Python level 3

Advanced Course
Class 3

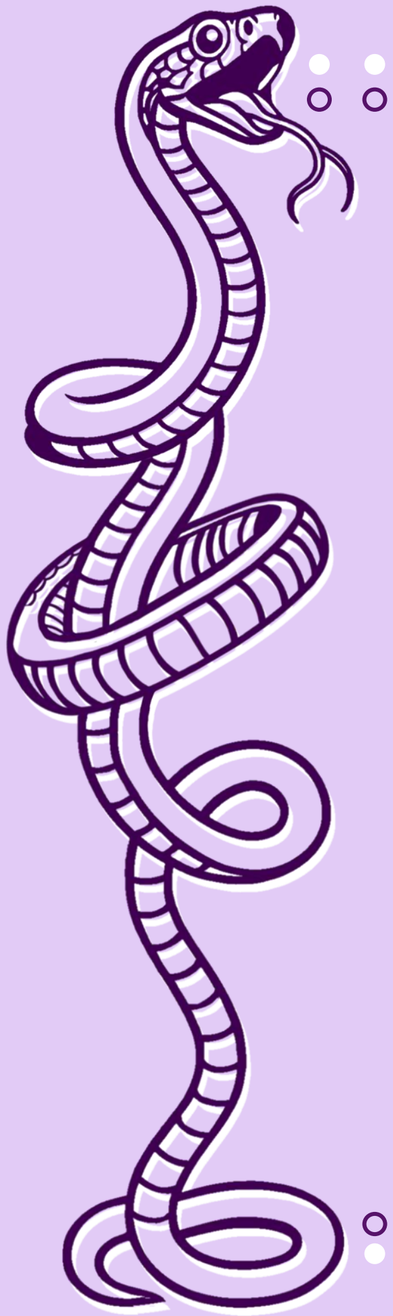
Alexandra Oliveira

October 7

AGENDA

Part I – Object Oriented Programming

1. Python classes
2. Attributes
3. Methods
4. Objects



Object oriented programming

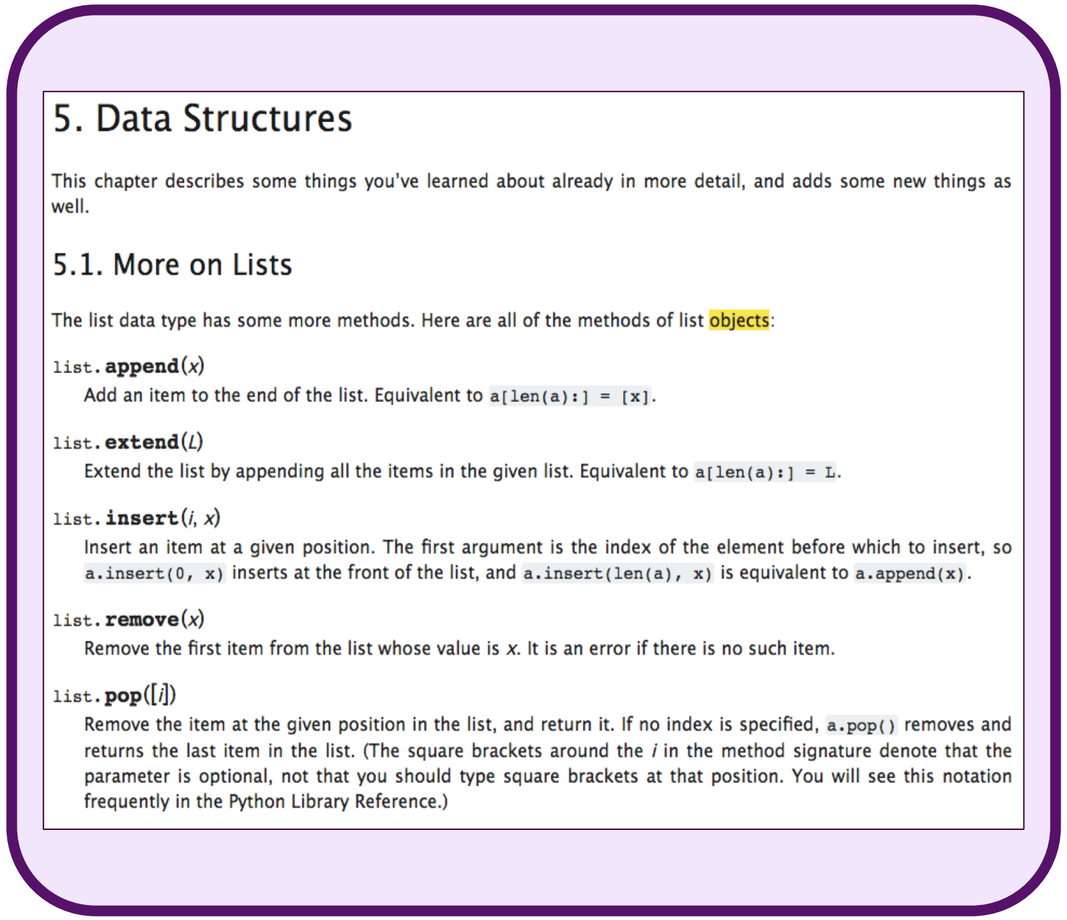
Object Oriented Programming

What is it?

- A fundamental programming paradigm **widely used today**
- It **models real-world entities** (e.g. cars, companies, students) and their **relationships**
- Built around the **concept of objects** that store data and define behaviours
- Helps **structure programs** into smaller, cooperative pieces instead of one large block
- Each **object acts as a self-contained unit, interacting** with others to form the whole program.

Everything in Python is an Object!

Do you remember what a list is? **It is an Python Object!**



5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**(*x*)
Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**(*L*)
Extend the list by appending all the items in the given list. Equivalent to `a[len(a):] = L`.

`list.`**`insert`**(*i*, *x*)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

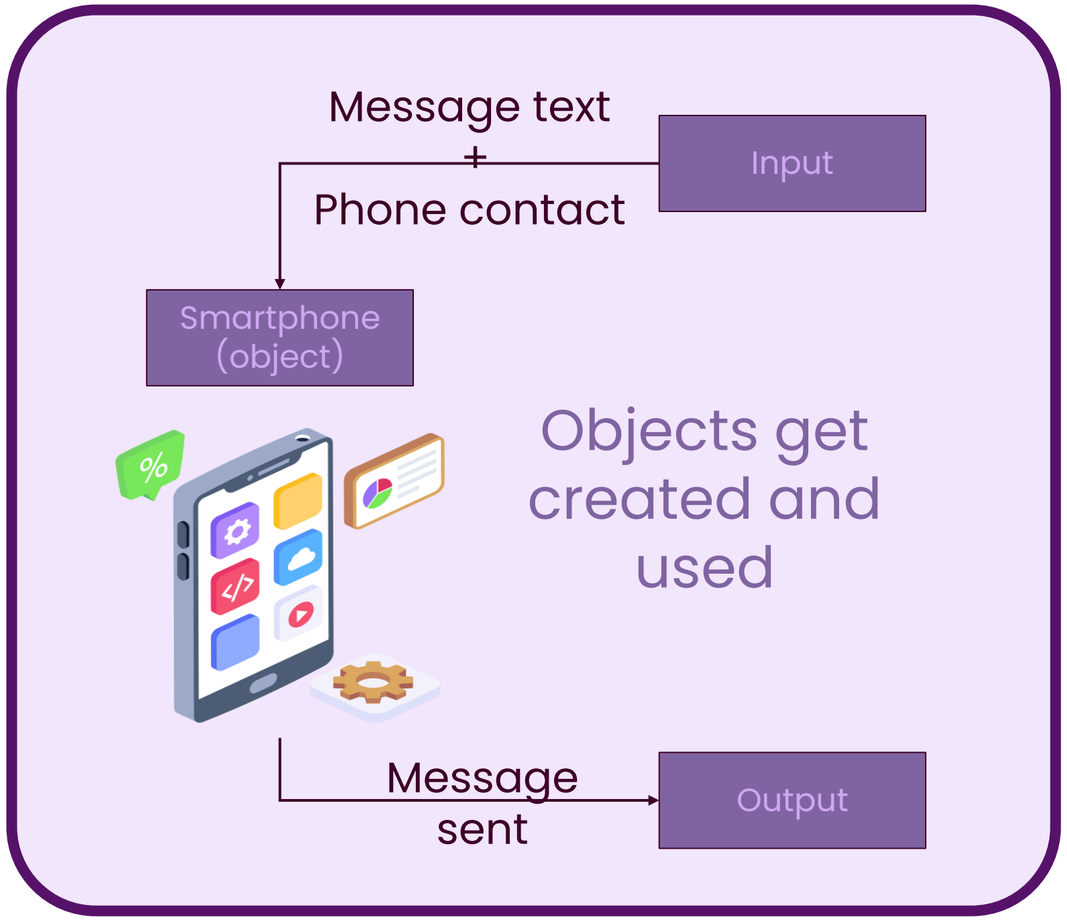
`list.`**`remove`**(*x*)
Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.`**`pop`**([*i*])
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

https://docs.python.org/3/tutorial/datastructures.html

Object Oriented Programming

Input: the information or actions given to the smartphone. When you tap on the screen, type a message, or speak into the microphone, you're providing input.

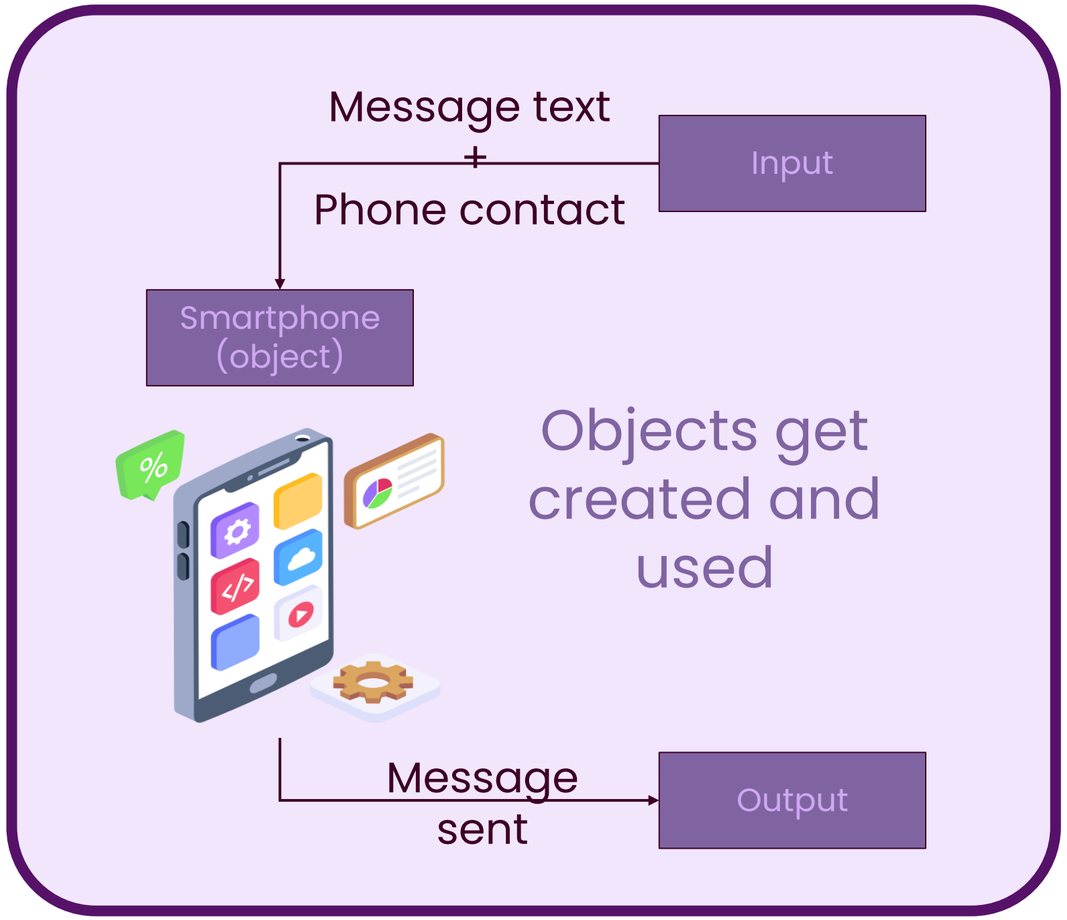


The smartphone example

Object Oriented Programming

Smartphone (Object): The smartphone itself is the object, and it has:

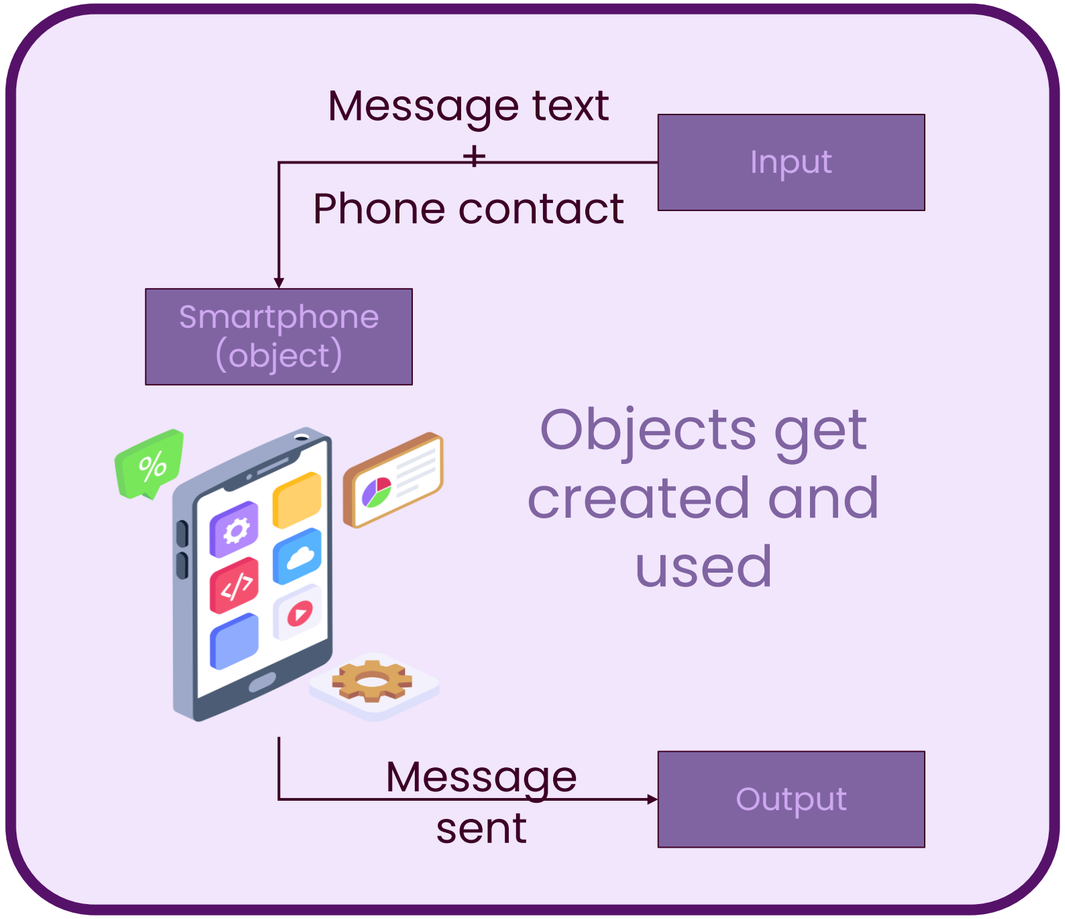
- **Attributes:** The visual icons on the screen, the battery level indicator, the settings you have in place, and so on
- **Methods:** The actions you can perform with it, such as opening an app (by tapping an icon), checking your emails, adjusting the volume, etc.



The smartphone example

Object Oriented Programming

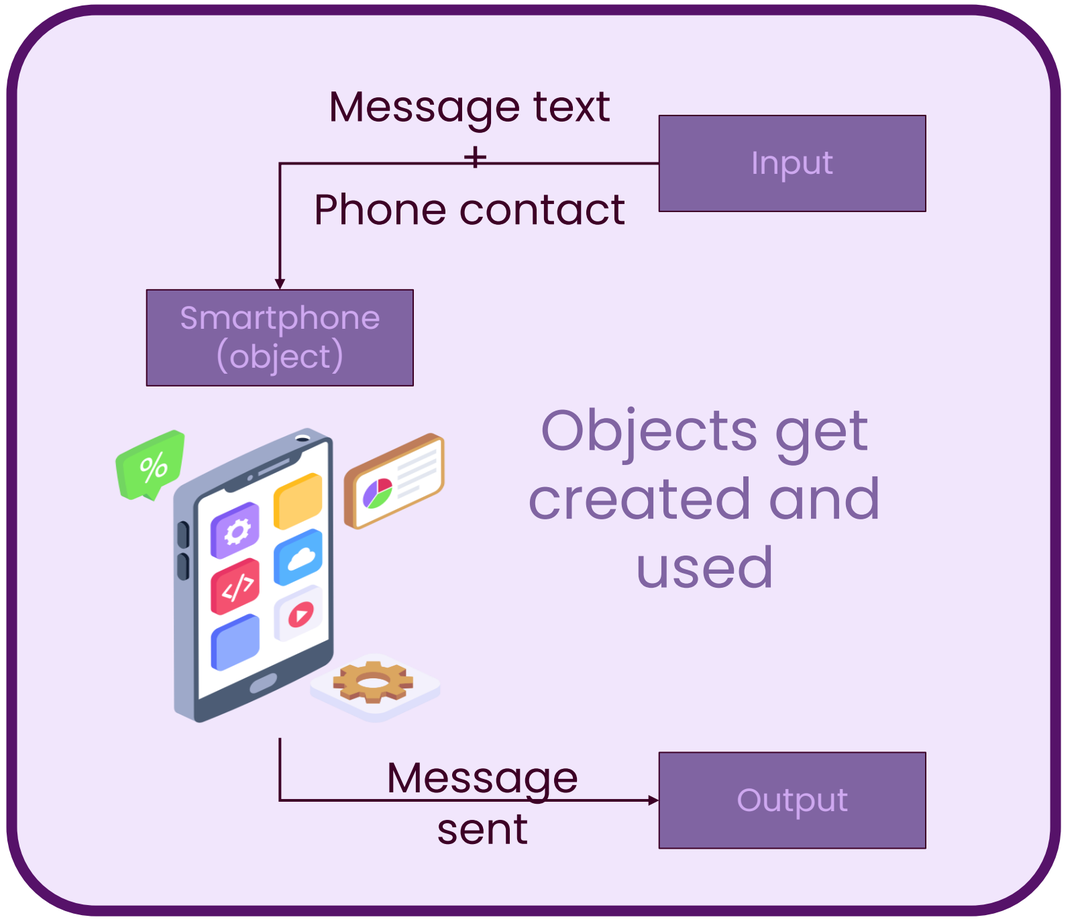
Objects get created and used: When you interact with your smartphone, you are using its capabilities. If you open an app, you create a 'session' of that app, which is like creating an object in programming. Each app has its own settings and data (attributes) and things you can do within the app (methods)



The smartphone example

Object Oriented Programming

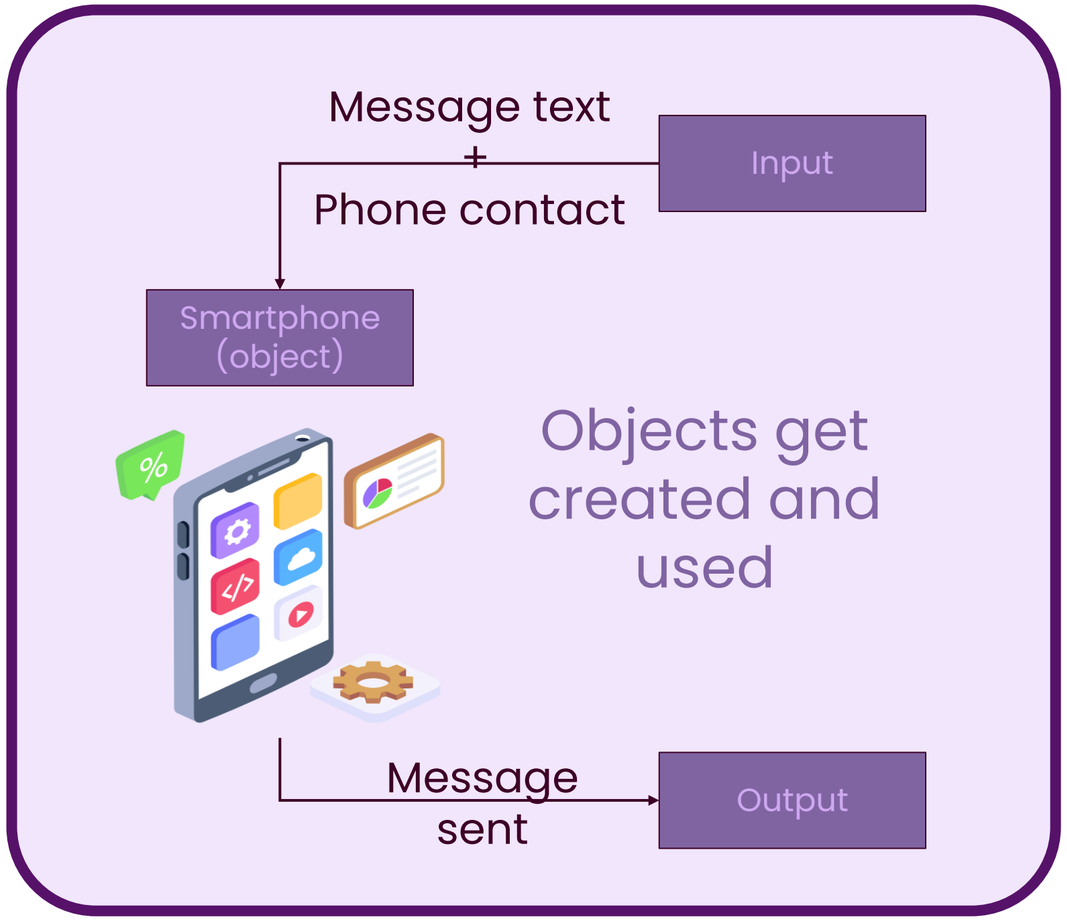
Output: The result of your interactions with the smartphone. If you send a message, the output is the message being delivered. If you set an alarm, the output is the alarm going off at the set time



The smartphone example

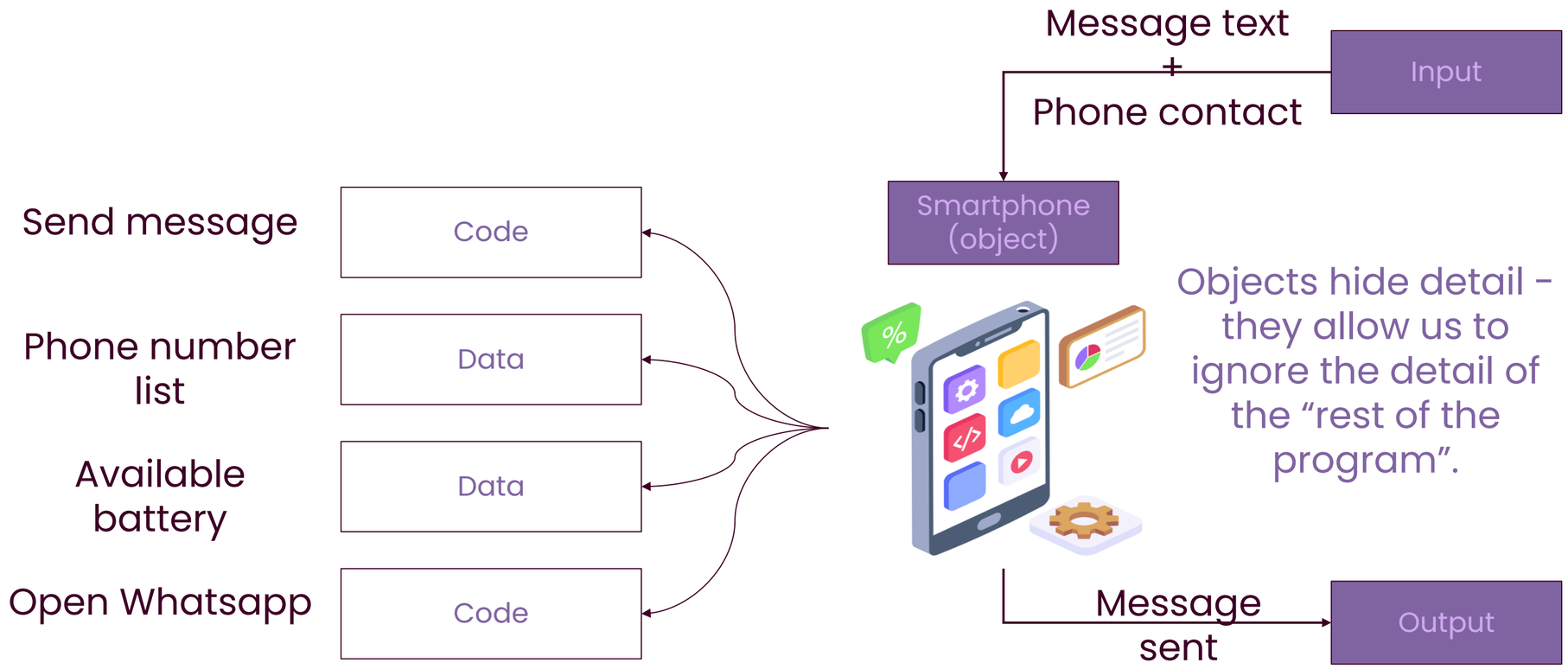
Object Oriented Programming

- In the real world, the smartphone and its capabilities are the result of programming using **OOP principles**
- **Each app** on the phone can also be considered **an object**, with its **own attributes and methods** that define what it is and how you can interact with it
- The simple and intuitive interface of a smartphone is a good representation of how objects in a program can be designed to interact with users and other objects in a system



The smartphone example

Python Object



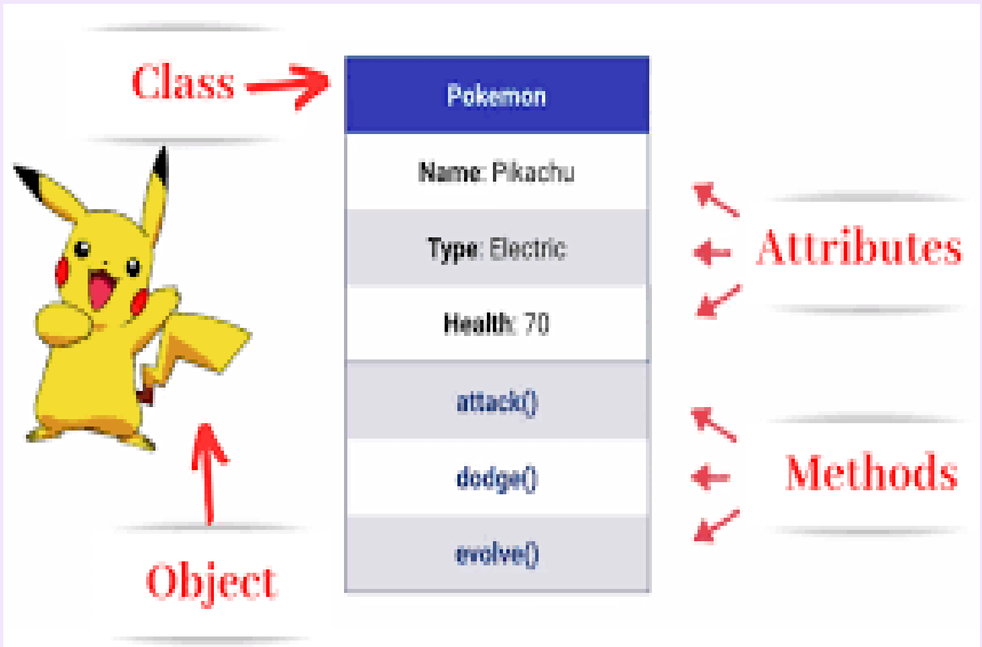
Python Object

What is it?

- **An Object is a bit of self-contained Code and Data**
- A key aspect of the Object approach is to **break the problem** into smaller understandable parts (divide and conquer)
- Objects have boundaries that allow us to ignore unneeded detail
- We have been using objects all along: String Objects, Integer Objects, Dictionary Objects, List Objects...
- For example, a string is just a sequence of characters. But instead of treating it as a raw array of letters, Python wraps it in a str class object.

What are the object main components?

- **Class** - a template
- **Method** - A defined capability of a class
- **Field or attribute** - A bit of data in a class
- **Object or Instance** - A particular instance of a class
- **Constructor** – Code that runs when an object is created



Python Classes

Properties

- When an object of a class is created, the class is said to be **instantiated**.
- All the **instances share** the attributes and the behaviour of the class. But the values of those attributes, i.e. the state are unique for each object.
- A single class may have **any number of instances**.

How to build a Python object?

- **Class Definition**: Pokemon is defined with the special method __init__, which **initializes the object with its attributes**: name, type, and health.
- **Instance Creation:** Pikachu is an instance of the Pokemon class, created with the specified attributes.

Motivation.txt

```
# Class definition
class Pokemon:
    def __init__(self, name, type, health):
        self.name = name
        self.type = type
        self.health = health

# Creating an instance of the Pokemon class
pikachu = Pokemon("Pikachu", "Electric", 70)
```

How to build a Python object?

The constructor: __init__ method

- It is a **special method** (function) that is **automatically called** when an **object is created** from a class.
- It is used to **initialize the attributes** of the object.
- It is not required in a class, but it is commonly used to set up the initial state of an object.
- It always **takes at least one parameter**, conventionally named self, which refers to the instance of the object being created.
- You can define additional parameters in **__init__** to accept values that will be used to initialize the attributes

How to build a Python object?

self parameter

- It is a convention in Python that **represents the instance** of the object.
- It is the **first parameter in instance methods** (including **__init__**), and it is **automatically passed** when a method is called on an instance
- It is used to **access and modify the attributes of the instance** within the class

How to build a Python object?

Attributes of the object instance

Motivation.txt

```
class MyClass:
    def __init__(self, attribute1, attribute2):

        self.attribute1 = attribute1
        self.attribute2 = attribute2

    def print_attributes(self):
        print(f"Attribute1: {self.attribute1},
Attribute2: {self.attribute2}")

my_object = MyClass(attribute1_value,
attribute2_value)
```

When creating an object from a class, Python automatically passes the instance (represented by **self**) to the **__init__** method

The **self** argument is not passed when the method/function is called. Python does that automatically

attribute1_value and **attribute2_value** are passed to **__init__** as arguments, and **self** refers to the newly created instance (my_object), allowing to set the initial state of the object

How to build a Python object?

- Attributes of class instances can be modified and deleted
- An object of an instance can be deleted

Motivation.txt

```
class MyClass:
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    def print_attributes(self):
        print(f"Attribute1: {self.attribute1},
Attribute2: {self.attribute2}")

my_object = MyClass(attribute1_value, attribute2_value)
# Modify attributes
my_object.attribute2 = attribute2_modifed
# Deleting attributes
del my_object.attribute1
# Delete the object
del my_object
```

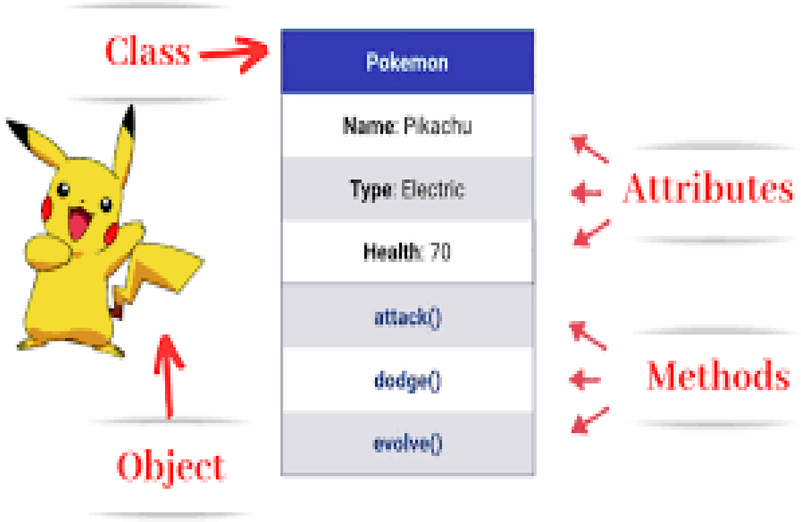
How to build a Python object?

Methods: The attack, dodge, and evolve methods are actions that the Pikachu object can perform. When called, they print out a message indicating that Pikachu has performed the action

Motivation.txt

```
class Pokemon:
    def __init__(self, name, type, health):
        self.name = name
        self.type = type
        self.health = health

    def attack(self):
        print(f"{self.name} used attack!")

    def dodge(self):
        print(f"{self.name} dodged the attack!")

    def evolve(self):
        print(f"{self.name} is evolving!")

# Creating an instance of the Pokemon class
pikachu = Pokemon("Pikachu", "Electric", 70)
# Using the object's methods
pikachu.attack()   # Pikachu used attack!
pikachu.dodge()    # Pikachu dodged the attack!
pikachu.evolve()   # Pikachu is evolving!
```

How to build a Python object?



Motivation.txt

```python
class Pokemon:
    def __init__(self, name, type, health):
        self.name = name
        self.type = type
        self.health = health

    def attack(self):
        print(f"{self.name} used attack!")

    def dodge(self):
        print(f"{self.name} dodged the attack!")

    def evolve(self):
        print(f"{self.name} is evolving!")

# Creating an instance of the Pokemon class
pikachu = Pokemon("Pikachu", "Electric", 70)
# Using the object's methods
pikachu.attack()   # Pikachu used attack!
pikachu.dodge()    # Pikachu dodged the attack!
pikachu.evolve()   # Pikachu is evolving!
```

Class variables

- Instance variables hold data unique to each instance, while **class variables store attributes and methods shared by all instances of the class**.
- Instance variables are assigned within a constructor or method using self, whereas **class variables are assigned directly within the class definition**.

Motivation.txt

```
class Pokemon:
    # Class variable
    size = "small"

    def __init__(self, name, type, health,
size=None):
        # Instance variables
        self.name = name
        self.type = type
        self.health = health
        # If no size is passed, fall back to the
class variable
        self.size = size if size else Pokemon.size

    def attack(self):
        print(f"{self.name} used attack!")
```

Class methods

- **Class methods** in Python are defined by using the **@classmethod decorator**
- They define a method within a class that's **not bound** to a specific instance of that class
- Unlike instance methods, which operate on an instance of the class and can access and modify the instance's attributes, **class methods affect the class as a whole**
- Class methods receive the **class as the first argument**, **cls,** instead of self which is used in instance methods. This means they can modify class state that applies across all instances of the class, rather than instance-specific data.

Motivation.txt

```
class Pokemon:
    def __init__(self, name, type, health):
        # Instance variables
        self.name = name
        self.type = type
        self.health = health

    def attack(self):
        print(f"{self.name} used attack!")

    @classmethod
    def from_egg(cls, name, type):
        return cls(name, type, health=100)

# Using the alternative constructor
new_pokemon = Pokemon.from_egg("Pikachu", "Electric")
print(new_pokemon.health)  # Output: 100
```

Static methods

- **Static methods** in Python are defined by using the **@staticmethod decorator**.
- It is a **method that belongs to a class** rather than belonging to an instance of a class.
- Static method can be called on the class itself, without creating an instance of the class.
- Static methods **do not have access to the self parameter nor the class**, and therefore **cannot modify** object instance state.

Motivation.txt

```
class Pokemon:
    # Class variable
    size = "small"

    def __init__(self, name, type, health):
        # Instance variables
        self.name = name
        self.type = type
        self.health = health

    def attack(self):
        print(f"{self.name} used attack!")

    @staticmethod
    def get_pokemon_types():
        print(f"The size is {Pokemon.size}")
        return ["Electric", "Water", "Fire"]

pokemon_types = Pokemon.get_pokemon_types()
print(pokemon_types)  # ['Electric', 'Water', 'Fire']
```

How to find python methods

Motivation.txt

```
>>> x = 'abc'
>>> type(x)
<class 'str'>

>>> type(2.5)
<class 'float'>

>>> type(2)
<class 'int'>

>>> y = list()
>>> type(y)
<class 'list'>

>>> z = dict()
>>> type(z)
<class 'dict'>
```

Motivation.txt

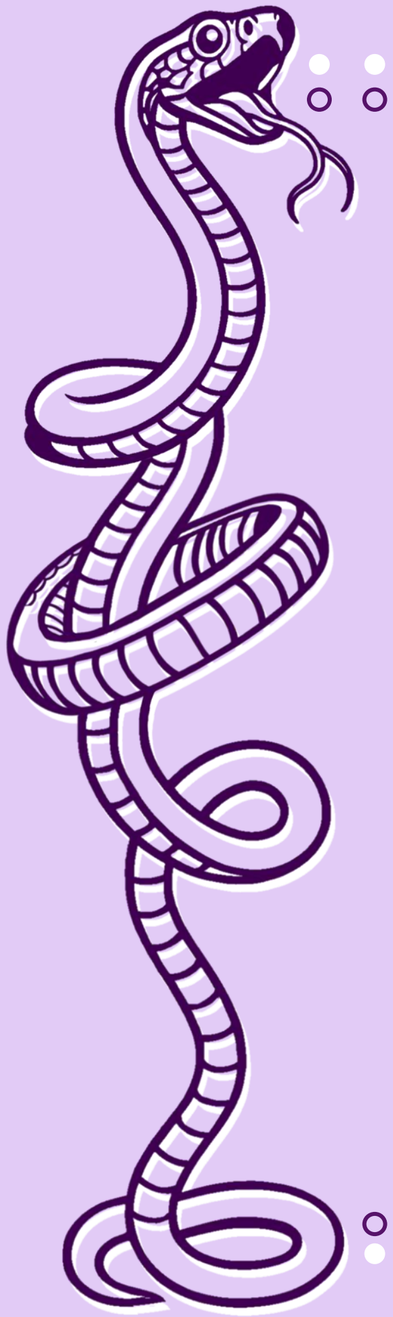
```
>>> dir(x)
['capitalize', 'casefold', 'center', 'count', 'encode',
 'endswith', 'expandtabs', 'find', 'format', 'lower', 'lstrip',
 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
'rjust',
 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper',
 'zfill']

>>> dir(y)
['append', 'clear', 'copy', 'count', 'extend', 'index',
'insert',
 'pop', 'remove', 'reverse', 'sort']

>>> dir(z)
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',
 'popitem', 'setdefault', 'update', 'values']
```

How to find python methods

- The **dir()** command **lists capabilities**
- You **should ignore the ones with underscores**. These are used by Python itself – private variables
- The rest are real operations that the object can perform
- It is like type() - it tells us something *about* a variable

Motivation.txt

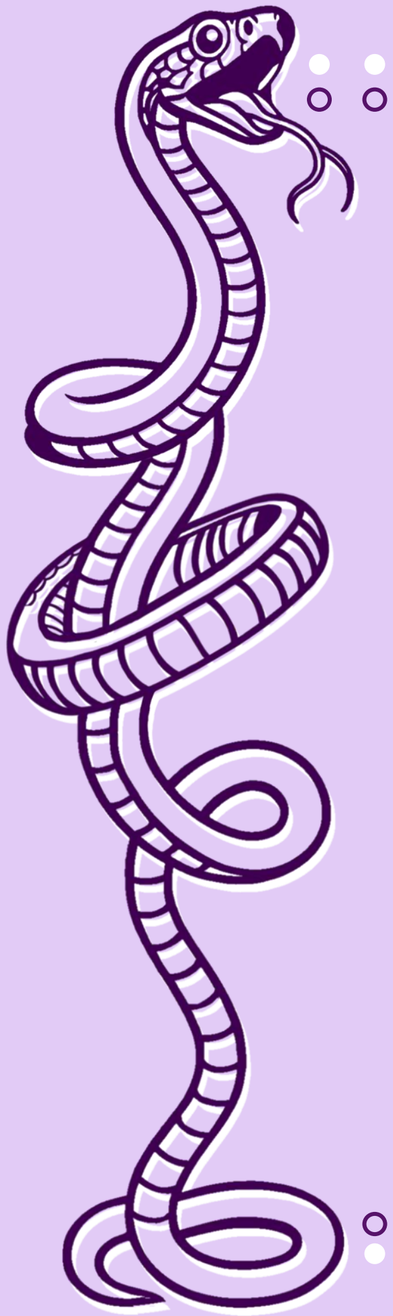
```
>>> dir(x)
['capitalize', 'casefold', 'center', 'count', 'encode',
 'endswith', 'expandtabs', 'find', 'format', 'lower', 'lstrip',
 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
'rjust',
 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper',
 'zfill']

>>> dir(y)
['append', 'clear', 'copy', 'count', 'extend', 'index',
'insert',
 'pop', 'remove', 'reverse', 'sort']

>>> dir(z)
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',
 'popitem', 'setdefault', 'update', 'values']
```



Class wrap-up

What have we learned today?

- Introduced the core principles of OOP, emphasizing the organization of code around objects, which encapsulate data and behaviour.
- Explored the concept of a class as a blueprint for creating objects, providing a structure for attributes and methods.
- Discussed attributes as properties of objects, representing data associated with instances of a class.
- Explored methods as functions defined within a class, enabling the encapsulation of behaviour specific to the class.
- Emphasized the creation of objects as instances of a class, where attributes and methods come together to represent and manipulate data.



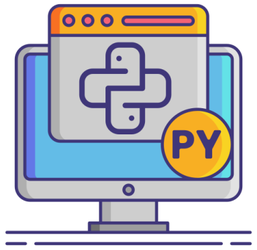
PRACTICE
PRACTICE
PRACTICE

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

README.md — added initial course structure and .README~ — 17 hours ago

README

Natixis Python Level 3

Course Overview

This advanced Python course is designed for students who have already mastered intermediate programming concepts and want to expand their skills toward object-oriented programming (OOP), modular design, and interactive application development. Students will learn to structure Python code professionally, understand reusability and abstraction, manage projects using Git and GitHub, and create Streamlit applications for data-driven interfaces.

Course Structure (Intermediate Python - Level 3)

0 forks

Report repository

Releases

No releases published

Create a new release

Packages

No packages published

Publish your first package

Link to exercises: https://github.com/cat-fss/natixis_python_level_3/blob/master/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?

Alexandra Oliveira m.alexandra.ro@gmail.com