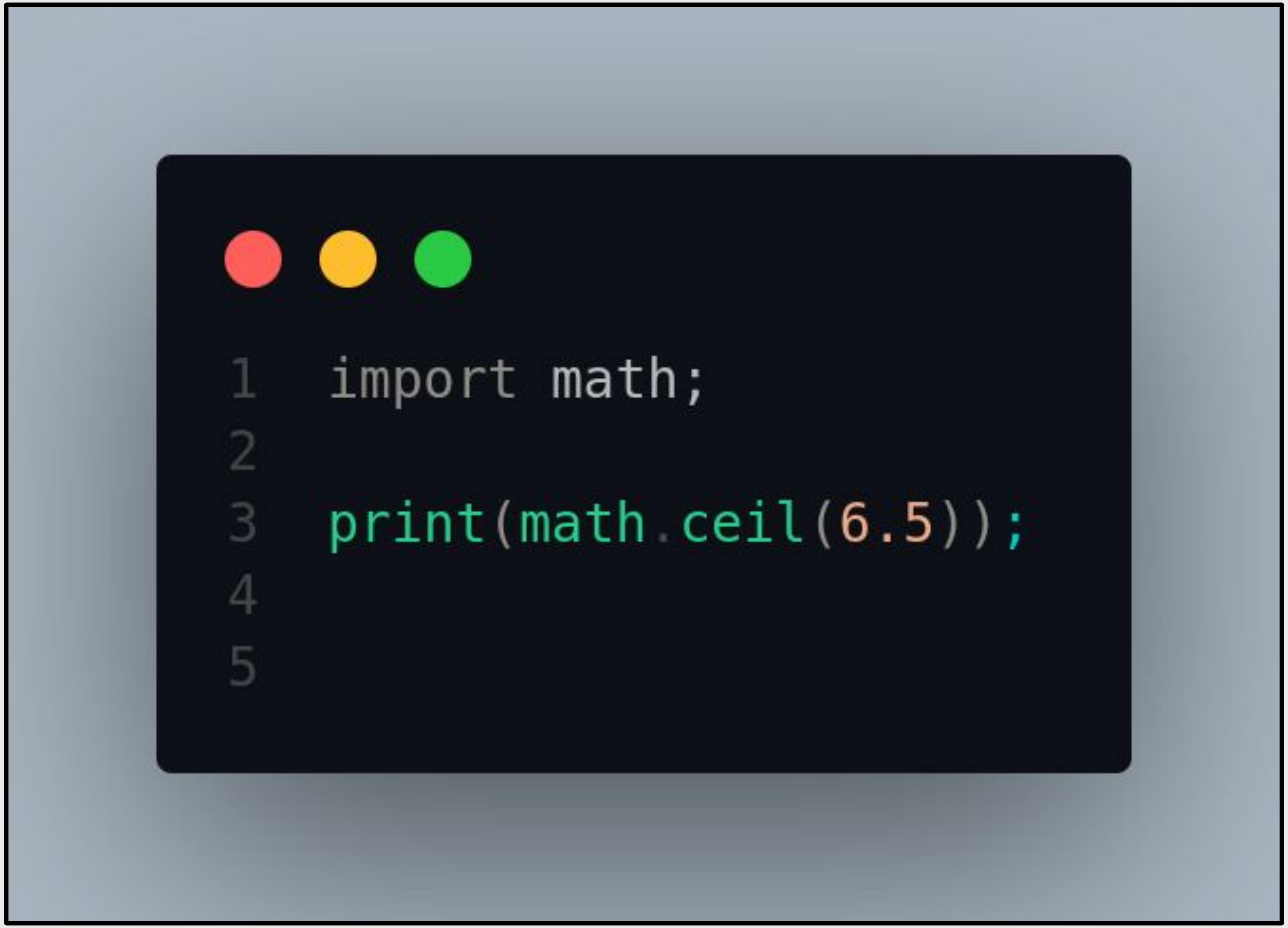
imports with an alias for easier reference

best practises;

use descriptive aliases for clarity.

import only what you need.

module	description	use case
math	Provides mathematical functions and constants.	Calculating square roots, trigonometry, logarithms.
os	Interfaces with the operating system.	Managing files, directories, and environment variables.
sys	Accesses system-specific parameters and functions.	Command-line arguments, interacting with the Python runtime.
random	Generates random numbers.	Simulating dice rolls, shuffling data, generating random passwords.
datetime	Works with dates and times.	Formatting dates, calculating time differences.
json	Parses JSON data.	Reading and writing JSON files for data exchange.
csv	Handles CSV (Comma-Separated Values) files.	Reading and writing data in tabular format.
urllib	Handles URL requests.	Fetching web pages, sending HTTP requests.
tkinter	Provides GUI (Graphical User Interface) elements.	Building simple desktop applications.

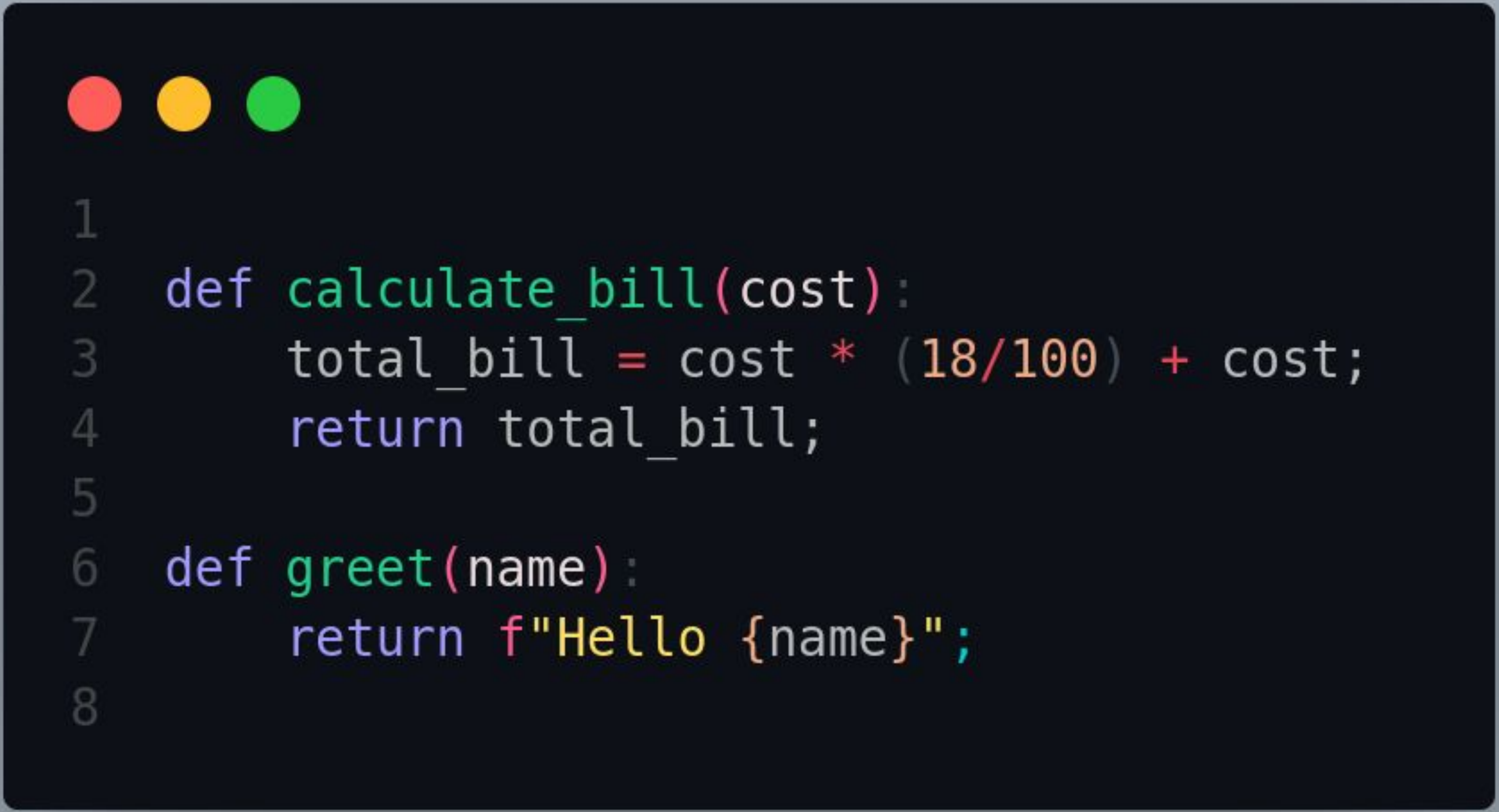


```
import math;

print(math.ceil(6.5));
```

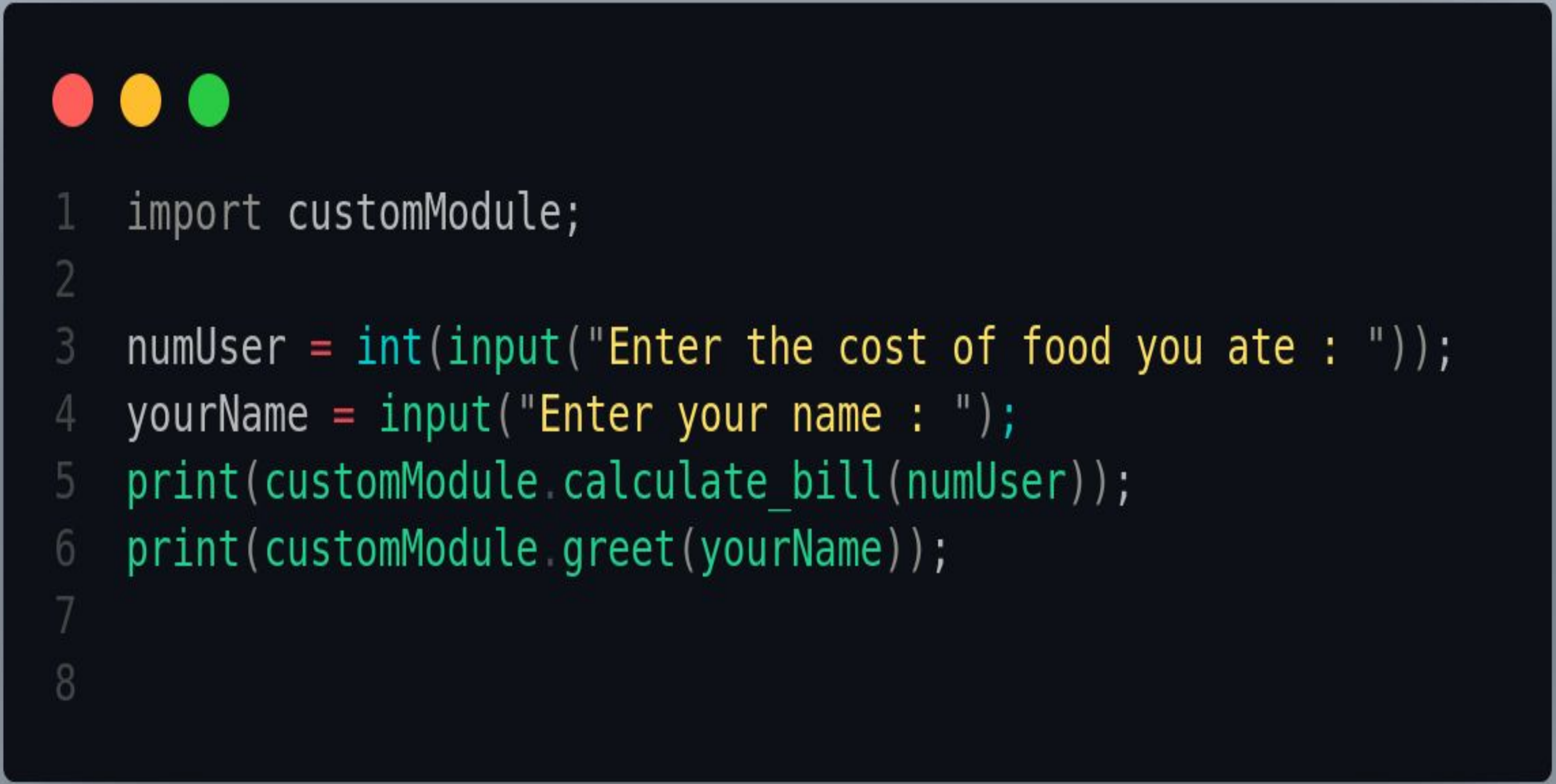
custom modules.

how will you create your own module.

- write functions or classes in a .py file.
- save the file with a name.
- import and use it in another program.
- you need to handle the directory very well though.



```python

def calculate_bill(cost):
    total_bill = cost * (18/100) + cost;
    return total_bill;

def greet(name):
    return f"Hello {name}";

```



```python
import customModule;

numUser = int(input("Enter the cost of food you ate : "));
yourName = input("Enter your name : ");
print(customModule.calculate_bill(numUser));
print(customModule.greet(yourName));


```

Aspect	Built-in Modules	External Libraries
pre-installed	Yes, included with Python.	No, must be installed manually (pip install).
examples	math, os, random, json.	pandas, numpy, tensorflow, pytorch, sklearn.
use case	General-purpose programming needs.	Specialized tasks like data analysis, machine learning, etc.
complexity	Lightweight and easy to use.	More complex and feature-rich, often for advanced users.
purpose	Basic utilities (e.g., math, file handling).	Advanced functionality (e.g., AI, data analysis, visualization).

any questions

resources

python.org (official documentation)
learnpython.org (great site to learn the basics)

replit (online practise)

“automate the boring stuff” GREAT BOOK & Course.
“think python” Book.

Thank you!