# Scientific Software Development with Python

Object oriented programming — Part 2

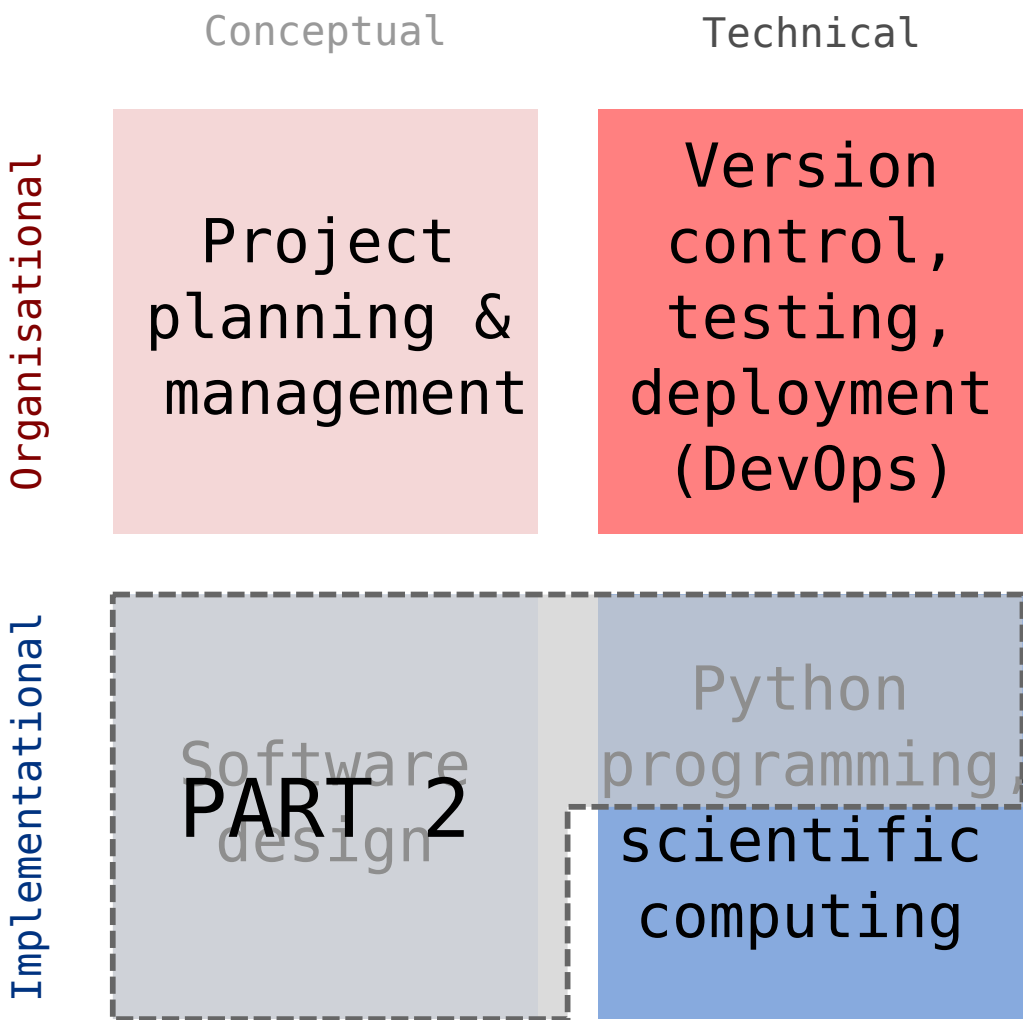

CHALMERS
UNIVERSITY OF TECHNOLOGY

Simon Pfreundschuh
Department of Space, Earth and Environment

# Agenda



1. Overview
2. Drawing diagrams revisited
3. Abstract base classes
4. Procedural vs. object oriented programming


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

## This lecture

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

## Object oriented design

- Modeling framework based on describing classes of objects and their interactions/relationships
- Relationships: Inheritance, aggregation, composition

## Benefits

- Avoiding code duplication (inheritance)
- Reducing complexity through abstraction (inheritance, composition)
- Modular code through encapsulation and interfaces (classes)

# Agenda



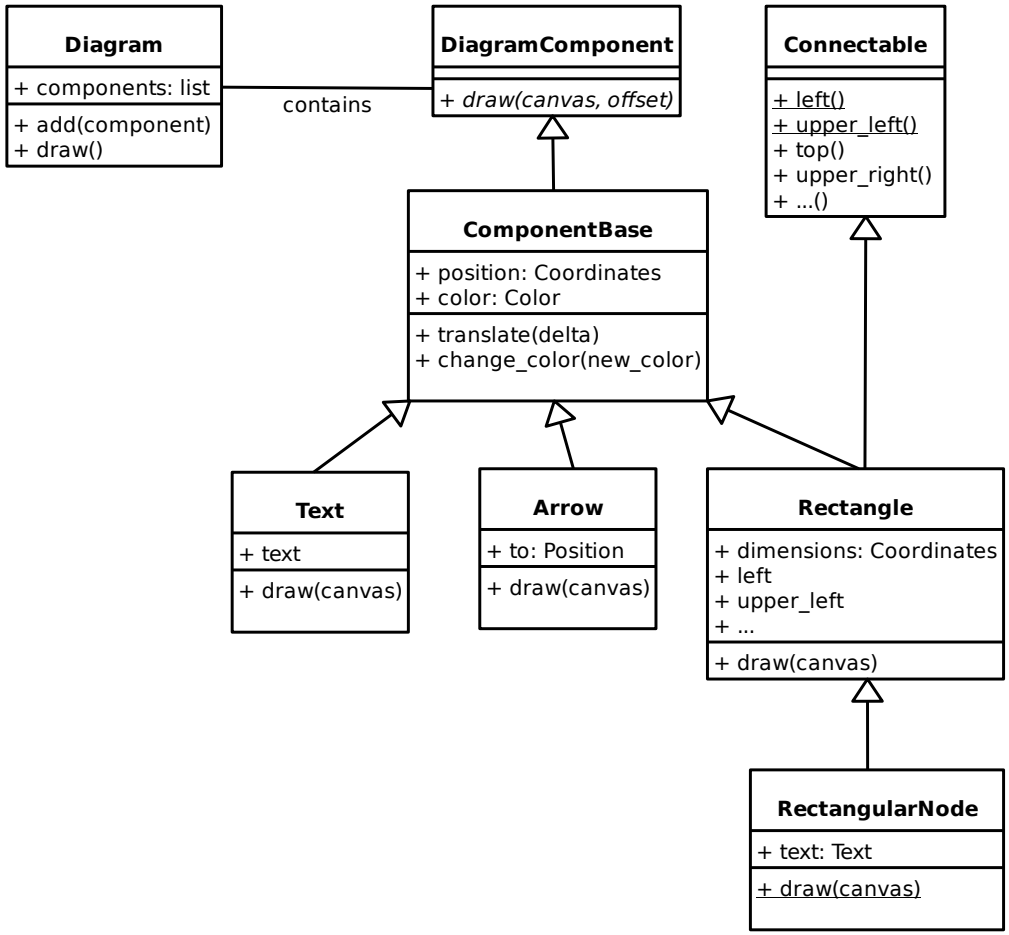
1. Overview

**2. Drawing diagrams revisited**

3. Abstract base classes

4. Procedural vs. object oriented programming

## Drawing diagrams



# Design improvements

1. Split `DiagramComponent` into two classes:
   - `DiagramComponent` now only specifies interface for `Diagram` class (interface)
   - `ComponentBase` contains basic properties of component classes but they are not required for the diagram interface.
2. `position` attribute of `DiagramComponent` class now represents a *relative position*
   - This makes the attribute meaning full for components that are part of other components
3. The `Node` class was renamed to `RectangularNode` and made a subclass of `Rectangle`.
4. Additional abstract base class `Connectable` for components that can be connected using arrows.

## The aims of object oriented design

- Handling complexity by breaking tasks down into different levels of abstraction[1]
  - Note how this is the basis of all technological progress
- A modular code base that allows for change, i.e. keep interdependencies to a minimum (*shy classes*)

## Problems of object oriented design

- If done badly, your code will be unnecessarily complex

[1] Although this can be achieved with functions alone.

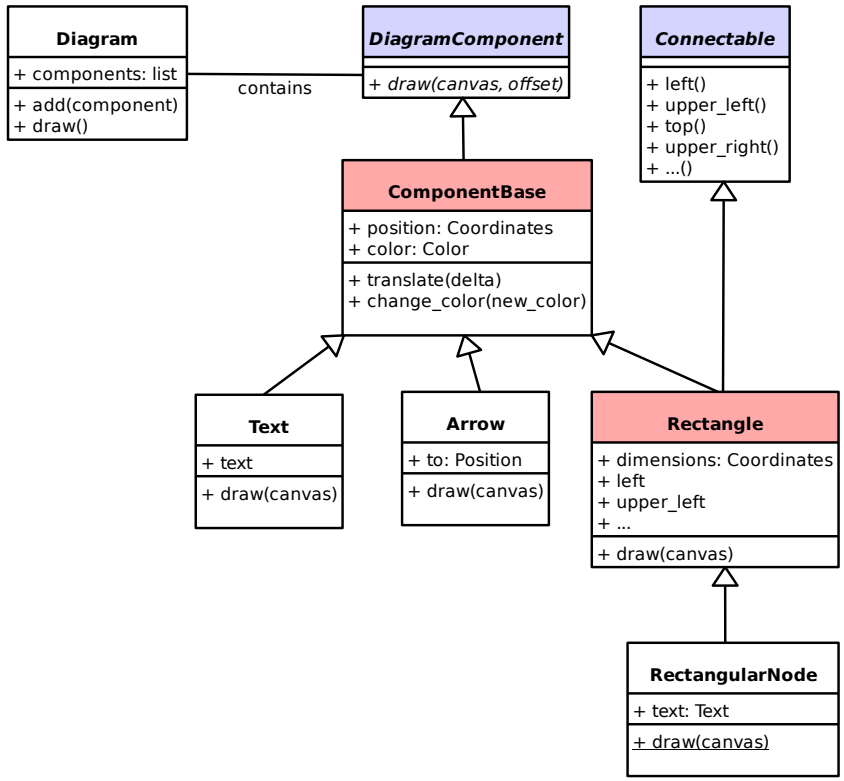
# Different roles of classes

## Defining interfaces (blue)

- Special case: abstract classes (classes that cannot be instantiated)
- Separation of concern between different parts of the code

## Abstraction (red)

- Encapsulation and information hiding: Expose only required information
- Break down tasks into different level of abstraction
- Code reuse (DRY-principle)



# Agenda



1. Overview

2. Drawing diagrams revisited

**3. Abstract base classes**

4. Procedural vs. object oriented programming

## The dangers of duck typing

- Python's dynamic type system allows you to pass any value as argument to a function.

```
diagram = Diagram()
node = RectangularNode((100, 100), (100, 100), "Node")
diagram.add(node)   # Work as expected.
diagram.add("node") # Works as well.
diagram.draw()      # Ohoh ...
...
```

## Solution

- Abstract base classes (ABC) allows specifying abstract class methods and properties, that a child class has to implement.
- Python will throw an error if a child class is instantiated that does not implement an abstract method.

```
from abc import ABC, abstractmethod

class DiagramComponent(ABC):
    """ ... """
    @abstractmethod
    def draw(self, canvas, offset=Coordinates(0, 0)):
        """ ... """
```

## Example

```
class A(DiagramComponent):
    def draw(self, *args, **kwargs):
        return 0

class B(Interface):
    pass

a = A() # Works fine.
b = B() # Error: Doesn't implement draw method.
```

## Advantages

- User functions can require a given interface by checking that an object inherits from the abstract base class using `isinstance`:

```
class Diagram:
  ...
  def add(self, component):
      ...
      if not isinstance(component, DiagramComponent):
          raise TypeError("The given component does not implement"
                          " the DiagramComponent interface.")
      self.components.append(component)
```

- ABCs serve as documentation for other developers who may want to extend your code.

## Some useful Python magic

```
from abc import ABC, abstractmethod

class DiagramComponent(ABC):
    ...

    @classmethod
    def __subclasshook__(cls, C):
        if cls is DiagramComponent:
            attributes = set(dir(C))
            if (set(cls.__abstractmethods__) <= attributes and
                set(cls.__abstractproperties__) <= attributes):
                return True
        return NotImplemented
```

# Exercise 1



- Exercise 1 on exercise sheet
- Time: 5 minutes

## Solution

- `@classmethod` decorator makes method callable on the `DiagramComponent` class with the `cls` parameter is set to the object's class
- `__subclasshook__` is used to determine whether a class is a subclass of the ABC
- `set(dir(C))` creates a (unique) set of the method and attribute names of the class `C`
- if statement checks whether abstract methods and properties of the ABC are subsets of those.

With this mechanism classes can fulfill the interface defined by the `DiagramComponent` ABC without explicitly inheriting from it.

## Example

```
class A(DiagramComponent):
    def draw(self, *args, **kwargs):
        return 0

class B:
    def draw(self, *args, **kwargs):
        return 0

print(isinstance(DiagramComponent, A()) # Prints: True
print(isinstance(DiagramComponent, B()) # Prints: True
```

# Agenda



1. Overview

2. Drawing diagrams revisited

3. Abstract base classes

**4. Procedural vs. object oriented programming**

# Exercise 2



- Exercise 2 on exercise sheet
- Time: 10 minutes

## Procedural programming

- Code organized as functions operating on data types
- Example languages: `C` and `Fortran`

```
from diagrams.procedural import create_node, draw
node_1 = create_node((50, 50), (100, 100), "node 1")
print(type(node_1)) # Prints: dict
draw(node_1)
```

## Procedural API

```
node_1 = create_node((50, 50), (100, 100), "node 1")
node_2 = create_node((300, 50), (100, 100), "node 2", "blue")
arrow = create_arrow(right(node_1),
                     left(node_2))

create_canvas(450, 200)
draw(node_1)
draw(node_2)
draw(arrow)
show()
```

## Object oriented API

```
from diagrams.object_oriented import (Diagram, Node, Arrow,
                                      Coordinates, Color)

node_1 = Node((50, 50), (100, 100), "Node 1", Color.Red())
node_2 = Node((200, 50), (100, 100), "Node 2", Color.Blue())
arrow = Arrow(node_1.right, node_2.left)

diagram = Diagram(350, 200)
diagram.add(node_1)
diagram.add(node_2)
diagram.add(arrow)
diagram.draw()
```

# User perspective

## Differences

- Usage is fairly similar
- Color handling in OO interface less error prone

## Similarities

- The procedural code *mimics* the object oriented code
- **Note:** Conceptual similarity between a function taking the object it acts upon as first argument and a class method taking `self` as first argument.

## OO programming and domain specific languages (DSL)

```
from diagrams.object_oriented import Color

red = Color.red()
blue = Color.blue()
purple = red + blue # Mix colors.
```

- Object oriented programming allows us to add new semantic layers to code
- With the right design our code essentially becomes an (embedded) domain specific language

## Procedural API

**diagram.py**

```
create_canvas(...)
draw(...)
show(...)
```

**components.py**

```
create_arrow(...)
create_node(...)
create_rectangle(...)
create_text(...)
```

**coordinates.py**

```
add_coordinates(...)
scale_coordinates(...)
left(...)
top_left(...)
top(...)
top_right(...)
right(...)
bottom_right(...)
bottom(...)
bottom_left(...)
```

# Code structure

- Since Python does not support overloading[2], the argument types need to be checked manually:

```
def draw(component):
    """
    Draw component on the current canvas.

    Args:
        component(``dict``): A dictionary representing the component
            to be drawn.
    """
    component_type = component["type"]
    if component_type == ComponentType.RECTANGLE:
        draw_rectangle(component)
    elif component_type == ComponentType.TEXT:
        draw_text(component)
    elif component_type == ComponentType.ARROW:
        draw_arrow(component)
    elif component_type == ComponentType.RECTANGULAR_NODE:
        draw_rectangular_node(component)
```

[2]Calling different function based on argument types.

## Object oriented API

**diagram.py**

```
class DiagramComponent
class Diagram
```

**components.py**

```
class Connectable
class ComponentBase
class Arrow
class Rectangle
class Text
class RectangularNode
```

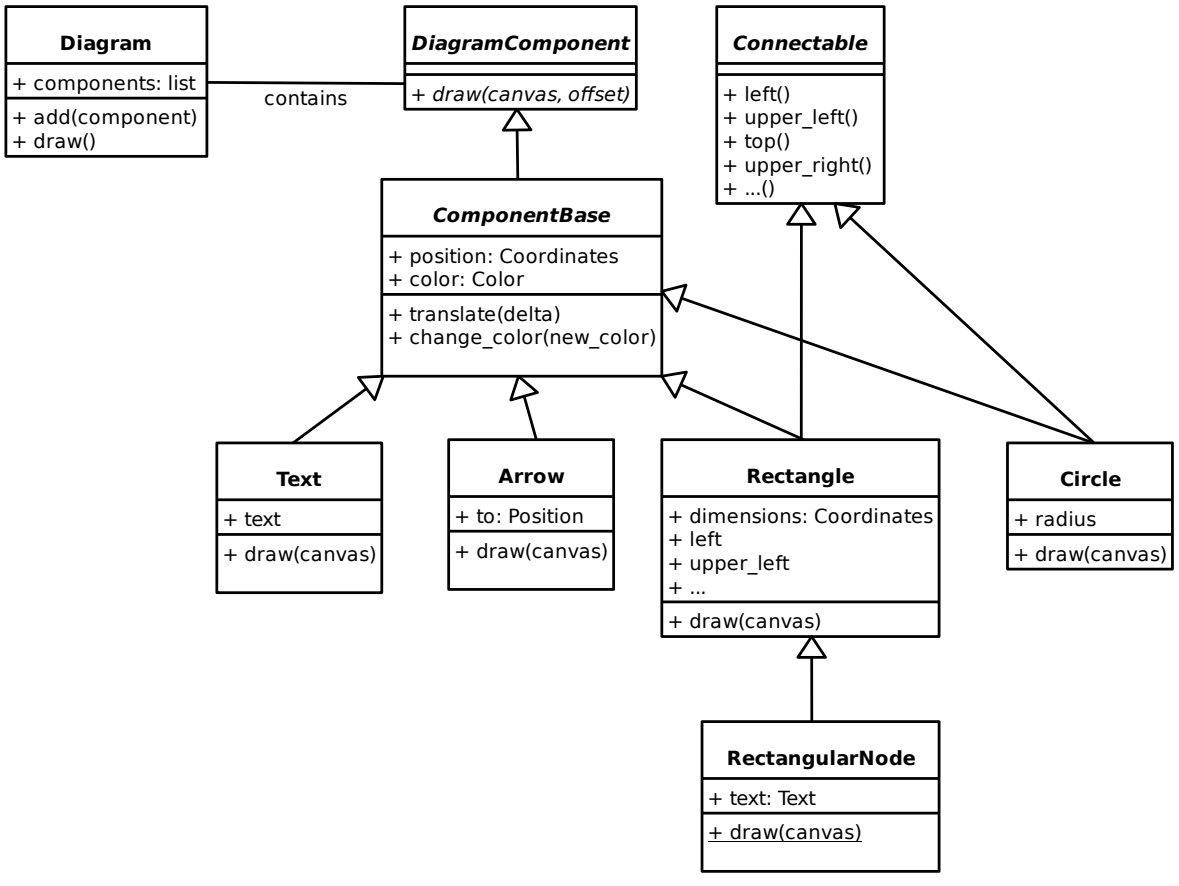
**coordinates.py**

```
class Coordinates
```

**color.py**

```
class Color
```

# Adding a new diagram component



# Exercise 3



- Exercise 3 on exercise sheet
- Time: 20 minutes

## Object oriented API

- Required changes:
  - Add new class to `diagrams/object_oriented/components.py` (green)

**diagram.py**

```
class DiagramComponent
class Diagram
```

**components.py**

```
class Connectable
class ComponentBase
class Arrow
class Rectangle
class Text
class RectangularNode
class Circle
```

**coordinates.py**

```
class Coordinates
```

**color.py**

```
class Color
```

# Exercise 4



- Exercise 5 on exercise sheet
- Time: 20 minutes

## Procedural API

- Required changes:
  - Add new function to create circle
  - Additional changes in 9 functions. `draw` and the 8 anchor functions.

**diagram.py**

```
create_canvas(...)
draw(...)
show(...)
```

**components.py**

```
create_arrow(...)
create_node(...)
create_rectangle(...)
create_text(...)
create_circle(...)
```

**coordinates.py**

```
add_coordinates(...)
scale_coordinates(...)
left(...)
top_left(...)
top(...)
top_right(...)
right(...)
bottom_right(...)
bottom(...)
bottom_left(...)
```

**OO vs. procedural:**

- Substantially less complex changes required in OO design.
- **But**: This depends on the kind of change. Defining a new function on diagram components is easier in the procedural paradigm.
- In general: OO design makes extending existing functionality easy. Procedural design makes adding new functionality easy.

**What we have learned today:**

- Using ABCs in Python to define generic interfaces
- Advantages of object oriented design:
  - More expressive code (DSL)
  - Keep it DRY, keep it shy: Avoiding code duplication and interdependencies leads to modular code that can be easily changed
- Procedural vs. object oriented:
  - Conceptual similarity between both approaches
  - OO design makes it easy to extend existing functionality
  - Procedural design makes it easy to add new functionality

## Design is hard

- It's not always black and white: Not everything must always be a class. Not everything must always be a function.
- The best design is of course the one that works for you
- **But**: Good design usually pays of in the long run