Symbolische Programmiersprache - Lecture 3

Prof. Dr. Barbara Plank

Symbolische Programmiersprache
MaiNLP, CIS LMU Munich

WS2025/2026

Outline

Recap

Recap: Object-Oriented Programming (OOP)

- Object Oriented Programming is a way of computer programming which model functionalities through the interactions of **"objects"** that represent data and methods (behavior).
- Objects in OOP are an *abstraction* of reality.
- In the OOP paradigm, classes describe objects, and each object is an instance of a class.

Recap: software objects / real-life objects

- Class: `Account`
- Instances: `annesAccount`, `stefansAccount`

Attributes/Instances	`annesAccount`	`stefansAccount`
id	1	2
holder	'Anne'	'Stefan'
balance	200	1000

Attributes

- describe the *state* or *data* of an object
- can change with time
- typically nouns in program specifications

Methods

- modify an object or interact with other objects
- typically verbs as potential methods of a class

UML Class-Diagrams

- Class name on top
- Attributes:
  - May be more detailed: (*visibility* (+/-)) *name* : *type*
- Methods
  - should not include inherited methods
  - may contain information about return type

Class name
Attributes
Methods

Example:

Account
id: int holder: str balance: float
deposit(amount): None withdraw(amount): float

- Today: To represent the way other classes interact with them

Recap: Class and UML diagram

```
1  class Account:
2    def __init__(self, num, holder):
3      self.id = num
4      self.holder = holder
5      self.balance = 0
6
7    def deposit(self, amount):
8      self.balance += amount
9
10   def withdraw(self, amount):
11     if self.balance < amount:
12             amount = self.balance
13     self.balance -= amount
14     return amount
15
16   def __repr__(self):
17     return "Account {} Holder: {} Balance: {}"\
18            .format(self.num, self.holder, self.balance)
```

Account
id : int holder : str balance : float
__init__(num, holder) deposit(amount) withdraw(amount): float __repr__()

Recap: Constructor / Initialization Methods

- The constructor is called when we create a new object
- It set initial values when creating the object
- To know which object the constructor refers to, it needs the keyword *self*, which points to the new object itself
- Attributes of the object: *self.num*, *self.holder* versus (note:) *num* und *holder*: would be local variables of the method

```
1  class Account:
2    # Constructor (here object requires 2 arguments)
3    def __init__(self, num, holder):
4      self.num = num
5      self.holder = holder
6
7  annesAcc = Account(1, "Anne")
8  stefansAcc = Account(2, "Stefan")
9  bensAcc = Account() # error
10 # here we defined the class in the same scope (file)
```

Python modules - Selective import

- Typically classes are defined in dedicated modules (separate files, not the same scope)
- A specific class can then be imported in the main application:
- `from modulename import classname`

```
1  from accounts import Account
2
3  if __name__ == "__main__":
4      annesAcc = Account(1, "Anne")
5      annesAcc.deposit(200)
```

Recap: Coding practices

- Note that the following code is bad practice
- Instead use the constructor. In general, try to expose object attributes via methods not direct access (cf. lecture 2)
- Python does not have *private* variables, in practice direct access cannot be prevented (unlike e.g. in Java)
- However, variables that start with a underscore (e.g. `_countAccess`) are in practice treated as private variables

```
1  from accounts import Account
2
3  if __name__ == "__main__":
4      annesAcc = Account()
5      annesAcc.balance = 200 #bad
```

Today's lecture - Motivation

- Today we will learn more about **inheritance** and **composition** in OOP
- These are concepts about **relationships** of two classes
- Important concepts in object-oriented programming, as they help write less redundant and more reusable code

Inheritance

Inheritance

- Inheritance models an **is-a** relationship.
- If a Derived class inherits from a Base class, this means we created a relationship where Derived is a specialized version of Base.
- Classes from which other classes are derived are called **super classes**.

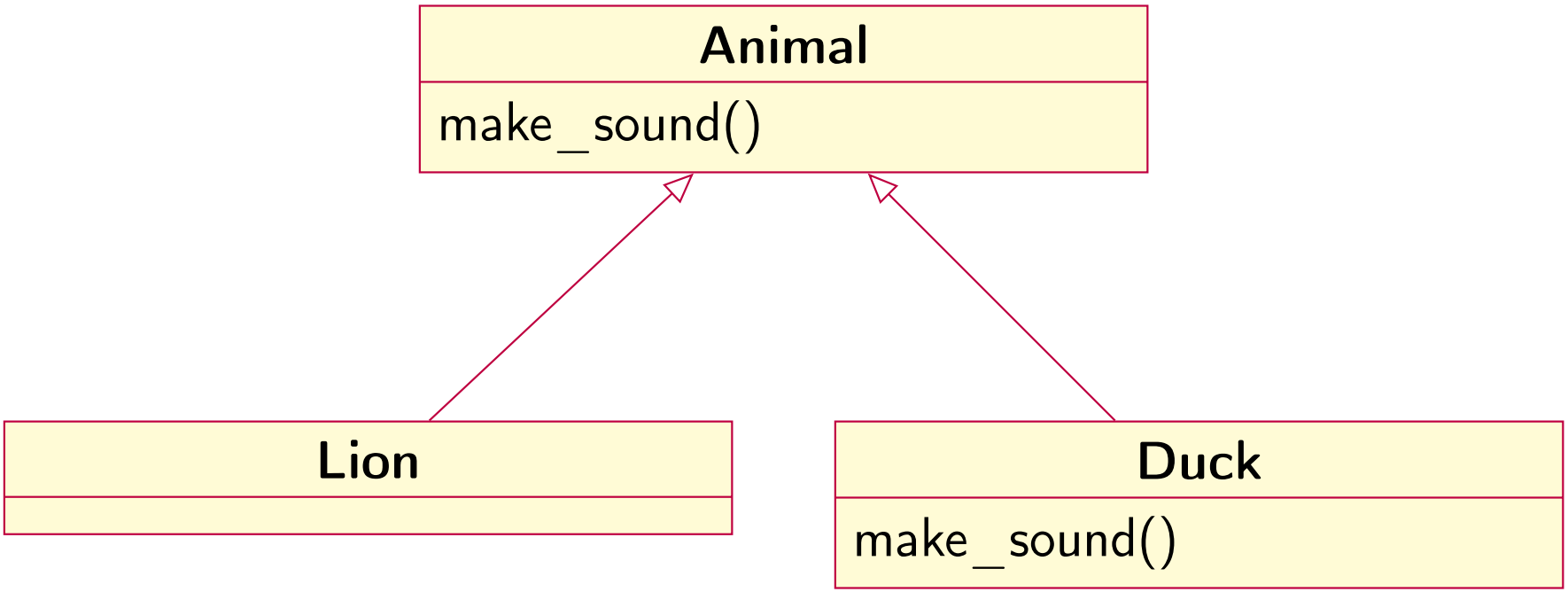
```
1  # Parent (or super) class
2  class Parent :
3              # Constructor
4              # Variables of Parent class
5              # Methods of Parent
6
7  # Child class inheriting from Parent class
8  class Child(Parent) :
9              # constructor of child class
10             # variables of child class
11             # methods of child class
12
```

Src: https://www.geeksforgeeks.org/

Inheritance in Python (Recap)

```
1  class Animal:
2      def make_sound(self):
3          print('ROAR')
4  # Uses the make_sound method of its parent.
5  class Lion(Animal):
6      pass
7  class Duck(Animal):
8          # Overrides the make_sound method of its parent.
9      def make_sound(self):
10         print('QUACK')
11
12 if __name__ == "__main__":
13     lion = Lion()
14     lion.make_sound() # prints ROAR
15     duck = Duck()
16     duck.make_sound() # prints QUACK
```

Inheritance in UML



- Name an advantage of UML

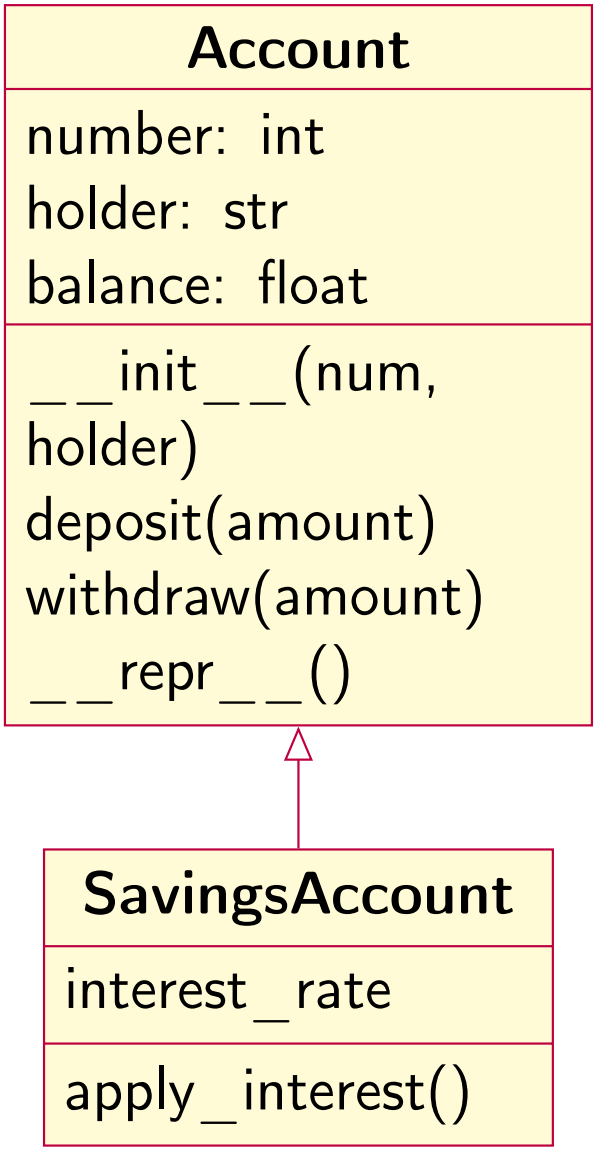
Example of General Functionality: Parent Class

```
1  class Account:
2  " a class providing general functionality for accounts"
3    def __init__(self, num, person):
4      self.balance = 0
5      self.id = num
6      self.holder = person
7
8    def deposit(self, amount):
9      self.balance += amount
10
11   def withdraw(self, amount):
12     if amount > self.balance:
13           amount = self.balance
14     self.balance -= amount
15     return amount
16
17   def __str__(self): # is like __repr__
18     res = ...
19         return res
```

Inheritance: bank example -

- **Savings account**: for each account we store the account number, the account holder and the account balance. The account balance should be ≥ 0. An interest rate, which is defined for all saving accounts, can also be applied. Money can be deposited into the account. The account statement, which can be printed, contains the account number, the account holder and the account balance.

Child Classes



- `SavingsAccount` "is based on" `Account`
- methods of the parent class are available in the child class
- line is-a: inheritance relation between SavingsAccount is-a (subclass of) Account
- dashed-line is-a: indicates instances of class SavingsAccount (objects)

```
1 annesAcc = SavingsAccount(1, "Anne")
2 annesAcc.deposit(200)
3 annesAcc.apply_interest()
4 print(annesAcc)
```

Child Classes

- SavingsAccount is based on Account
- SavingsAccount is derived from Account
- SavingsAccount extends Account
- SavingsAccount provides further functionalities:

```
1  class SavingsAccount(Account):
2  ''' class for objects representing savings accounts.
3  Derived from Account.
4  '''
5                  # METHODS added in SavingsAccount:
6                  ...
```

Inheritance: bank example -

- **Credit account**: for each account we store the account number, the account holder and the account balance. The account balance should lie within the credit limit defined for each client. If Anne's credit limit is $500, then her account balance can be a minimum of -$500. Money can be deposited into the account. The account statement, which can be printed, contains the account number, the account holder and the account balance.

Static variables (class variables)

- So far attributes were associated with objects (instances), and they could change through instance methods
- In Python, a **static variable** is said to be a class variable that is common to all class members.
- A static variable is declared within that class but outside the methods within the given class.
- Example: interest rate

```
1  class SavingsAccount(Account):
2  ''' class for objects representing savings accounts.
3  shows how a class can be extended. '''
4      # CLASS ATTRIBUTES
5      interest_rate = 0.035
6      # METHODS
7      def apply_interest(self):
8          self.balance *= (1+SavingsAccount.interest_rate)
```

Static variables (class variables)

- Accessing static variables inside the class via the class name `SavingsAccount.interest_rate`, instead of **self**
- How can we access a static variable outside the class? With the dot notation

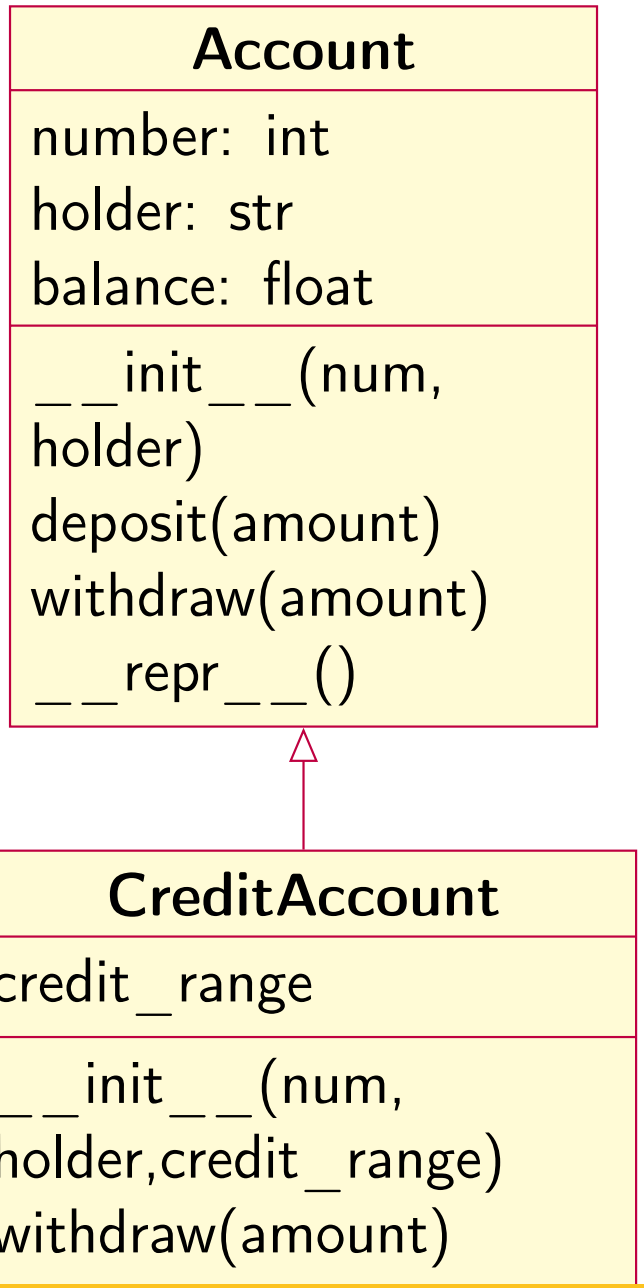
```
1 a = SavingsAccount()
2 print(a.interest_rate)
```

Static variables (class variables)


Instance variables vs. static/class variables

Method Overwriting/ Polymorphism/Redundancy

Overwriting Methods



- the `Account` class a `withdraw` method
- the `CreditAccount` class now **overwrites** it
- Which `withdraw` method is called?

```
1 annesAcc = CreditAccount
2 (1, "Anne", 500)
3 annesAcc.deposit(200)
4 annesAcc.withdraw(400)
5 print(annesAcc)
```

Polymorphism

- The method is called on the object $\rightarrow$ what happens depends on class hierarchy, i.e., one (same) call, different behavior possible due to for example overwriting

Example

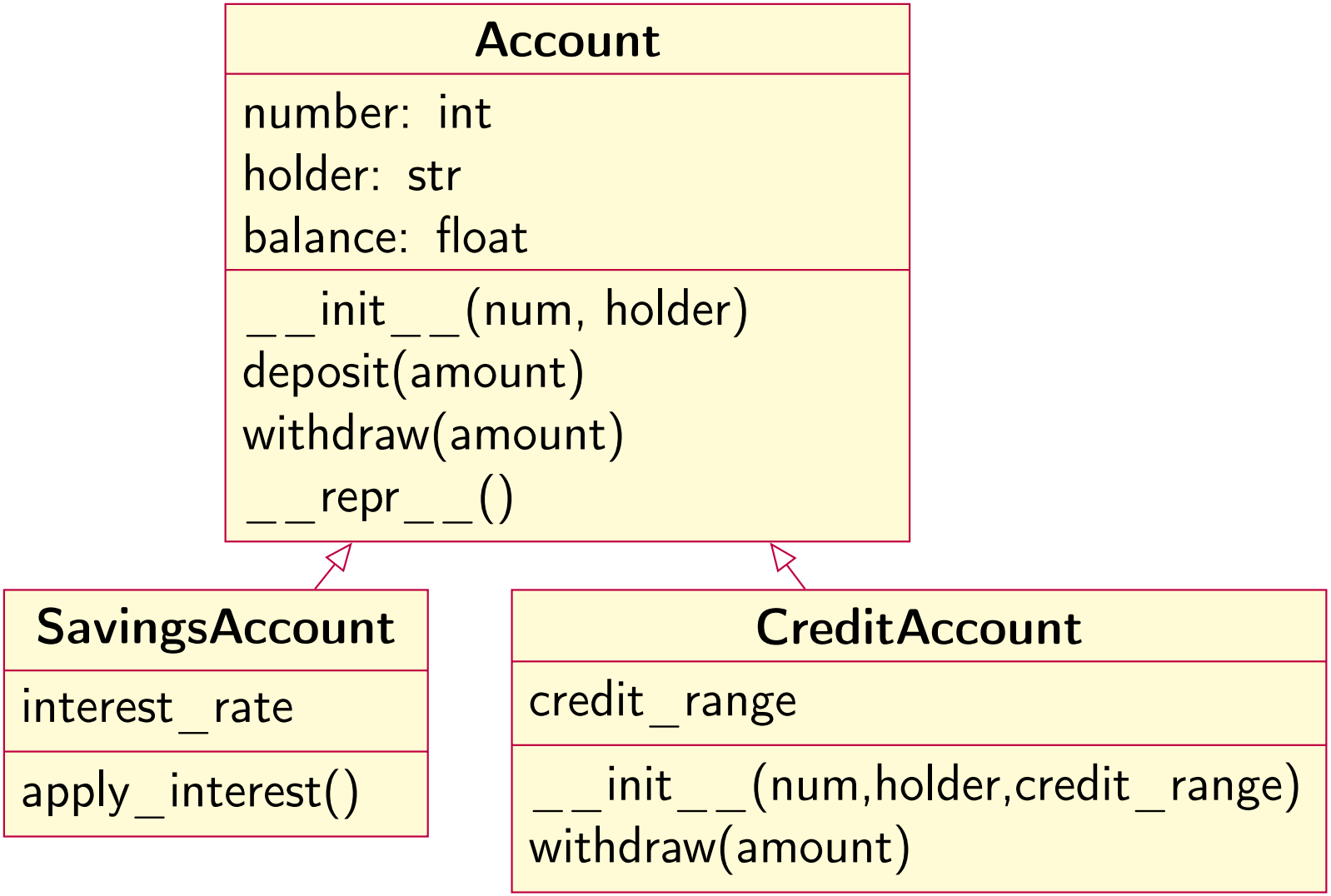
Application: `annesAcc.withdraw(400)`

- Python follows the class hierarchy and outputs the desired result

Overwriting Methods

```
1  class CreditAccount(Account):
2    # CONSTRUCTOR
3    def __init__(self, num, person, credit_range):
4      print("Creating a checkings account")
5      self.id = num
6      self.holder = person
7      self.balance = 0
8      self.credit_range = credit_range
9    # METHODS
10   def withdraw(self, amount):
11     amount = min(amount, abs(self.balance +
12                  self.credit_range))
13     self.balance -= amount
14     return amount
```

Inheritance in UML: Class Hierarchy



We could also have defined two separate `withdraw` methods (one in each child class). Why is it useful to have it in the `Account` class?

Redundancy

- when the data for an object exist in multiple places (e.g., the account holder information exists separately for each type of account)
  $\Rightarrow$ possible inconsistencies
- the same code is written multiple times
  $\Rightarrow$ hard to maintain

```
1  class Account:
2    def __init__(self, num, person):
3      self.balance = 0
4      self.id = num
5      self.holder = person
6
7  class CreditAccount(Account):
8    def __init__(self, num, person, credit_range):
9      self.id = num
10     self.holder = person
11     self.balance = 0
12     self.credit_range = credit_range
```

Minimizing Redundancies

- Here: call parent method from within a child and extend it with additional functionality
- Method is called on the class ⇒ the reference to the instantiated object has to be explicitly given to `self`

```
class Account:
  def __init__(self, num, person):
    self.balance = 0
    self.id = num
    self.holder = person

class CreditAccount(Account):
  def __init__(self, num, person, credit_range):
    Account.__init__(self, num, person)
    self.credit_range = credit_range
```

Multiple Inheritance

- in some object-oriented languages classes can only inherit from a single super class
- in Python a class can inherit from multiple classes
  $\Rightarrow$ Multiple Inheritance (Mehrfachvererbung)
- this is beyond the scope of this class
- recommendation: use at most one parent class

Can we create new objects from built-in Python classes?

Everything in Python is an object

- Yes! We have been using objects all the time
- Lists, dictionaries, strings, integers/floats etc are all objects

```
1  # create a new list object
2  myList = []
3  # call a method of the list object
4  myList.append(4)
5  # create a new dictionary object
6  myDict = {}
7  # call a method of the dictionary object
8  myDict["someKey"] = "someValue"
```

- Line 8 calls a magical method of the dictionary: `__setitem__(self, key, value)`

Everything in Python is an Object

- We can create child classes of the **built-in**-classes
- Here: overwrite magical methods

```
1  class TalkingDict(dict):
2      # Constructor
3      def __init__(self):
4          print("Starting to create a new dictionary...")
5          dict.__init__(self)
6          print("Done!")
7      # Methods
8      def __setitem__(self, key, value):
9          print("Setting", key, "to", value)
10         dict.__setitem__(self, key, value)
11         print("Done!")
12
13 >>> print("We are going to create a talking dictionary!")
14 >>> myDict = TalkingDict()
15 >>> myDict["x"] = 42
```

Everything in Python is an Object

```
>>> print("We are going to create a talking dictionary!")
>>> myDict = TalkingDict()
>>> myDict["x"] = 42

# OUTPUT
We are going to create a talking dictionary!
Starting to create a new dictionary...
Done!
Setting x to 42
Done!
```

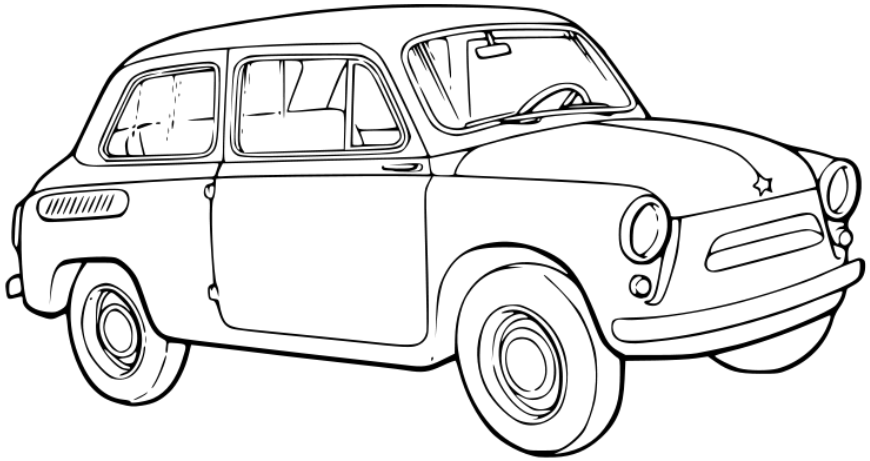
OOP, UML, Inheritance and Encapsulation

- OOP is widely used.
- We need to understand how the classes work together
- UML is a helpful tool to understand properties of objects and interactions of objects
- With inheritance we define subclasses which specify the behavior for our special application scenario
  - Encapsulation refers to the idea that some attributes or methods do not need to be exposed to other objects
    - Some data should preferably be only accessed within the instance itself (example: private variables)

OOP: Composition and Aggregation

Object Composition

Objects can be composed of different objects:



Concepts: Composition and Aggregation

- **Aggregation**: "is part of"
  - Example: an engine is part of a car
  - UML: clear white diamond
- **Composition**: "is entirely made of"
  - stronger version of aggregation
  - the parts live and die with the whole
  - UML: black diamond

Other classes as attributes

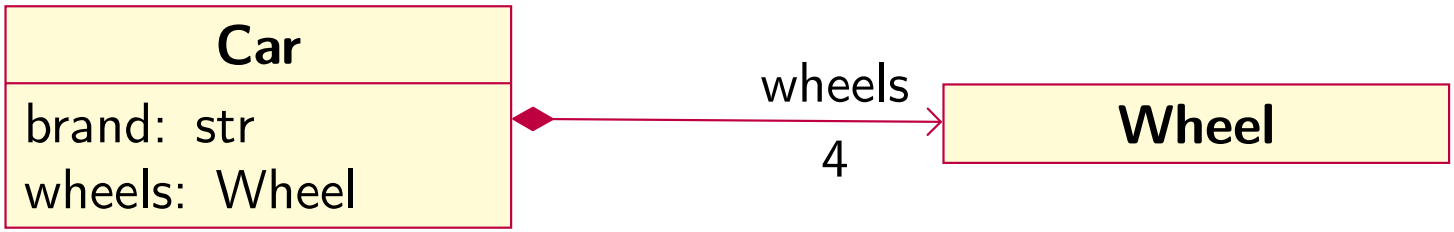
- Attributes of an object can have any type.
- They can themselves be (**complex**) objects:

Car
brand: str
engine: Engine

Book
title: str
pages: Page

Composition

- **Composition** is a type of aggregation in which two entities are reliant on one another.
  - complex objects are built based on other objects;
  - both entities are dependent on each other in composition.
  - the composed object usually cannot exist without the other entity when there is a composition between two entities.
  - In composition, one class acts as a container of the other class (contents). If you destroy the container, no content can exist. That means, that the container class creates an object or hold an object of contents.
  - Example: a `car` has 4 wheels



Another Example - Composition

(Example by dineshmadhup): A person has a heart (with 4 valves)

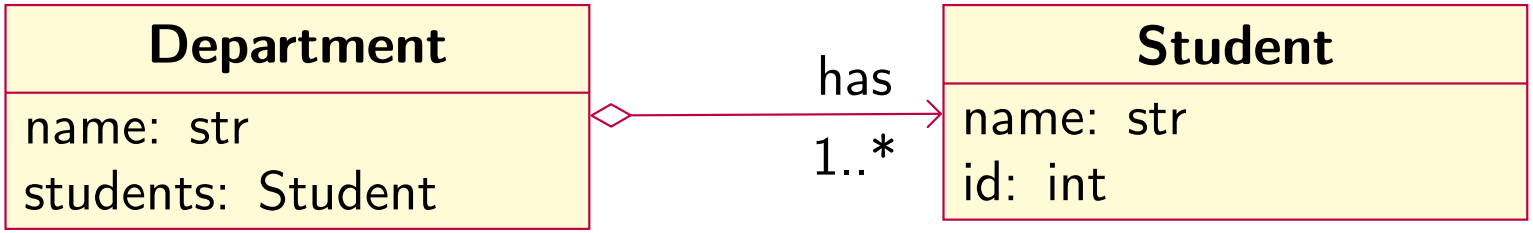
```
1   class Heart:
2       def __init__(self, heartValves):
3           self.heartValves = heartValves
4       def display(self):
5           return self.heartValves
6   class Person:
7       def __init__(self, fname, lname, address, heartValves):
8           self.fname = fname
9           self.lname = lname
10          self.heartValves = heartValves
11          self.heartObject = Heart(self.heartValves) #composition
12      def display(self):
13          .....
14  p = Person("Adam", "syn", "876 Zyx Ln", 4)
15  p.display()
```

Example - Composition

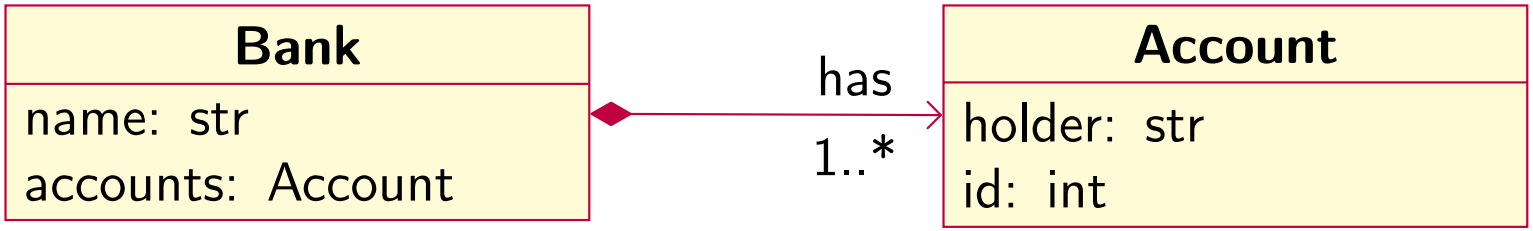
```
1  class Person:
2    def __init__(self, name, age):
3      self.name = name
4      self.age = age
5
6  class Account:
7    def __init__(self, num, name, age):
8      self.holder = Person(name, age) # composition
9      self.num = num
10     self.balance = 0
11
12   def deposit(self, amount):
13     self.balance += amount
14
15 annesAcc = Account(1, "Anne", 85)
16 annesAcc2 = Account(2, "Anne", 85)
17
```

Aggregation

- **Aggregation** is a concept in which an object of one class can own or access another independent object of another class.
  - unidirectional has-a relationship;
  - one entity has a relationship to the other entity, but the other direction does not hold. For example, a department has students, but the opposite is not possible (one-way relationship)
  - the composed object **can exist** in isolation (in contrast to composition)
  - Examples: `department` has `students` (1 or more)



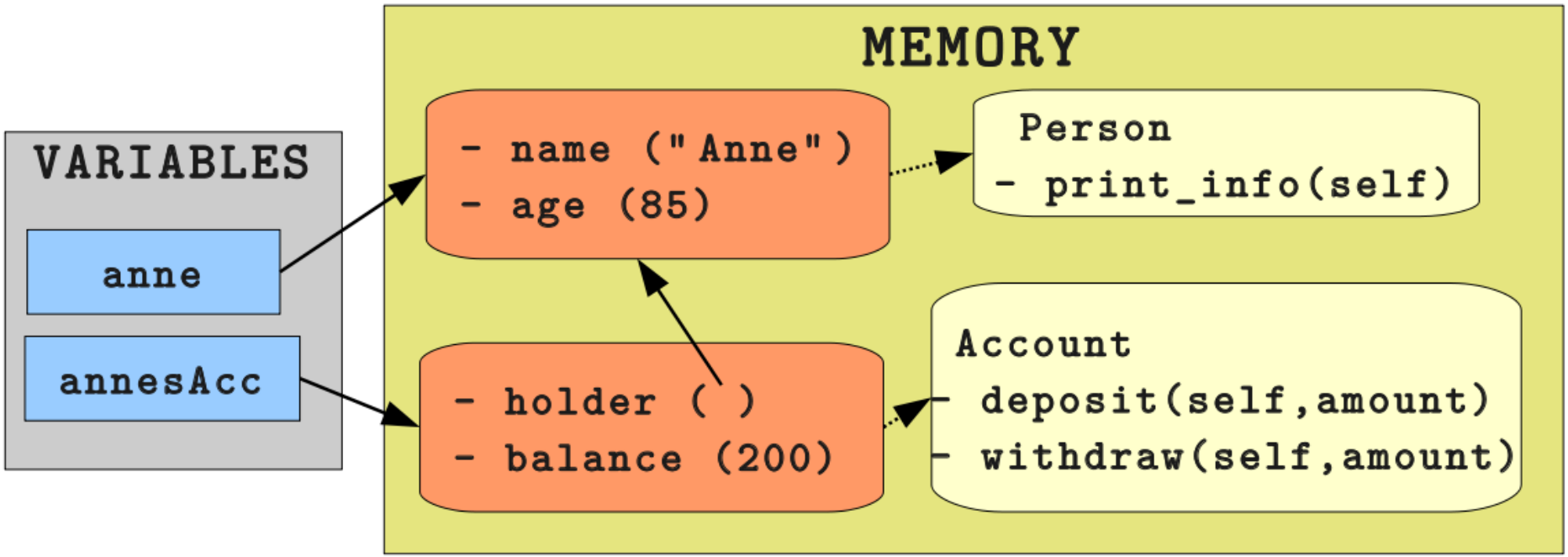
  - **In contrast: composition example:** If Bank ceases to exist, then any Accounts at that Bank also cease to exist.



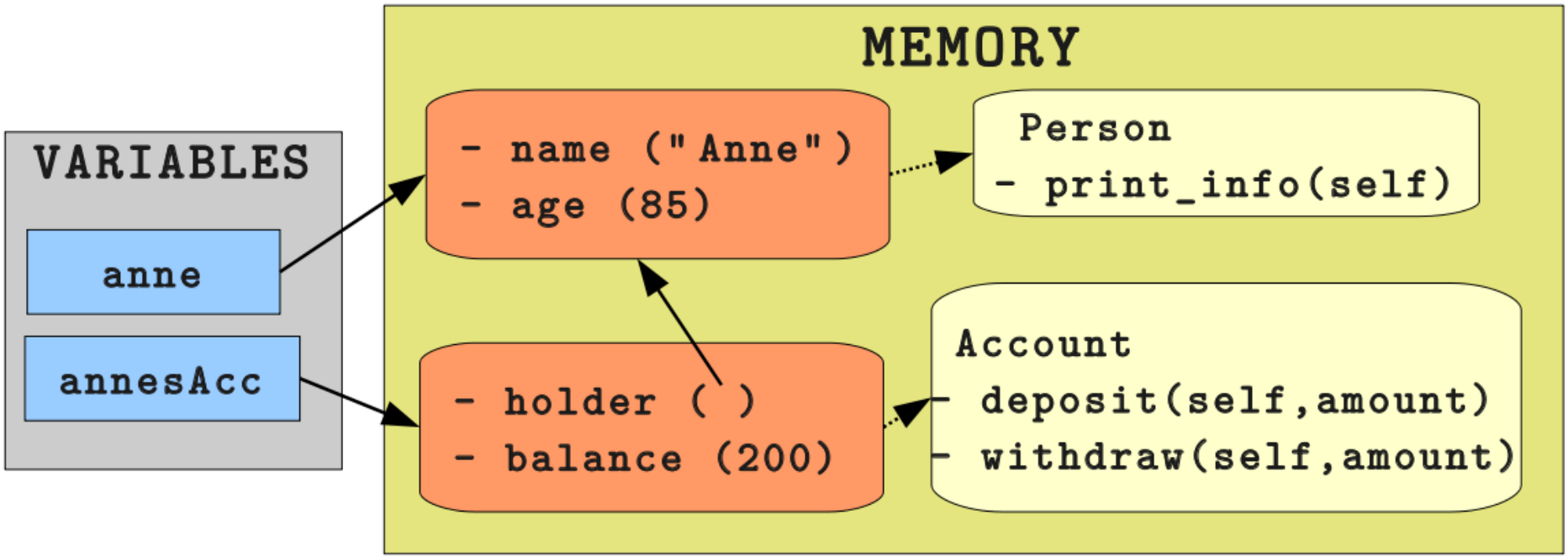
Example - Aggregation

```
1  class Person:
2    def __init__(self, name, age):
3      self.name = name
4      self.age = age
5
6  class Account:
7    def __init__(self, num, person):
8      self.num = num
9      self.holder = person
10     self.balance = 0
11
12   def deposit(self, amount):
13     self.balance += amount
14
15 anne = Person("Anne", 85). # Aggregation
16 annesAcc = Account(1, anne)
17 annesAcc2 = Account(2, anne)
18
```

Shared References



Shared References



- watch out where the attributes point: *annesAcc.holder.age += 1* also changes *annesAcc2.holder.age*
- this is ok here but could cause bugs in other cases

Summary

- Python OOP, UML
- Inheritance, Redundancy
- Composition and Aggregation

Regular Expressions (regex)

Regular Expressions

- **Pattern Matching** A regular expression (Regex) is a text matching pattern with a specific syntax.
- **Usage** Regular expressions are mostly used to match specific strings and extract sub-patterns from strings. They are widely used in computational linguistics, historically for various text processing tasks, and are still applied for tasks such as tokenization.

Regular Expressions

- A regular expression can contain:
  - characters to search for
  - operators – special symbols
  - reserved symbols (e.g. ‘any digit’ or ‘beginning of line’)
- By convention Regex is often given between slashes (Perl syntax – not actual Python)
  - ‘hello’: a string of characters, exactly ‘hello‘
  - /hello/: a Regex matching only the pattern ‘hello’

Regex Language

- A regular expression defines a formal language (set of strings licensed by the regex, including possibly **the empty string** ε).
- Examples:

Regex	**Language**
`a`	$L = \{a\}$
`ab`	$L = \{ab\}$
`a|bc*`	$L = \{a, b, ac, bc, acc, bcc, ...\}$ *)
`a*b*`	$L = \{\varepsilon, a, b, aa, ab, bb, ..., aaaaaab, ...\}$

*) strings starting with a or b followed by optional c

Regex Syntax

- **String patterns:** `H[ae]llo`, with `[ ]` set of possible characters, and H,l,o literal characters
  - Character sets: [aeiou]
  - Ranges: [A-Z], [0-9]
- The dot wildcard: . (matches any single character)
- Escaping: `\.` (matches the dot), literal meaning of meta character
- List of meta characters:
  - * (kleene star): zero or more occurrences of the immediately previous character or regular expression
  - + (kleene plus): one or more occurrences of the immediately preceding character or regular expression
  - ?: zero or one occurrences of the immediately preceding character or regular expression (in other words: is optional)
  - |: marks alternatives (disjunction), e.g. a|b, (cat)|(dog)
- Anchors:
  - ^: matches the beginning of a line
  - $: matches the end of a line

Regex Syntax (cont.)

- Character classes:
  - `\w` (alphanumeric character: a-zA-Z0-9_)
  - `\d` (digit, 0-9)
  - `\s` (whitespace, includes `\t` `\n` `\r`)
  - `\b` (matches the boundary (or empty string) at the start and end of a word)
- Capture groups: (abc), (a|b)
- Quantifiers: {n}, {n, m} – specify n to m repetitions of previous
- `[^ab5]`: ^ at start of set: exclude any character in the set. Here, it matches characters that are not a, b, or 5.

Regex Syntax (cont.)

Regex	Expansion	Match	First Matches
\d	[0-9]	any digit	Party␣of␣5
\D	[^0-9]	any non-digit	Blue␣moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!!
\s	[␣\r\t\n\f]	whitespace (space, tab)	in␣Concord
\S	[^\s]	Non-whitespace	in␣Concord

Figure: Aliases for Common Sets of Characters (taken from Chapter 2 of Speech and Language Processing by Daniel Jurafsky & James H. Martin).

Regex	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	zero or one occurrence of the previous char or expression
{n}	exactly *n* occurrences of the previous char or expression
{n,m}	from *n* to *m* occurrences of the previous char or expression
{n,}	at least *n* occurrences of the previous char or expression
{,m}	up to *m* occurrences of the previous char or expression

Figure: Regular Expression Operators for Counting (taken from Chapter 2 of Speech and Language Processing by Daniel Jurafsky & James H. Martin).

Regular Expressions - 

What can you match with the following regular expressions?

Regular Expressions - 

What can you match with the following regular expressions?

1. `[ts]h\w*`
2. `[^trs][a-z]+`
3. `<.*/?>`
4. `\d+-year-old`

Regular Expressions - Example solutions

1. `[ts]h\w*`

   ```
   # Example matches:
   the
   she
   show
   ```

2. `[^trs][a-z]+`

   ```
   # Example match:
   blue
   ```

3. `<.*/?>`

   ```
   # Example matches:
   <img/>
   <a>
   ```

4. `\d+-year-old`

   ```
   # Examples matches:
   1-year-old
   100-year-old
   ```

More reading: https://docs.python.org/3/howto/regex.html

Regular Expressions - Links

- Documentation:
  - `https://docs.python.org/3/howto/regex.html`
  - `https://docs.python.org/3/library/re.html`
- Test your regex online: `https://pythex.org/`
- Python regex cheatsheet
  `https://www.dataquest.io/blog/regex-cheatsheet/`

Regular Expressions - Challenge

Come up with a regular expression that matches the entire string of *afoot*, *foody*, *fool*, but does **NOT** match *forest*, *afluent*, *pool*, *foos*. Use as few characters as possible.

Regular Expressions - Challenge

Some solutions:

- /a?foo[tdl]y?/ 12
- /a?foo(t|l|dy)/ 13
- /a?foo([tl]|dy)/ 14

Regular Expressions in Python

- To use Regular Expressions in Python, import the module re
- Then, there are two basic ways to match patterns:
  - `re.match()`:
    Finds match of pattern at the beginning of a string
  - `re.search()`:
    Finds match of pattern anywhere in a string
- Both return a *match* object, that stores more information about the match, and `None` when there is no match.

Regular Expressions - Match

```
import re
wordlist = ['farmhouse', 'greenhouse', 'guesthouse']
for w in wordlist:
    if re.match('g.*', w):
        print(w)
```

Compiling regular expressions

If the same regular expression is used repeatedly (in a loop), it is more efficient to compile it outside the loop.

```
import re
wordlist = ['farmhouse', 'greenhouse', 'guesthouse']
regex = re.compile('g.*')
for w in wordlist:
    if regex.match(w):
        print(w)

# prints:
greenhouse
guesthouse
```

Regular Expressions - search vs match

```
re.match("c", "abcdef")     # No match
re.search("c", "abcdef")    # Match


# with prints
>>> print(re.match("c", "abcdef"))
None
>>> print(re.search("c", "abcdef"))
<re.Match object; span=(2, 3), match='c'>
```

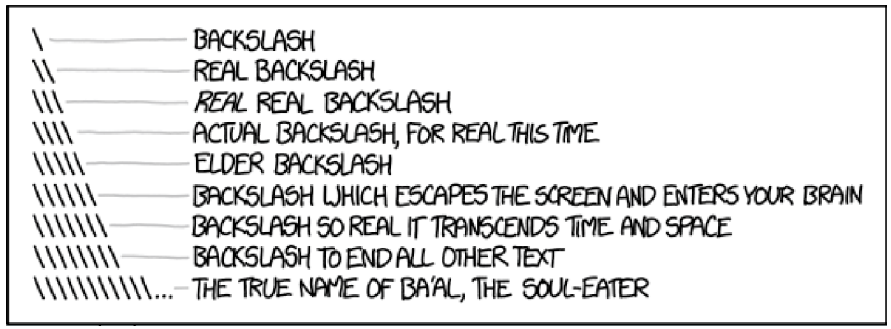
Regular Expressions - Find all

```
text = "He was carefully and quickly captured by police."
re.findall(r"\w+ly\b", text)
['carefully', 'quickly']
```

https://docs.python.org/3/library/re.html

Python Regular Expressions

- Regex and Python both use backslash "\" for special characters, and as a result you must type extra backslashes!
  - \\d+: to search for 1 or more digits
  - \n: in Python this means the 'newline' character, not a 'slash' followed by an 'n'. Need "\\n' for two characters.



- Use Python's raw string notation for Regex:

```
>>> r'[tT]he'
>>> r'\d+'
# matches one or more digits (instead of '\\d+')
```

Regular Expressions - The Backslash Plague

Python's raw string notation for regular expressions with backslashes (prefixed with 'r'): `https://docs.python.org/3/howto/regex.html`

```
>>> re.search("\ba","abcdef") # no match
>>> re.search(r"\ba","abcdef") # match (r=raw string)
<re.Match object; span=(0, 1), match='a'>
# Why? lets compile the pattern:
>>> p = re.compile("\b")
>>> p
re.compile('\x08') # Escape sequence \b is equivalent to \x08.
>>> p = re.compile(r"\b")
>>> p
re.compile('\\b')
```

Regular Expressions - Examples

```
print(re.match(r"[Tt]he\b.*", "The blue dragon"))
<re.Match object; span=(0, 15), match='The blue dragon'>

re.search(r"dog|cat","the dog sits next to the cat")
<re.Match object; span=(4, 7), match='dog'>

re.findall(r"dog|cat","the dog sits next to the cat")
# Find all substrings where the RE matches, as list:
['dog', 'cat']

>>> m = re.search(r"dog|cat","the dog sits next to the cat")
>>> m.group() # Match object, return  substring:
'dog'
>>> m.start() # start position - also: end()
4
>>> m.span() # returns both start and end indexes as tuple
(4, 7)
```

Questions

Questions?

Next Week: Representing Documents and NLTK: Corpus Linguistics

Credits

These slides were extended and or adapted from/by a.o. Katerina Kalouli, Annemarie Friedrich, Benjamin Roth, Alex Wisiorek, Janet Liu.