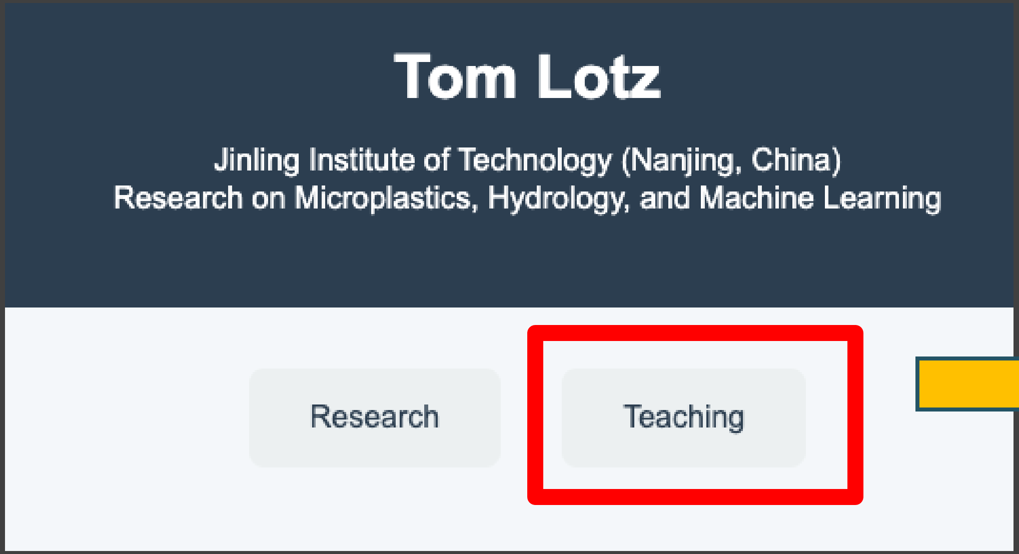
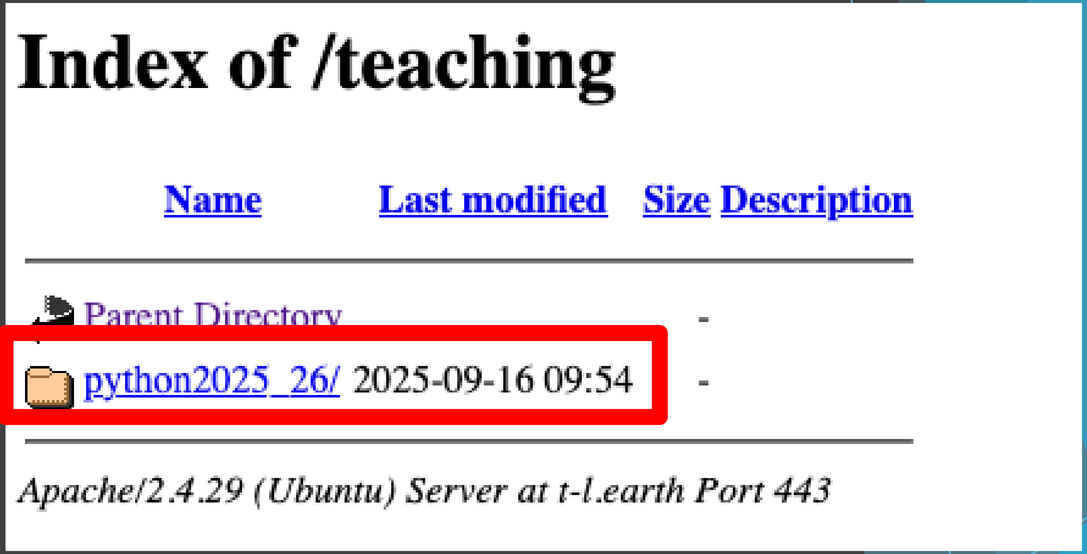
Download the cheat sheets and slides from here

t-l.earth



Tom Lotz

Jinling Institute of Technology (Nanjing, China)
Research on Microplastics, Hydrology, and Machine Learning



Index of /teaching

Name	Last modified	Size	Description
Parent Directory		-	
python2025_26/	2025-09-16 09:54	-	

Apache/2.4.29 (Ubuntu) Server at t-l.earth Port 443

Python语言程序设计

Python Programming

2025/26

Session 07

Tom Lotz (tom.lotz@outlook.com)

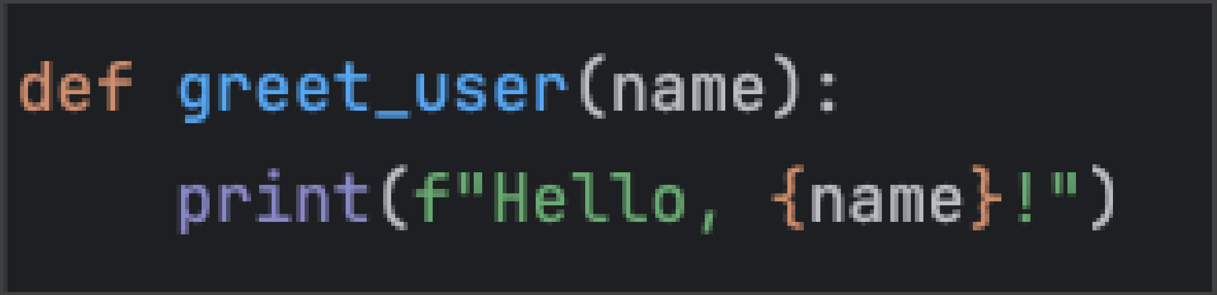
Content

	Brain Activation + Review
01	Modeling the Real World
02	Introduction to Classes and Objects
03	Working with Attributes and Methods
04	Exercises

Brain Activation

Greeting Function

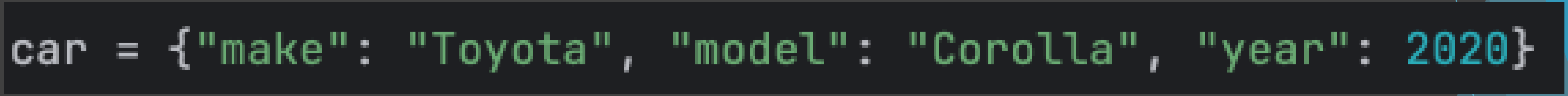
- Write a simple function that prints a greeting.



```python
def greet_user(name):
    print(f"Hello, {name}!")
```

Dictionary Review

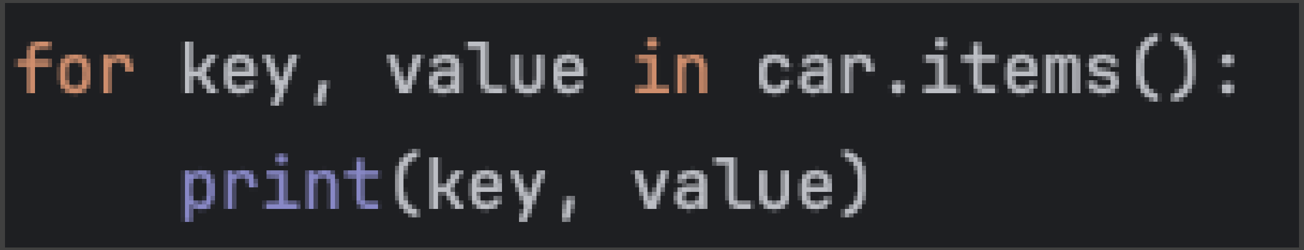
- Make a small dictionary representing a car.
- Print the make and model in one line.



```
car = {"make": "Toyota", "model": "Corolla", "year": 2020}
```

Looping Through a Dictionary

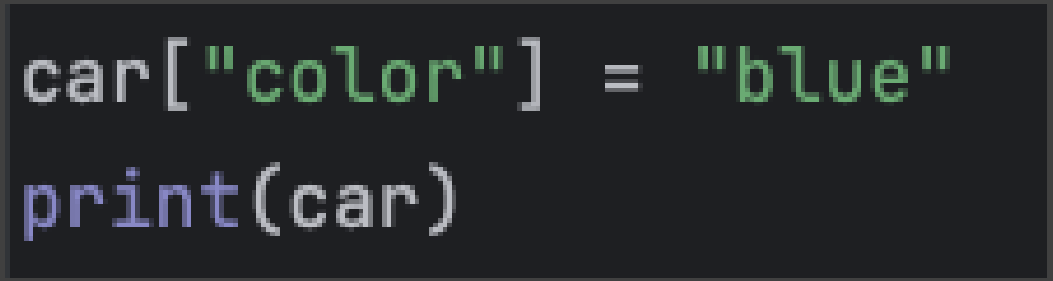
- Loop through all key-value pairs in your car dictionary.



```
for key, value in car.items():
    print(key, value)
```

Modify a Dictionary

- Add a new key and value.



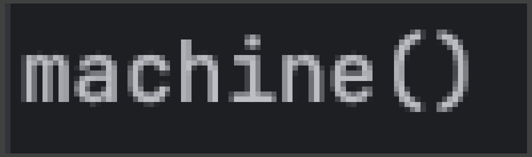
```
car["color"] = "blue"
print(car)
```

Review

What Is a Function?

REVIEW!

- A function is a named block of code that performs one task.
- We use it to:
  - Reuse code instead of writing the same thing many times.
  - Keep programs organized and readable.
- Example idea: a machine that does one job when you press its button.

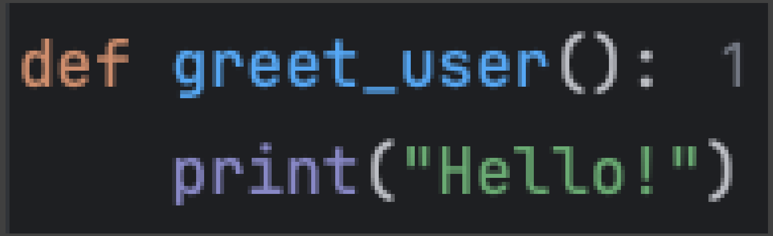


```
machine()
```

Defining a Function

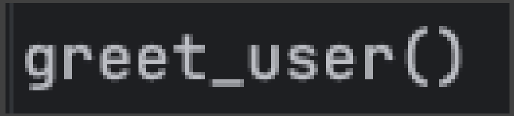
REVIEW!

- Basic syntax:



```python
def greet_user():
    print("Hello!")
```

- def introduces a function.
- The name (greet_user) describes its job.
- The colon starts an indented block (function body).
- To run it: you must call it. Before that, nothing happens.

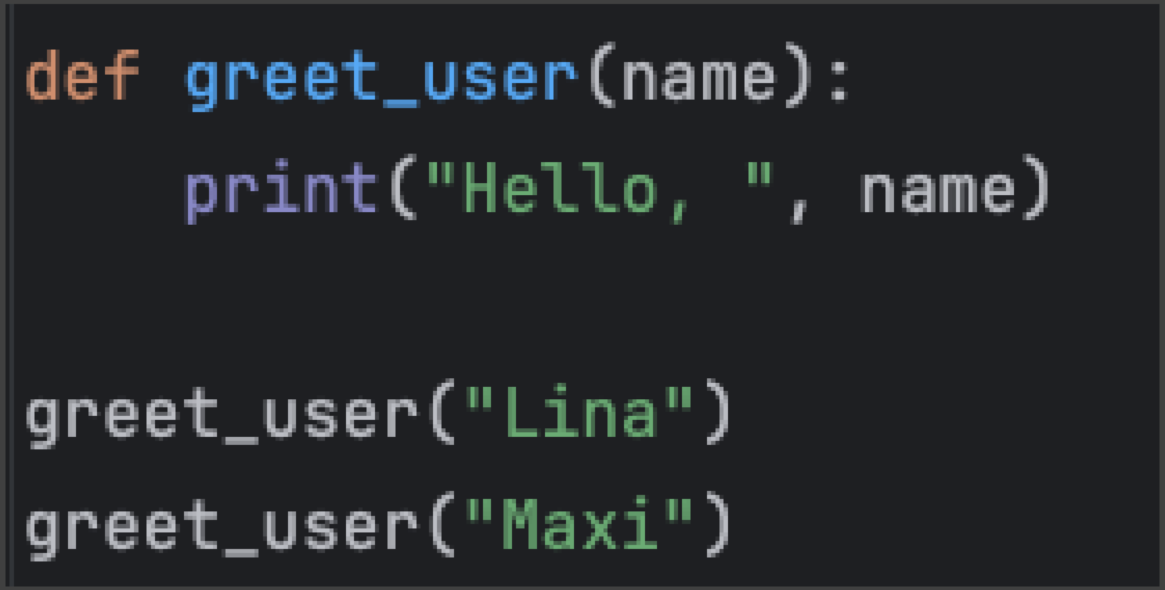


```python
greet_user()
```

Adding a Parameter

REVIEW!

- Functions can accept input values.



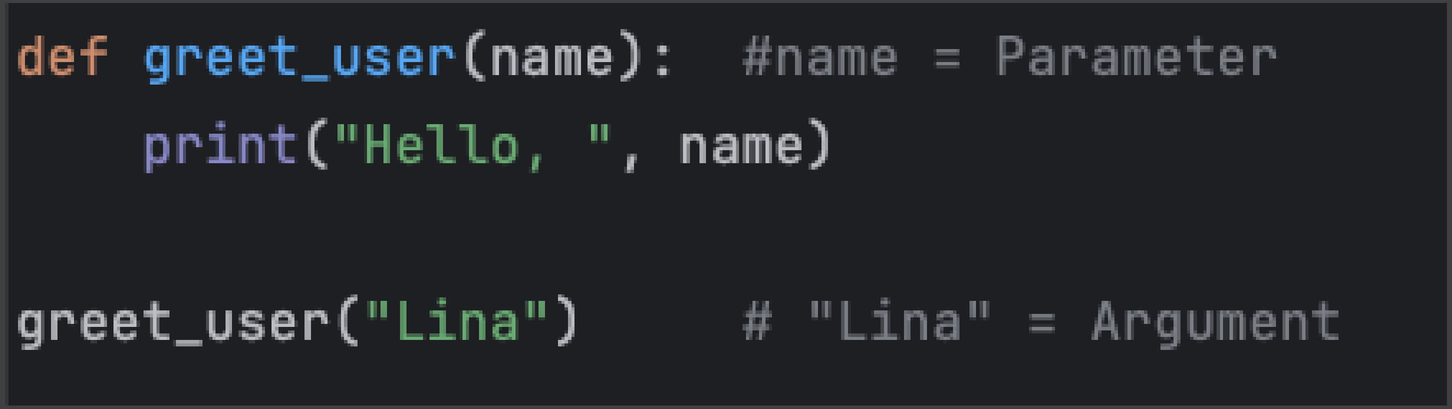
```
def greet_user(name):
    print("Hello, ", name)

greet_user("Lina")
greet_user("Maxi")
```

Arguments vs Parameters

REVIEW!

- Parameter: variable name inside the function definition.
- Argument: actual value passed when calling the function.

- Example:

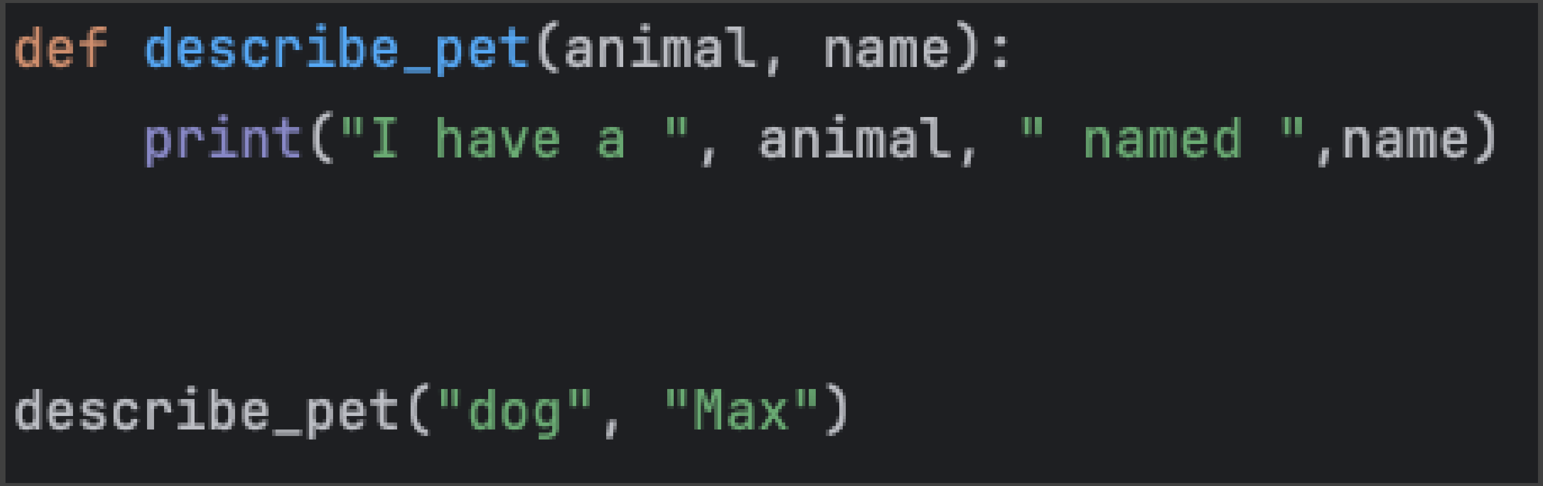


```
def greet_user(name):  #name = Parameter
    print("Hello, ", name)

greet_user("Lina")     # "Lina" = Argument
```

Functions with Multiple Parameters

REVIEW!

- You can pass more than one piece of information. Functions can be designed with any number of parameters.

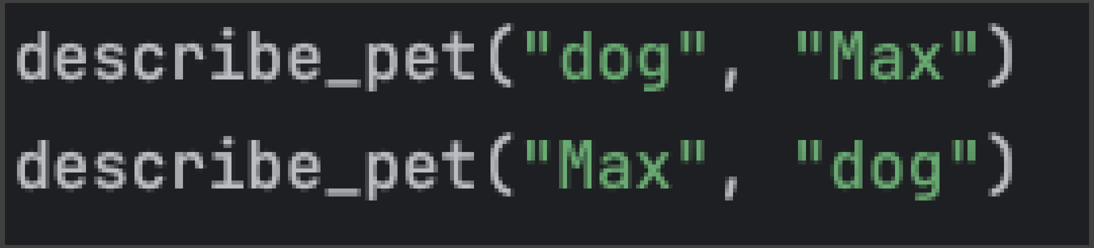


```
def describe_pet(animal, name):
    print("I have a ", animal, " named ",name)


describe_pet("dog", "Max")
```

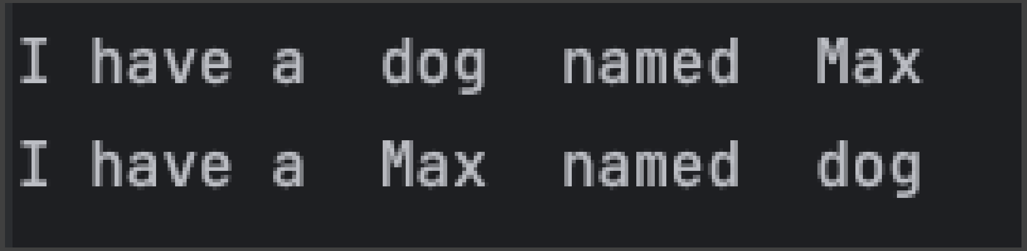
Order matters!

REVIEW!

- The arguments must be provided in the same order as the function definition expects them.



```
describe_pet("dog", "Max")
describe_pet("Max", "dog")
```

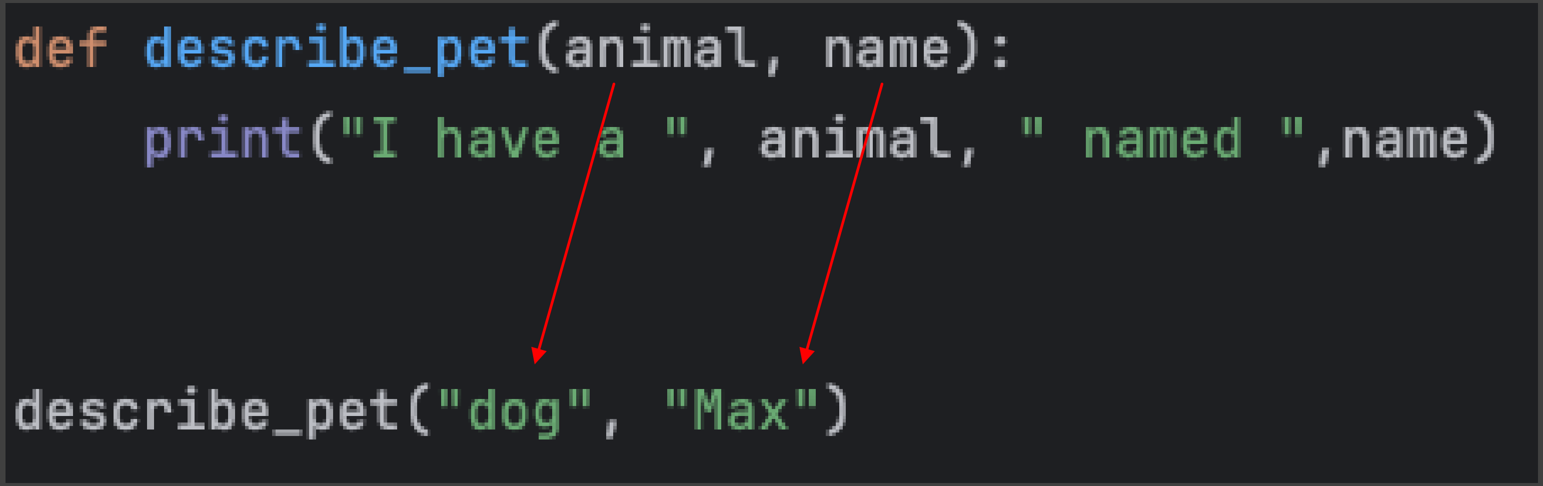


```
I have a  dog  named  Max
I have a  Max  named  dog
```

Positional Arguments

REVIEW!

- The arguments we have used so far are called positional arguments.



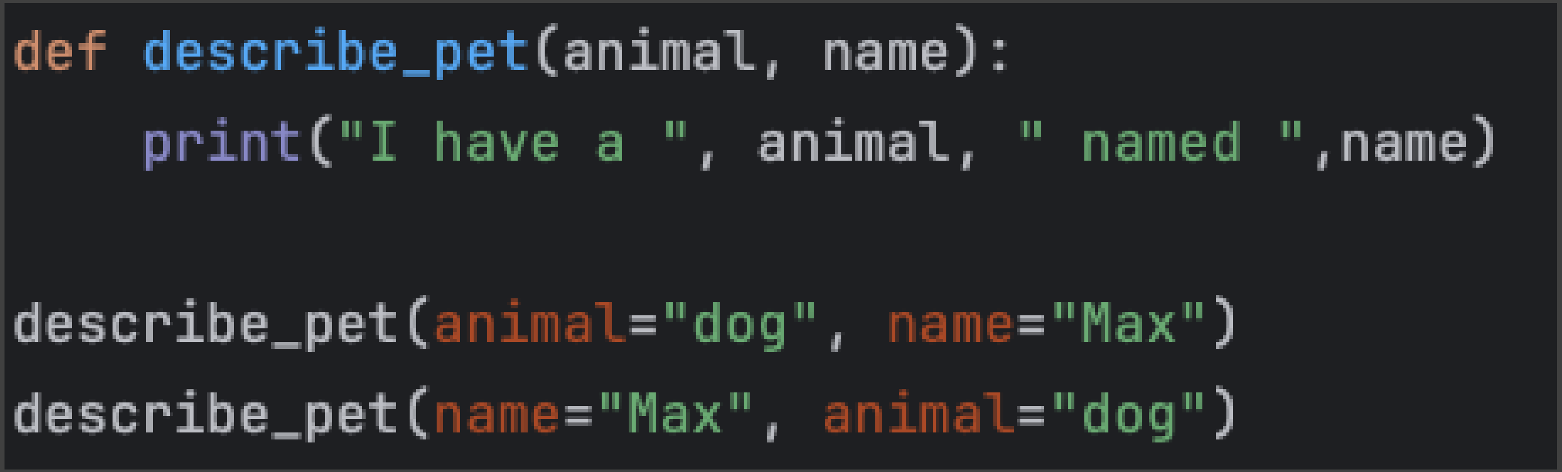
```
def describe_pet(animal, name):
    print("I have a ", animal, " named ",name)


describe_pet("dog", "Max")
```

- The order is absolute!

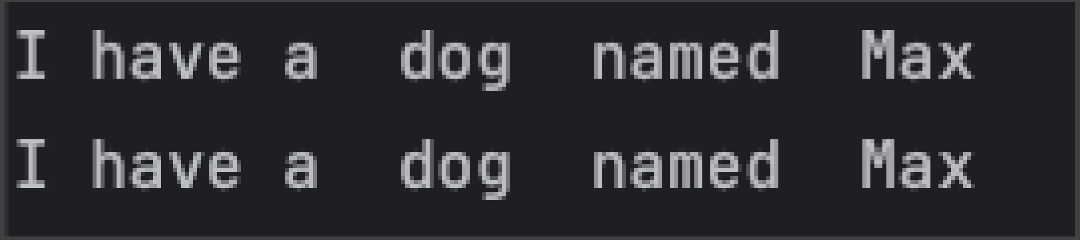
Keyword Arguments

REVIEW!

- The alternative is to use keywords (argument names) when calling the function.



```
def describe_pet(animal, name):
    print("I have a ", animal, " named ",name)

describe_pet(animal="dog", name="Max")
describe_pet(name="Max", animal="dog")
```

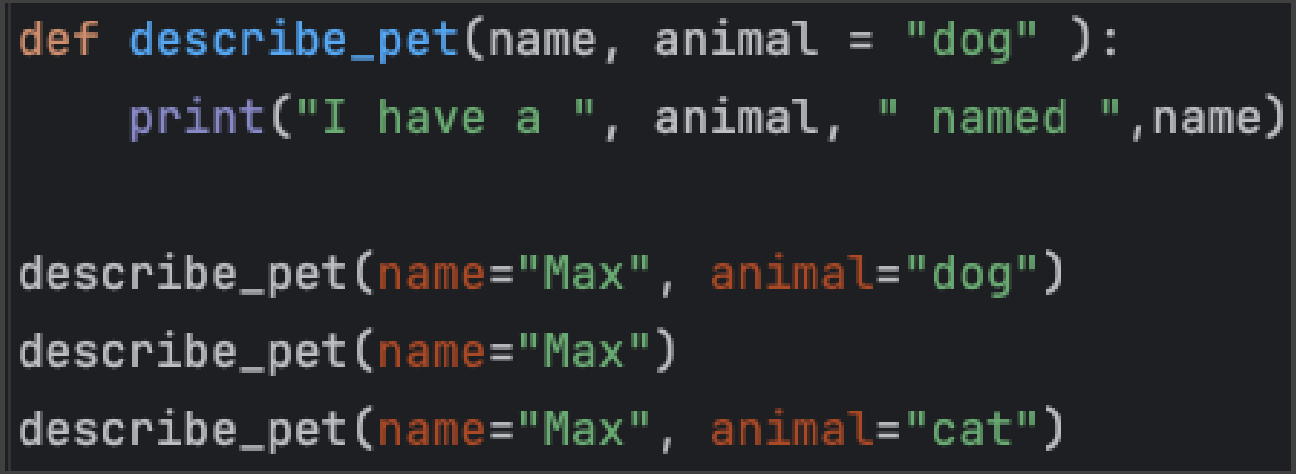


```
I have a  dog  named  Max
I have a  dog  named  Max
```

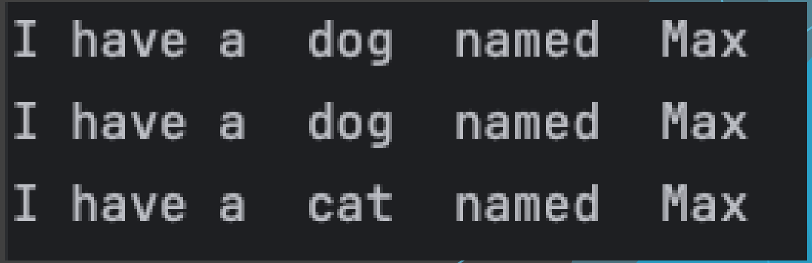
Default Values

REVIEW!

- You can provide default values to parameters.
  - In the definition, default parameters come after required parameters.



```python
def describe_pet(name, animal = "dog" ):
    print("I have a ", animal, " named ",name)

describe_pet(name="Max", animal="dog")
describe_pet(name="Max")
describe_pet(name="Max", animal="cat")
```

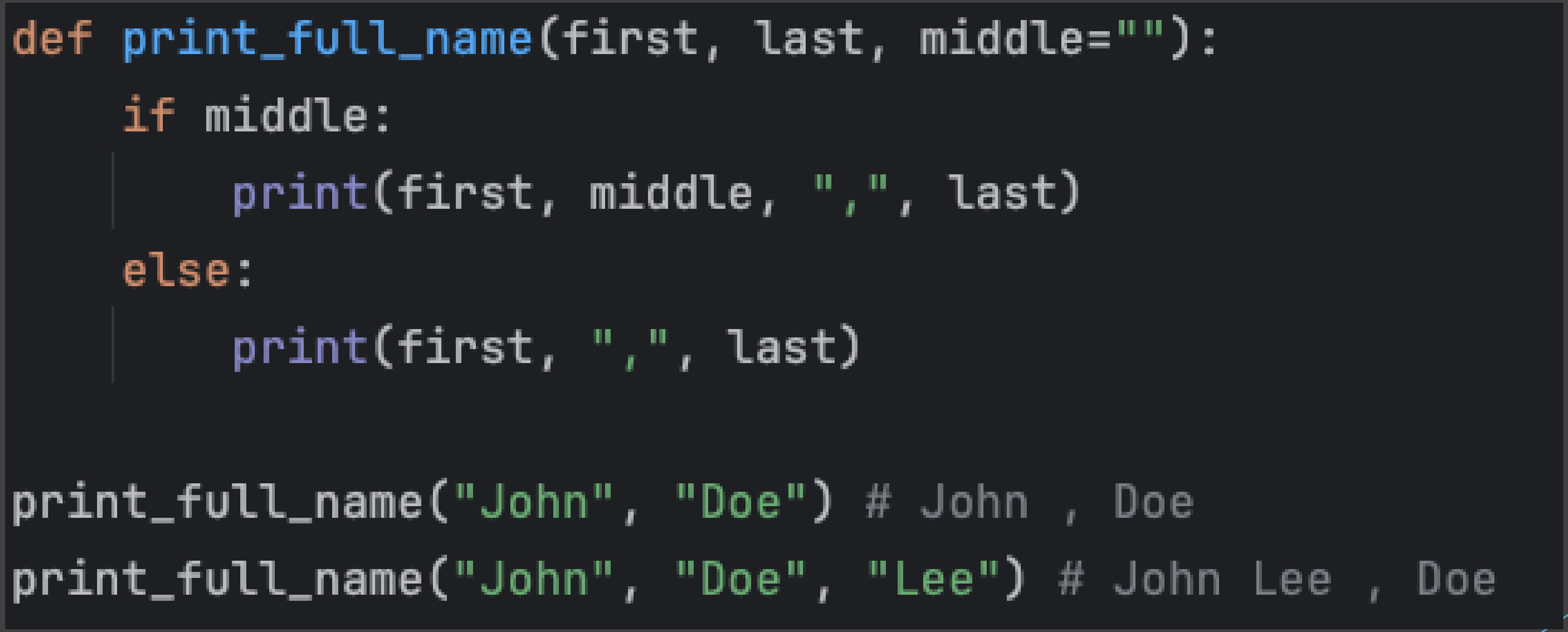


```
I have a  dog  named  Max
I have a  dog  named  Max
I have a  cat  named  Max
```

Optional Values

REVIEW!

- We can use an empty string “” or None to make an argument optional.

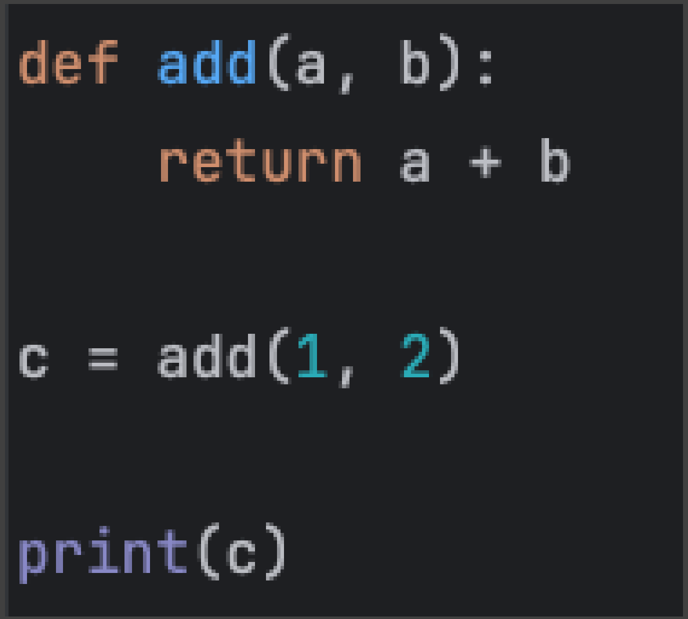


```
def print_full_name(first, last, middle=""):
    if middle:
        print(first, middle, ",", last)
    else:
        print(first, ",", last)

print_full_name("John", "Doe") # John , Doe
print_full_name("John", "Doe", "Lee") # John Lee , Doe
```

Capturing Return Values

REVIEW!

- We can capture the return value of a function with an assignment.

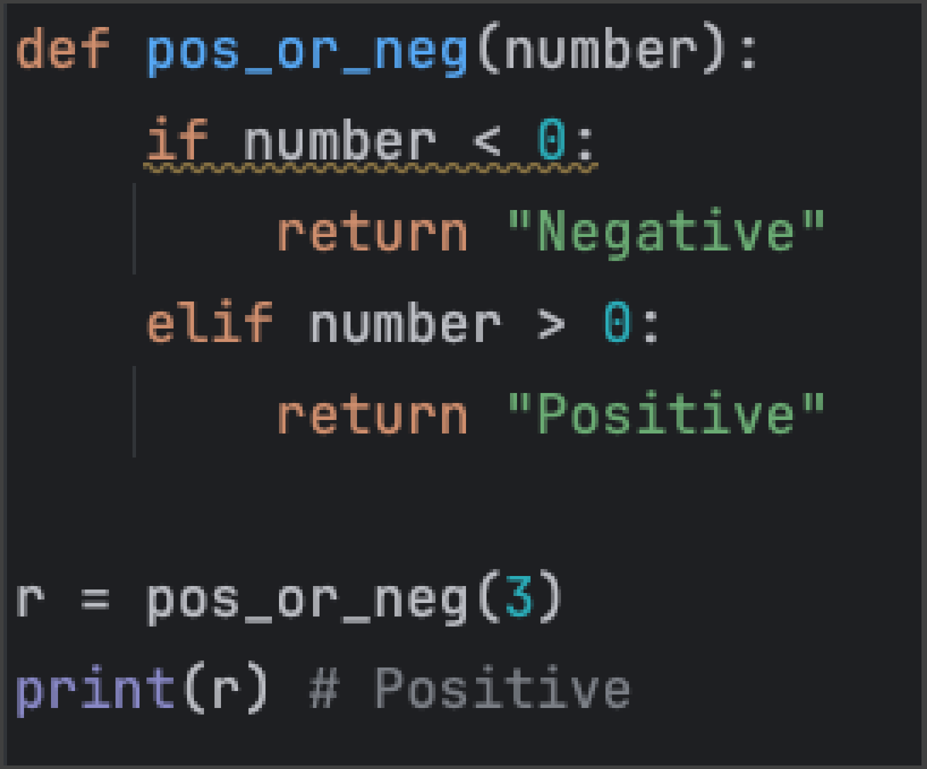


```python
def add(a, b):
    return a + b


c = add(1, 2)


print(c)
```

Returning from Conditional Logic

REVIEW!

- Functions can decide what value to return based on conditions.



```python
def pos_or_neg(number):
    if number < 0:
        return "Negative"
    elif number > 0:
        return "Positive"

r = pos_or_neg(3)
print(r) # Positive
```

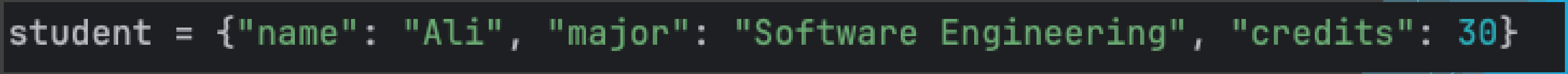
Modeling the Real World

From Code to Models

- Programming is not just writing instructions; it’s building models of reality. We use variables and lists for data, but when data and behavior belong together, we need something more structured.
- Example:
  - A student has a name, major, and GPA.
  - A bank account has an owner, balance, and actions (deposit, withdraw).
- This combination of data + actions is what a class represents.

Why Not Just Use Dictionaries?

- Dictionaries can store information, but they cannot do anything by themselves.
- If we want to update credits or calculate GPA, we need separate functions.



```
student = {"name": "Ali", "major": "Software Engineering", "credits": 30}
```

The Core Idea of Object-Oriented Programming (OOP)

- In OOP, we describe things — not just actions. Each object in code mirrors an object in the real world.
- Concepts:
  - Class: the blueprint (defines what all objects of that kind know and do)
  - Object (instance): one specific example of that class
  - Attributes: data values inside each object
  - Methods: actions that object can perform

Thinking Like a Designer

- When designing a class:
  - Identify the entity (noun) you want to represent.
  - Decide which details are important (attributes).
  - Define the actions (methods) that belong to it.
- Example: Modeling a book. What do we need?

Abstraction: Simplifying the Real World

- A class is not the real object; it’s a simplified version of it. You choose only the details that matter for your program.

- Example: If modeling a car, you might include speed, fuel, and drive(), but not color_of_seatbelt().

- Abstraction = ignore unnecessary details, focus on purpose.

Mini Task 0

- Pick something familiar (phone, pet, course, or device). Write down:
  - 2-3 attributes (what it has)
  - 2-3 actions (what it does)

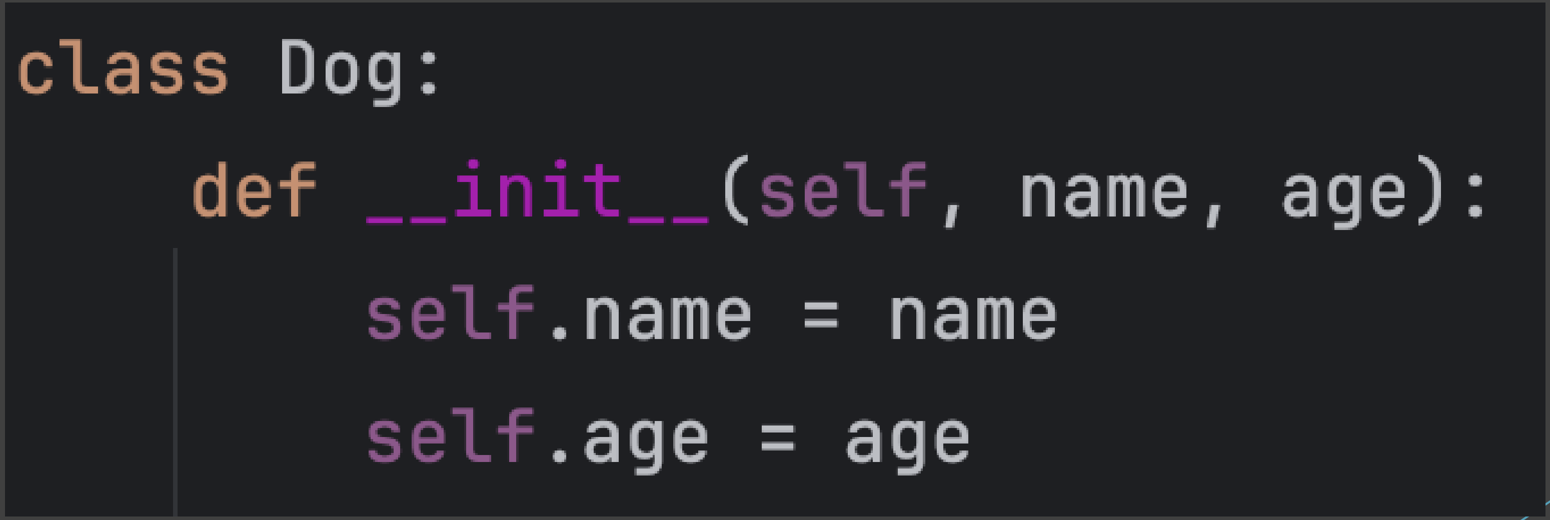
Introduction to Classes and Objects

What Is a Class?

- A class is a blueprint for creating objects.
  - It defines what data (attributes) and actions (methods) the objects have.
  - Objects are called instances of that class.
- Example analogy:
  - Class = recipe 🍰 → Instance = actual cake 🎂

Defining a Class

- class keyword starts a class definition.
- __init__() initializes new objects.
- self refers to this particular instance.



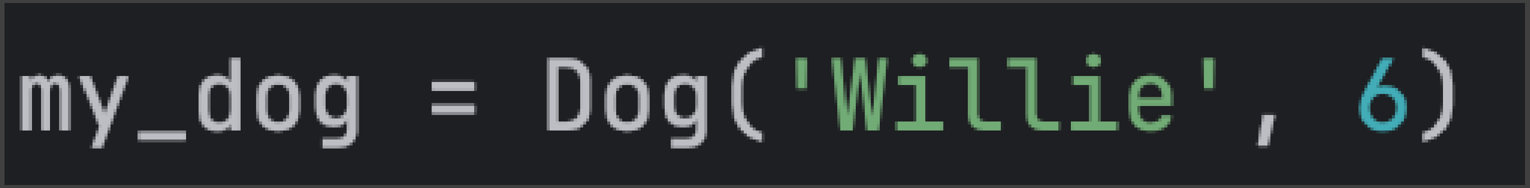
```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Mini Task 1

- Create your own class Student with:
  - Attributes: name, major.

Creating an Object

- We can create instances of the Dog class by calling it.

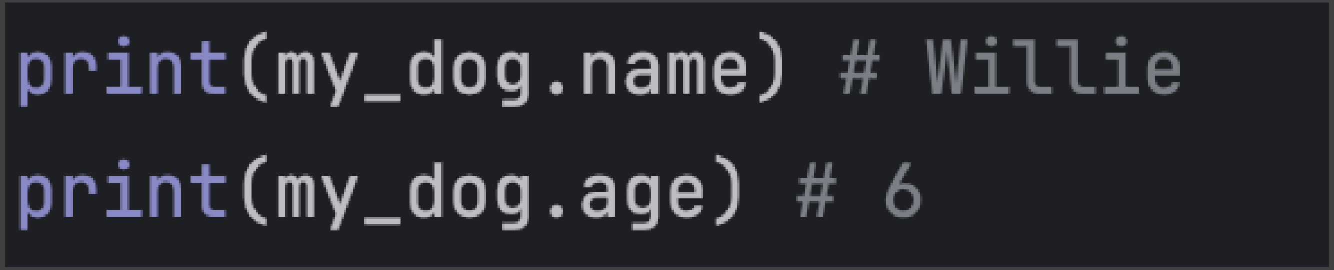


```python
my_dog = Dog('Willie', 6)
```

- Python calls __init__() automatically.
- my_dog is now an object with data inside.

Accessing attributes

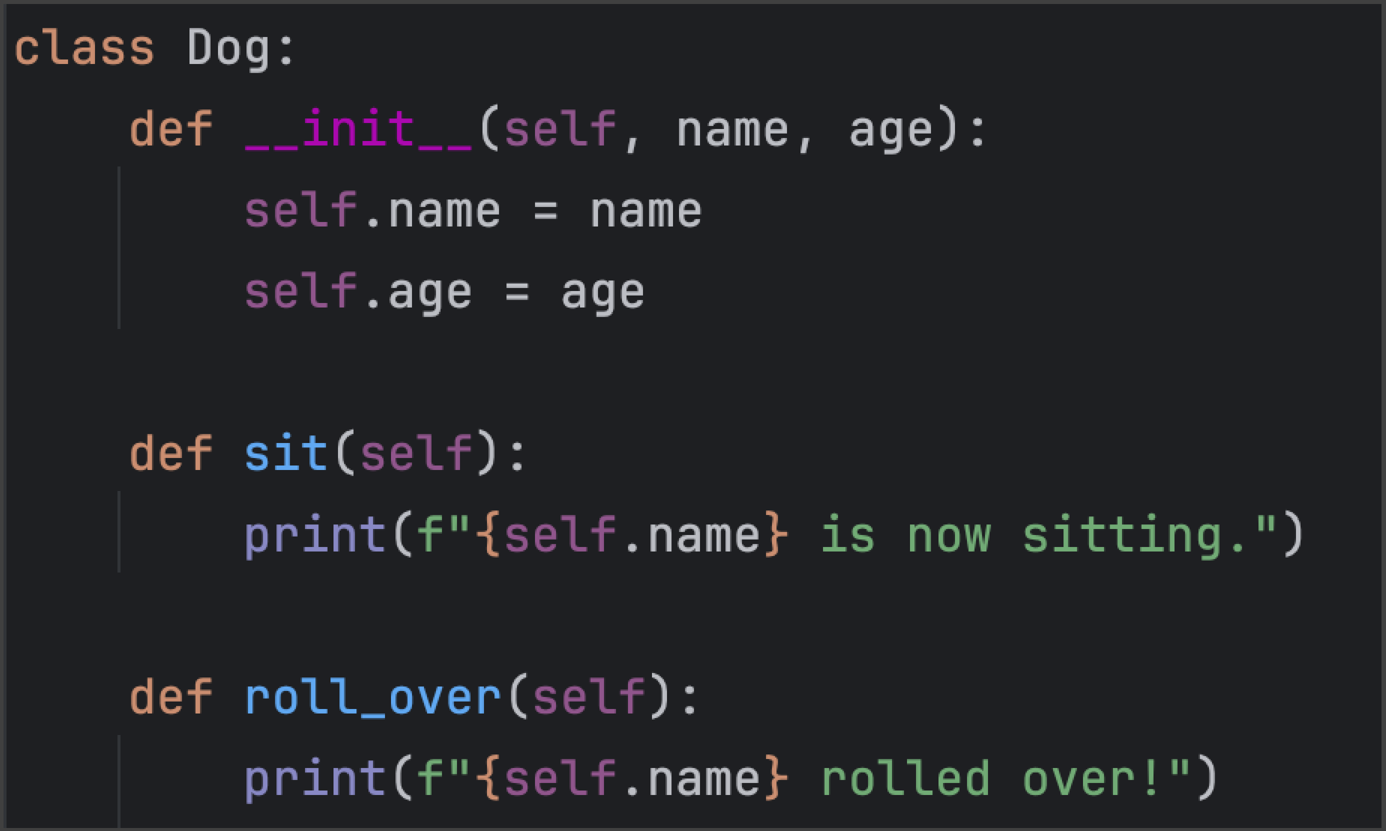
- We can access the attributes of our instance:



```
print(my_dog.name) # Willie
print(my_dog.age) # 6
```

Adding Behavior (Methods)

- We can add methods to a class.

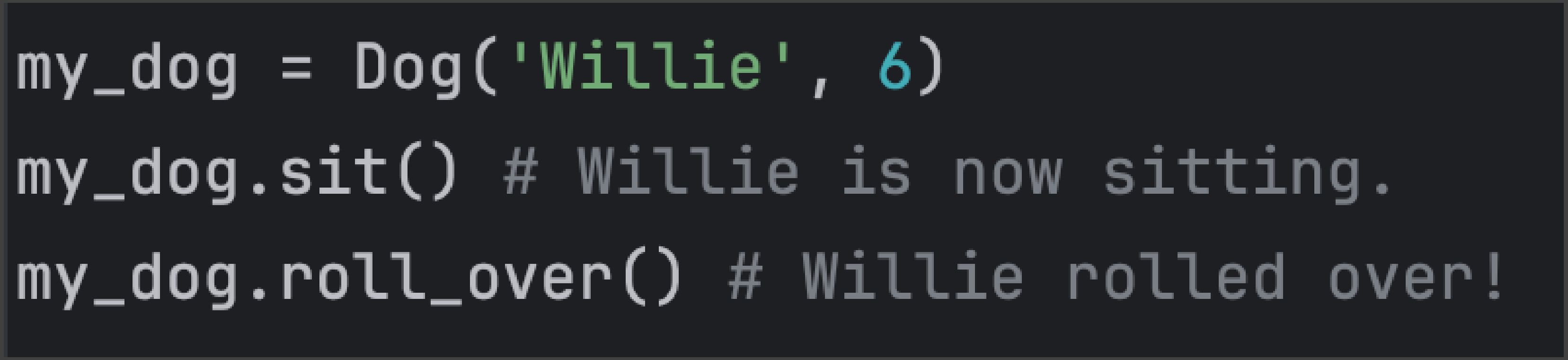


```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def sit(self):
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        print(f"{self.name} rolled over!")
```

- Each method must include self.

Using Methods

- The methods of the instance are called in this way:



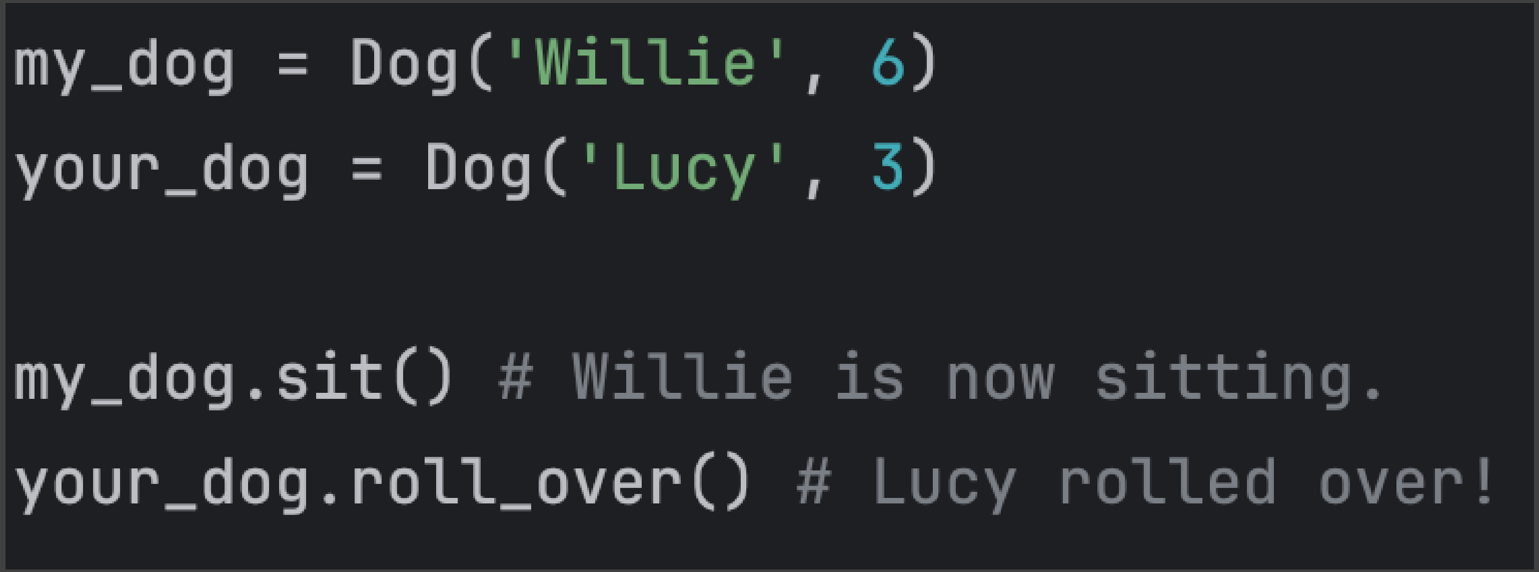
```
my_dog = Dog('Willie', 6)
my_dog.sit() # Willie is now sitting.
my_dog.roll_over() # Willie rolled over!
```

Mini Task 2

- Modify your class Student to contain:
  - Attributes: name, major.
  - Method: introduce() → prints a short intro.
    - Example:
      - "Hi, I’m Lina, and I study Software Engineering."

Multiple Instances

- Once a class has been defined, we can create as many instances as we want.



```
my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)

my_dog.sit() # Willie is now sitting.
your_dog.roll_over() # Lucy rolled over!
```

Mini Task 3

- Create several instances of your student class and call their introduction method.

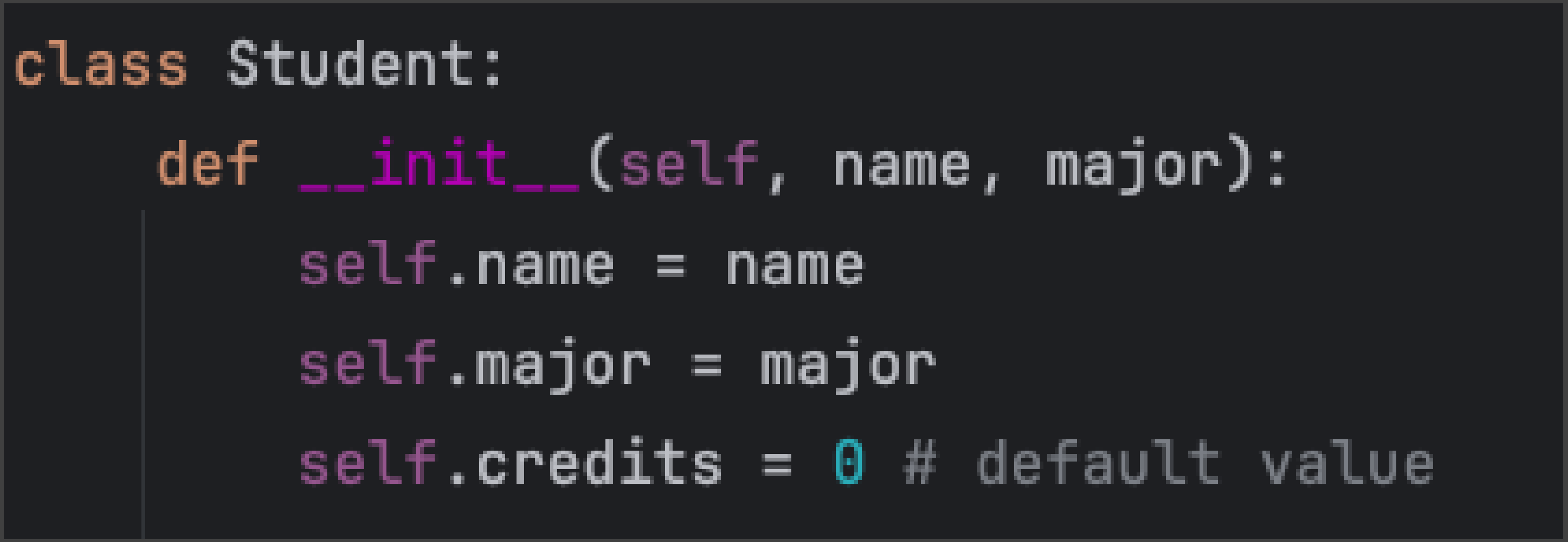
Wrap-up

- Define a class using class.
- Initialize data in __init__().
- Access data and methods via dot notation.
-
- Each instance is unique but shares behavior.

Working with Attributes and Methods

Adding Default Attributes

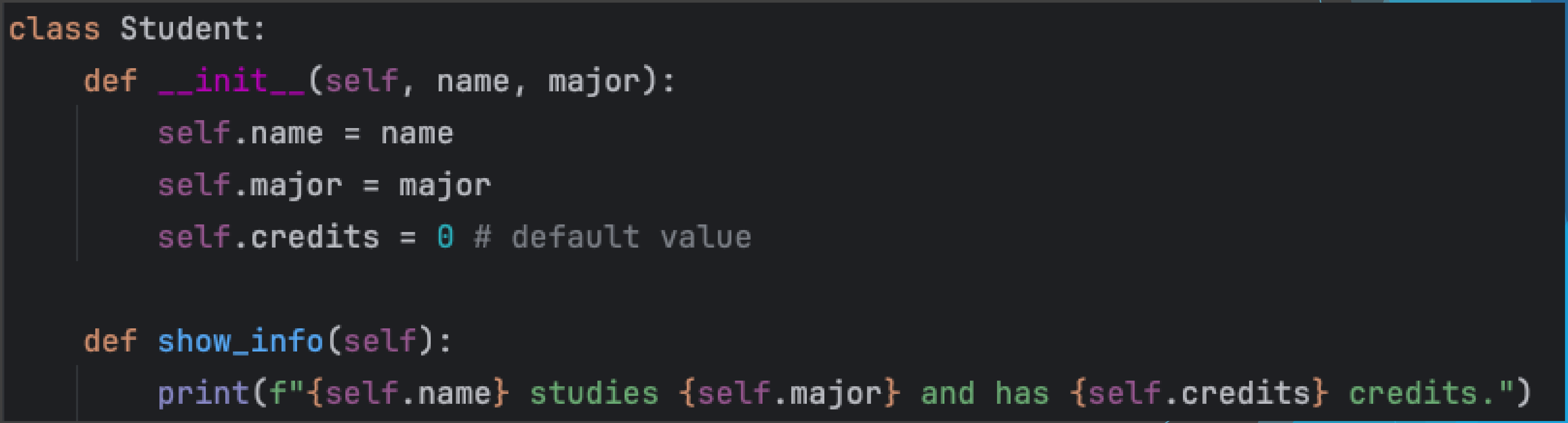
- Every student starts with 0 credits



```
class Student:
    def __init__(self, name, major):
        self.name = name
        self.major = major
        self.credits = 0 # default value
```

Mini Task 4

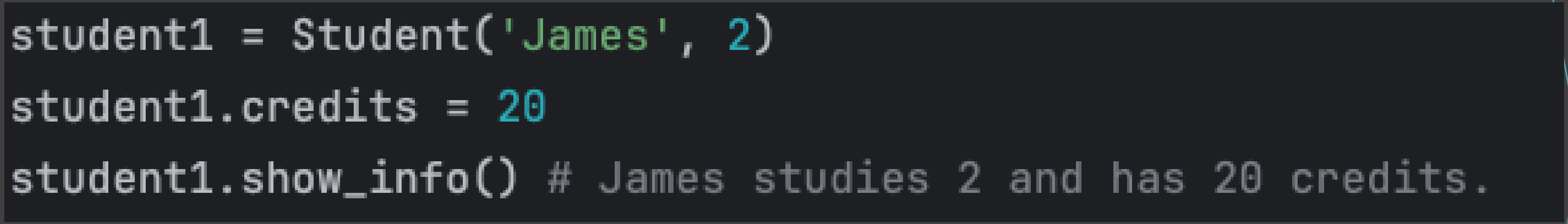
- Add a default credits value to your class and add a method to print all information.



```python
class Student:
    def __init__(self, name, major):
        self.name = name
        self.major = major
        self.credits = 0 # default value

    def show_info(self):
        print(f"{self.name} studies {self.major} and has {self.credits} credits.")
```

Modifying Attributes Directly

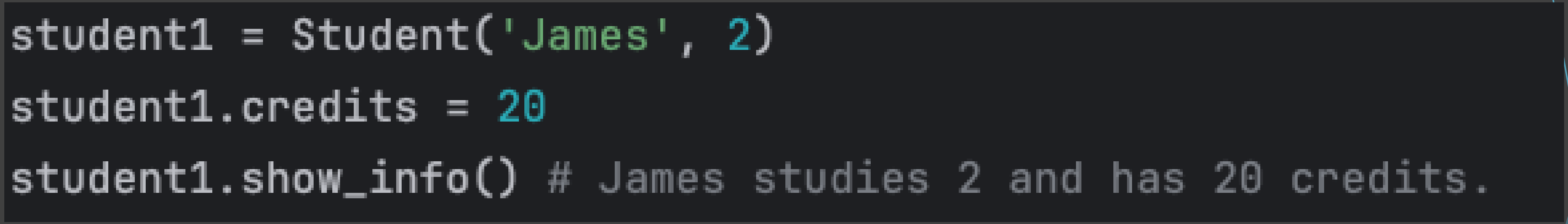
- We can change values manually:



```
student1 = Student('James', 2)
student1.credits = 20
student1.show_info() # James studies 2 and has 20 credits.
```

Modifying Attributes Directly

- We can change values manually:

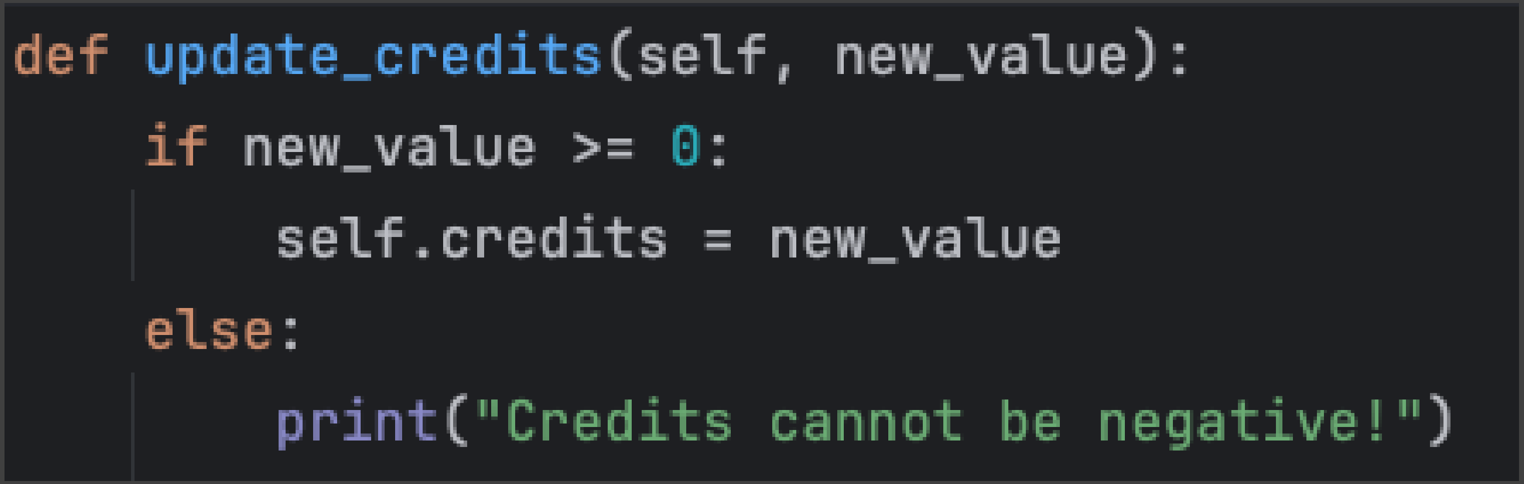


```
student1 = Student('James', 2)
student1.credits = 20
student1.show_info() # James studies 2 and has 20 credits.
```

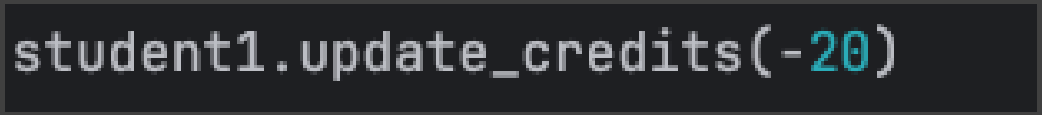
- This works, but there is no control over the data added for the attribute (negative values, strings, ...).

Updating Attributes Safely

- Better to add a method that updates credits but protects against invalid data.



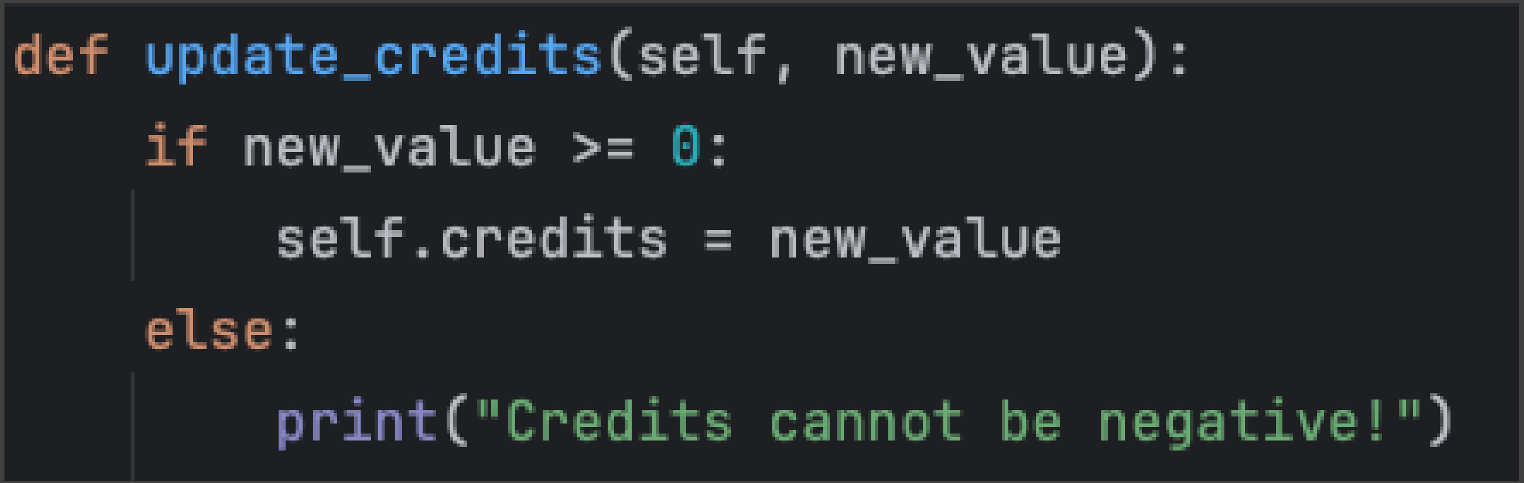
```
def update_credits(self, new_value):
    if new_value >= 0:
        self.credits = new_value
    else:
        print("Credits cannot be negative!")
```



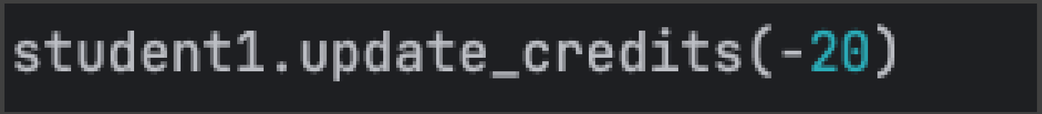
```
student1.update_credits(-20)
```

Updating Attributes Safely

- Better to add a method that updates credits but protects against invalid data.



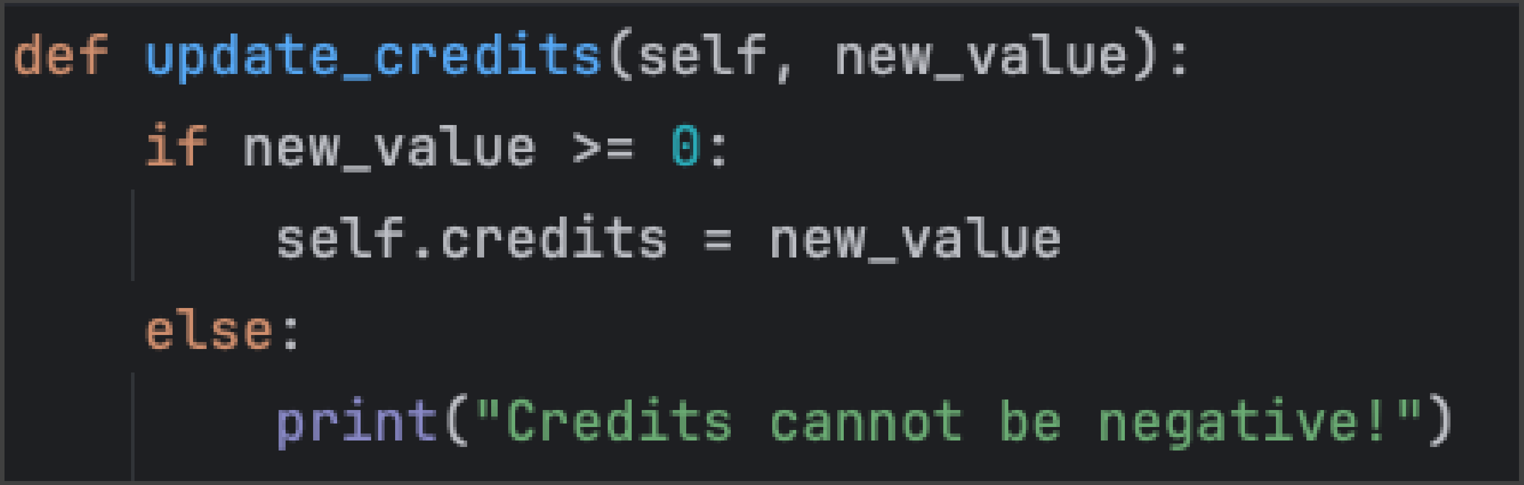
```
def update_credits(self, new_value):
    if new_value >= 0:
        self.credits = new_value
    else:
        print("Credits cannot be negative!")
```



```
student1.update_credits(-20)
```

Mini Task 5

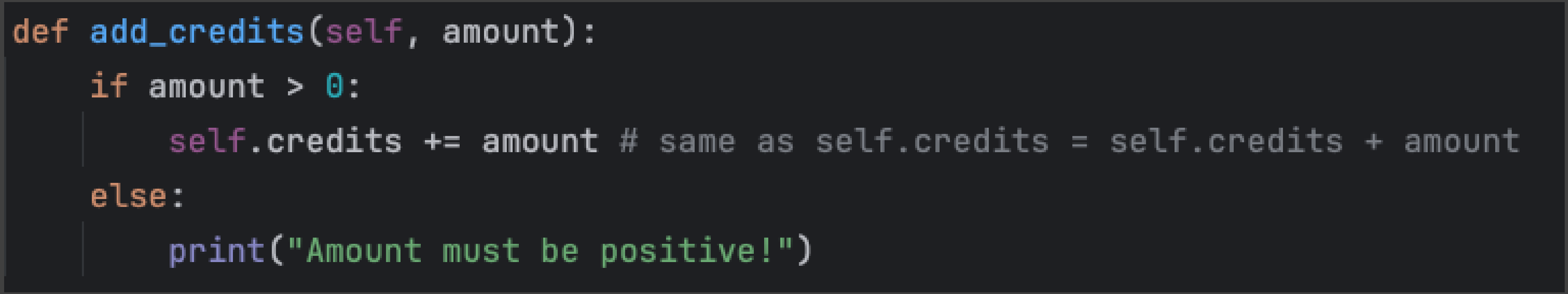
- Add update_credits() and test it with positive and negative numbers.



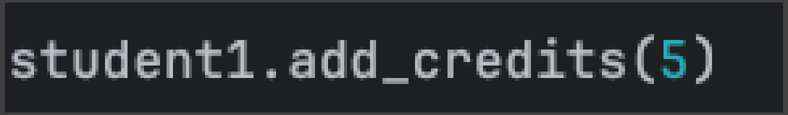
```python
def update_credits(self, new_value):
    if new_value >= 0:
        self.credits = new_value
    else:
        print("Credits cannot be negative!")
```

Incrementing Attribute Values

- Instead of replacing the value, we can increase it gradually.



```python
def add_credits(self, amount):
    if amount > 0:
        self.credits += amount # same as self.credits = self.credits + amount
    else:
        print("Amount must be positive!")
```

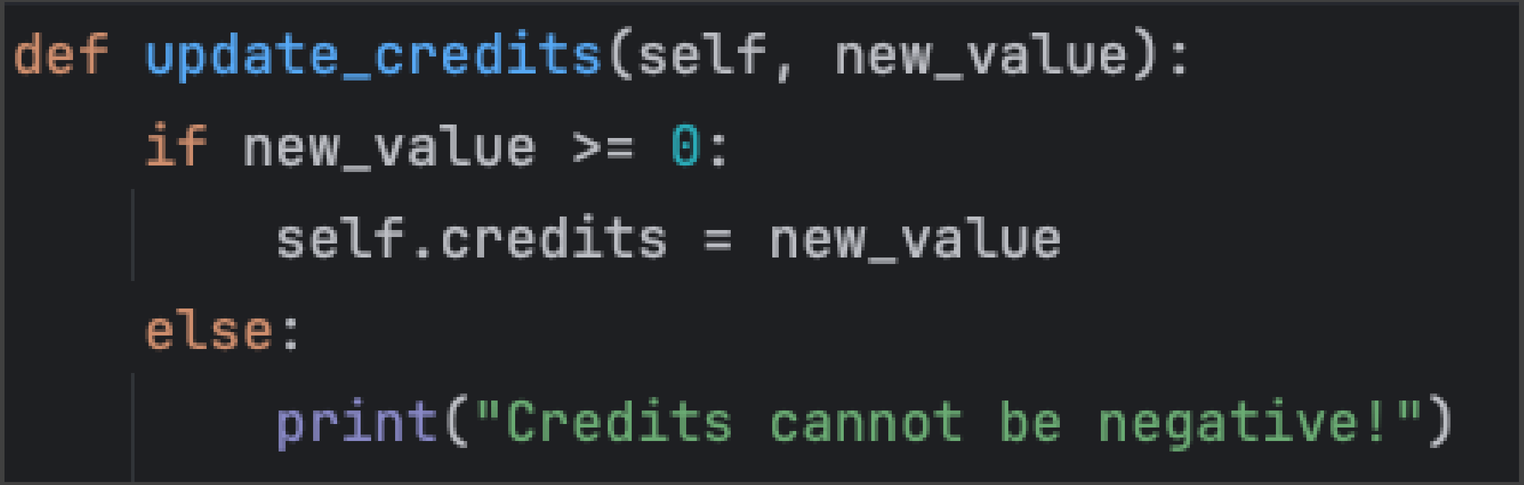


```python
student1.add_credits(5)
```

Mini Task 6

- Add a method called change_credits(), that allows positive AND negative values. Meanwhile, if self.credits is ever below 0 after an update, set it to 0.

- Reminder:



```
def update_credits(self, new_value):
    if new_value >= 0:
        self.credits = new_value
    else:
        print("Credits cannot be negative!")
```

Wrap-up

- We can change attributes directly, but that can be risky.
- Better to add a method to change attributes with a security check.

Exercises

Bank Account

- Build the class BankAccount that grows with each task.
- You can find the task list in the CheatSheet_07.py file on the server.
- Try to finish as many tasks as possible!