OOP summary

Features

- **Class**: a data type with attributes and methods.
- **Object**: an instance of a class
- Type of methods
    - **Instance** methods: works on instance, first argument is *self*; it can change the object state;
    - **Class** methods: works on the class, first argument is *class*; it can change class;
    - **Static** methods: works without any object/class; can’t change class/object state.

```
class Signal:

  description = "That's the signal"

  # initialize attributes with the special method called __init__
  def __init__(self, amplitude, frequency, offset, func):
    self.amp = amplitude
    self.freq = frequency
    self.offset = offset
    self.function = func

# instance method
def print_period(self):
  print(1/self.freq)

@classmethod
def print_period(cls):
  print(cls.description)

@staticmethod
def print_period():
  print("That's a static method")
```

Add feature to existing class

- Inheritance, see below

- Composition, a new class being composed by other class instances
- In simple terms:
  - A *SinBasedSignal* is a Signal (**inheritance**)
  - A *SinBasedSignal* has a Signal Waveform (**compositions**)

```
class SinSignal(Signal):
  pass

class SinSignal:

def __init__(self):
  function = SinWaveform()
```

Intefaces

- Defining how class behaves without detailing how they behave
- Work as blueprint or templates, they need to be implemented before getting used

```
class Modulator:

  @abstractmethod
  def modulate(self, ...):
    raise NotImplemented("You can't call this method. Implement this ABC first")

@abstractmethod
def demodulate(self, ...)
    raise NotImplemented("You can't call this method. Implement this ABC first")

class AMModulator(Modulator):
  def modulate(self, ...):
    # implement it here
    pass

  def demodulate(self, ...):
    # implement it here
    pass
```

Iterator DP

Hide the traversing complexity of an *iterable* object using an *iterator* that gets called each time the next value is required.

- It needs to handle the case when the iterable object has been fully looped through

Abstract Base Classes Required:

- Iterator: define that the object is an iterator
- Iterable: define that the object is iterable

```
class Collection(Iterable):
  def __iter__(self):
    pass

class Iterator(Iterator):
  def __next__(self):
    pass
```

Going deeper

- **Encapsulation**: keep internal logic private and separated from public access
    - Information hiding is the principle
- **Inheritance**: Derive new classes from a parent one to add new features or override existing behavior
    - Inheritance generates an hierarchy
    - Access to ancestor classes is supported
    - Python has support for multiple inheritance
        - Example: mixing
- **Polymorphism**: classes that derives from the same base class can be used instead of the base class
- **Loose coupling:** Detaching components from each other paying in terms of complexity

    - Adding a new feature is a matter of changing the components, without breaking the existing functionality
        - Using interfaces instead of concret classes makes the logic loose coupled
    - Is the opposite of **Tight Coupling**, where a component cannot exist without another one
        - Interfaces and implementations are tightly coupled

SOLID Principles

Single Responsability: a class with a single responsability

- Multiple responsability, multiple classes

OpenClose: open to extension, closed to modifications

- Legacy code cannot be changed: extend it

Liskov substitution: use specific types without altering behavior

- If you use subtypes mixed with a supertype, you'd not get any error

Interface segregation: small and specific interfaces

- Narrow contracts: define exactly what they're going to do

Dependency Injection: Abstraction as dependency

- Depends on abstractions (interfaces), not concrete classes
- Let someone *injects* the correct implementation

Unified Model Language

From https://www.codeproject.com/Articles/618/OOP-and-UML

> UML, Unified Modeling Language, is a standard notation for the modeling of real-world objects as a first step in developing an object oriented program. It describes one consistent language for

> specifying, visualizing, constructing and documenting the artifacts of software systems.

Design patterns

General solutions to recurrent problem when dealing with objects ⇒ try to do not reinventing the wheel

Splitted into three categories:

- **Creational**, deals with object creation
    - Example:
        - factory method: return an instance based on some condition
        - builder: build the instance setting attributes first and then calling a *build* method
- **Behavioural**, deals with defining algorithms on top of objects
    - Example:
        - iterator: define how to iterate an iterable
        - strategy: define how to run an algorithm that belongs to the same family
- **Structural**, deals with assembling objects in larger structure
    - Example:
        - decorator: add new features to existing objects
        - facade: hide the complexity of using multiple objects with simplified access
    - Worth to think about: decorator pattern and Python decorators
        - Where’s the difference?

Links

- Going deeper on the fantastic world of signal/waves with Python
    - https://github.com/AllenDowney/ThinkDSP

- Playing with UML (not just with classes)
    - https://plantuml.com/
    - https://www.planttext.com
- Instance methods, class methods, static methods demystified
    - https://realpython.com/instance-class-and-static-methods-demystified/
- More on design patterns
    - https://refactoring.guru
- Design patterns, aka “do not reinvent the wheel”
    - https://en.wikipedia.org/wiki/Design_Patterns
- Dataclasses
    - https://docs.python.org/3/library/dataclasses.html
- More about abstract base classes
    - https://docs.python.org/3/library/collections.abc.html
    - https://docs.python.org/3/library/abc.html
- Everything’s an object in Python, classes too
    - https://medium.com/swlh/everything-is-an-object-in-python-learn-to-use-functions-as-objects-ace7f30e283e
- What’s the Dependency Injection
    - https://en.wikipedia.org/wiki/Dependency_injection
    - How the Java Spring Framework uses it
        - https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring
- About loose coupling vs tight coupling (with non-programming world example)
    - https://stackoverflow.com/questions/2832017/what-is-the-difference-between-loose-coupling-and-tight-coupling-in-the-object-o
- Abstract classes vs Interfaces in Java

- https://www.guru99.com/interface-vs-abstract-class-java.html