Universiteit Gent
Faculteit Wetenschappen

Programmeren in Python

inleiding tot objectgericht programmeren

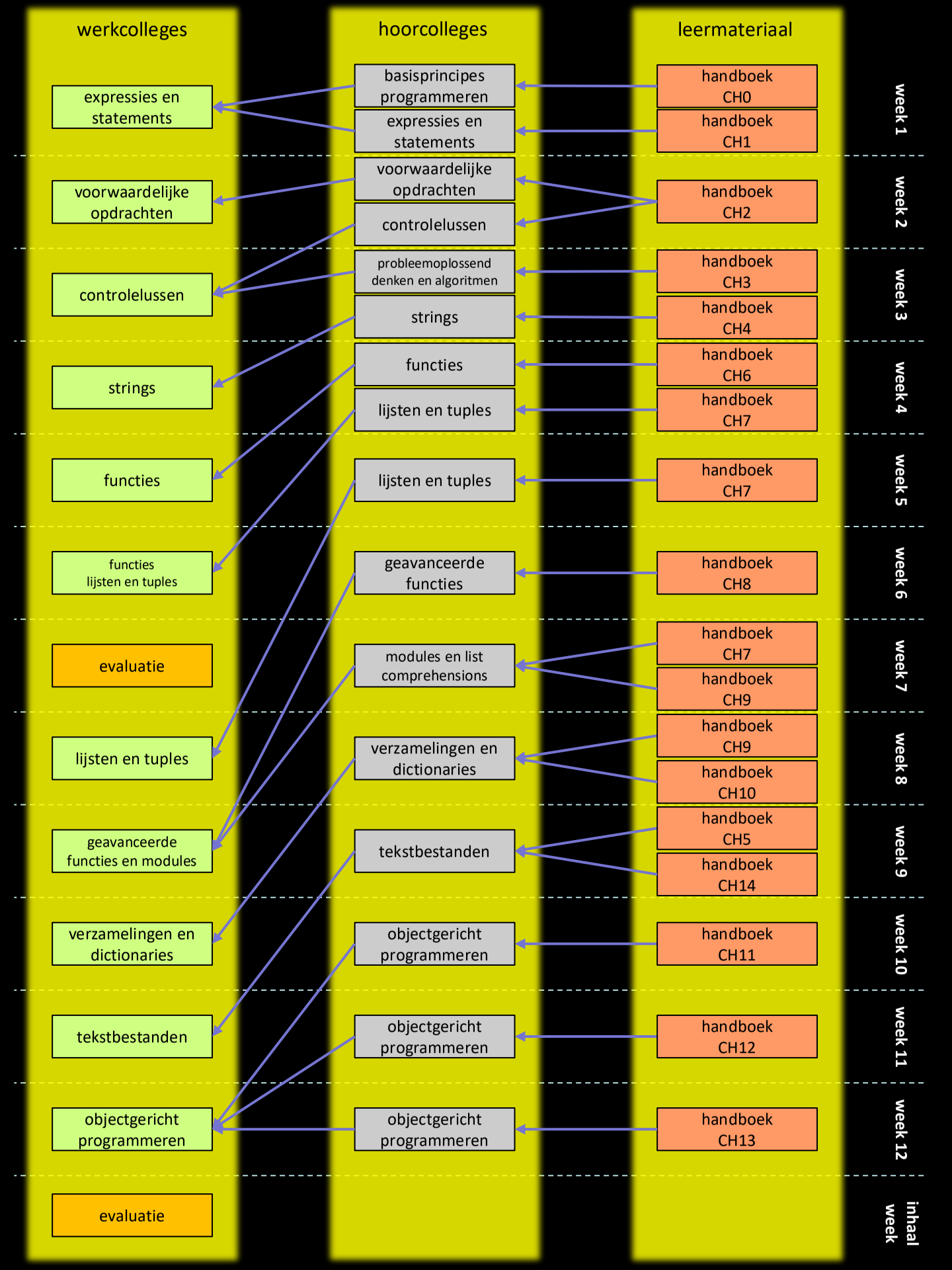
prof. dr. Peter Dawyndt

peter.dawyndt@ugent.be

@dawyndt

UNIVERSITEIT GENT

11


werkcolleges
hoorcolleges
leermateriaal
expressies en statements
voorwaardelijke opdrachten
controlelussen
strings
functies
functies lijsten en tuples
evaluatie
lijsten en tuples
geavanceerde functies en modules
verzamelingen en dictionaries
tekstbestanden
objectgericht programmeren
evaluatie
basisprincipes programmeren
expressies en statements
voorwaardelijke opdrachten
controlelussen
probleemoplossend denken en algoritmen
strings
functies
lijsten en tuples
lijsten en tuples
geavanceerde functies
modules en list comprehensions
verzamelingen en dictionaries
tekstbestanden
objectgericht programmeren
objectgericht programmeren
objectgericht programmeren
handboek CH0
handboek CH1
handboek CH2
handboek CH3
handboek CH4
handboek CH6
handboek CH7
handboek CH7
handboek CH8
handboek CH7
handboek CH9
handboek CH9
handboek CH10
handboek CH5
handboek CH14
handboek CH11
handboek CH12
handboek CH13
week 1
week 2
week 3
week 4
week 5
week 6
week 7
week 8
week 9
week 10
week 11
week 12
inhaal week

Objectgericht programmeren

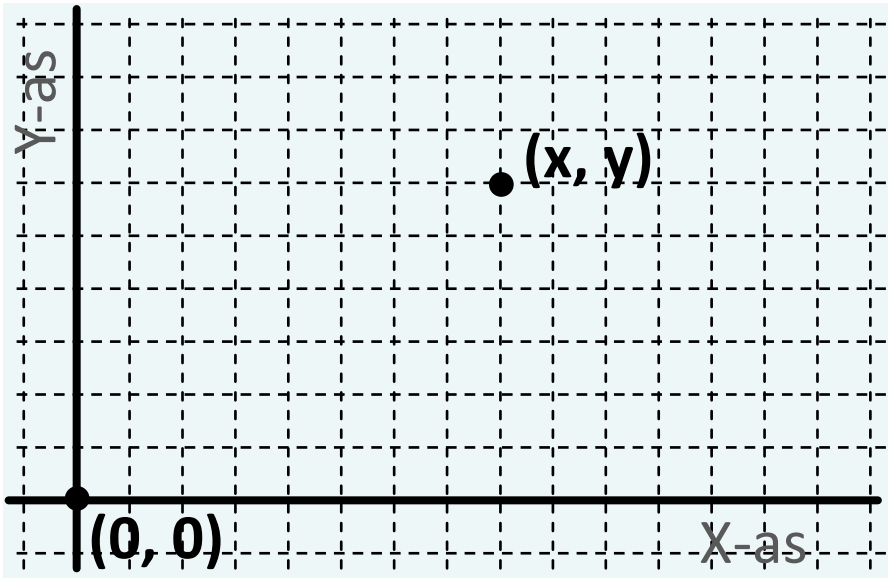
- Python is een objectgerichte programmeertaal
  - populair programmeerparadigma sinds mid jaren '80
  - gericht op ontwikkelen van grote en complexe software

aanmaken van objecten die zowel gegevens als manipulaties bundelen

schrijven van functies die gegevens manipuleren

Zelf gegevenstypes definiëren

- **klasse**: definieert nieuw gegevenstype
  - uitbreiding op ingebouwde gegevenstypes

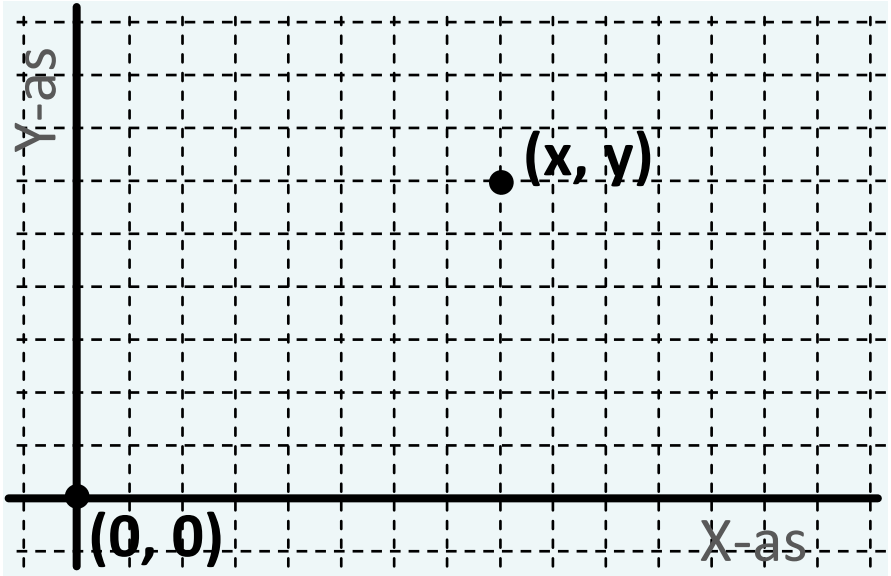



```
class Punt:
    pass
```

UNIVERSITEIT GENT

Zelf gegevenstypes definiëren

- **klasse**: definieert nieuw gegevenstype
  - uitbreiding op ingebouwde gegevenstypes




```
class Punt:
    '''Gegevenstype voor 2-dimensionale punten'''
```

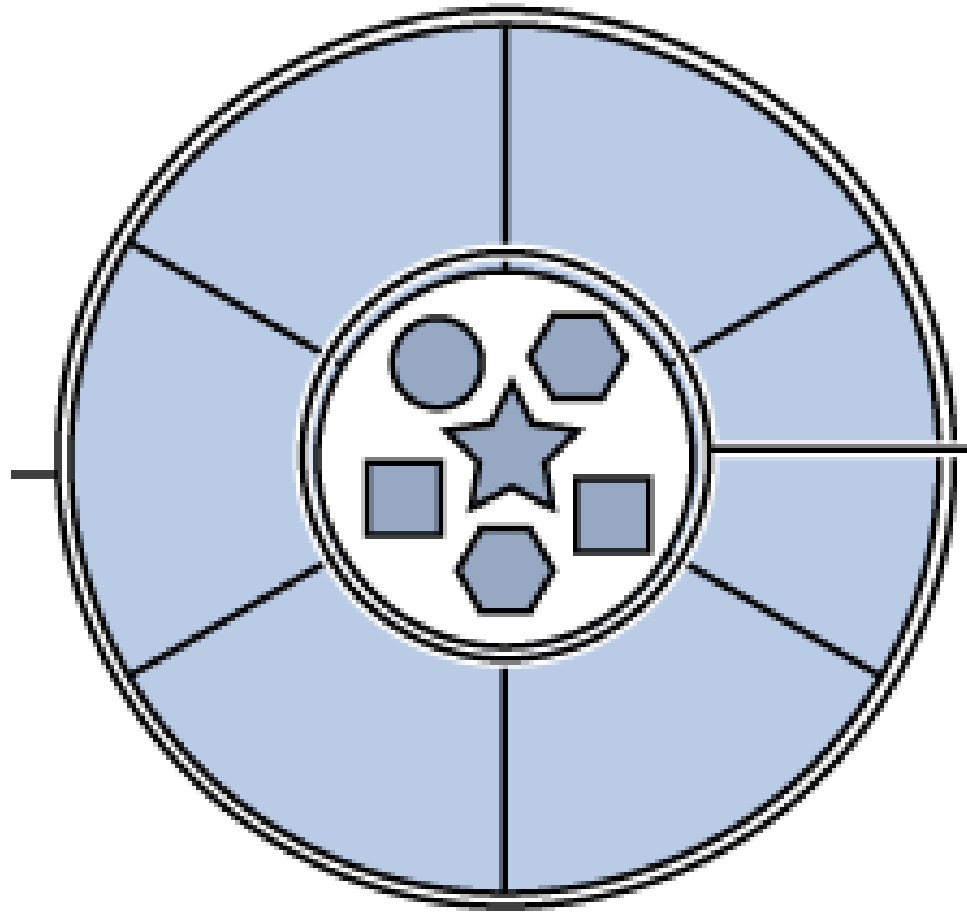
UNIVERSITEIT GENT

Zelf gegevenstypes definiëren

- **klasse**: definieert nieuw gegevenstype
  - Python laat toe om abstracte gegevenstypes te definiëren
    - *abstract data types* (ADT)
    - "abstract" omdat ze implementatiedetails verbergen
    - interactie gebeurt via beperkte set van methoden, eerder dan rechtstreeks gegevens te manipuleren (*encapsulation*)
      - hierdoor kan er (in theorie) minder mislopen
      - programmacode is makkelijker te lezen
      - programmacode is makkelijker te onderhouden

```
class Punt:
    '''Gegevenstype voor 2-dimensionale punten'''
```

Zelf gegevenstypes definiëren

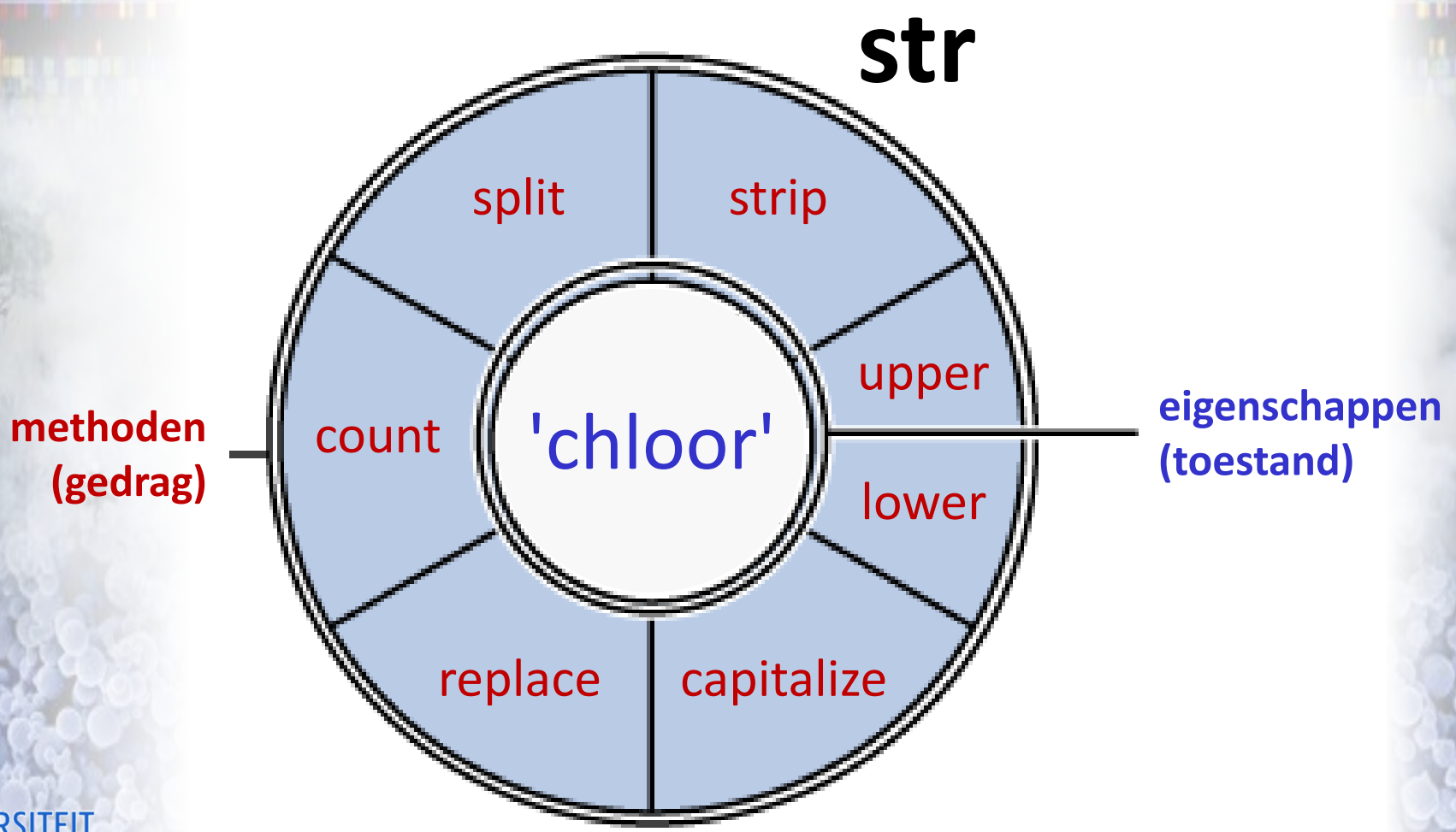


methoden (gedrag)

eigenschappen (toestand)

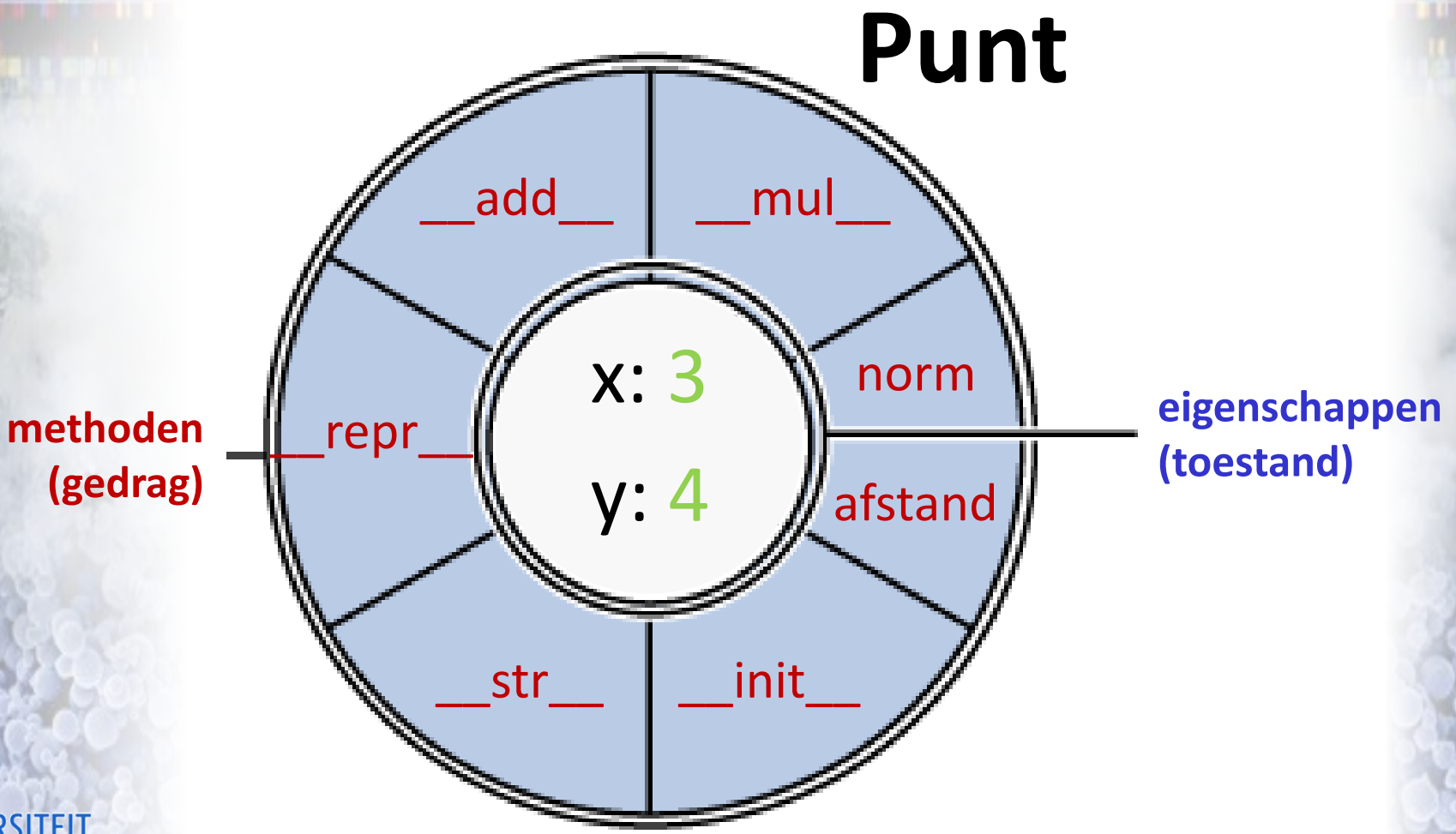
UNIVERSITEIT GENT

Zelf gegevenstypes definiëren



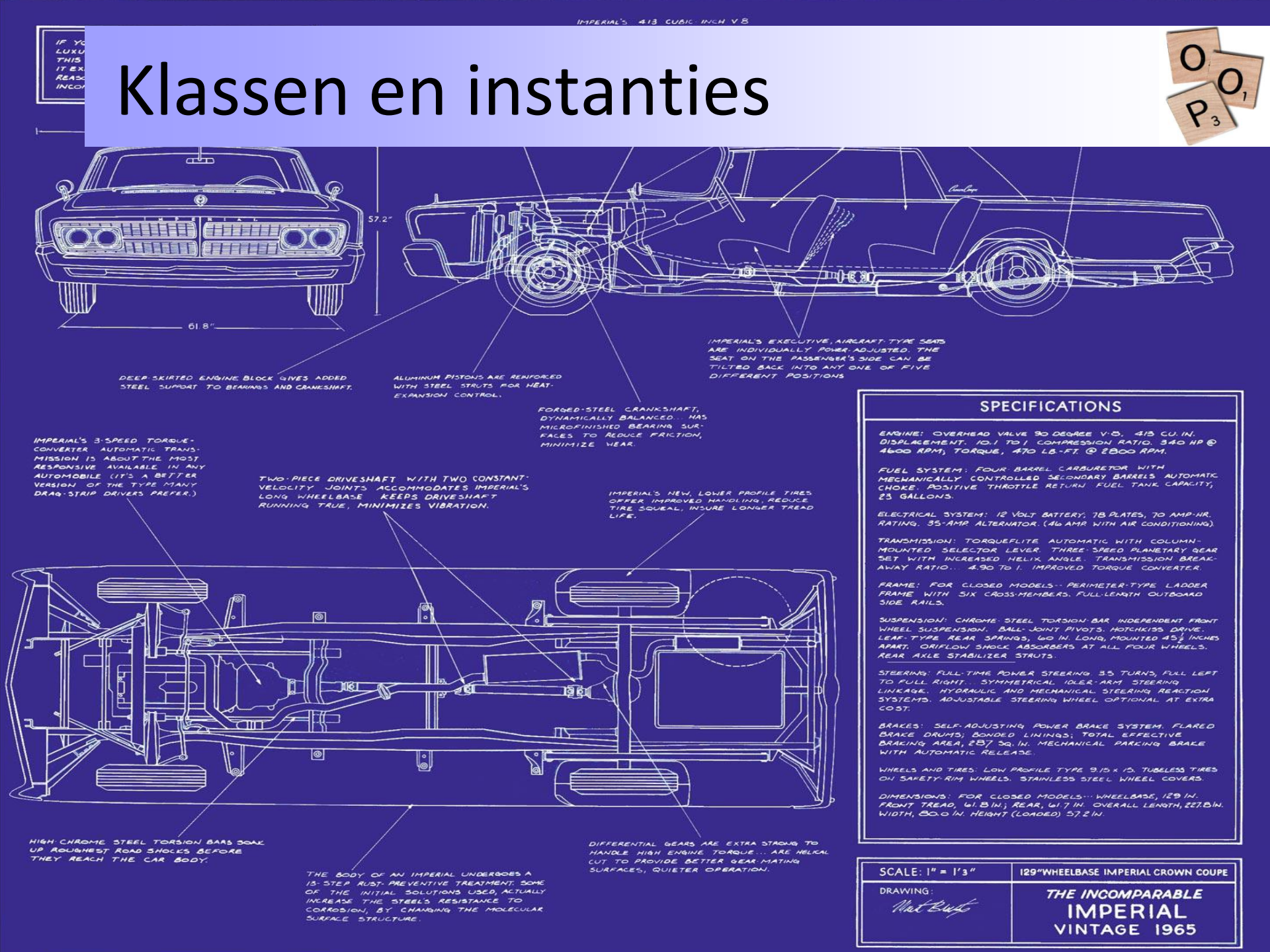
UNIVERSITEIT GENT

Zelf gegevenstypes definiëren



UNIVERSITEIT GENT

Klassen en instanties



Klassen en instanties

- ADT wordt gecreëerd door definitie van klasse
  - geeft aan hoe toestand wordt voorgesteld (**eigenschappen**)
  - geeft aan wat instanties van klasse kunnen doen (**methoden**)
  - klassen zijn zelf ook objecten, net als functies
- objecten worden aangemaakt als instanties van klassen
  - alle objecten van bepaalde ADT hebben zelfde methoden, maar individuele waarden voor hun eigenschappen
  - object wijzigen heeft geen invloed op toestand ander object

UNIVERSITEIT GENT

```
class Punt:
    '''Gegevenstype voor 2-dimensionale punten'''
```

Klassen en instanties

- klasse **`Punt`** creëert nieuw gegevenstype: **`Punt`**
  - ***instantiation***: nieuw object aanmaken
    - wordt gedaan door klasse aan te roepen
    - klassen zijn net als functies 'aanroepbaar' (*callable*)
  - leden van gegevenstype: **instanties** of **objecten**

```
>>> class Punt:
...     pass
...
>>> type(Punt)
type
>>> p = Punt()
>>> type(p)
__main__.Punt
```

UNIVERSITEIT GENT

Klassen en instanties

- bekijk klasse als blauwdruk om objecten te maken
  - se **Punt** is blauwdruk om punten te maken
  - k se **Punt** is zelf geen instantie van een punt, maar lige machinerie om punten te kunnen aanmaken


```
>>> class Punt:
...     pass
...
>>> type(Punt)
type
>>> p = Punt()
>>> type(p)
__main__.Punt
```

UNIVERSITEIT GENT

Eigenschappen

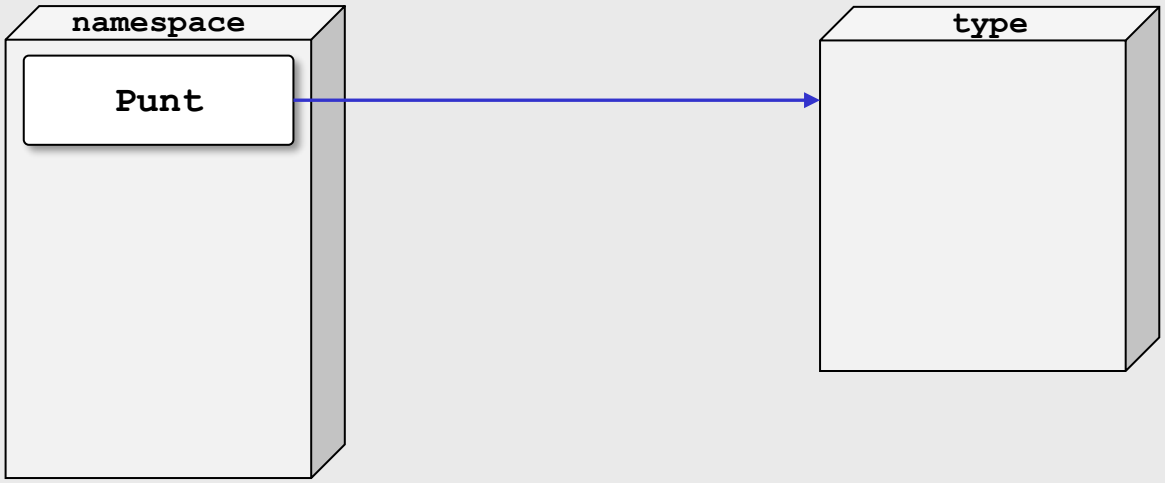
- objecten hebben een toestand en een gedrag
  - toestand bepaald door eigenschappen van instanties
  - instanties vormen eigen *namespace* (net als modules)
  - **attribuut**: naam binnen namespace (instanties en modules)
    - gebruik **dot-notatie** om naam in namespace aan te spreken
    - ook bij modules: **`math.pi`**, **`string.ascii_uppercase`**

```
>>> class Punt:
...     pass
...
>>> p = Punt()
>>> p.x = 3
>>> p.y = 4
```

UNIVERSITEIT GENT

Eigenschappen

objecten (computergeheugen)

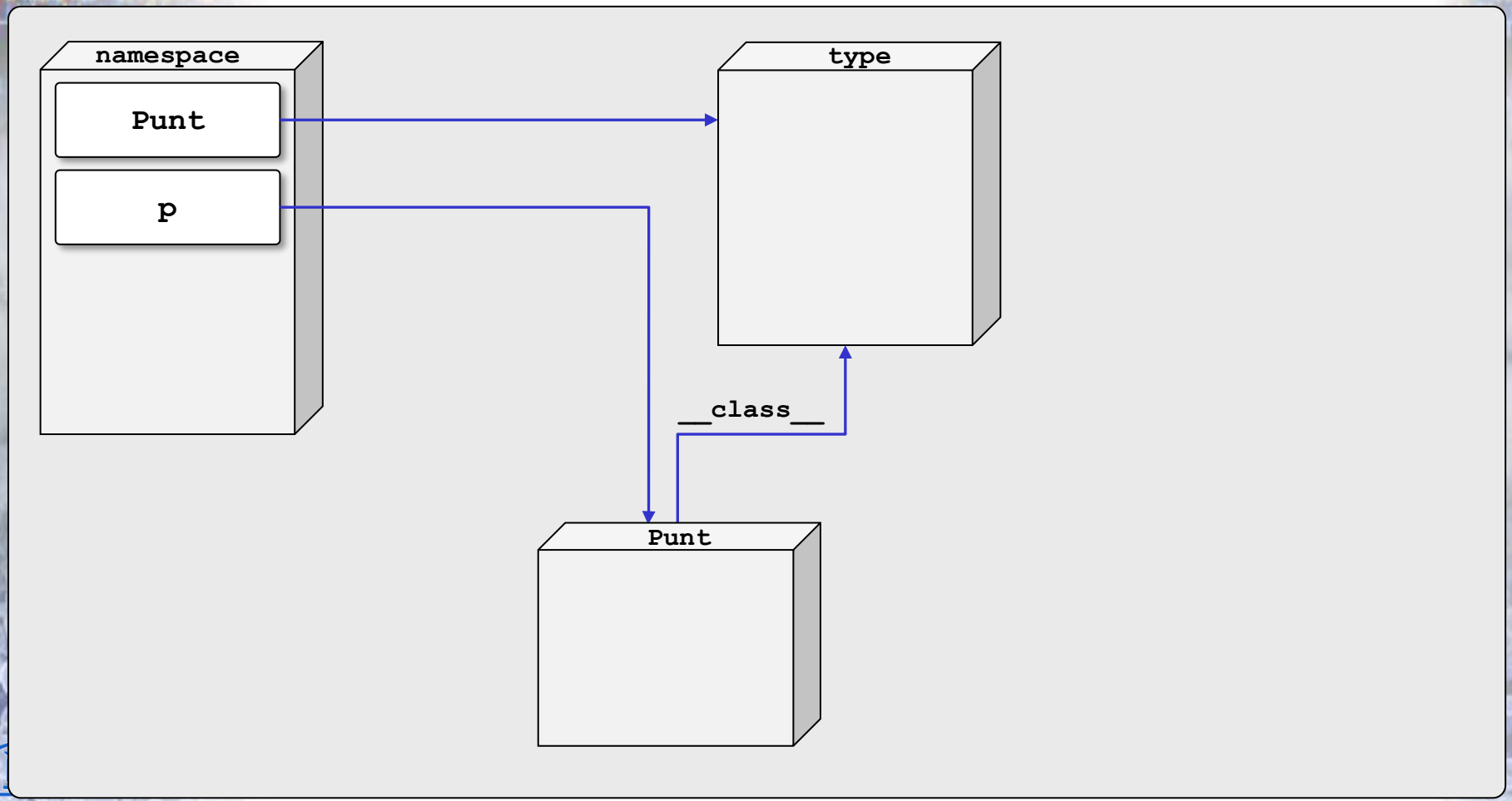


UNIVERSITEIT GENT

```
>>> class Punt: pass
```

Eigenschappen

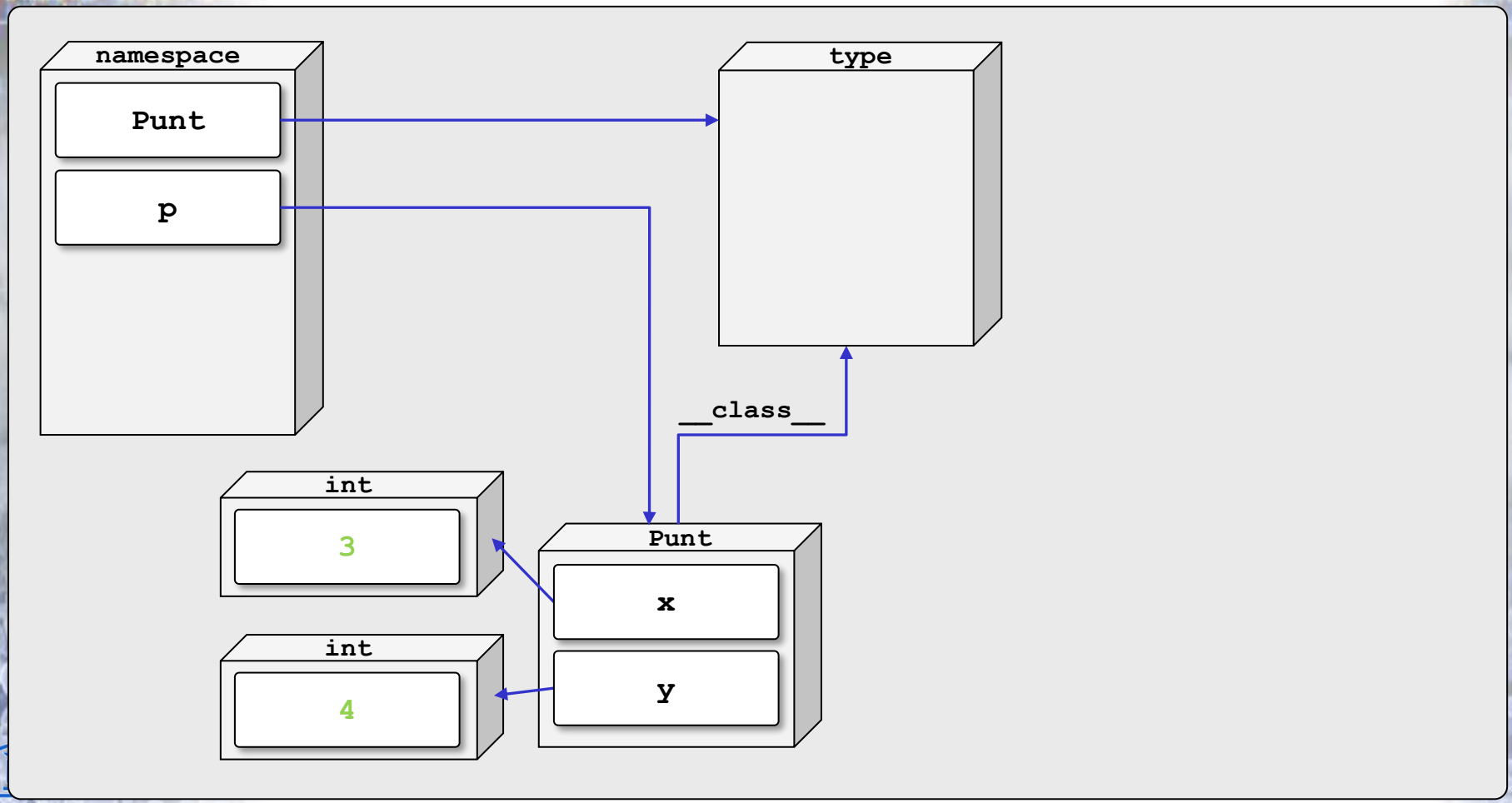
objecten (computergeheugen)




```
>>> p = Punt()
```

Eigenschappen

objecten (computergeheugen)




```
>>> p.x, p.y = 3, 4
```

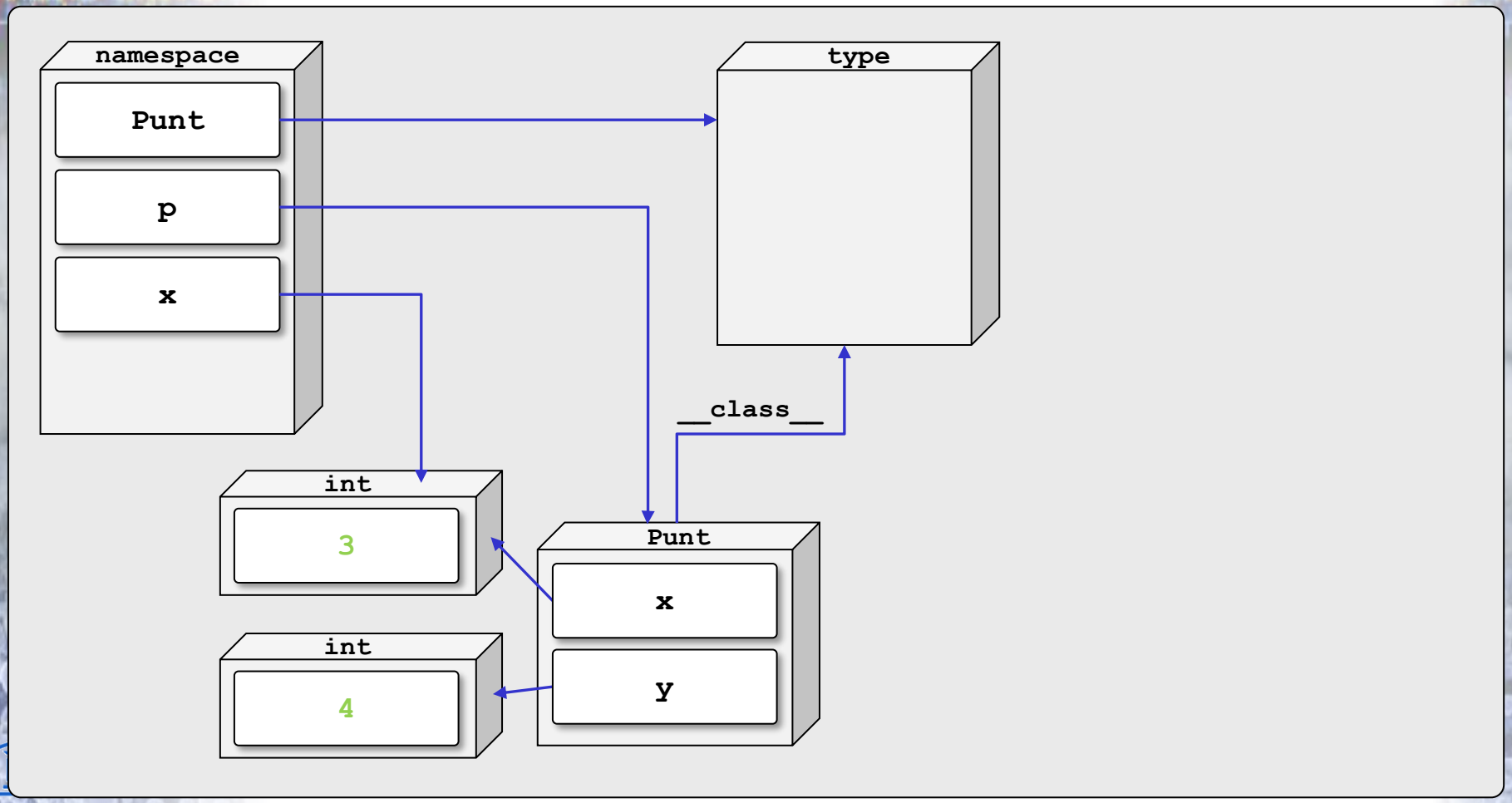
Eigenschappen


```
>>> class Punt:
...     pass
...
>>> p = Punt()
>>> p.x = 3
>>> p.y = 4
>>> p.y
4
>>> x = p.x
```

UNIVERSITEIT GENT

Eigenschappen

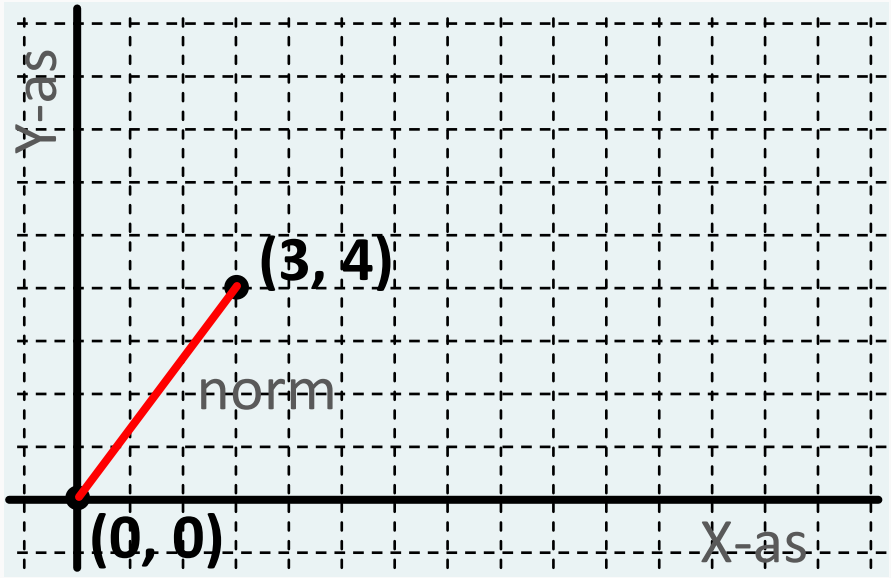
objecten (computergeheugen)




```
>>> x = p.x
```

Eigenschappen


```
>>> class Punt:
...     pass
...
>>> p = Punt()
>>> p.x = 3
>>> p.y = 4
>>> p.y
4
>>> x = p.x
>>> x
3
>>> print(f'({p.x:.2f}, {p.y:.2f})')
(3.00, 4.00)
>>> norm = (p.x ** 2 + p.y ** 2) ** 0.5
>>> norm
5.0
```



UNIVERSITEIT GENT

Methoden

- voeg methode toe door functie te definiëren in klasse
  - instantie altijd als eerste argument doorgegeven aan methode
    - conventie: geef eerste parameter naam **`self`**
    - in tegenstelling tot **`this`** in C++ en Java is deze naam een conventie
    - iedereen gebruikt die, dus wijk daar best niet van af

```
>>> class Punt:
...     def norm(self):
...         return (self.x**2 + self.y**2) ** 0.5
...
>>> p = Punt()
```

UNIVERSITEIT GENT

Methoden

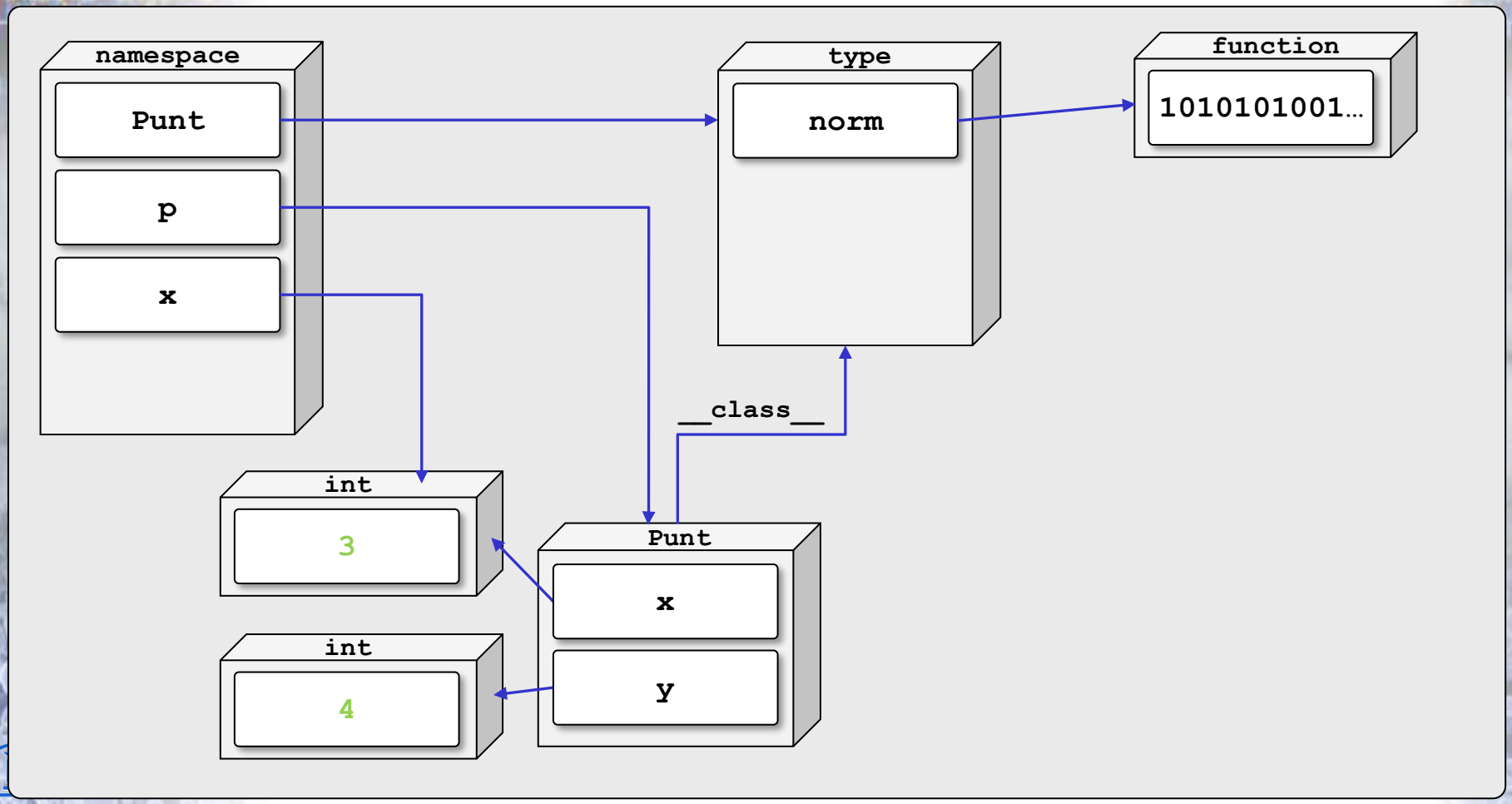
- voeg methode toe door functie te definiëren in klasse
  - methode aanroepen: **`object.methode(argumenten)`**
    - dit zoekt de klasse **`K`** waarvan **`object`** een instantie is
    - roept daarna **`K.methode(object, argumenten)`** aan

```
>>> class Punt:
...     def norm(self):
...         return (self.x**2 + self.y**2) ** 0.5
...
>>> p = Punt()
>>> p.x, p.y = 3, 4
>>> p.norm()
5.0
>>> Punt.norm(p)
5.0
```

UNIVERSITEIT GENT

Methoden

objecten (computergeheugen)

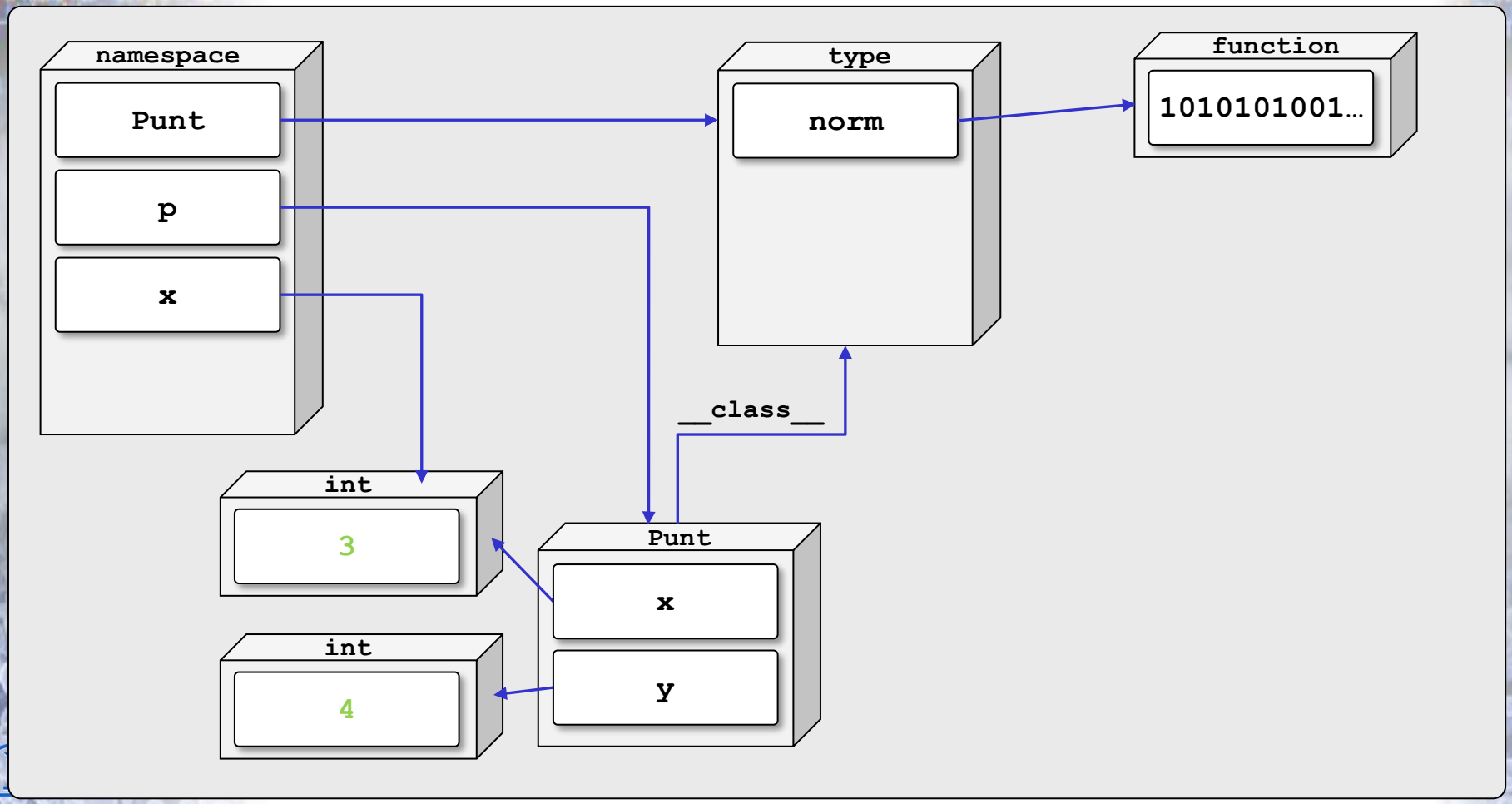


UNIVERSITEIT GENT

```
>>> p.norm()
```

Methoden

objecten (computergeheugen)




```
>>> Punt.norm(p)
```

Initialisatiemethode

- klasse **`Punt`** stelt 2-dimensionale punten voor
  - *alle* punten moeten attributen **`x`** en **`y`** hebben

```
>>> q = Punt()
>>> q.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Punt instance has no attribute 'x'
```

UNIVERSITEIT GENT

Initialisatiemethode

- klasse **`Punt`** stelt 2-dimensionale punten voor
  - *alle* punten moeten attributen **`x`** en **`y`** hebben
  - initialisatiemethode **`__init__`** initialiseert **`Punt`** objecten
    - automatisch aangeroepen als object aangemaakt wordt
    - wordt daarom ook vaak *constructor* genoemd
    - analoog: **`list()`** maakt een lijst, **`str()`** maakt een string

punt1.py

```
class Punt:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def norm(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5
```

UNIVERSITEIT GENT

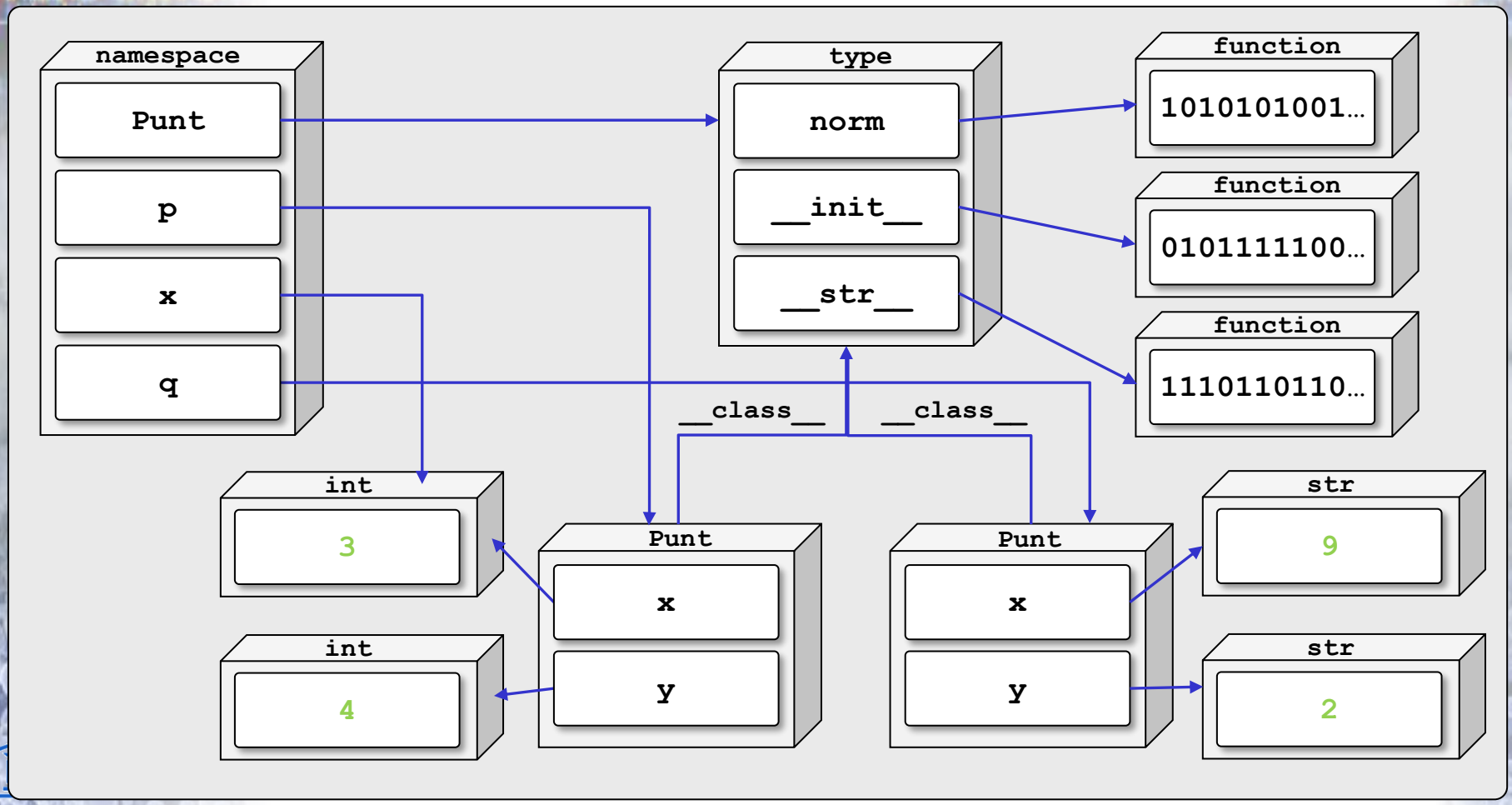
Initialisatiemethode


```
>>> from punt1 import Punt
>>> p = Punt(3, 4)
>>> p.x
3
>>> p.y
4
>>> p.norm()
5.0
>>> x = p.x
>>> q = Punt(9, 2)
>>> q.x
9
>>> q.y
2
>>> q.norm()
9.219544572928871
```

UNIVERSITEIT GENT

Initialisatiemethode

objecten (computergeheugen)




```
>>> p, q = Punt(3, 4), Punt(9, 2)
```

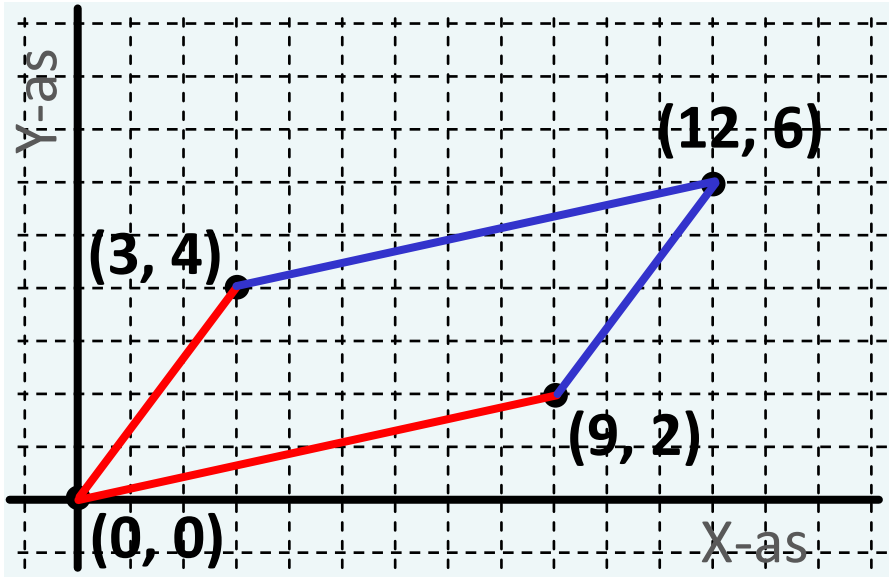
Instanties als parameters

- objecten kunnen op de gebruikelijke manier als argument aan functies doorgegeven worden

```
>>> p, q = Punt(3, 4), Punt(9, 2)
>>> def toon(p):
...     print(f'({p.x:.2f}, {p.y:.2f})')
>>> toon(p)
(3.00, 4.00)
>>> def vectorsom(p1, p2):
...     return Punt(p1.x + p2.x, p1.y + p2.y)
```

UNIVERSITEIT GENT

Instanties als parameters



UNIVERSITEIT GENT

```
>>> def vectorsom(p1, p2):
...     return Punt(p1.x + p2.x, p1.y + p2.y)
```

Instanties als parameters

- objecten kunnen op de gebruikelijke manier als parameter aan functies doorgegeven worden

```
>>> p, q = Punt(3, 4), Punt(9, 2)
>>> def toon(p):
...     print(f'({p.x}, {p.y})')
>>> toon(p)
(3.00, 4.00)
>>> def vectorsom(p1, p2):
...     return Punt(p1.x + p2.x, p1.y + p2.y)
>>> r = vectorsom(p, q)
>>> isinstance(r, Punt)
True
>>> toon(r)
(12.00, 6.00)
```

UNIVERSITEIT GENT

Operator overloading

Operator overloading

- methoden gedragen zich als functies, maar
  - methoden worden gedefinieerd binnen klassedefinitie
    - expliciete relatie tussen klasse en methode
  - syntaxis om methode aan te roepen verschilt van syntaxis om functie aan te roepen

punt1.py

```
class Punt:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def norm(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5
```

UNIVERSITEIT GENT

Operator overloading


```
>>> from punt2 import Punt
>>> p, q = Punt(3, 4), Punt(9, 2)
>>> p.toon()
'(3.00, 4.00)'
>>> r = p.vectorsom(q)
>>> r.toon()
'(12.00, 6.00)'
```

punt2.py

```
class Punt:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def norm(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

    def toon(self):
        return f'({self.x:.2f}, {self.y:.2f})'

    def vectorsom(self, other):
        return Punt(self.x + other.x, self.y + other.y)
```

UNIVERSITEIT GENT

Operator overloading

- ingebouwde functies en operatoren toegepast op **o** automatisch omgezet naar ingebouwde methoden van **K**
  - **str(o) → K.__str__(o)**
    - geeft voorstelling van object als string terug
    - functie **str** impliciet aangeroepen door functie **print**
  - **repr(o) → K.__repr__(o)**
    - interne voorstelling van object als string (initialisatiemethode)
  - **len(o) → K.__len__(o)**
  - **o + o2 → K.__add__(o, o2)**
  - **o * o2 → K.__mul__(o, o2)**

```
class K: pass
o = K()
```

Operator overloading


```
>>> from punt3 import Punt
>>> p, q = Punt(3, 4), Punt(9, 2)
>>> p
Punt(3, 4)
>>> r = p + q
>>> print(r)
(12.00, 6.00)
```

punt3.py

```
class Punt:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def norm(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5
    def __str__(self):
        return f'({self.x:.2f}, {self.y:.2f})'
    def __repr__(self):
        return f'Punt({self.x}, {self.y})'
    def __add__(self, other):
        return Punt(self.x + other.x, self.y + other.y)
```

UNIVERSITEIT GENT

Romeins rekenen

I = 1
II = 2
III = 3
IV = 4
V = 5
VI = 6
VII = 7
VIII = 8
IX = 9
X = 10

XL = 40
L = 50
LX = 60
LXX = 70
LXXX = 80
XC = 90
C = 100
D = 500
M = 1000

Klassen: 'new style' vs 'old style'

- Python 2.x
  - onderscheid tussen 'new style' en 'old style' klassen
  - verschil zit enkel in geavanceerde (technische) details

```
class Punt:
    pass
```
old style

```
class Punt(object):
    pass
```
new style

- Python 3.x
  - geen onderscheid tussen 'new style' en 'old style' klassen

```
class Punt:
    pass
```
new style

```
class Punt(object):
    pass
```
new style

UNIVERSITEIT GENT

OOP karakteristieken

Huiswerk (volgende les)

- handboek
  - lees hoofdstuk 12 (more on classes)
  - lees hoofdstuk 13 (OOP development)
  - let in het bijzonder op gebruikte OOP terminologie
- voorbeelden volgende les (reeks 10)
  - immuunsysteem
  - dominostenen
  - romeins rekenen

UNIVERSITEIT GENT

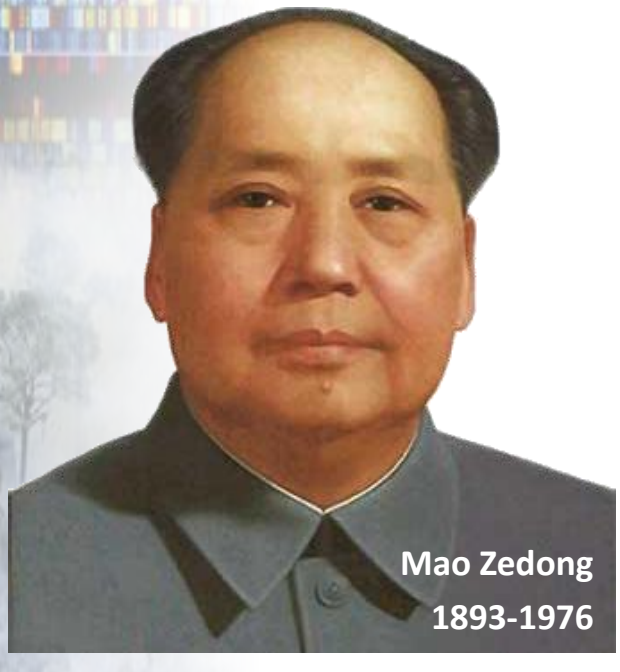
Huiswerk (werkcolleges)

- tweede evaluatie
  - maandag 16 december 2024 (14:30-16:45)
  - dinsdag 17 december 2024 (10:00-12:15)
  - inschrijven in Ufora groepen

Vragen of opmerkingen?

UNIVERSITEIT GENT

The sky is the limit …



"Classes struggle,
some classes triumph,
others are eliminated."

— Mao Zedong

UNIVERSITEIT GENT