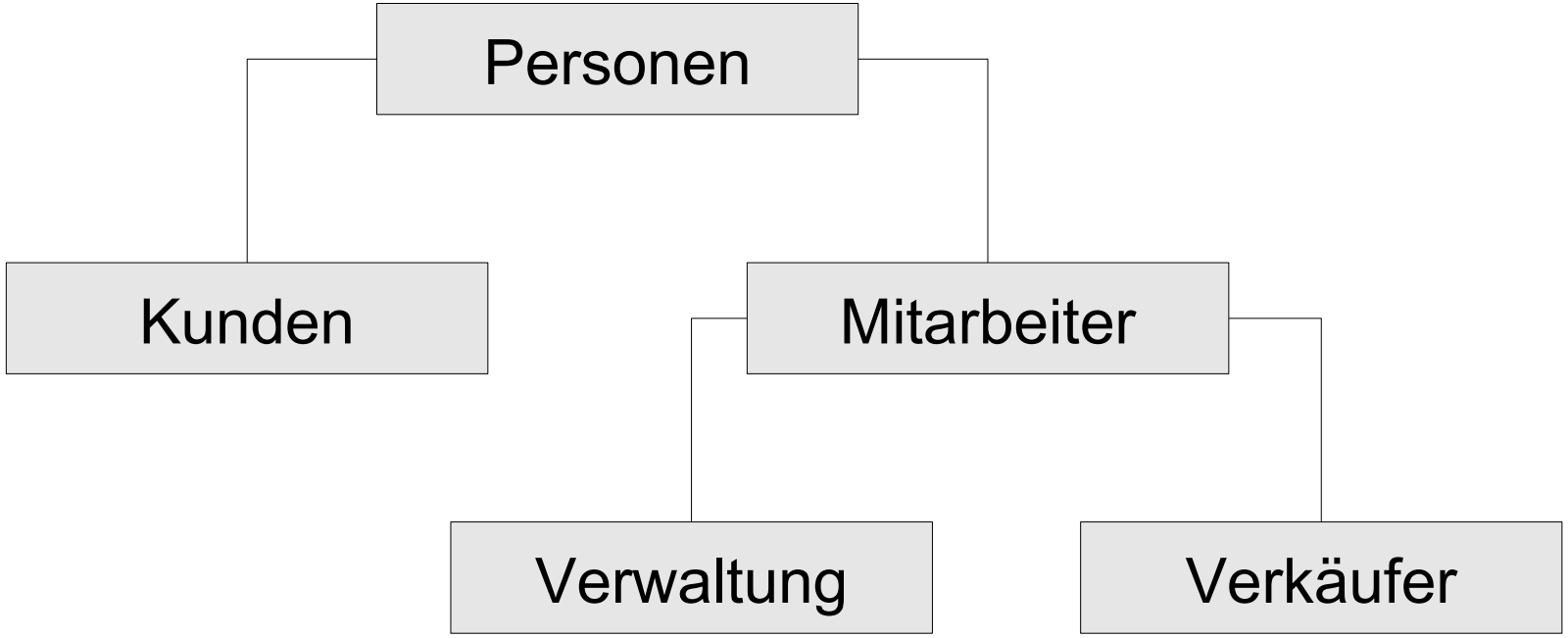
Python

„Vererbung“



Vererbung

- Definition von Klassen auf Basis von bestehenden Klassen.
- Ableitung einer Klasse von einer anderen.
- Darstellung von Klassen in einem hierarchischen Modell.
- Eltern-Kind-Beziehung.
- Implementierung von „ist ein“-Beziehungen.

… in Python

```
class ClsPerson(object):
        def __init__(self, nachname, anrede, vorname = ""):
            self.anrede = anrede
            self.vorname = vorname
            self.nachname = nachname
            self.oldNachname = ""
```

```
from enum import Enum

class Fehlertyp(Enum):
    WARNUNG = 100
    EXCEPTION = 200
    ERROR = 300
```

Klassenkopf

class	clsPerson	(	object	)	:
class	Fehlertyp	(	Enum	)	:
class	klasse	(	Basisklasse	)	:

- Jede Klasse in Python erbt mindestens von einer Klasse.
- Die Klasse, die als Basis dient, wird in den runden Klammern angegeben.
- Die Klasse bekommt alle Attribute und Methoden vererbt und kann diese verändern.

Basisklasse

- Oberklasse, Elternklasse, Superklasse.
- Allgemeine Beschreibung einer Gruppe von Objekten.
- Weitergabe von Attributen und Methoden.

Basisklasse „object“

- Jede Klasse erbt von der, in Python definierten Klasse `object`.
- Die Klasse `object` beschreibt allgemein den Lebenszyklus mit Hilfe der Methoden `__new__()`, `__init__()` und `__del__()`.
- Jede der Methoden kann von untergeordneten Klasse überschrieben werden.

Klasse erbt von „Enumeration“

```
from enum import Enum


class Fehlertyp(Enum):
    WARNUNG = 100
    EXCEPTION = 200
    ERROR = 300
```

Erläuterung

- Von (from) dem Modul enum wird die Klasse Enum importiert (import).
- Diese Klasse wird als Basisklasse für eine benutzerdefinierte Aufzählung genutzt.
- Die Aufzählung definiert eigene Laufzeitfehler.

Definition der Laufzeitfehler

```
from enum import Enum


class Fehlertyp(Enum):
    WARNUNG = 100
    EXCEPTION = 200
    ERROR = 300
```

- Die Elemente der Aufzählung werden als Klassenvariable definiert.
- Klassenvariablen können unabhängig von der Erzeugung einer Instanz genutzt werden.

Definition einer eigenen Fehlerklasse

```
from basFehlertyp import Fehlertyp

class Fehlerklasse(Exception):

    def __init__(self, msg, typ = Fehlertyp.WARNUNG):
        self.msg = msg
        self.typ = typ

    def __str__(self):
        errorTyp = self.typ
        return(errorTyp.name + ": " + repr(self.msg))
```

Aufbau des Klassenkopfs

class	Fehlerklasse	(	Exception	)	:

- Der Klassenkopf beginnt mit dem Schlüsselwort class.
- Dem Klassennamen folgt eine beliebige Bezeichnung. Die Bezeichnung sollte den realen Gegenstand abbilden.
- Dem Klassennamen folgen runde Klammern. Die eigene Fehlerklasse erbt von der Basisklasse Exception. Die Klasse basiert auf der Definition der Ausnahmen in Python.
- Dem Klassennamen folgt der Doppelpunkt.

Initialisierungsmethode

```
def __init__(self, msg, objekt, typ = Fehlertyp.WARNUNG):
        self.msg = msg
        self.typ = typ
        self.objekt = objekt
```

- Von der Klasse Fehlerklasse wird eine Instanz erstellt.
- Die Attribute dieser Instanz wird mit Hilfe der Initialisierungsmethode definiert.
- In diesem Beispiel werden drei Attribute mit Informationen zu dem Fehler erzeugt.

Attribute

```
def __init__(self, msg, objekt, typ = Fehlertyp.WARNUNG):
    self.msg = msg
    self.typ = typ
    self.objekt = objekt
```

- Das Attribut msg speichert die gewünschte Fehlermeldung.
- Das Attribut typ nutzt eine Enumeration, um die Fehlermeldung zu klassifizieren.
- Das Attribut objekt speichert Informationen über das auslösende Objekt.

Nutzung der Aufzählung

- Die Aufzählung wird mit Hilfe von `from … import` in das Modul eingebunden.
- Die Elemente der Aufzählung werden über einen unqualifizierten Namen in der Klasse `Fehlerklasse` aufgerufen.
- In der Initialisierungsmethode wird Klassenvariable `Fehlertyp.WARNUNG` als Standardwert für ein Parameter genutzt. Der Parameter legt fest, welche Fehlermeldung ausgegeben wird.

Magische Methoden

- Automatisierter Aufruf durch Python.
- Methoden, die mit zwei Unterstrichen beginnen und enden.
- Zum Beispiel __init__(), __str__().

… str()

```
def __str__(self):
        errorTyp = self.typ
        return(errorTyp.name + ": " + self.msg +
                " von dem Objekt " + repr(self.objekt))
```

- Die Methode str() zur Konvertierung eines beliebigen Datentyps in einem String wird überschrieben.
- In diesem Beispiel gibt die Funktion eine Fehlermeldung einer Instanz zurück, die mit Hilfe von print() in einer Shell angezeigt werden kann.

Definition der Fehlermeldung

```
def __str__(self):
        errorTyp = self.typ
        return(errorTyp.name + ": " + self.msg +
                " von dem Objekt " + repr(self.objekt))
```

- Die Variable errorTyp.name gibt den Namen eines Elements in der Aufzählung zurück.
- Die Funktion repr(self.objekt) repräsentiert Informationen zu dem Objekt self.objekt als String.
- Die Fehlermeldung self.msg wird im String angezeigt.

Nutzung der Fehlermeldungen

```
import basPerson
import basFehlerklasse

def start():
    personA = None
    personB = None

    try:
        personB = basPerson.ClsPerson(None, "Herr")
        print("Person B:")
        print(personB.getPersonInfo())

    except basFehlerklasse.Fehlerklasse as e:
        print(e)
```

Fehlerbehandlung (Exception-Handling)

try:
  personA =
  basPerson.ClsPerson("Person", "Herr")
	Versuche ...

except basFehlerklasse.Fehlerklasse as e: print("Benutzerdefinierter Fehler")	Abfangen von speziellen Fehlern
except: print("x beliebiger Laufzeitfehler")	Abfangen von allen anderen Fehlern
else: print("Durchführung der Anweisung")	Wenn kein Fehler aufgetreten ist ...
finally: print("Immer ausführen")	Wird immer ausgeführt

Versuche die Anweisungen auszuführen

```
try:
    personA = basPerson.ClsPerson("Person", "Herr")
    print("Person A:")
    print(personA.getPersonInfo())
```

- Beginn mit dem Schlüsselwort try.
- Dem Schlüsselwort folgt der Doppelpunkt. In der nächsten Zeile beginnt der, zu dem Befehl gehörende Codeblock.
- In dem Codeblock können Laufzeitfehler auftreten, müssen aber nicht.

Abfangen aller Laufzeitfehler

```
try:
    personA = basPerson.ClsPerson("Person", "Herr")

except:
    print("Ein unbekannter Fehler ist aufgetreten")
```

- Dem Schlüsselwort except folgt der Doppelpunkt. In der nächsten Zeile beginnt der, zu dem Befehl gehörende Codeblock. Der Codeblock wird entsprechend eingerückt.
- Der Codeblock behandelt alle Laufzeitfehler, die nicht explizit abgefangen werden.

Abfangen von benutzerdefinierten Laufzeitfehler

```
try:
    personA = basPerson.ClsPerson("Person", "Herr")

except basFehlerklasse.Fehlerklasse as e:
        print(e)
```

- Dem Schlüsselwort except folgt die Bezeichnung des abzufangenden Fehlers.
- In diesem Beispiel werden alle Fehler der Klasse Fehlerklasse aus dem Modul basFehlerklasse behandelt. Das Modul ist in dieser Codedatei eingebunden.

Nutzung eines Alias-Namen

```
try:
    personA = basPerson.ClsPerson("Person", "Herr")

except basFehlerklasse.Fehlerklasse as e:
        print(e)
```

- Mit Hilfe von as kann der benutzerdefinierten Fehlerklasse ein Alias zugewiesen werden.
- Durch Überschreiben der String-Funktion in dieser Fehlerklasse kann direkt über den Alias die Fehlermeldung ausgedruckt werden.

Auslösen eines Fehlers

```
raise basFehlerklasse.Fehlerklasse("Nachname fehlt", self,
                                   basFehlertyp.Fehlertyp.ERROR)
```

- Der Befehl raise löst einen Fehler aus. Die Ausnahme wird an den Aufrufer weitergereicht und kann dort behandelt werden.
- In diesem Beispiel wird eine Exception der Klasse Fehlerklasse in dem Modul basFehlerklasse erzeugt und initialisiert.

Benutzerdefinierte Basisklasse „Person“

Person
Vorname Nachname Straße Postleitzahl Ort
Lese Info Briefkopf Setze Nachname

… in Python

```
class ClsPerson(object):
    def __init__(self, nachname, anrede, vorname = ""):
        self.anrede = anrede
        self.vorname = vorname
        self.nachname = nachname
        self.oldNachname = ""

    def getPersonInfo(self):
        personName = self.nachname

        if self.vorname != "":
            personName = self.vorname + " " + self.nachname

        return personName
```

Aufbau

class ClsPerson(object):	Klassenkopf

def __init__(self, nachname, anrede, vorname = ""):
self.anrede = anrede
self.vorname = vorname
self.nachname = nachname
self.oldNachname = ""
 	
def getPersonInfo(self):
pass
	Klassenrumpf

Aufbau des Klassenkopfs

class	ClsPerson	(	object	)	:

- Der Klassenkopf beginnt mit dem Schlüsselwort class.
- Dem Klassennamen folgt eine beliebige Bezeichnung. Die Bezeichnung sollte den realen Gegenstand abbilden.
- Dem Klassennamen folgen runde Klammern. In den runden Klammern werden alle Basisklassen der Klasse aufgelistet. Die verschiedenen Elternklassen werden durch ein Komma getrennt.
- Dem Klassennamen folgt der Doppelpunkt.

… erbt von ...

class	ClsPerson	(	object	)	:

- Seit Python 3.x wird „new-style-class“ verwendet.
- Jede Klasse in Python erbt Attribute und Methoden von irgendeiner anderen Klasse.
- Basisklassen, die von keiner benutzerdefinierten Klasse erben, erben von der Klasse `object`. Diese Klasse ist in Python definiert.

Subklasse

- Unterklasse, Kindklasse.
- Eltern-Kind-Beziehung.
- Abgeleitete Klasse von ein oder mehreren Basisklassen.
- Erweiterung oder Spezialisierung der Basisklasse.

Subklassen

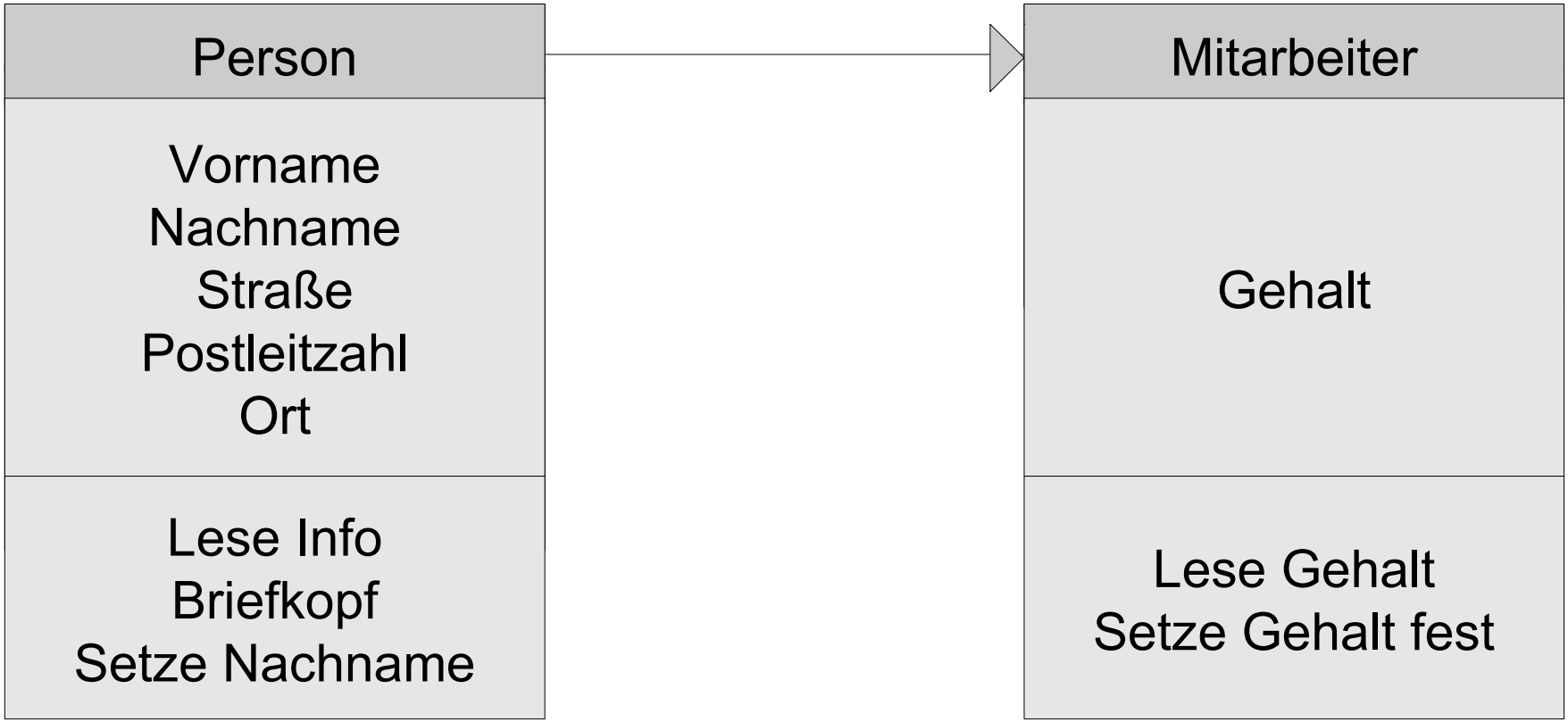
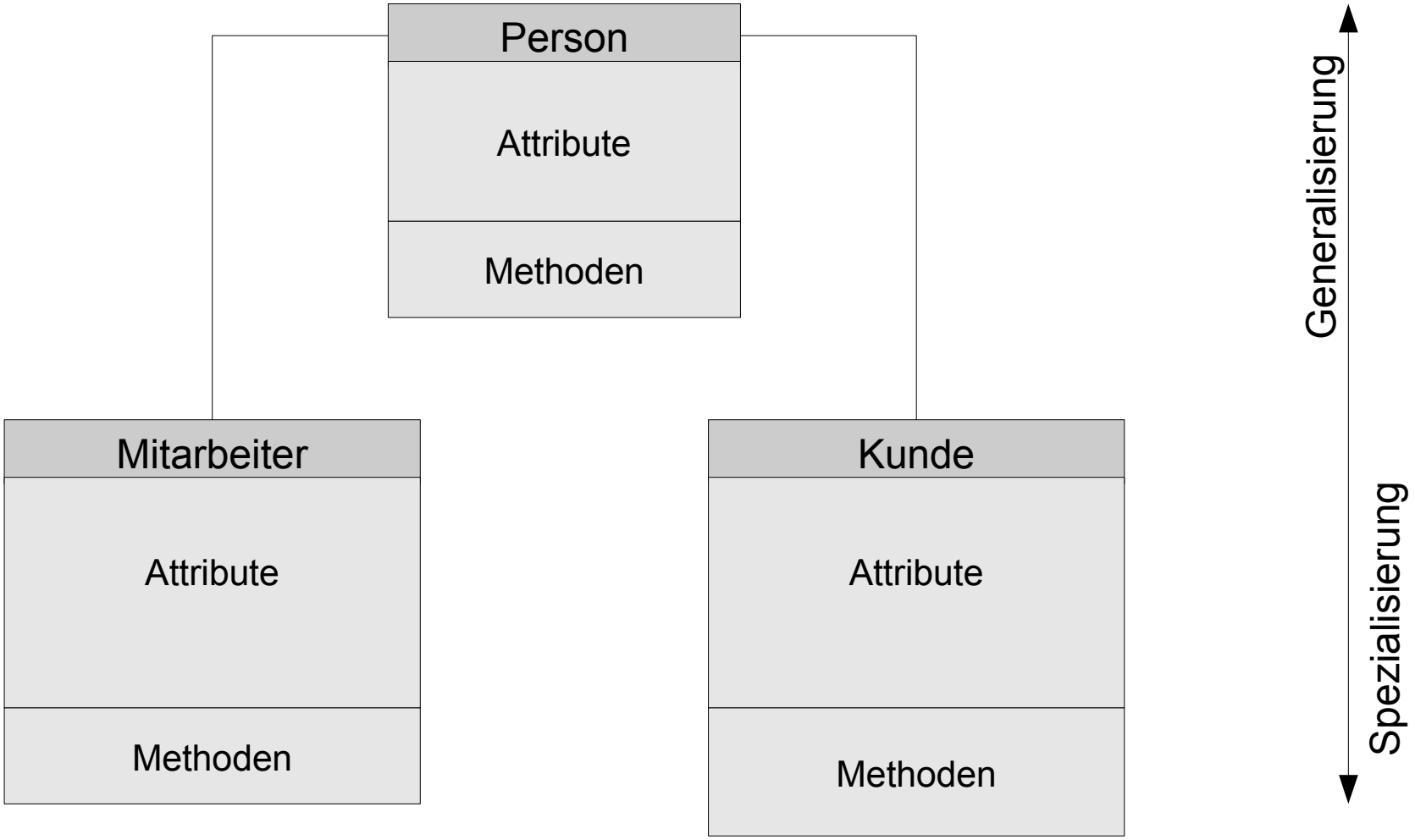

Person
Vorname
Nachname
Straße
Postleitzahl
Ort
Lese Info
Briefkopf
Setze Nachname
Mitarbeiter
Gehalt
Lese Gehalt
Setze Gehalt fest

Abbildung als hierarchisches Modell



Generalisierung

- Von den Blättern zur Wurzel.
- Um so höher man in der Hierarchie geht, um so allgemeiner werden die Attribute und Methoden der Klasse.
- Klassen werden zu „Oberbegriffen“ zusammengefasst, die die einzelnen Klassen generell beschreiben.

Spezialisierung

- Von der Wurzel zu den Blättern.
- Um so tiefer man in der Hierarchie geht, um so spezieller werden die Attribute der Klasse.
- Die Klassen beschreiben Spezialfälle eines Oberbegriffs.
- Objekte einer Gruppe werden detailliert beschrieben.

Subklasse in Python

```
import basisklasse

class ClsMitarbeiter(basisklasse.ClsPerson):
        def __init__(self, nachname, anrede, vorname = "", gehalt = 1200):
                super().__init__(nachname, anrede, vorname)
                self.gehalt = gehalt

        def getGehalt(self):
            return self.gehalt

        def setGehalt(self, gehalt):
                self.gehalt = gehalt
```

Einbindung eines Moduls

```
import basisklasse
```

- Mit Hilfe des Schlüsselwortes import wird eine Codedatei in eine andere eingebunden.
- Dem Schlüsselwort import folgt der Dateiname eines Moduls. Der Interpreter von Python sucht diesen Dateinamen zuerst in dem momentan aktuellen Verzeichnis der Subklasse.
- Die gewünschte Basisklasse ist in dem angegebenen Modul definiert.

Klassenkopf der Subklasse

class	ClsMitarbeiter	(	basisklasse.ClsPerson	)	:

- Der Klassenkopf beginnt mit dem Schlüsselwort class.
- Dem Klassennamen folgt eine beliebige Bezeichnung. Die Bezeichnung sollte den realen Gegenstand abbilden.
- Dem Klassennamen folgen runde Klammern. In den runden Klammern werden alle Basisklassen der Subklasse aufgelistet
- Dem Klassennamen folgt der Doppelpunkt.

Nutzung von qualifizierten Namen

basisklasse.	.	ClsPerson
namespace	.	elment

- Falls das Modul vollständig eingebunden wurde, wird die Basisklasse mit Hilfe eines qualifizierten Namens angegeben.
- Links vom Punktoperator wird das Modul angegeben, in dem die gewünschte Basisklasse definiert ist.
- Rechts vom Punktoperator steht der Name der Basisklasse. Von dieser Klasse erbt die Subklasse.

Erzeugung der Subklasse

```
mitarbeiter = ClsMitarbeiter(nachname="nachname",
                             anrede = "Frau", gehalt=2300)
```

- Der Instanz mitarbeiter verweist auf ein Objekt vom Typ ClsMitarbeiter.
- Der Name der Klasse folgt eine Argumentliste in runden Klammern.

Argumentliste

```
mitarbeiter = ClsMitarbeiter(nachname="nachname",
                             anrede = "Frau", gehalt=2300)
```

- Die Argumentliste kann leer sein. Die Instanz wird mit den Standardwerten initialisiert.
- Die Argumentliste hat x Elemente. In diesem Beispiel werden Schlüssel-Wert-Paare zur Initialisierung genutzt. Die Schlüssel sind in der Initialisierungsmethode der Klasse ClsMitarbeiter definiert.

Initialisierungsmethode der Subklasse

```
def __init__(self, nachname, anrede, vorname = "",
                   Gehalt = 1200):

    super().__init__(nachname, anrede, vorname)
    self.gehalt = gehalt
```

- Die Initialisierungsmethode existiert nur einmal pro Klasse.
- Die Subklasse überschreibt die Initialisierungsmethode der Basisklasse.

Kopf der Initialisierungsmethode

```
def __init__(self, nachname,
                   anrede, vorname = "",
                   Gehalt = 1200):


    pass
```

- Jede Methode einer Klasse beginnt mit dem Schlüsselwort def.
- Die Initialisierungsmethode hat immer den Bezeichner __init__.
- Definition einer Parameterliste in den runden Klammern.
- Der Methodenkopf endet mit dem Doppelpunkt.

Methodenrumpf der Initialisierungsmethode

```
def __init__(self, nachname, anrede, vorname = "",
                   Gehalt = 1200):
        super().__init__(nachname, anrede, vorname)
        self.gehalt = gehalt
```

- Im Methodenrumpf wird zuerst die Basisklasse initialisiert.
- Anschließend werden alle Objektattribute der Subklasse definiert.

Aufruf des Basisklassen-Initialisierer

```
basisklasse.ClsPerson.__init__(self,
                               nachname,
                               anrede, vorname)
```

- In Python 2.x: Durch Angabe eines qualifizierten Namens wird die Basisklasse definiert.
- Der Punktoperator verbindet den Bezeichner der Basisklasse mit der darin definierten Initialisierungsmethode __init__ .
- Dem Namen der Methode folgt die Argumentliste in runden Klammern in Abhängigkeit der Definition der Initialisierungsmethode.

Parameter self

- Der erste Parameter beim Aufruf eines Basisklassen-Initialisierers muss immer `self` sein.
- Der Platzhalter `self` verweist auf eine Instanz von der Klasse.
- Der Name `self` ist eine Konvention. Es kann aber jeder beliebige Name genutzt werden.

… kann ab Python 3.x ersetzt werden durch...

```
super().__init__(nachname, anrede, vorname)
```

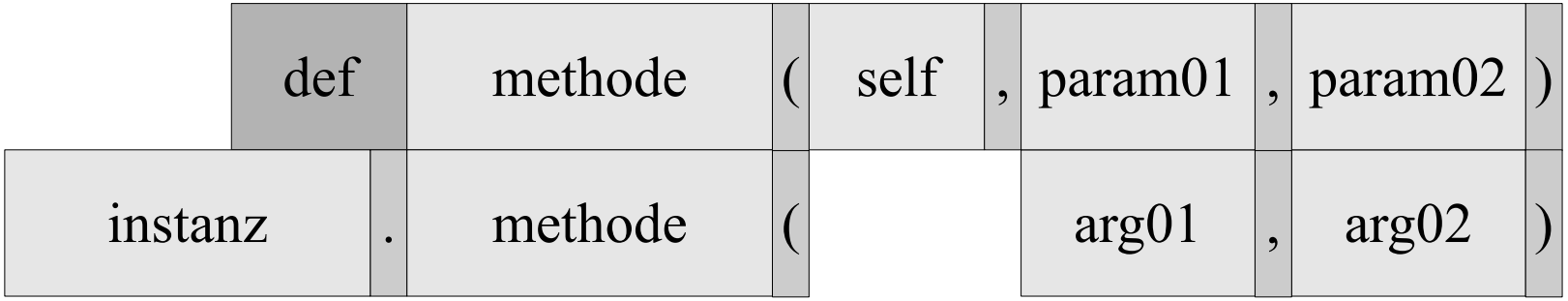
- Die Funktion super() liefert ein Verweis auf die Basisklasse.
- Die Parameterliste der Funktion ist leer.
- Passend zu der Basisklasse der Subklasse wird die Initialisierungsmethode __init__ aufgerufen.

Kompatible Variante zu Python 2.x

```
super(ClsMitarbeiter, self).__init__(nachname,
                                     anrede,
                                     vorname)
```

- Die Funktion super() liefert ein Verweis auf die Basisklasse der Subklasse.
- In der Parameterliste wird der Funktion als erstes Argument der Name der zu initialisierenden Subklasse übergeben. Zu dieser Subklasse wird die passende Basisklasse gesucht.
- Als zweites Argument wird der Funktion eine Instanz übergeben. In diesem Beispiel wird die Instanz durch den Platzhalter self festgelegt.

Aufruf von Objektmethoden



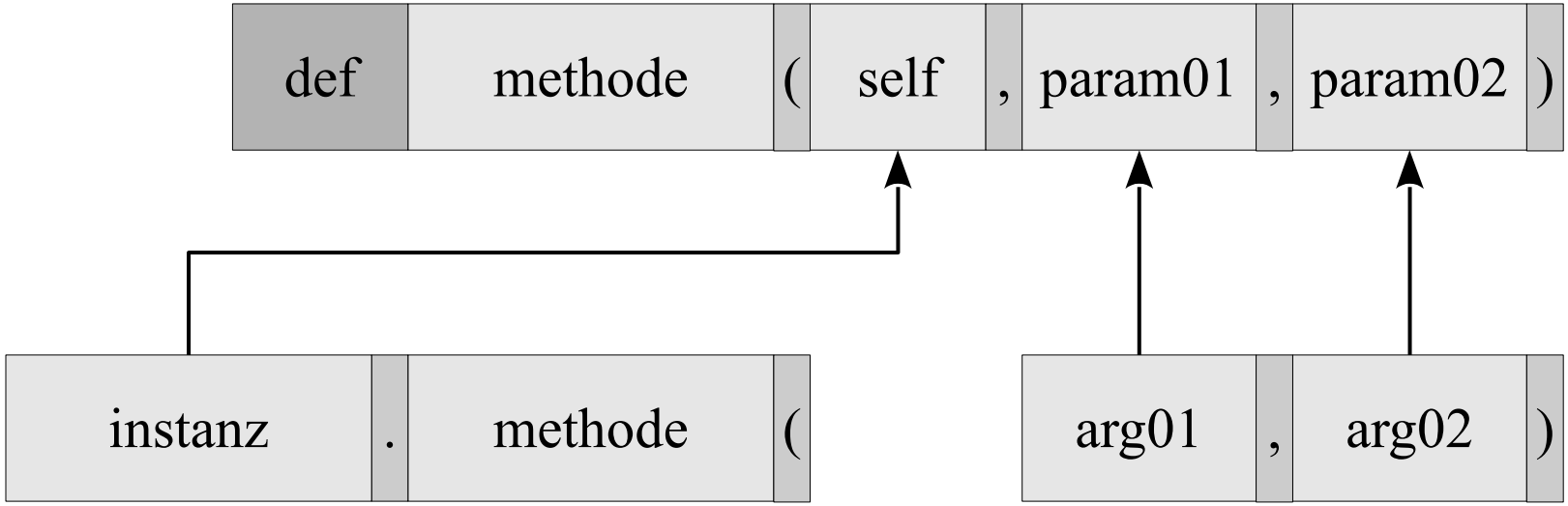
- Mit Hilfe des Namens wird eine Methode aufgerufen.
- Eine Objektmethode wird immer über die Instanz aufgerufen.
- Der Aufruf der Methode entspricht der definierten Signatur.
- Falls die Methode nicht in der Subklasse vorhanden ist, wird die passende Methode in der Basisklasse gesucht.

… über eine Instanz

	def	methode	(	self	,	param01	,	param02	)
instanz	.	methode	(			arg01	,	arg02	)

- Die Instanz muss vor Aufruf einer Methode deklariert werden.
- Der Punkt-Operator verbindet eine Instanz mit einer Methode.
- Eine Instanz basiert auf einer Klasse. In dieser Klasse ist die Methode definiert. Andernfalls wird der Laufzeitfehler *AttributeError: ... object has no attribute ...* angezeigt.

Parameterliste beim Aufruf



Überschreiben von Methoden

- In der Basisklasse definierte Methoden können von Methoden in der Subklasse überschrieben werden.
- Methoden der Basisklasse werden entsprechend der gewünschten Funktionalität in der Subklasse angepasst.
- Häufig wird die Methode `__init__` der Basisklasse überschrieben.

Die Methode aus der Basisklasse

```
class ClsPerson:

    def getPersonInfo(self):
        personName = self.nachname

        if self.vorname != "":
            personName = self.vorname + " " + self.nachname

        return personName
```

… wird in der Subklasse überschreiben

```
def getPersonInfo(self):
    personName = self.nachname
    ausgabe = ""

    if self.vorname != "":
        personName = self.vorname + " " + self.nachname

    if self.abteilung != "":
        ausgabe = "beschäftigt in der Abteilung " + self.abteilung

    ausgabe = personName + '\n' + ausgabe

    return ausgabe
```

Regeln

- Beide Methoden haben den gleichen Namen.
- Die Parameterliste der Methode in der Subklasse hat mindestens so viele Parameter wie die zu überschreibende Methode. Die Parameterliste in der Subklasse kann aber mehr Parameter haben.

Aufruf von Methoden aus der Basisklasse

```
personName = super().getPersonInfo()
```

- Die Funktion super() liefert ein Verweis auf die Basisklasse.
- Die Methode, rechts vom Punktoperator, ist in der Basisklasse definiert.
- In diesem Beispiel ist in der angegebenen Basisklasse eine Methode getPersonInfo() definiert. Die Parameterliste dieser Methode ist leer. Der Rückgabewert der Methode wird in einer Variablen gespeichert.

Abstrakte Klassen

- Vorlage für für Subklassen.
- Basisklasse, von der keine Instanz erzeugt werden kann.
- Beschreibung eines Oberbegriffes. Ein Mitarbeiter oder Kunde ist auch immer eine Person.
- Implementierung der Generalisierung von Klassen.

… in Python

```
import abc

class ClsPerson(metaclass = abc.ABCMeta):
    def __init__(self, nachname, anrede, vorname = ""):
        self.anrede = anrede
        self.vorname = vorname
        self.nachname = nachname

    @abc.abstractmethod
    def getPersonInfo(self):
        pass

    @abc.abstractmethod
    def getBriefkopf(self):
        pass
```

Import des Moduls „Abstract Base Class“

```
import abc
```

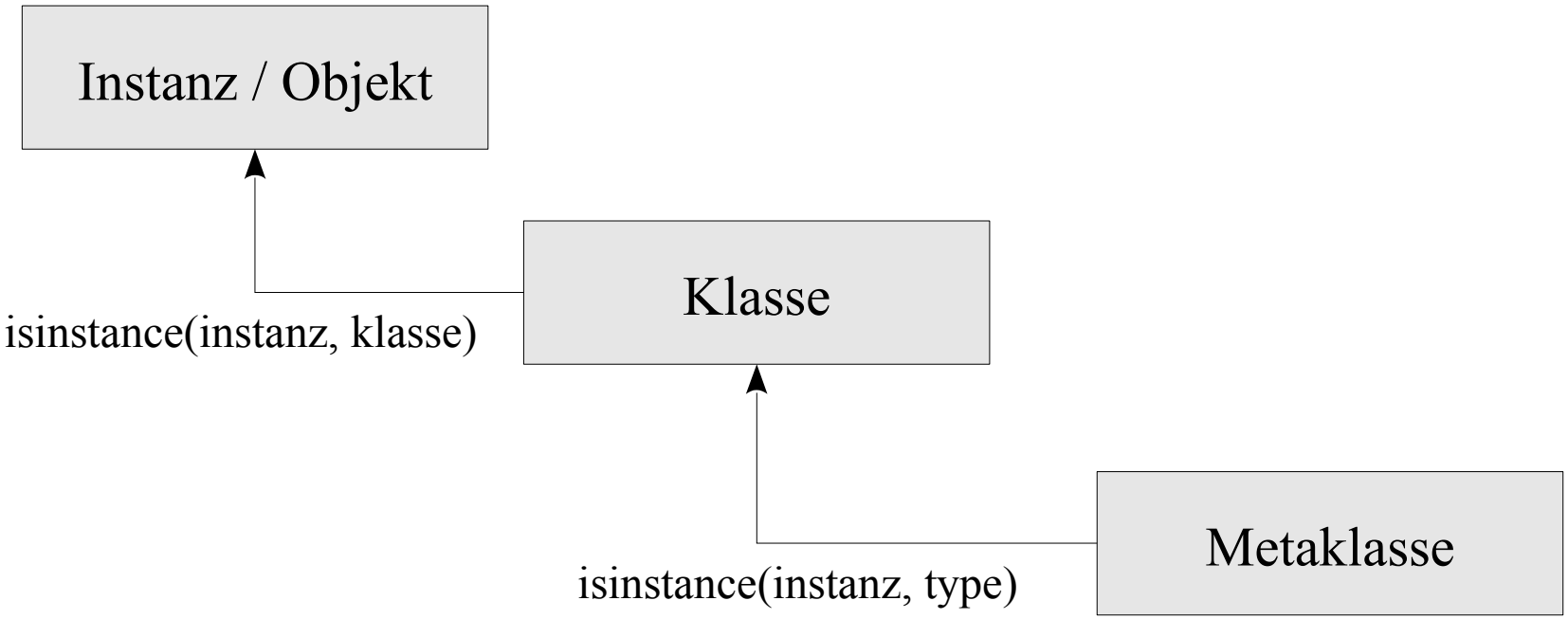
- Standardmäßig sind keine abstrakten Klassen in Python vorhanden,
- In dem Modul `abc` ist eine Vorlage für eine abstrakte Klasse definiert.

Kopf einer abstrakten Klasse

class	ClsPerson	(	metaclass = abc.ABCMeta	)	:

- In Python kann eine abstrakte Klasse von der Metaklasse abc.ABCMeta erben.

Metaklassen



- Klassen von Klassen.

Abstrakte Methoden

- Jede abstrakte Klasse enthält mindestens eine abstrakte Methode.
- Abstrakte Methoden haben einen undefinierten Methodenrumpf. Die Subklasse muss diese Methode implementieren.

Abstrakte Methode

```
@abc.abstractmethod
def getPersonInfo(self):
        pass
```

- In der Zeile vor dem Methodenkopf wird die Anweisung @abc.abstractmethod eingefügt.
- Nur der Methodenkopf wird definiert.
- Der Methodenrumpf enthält eine leere Anweisung (pass).