Classes and Class Diagrams
A **class** is similar to an entity. It describes compound data that consists of subsidiary data (called **attributes**) collected in an **instance** of the class. Additionally, it can describe **operations** on that data (later).
Example for simple class: Tea

Class description for Tea

A tea shop describes a particular brand of tea in stock by its name; a description of its color, flavor, etc; the weight in stock (in g); and its price in cent per kg.
Example for simple class: Tea

Class description for Tea

A tea shop describes a particular brand of tea in stock by its name; a description of its color, flavor, etc; the weight in stock (in g); and its price in cent per kg.

Class diagram for Tea
A class diagram can be mapped line-by-line to (Python) code.

Class declaration

```python
>>> class Tea:
...     def __init__(self, name, desc, wgt, price):
...         self.name = name
...         self.description = desc
...         self.weight = wgt
...         self.price = price
...
```

- `__init__` is a function that is called, when a new `Tea` instance is created. The `self` parameter is the new instance, `name`, `desc`, `wgt`, and `price` are used to initialize the respective attributes as shown.
Using simple classes

Creating and examining tea

```python
>>> earl_grey = Tea("Earl Grey",
                "Flavored black tea",
                10000, 4335)

>>> earl_grey
<__main__.Tea instance at 0x1051dd950>

>>> earl_grey.name  # get name attribute
'Earl Grey'

>>> earl_grey.price  # get price attribute
4335
```

- `Tea()` creates a new `Tea` instance and calls its `__init__` method
- Access attributes using `instance.attribute`
Extended class description for Tea

A tea shop describes a particular brand of tea in stock by its name; a description of its color, flavor, etc; the weight in stock (in g); and its price in cent per kg. The shop wants to determine the stock value. It also wants to be able to print an inventory line.
**Simple class with operation**

**Extended class description for Tea**

A tea shop describes a particular brand of tea in stock by its name; a description of its color, flavor, etc; the weight in stock (in g); and its price in cent per kg. The shop wants to determine the stock value. It also wants to be able to print an inventory line.

**Two operations**

- **stockPrice()**: no parameters, return total value of the tea brand in stock
- **inventoryLine()**: no parameters, return a string for printing the tea as an inventory item
The implementation of stockPrice and inventoryLine belongs to the class declaration.

Their first parameter is `self` and they can access all attributes.
Revised class declaration

class Tea:
    # __init__ omitted (same as before)
    def stockPrice(self):
        return self.weight * self.price / 1000
    def inventoryLine(self):
        return (self.name + '. ' +
                self.description + '. ' +
                str(self.weight) + 'g. ' +
                str(self.price) + ' c/kg.')

Remarks

- `str()` converts a number to a string
A reading of a metering device consists of a **reading date** and a **reading value**.
A reading of a metering device consists of a reading date and a reading value.

```
Reading

date: datetime.date
value: float

difference(previous: reading): float
yearly_prediction(previous: reading): float
```
Explanation

- **datetime** is a module that contains utilities for manipulating dates
- made available using
  ```python
  import datetime
  ```
import datetime

class Reading:
    def __init__(self, date, value):
        self.date = date  # datetime.date
        self.value = value  # float
    def difference(self, previous):
        return self.value - previous.value
    def yearly_prediction(self, previous):
        value_diff = self.value - previous.value
        date_diff = self.date - previous.date
        factor = 365.25 / date_diff.days
        return value_diff * factor
### Compound Classes

**Household**

A household has an allocated amount of space (in square meters) and a number of occupants. Furthermore, a household has meter readings for several dates in the past.
Compound Classes

Household

A household has an allocated amount of space (in square meters) and a number of occupants. Furthermore, a household has meter readings for several dates in the past.

Class diagram
The connection between Household and Reading in the class diagram is an **association**.

It comes with a direction (arrow) that indicates the direction in which it can be traversed.

We (choose to) represent the association with a list of readings stored in the Household instance.

Requires a “housekeeping” method to add new readings.
class Household:
    def __init__(self, space, occupants):
        self.space = space
        self.occupants = occupants
        self.readings = []
    def add_reading(self, reading):
        self.readings = [reading] + self.readings
Further Household Methods

Requirements

For a household, we want to be able to determine the number of readings taken. If there are multiple readings, we want to give a statistical yearly prediction.

Implementation

class Household:  # __init__ ... as before
    def nr_readings(self):
        return len(self.readings)
    def yearly_average(self):
        if len(self.readings) < 2:
            return None  # more than one reading
        first_reading = self.readings[-1]
        last_reading = self.readings[0]
        return last_reading.yearly_prediction(first_reading)
Data Modeling II
Data Modeling II

- Union
- Abstraction
- Inheritance
## Task

A drawing program wants to manage different geometric shapes in a coordinate system. Initially, there are three kinds of figures:

- squares with reference point upper left and given side length
- circles with reference point in the middle and a given radius
- points that just consist of the reference point
Task

A drawing program wants to manage different geometric shapes in a coordinate system. Initially, there are three kinds of figures:

- squares with reference point upper left and given side length
- circles with reference point in the middle and a given radius
- points that just consist of the reference point

Approach

- Each kind of figure can be represented by a compound class. The reference point is a separate Point object.
- In many languages, they could not be used together, but no problem in Python
Union of classes

UML class diagram

Square
side: float

Circle
Radius: float

Dot

Point
x: float
y: float
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Square:
    def __init__(self, ref, side):
        self.ref = ref
        self.side = side

    and so on
For each shape, we want to be able to compute the area and we want to move it around.
Functionality for shapes

Task

For each shape, we want to be able to compute the area and we want to move it around.

UML diagram
Python implementation

Square

def area(self):
    return self.side * self.side

def move(self, dx, dy):
    self.ref.move(dx, dy)
Python implementation

Square

```python
def area(self):
    return self.side * self.side

def move(self, dx, dy):
    self.ref.move(dx, dy)
```

Circle

```python
def area(self):
    return 2 * math.pi * self.radius

def move(self, dx, dy):
    self.ref.move(dx, dy)
```
Square

```python
def area(self):
    return self.side * self.side

def move(self, dx, dy):
    self.ref.move(dx, dy)
```

Circle

```python
def area(self):
    return 2 * math.pi * self.radius

def move(self, dx, dy):
    self.ref.move(dx, dy)
```

Dot ...
All implementations assume a `move` method in `Point`.

```python
Point

def move(self, dx, dy):
    self.x += dx
    self.y += dy
```

Observation: the move methods in `Square`, `Circle`, and `Dot` are all identical; it would be nice to be able to advertise that all shape classes have methods `move` and `area`.
Python implementation II

- All implementations assume a `move` method in `Point`.

**Point**

```python
def move(self, dx, dy):
    self.x += dx
    self.y += dy
```

**Observation**

- the `move` methods in `Square`, `Circle`, and `Dot` are all identical
- it would be nice to be able to advertise that all shape classes have methods `move` and `area`. 
Abstraction in programming

- identify programming patterns
  repeated program fragments with similar semantics

- generalization
  replace specific parts by variables

- extraction
  give a name to the thus generalized program fragment
  invoke in the original places
Abstraction

Abstraction in programming

- identify programming patterns
  repeated program fragments with similar semantics
- generalization
  replace specific parts by variables
- extraction
  give a name to the thus generalized program fragment
  invoke in the original places

What does that mean?

- generally avoid duplication
- look for similarities
- try to solve each problem only once
Similarity among classes

Goal

- identify similar field and method declarations
Similarity among classes

Goal

- identify similar field and method declarations
- example: Square.move, Circle.move, Dot.move
Goal

- identify similar field and method declarations
- example: `Square.move`, `Circle.move`, `Dot.move`
- approach: introduce common **super class** `Shape`
Goal

- identify similar field and method declarations
- example: Square.move, Circle.move, Dot.move
- approach: introduce common super class Shape
- indicated by arrow with open triangle head
Inheritance

UML diagram: shapes

Inheritance

UML diagram: shapes

Shape

area() : float
move(dx, dy: float)

Square

side: float
area(): float

Circle

radius: float
area(): float

Dot

area(): float

Point

x: float
y: float
**Inheritance**

**UML diagram: shapes**

- *Shape* is an **abstract class**: no instances
- *Shape.area()* is an **abstract method**: no implementation
Inheritance in Python

Super class Shape

class Shape:
    def __init__(self, ref):
        self.ref = ref
    def move(self, dx, dy):
        self.ref.move(dx, dy)
    def area(self):
        return 0

- it’s not easily possible to define proper abstract classes in Python (you can create Shape instances)
- it’s not possible to define abstract methods in Python; the way to do it would be to drop the definition of area()
Subclasses in Python

Square

class Square (Shape):
    def __init__(self, ref, side):
        Shape.__init__(self, ref)
        self.side = side
    def area(self):
        return self.side * self.side
Subclasses in Python

Square

```python
class Square (Shape):
    def __init__ (self, ref, side):
        Shape.__init__(self, ref)
        self.side = side
    def area(self):
        return self.side * self.side
```

Notes

- call `__init__` method of the super class `Shape`
- no need to define `move()`, its definition is inherited from `Shape`
- override `Shape`'s definition of `area()`
Weather data

We want to keep track of various recordings of weather data all comprising of a high and a low reading. Two examples are temperature and pressure readings. All should be printable.
Exploiting inheritance

Weather data

We want to keep track of various recordings of weather data all comprising of a high and a low reading. Two examples are temperature and pressure readings. All should be printable.

Consider this class diagram
If a Python object has a method `__str__`, then that method is used to convert the object to a string.
Implementing weather data

If a Python object has a method `__str__`, then that method is used to convert the object to a string.

```python
class Recording:
    def __init__(self, low, high):
        self.low = low
        self.high = high
    def __str__(self):
        return (str(self.low) + ' - ' +
                str(self.high) + ' ' +
                self.unit())
```
Printable Temperature recording

Temperature/Pressure can inherit printing from Recording, but it has to define the `unit()` method to make printing work!
Printable Temperature recording

Temperature/Pressure can inherit printing from Recording, but it has to define the `unit()` method to make printing work!

Implementing concrete recordings

class Temperature (Recording):
    def unit():
        return "degrees"

class Pressure (Recording):
    def unit():
        return "hPa"
End Part II