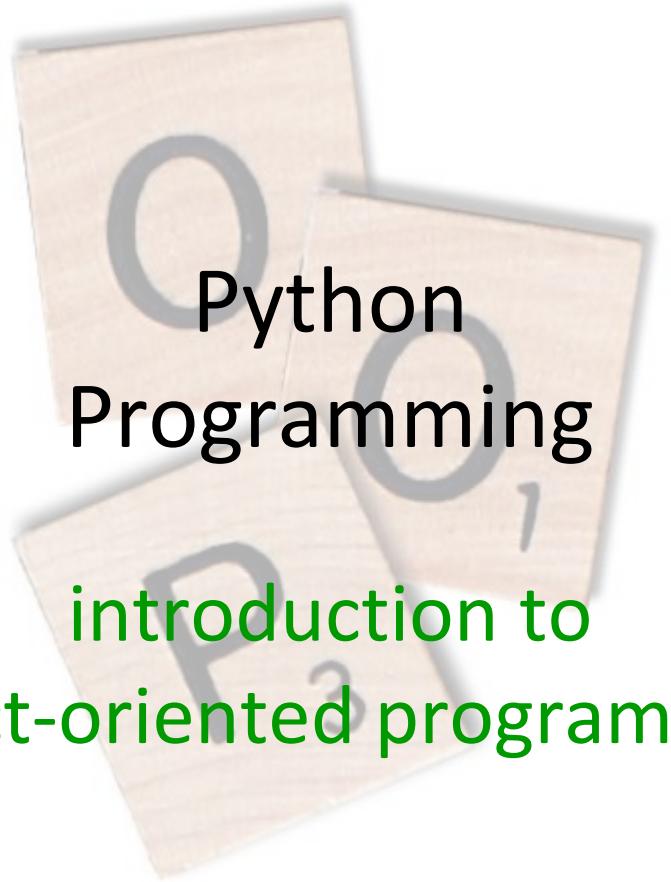
Ghent University
Faculty of Sciences



Python Programming

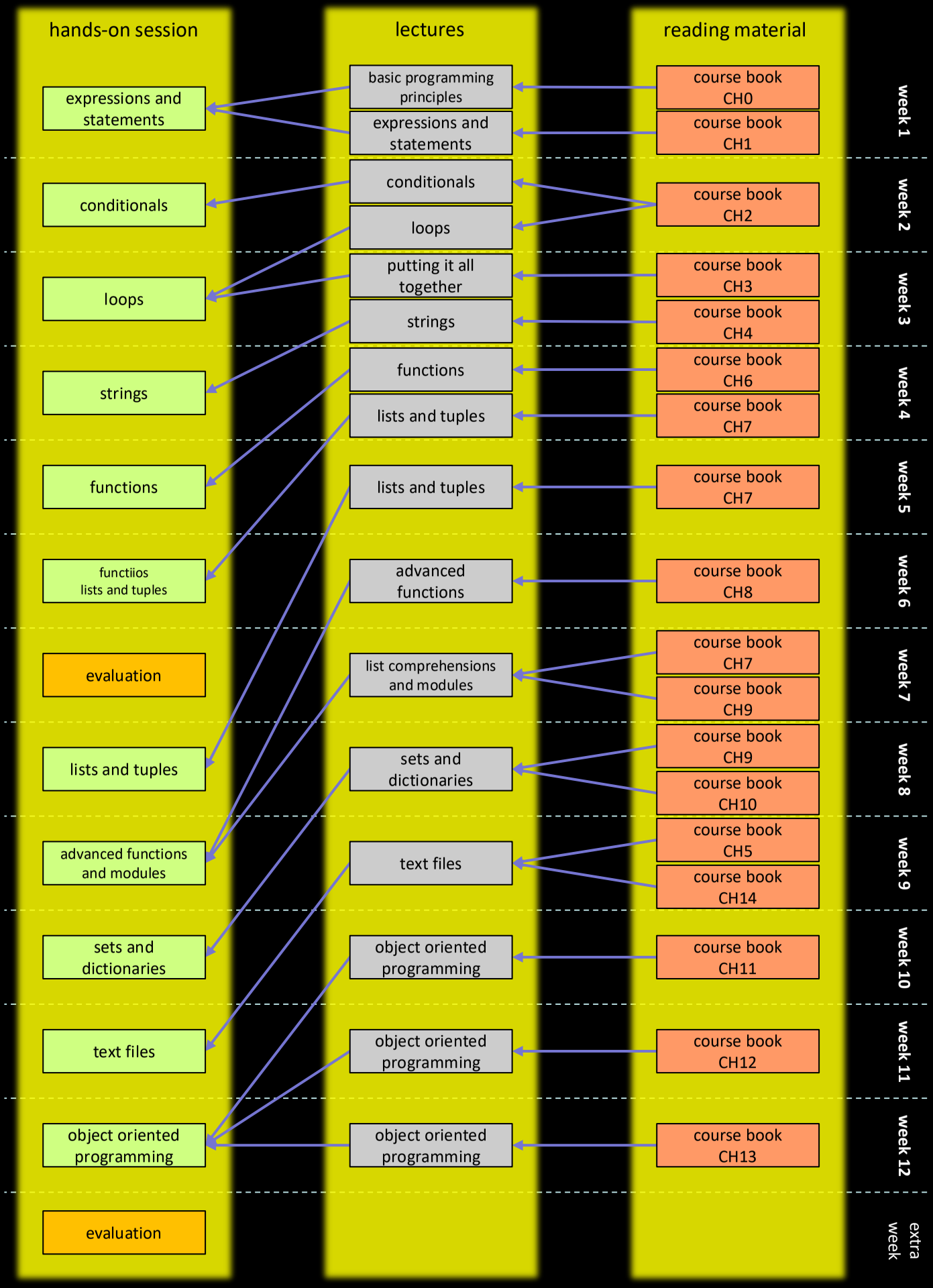
introduction to object-oriented programming

Prof. Dr. Peter Dawyndt

peter.dawyndt@ugent.be

@dawyndt

UNIVERSITEIT GENT


hands-on session
lectures
reading material
week 1
week 2
week 3
week 4
week 5
week 6
week 7
week 8
week 9
week 10
week 11
week 12
extra week
expressions and statements
conditionals
loops
strings
functions
functiios lists and tuples
evaluation
lists and tuples
advanced functions and modules
sets and dictionaries
text files
object oriented programming
evaluation
basic programming principles
expressions and statements
conditionals
loops
putting it all together
strings
functions
lists and tuples
lists and tuples
advanced functions
list comprehensions and modules
sets and dictionaries
text files
object oriented programming
object oriented programming
object oriented programming
course book CH0
course book CH1
course book CH2
course book CH3
course book CH4
course book CH6
course book CH7
course book CH7
course book CH8
course book CH7
course book CH9
course book CH9
course book CH10
course book CH5
course book CH14
course book CH11
course book CH12
course book CH13

Object-oriented programming

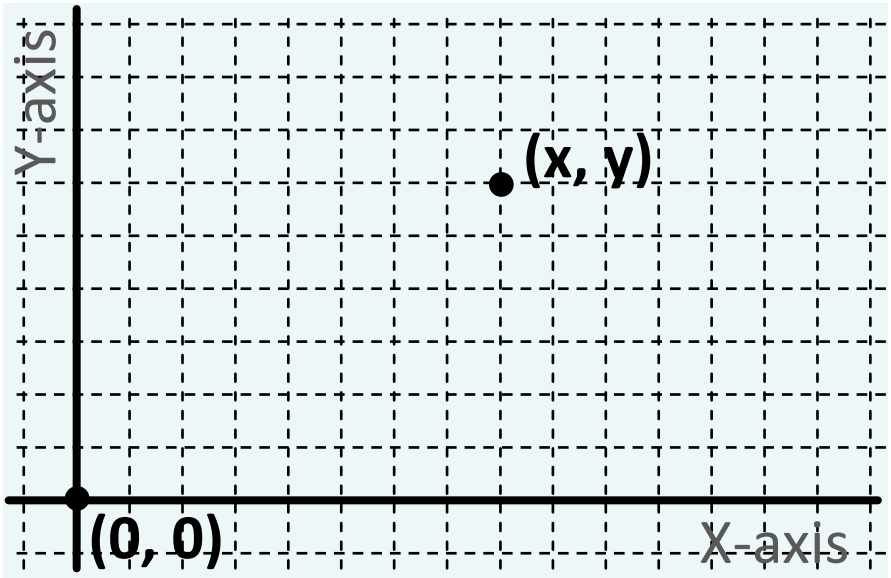
- Python is an object-oriented programming language
  - popular programming paradigm since mid 1980s
  - handles rapidly increasing size and complexity of software

create objects that bundle both data and functionality

write functions that operate on data

User-defined compound types

- **class**: defines a new (compound) data type
  - extension to built-in data types

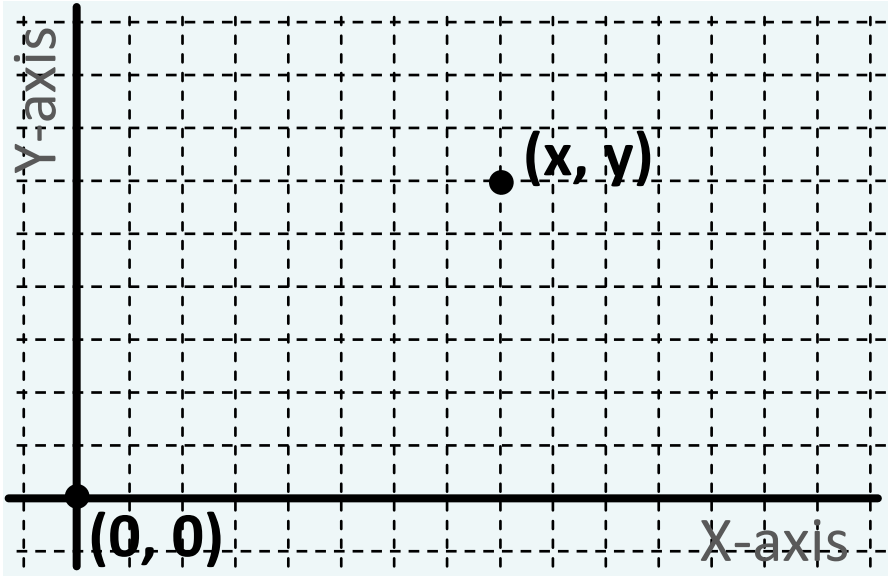



```
class Point:
    pass
```

UNIVERSITEIT GENT

User-defined compound types

- **class**: defines a new (compound) data type
  - extension to built-in data types




```
class Point:
    '''Representation of two-dimensional points'''
```

UNIVERSITEIT GENT

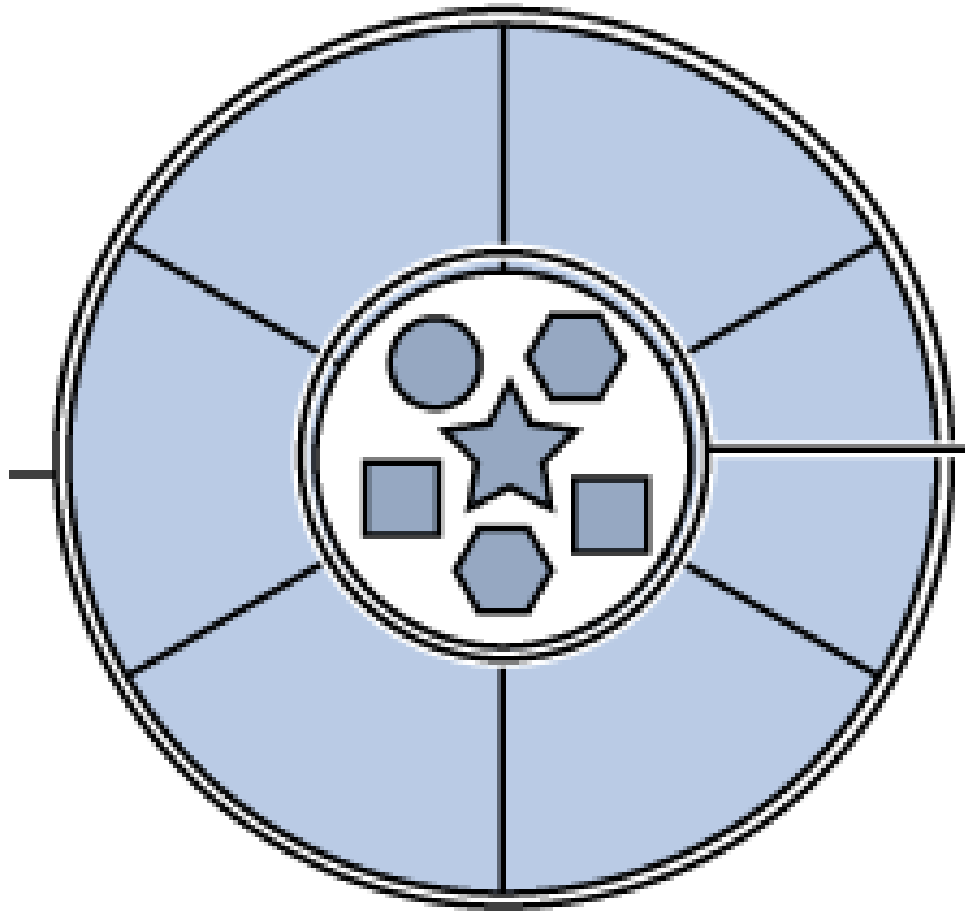
User-defined compound types

- **class**: defines a new (compound) data type
  - Python enables creation of *abstract data types* (ADTs)
    - "abstract" because they hide implementation details
    - programmers interact with them through a limited set of operations (*methods*), rather than by manipulating data directly (*encapsulation*)
      - fewer thing can go wrong (at least in theory)
      - resulting code is easier to read
      - makes code easier to maintain, since internals can be changed without changing calling code

UNIVERSITEIT GENT

```
class Point:
    '''Representation of two-dimensional points'''
```

User-defined compound types

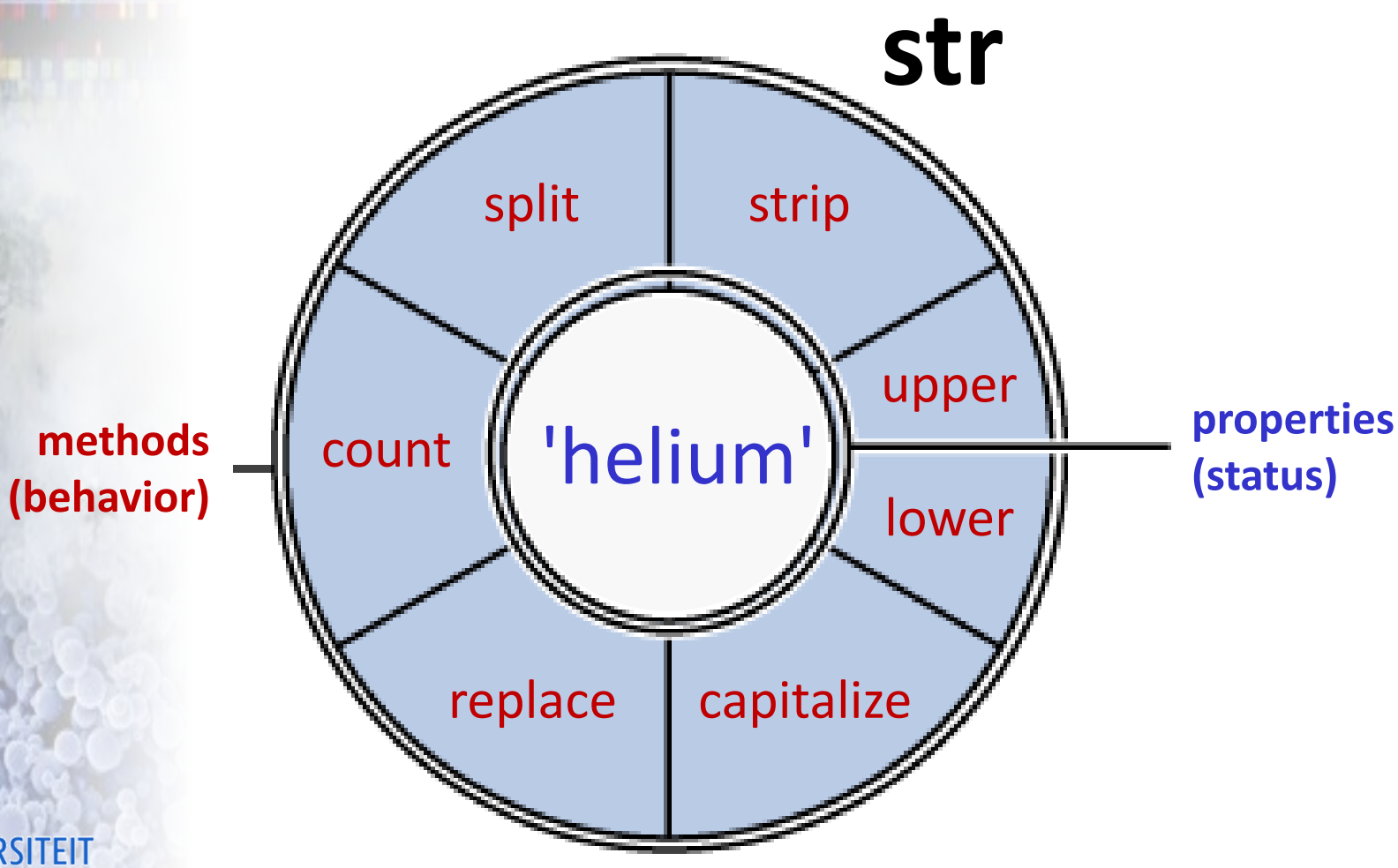


methods (behavior)

properties (status)

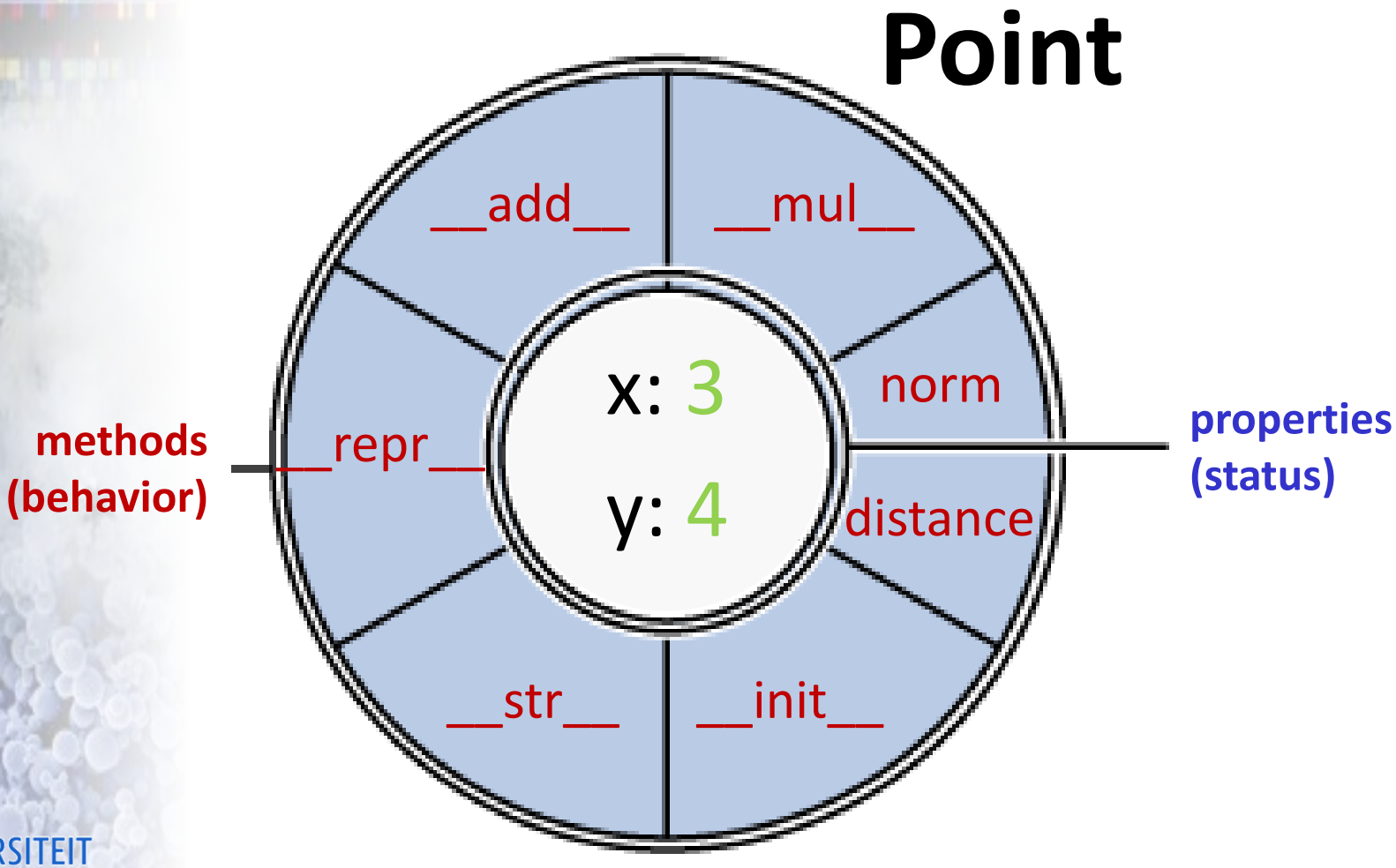
UNIVERSITEIT GENT

User-defined compound types



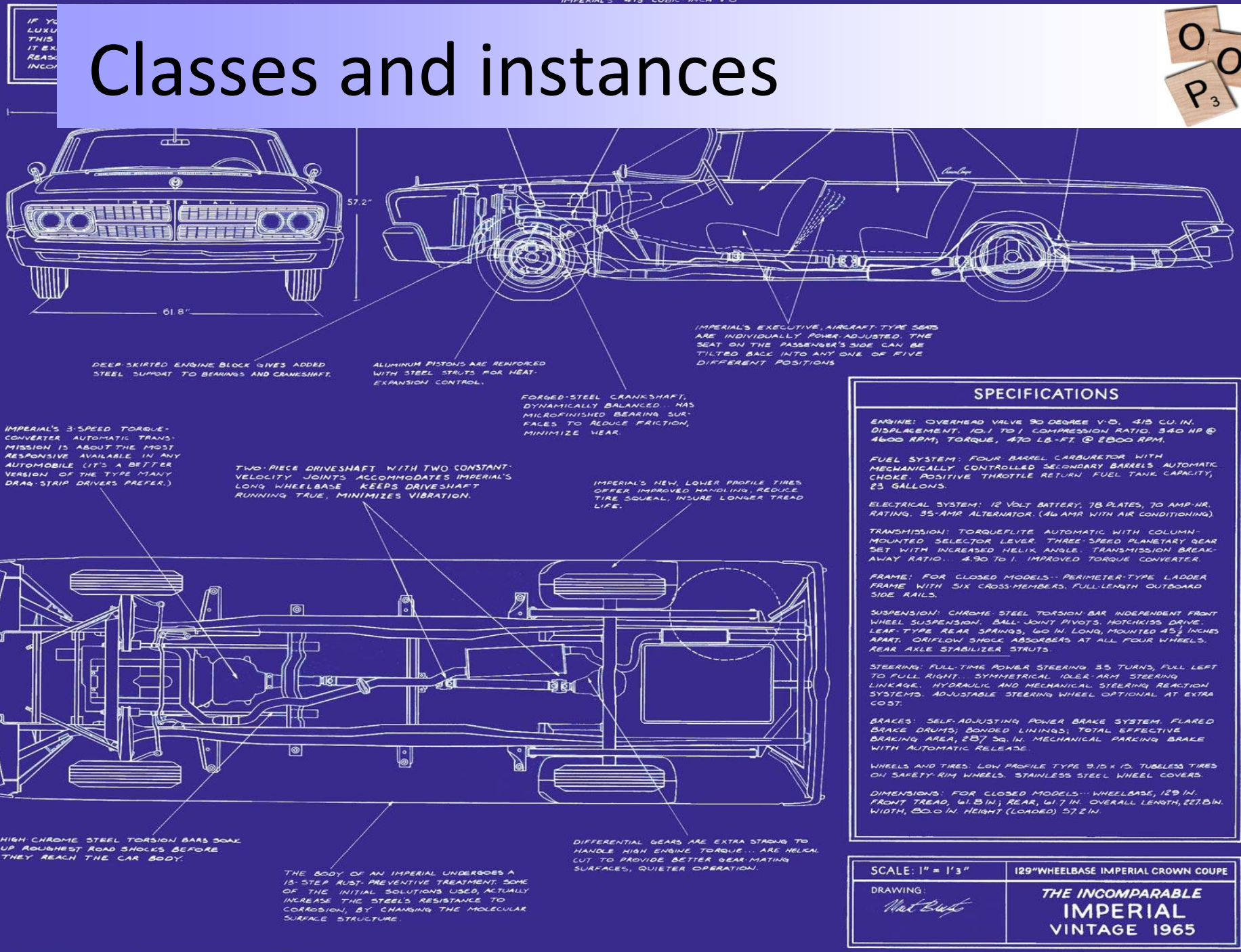
UNIVERSITEIT GENT

User-defined compound types



UNIVERSITEIT GENT

Classes and instances



Classes and instances

- ADT usually created by defining a *class* that specifies
  - how it stores state (*properties*)
  - what it can do (*methods*)
  - classes are also objects themselves, just like functions
- *objects* are created as *instances* of a class
  - each object of a particular ADT shares the class's methods, but usually has its own properties
  - changes to one object thus do not affect the state of others

UNIVERSITEIT GENT

```
class Point:
    '''Representation of two-dimensional points'''
```

Classes and instances

- class **`Point`** defines a new data type: **`Point`**
  - *instantiation*: creating a new instance of a class
    - done by calling the class name as if it were a function
    - classes are *callable* just like functions
  - members of a data type: *instances* or *objects*

```
>>> class Point:
...     pass
...
>>> type(Point)
<class 'type'>
>>> p = Point()
>>> type(p)
<class '__main__.Point'>
```

UNIVERSITEIT GENT

Classes and instances

- think of a class as a blueprint for making objects
  - s **`Point`** is a factory for making points
  - cl s **`Point`** isn't an instance of a point, but it
    he machinery to make point instances


```
>>> class Point:
...     pass
...
>>> type(Point)
<class 'type'>
>>> p = Point()
>>> type(p)
<class '__main__.Point'>
```

UNIVERSITEIT GENT

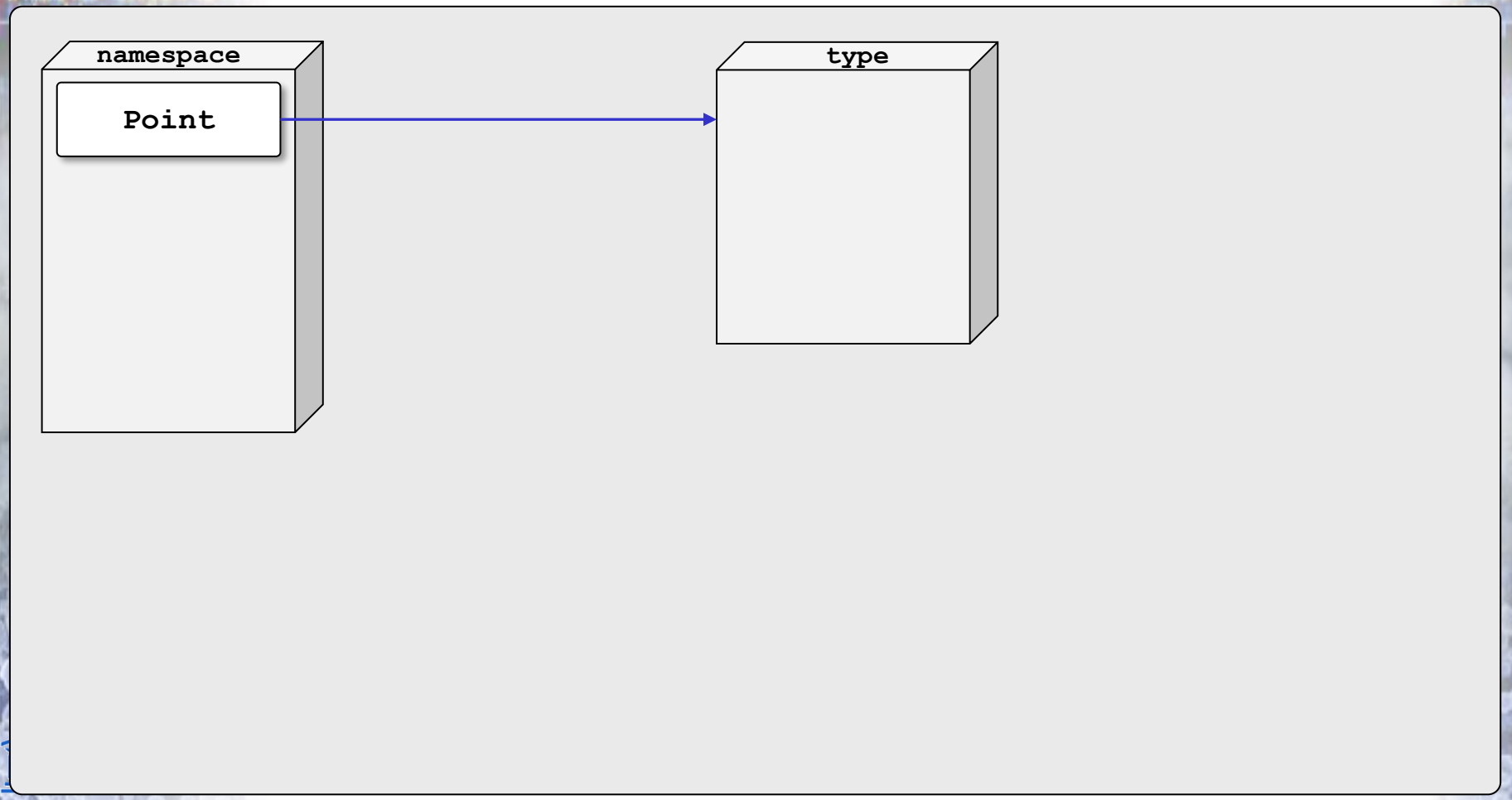
Properties

- object instances both have state and behavior
  - attributes of an instance determine its state
  - instances have their own *namespace* (just like modules)
  - *attribute*: name in namespace (instances and modules)
    - use *dot notation* to access name from namespace
    - similar to module syntax: **`math.pi`**, **`string.uppercase`**

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

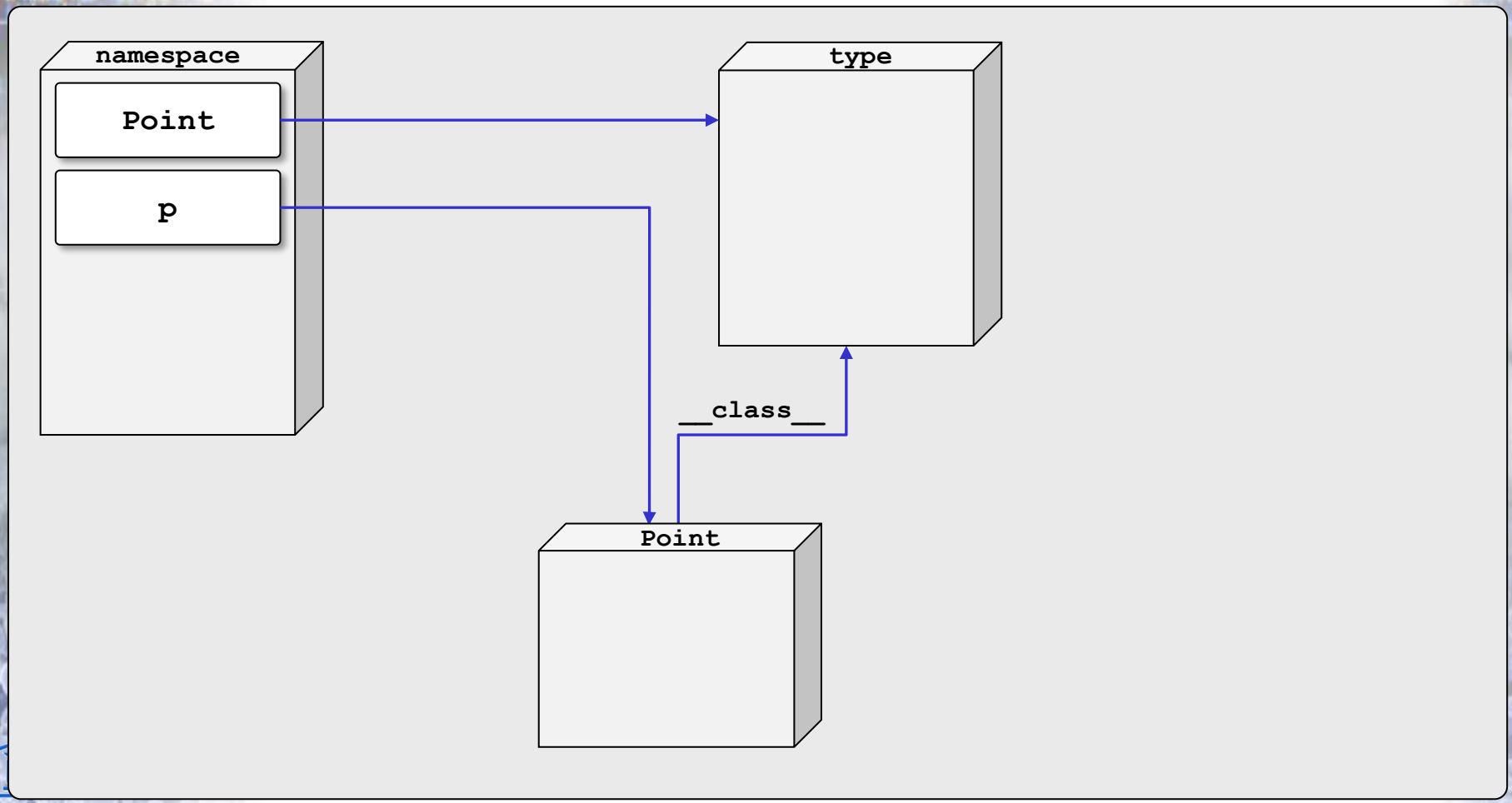
UNIVERSITEIT GENT

Properties



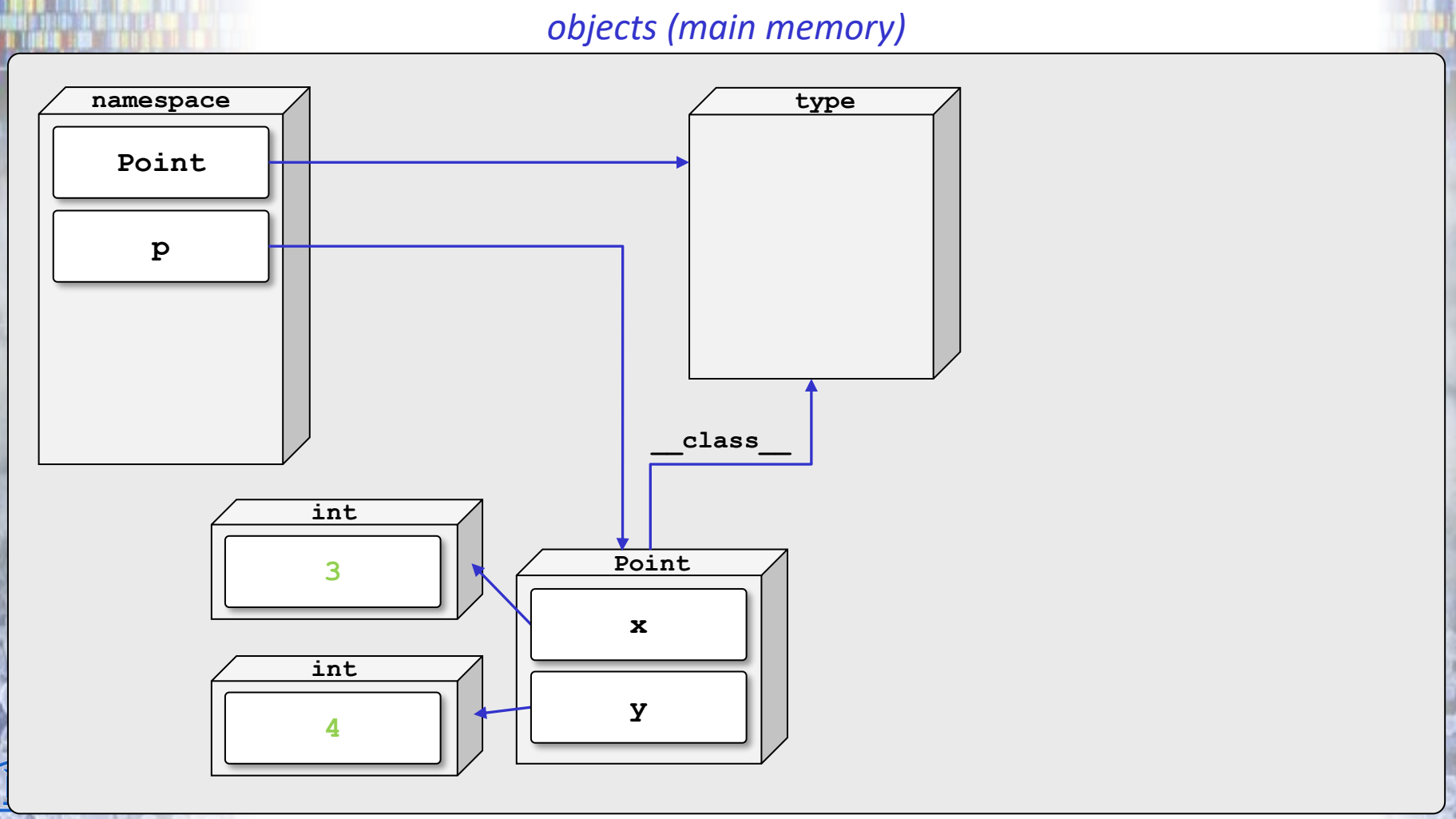

```
>>> class Point: pass
```

Properties




```
>>> p = Point()
```

Properties



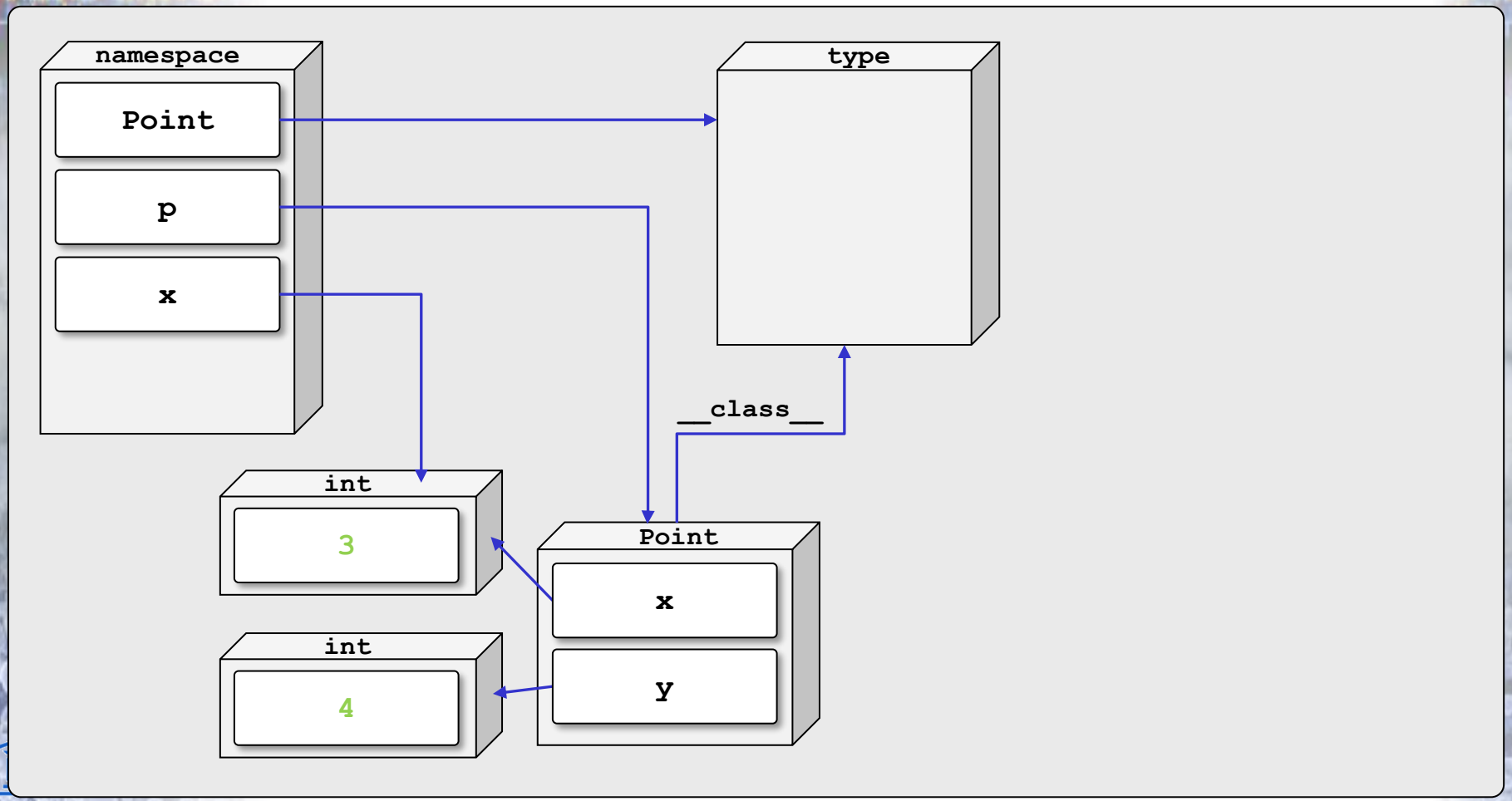

```
>>> p.x, p.y = 3, 4
```

Properties


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

UNIVERSITEIT GENT

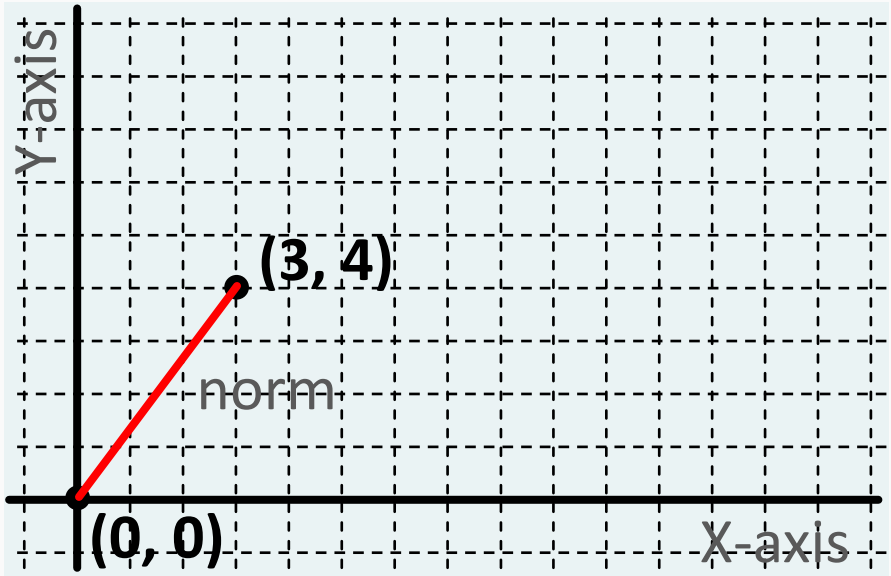
Properties




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

Properties


```
>>> class Point:
...     pass
...
>>> p = Point()
>>> 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

Methods

- give a class methods by defining functions inside of it
  - object itself is always passed to the method as its first argument
    - universally called **`self`**
    - unlike **`this`** in C++ and Java, the name is just a convention
    - but everyone uses it, and you should too

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

UNIVERSITEIT GENT

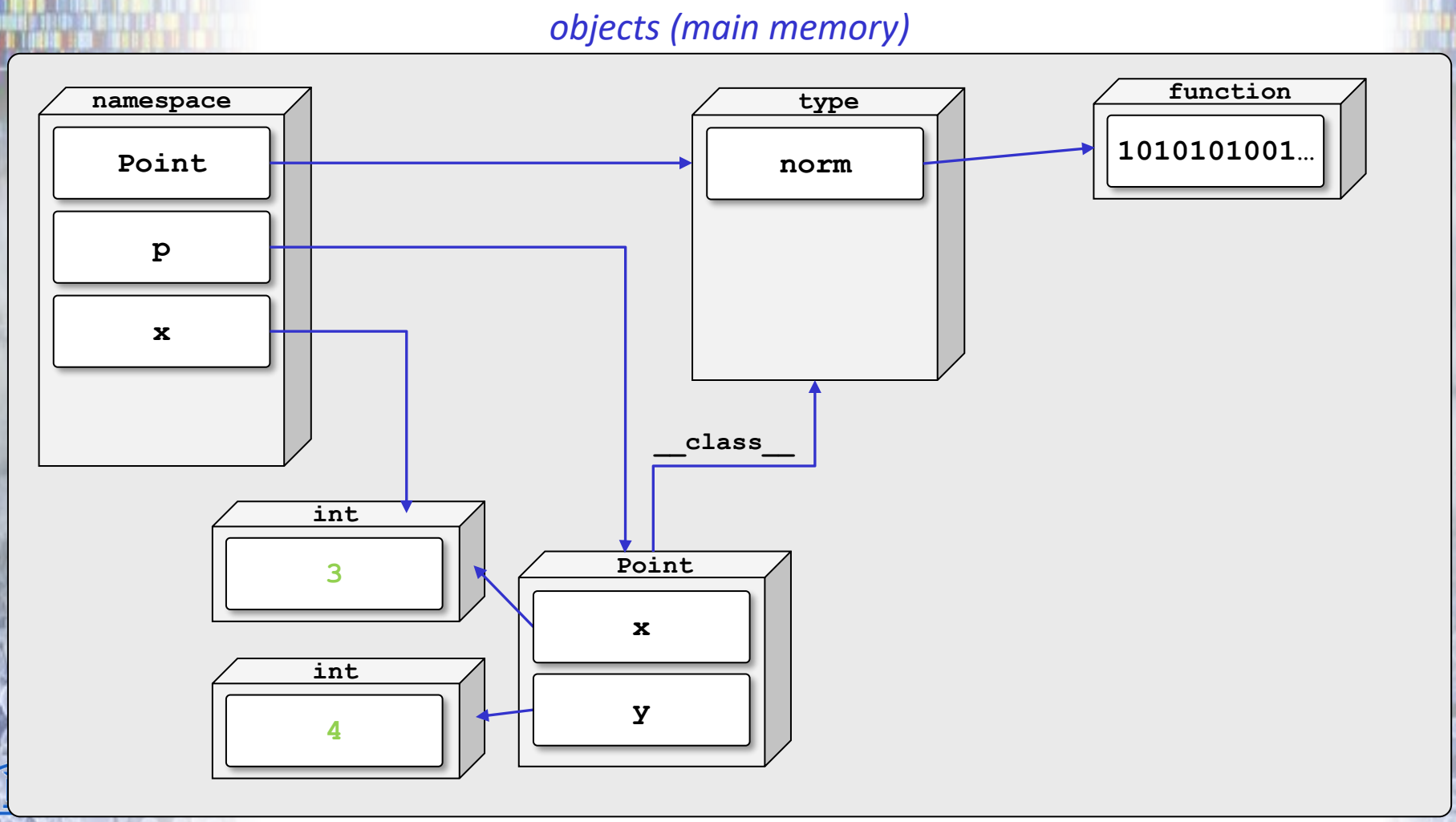
Methods

- give a class methods by defining functions inside of it
  - calling methods: **`object.method(arguments)`**
    - finds the class **`C`** that **`object`** is an instance of
    - then calls **`C.method(object, arguments)`**

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

UNIVERSITEIT GENT

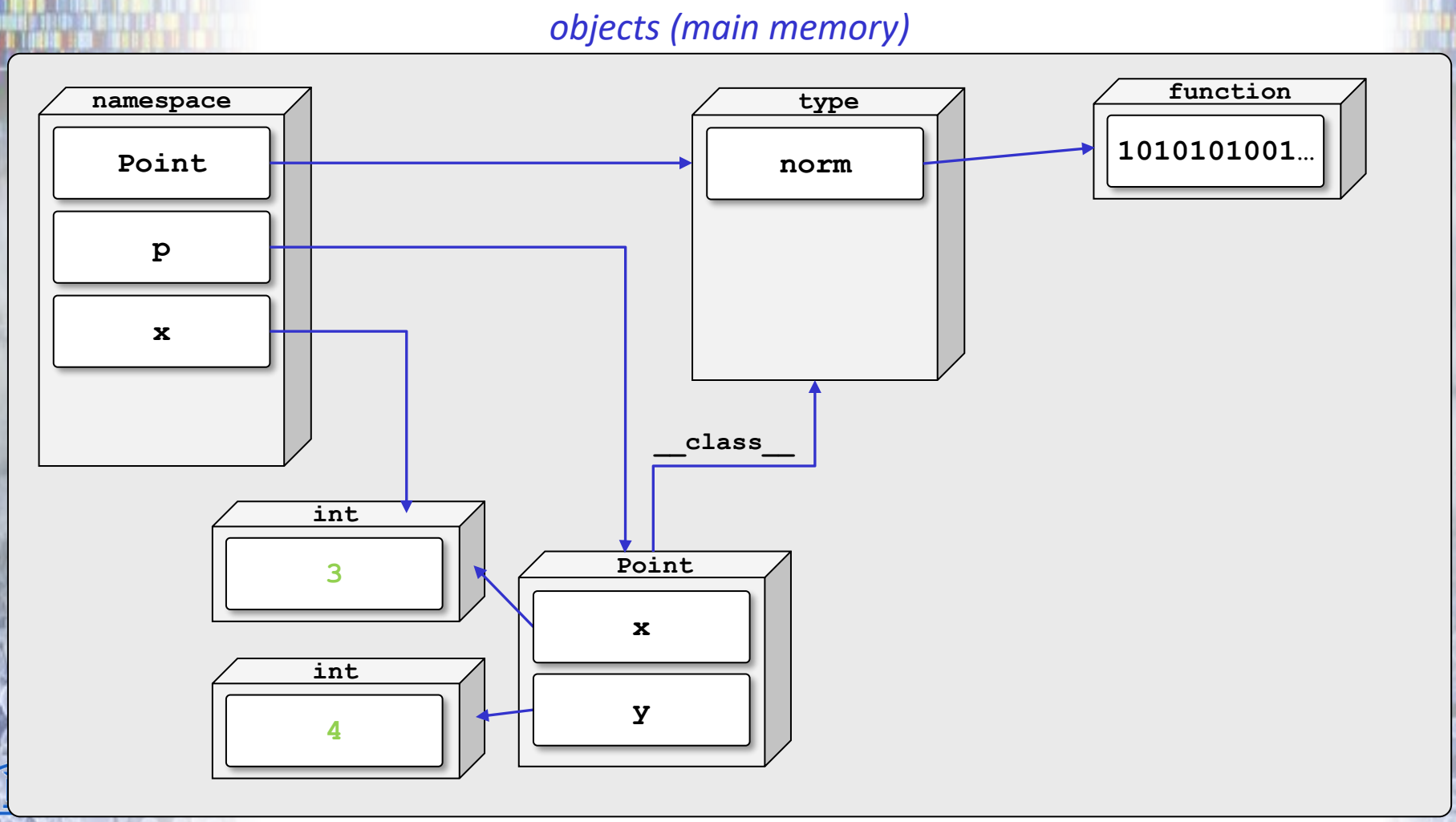
Methods



UNIVERSITEIT GENT

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

Methods




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

The initialisation method

- class **`Point`** intends to represent two-dimensional points
  - *all* point instances ought to have **`x`** and **`y`** attributes

```
>>> p2 = Point()
>>> p2.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Point instance has no attribute 'x'
```

UNIVERSITEIT GENT

The initialisation method

- class **`Point`** intends to represent two-dimensional points
  - *all* point instances ought to have **`x`** and **`y`** attributes
  - initialisation method **`__init__`** initialises **`Point`** objects
    - called automatically when the class is called (*instantiation*)
    - for that reason it is sometimes named the *constructor*
    - analogy: **`list()`** constructs a list, **`str()`** constructs a string

point1.py

```
class Point:
    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
```

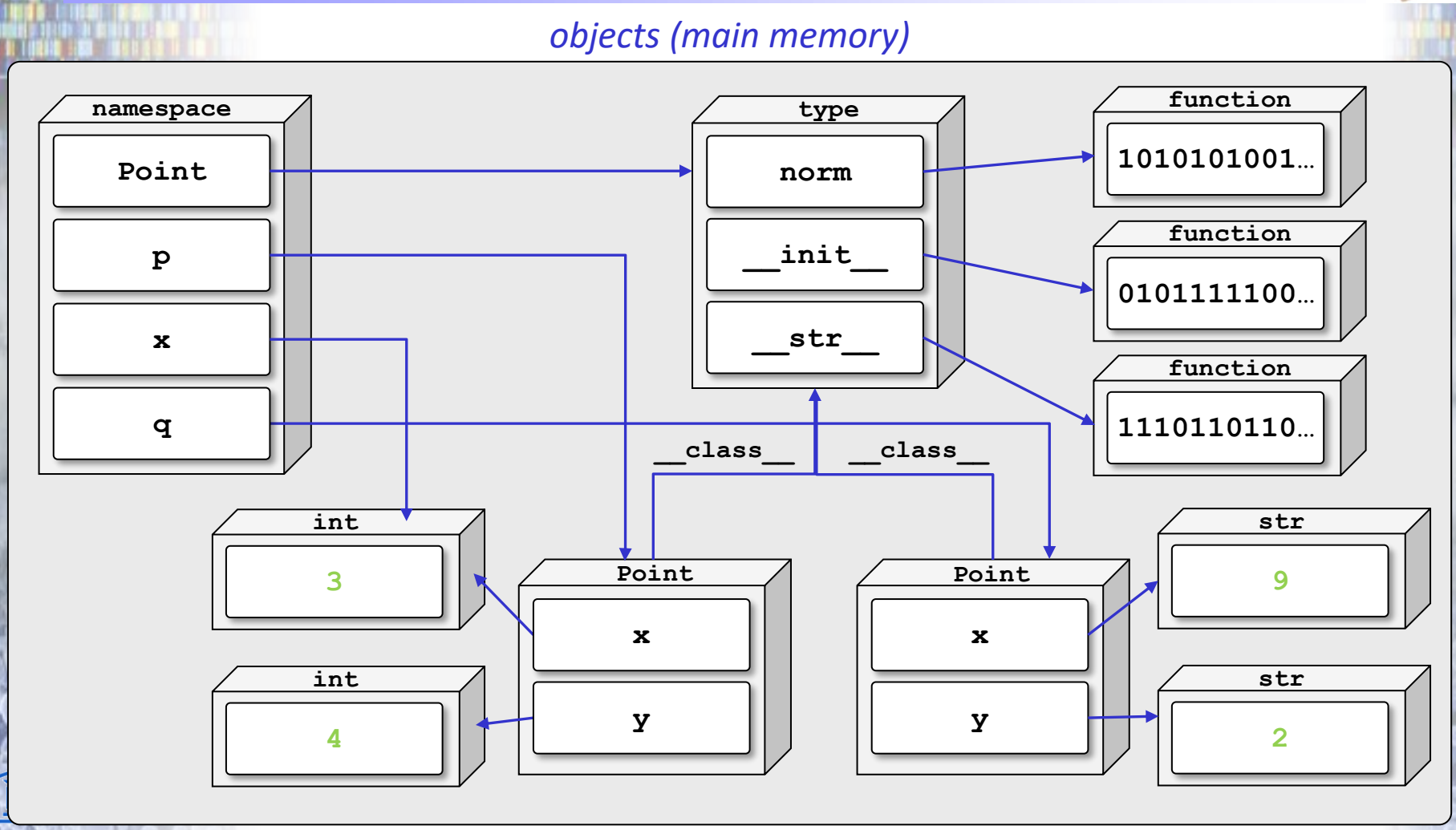
UNIVERSITEIT GENT

The initialisation method


```
>>> from point1 import Point
>>> p = Point(3, 4)
>>> p.x
3
>>> p.y
4
>>> p.norm()
5.0
>>> x = p.x
>>> q = Point(9, 2)
>>> q.x
9
>>> q.y
2
>>> q.norm()
9.219544572928871
```

UNIVERSITEIT GENT

The initialisation method




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

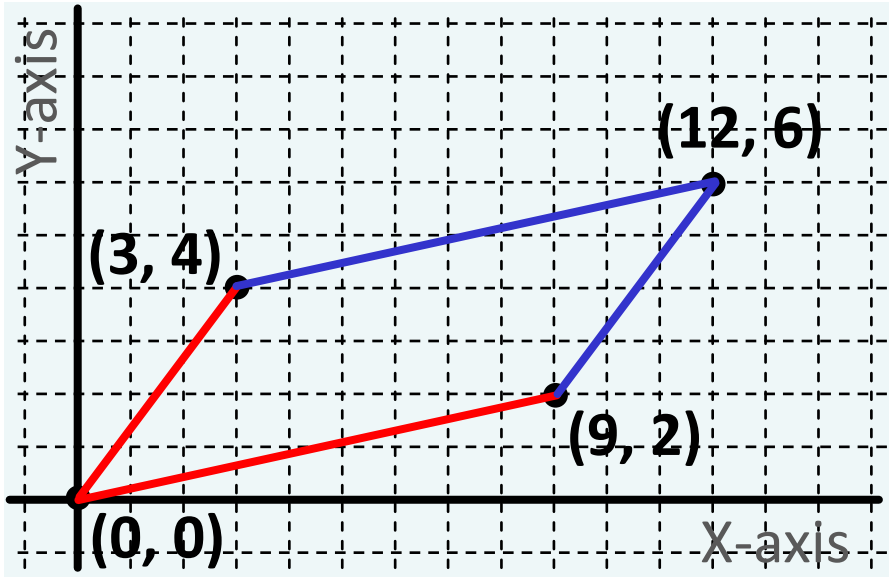
Instances as parameters

- an object can be passed as an argument to a function in the usual way

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

UNIVERSITEIT GENT

Instances as parameters




```
>>> def vectorsum(p1, p2):
...     return Point(p1.x + p2.x, p1.y + p2.y)
```

UNIVERSITEIT GENT

Instances as parameters

- an object can be passed as an argument to a function in the usual way

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

Operator overloading

Operator overloading

- methods behave like functions, but
  - methods are defined inside class definitions
    - explicit relationship between class and method
  - syntax used to call methods differs from syntax use to call functions

point1.py

```
class Point:
    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 point2 import Point
>>> p, q = Point(3, 4), Point(9, 2)
>>> p.display()
'(3, 4)'
>>> r = p.vectorsum(q)
>>> r.display()
'(12, 6)'
```

point2.py

```
class Point:
    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 display(self):
        return f'({self.x:.2f}, {self.y:.2f})'

    def vectorsum(self, other):
        return Point(self.x + other.x, self.y + other.y)
```

UNIVERSITEIT GENT

Operator overloading

- built-in functions and operators applied to object **o** are automatically forwarded to method calls on class **C**
  - **str(o) → C.__str__(o)**
    - returns string representation of object **o**
    - **str** function is called implicitly by **print** function
  - **repr(o) → C.__repr__(o)**
    - returns "internal" string representation of object **o**
  - **len(o) → C.__len__(o)**
  - **o + o2 → C.__add__(o, o2)**
  - **o * o2 → C.__mul__(o, o2)**

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

Operator overloading

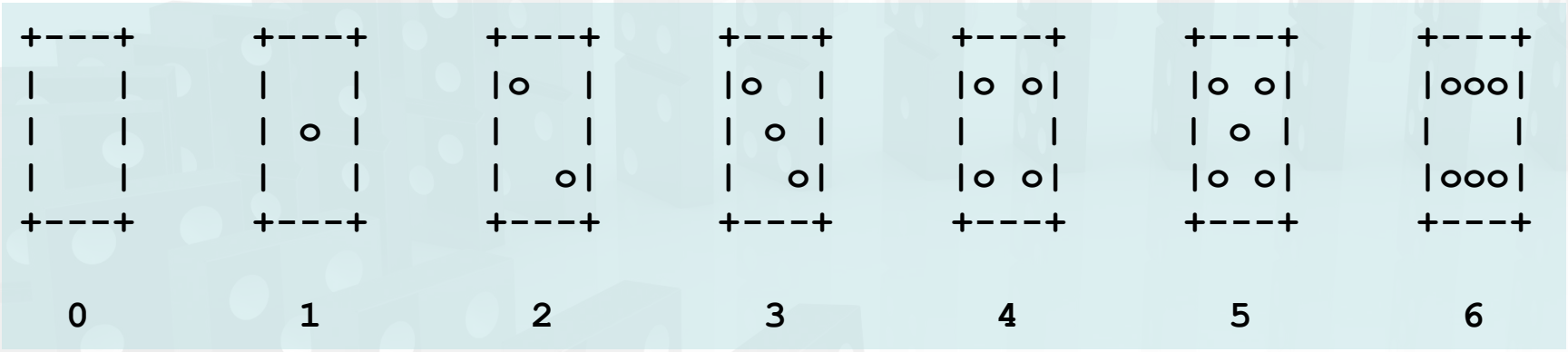

```
>>> from point3 import Point
>>> p, q = Point(3, 4), Point(9, 2)
>>> p
Point(3, 4)
>>> r = p + q
>>> print(r)
(12.00, 6.00)
```

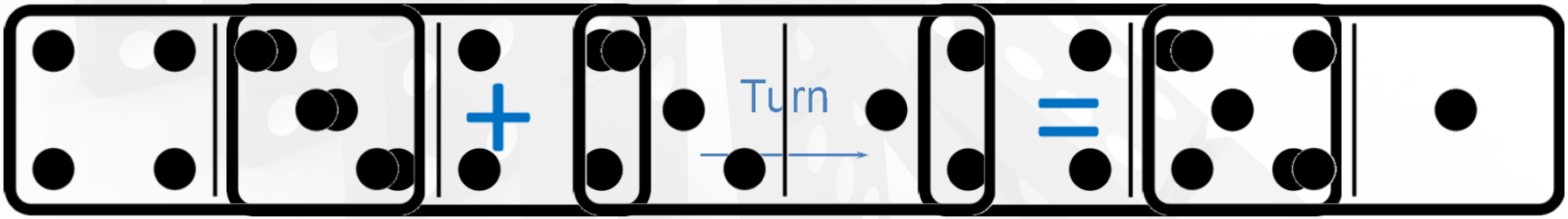
point3.py

```
class Point:
    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'Point({self.x}, {self.y})'
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
```

UNIVERSITEIT GENT

Domino tiles





Roman numerals

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

Classes: 'new style' vs 'old style'

- Python 2.x
  - differentiates between 'new style' and 'old style' classes
  - most differences are in advanced (technical) details

```
class Point:
    pass
```
old style

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

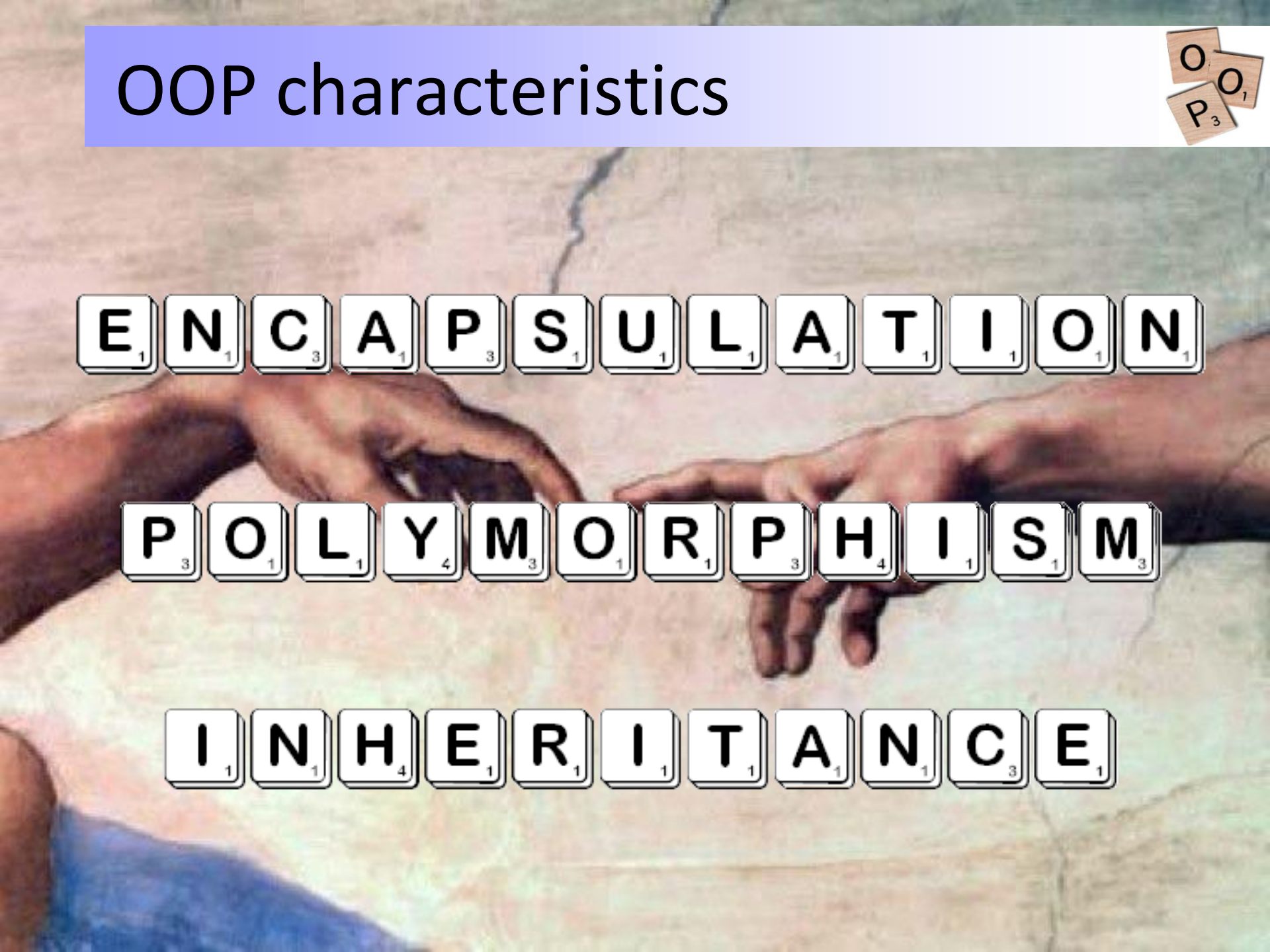
- Python 3.x
  - no difference between 'new style' and 'old style' classes

```
class Point:
    pass
```
new style

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

UNIVERSITEIT GENT

OOP characteristics



Homework (next lecture)

- course book
  - read chapter 12 (more on classes)
  - read chapter 13 (OOP development)
  - pay attention in particular to OOP terminology
- classroom exercises (series 10)
  - Roman numerals
  - Immune system

UNIVERSITEIT GENT

Homework (hands-on sessions)

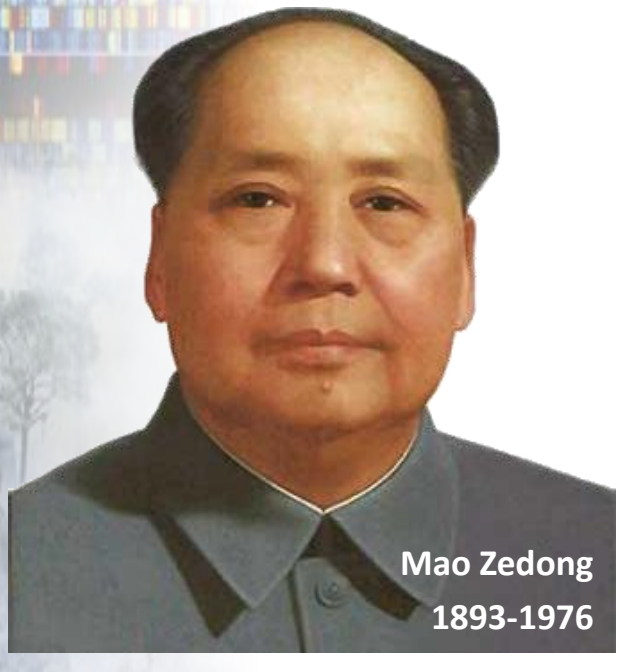
- mandatory exercises series 10 (OOP)
  - deadline: Tuesday, December 17, 2024
- second evaluation
  - Monday, December 16, 2024 (14:30-16:45)
  - Tuesday, December 17, 2024 (10:00-12:15)
  - register via Ufora groups

UNIVERSITEIT GENT

Questions or remarks?

UNIVERSITEIT GENT

The sky is the limit …



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

— Mao Zedong

UNIVERSITEIT GENT