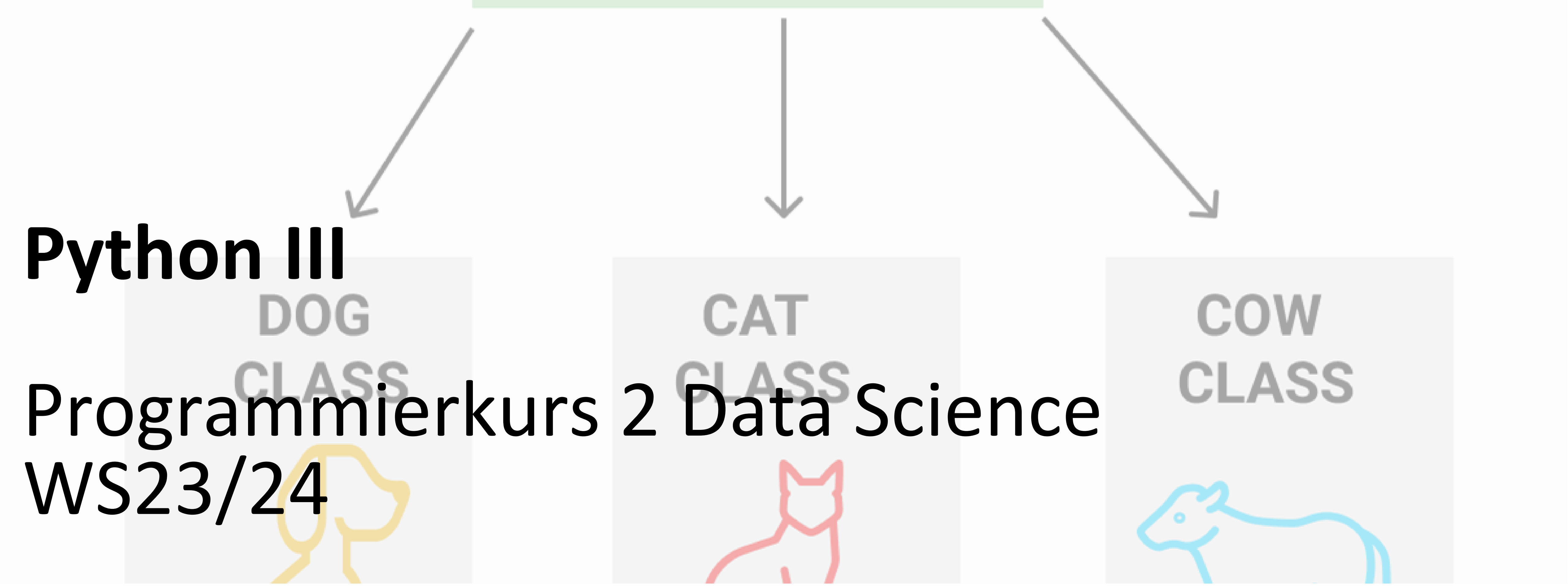


Python III

Programmierkurs 2 Data Science
WS23/24

Leonard Traeger
M. Sc. Information Systems
leonard.traeger@fh-dortmund.de

Learning Goals Python III

- **Explain** the imperative and declarative programming paradigm and **classify** Python and SQL as a programming language.
- **Explain** Python in the context of object-oriented programming and **link** its concepts to a relational table.
- **List** advantages for object-oriented programming.
- **Demonstrate** deploying a class with a constructor, destructor, decorator annotated and regular class methods, and inheritance given simple examples (e.g., sport players and clubs).

Programming Paradigms

IMPERATIVE

…programmer instructs the machine **how to change** its state

DECLARATIVE

…programmer (merely) **declares what** the desired **result** should look like, but not how to compute it

Many programming languages allow both in combination.

Distinct approaches exist under imperative programming umbrella :

- Procedural: linear series of instructions
- Functional: move from one function to another
- **Object-oriented**: groups instructions with the part of the state they operate on!

Programming Paradigms (cont.)

```
scoredPoints = [55, 12, 66, 1, 99];

#imperative: programmer instructs the machine how to change its state
scoredPointsLessThan50 = []
for i in range(len(scoredPoints)):
  if(scoredPoints[i] < 50):
    scoredPointsLessThan50.append(scoredPoints[i])
print(scoredPointsLessThan50)

#declarative: programmer merely declares properties of the desired result,
#             but not how to compute it
print([x for x in scoredPoints if x < 50])
```

SQL: `SELECT * FROM scoredPoints WHERE scoredPoints < 50;`

Object-oriented programming (OOP)

- **Everything in Python** is an **object** and many **details** are **hidden** away.
- Objects are instances of a class with internal data and (usually) associated methods.
- **Methods** are **functions** that **belong to objects** and have access to associated data.

Object

- An Object is a bit of **self-contained Code and Data.**
- A key aspect of the Object approach is to **break the problem into smaller understandable parts** (**divide and conquer**).
- **Objects** have **boundaries** that allow us to **ignore unnecessary detail**.
- We have been using objects all along: String Objects, Integer Objects, Dictionary Objects, List Objects...
- Object Lifecycle: **created, used, and discarded**.

Why OOP?

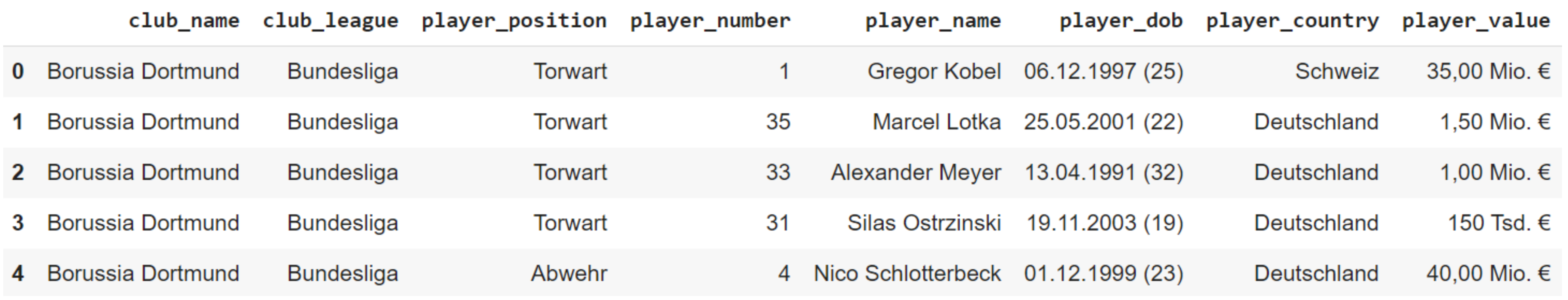
- **Encapsulation:** bundle code into a single unit and scope definitions of each data piece.
  - Each object has natural boundaries.
  - Each object is on its own a program.
- **Abstraction:** use classes to generalize your object types.
  - Simplifying your program.
  - Outsource referencing into logical pieces.
- **Inheritance:** inherit attributes and behaviors from another class.
  - Reuse code.
- **Polymorphism:** create many objects from one class.
  - Same flexible piece of code serves many use cases.

...to define some domain

Extract Transform Load

Definitions

- **Class** : A template
- **Method (or Message or object-restricted function)** : A defined capability of a class
- **Field (or Attribute or Column)** : A bit of data in a class
- **Object (or Instance or Row-Value)** : A particular instance of a class



	club_name	club_league	player_position	player_number	player_name	player_dob	player_country	player_value
0	Borussia Dortmund	Bundesliga	Torwart	1	Gregor Kobel	06.12.1997 (25)	Schweiz	35,00 Mio. €
1	Borussia Dortmund	Bundesliga	Torwart	35	Marcel Lotka	25.05.2001 (22)	Deutschland	1,50 Mio. €
2	Borussia Dortmund	Bundesliga	Torwart	33	Alexander Meyer	13.04.1991 (32)	Deutschland	1,00 Mio. €
3	Borussia Dortmund	Bundesliga	Torwart	31	Silas Ostrzinski	19.11.2003 (19)	Deutschland	150 Tsd. €
4	Borussia Dortmund	Bundesliga	Abwehr	4	Nico Schlotterbeck	01.12.1999 (23)	Deutschland	40,00 Mio. €

Variable and Datatypes (Recap Python I)

Assignment uses dynamic referencing.

- The type/class is determined from the value, not declared.
- Type/class information belongs to the data, not the name bound to that data.

```
x = 10000
```

- x is not just a “raw” integer.
- x is a pointer to a compound C structure, which contains several values.
- Dynamic referencing in Python is more flexible but also more time and space consuming than compared to raw C.

Object and Methods

```
# In Python, we have several classes, e.g. string, float, int, list, etc
# By writing ... we actually create an instance.
club_name = 'Borussia Dortmund'
type(club_name)
```

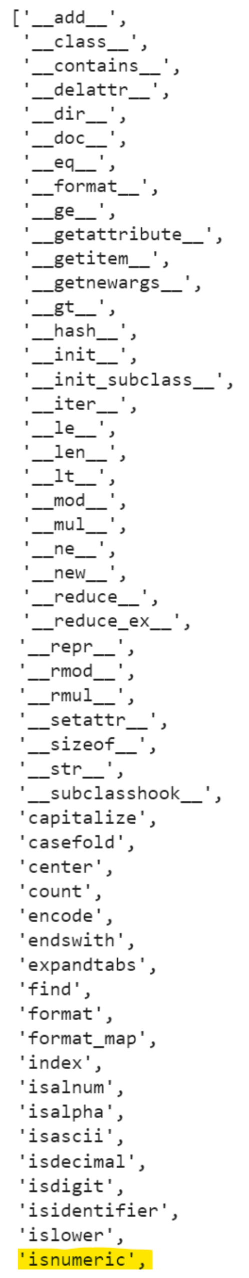
```
str
```

- Method calls take the form `object.method()`

```
# We can learn the available functions e.g. isnumeric()
dir(club_name)
# the ones starting and ending with "__" are "internal use"
```

```
print(club_name.isnumeric()) #Prints "False"
print(club_name.title()) #Prints "'Borussia Dortmund'"
```

```
False
Borussia Dortmund
```

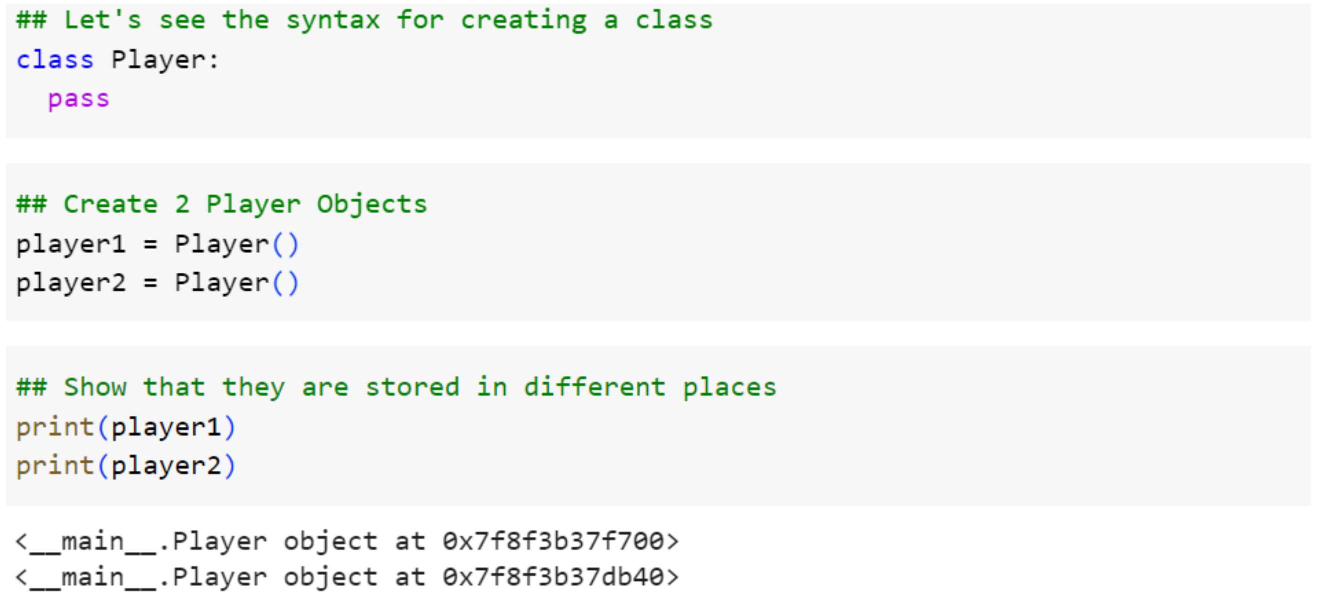


```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
```

Technology Arts Sciences TH Köln

Classes

- We can create **lots** of **objects instanced** from the **same class** (template for the object).
- We can **store each distinct object** in its **own** variable.
- Each instance has its own copy of the instance variables (and we change those variables separately for each instance).



```python
## Let's see the syntax for creating a class
class Player:
  pass
```

```python
## Create 2 Player Objects
player1 = Player()
player2 = Player()
```

```python
## Show that they are stored in different places
print(player1)
print(player2)
```

```
<__main__.Player object at 0x7f8f3b37f700>
<__main__.Player object at 0x7f8f3b37db40>
```

Classes (cont.)

```
## Each player will have a name
player1.player_name = 'Gregor Kobel'
player2.player_name = 'Marcel Lotka'
## let's see whether the player names were assigned
print(player1.player_name)
print(player2.player_name)
```

```
Gregor Kobel
Marcel Lotka
```

```
# let's create another player object
player3 = Player()
### This will fail because ..
player3.player_name
```

Technology
Arts Sciences
TH Köln

Constructor

- The constructor is a method that is **called** when an **object** is **initialized**.
- Every class has a constructor, but its **not required** to **explicitly** define it.
- Constructors are **used a lot**: define `__init__(self)`

```
## let's add player_number, player_name, player_value, club_name
class Player:
  ## We use a constructor method to create a player
  def __init__(self, player_number, player_name,
               player_value, club_name="Borussia Dortmund"):
    self.club_name = club_name
    self.player_number = player_number
    self.player_name = player_name
    self.player_value = player_value
```

Parameterize constructor arguments **with default in** the **end**

Constructor (cont.)

```python
## let's define player1, player2, player3
player1 = Player(player_number = 1, player_name = 'Gregor Kobel',
                 player_value = "35,00 Mio. €")

player2 = Player(player_number = 35, player_name = 'Silas Ostrzinski',
                 player_value = "150 Tsd. €")

player3 = Player(player_number = 1, player_name = 'Marc-André ter Stegen',
                 player_value = "35,00 Mio. €", club_name="FC Barcelona")
```

Primary **purpose** of **defaulter**: to have **initial values** when the object is created to then run **without subsequent error messages**.

```python
print(player1.club_name) #Prints ???
print(player2.club_name) #Prints ???
print(player3.club_name) #Prints "FC Barcelona"
```

Methods

```python
## let's create a method that uniforms player values
class Player:
  def __init__(self, player_number, player_name,
               player_value, club_name="Borussia Dortmund"):
    self.club_name = club_name
    self.player_number = player_number
    self.player_name = player_name
    self.player_value = player_value

  def get_player_value_numeric(self):
    if(type(self.player_value) == float):
      return self.player_value
    else:
      player_value_arr = self.player_value.split(" ")
      #"150 Tsd. €".split(" ") --> ['150,00', 'Tsd.', '€']
      value = player_value_arr[0].replace(",",".")
      unit = 1000 if "Tsd." in player_value_arr[1] else 1000000
      return float(float(value)*unit)

  def saved_goal(self):
    self.player_value = 50000 + self.get_player_value_numeric()
    return self
```

Such methods are day to day jobs of data scientists

Extract
Transform
Load

Methods (cont.)

```python
## let's define player1, player2, player3
player1 = Player(player_number = 1, player_name = 'Gregor Kobel',
                 player_value = "35,00 Mio. €")

player2 = Player(player_number = 35, player_name = 'Silas Ostrzinski',
                 player_value = "150 Tsd. €")

player3 = Player(player_number = 1, player_name = 'Marc-André ter Stegen',
                 player_value = "35,00 Mio. €", club_name="FC Barcelona")
```

```python
print(player1.get_player_value_numeric())          #Prints "35000000.0"
print(player2.get_player_value_numeric())          #Prints "150000.0"
print(player3.get_player_value_numeric())          #Prints "35000000.0"

print(player1.saved_goal().player_value)           #Prints "35050000.0"
print(player2.saved_goal().saved_goal().player_value) #Prints ???
print(player3.player_value)                        #Prints ???
```

Technology
Arts Sciences
TH Köln

Methods versus	**Functions**
Method definitions are always present inside a class.	We don’t need a class to define a function.
Methods are associated with the objects of the class they belong to.	Functions are not associated with any object.
A method is called ‘on’ an object. We cannot invoke it just by its name	We can invoke a function just by its name.
Methods can operate on the data of the object they associate with	Functions operate on the data you pass to them as arguments.
Methods are dependent on the class they belong to.	Functions are independent entities in a program.
A method requires to have ‘self’ as its first argument.	Functions do not require any ‘self’ argument. They can have zero or more arguments.

#Training 1

1. Create a class club with a constructor and the attributes club_name, club_league, club_stadium, and club_stadium_seats (default=100).
2. Add two methods to the class
   1. `in_construction` which reduces the number of seats by 10%
   2. `renovated` which adds seats passed through an input argument.
3. Instantiate three or more stadiums of your personal choice (any sports), or make some up.
   - You can find real stadiums here: https://www.transfermarkt.co/25-stadiums-from-25-leagues-the-smallest-first-division-stadiums-in-europe/index/galerie/10182
   - Test each of your methods.
   - Store all clubs in a suitable container (e.g., list, set, tuple see Python II lecture).
4. Loop through the container and simulate all stadiums being renovated adding 20% more seats. Print all new club information.

Inheritence

#protected
`_attr/method()`
#private
`__attr/method()`

- Reuse an existing class and inherit all the capabilities of an existing class and then add our own little bit.
- Another form of store and reuse: **write once - reuse many times**.
- The new class (child) has all the capabilities of the old class (parent).

Technology
Arts Sciences
TH Köln

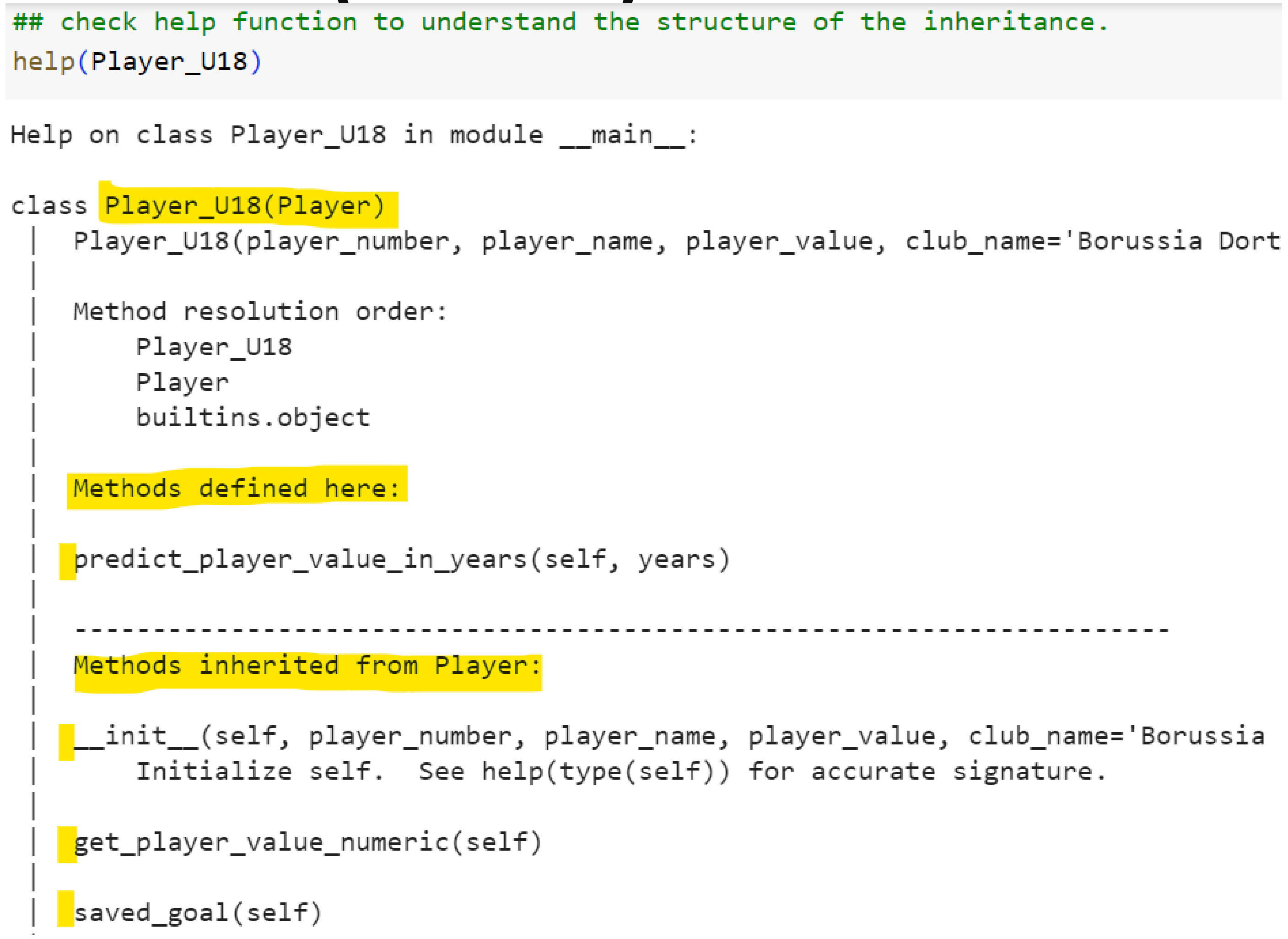
Inheritence (cont.)

```
class Player_U18(Player):
  pass
  def predict_player_value_in_years(self, years):
    return self.get_player_value_numeric() * ((18+years) / 18)
```

```
player4 = Player_U18(player_number = 16, player_name = 'Julien Duranville',
                     player_value = "8,50 Mio. €")
```

```
print(player4.club_name)                         #Prints "Borussia Dortmund"
print(player4.player_number)                     #Prints "16"
print(player4.player_name)                       #Prints "Julien Duranville"
print(player4.player_value)                      #Prints "8,50 Mio. €"
print(player4.get_player_value_numeric())        #Prints "8500000.0"
print(player4.predict_player_value_in_years(5)) #Prints "10861111.11"
```

Inheritence (cont.)



```
## check help function to understand the structure of the inheritance.
help(Player_U18)
```

```
Help on class Player_U18 in module __main__:

class Player_U18(Player)
 |  Player_U18(player_number, player_name, player_value, club_name='Borussia Dort
 |
 |  Method resolution order:
 |      Player_U18
 |      Player
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  predict_player_value_in_years(self, years)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Player:
 |
 |  __init__(self, player_number, player_name, player_value, club_name='Borussia
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  get_player_value_numeric(self)
 |
 |  saved_goal(self)
```

Overwrite Parent Method

#access to parent attr/methods
`super()`

```
class Player_U18(Player):
  pass
  def get_player_value_in_years(self, years):
    return self.get_player_value_numeric() * ((18+years) / 18)

  def saved_goal(self): #instead 50000
    self.player_value = 100000 + self.get_player_value_numeric()
    super().saved_goal()
    return self
```

```
player4 = Player_U18(player_number = 16, player_name = 'Julien Duranville',
                     player_value = "8,50 Mio. €")
```

```
print(player4.saved_goal().player_value) #Print ???
```

Decorator @property

- Used to convert the method access to an attribute access, without changing the interface of the class.
- Constrained to zero-argument methods.

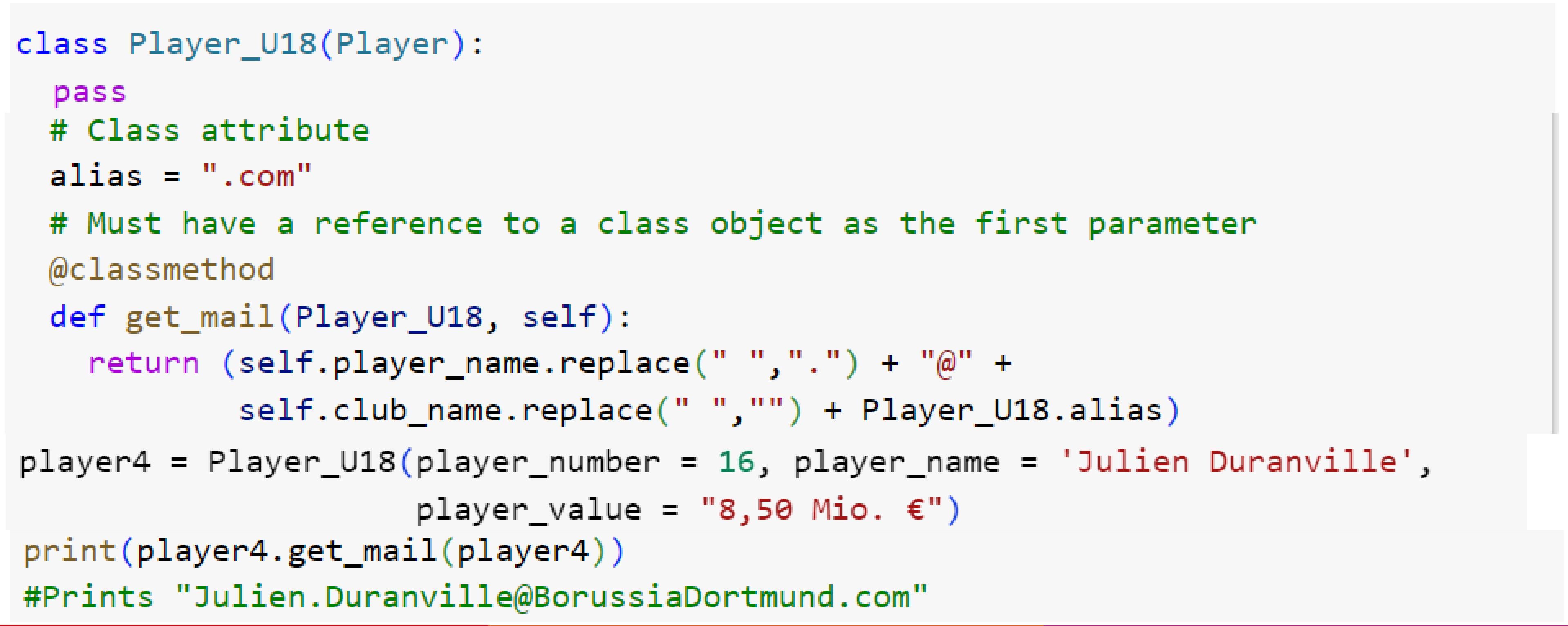
```
class Player_U18(Player):
  pass
  # Converts the method access to attribute access
  @property
  def player_first_name(self):
    return self.player_name.split(" ")[0]
```

```
player4 = Player_U18(player_number = 16, player_name = 'Julien Duranville',
                     player_value = "8,50 Mio. €")
```

```
print(player4.player_first_name)
#Prints "Julien"
```

Decorator @classmethod

- Must have a reference to a **class object** as the **first parameter**.
- Optional subsequent parameters including `self` object.



```
class Player_U18(Player):
  pass
  # Class attribute
  alias = ".com"
  # Must have a reference to a class object as the first parameter
  @classmethod
  def get_mail(Player_U18, self):
    return (self.player_name.replace(" ",".") + "@" +
      self.club_name.replace(" ","") + Player_U18.alias)
player4 = Player_U18(player_number = 16, player_name = 'Julien Duranville',
                     player_value = "8,50 Mio. €")
print(player4.get_mail(player4))
#Prints "Julien.Duranville@BorussiaDortmund.com"
```

Decorator @staticmethod

- **Does not** take **any obligatory parameters**.
- Basically, just a function, called syntactically like a method.

```python
from datetime import date, timedelta, datetime
```

```python
class Player_U18(Player):
  pass
  @staticmethod
  def get_age(player_birth_date):
    return (date.today() - player_birth_date) // timedelta(days=365.2425)
```

```python
player4 = Player_U18(player_number = 16, player_name = 'Julien Duranville',
                     player_value = "8,50 Mio. €")
```

```python
print(player4.get_age(date(2006, 1, 1)))
#Prints "17"
```

Destructor

- After relevant computation with **long arrays/matrices subsequently unused**: **delete** them to **save** some **memory**!
- `def __del__(self):` is a destructor method which is **called** as soon as all references of the **object** are **deleted** i.e., when an object is garbage collected.

```
class Player_U18(Player):
  pass
  def __del__(self):
    print("I do not exist anymore")
```

```
player4 = Player_U18(player_number = 16, player_name = 'Julien Duranville',
                     player_value = "8,50 Mio. €")
```

```
del(player4) #Prints "I do not exist anymore"
print(player4) #NameError: name 'player4' is not defined
```

- Destructors are **seldomly** used.

Technology
Arts Sciences
TH Köln

Takeaways

- Object-oriented programming is an imperative paradigm.
- Everything in Python is an object and many details are hidden away, but Python also can be developed functional and procedural.
- Even though Python is a high-level programming language with dynamic referencing, it can also be used to deploy object-oriented programming as in C.