Object-Oriented Programming

LINCS Python workshop
By Matthieu Gouel (matthieu.gouel@lip6.fr)

"So there is a duck..."

Imagine you often interact with different input/output formats in your programs.
You decide to write once and for all a set of functions to handle this.

In [16]:
```python
import json
import yaml

def to_json(data: dict):
    """Convert a dict to JSON."""
    return json.dumps(data)

def to_yaml(data: dict):
    """Convert a dict to YAML."""
    return yaml.dump(data)

config = {"parameter1": 1, "parameter2": 1.0}

print(to_json(config))
print(to_yaml(config))
```

```
{"parameter1": 1, "parameter2": 1.0}
parameter1: 1
parameter2: 1.0
```

Can we do better?

This "procedural" approach works but, we could see the problem in a different perspective.

We could rather have an *object* that would hold the state of our config data, and allow us to output it in different formats.

In [33]:
```python
# Let's create a "class" which is like the blueprint of the object we want to create
# The class has one "attribute": data
class Config:
    def __init__(self, data):
        self.data = data  # This is one attribute of the class Config

# Now let's instantiate our object
config = Config({"parameter1": 1, "parameter2": 1.0})

print(config.data)
```

```
{'parameter1': 1, 'parameter2': 1.0}
```

But we didn't add methods to interact with the attribute of the object, for instance output the config data in different formats.

In [47]:
```
import json
import yaml


class Config:
    def __init__(self, data):
        self.data = data
    # Here we add some methods to the class
    def json(self):
        return json.dumps(self.data)
    def yaml(self):
        return yaml.dump(self.data)

config = Config({"parameter1": 1, "parameter2": 1.0})

print(config.json())
print(config.yaml())
```

```
{"parameter1": 1, "parameter2": 1.0}
parameter1: 1
parameter2: 1.0
```

Now we have an object "config", we can also access to the state of its attribute very easily. No need for additional methods.

In [48]:
```
# We create an object "config" with intial attribute
config = Config({"parameter1": 1, "parameter2": 1.0})
print(config.data)

# Let's change the attribute of the object
config.data["parameter1"] = 2

print(config.json())
```

```
{'parameter1': 1, 'parameter2': 1.0}
{"parameter1": 2, "parameter2": 1.0}
```

Note that you can also create methods to implement more sophisticated changes on the attributes

In [49]:
```
class Config:
    def __init__(self, data):
        self.data = data
    # This method changes the state of the data
    def reset(self):
        self.data = {}

config = Config({"parameter1": 1, "parameter2": 1.0})
print(config.data)

config.reset()
print(config.data)
```

```
{'parameter1': 1, 'parameter2': 1.0}
{}
```

Also, methods are like regular functions, they can take parameters.

In [39]:
```
class Config:
    def __init__(self, data):
        self.data = data
```

```
    def display(self, prefix="data: "):
        return prefix + str(self.data)

config = Config({"parameter1": 1, "parameter2": 1.0})
print(config.data)

print(config.display(prefix="mydata: "))
```

```
{'parameter1': 1, 'parameter2': 1.0}
mydata: {'parameter1': 1, 'parameter2': 1.0}
```

Exercise #1

Write a class "Rectangle".

- 2 attributes (width, height) that corresponds to the lengths of each side
- a method "area"
- a method "perimeter"

In [62]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def perimeter(self):
        return 2 * (self.width  + self.height)

    def area(self):
        return self.width * self.height

rectangle = Rectangle(2, 3)
assert rectangle.perimeter() == 10
assert rectangle.area() == 6
```

In [64]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def perimeter(self):
        return 2 * (self.width  + self.height)

    @property
    def area(self):
        return self.width * self.height

rectangle = Rectangle(2, 3)
assert rectangle.perimeter == 10
assert rectangle.area == 6
```

Encapsulation

Encapsulation is the general concept of having data and methods as a same unit: an object. The data (attributes) are encapsulated and can only be manipulated with the methods.

In Python, this concept is loosen:

- All attributes are by default public (accessible directly).
- Protected attributes begins with `_` but it's just a convention
- Private attributes begins `__` (but you can hack to access it directly).

So, in Python, it's a good practice to have the attributes public and never create getters and setters methods.

In [40]:
```
class Config:
    def __init__(self, data):
        self.data = data
        self._data = data
        self.__data = data

config = Config("data")
print(config.data)
print(config._data)

# This is, in principle, not directly accessible
print(config.__data)
```

```
data
data
```

```
Traceback (most recent call last)
/var/folders/rv/1898p2xd2dg1m1tg751kx7j80000gn/T/ipykernel_54576/1031983735
<module>

[Errno 2] No such file or directory:
'/var/folders/rv/1898p2xd2dg1m1tg751kx7j80000gn/T/ipykernel_54576/103198373
```

```
AttributeError: 'Config' object has no attribute '__data'
```

Inheritance

Instead of starting from scratch, a class can inherit from a parent class and obtain its attributes and methods.

In [41]:
```
class Config:
    def __init__(self, data):
        self.data = data
    def reset(self):
        self.data = {}

# This class inherits for all the methods of "Config"
class SpecialConfig(ProgramConfig):
    pass

special_config = SpecialConfig({"parameter": 1})
print(special_config.data)

special_config.reset()
print(special_config.data)
```

```
{'parameter': 1}
{}
```

Of course, we can add methods and attributes to the child class.

In [42]:
```
class Config:
    def __init__(self, data):
        self.data = data
    def reset(self):
        self.data = {}

class SpecialConfig(Config):
    def __init__(self, data, special_tag="special: "):
        super().__init__(data)  # super() allow to call a parent class method (here
        self.special_tag = special_tag

    def display(self):
        return self.special_tag + str(self.data)

special_config = SpecialConfig({"parameter": 1})
print(special_config.data)

print(special_config.display())

special_config.reset()
print(special_config.data)
```

```
{'parameter': 1}
special: {'parameter': 1}
{}
```

In some cases, we make use of "multiple inheritence" where a class can inheritsfrom multiple parent classes.

It's often not a very good practice because it complicates the code. One useful pattern: *mixins*.

In [45]:
```
class Config:
    def __init__(self, data):
        self.data = data
    def reset(self):
        self.data = {}

# This is a mixin class.
# It implements a feature that can be added by inheritance to different classes.
class PrettyConfigMixin:
    def pretty(self):
        return f"✨ {self.data} ✨"


# This class inherits for all the methods of "Config" and "PrettyConfigMixin"
class SpecialConfig(Config, PrettyConfigMixin):
    pass


special_config = SpecialConfig({"parameter": 1})
print(special_config.pretty())
```

```
✨ {'parameter': 1} ✨
```

Exercise #2

Write a "Square" class that inherit from your class rectangle

In [66]:
```
class Square(Rectangle):
    def __init__(self, side):
```

```
        super().__init__(side, side)

square = Square(2)
assert square.perimeter == 8
assert square.area == 4
```

Polymorphism

Polymorphism is multiple objects having a shared interface with different behavior.

In [26]:
```
print(len([1, 2, 3]))  # count the number of item on the list
print(len("toto"))  # count the number of characters on the string
```

```
3
4
```

We can simply implement polymorphism by having different classes with the same methods. But we can also have a parent class with the shared methods implemented.

In [29]:
```
class BaseConfig:
    def __init__(self, data):
        self.data = data
    def json(self):
        return json.dumps(self.data)
    def yaml(self):
        return yaml.dump(self.data)

class ComponentConfigA(BaseConfig):
    pass

class ComponentConfigB(BaseConfig):
    pass

component_config_A = ComponentConfigA({"parameter":1})
component_config_B = ComponentConfigB({"parameter":2})

for component in (component_config_A, component_config_B):
    print(component.json())
```

```
{"parameter": 1}
{"parameter": 2}
```

Method overloading

Method overloading is the ability to have the same method name but with different parameters. It's not possible in Python.

In [23]:
```
class Test:
    def display(self, data):
        return data
    def display(self,data, prefix):
        return prefix + data

test = Test().display(1)
```

```
Traceback (most recent call last)
/var/folders/rv/1898p2xd2dg1m1tg751kx7j80000gn/T/ipykernel_54576/791214789.
<module>
```

```
[Errno 2] No such file or directory:
'/var/folders/rv/1898p2xd2dg1m1tg751kx7j80000gn/T/ipykernel_54576/791214789
```

```
TypeError: display() missing 1 required positional argument: 'prefix'
```

Magic methods

Python has a system of "magic methods" or "dunder methods" you can override in your classes. They always begin and end with `__` .

You can describe how your class behave with operators (`>` , `<` , ...), with built-in methods (`len()` , `print()` , ...), class representation, ...

See: https://docs.python.org/3/reference/datamodel.html

In [24]:
```
class MyContainer:
    data = [1, 2, 3]

container = MyContainer()
print(len(container))
```

```
Traceback (most recent call last)
/var/folders/rv/1898p2xd2dg1m1tg751kx7j80000gn/T/ipykernel_53632/3244311075
<module>

[Errno 2] No such file or directory:
'/var/folders/rv/1898p2xd2dg1m1tg751kx7j80000gn/T/ipykernel_53632/324431107
```

```
TypeError: object of type 'MyContainer' has no len()
```

In [26]:
```
class MyContainer:
    data = [1, 2, 3]

    def __len__(self):
        return len(self.data)

container = MyContainer()
print(len(container))
```

```
3
```

In [27]:
```
class MyContainer:
    data = [1, 2, 3]

container = MyContainer()
print(container)
```

```
<__main__.MyContainer object at 0x110c1da00>
```

In [28]:
```
class MyContainer:
    data = [1, 2, 3]

    def __repr__(self):
        return f"✨ {self.data} ✨"
```

```
container = MyContainer()
print(container)
```

✨ [1, 2, 3] ✨

In [18]:
```
class MyContainer:
    data = [1, 2, 3]

container = MyContainer()
print(container())
```

```
Traceback (most recent call last)
/var/folders/rv/1898p2xd2dg1m1tg751kx7j80000gn/T/ipykernel_54576/387453558.
<module>

[Errno 2] No such file or directory:
'/var/folders/rv/1898p2xd2dg1m1tg751kx7j80000gn/T/ipykernel_54576/387453558
```

```
TypeError: 'MyContainer' object is not callable
```

In [19]:
```
class MyContainer:
    data = [1, 2, 3]

    def __call__(self):
        return f"✨ {self.data} ✨"

container = MyContainer()
print(container())
```

✨ [1, 2, 3] ✨

Abstract classes

Abstract classes are kind of parent classes that don't mean to be instantiated.
It allow to "force" a child class to implement methods so it's useful for polymorphism.
It's possible to do this with the "ABC" built-in library.

In [32]:
```
from abc import ABC, abstractmethod

class BaseConfig(ABC):
    def __init__(self, data):
        self.data = data

    @abstractmethod
    def json(data):
        pass
    @abstractmethod
    def yaml(data):
        pass

class Config(BaseConfig):
    pass

config = Config({"parameter":1})
```

```
Traceback (most recent call last)
/var/folders/rv/1898p2xd2dg1m1tg751kx7j80000gn/T/ipykernel_54576/3589430868
<module>
```

```
[Errno 2] No such file or directory:
'/var/folders/rv/1898p2xd2dg1m1tg751kx7j80000gn/T/ipykernel_54576/358943086
```

```
TypeError: Can't instantiate abstract class Config with abstract methods json
```