Scientific Software Development with Python

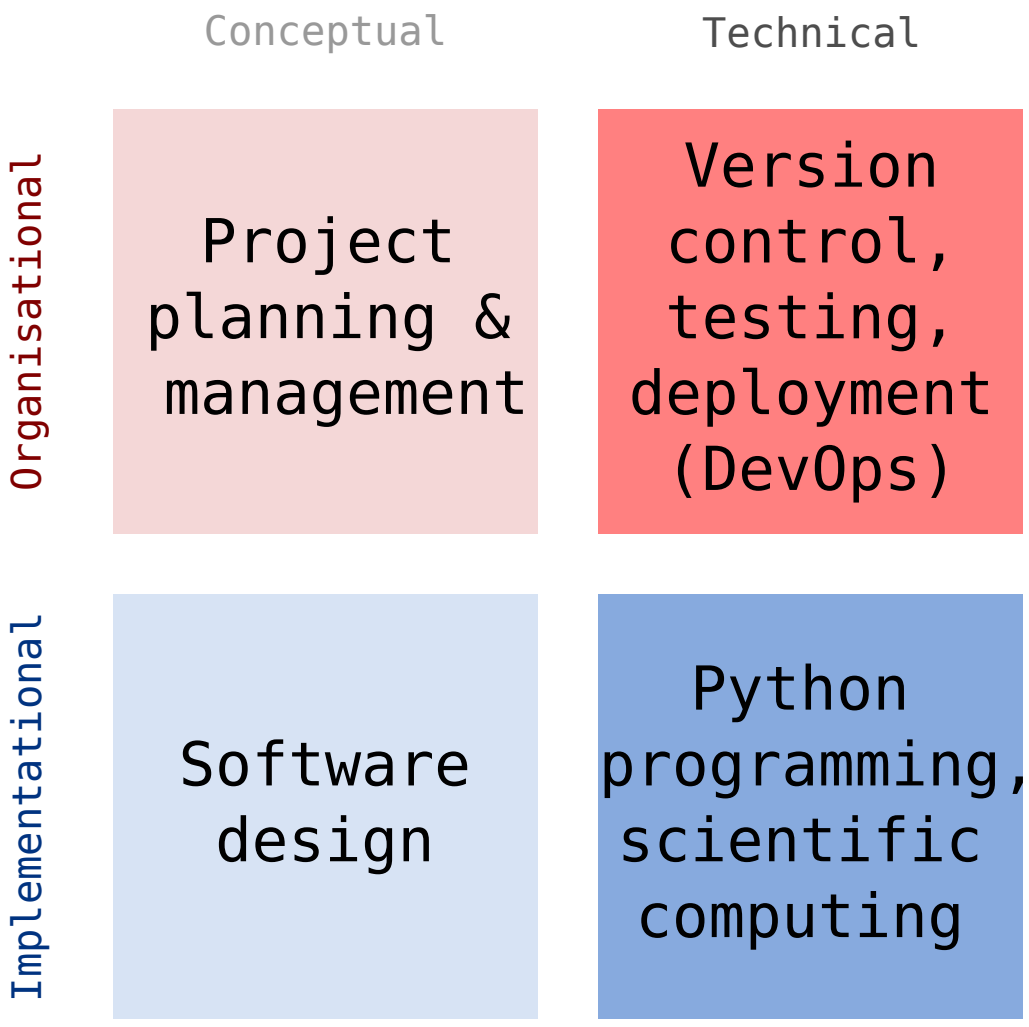
Object oriented programming — Part 1

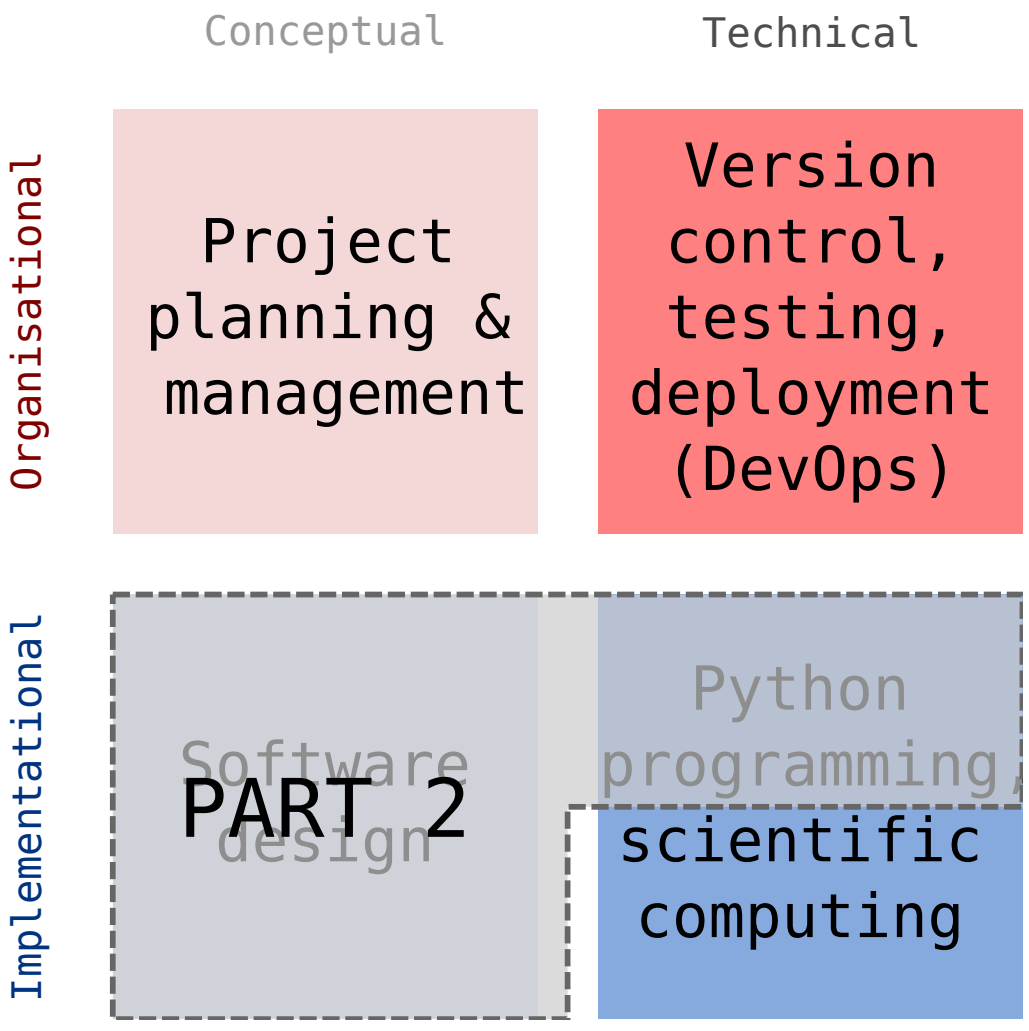
CHALMERS
UNIVERSITY OF TECHNOLOGY

Simon Pfreundschuh
Department of Space, Earth and Environment

Agenda

1. Overview
2. Object oriented thinking
3. Classes
4. Inheritance
5. Aggregation and composition


Conceptual
Technical
Organisational
Implementational
Project planning & management
Version control, testing, deployment (DevOps)
Software design
Python programming, scientific computing


Conceptual
Technical
Organisational
Implementational
Project planning & management
Version control, testing, deployment (DevOps)
Software design
Python programming
scientific computing
PART 2

This lecture

- Example of object oriented design
- Object oriented programming in Python
- Principle of object oriented programming

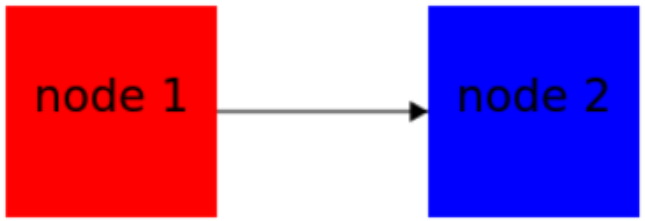
Next lecture

- Specifying interfaces using abstract classes
- Common Python design patterns
- Structural vs. object oriented programming

Agenda

- Modeling approach
- Processes are modeled using interacting objects.
- A class describes a type of object.
- Objects of a given class are called *instances*.

Example: Drawing diagrams



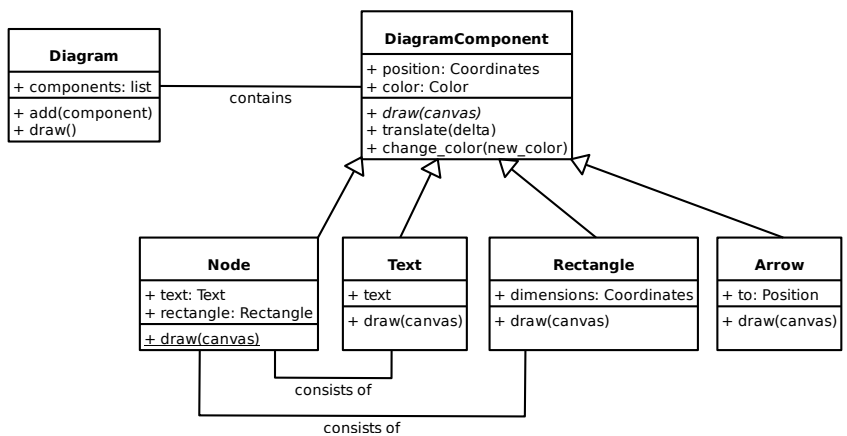
- Objects: Figure, rectangles, text, arrows
- Actions:
  - add rectangle
  - add text to rectangle
  - draw arrow from rectangle to rectangle

Unified modeling language (UML)

- Graphical modeling language
- Formal application quite complex
- But: Useful and intuitive way to communicate class relations

Example: Drawing diagrams

- Corresponding UML class diagram:



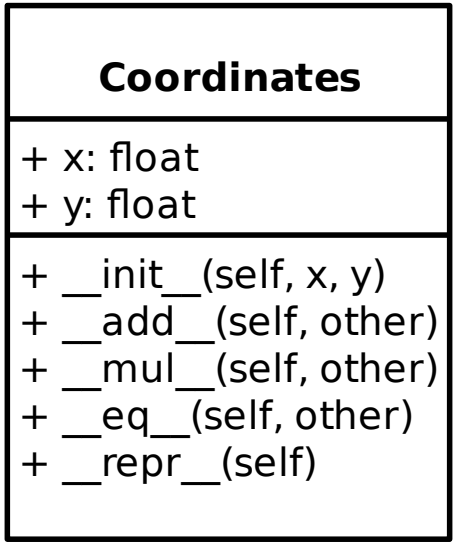
Unified modeling language (UML)

- UML class diagrams visualize classes and their relationships.
- Can be mapped directly to code.

So how can we map the different components of a UML diagram to Python?

Classes

- Block represents class
- Top: Class name
- Middle: Attributes (data)
- Bottom: Methods (actions)



Agenda

Classes in Python

- Defined using the `class` keyword.
- Definition consists of class methods.
- `__init__` function used to initialize new object[1].

```
class Coordinates:
    """
    The coordinate class represents two-dimensional, Cartesian
    coordinates.
    ...
    """
    def __init__(self, x, y):
        """
        Create pair of coordinates.
        ...
        """
        self.x = x
        self.y = y
    ...
```

[1] There's also the `__new__` function, which is called when a new object is created, but its usage is quite advanced.

Methods

- Methods are functions that act on class objects.
- General syntax:

```
object.method_name(arg_1, arg_2, ...)
```

  - Calls method `method_name` defined in `object`'s class with arguments `object, arg_1, arg_2, ...`

The object whose method is called upon is always passed as the first argument (`self`) to the class method. **This is how the method gains access to the class attributes.**

Attributes

- Attributes represent the specific properties (data) of a class instance.
- Attributes *should be set in* `__init__` *method*.
- But: This is not enforced. Attributes can be defined dynamically:

```
coordinates = Coordinates(1.0, 2.0)
print(coordinates.z) # Error
coordinates.z = 3.0
print(coordinates.z) # Prints: 3
```

Attribute access

- In contrast to other language, Python does not restrict access to class attributes
- **But**: Python does apply name mangling to attributes starting with two undescores[2]:

```
class A:
    def __init__(self):
        self.__attribute = 1

a = A()
print(a.__attribute) # Error
print(a._A__attribute) # Prints: 1
```

- Convention: Attributes prefixed with 1 underscore (_) should not be accessed from the outside.

[2] The attribute name becomes `_<class_name><attribute_name>`.

Attributes

- An alternative way of defining attributes in Python is to use the `@property` decorator:
- The function marked with `@property` is called when `obect.function_name` is accessed
- The function marked with `@<property_name>.setter` is called when a value is assigned to `object.<property_name>`

```
class Number:
    def __init__(self, number):
        self._number = number

    @property
    def plus_one(self):
        return self._number + 1

    @plus_one.setter
    def plus_one(self, plus_one):
        self._number = plus_one - 1
```

Example

```
number = Number(1)
print(number.plus_one) # Print: 2
number.plus_one = 1
print(number.plus_one) # Print: 1
print(number._number)  # Print: 0
```

Advantages

- The `@property` decorator allows defining getter and setter mehtods for Python attributes
- Allows computing properties on the fly (dynamic attributes)
- Omitting the setter make the attribute read-only
- Setter can be used to check validity of value

Static methods

- Static methods are methods that do not depend on a specific object and can therefore be called directly on the class
- Static methods are defined using the `@staticmethod` decorator[3, 4]

Example

```
class Color:
    @staticmethod
    def Black():
        return Color("#000000")
    ...
    def __init__(self, color_code):
        self.color_code = color_code

black = Color.Black()
```

[3] We'll learn more about decorators later.

[4] There's also the `@classmethod` decorator, which serves a similar purpose

Special class methods

- Python uses magic (or dunder) methods to implement special functionality (syntactic sugar):
  - `__repr__(self)`: Used to print output in interpreter.
  - `__str__(self)`: Called by `print` and `str` methods. Will use `__repr__` is defined.
  - `__add__(self, other)`: Implements + operator
  - `__mul__(self, other)`: Implements * operator
  - `__eq__(self, other)`: Implements == operator
- All special syntax in Python is implemented in this way[5]

[5] See official docs or `https://levelup.gitconnected.com/python-dunder-methods-ea98ceabad15` for an overview.

Exercise

- Exercise 1 in exercise notebook
- Time: 10 minutes

Brief summary

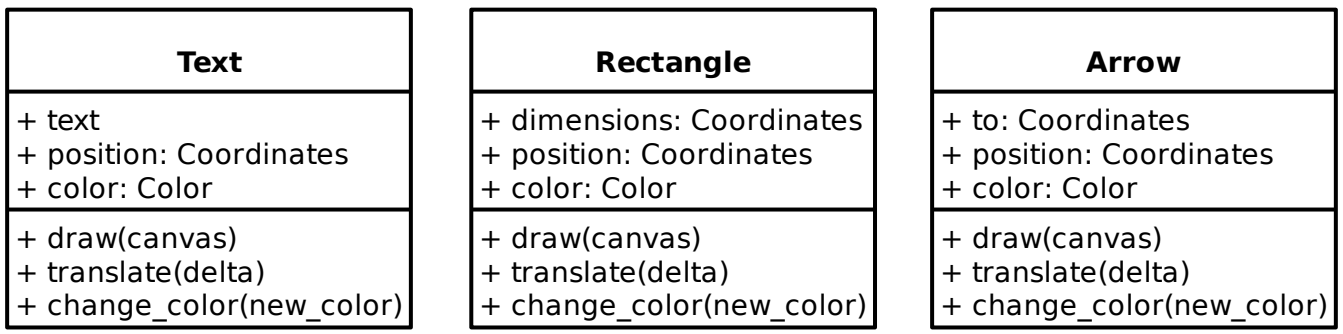
What we have learned:

- Classes define types of objects with given properties and actions
- How to define class methods in Python
- Two ways to define class attributes in Python
- How to define static and special methods

Agenda

Moving on with the design

- Let's add classes for the object we would like to draw:



Problem

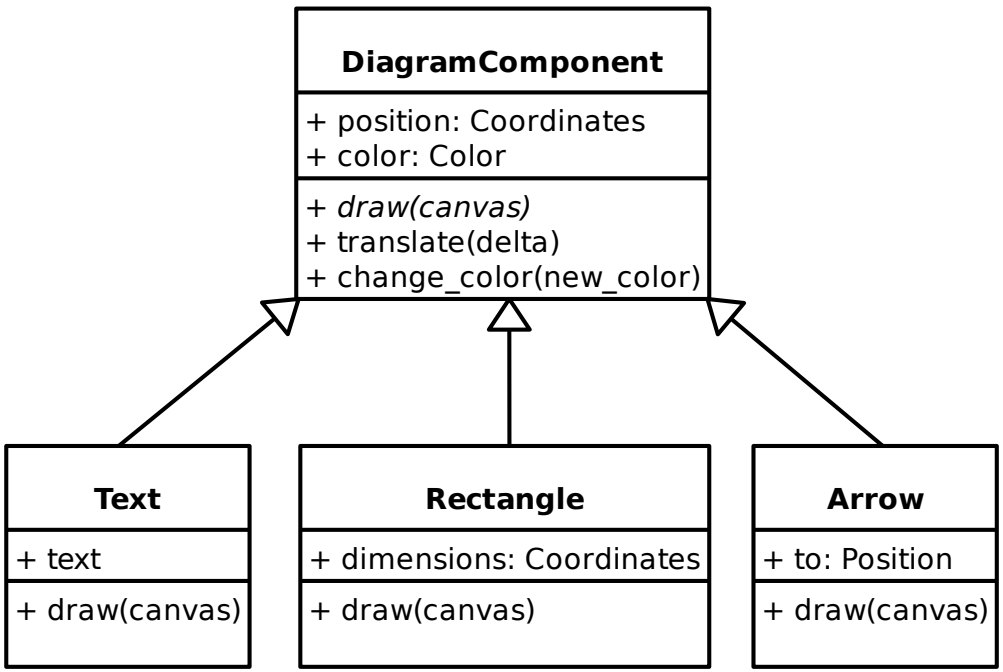
All classes have `position` and `color` attributes as well as `translate` and `change_color` functions, that do the same.

The DRY principle

- **Do not Repeat Yourself**
- Every piece of knowledge must have a single, unambiguous, authoritative representation within a system
- Duplicate code will sooner or later become inconsistent

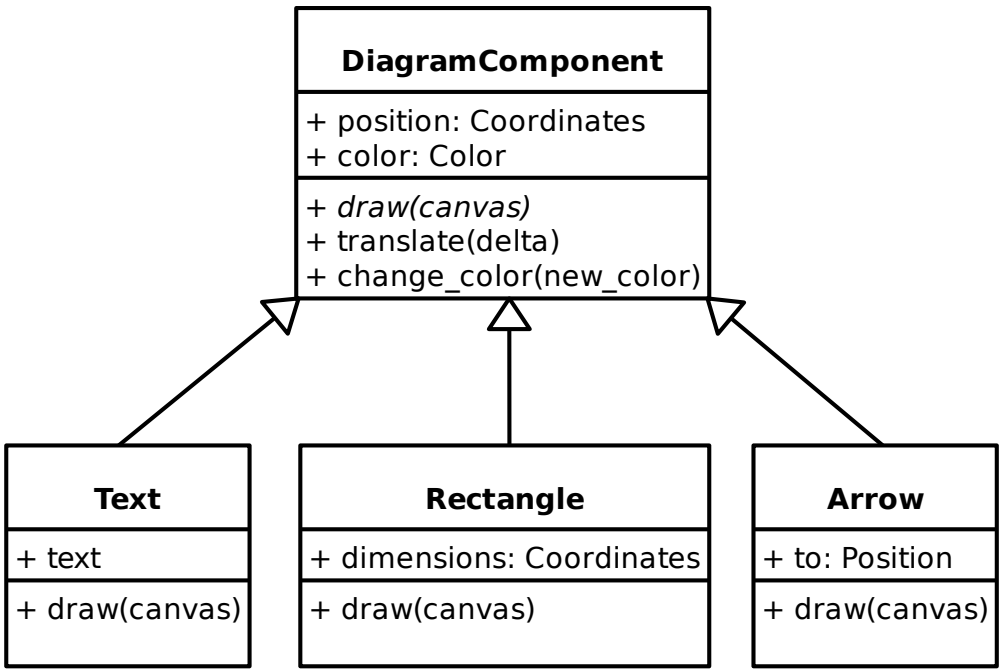
Inheritance

- Represents a *is a* relationship.
- *Child class* inherits from *base class*.
- The child class inherits all *methods and attributes* from its parent.



Inheritance in UML

- Represented by arrow with hollow head
- The arrow represents a *generalization* relation.



Inheritance in Python

- Basic syntax:

```
class ChildClass(BaseClass):
    def __init__(self, ...):
        super().__init__(...) # Calls __init__ of BaseClass
    ...
```

- All functions defined in `BaseClass` are available in `ChildClass`.

Important

`__init__` function of child class must call `__init__` function of base class to ensure object is properly initialized.[6]

[6] Exceptions are classes that don't define any attributes and therefore don't need to be initialized.

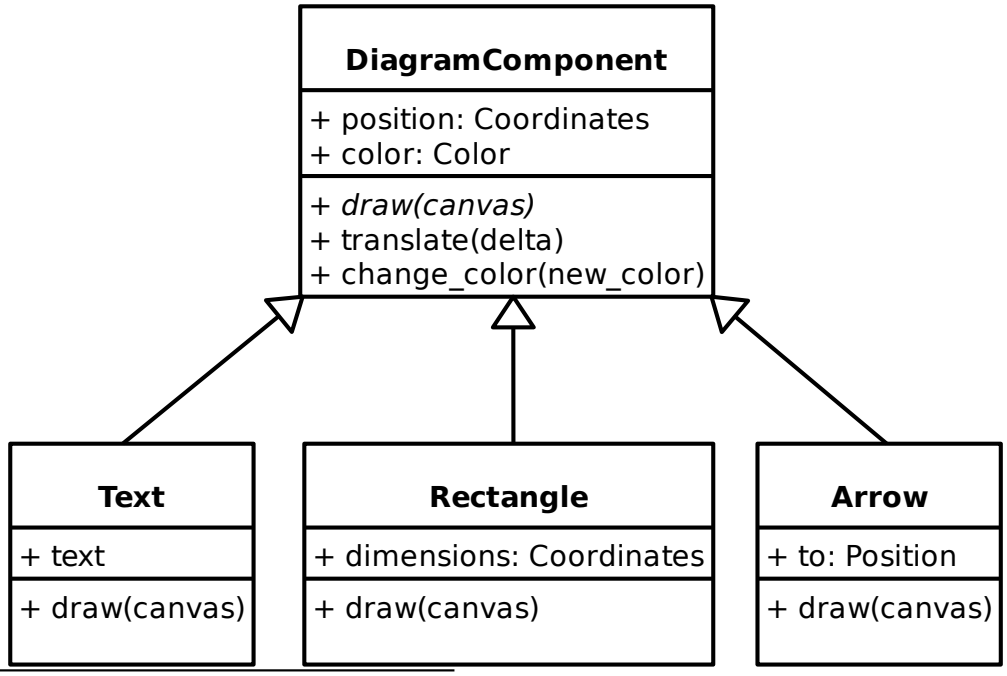
Overriding

- If the child class redefines a method of the base class, it *overrides* the implementation of the base class

```
class A:
    def print_class(self):
        print("A")

    def print_base(self):
        print("A")

class B(A):
    def print_class(self):
        print("B")

b = B()
b.print_class() # Prints: B
b.print_base()  # Prints: A
```

Overriding

- The `draw` method in the `DiagramComponent` class is an abstract method[7].
- An abstract method is a method that *must* be overridden by the child classes.



[7] Illustrated by italic function name in UML diagram.

Polymorphism

- Polymorphism is when a functions executes different code based the object types of its arguments

```
object = Rectangle(...)
object.draw(canvas) # Draws a rectangle
object = Text(...)
object.draw(canvas) # Draws text
object = Arrow(...)
object.draw(canvas) # Draws an arrow
```

- Python achieves polymorphism through *duck typing*.[8]

[8] "If it walks like a duck and it quacks like a duck, then it must be a duck."

Multiple inheritance

- *If you think you need multiple inheritance, you're probably wrong, but if you know you need it, you might be right.*[9]
- Python allows classes to inherit from multiple base classes:

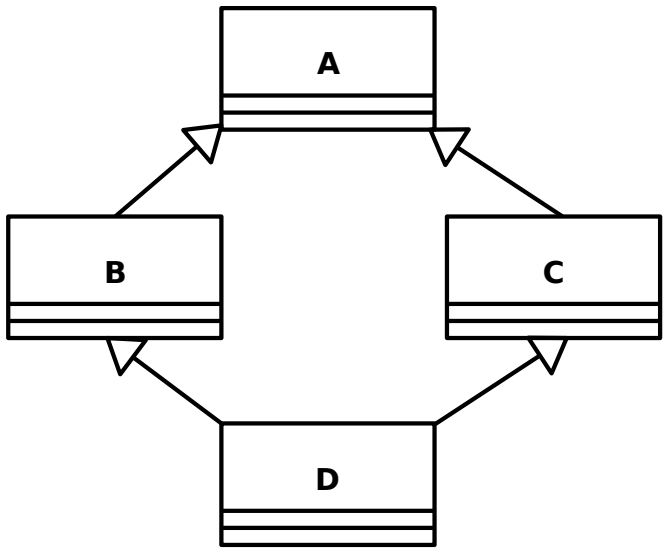
```
class A:
    def print_a(self):
        print("A")

class B:
    def print_b(self):
        print("A")

class C(A, B):
    pass

c = C()
c.print_a() # Prints "A"
c.print_b() # Prints "B"
```

[9] Phillips, Dusty. Python 3 object oriented programming.

The diamond problem

- Things get messy, when multiple base classes share a common ancestor

```
class A:
    def __init__(self):
        print("Initializing A ...")

class B(A):
    def __init__(self):
        super().__init__()
        print("Initializing B ...")

class C(A):
    def __init__(self):
        super().__init__()
        print("Initializing C ...")

class D(B, C):
    def __init__(self):
        super().__init__()
```



The diamond problem

- How can we know which `__init__` function is called?

```
d = D()
```

- Output:

```
Initializing A ...
Initializing C ...
Initializing B ...
```

- `super()` linearizes the class hierarchy and calls all functions in sequence, so this works.
- **But:** This becomes problematic when these functions take different parameters.

Mixin classes

- A mixin class is a super class that only implements functionality (no attributes) and can be easily added to a class.

```
class PrettyPrint:
    class pretty_print(self):
        print(f"~~ {self} ~~")

class A(PrettyPrint):
    def __str__(self):
        return "A"

a = A()
a.pretty_print() # Prints: ~~ A ~~
```

Exercise

- Exercise 2 in exercise notebook
- Time: 15 minutes

Summary

What we have learned:

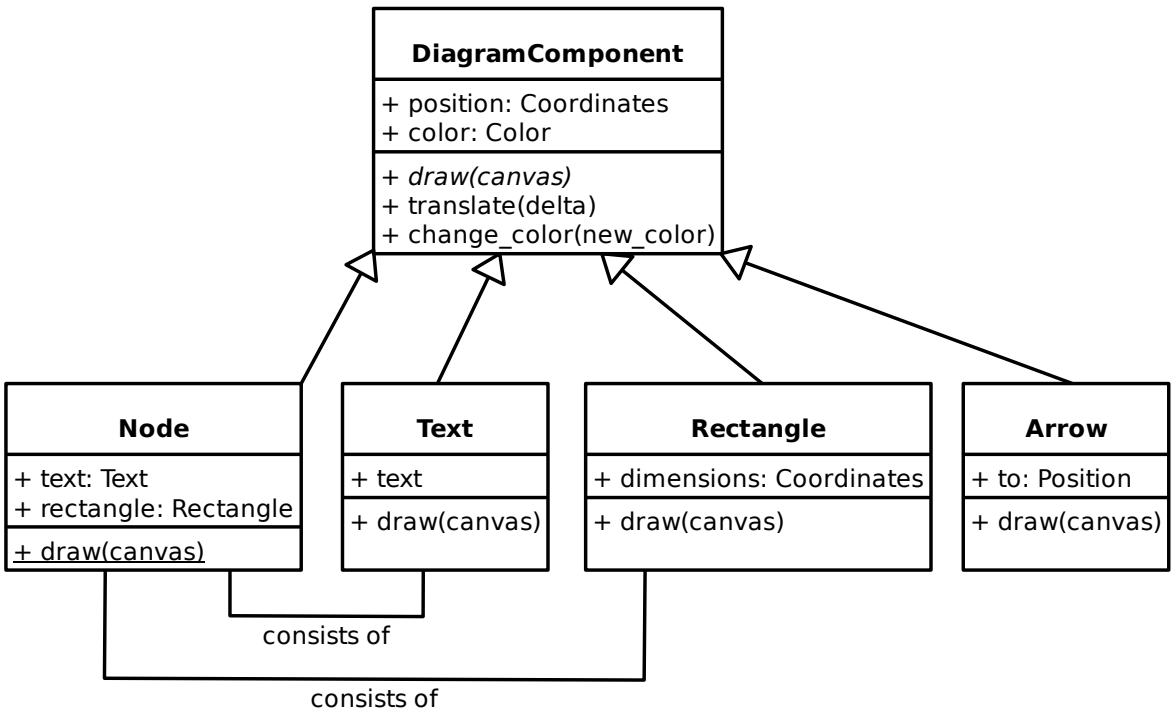
- Inheritance allows different objects to share common code
- How to achieve polymorphism using inheritance and overriding
- How to implement inheritance in Python
- The difficulties of multiple inheritance and when it is useful (Mixins)

Agenda

1. Overview
2. Object oriented thinking
3. Classes
4. Inheritance
5. **Aggregation and composition**

Composition

- A node *consists of* a rectangle and a text.
- Objects are in a composition relation when their lifetimes are dependent on each other.

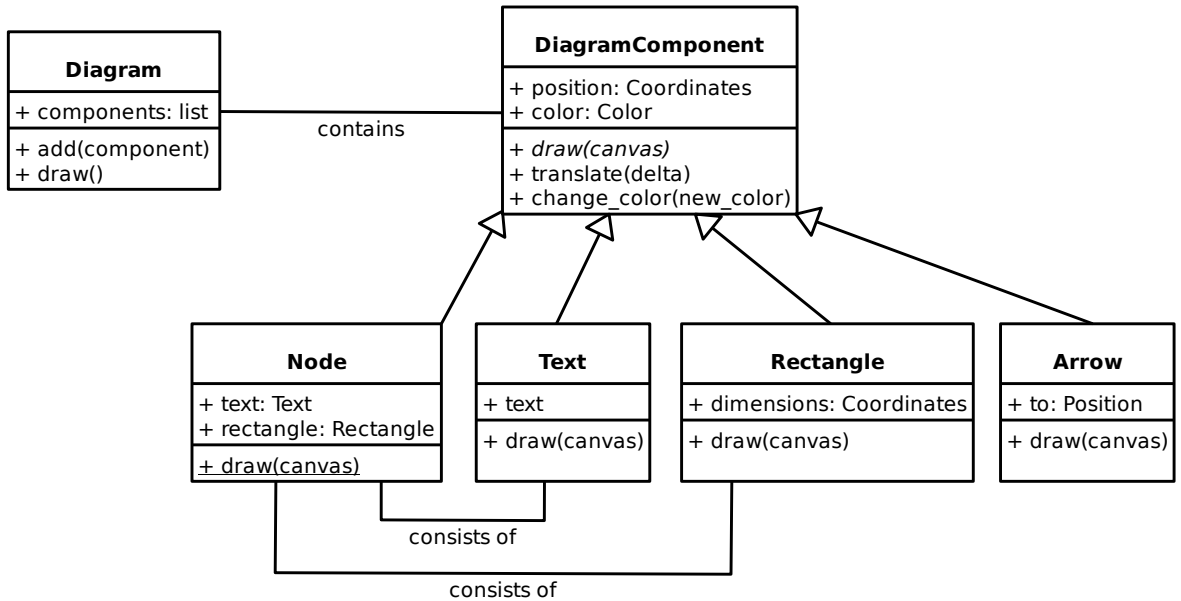


Composition as abstraction

- {Abstraction: Dealing with the level of detail that is most appropriate to a given task}
- Manually creating a node from a rectangle and text is complex and error prone:
  - Text must be placed correctly, both rectangle and text must be drawn on diagram.
- `Node` class hides a way information (the rectangle and text) to simplify creation of diagram nodes.

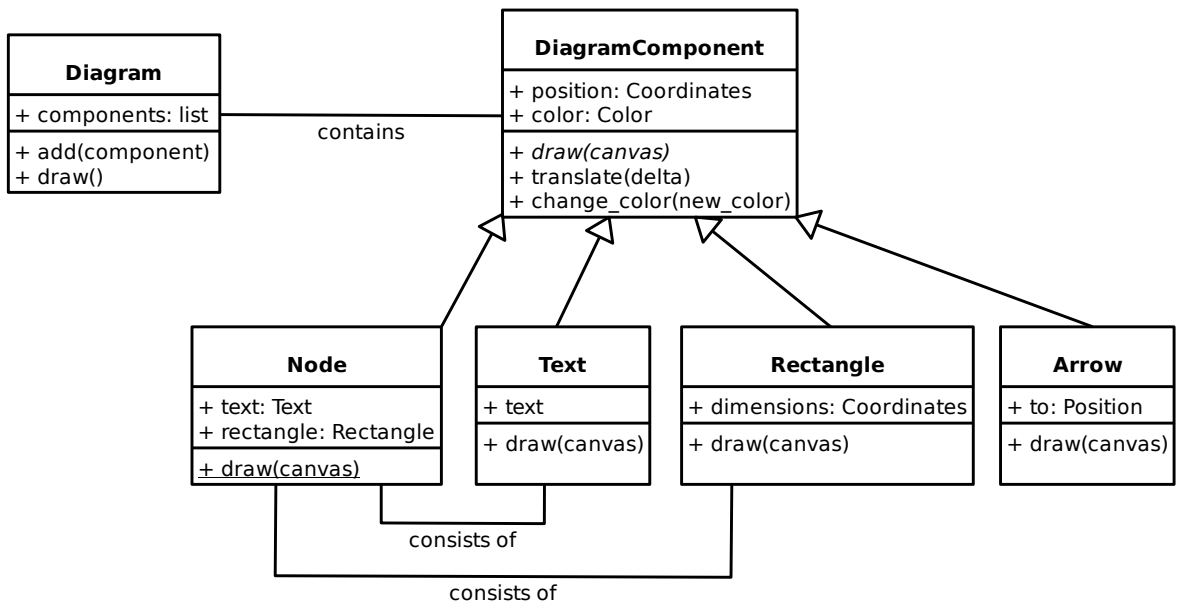
Finalizing the design

- The diagram class collects and draws the components of the diagram.



The aggregation relation

- Two objects are in an aggregation relation if one *contains* the other but when their lifetimes are independent
- Example: A given node may appear in multiple diagrams.
- Difference to composition is mostly formal.



Exercise

- Exercise 3 in exercise notebook
- Time: 15 minutes

Summary