Mer om OOP! =)

Dagens föreläsning

- Snabb koll på vad vi har lärt oss hittills
- Riktlinjer på hur man skriver Python-kod
- Inkapsling
- Fler möjligheter / tips om klasser
- Verklighetsexempel av Arv (inte bara *hund* är ett *djur*)

Repetition

```python
# Dog, as a dictionary
dog = {
    "name": "Doug",
    "breed": "Pug",
    "age": 8,
    "colors": ["white", "black", "beige"]
}

def print_info(dog):
    """
    Prints out dog information
    """
    print "Woof! I'm %s the %s (%s years)." % (dog["name"], dog["breed"], dog["age"])

def print_fur_colors(dog):
    """
    Prints out all fur colors of the dog
    """
    print "%s has the following fur colors: %s" % (dog["name"], ", ".join(dog["colors"]))

```

```python
class Dog(object):
    """
    Represents a dog
    """

    def __init__(self, name, breed, age, colors):
        """
        Initialize class attributes
        """
        self.name = name
        self.breed = breed
        self.age = age
        self.colors = colors

# ex.
dog = Dog("Doug", "Pug", 8, ["white", "black", "beige"])
# => "dog" is an instance of the class "Dog"
```

```python
class Dog(object):
    """
    Represents a dog
    """

    def __init__(self, name, breed, age, colors):
        """
        Initialize class attributes
        """
        self.name = name
        self.breed = breed
        self.age = age
        self.colors = colors

# ex.
dog = Dog("Doug", "Pug", 8, ["white", "black", "beige"])
# => "dog" is an instance of the class "Dog"
```

```python
class Dog(object):

    def __init__(self, name, breed, age, colors):
        self.name = name
        self.breed = breed
        self.age = age
        self.colors = colors

    def print_fur_colors(self):
        """
        Prints out all fur colors of the dog
        """
        print "%s has the following fur colors: %s" % (self.name, ", ".join(self.colors))

    def __str__(self):
        """
        String representation of a dog
        """
        return "Woof! I'm %s the %s (%s years)." % (self.name, self.breed, self.age)

# ex
dog = Dog("Doug", "Pug", 8, ["white", "black", "beige"])

print dog # the method __str__ is called
# => Woof! I'm Doug the Pug (8 years).

dog.print_fur_colors()
# => Doug has the following fur colors: white, black, beige
```

```python
class Dog(object):

    def __init__(self, name, breed, age, colors):
        self.name = name
        self.breed = breed
        self.age = age
        self.colors = colors

    def print_fur_colors(self):
        """
        Prints out all fur colors of the dog
        """
        print "%s has the following fur colors: %s" % (self.name, ", ".join(self.colors))

    def __str__(self):
        """
        String representation of a dog
        """
        return "Woof! I'm %s the %s (%s years)." % (self.name, self.breed, self.age)

# ex
dog = Dog("Doug", "Pug", 8, ["white", "black", "beige"])

print dog # the method __str__ is called
# => Woof! I'm Doug the Pug (8 years).

dog.print_fur_colors()
# => Doug has the following fur colors: white, black, beige
```

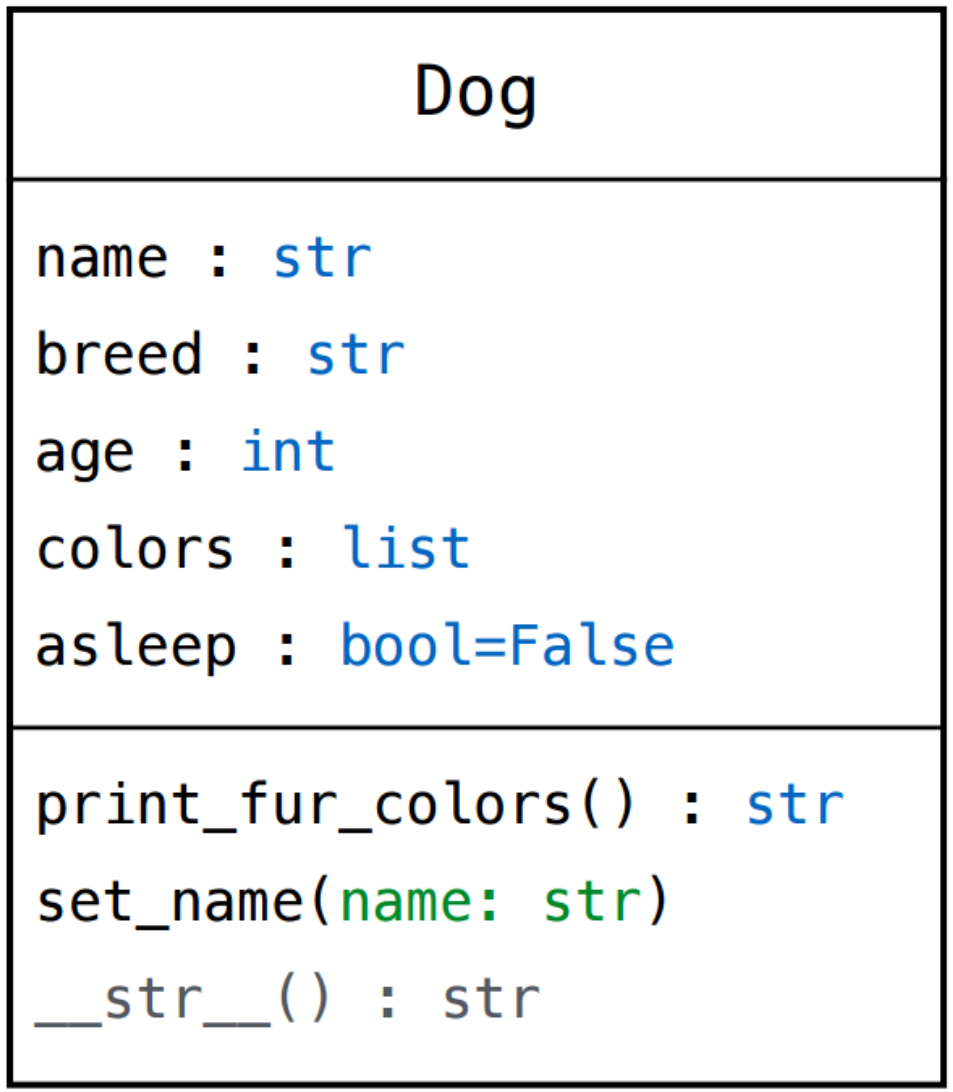
Klassdiagram

datatyp

attribut

metoder

Klassdiagram



datatyp

attribut

metoder

Arbetsflöde

- Identifiera vad som ska modelleras (substantiv)
- Skissa upp ett klassdiagram
- Implementera (stubb)
- Vidareutveckla klassdiagram och implementation

Om att skriva bra kod

The Zen of Python

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

De viktigaste punkterna

- Beautiful is better than ugly.
- Simple is better than complex.
- Readability counts.
- If the implementation is hard to explain, it's a bad idea.

PEP 8

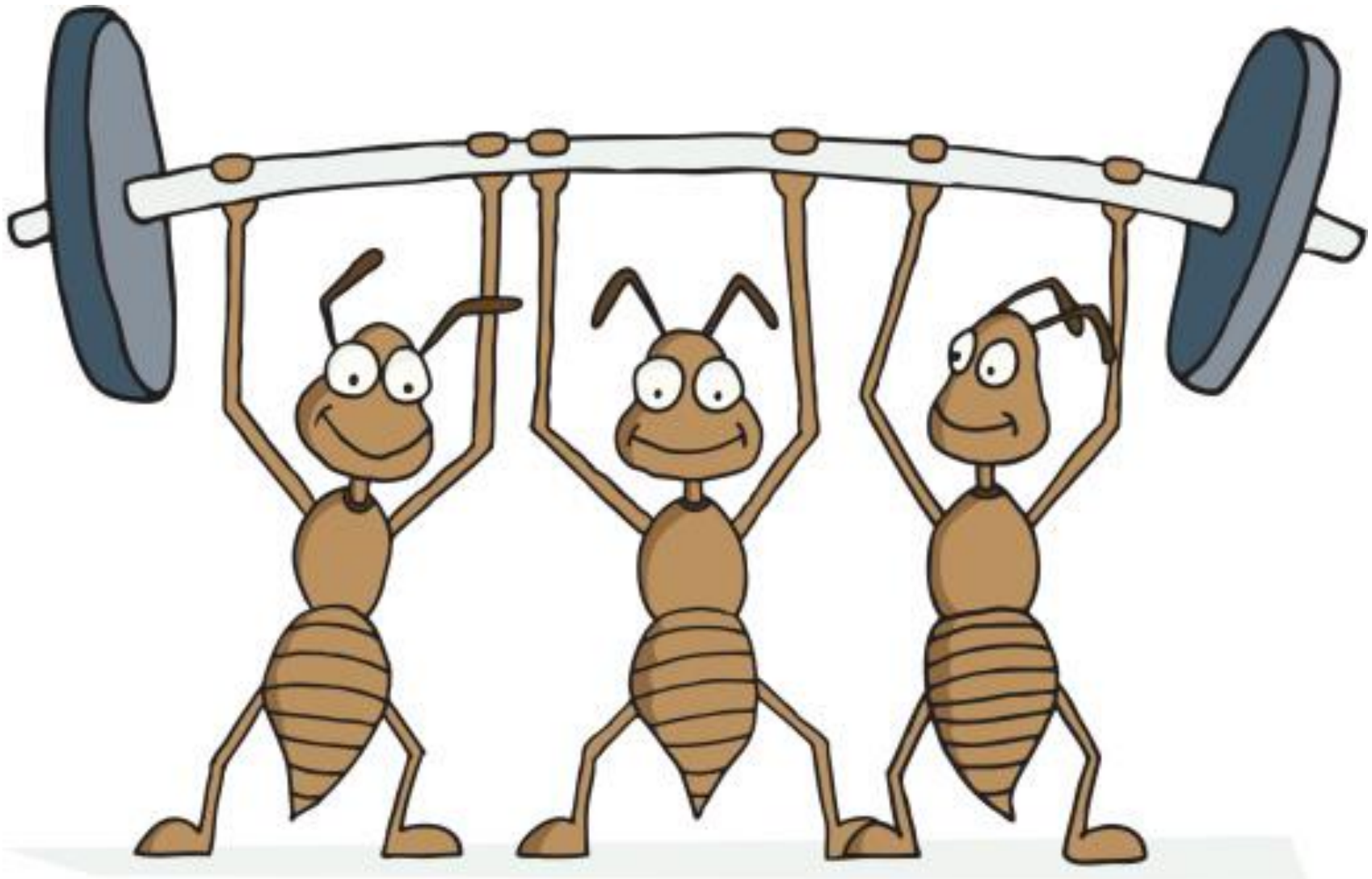
https://www.python.org/dev/peps/

PEP 0 -- Index of Python Enhancement Proposals (PEPs)

PEP:	0
Title:	Index of Python Enhancement Proposals (PEPs)
Last-Modified:	2017-11-28
Author:	David Goodger <goodger at python.org>, Barry Warsaw <barry at python.org>
Status:	Active
Type:	Informational
Created:	13-Jul-2000

Introduction

This PEP contains the index of all Python Enhancement Proposals, known as PEPs. PEP numbers are assigned by the PEP editors, and once assigned are never changed[1]. The Mercurial history[2] of the PEP texts represent their historical record.



PEP 8 är en style-guide för att skriva Python-kod

Det viktigaste med PEP 8

- Kod läses många fler gånger än den skrivs
- Samstämmighet för de som skriver/läser kod

Undantag mot PEP 8? När?

- Lyssna på omgivningen (kanske ert företag har andra designregler för kod?)
- Minskar läsbarheten

De viktigaste punkterna!

- 4st mellanslag (alt. TAB)
- Mixa *aldrig* tabbar och mellanslag
- Ha maximalt 79 tecken per rad, annars radbryt
- Använd blanka rader med måtta

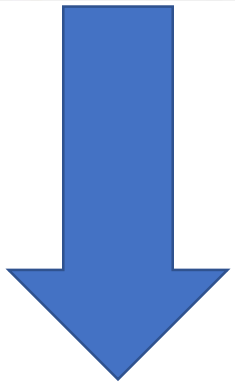
Dålig radlängd

```
# Example 1
things = ['overwrite', 'photobathic', 'tranquillization', 'resiny', 'runt', 'elpidite', 'Siganus', 'upplough', 'coed']
# This list comprehension is insane and should probably be split up into multiple statements
special_things = [special_thing for special_thing in special_things if special_thing == 'elpidite']

#                                                                  79 columns ->

# Example 2
if event.new_state.id == 'offline' and (state == 'published' or state == 'external'):
    workflow.doActionFor(content, 'reject', workflow='my_custom_workflow', comment='Rejecting content automatically')
```

```
# Example 1
things = ['overwrite', 'photobathic', 'tranquillization', 'resiny', 'runt', 'elpidite', 'Siganus', 'upplough', 'coed']
# This list comprehension is insane and should probably be split up into multiple statements
special_things = [special_thing for special_thing in special_things if special_thing == 'elpidite']

#                                                                    79 columns ->

# Example 2
if event.new_state.id == 'offline' and (state == 'published' or state == 'external'):
    workflow.doActionFor(content, 'reject', workflow='my_custom_workflow', comment='Rejecting content automatically')
```

```
# Example 1
things = [
    'overwrite',
    'photobathic',
    'tranquillization',
    'resiny',
    'runt',
    'elpidite',
    'Siganus',
    'upplough',
    'coed']


# Instead of using a list comprehension, we'll use the filter built-in
# to make the code have more clarity.
def my_checker(item):
    if item == "elpidite":
        return item


special_things = filter(my_checker, things)
```

```python
# Example 1
things = ['overwrite', 'photobathic', 'tranquillization', 'resiny', 'runt', 'elpidite', 'Siganus', 'upplough', 'coed']
# This list comprehension is insane and should probably be split up into multiple statements
special_things = [special_thing for special_thing in special_things if special_thing == 'elpidite']

#                                                                    79 columns ->

# Example 2
if event.new_state.id == 'offline' and (state == 'published' or state == 'external'):
    workflow.doActionFor(content, 'reject', workflow='my_custom_workflow', comment='Rejecting content automatically')
```



```python
# Example 2
public_state = state in ['published', 'external']
if event.new_state.id == 'offline' and public_state:
    workflow.doActionFor(
        content,
        'reject',
        workflow='my_custom_workflow',
        comment='Rejecting content automatically')
```

Radbrytningar

```
import random

ASCII_CAT1 = """\
 /\\_/\\\\
( o.o )
 > ^ <
"""
ASCII_CAT2 = """\
  _ _/|
 \\'o.0'
 =(___)=
    U
"""
CATS = [ASCII_CAT1, ASCII_CAT2]


class CatMadness(object):
    """Cats are curious animals. This is a silly example"""

    def __init__(self, num_cats=0):
        self.num_cats = num_cats

    def make_it_rain(self):
        """Just cats, no dogs yet."""
        count = self.num_cats
        while count > 0:
            count -= 1
            print random.choice(CATS)
```

Imports

- Dåligt

```
import os, sys
import config
from my.package.content import *
```

- Bra

```
import os
import sys
# explicit is better than implicit
from my.package import config
from my.package.content import Octopus, Blowfish
```

Blanksteg - dåligt

```
counter          =5
another_counter =15
more_cowbell= counter+10
my_dict ={'spam':'eggs','ham':'parrot'}



def complex (real, imag = 0.0):
    return magic(r = real, i = imag)



my_list=[1, 2,3]
another_list = [4,5,6]
combined_list=my_list+another_list
```

Blanksteg - bra

```
counter = 5
another_counter = 15
more_cowbell = counter + 10
my_dict = {'spam': 'eggs', 'ham': 'parrot'}



def complex(real, imag=0.0):
    return magic(r=real, i=imag)



my_list = [1, 2, 3]
another_list = [4, 5, 6]
combined_list =  my_list + another_list
```

Dåligt

```
counter           =5
another_counter =15
more_cowbell= counter+10
my_dict ={'spam':'eggs','ham':'parrot'}


def complex (real, imag = 0.0):
    return magic(r = real, i = imag)


my_list=[1, 2,3]
another_list = [4,5,6]
combined_list=my_list+another_list
```

Bra

```
counter = 5
another_counter = 15
more_cowbell = counter + 10
my_dict = {'spam': 'eggs', 'ham': 'parrot'}


def complex(real, imag=0.0):
    return magic(r=real, i=imag)


my_list = [1, 2, 3]
another_list = [4, 5, 6]
combined_list =  my_list + another_list
```

Kommentarer

```
# Comments start with a space after the comment symbol. Use complete
# sentences and proper grammar when writing comments. Comments should
# be in English unless you are certain the readers will *not* be
# English speaking.

# Long flowing text should be kept to under 72 characters like above.

x = 5  # Use inline comments sparingly.
```

Rekommendationer

Bra

```
if isinstance(obj, int):

if my_variable is None

if not my_list

if boolean_value
```

Dåligt

```
if type(obj) is type(1)

if my_variable == None

if not len(my_list)

if boolean_value == True
```

Kontrollera din kod

http://pep8online.com/

Eller installera pep8 genom *pip*

Automatisera PEP8

http://stackoverflow.com/questions/14328406/tool-to-convert-python-code-to-be-pep8-compliant

Cheat sheet

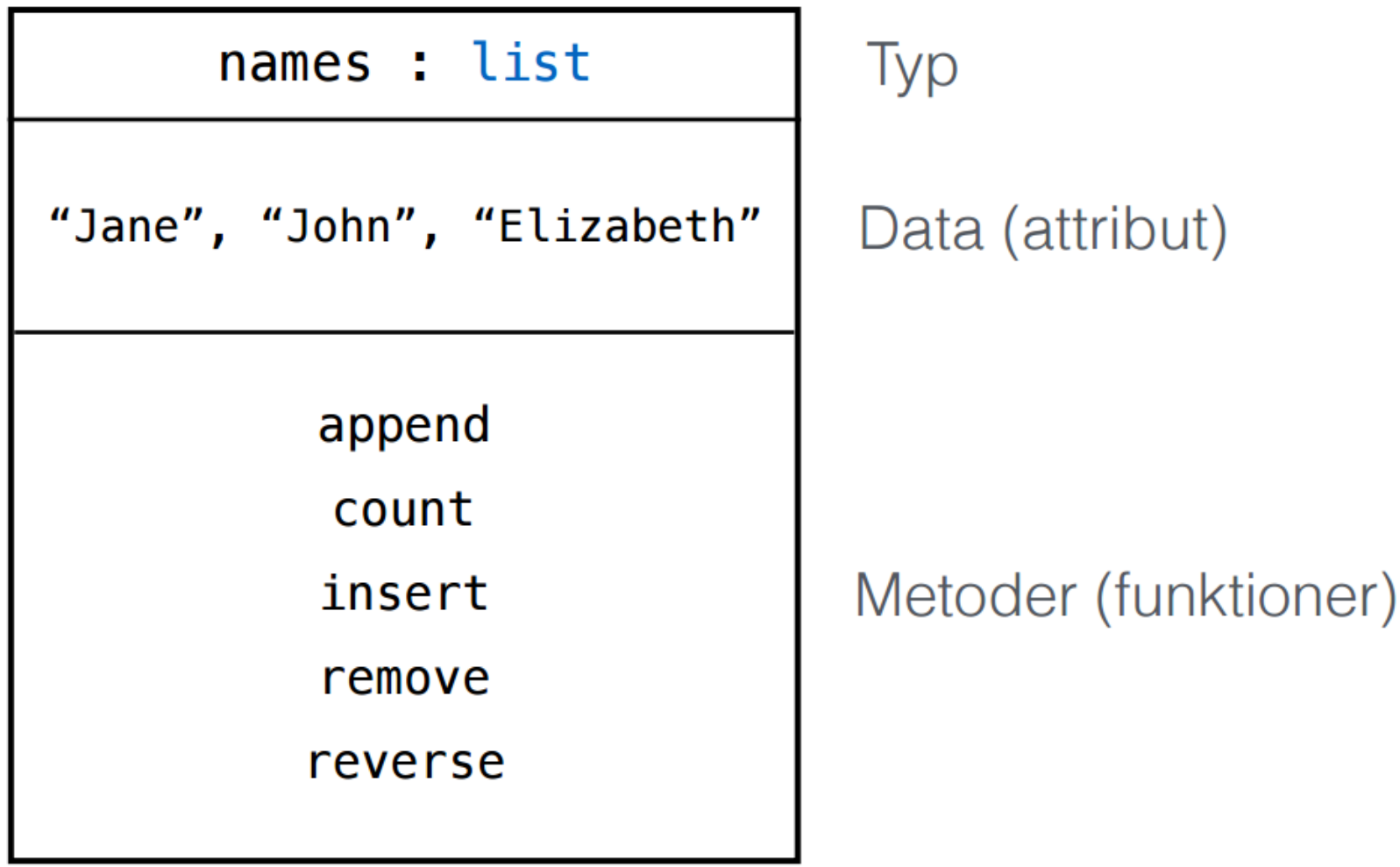
https://gist.github.com/RichardBronosky/454964087739a449da04

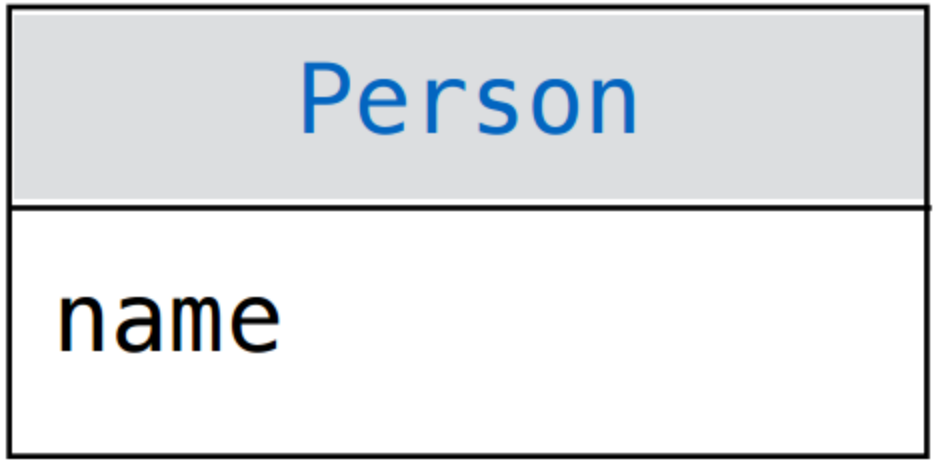
OOP med Python

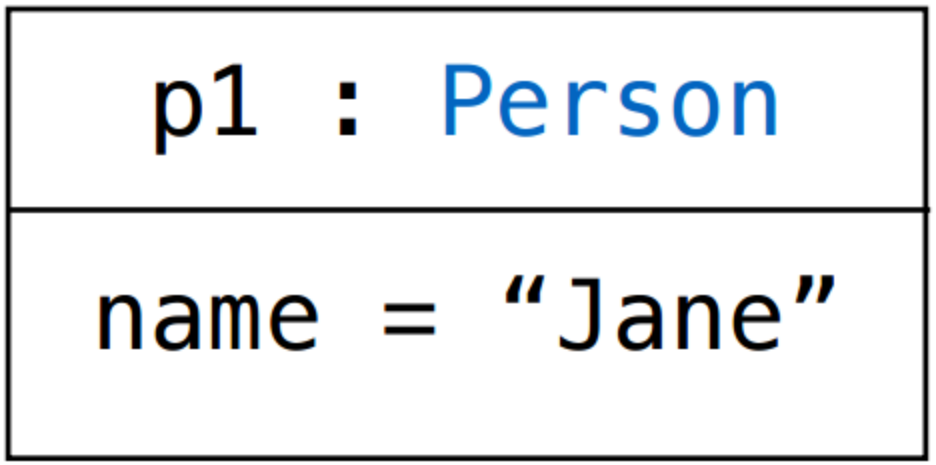
Mer om OOP

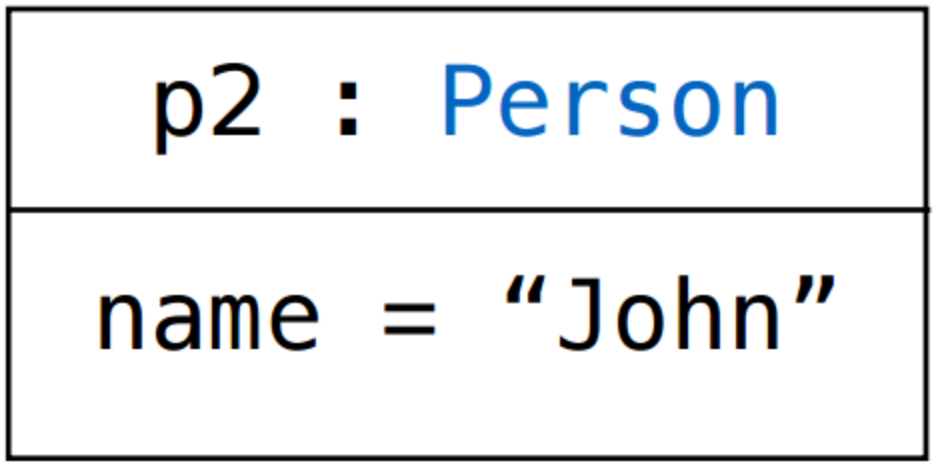
A class is code that specifies data attributes and methods for a particular type of data.

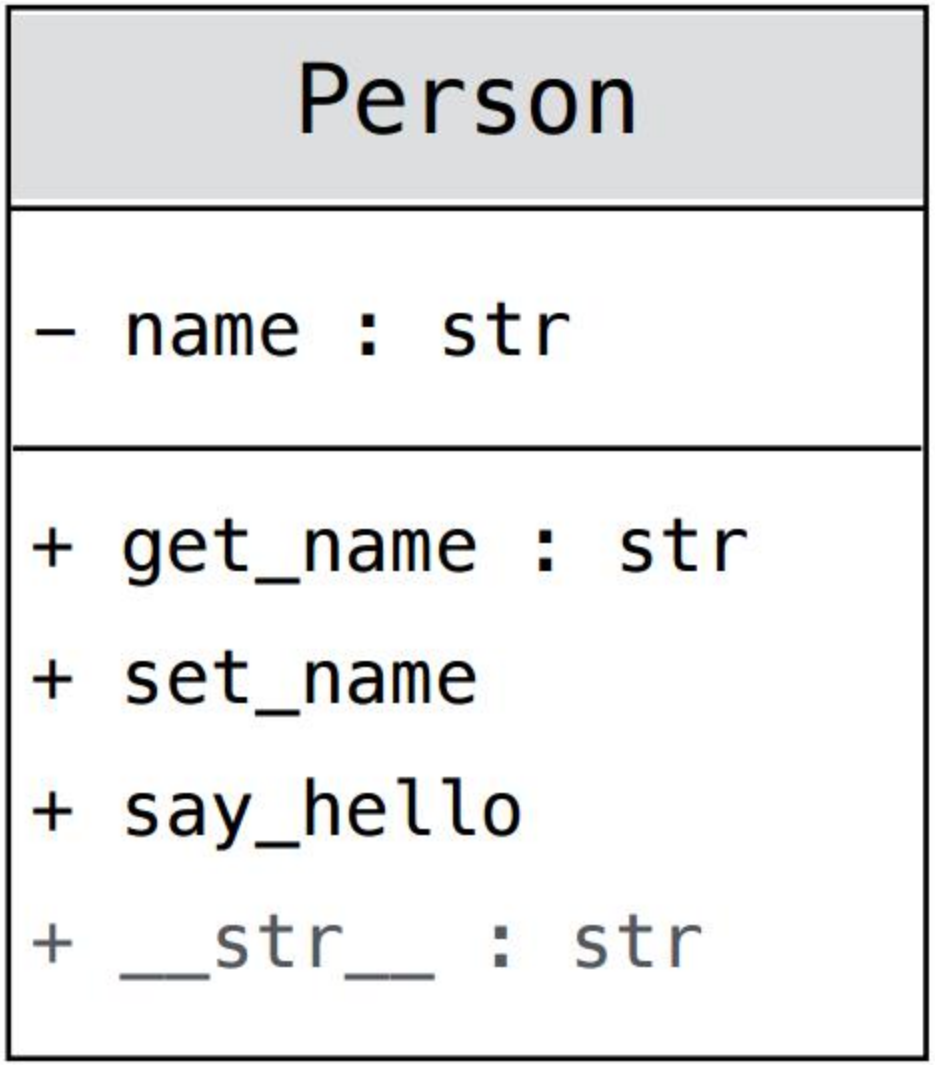
```
names = [“Jane”, “John”, “Elizabeth”]
```




Person
name

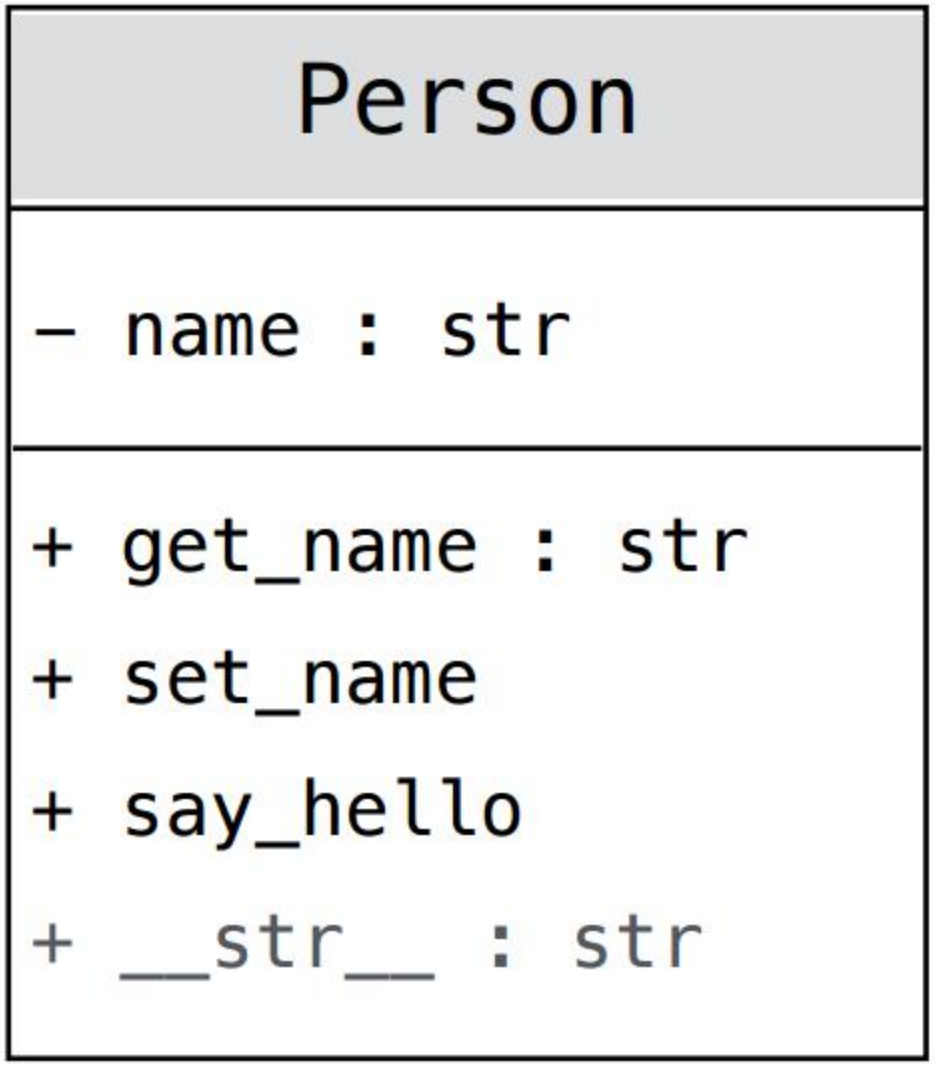

p1 : Person
name = “Jane”


p2 : Person
name = “John”




```python
class Person(object):

    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_name(self, name):
        self.name = name

    def say_hello(self):
        print self.name, "says hello!"

    def __str__(self):
        return self.name

```

Object-oriented programming is centered on objects. Objects are created from abstract data types that encapsulate data and function together.




```python
class Person(object):

    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_name(self, name):
        self.name = name

    def say_hello(self):
        print self.name, "says hello!"

    def __str__(self):
        return self.name

```

Inkapsling

Vem har tillgång till en klass attribut och metoder?

Varför inkapsling?

1. Vi vill ha kontroll på vem som får använda våra funktioner / attribut
2. Vi vill kontrollera hur våra attribut modifieras, t.ex.
    1. Validera ett värde
    2. Välja hur våra attribut får användas
        1. Läs / skrivbara attribut
        2. Välja hur attribut ska returneras

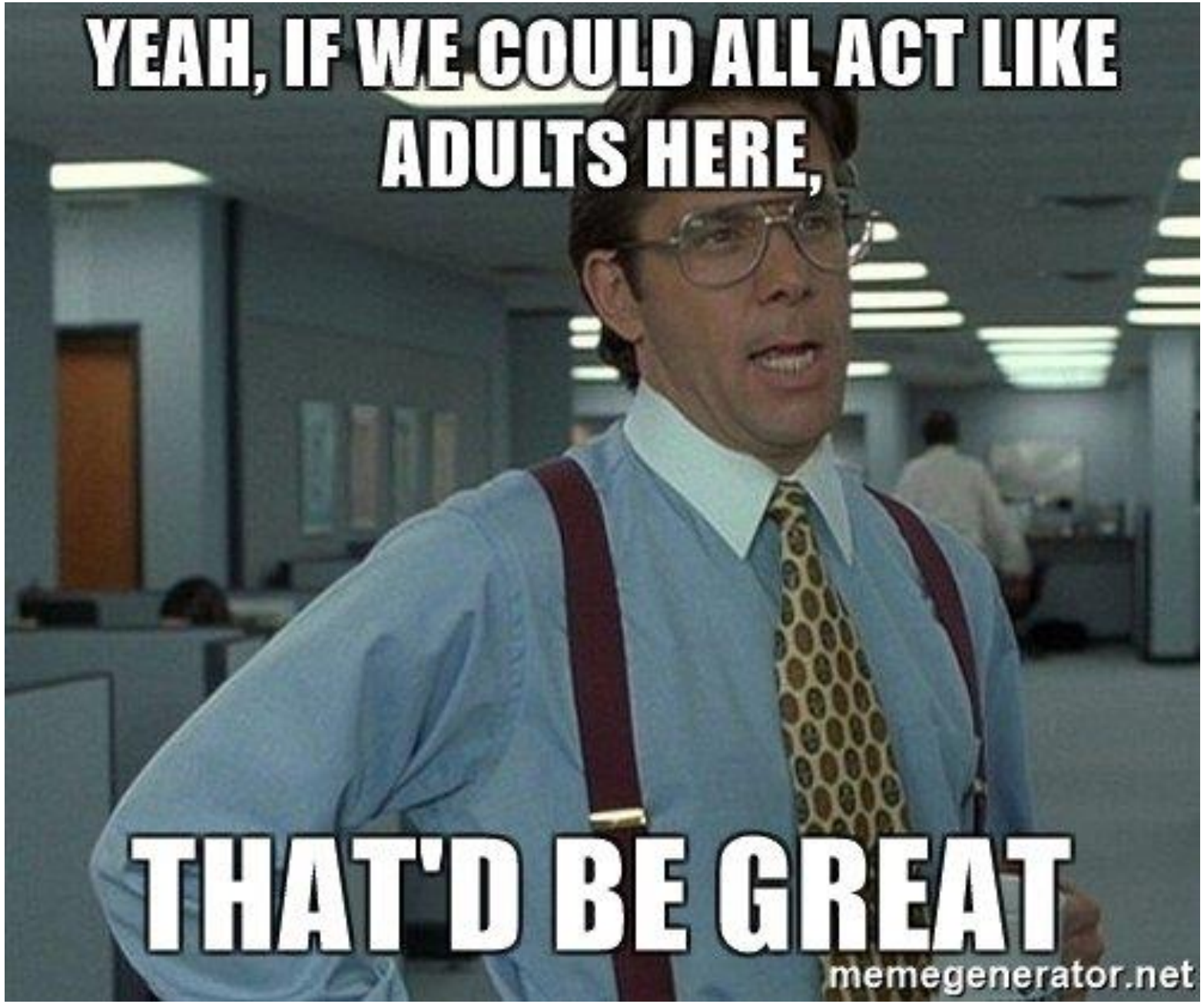
- Objektet har ett gränssnitt — en tydlig definition över **vad** som kan göras.
- Exakt **hur** saker och ting utförs spelar ingen roll utifrån.
- Men objektet måste ha **kontroll** över sitt tillstånd.

Exempel

Test.py - C:/Users/TSANTI/Desktop/Test.py (3.5.2)

File Edit Format Run Options Window Help

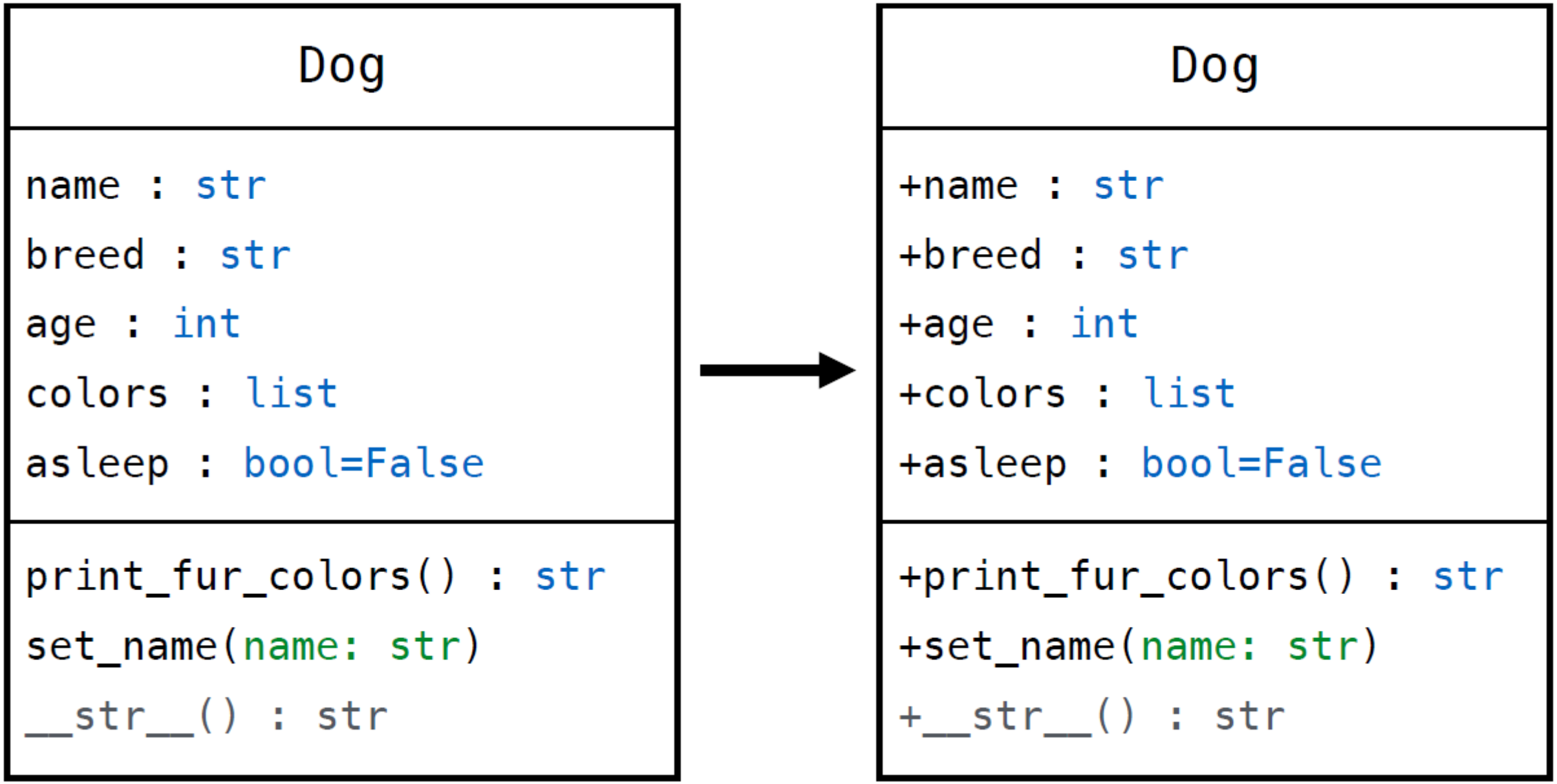
```
from datetime import datetime

class Kalle:

    def __init__(self):
        self.__personnr = "19941128-1234"

    def get_age(self):
        today = datetime.today()
        age = today.year - int(self.__personnr[0:4])
        if int(self.__personnr[4:6]) > today.month:
            age = age -1
        elif int(self.__personnr[4:6]) == today.month:
            if int(self.__personnr[6:8]) > today.day:
                age = age -1

        print(age)


kalle = Kalle()
kalle.get_age()
print(kalle.__personnr)
```


YEAH, IF WE COULD ALL ACT LIKE ADULTS HERE,
THAT'D BE GREAT
memegenerator.net

Test.py - C:/Users/TSANTI/Desktop/Test.py (3.5.2)

File Edit Format Run Options Window Help

```
from datetime import datetime

class Kalle:

    def __init__(self):
        self.__personnr = "19941128-1234"

    def get_age(self):
        today = datetime.today()
        age = today.year - int(self.__personnr[0:4])
        if int(self.__personnr[4:6]) > today.month:
            age = age -1
        elif int(self.__personnr[4:6]) == today.month:
            if int(self.__personnr[6:8]) > today.day:
                age = age -1

        print(age)


kalle = Kalle()
kalle.get_age()
print(kalle._Kalle__personnr)
print(kalle.__personnr)
```


Dog
name : str
breed : str
age : int
colors : list
asleep : bool=False
print_fur_colors() : str
set_name(name: str)
__str__() : str
Dog
+name : str
+breed : str
+age : int
+colors : list
+asleep : bool=False
+print_fur_colors() : str
+set_name(name: str)
+__str__() : str

Specialmetoder i klasser

Specialmetoder

- `__eq__`, equals, `==`
- `__lt__`, lesser then, `<`
- `__gt__`, greater then, `>`

```python
# Class definition
class Dog(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        """Equals"""
        return self.age == other.age

    def __lt__(self, other):
        """Lesser then"""
        return self.age < other.age

    def __gt__(self, other):
        """Greater then"""
        return self.age > other.age

doug = Dog("Doug", 8)
watson = Dog("Watson", 12)

print doug == watson
# => False
print doug > watson
# => False
print doug < watson
# => True
```

```python
# Class definition
class Dog(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        """Equals"""
        return self.age == other.age

    def __lt__(self, other):
        """Lesser then"""
        return self.age < other.age

    def __gt__(self, other):
        """Greater then"""
        return self.age > other.age

doug = Dog("Doug", 8)
watson = Dog("Watson", 12)

print doug == watson
# => False
print doug > watson
# => False
print doug < watson
# => True
```

Attribut i klasser

Alla attribut måste inte anges i konstruktor

```python
import datetime # we will use this for date objects

class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        return age
```

```python
import datetime # we will use this for date objects

class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        return age

person = Person(
    "Anton",
    "Tibblin",
    datetime.date(1989, 10, 2), # year, month, day
    "Måsvägen 12B",
    "070-533 74 76",
    "anton.tibblin@mah.se"
)

print(person.name)
print(person.email)
print(person.age())
```

```
========== RESTA
Anton
anton.tibblin@mah.se
27
>>>
```

```python
import datetime # we will use this for date objects

class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def get_email2(self):
        if hasattr(self, "email2"):
            return self.email2
        else:
            self.email2 = input("Please enter email2: ")
            return self.email2

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        return age

person = Person(
    "Anton",
    "Tibblin",
    datetime.date(1989, 10, 2), # year, month, day
    "Måsvägen 12B",
    "070-533 74 76",
    "anton.tibblin@mah.se"
)

print(person.name)
print(person.get_email2())
print(person.get_email2())
```

```
================== RESTART: C:/Users/TSA
Anton
Please enter email2: anton2@mah.se
anton2@mah.se
anton2@mah.se
```

```python
import datetime # we will use this for date objects

class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def get_email2(self):
        if hasattr(self, "email2"):
            return self.email2
        else:
            self.email2 = input("Please enter email2: ")
            return self.email2

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        return age

person = Person(
    "Anton",
    "Tibblin",
    datetime.date(1989, 10, 2), # year, month, day
    "Måsvägen 12B",
    "070-533 74 76",
    "anton.tibblin@mah.se"
)

print(person.name)
print(person.get_email2())
print(person.get_email2())
```

```
Anton
Please enter email2: anton2@mah.se
anton2@mah.se
anton2@mah.se
>>> person.pets =["cat", "dog"]
>>> person.pets
['cat', 'dog']
```

Hämta flera attribut

```python
import datetime # we will use this for date objects

class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def get_email2(self):
        if hasattr(self, "email2"):
            return self.email2
        else:
            self.email2 = input("Please enter email2: ")
            return self.email2

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        return age
```

```python
person = Person(
    "Anton",
    "Tibblin",
    datetime.date(1989, 10, 2), # year, month, day
    "Måsvägen 12B",
    "070-533 74 76",
    "anton.tibblin@mah.se"
)

# Get following attributs of the person
for key in ["name", "birthdate", "email"]:
    attr = getattr(person, key)
    print(attr)

#=> Anton
#=> 1989-10-02
#=> anton.tibblin@mah.se
```

```python
getattr(myobject, "a")

# means the same thing as

myobject.a
```

Klassattribut

```
class Person:

    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, title, name, surname):
        if title not in self.TITLES:
            raise ValueError("%s is not a valid title." % title)

        self.title = title
        self.name = name
        self.surname = surname
```

```
# we can access a class attribute from an instance
person.TITLES

# but we can also access it from the class
Person.TITLES
```

Class attributes can also sometimes be used to provide default attribute values:

```
class Person:
    deceased = False

    def mark_as_deceased(self):
        self.deceased = True
```

Skillnad mellan attribut (instans) och klassattribut

```
class Person:
    pets = []

    def add_pet(self, pet):
        self.pets.append(pet)

jane = Person()
bob = Person()

jane.add_pet("cat")
print(jane.pets)
print(bob.pets) # oops!
```

```
class Person:

    def __init__(self):
        self.pets = []

    def add_pet(self, pet):
        self.pets.append(pet)

jane = Person()
bob = Person()

jane.add_pet("cat")
print(jane.pets)
print(bob.pets)
```

Klassmetoder

```
class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        # (...)

    @classmethod
    def from_text_file(cls, filename):
        # extract all the parameters from the text file
        return cls(*params) # this is the same as calling Person(*params)
```

```
class Dog:

    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    @classmethod
    def create_from_string(cls, string):
        parts = string.split(" ")
        return cls(parts[0], parts[1])
```

```
>>> d1 = Dog("Doug", "Pug")
>>> d2 = Dog.create_from_string("Steve Pug")
>>> d1.name
'Doug'
>>> d2.name
'Steve'
>>>
```

Statiska metoder

```python
class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def fullname(self): # instance method
        # instance object accessible through self
        return "%s %s" % (self.name, self.surname)

    @classmethod
    def allowed_titles_starting_with(cls, startswith): # class method
        # class or instance object accessible through cls
        return [t for t in cls.TITLES if t.startswith(startswith)]

    @staticmethod
    def allowed_titles_ending_with(endswith): # static method
        # no parameter for class or instance object
        # we have to use Person directly
        return [t for t in Person.TITLES if t.endswith(endswith)]


jane = Person("Jane", "Smith")

print(jane.fullname())

print(jane.allowed_titles_starting_with("M"))
print(Person.allowed_titles_starting_with("M"))

print(jane.allowed_titles_ending_with("s"))
print(Person.allowed_titles_ending_with("s"))
```

```
========== RESTART:
Jane Smith
['Mr', 'Mrs', 'Ms']
['Mr', 'Mrs', 'Ms']
['Mrs', 'Ms']
['Mrs', 'Ms']
>>>
```

Lägga till funktioner som ”extra” attribut

```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    @property
    def fullname(self):
        return "%s %s" % (self.name, self.surname)

jane = Person("Jane", "Smith")
print(jane.fullname) # no brackets!
```

```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    @property
    def fullname(self):
        return "%s %s" % (self.name, self.surname)

    @fullname.setter
    def fullname(self, value):
        # this is much more complicated in real life
        name, surname = value.split(" ", 1)
        self.name = name
        self.surname = surname

    @fullname.deleter
    def fullname(self):
        del self.name
        del self.surname

jane = Person("Jane", "Smith")
print(jane.fullname)

jane.fullname = "Jane Doe"
print(jane.fullname)
print(jane.name)
print(jane.surname)
```

Mer om *args, **kwargs

http://stackoverflow.com/questions/3394835/args-and-kwargs

179

Here's an example that uses 3 different types of parameters.

```
def func(required_arg, *args, **kwargs):
    # required_arg is a positional-only parameter.
    print required_arg

    # args is a tuple of positional arguments,
    # because the parameter name has * prepended.
    if args: # If args is not empty.
        print args

    # kwargs is a dictionary of keyword arguments,
    # because the parameter name has ** prepended.
    if kwargs: # If kwargs is not empty.
        print kwargs

>>> func()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: func() takes at least 1 argument (0 given)

>>> func("required argument")
required argument

>>> func("required argument", 1, 2, '3')
required argument
(1, 2, '3')

>>> func("required argument", 1, 2, '3', keyword1=4, keyword2="foo")
required argument
(1, 2, '3')
{'keyword2': 'foo', 'keyword1': 4}
```

One place where the use of `*args` and `**kwargs` is quite useful is for subclassing.

```
class Foo(object):
    def __init__(self, value1, value2):
        # do something with the values
        print value1, value2

class MyFoo(Foo):
    def __init__(self, *args, **kwargs):
        # do something else, don't care about the args
        print 'myfoo'
        super(MyFoo, self).__init__(*args, **kwargs)
```

Ändringsbara typer

MUTABILITY OF COMMON TYPES

The following are some **immutable** objects:

- int
- float
- decimal
- complex
- bool
- string
- tuple
- range
- frozenset
- bytes

The following are some **mutable** objects:

- list
- dict
- set
- bytearray
- user-defined classes (unless specifically made immutable)

Ex! Ett litet program

Lag & spelare

Mer om Arv

T.ex. ORM

http://docs.peewee-orm.com/en/latest/