The Midterm…

Prof. Harald Gall
Dr. Carol Alexandru-Funakoshi
University of Zurich, Department of Informatics

Tasks

- Kprim evaluations: trivial, just run them in a Python terminal...

Question

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

	False	True
This expression evaluates as 1: `int(2.9-1)`	○	○
This expression evaluates as 1: `int('1')`	○	○
This expression evaluates as 1: `'3' - '2'`	○	○
This expression evaluates as 1: `int(True)`	○	○

Tasks

- Kprim evaluations: trivial, just run them in a Python terminal...
- Kprim knowledge questions: Ctrl-F the slides if unsure

Tasks

- Kprim evaluations: trivial, just run them in a Python terminal...
- Kprim knowledge questions: Ctrl-F the slides if unsure
- Programming:
  - "If / Elif / Else" and "Loops" tasks: straight-forward basic stuff

```
def normalize(number, lower, upper):
    if number > upper:
        return upper
    elif number < lower:
        return lower
    return number

    # or:
    #return min(upper, max(lower, number))
```

3 Points ≈ 3 Minutes

```
def product(xs, ys):
    for x in xs:
        for y in ys:
            print(x*y)
```

5 Points ≈ 5 Minutes

Tasks

- Kprim evaluations: trivial, just run them in a Python terminal...
- Kprim knowledge questions: Ctrl-F the slides if unsure
- Programming:
  - "If / Elif / Else" and "Loops" tasks: straight-forward basic stuff
  - "Functions" tasks: also straight-forward basic stuff

```
def apply(f1, f2, value):
    return f2(f1(value))
```

```
def add(a=0, b=0):
    return a+b
```

```
def add(n):
    def f(x):
        return n + x
    return f
```

```
def multiply(n, factor=1):
    return n*factor
```

6 Points ≈ 6 Minutes (?!)

Tasks

- Kprim evaluations: trivial, just run them in a Python terminal...
- Kprim knowledge questions: Ctrl-F the slides if unsure
- Programming:
  - "If / Elif / Else" and "Loops" tasks: straight-forward basic stuff
  - "Functions" tasks: also straight-forward basic stuff
  - "Déjà vu": copy paste from the slides and modify a tiny bit


```
def length(s):
  if (s == ""):
    return 0
  return 1 + length(s[1:])

length("asdf") # prints 4
```

```
def length(iterable):
    if iterable in ["", [], ()]:
        return 0
    return 1 + length(iterable[1:])
```

Tasks

- Kprim evaluations: trivial, just run them in a Python terminal...
- Kprim knowledge questions: Ctrl-F the slides if unsure
- Programming:
  - "If / Elif / Else" and "Loops" tasks: straight-forward basic stuff
  - "Functions" tasks: also straight-forward basic stuff
  - "Déjà vu": copy paste from the slides and modify a tiny bit
  - "Data structures": you needed to know how to iterate, append, sort, etc...

```python
def remove_every(l, n):
    res = []
    for i, item in enumerate(l):
        if (i+1) % n == 0:
            continue
        res.append(item)
    return res
```

```python
def duplicate_every(l, n):
    res = []
    for i, item in enumerate(l):
        if (i+1) % n == 0:
            res.append(item)
        res.append(item)
    return res
```

```python
def sort_dict_values(d):
    res = {}
    for key, value in d.items():
        res[key] = sorted(value)
    return res
```

Tasks

- Kprim evaluations: trivial, just run them in a Python terminal...
- Kprim knowledge questions: Ctrl-F the slides if unsure
- Programming:
  - "If / Elif / Else" and "Loops" tasks: straight-forward basic stuff
  - "Functions" tasks: also straight-forward basic stuff
  - "Déjà vu": copy paste from the slides and modify a tiny bit
  - "Data structures": you needed to know how to iterate, append, sort, etc...
  - "Riddle me this!": the only real challenge in this exam
    - Partial points awarded for each of the three required return values

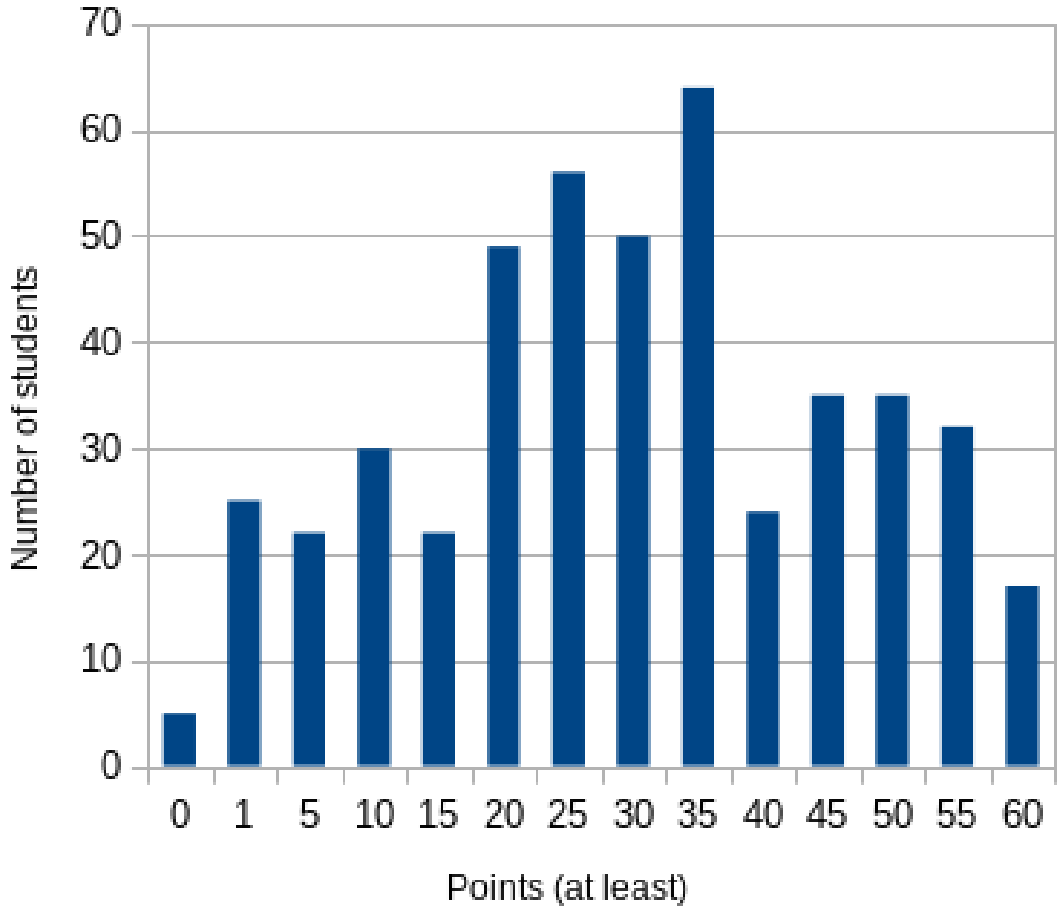
Tasks

Task	Points / Minutes	More likely minutes
Kprim 1	2	1
Kprim 2	2	1
Kprim 3	2	1
If / Elif / Else	3	1
Déjà vu	6	5
Loops	5	2
Functions (1)	6	2
Data structures	7	5
Functions (2)	6	2
Riddle me this!	21	?

- Given how simple the first 9 tasks were, there was plenty of time for the last task.
- If you did the first 9 tasks correctly, you got 39 points.
- The exam was absolutely doable in 60 minutes

Score Distribution

- Average: 31.8 points
- Median: 32 points
- 17 students got 60 points
  - 258 students got at least 30 points
  - 144 students got at least 40 points
  - 85 students got at least 50 points
- Last task:
  - 63 students got 21 points
  - 118 students got at least 12 points
  - 271 students got 0 points ._.



Submitting tasks without errors

- Follow these steps when done implementing a task:
  1. Run code with provided example calls to see if the output is correct
  2. Remove ONLY the example calls
  3. Run code again to make sure that there are still no errors
  4. Ctrl-A / ⌘-A or whatever it is to *select-all* in your IDE
  5. Ctrl-C / ⌘-C or whatever it is to *copy* in your IDE
  6. Click in textbox in Inspera, Ctrl-A / ⌘-A to select any existing stuff
  7. Ctrl-V / ⌘-V to *insert/overwrite* your solution
  8. Submit
- You will never have a Syntax or Import error this way
- It takes at most 15 seconds to do this

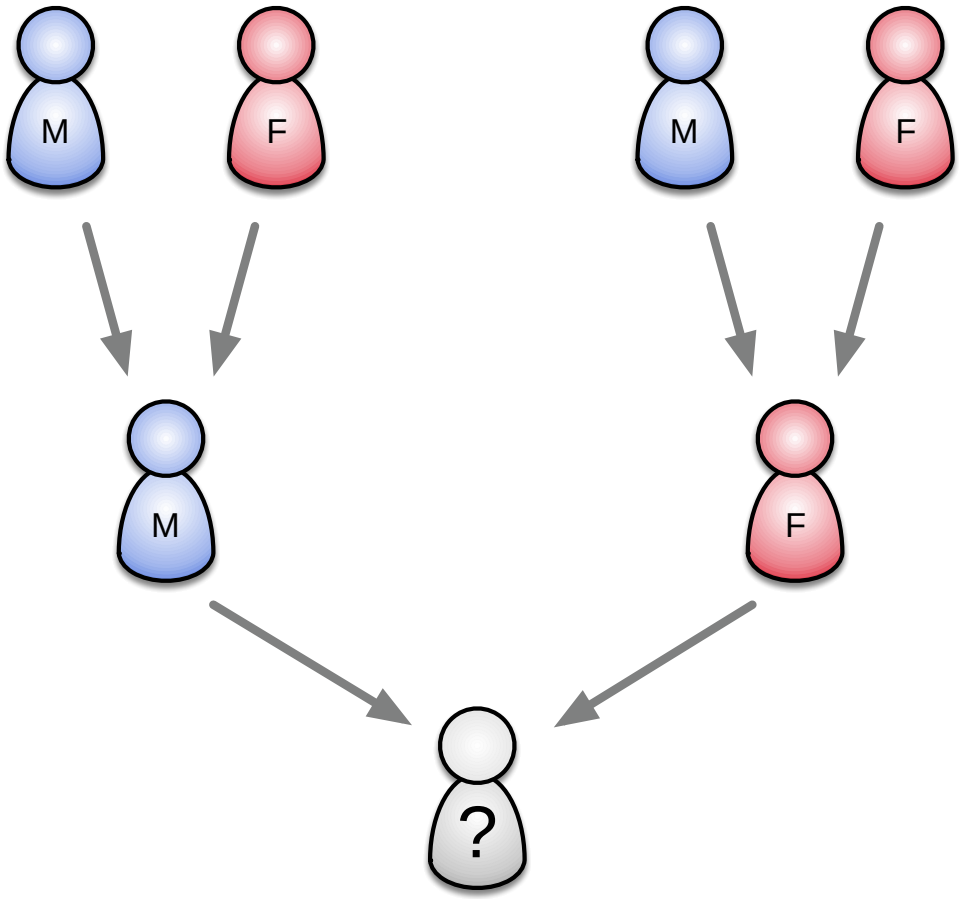
If you struggled in the midterm...

1. Practice programming. Again: it's 90 % crafting, 10% knowledge
   - If you struggle with the basics, solving actual tasks/problems is very hard, so first of all, just open a Python shell and start typing. For example:
     - Let's create a list of numbers and strings
     - How do I sort this list? Why can't I? Can I just sort the numbers? How do I add elements? How do I remove THAT one specific element? What if there's more than one? Can I count how long each element is? What if some elements don't have a length?
   - Once you truly feel comfortable with numbers, lists, dicts, function calls, classes, objects, etc..., then start solving old ACCESS and exam tasks
2. Organize your workspace for the final exam
   - Ready your Python shell, IDE, Slides, Online resources

Inheritance

Prof. Harald Gall
Dr. Carol Alexandru-Funakoshi
University of Zurich, Department of Informatics

Family Tree



Specific traits are passed on in a family tree.

Eye Color

Hair Color

Facial Shape

…

Animal Class (last week)

Type Name

Constructor

Instance Variables

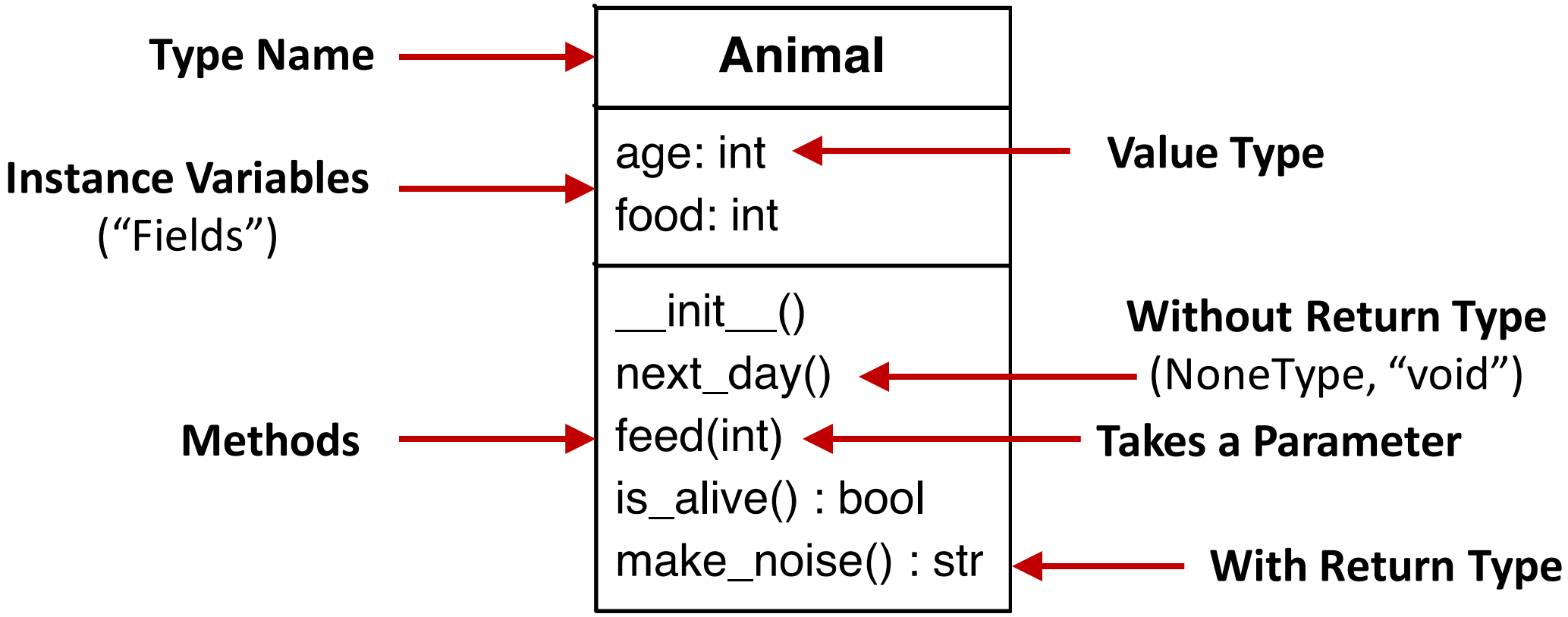
References to "self"

Methods

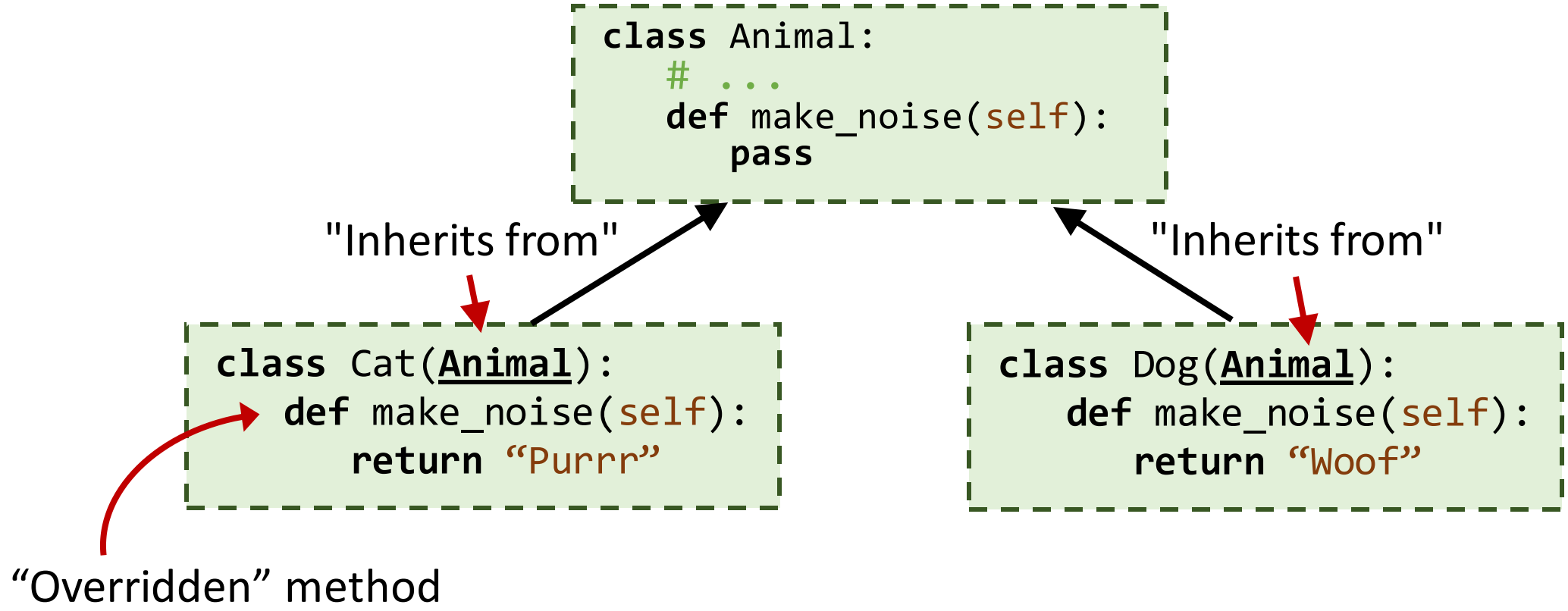
```
class Animal:
  def __init__(self):
    self.age = 0
    self.food = 1

  def next_day(self):
    self.age += 1
    self.food -= 1

  def feed(self, amount=1): self.food += amount

  def is_alive(self): return self.food > 0
It
  def make_noise(self): pass # depends
```

A class is an abstraction and represents a consistent combination of state (instance variables, so called "fields"/"attributes") and operations to alter this state ("methods").

Unified Modelling Language (UML)



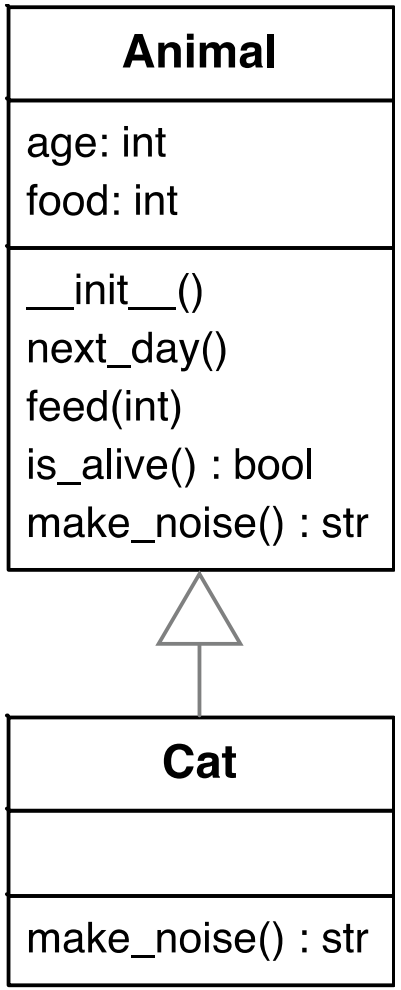
Ineritance Tree



NEW THIS WEEK:

In OOP languages, classes can extend existing classes to inherit their behavior. At the same time, they can override and extend the behavior.

UML: Inheritance



Animal is a super class of **Cat**

Cat is a sub class of **Animal**

Cat is an **Animal**

Cat() is an instance of **Animal**

Cat() is an instance of **Cat**

Cat and **Animal** are in a type hierarchy.

make_noise is dynamically "dispatched"

OOP languages support Polymorphism. They can decide at the runtime of a program, depending on the type of the receiver object, which implementation of a specific method needs to be invoked.

```
Cat().make_noise() # Purrr
Dog().make_noise() # Woof
Animal().make_noise() # ???
```

It does not make sense to **instantiate** `Animal`, how can we prevent it?

Abstract Base Class

Inherit from ABC

```
from abc import ABC, abstractmethod

class Animal(ABC): # ”A”bstract “B”ase “C”lass
  ...
  @abstractmethod
  def make_noise(self):
    pass
```

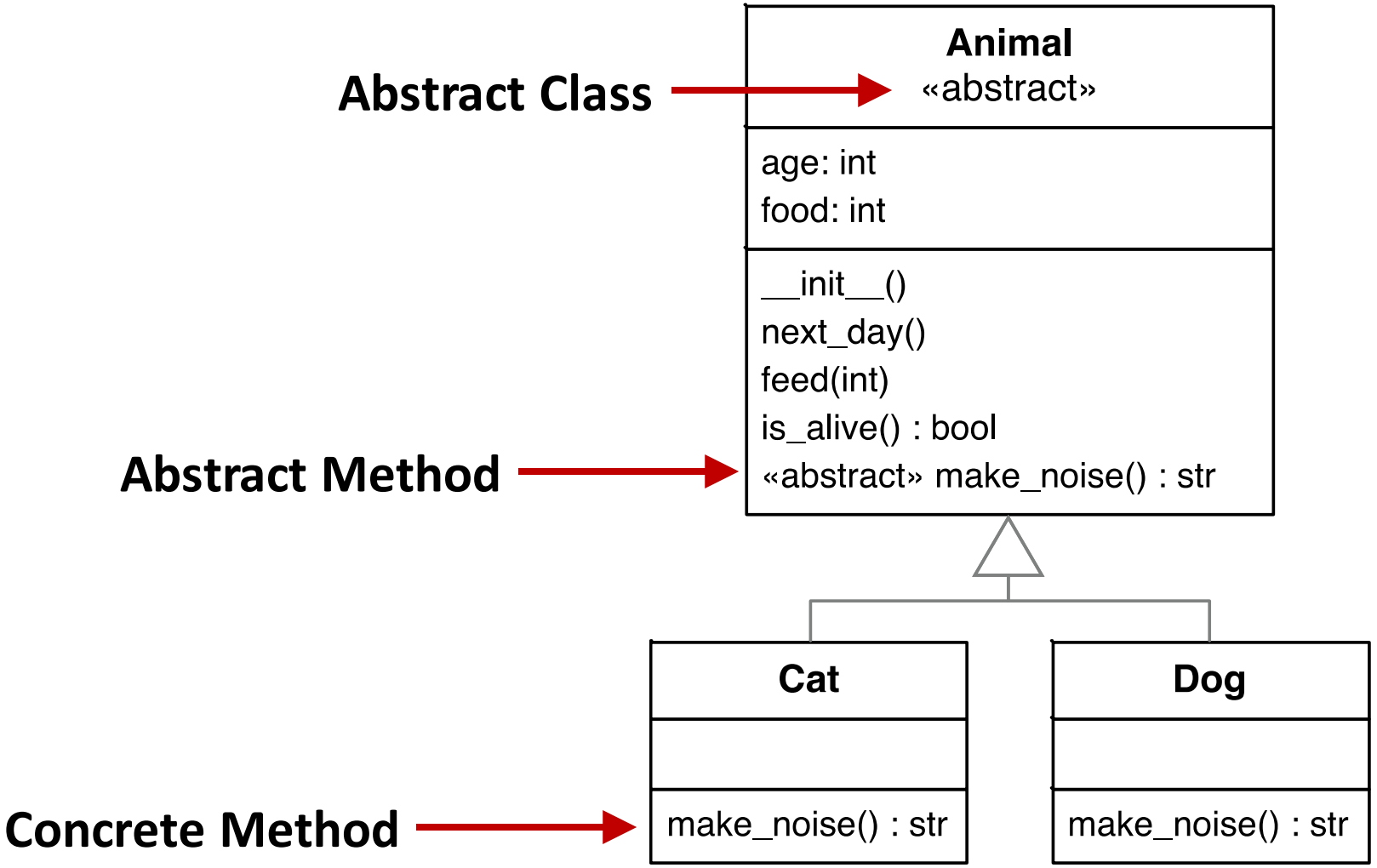
Use `@abstractmethod` decorator

```
a = Animal()
# TypeError: Can't instantiate abstract class
Animal with abstract methods make_noise
```

An **abstract** base class can provide functionality that is relevant for subclasses **without being constructable** (cannot be instantiated).

Define abstract methods in an abstract base class to indicate that subclasses **must** implement a particular method (template methods).

UML: Abstract



Elephants Need More Food

```
class Elephant(Animal):

  def next_day(self):
    self.food -= 3

  def make_noise(self):
    return “Toot”
```

We could override next_day, but that's crude.

It would make more sense to override how much food the animal requires each day, because there might be other shared functionality in next_day unrelated to food.

This extension should be done in the base class (Animal), then each concrete Subclass can override this.

Elephants Need More Food

```
class Animal(ABC):
  ...
  def next_day(self):
    self.__food -= self._hunger()
  def _hunger(self):
    return 1
```

```
class Elephant(Animal):
  def _hunger(self):
    return 3
  def make_noise(self):
    return “Toot”
```

We define a new method `_hunger` which returns the food requirement. We provide a default implementation (returning 1).

We could also have decorated this with `@abstractmethod` to require subclasses to always specify how much food they need

Visibility modifiers

```
class Animal(ABC):
  ...
  def next_day(self):
    self.__food -= self._hunger()
  def _hunger(self):
    return 1
```

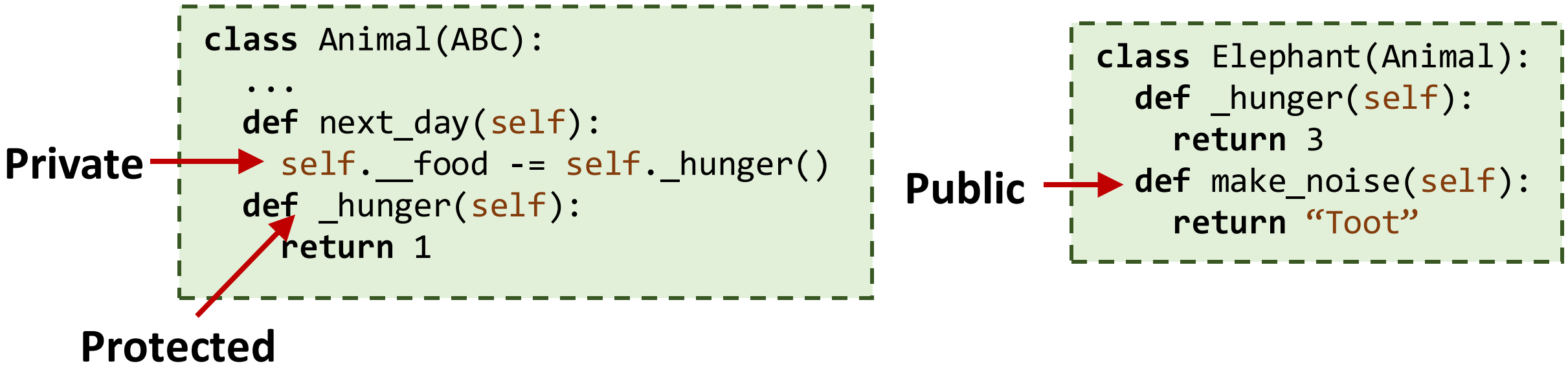
```
class Elephant(Animal):
  def _hunger(self):
    return 3
  def make_noise(self):
    return "Toot"
```

Private : __food is only visible from within the class and subclasses

Protected: _hunger is only visible from within the class and subclasses in the same module - *in Python only by convention!*

Public: make_noise is visible to any other class

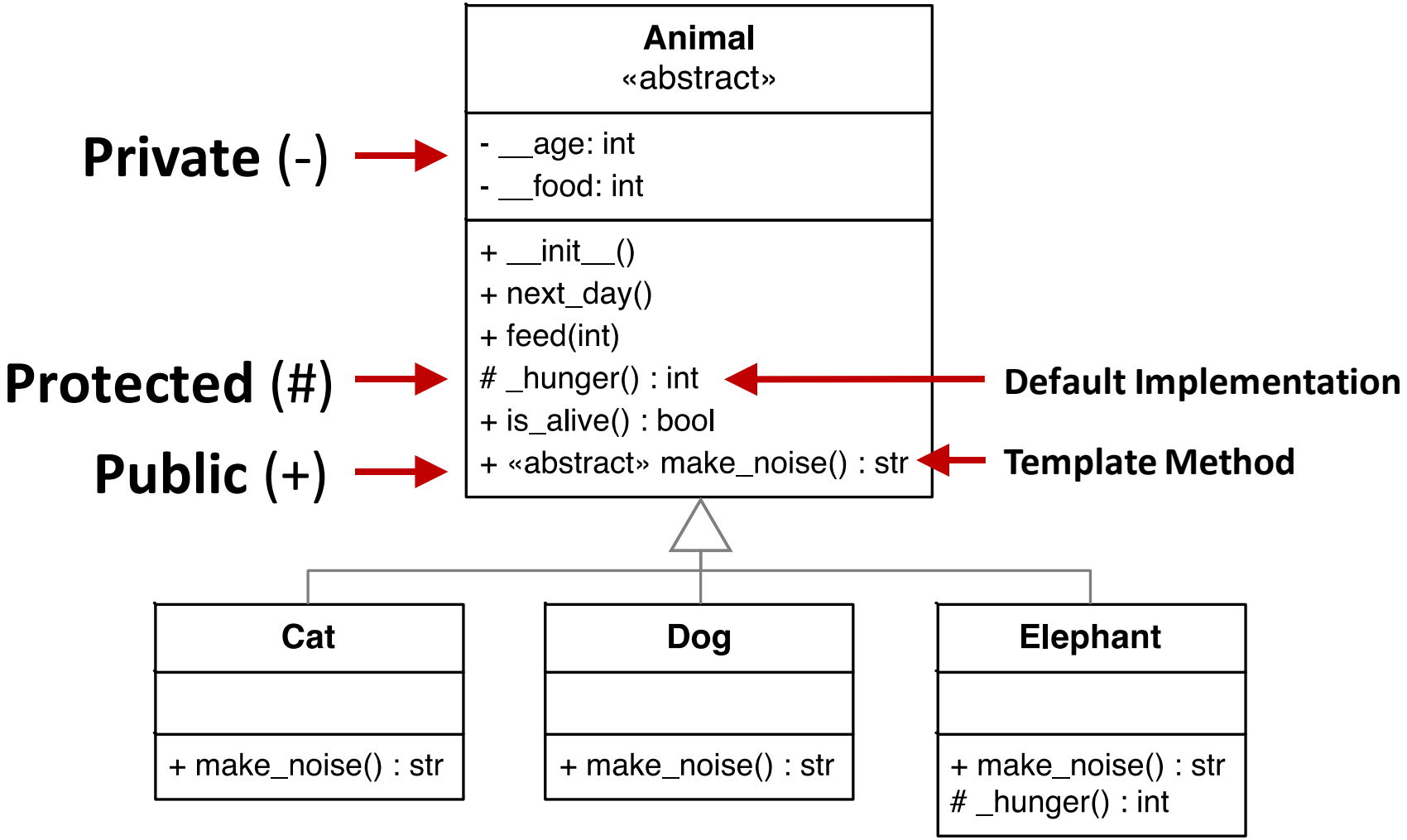
Visibility modifiers



Generalized UML definition of **visibility modifiers***:
Public: Always visible
Protected: Visible only to subclasses in the same module
Private: Visible only within class

* depends on programming language, for example in Java, there's also "no modifier", which indicates visibility for subclasses, even from different modules (called "packages" in Java).

UML: Visibility



What is the advantage of OOP?

```
zoo = [Cat(), Dog(), Elephant()]

for animal in zoo:
  assert isinstance(animal, Animal)

  # next day
  animal.next_day()

  # feed it
  if animal.is_alive():
    animal.feed(animal.hunger())
  else:
    zoo.remove(animal)
```

Only the abstract base class matters

Let's define hunger to be public after all

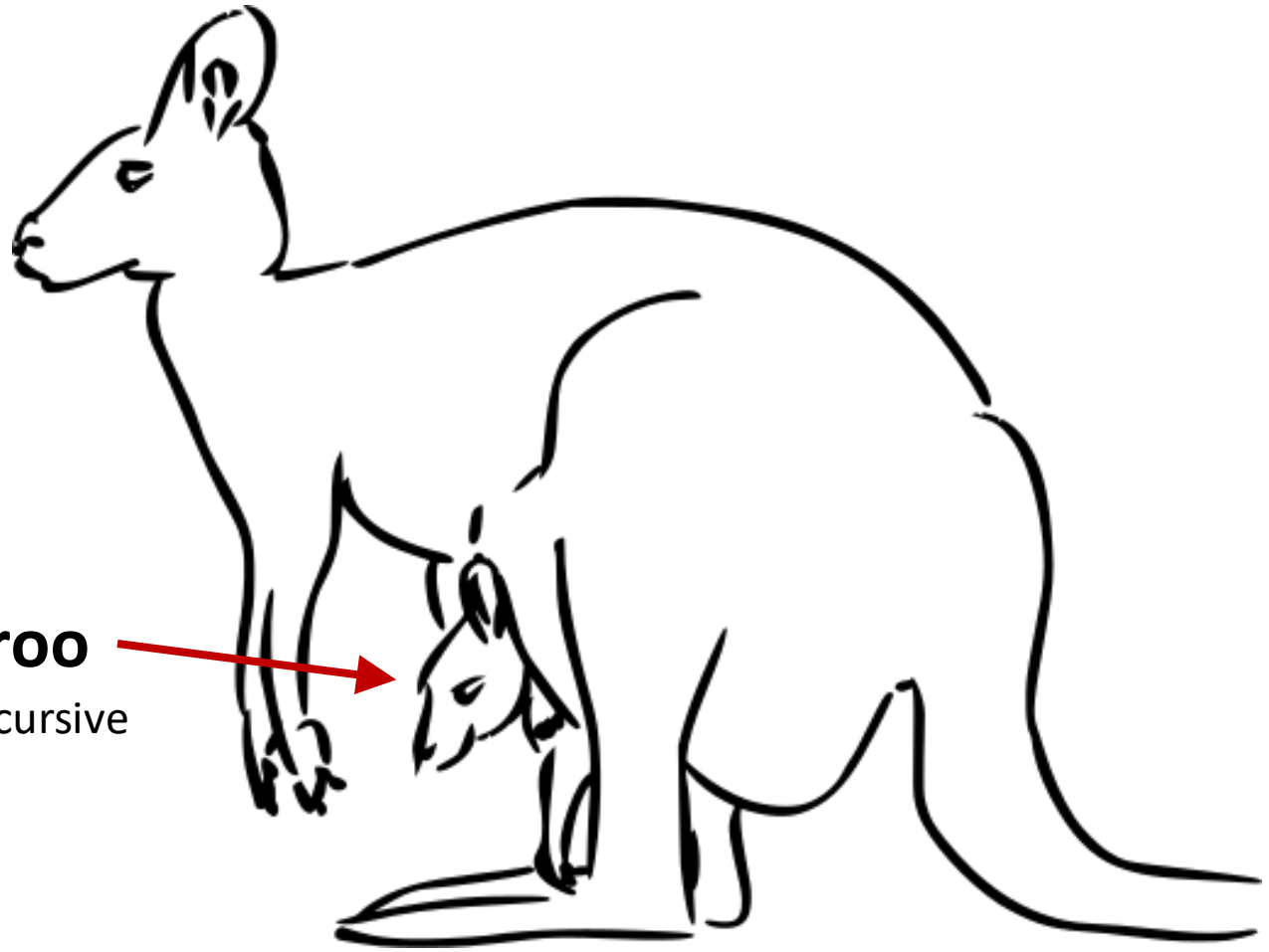
Often, you do not need the exact type of an object, it is enough to know the interface of an abstraction. This allows hiding implementation details of subtypes that add features.

Every instance gets the correct amount of food.
Implementation details of different types are hidden away.

⚠ Little Word of Caution

- You CAN check for the concrete type, i.e., isinstance(o, «Type»)
- We are in an *Introduction to Programming* course, so this is ok!
- However, adding checks for concrete types breaks extensibility!
- If it is strictly required, it is a *smell* of a bad system design.
- More advanced techniques for type checking exceed the scope the lecture; if you are interested have a look at the Visitor design pattern.

Kangaroos Have Pouches



Baby Kangaroo

Kangaroos are a recursive data structure!

https://svgsilh.com/image/47642.html

Let's Implement Kangaroos (Take 1)

```
class Kangaroo(Animal):
  def __init__(self):
    self.__pouch = []
  def reproduce(self):
    self.__pouch.append(Kangaroo())
  ...
```

We have to override `__init__` to add our **pouch**.

But now, what about **age** and **food**? These attributes are now undefined instances of this subclass because we've replaced `__init__`!

Let's Implement Kangaroo (Take 2)

```
class Kangaroo(Animal):
  def __init__(self):
    super().__init__()
    self.__pouch = []
  def reproduce(self):
    self.__pouch.append(Kangaroo())
  ...
```

What about age and `food`?

Use `super()` to redirect method calls to the implementation in the super class!
As a *general rule*, you **must** do this when overriding `__init__`, unless you're taking care of all instantiation matters in this method*.

*Here, this would mean again defining self.age and self.food, which would be an unnecessary duplication and **bad style.**

How do the baby kangaroos get fed? How do they age?

Let's Implement Kangaroo (Take 3)

"Override" existing methods

```
class Kangaroo(Animal):
  ...
  def next_day():
    for baby in self.__pouch:
      baby.next_day()
  def feed(self, amount):
    for baby in self.__pouch:
      baby.feed(amount)
```

How do the baby kangaroos get fed? How do they age?

Pass on the calls to the children!

Now when the mother is fed, the children are fed, too.

But now how does the mother get fed? How does she age?

Let's Implement Kangaroo (Final)

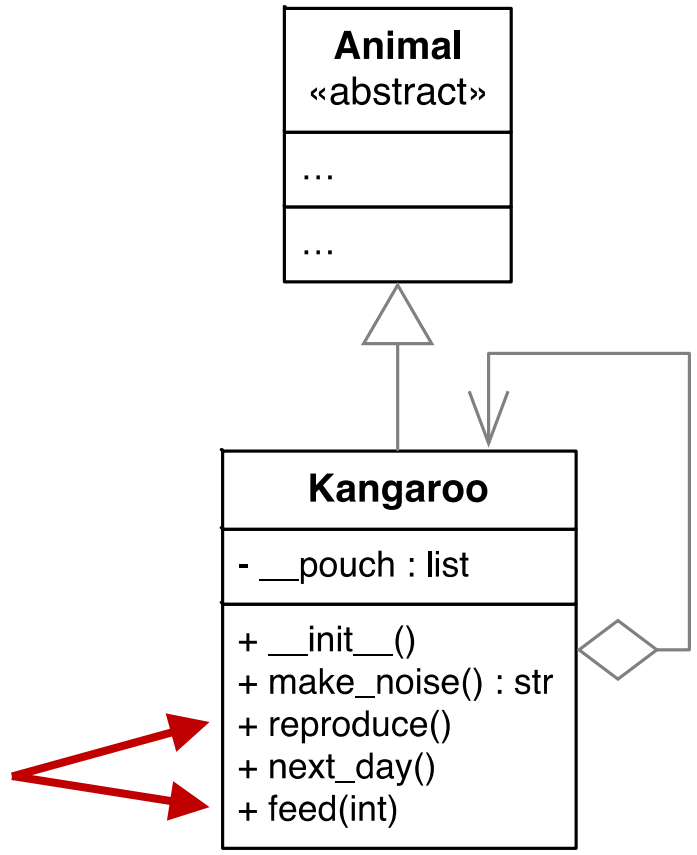
```
class Kangaroo(Animal):
  ...
  def next_day():
    super().next_day()
    for baby in self.__pouch:
      baby.next_day()

  def feed(self, amount):
    super().feed(amount)
    for baby in self.__pouch:
      baby.feed(amount)
```

How does the mother get fed?
How does she age?

Also call the super class implementation.

This way, the regular `Animal.feed` function is executed first, then the custom code is executed

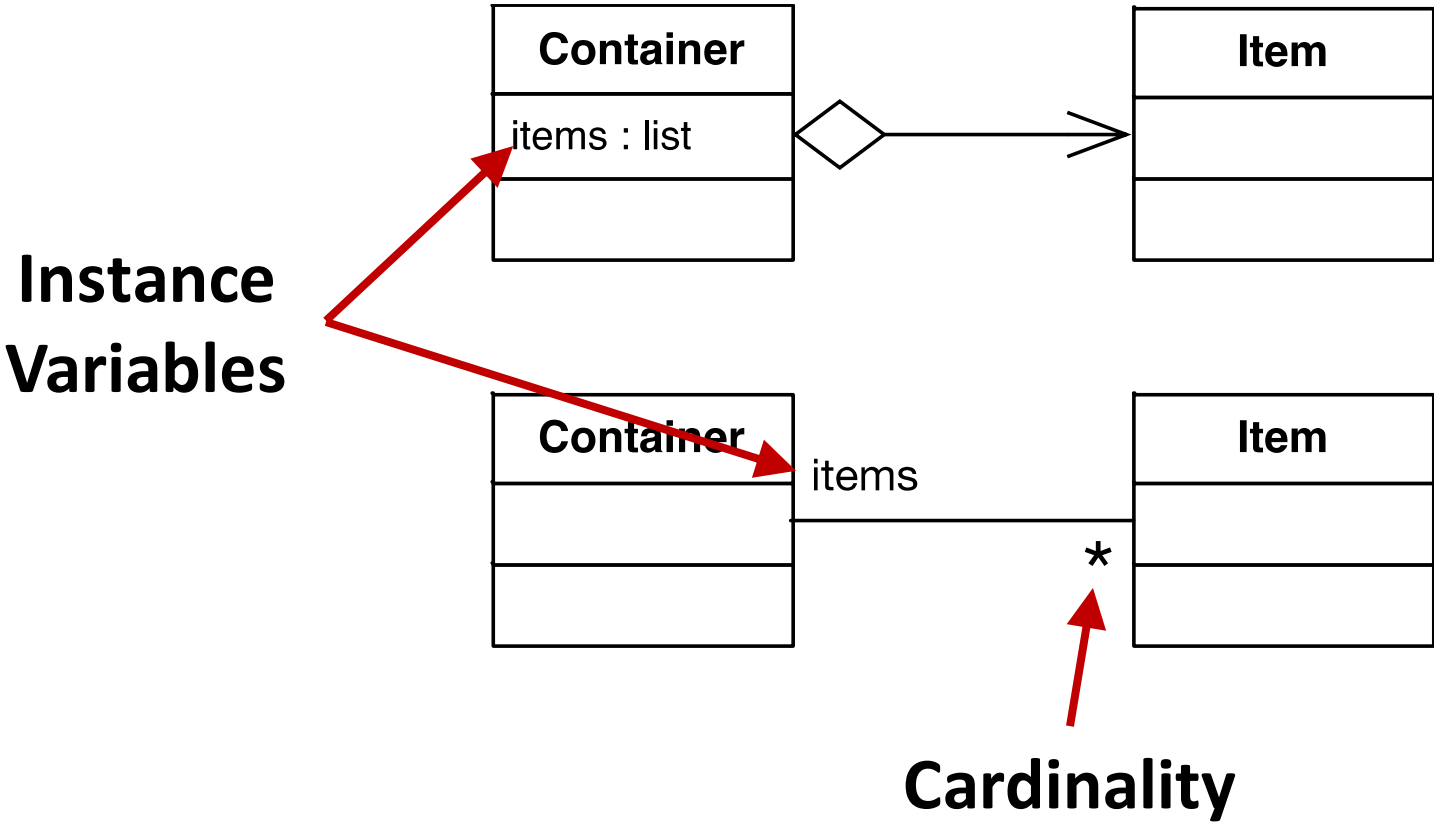
UML: Composition



Subclasses can replace existing methods and also add new ones

Kangaroo is a composite for other Kangaroos

UML: Composition and Cardinality



Exactly Once: 1
One or More: 1+
Optional: 1?
Bounded: 1..10
Unbounded: * (0..n)

Designin an implementatin for a "Safe"

https://www.allaboutsafes.co.uk/safes-c128/chubbsafes-elements-air-hotel-safe-p112

Requirements for a Safe Implementation

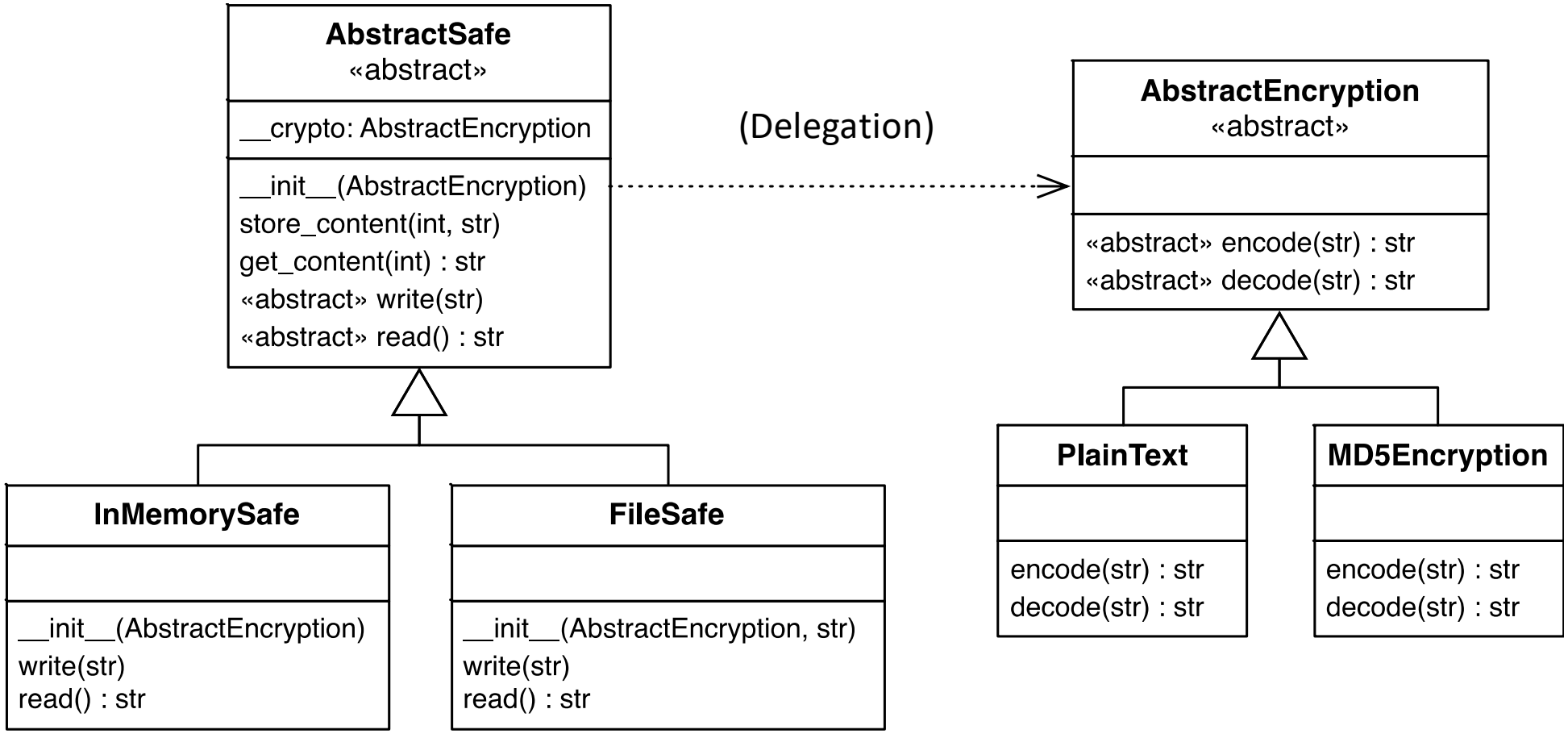
- Two Safe Variants
  - InMemorySafe
  - FileBasedSafe
- Two Encryptions
  - PlainText
  - MD5Encryption

```
a = InMemorySafe(PlainText())
b = InMemorySafe(MD5Encryption())
c = FileBasedSafe(PlainText(), “out.sav”)
d = FileBasedSafe(MD5Encryption(), “out.sav”)
```

```
def use_safe(safe):
    assert isinstance(safe, AbstractSafe)
    safe.store_content(123, “secret”)
    secret = safe.get_content(123)
```

Using abstractions allows to freely mix and match concrete implementations. Users **don’t need to discern**.

UML: Designing a Safe



Advanced: Multiple Inheritance

Date and Time

```
class Time:
  def __init__(self, h, m, s): ...
  def add_hour(self): ...
  def add_minute(self): ...
  def add_second(self):
    self.__sec += 1
    if self.__sec > 59:
      self.add_min()
    self.__sec %= 60
  def __repr__(self):
    return "{:02}:{:02}:{:02}".format( \
      self.__hour, self.__min, self.__sec)
```

```
t = Time(1, 2, 3)
print(t) # 01:02:03
for i in range(70):
  t.add_second()
print(t) # 01:03:13
```

```
class Date:
  def add_year(self):
    ...
  def add_month(self):
    ...
  def add_day(self):
    ...
```

Multiple Inheritance

Inherit from **multiple** superclasses!

```
class DateTime(Date, Time):

  def __init__(self, d,m,y, h,min,s):
    Date.__init__(self, d, m, y)
    Time.__init__(self, h, min, s)

  def __repr__(self):
    d = Date.__repr__(self)
    t = Time.__repr__(self)
    return "{}-{}".format(d, t)

  def add_hour(self):
    before = self._hour
    Time.add_hour(self)     # 23 -> 0
    if before > self._hour:
      self.add_day()
```

```
t = DateTime(1, 2, 3, 4, 5, 6)
print(t) # 01.02.0003-04:05:06
for i in range(25)
  t.add_hour()
print(t) # 02.02.0003-05:05:06
```

Using multiple inheritance, a class inherits the behavior of multiple parents. Conflicts are resolved by parent order, super names and calls need to be explicit*.

* Remember different ways of invoking a method from last week's slide 14

Mix-In

`TimeMixin` does not inherit from `Time`. It could be Used in other classes that have an `add_hour` method!


```
class TimeMixin:
  def add_several_hours(self, amount):
    for i in range(amount):
      self.add_hour()

tm = TimeMixin()
tm.add_several_hours()
# AttributeError: 'TimeMixin' object
has no attribute 'add_hour'
```

```
class MyTime(Time, TimeMixin): pass

t = MyTime(1, 2, 3)
print(t) # 01:02:03
t.add_several_hours(10)
print(t) # 11:02:03
```

Using mixins, it is possible to add/change behavior via a class that does not belong to the same type hierarchy.

Summary

The Four Core Principles of OOP

- **Abstraction**
  - Define a concept, unrelated to a concrete instance.
- **Encapsulation**
  - Hide implementation details (Fields cannot be overridden!).
- **Inheritance**
  - Extend classes to reuse code (inherit behavior), *is-a* relationships.
- **Polymorphism**
  - Replacing functionality in specializations. Dynamic selection of methods.

You should be able to answer these questions

- What is the idea behind inheritance? What is a type hierarchy?
- How to define (abstract) base classes? How to extend them?
- Which visibility levels exist and how do you declare them?
- How to provide classes that act as a composite for other items.
- How to delegate sub-problems to other referenced classes.
- How to read and write OOP designs in Unified Modelling Language
- How to use multiple inheritance and mixins in Python?
- OOP Core Principles: Polymorphism/Information Hiding

Example 1: Serializer!

Serializer

Data (e.g. list of tuples):

Name	Age	Height
Ann	31	168
Bob	28	191
Craig	23	171

HTML

```
<li>Ann (31): 1.68m</li>
<li>Bob (28): 1.91m</li>
<li>Craig (23): 1.71m</li>
```

CSV

```
"Ann",31,168
"Bob",28,191
"Craig",23,171
```

Natural Language

```
Ann is 31 years old and 168cm tall
Bob is 28 years old and 191cm tall
Craig is 23 years old and 171cm tall
```

Format-specific things?

- Chars before/after a line
  - Line template?
- How to format each column?
  - String with quotes?
  - cm or m?

Serializer

- Person (data class)
  - name, age, height
- PersonSerializer
  - serialize(person)
  - lineTemplate()
  - formatName(name)
  - formatAge(age)
  - formatHeight(height)

Example 2: Safe

Left as an exercise for the reader