



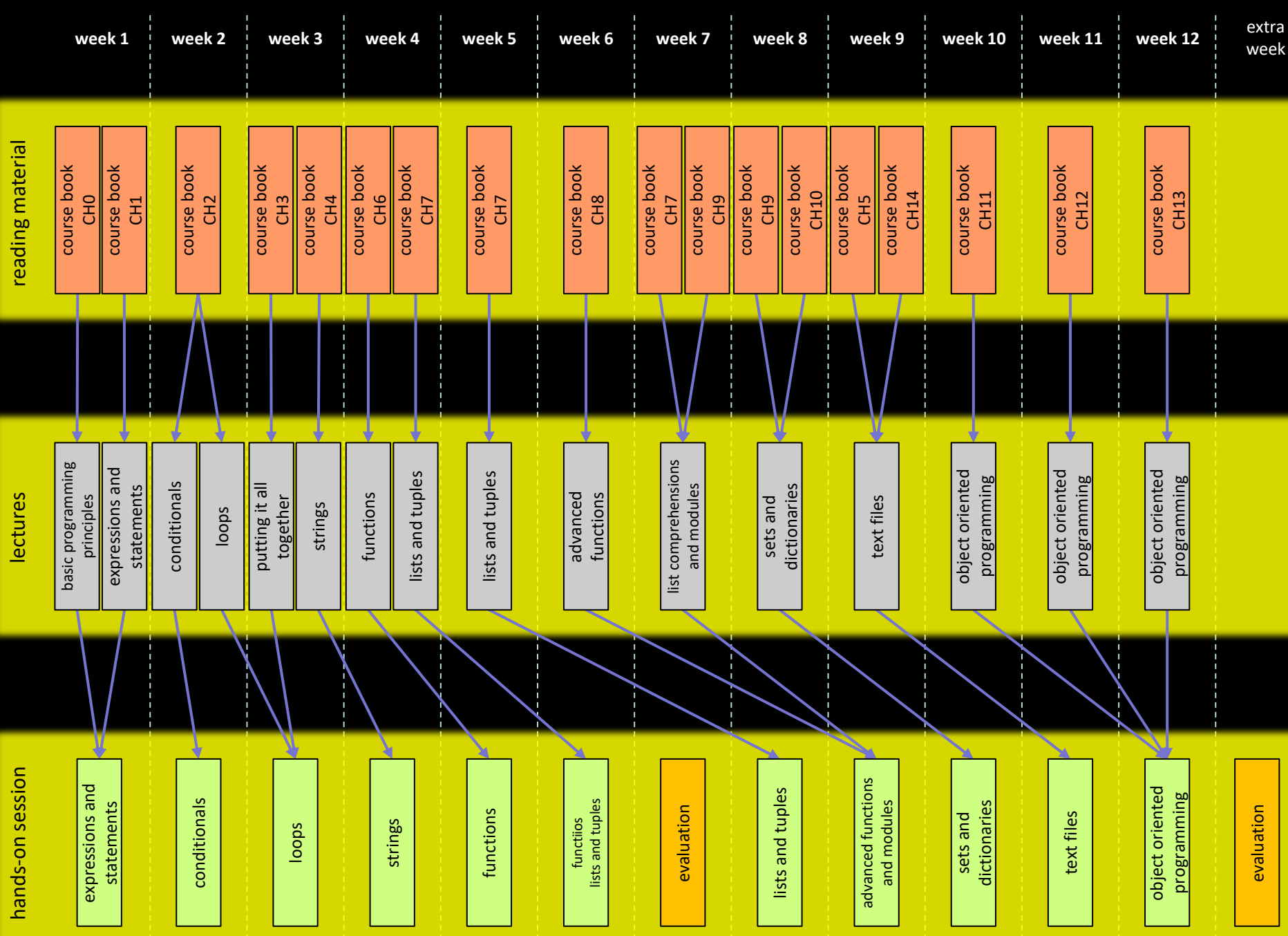
# Python Programming

## introduction to object-oriented programming

Prof. Dr. Peter Dawyndt

✉ [peter.dawyndt@ugent.be](mailto:peter.dawyndt@ugent.be)

🐦 [@dawyndt](https://twitter.com/dawyndt)



# 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*

A red Everlast boxing glove is shown from a side-on perspective. The glove is made of a shiny, reddish-brown material. The Everlast logo is visible on the wrist strap.

object-oriented  
programming

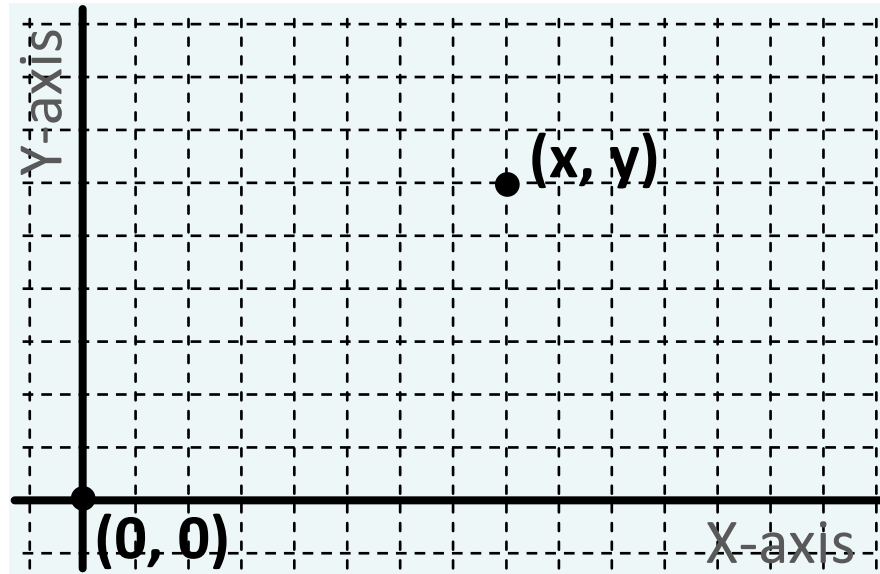
A red Everlast boxing glove is shown from a side-on perspective. The glove is made of a shiny, reddish-brown material. The Everlast logo is visible on the wrist strap.

procedural  
programming

# User-defined compound types



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



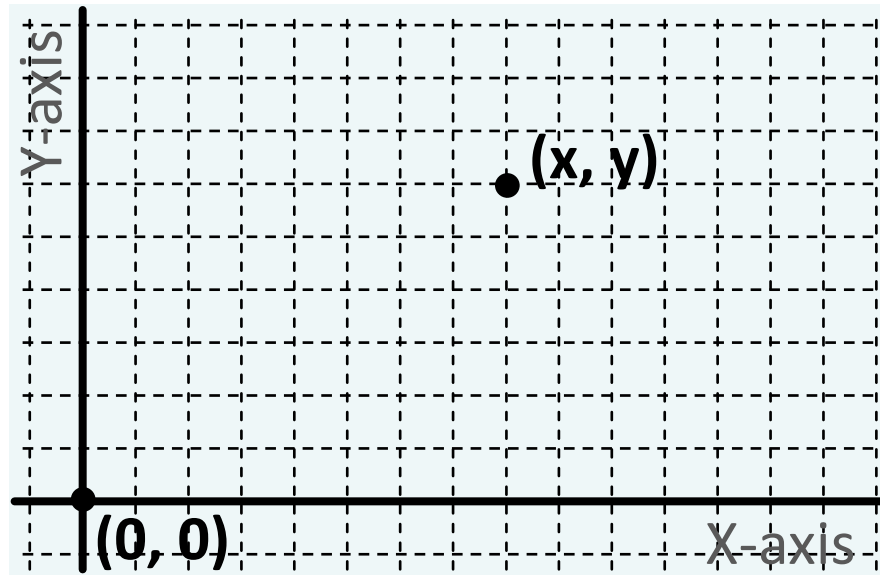
```
class Point:  
    pass
```



# User-defined compound types



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



```
class Point:
```

```
    '''Representation of two-dimensional points'''
```



# User-defined compound types



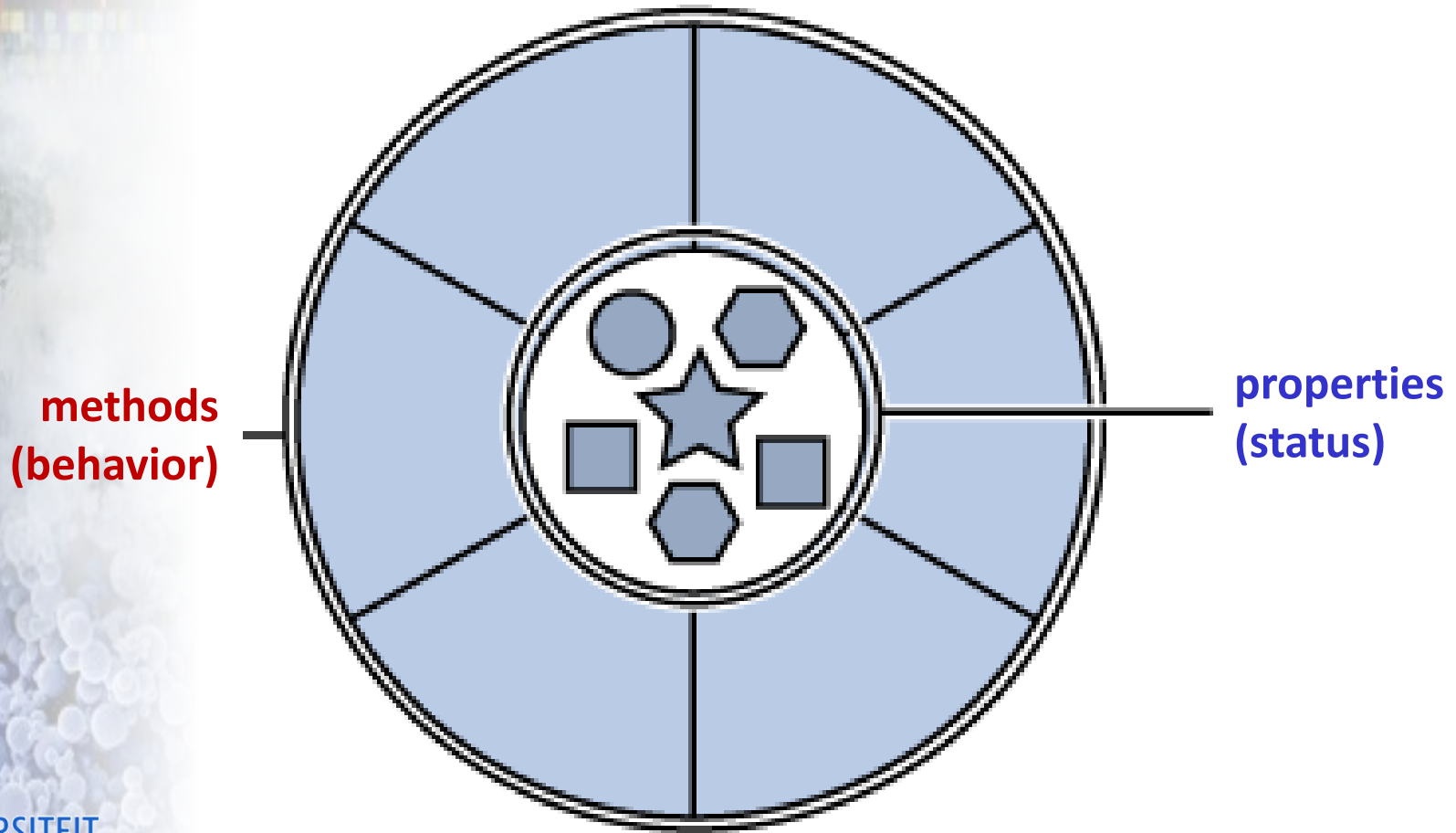
- **class:** defines a new (compound) data type
  - Python enables creation of *abstract data types* (ADTs)
    - "abstract" because they hide implementation details
    - programmers interact with them through a limited set of operations (*methods*), rather than by manipulating data directly (*encapsulation*)
      - fewer things 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

```
class Point:
```

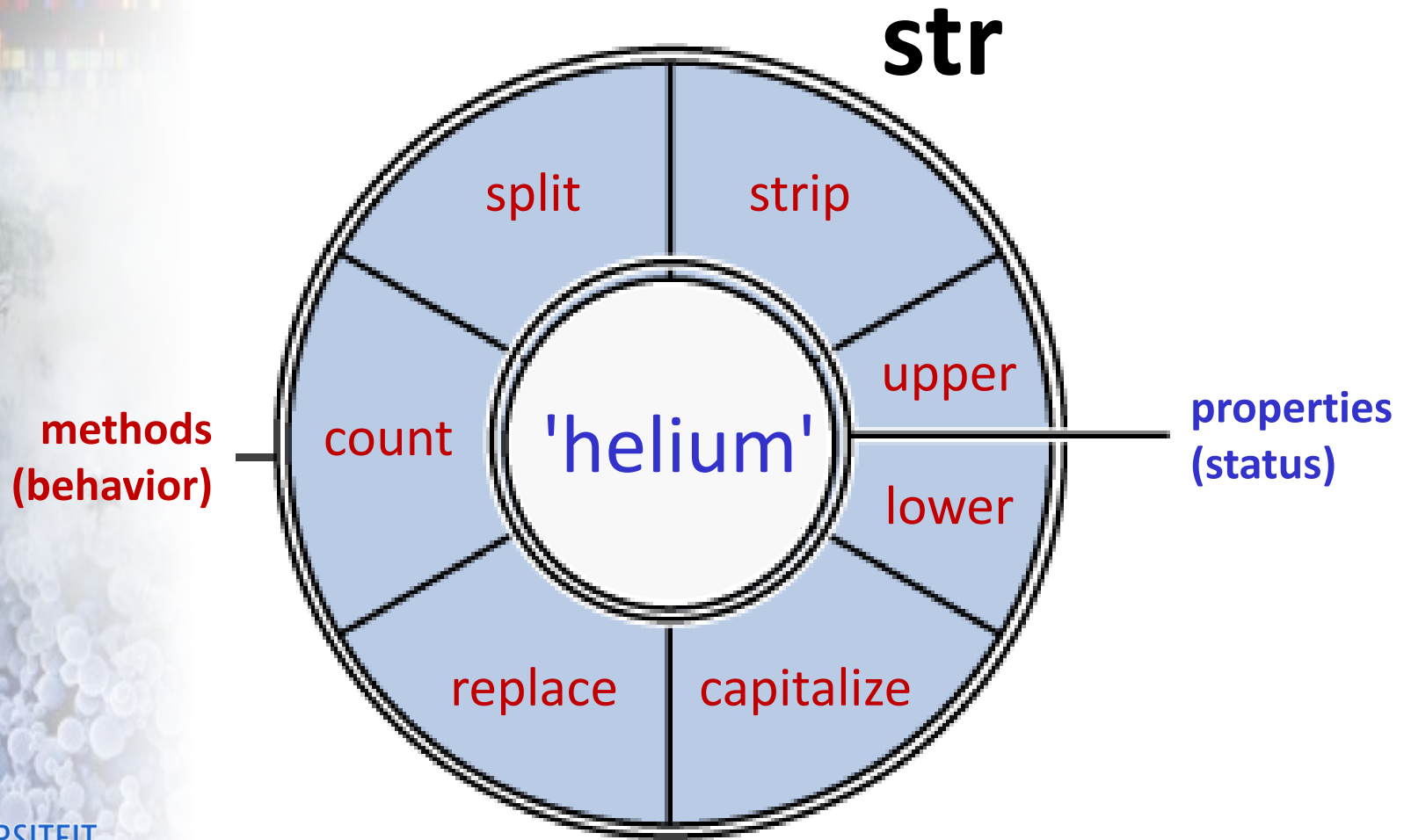
```
    '''Representation of two-dimensional points'''
```



# User-defined compound types



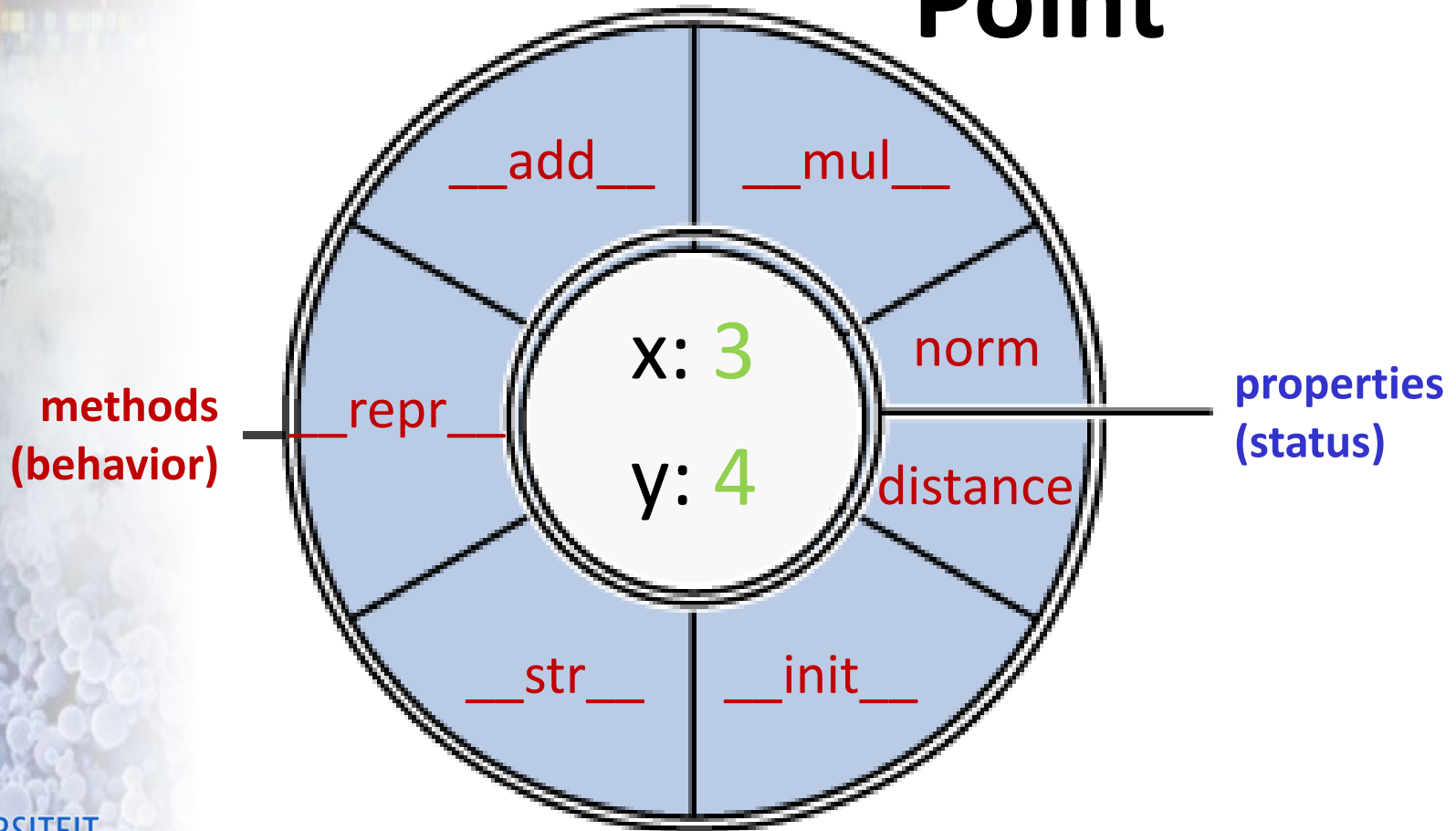
# User-defined compound types



# User-defined compound types



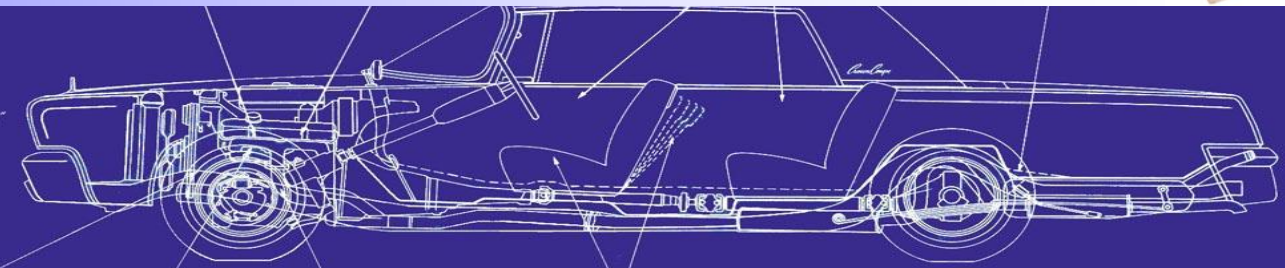
## Point



# Classes and instances



DEEP-SKIRTED ENGINE BLOCK GIVES ADDED STEEL SUPPORT TO BEARINGS AND CRANKSHAFT.



ALUMINUM PISTONS ARE REINFORCED WITH STEEL STRUTS FOR HEAT-EXPANSION CONTROL.

IMPERIAL'S EXECUTIVE, AIRCRAFT TYPE SEATS ARE INDIVIDUALLY POWER-ADJUSTED. THE SEAT ON THE PASSENGER'S SIDE CAN BE TILTED BACK INTO ANY ONE OF FIVE DIFFERENT POSITIONS.

FORGED-STEEL CRANKSHAFT, DYNAMICALLY BALANCED... HAS MICROFINISHED BEARING SURFACES TO REDUCE FRICTION, MINIMIZE WEAR.

IMPERIAL'S 3-SPEED TORQUE-CONVERTER AUTOMATIC TRANSMISSION IS ABOUT THE MOST RESPONSIVE AVAILABLE IN ANY AUTOMOBILE (IT'S A BETTER VERSION OF THE TYPE MANY DRAG-STRIP DRIVERS PREFER.)

TWO-PIECE DRIVESHAFT WITH TWO CONSTANT-VELOCITY JOINTS ACCOMMODATES IMPERIAL'S LONG WHEELBASE... KEEPS DRIVESHAFT RUNNING TRUE, MINIMIZES VIBRATION.

IMPERIAL'S NEW, LOWER PROFILE TIRES OFFER IMPROVED HANDLING, REDUCE TIRE SQUEAL, INSURE LONGER TREAD LIFE.

## SPECIFICATIONS

**ENGINE:** OVERHEAD VALVE 90 DEGREE V-8, 413 CU. IN. DISPLACEMENT, 10.1 TO 1 COMPRESSION RATIO, 340 HP @ 4600 RPM, TORQUE, 470 LB.-FT. @ 2800 RPM.

**FUEL SYSTEM:** FOUR-BARREL CARBURETOR WITH MECHANICALLY CONTROLLED SECONDARY BARRELS, AUTOMATIC CHOKE, POSITIVE THROTTLE RETURN. FUEL TANK CAPACITY, 23 GALLONS.

**ELECTRICAL SYSTEM:** 12 VOLT BATTERY, 78 PLATES, 70 AMP-HR. RATING, 35-AMP ALTERNATOR (46 AMP WITH AIR CONDITIONING).

**TRANSMISSION:** TORQUEFLITE AUTOMATIC WITH COLUMN-MOUNTED SELECTOR LEVER. THREE-SPEED PLANETARY GEAR SET WITH INCREASED HELIX ANGLE. TRANSMISSION BREAK-AWAY RATIO... 4.90 TO 1. IMPROVED TORQUE CONVERTER.

**FRAME:** FOR CLOSED MODELS... PERIMETER-TYPE LADDER FRAME WITH SIX CROSS-MEMBERS, FULL-LENGTH OUTBOARD SIDE RAILS.

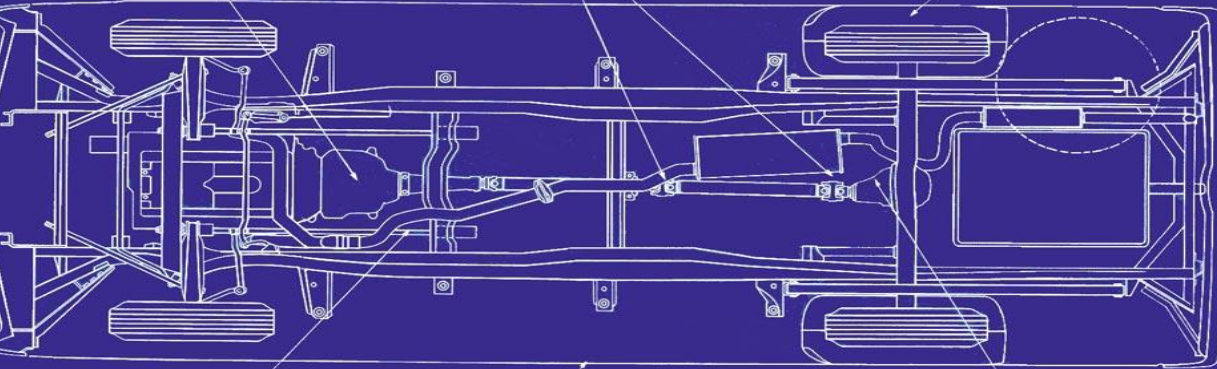
**SUSPENSION:** CHROME-STEEL TORSION-BAR INDEPENDENT FRONT WHEEL SUSPENSION. BALL-JOINT PIVOTS. HOTCHKISS DRIVE. LEAF-TYPE REAR SPRINGS, 60 IN. LONG, MOUNTED 45 1/2 INCHES APART. ORIFLOW SHOCK ABSORBERS AT ALL FOUR WHEELS. REAR AXLE STABILIZER STRUTS.

**STEERING:** FULL-TIME POWER STEERING 35 TURNS, FULL LEFT TO FULL RIGHT... SYMMETRICAL IDLER-ARM STEERING LINKAGE. HYDRAULIC AND MECHANICAL STEERING REACTION SYSTEMS. ADJUSTABLE STEERING WHEEL OPTIONAL AT EXTRA COST.

**BRAKES:** SELF-ADJUSTING POWER BRAKE SYSTEM, FLARED BRAKE DRUMS, BONDED LININGS; TOTAL EFFECTIVE BRAKING AREA, 287 SQ. IN. MECHANICAL PARKING BRAKE WITH AUTOMATIC RELEASE.

**WHEELS AND TIRES:** LOW PROFILE TYPE 9.15 x 15, TUBELESS TIRES ON SAFETY-RIM WHEELS. STAINLESS STEEL WHEEL COVERS.

**DIMENSIONS:** FOR CLOSED MODELS... WHEELBASE, 129 IN. FRONT TREAD, 61.5 IN.; REAR, 61.7 IN. OVERALL LENGTH, 227.0 IN. WIDTH, 80.0 IN. HEIGHT (LOADED) 57.2 IN.



HIGH-CHROME STEEL TORSION BARS SOAK UP ROUGHEST ROAD SHOCKS BEFORE THEY REACH THE CAR BODY.

THE BODY OF AN IMPERIAL UNDERGOES A 15-STEP RUST-PREVENTIVE TREATMENT. SOME OF THE INITIAL SOLUTIONS USED, ACTUALLY INCREASE THE STEEL'S RESISTANCE TO CORROSION, BY CHANGING THE MOLECULAR SURFACE STRUCTURE.

DIFFERENTIAL GEARS ARE EXTRA STRONG TO HANDLE HIGH ENGINE TORQUE... ARE HELICAL CUT TO PROVIDE BETTER GEAR-MATING SURFACES, QUIETER OPERATION.

SCALE: 1" = 1' 3"

DRAWING:

*Walt Disney*

129" WHEELBASE IMPERIAL CROWN COUPE

**THE INCOMPARABLE  
IMPERIAL  
VINTAGE 1965**

# 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

```
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'>
```

# Classes and instances



- think of a class as a blueprint for making objects

- class **Point** is a factory for making points
- class **Point** isn't an instance of a point, but it has the machinery to make point instances



*INSTANCE*

*INSTANCE*

*INSTANCE*

*INSTANCE*

*INSTANCE*

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



# Properties



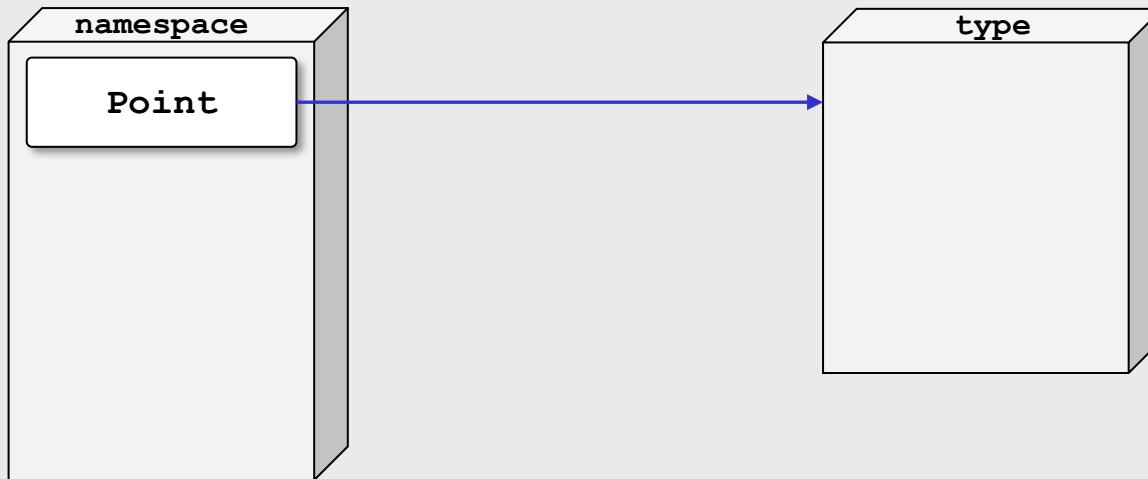
- object instances both have state and behavior
  - attributes of an instance determine its state
  - instances have their own *namespace* (just like modules)
  - *attribute*: name in namespace (instances and modules)
    - use *dot notation* to access name from namespace
    - similar to module syntax: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

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

# Properties



*objects (main memory)*

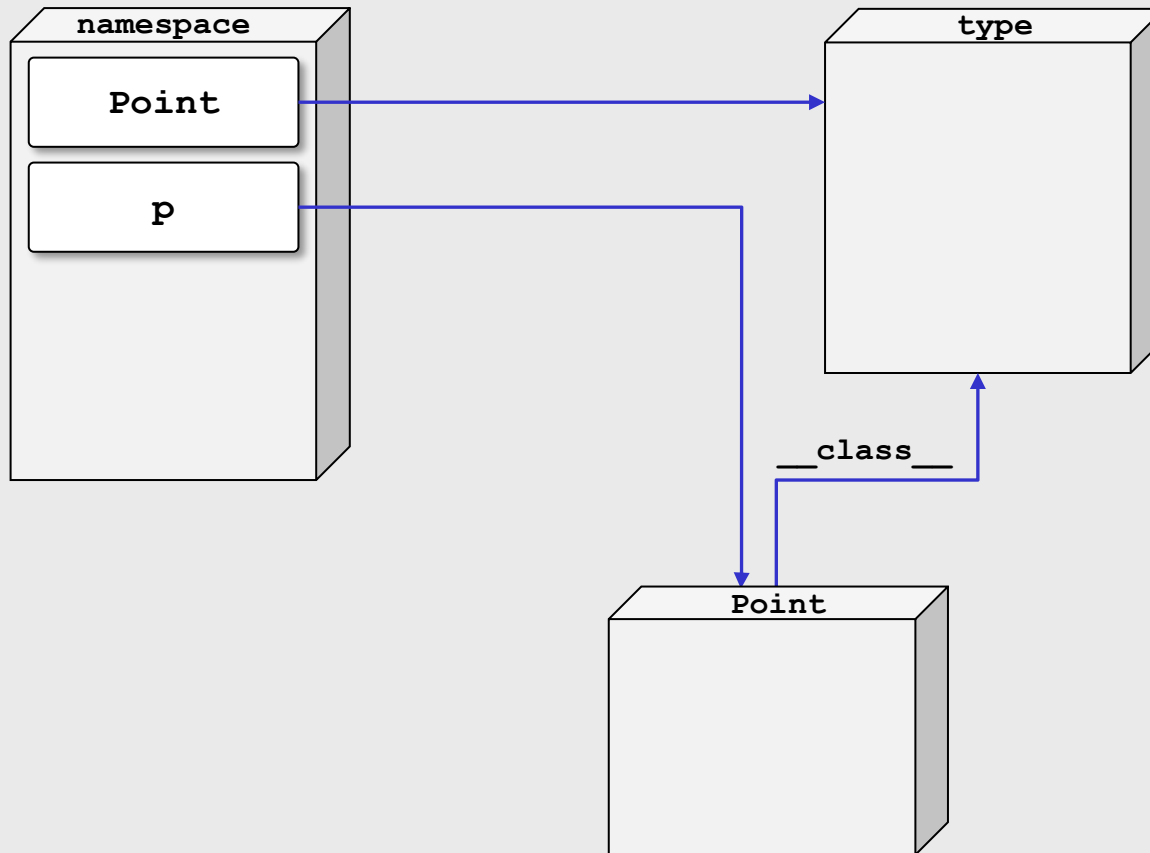


```
>>> class Point: pass
```

# Properties



*objects (main memory)*

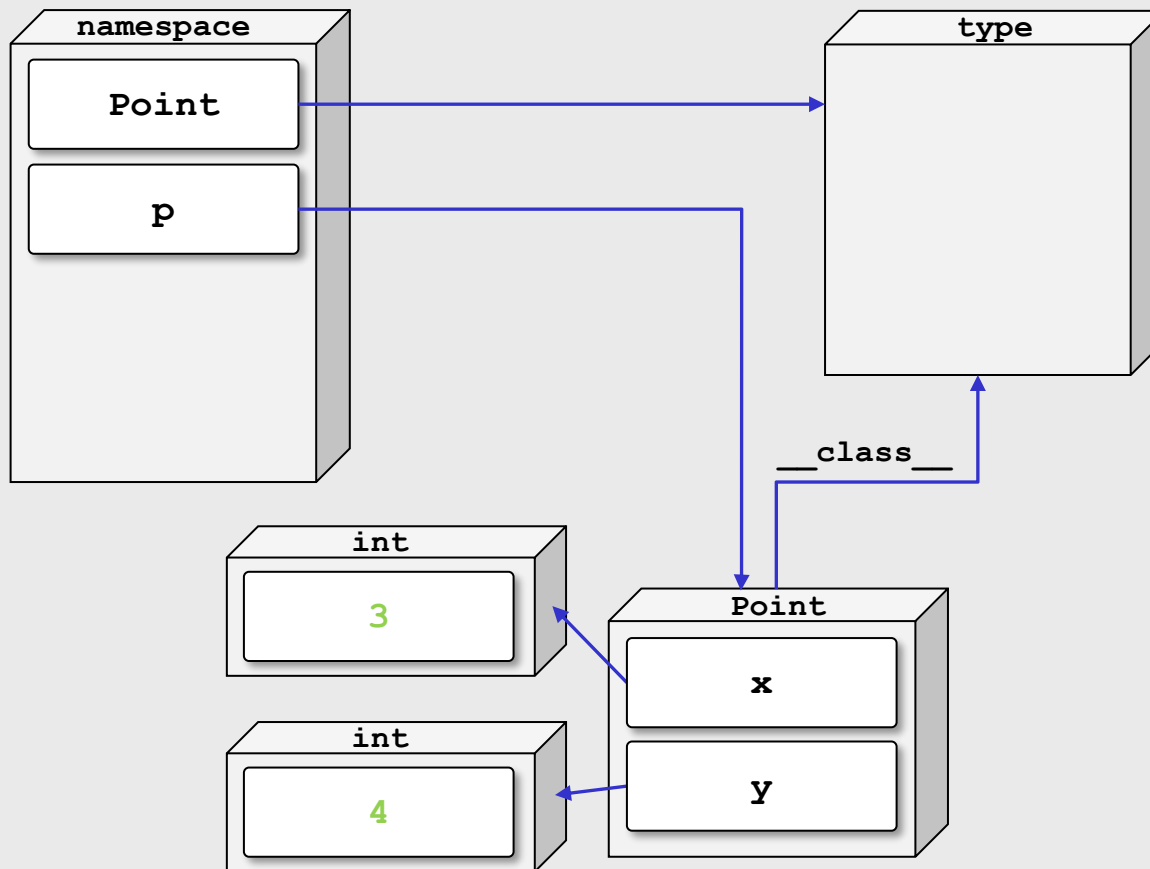


```
>>> p = Point()
```

# Properties



*objects (main memory)*



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

# Properties



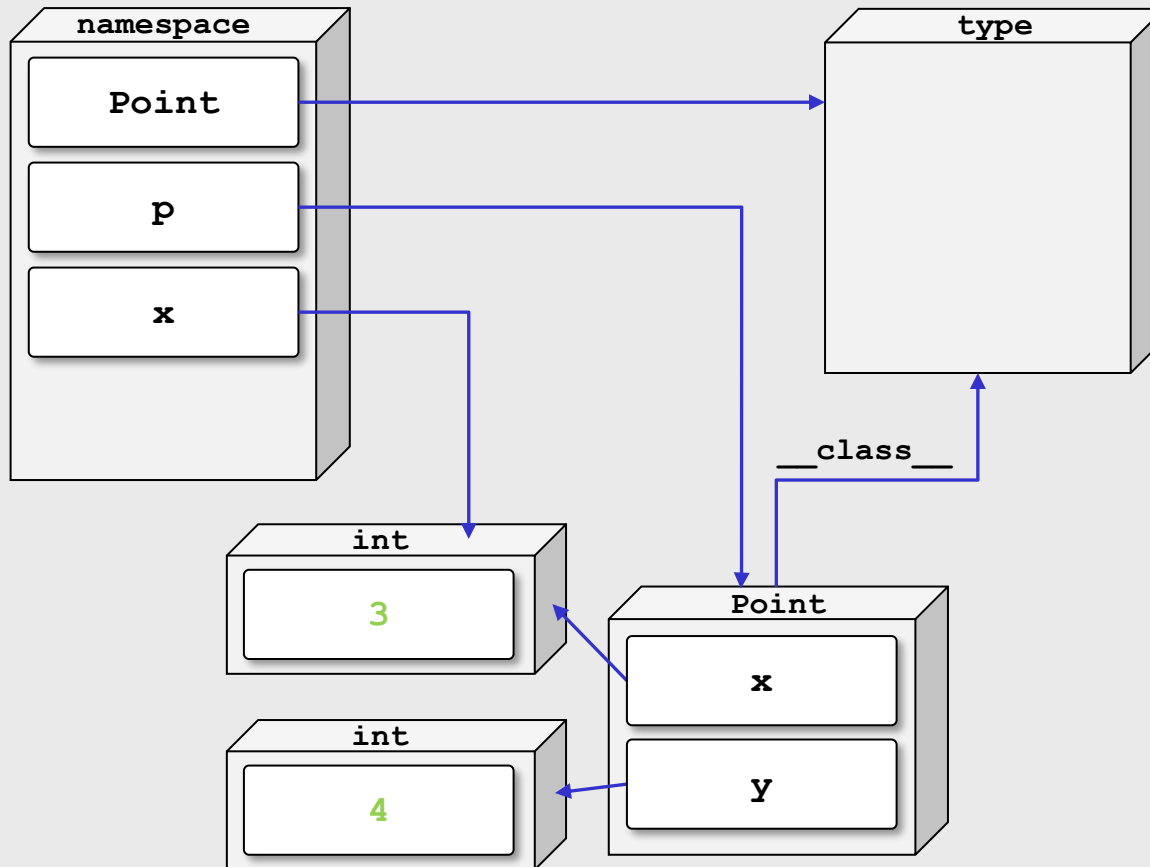
```
>>> class Point:
...     pass
...
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.y
4
>>> x = p.x
```



# Properties



*objects (main memory)*



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

# Properties



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

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

```
>>> p.x = 3
```

```
>>> p.y = 4
```

```
>>> p.y
```

4

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

```
>>> x
```

3

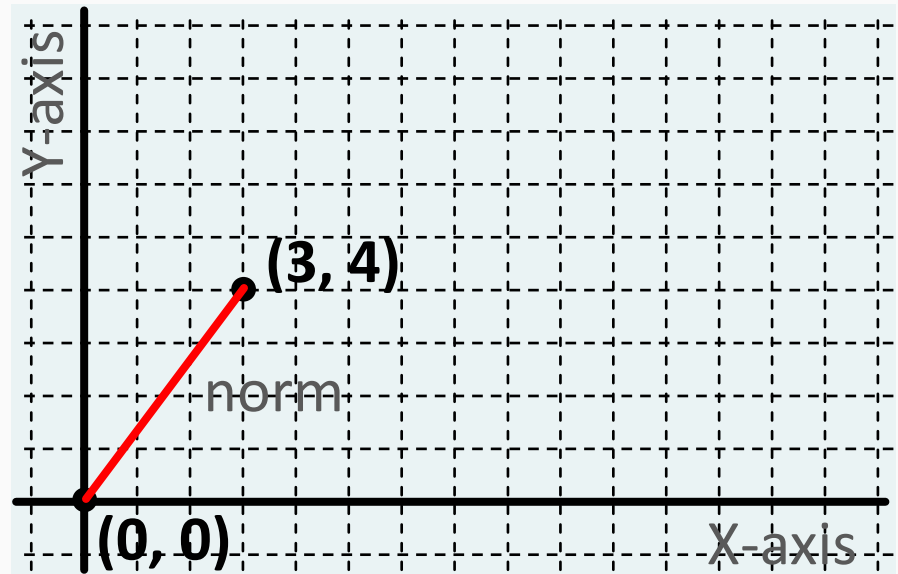
```
>>> print(f'({p.x:.2f}, {p.y:.2f})')
```

(3.00, 4.00)

```
>>> norm = (p.x ** 2 + p.y ** 2) ** 0.5
```

```
>>> norm
```

5.0



# 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()
```



# 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)`

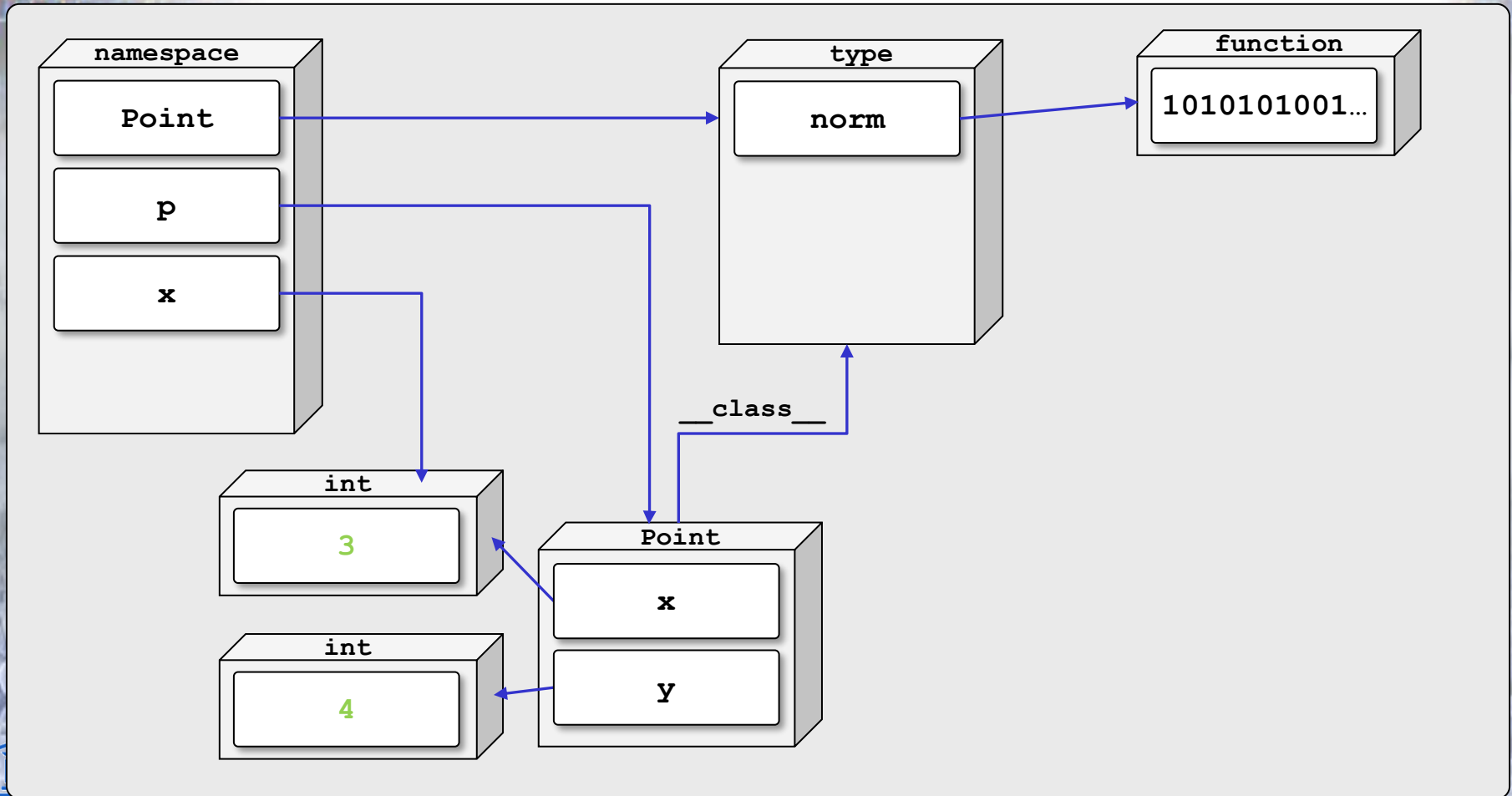
```
>>> class Point:
...     def norm(self):
...         return (self.x**2 + self.y**2) ** 0.5
...
>>> p = Point()
>>> p.x, p.y = 3, 4
>>> p.norm()
5.0
>>> Point.norm(p)
5.0
```



# Methods



*objects (main memory)*

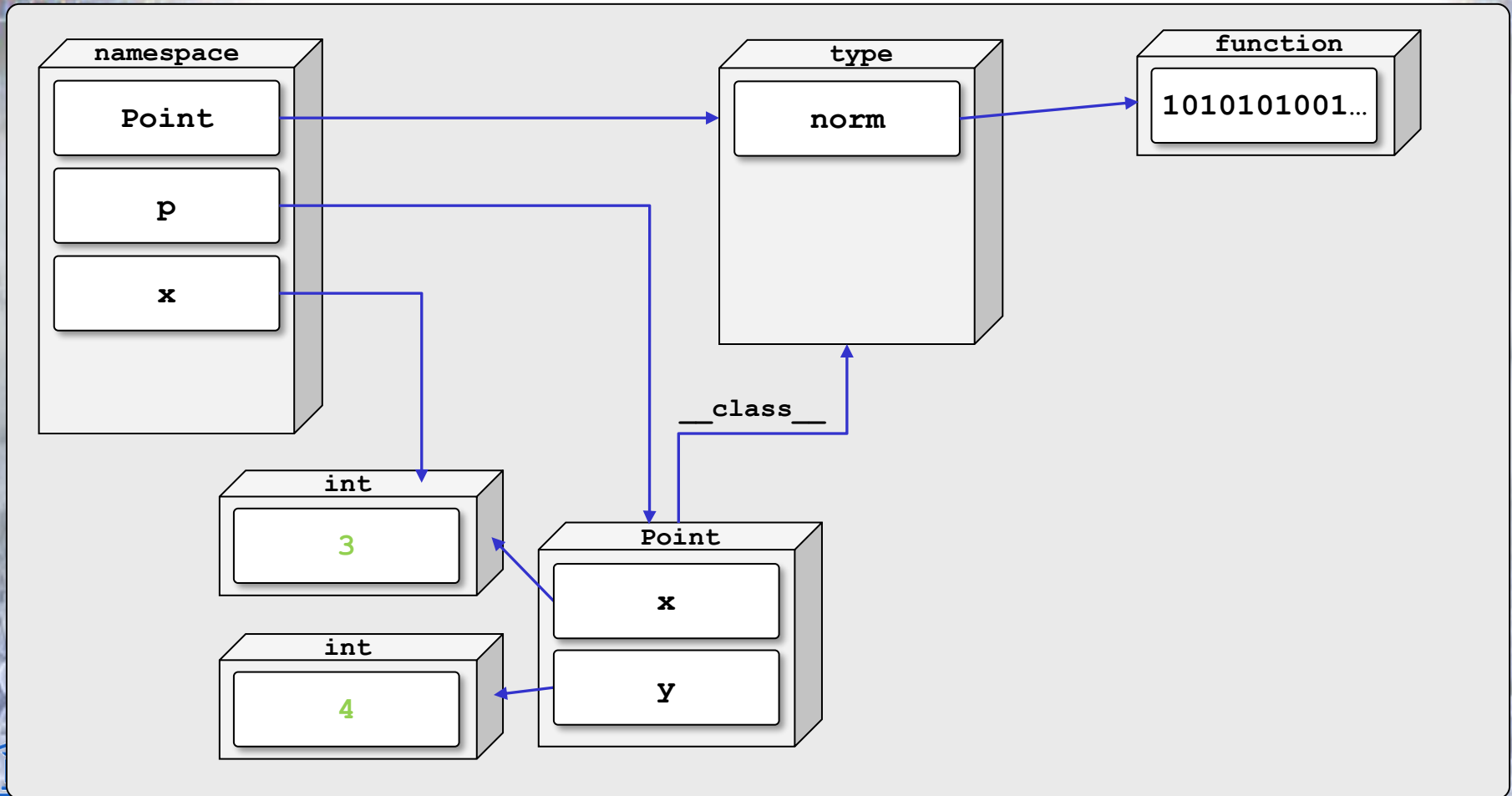


```
>>> p.norm()
```

# Methods



*objects (main memory)*



```
>>> 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'
```



# 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
```



# The initialisation method



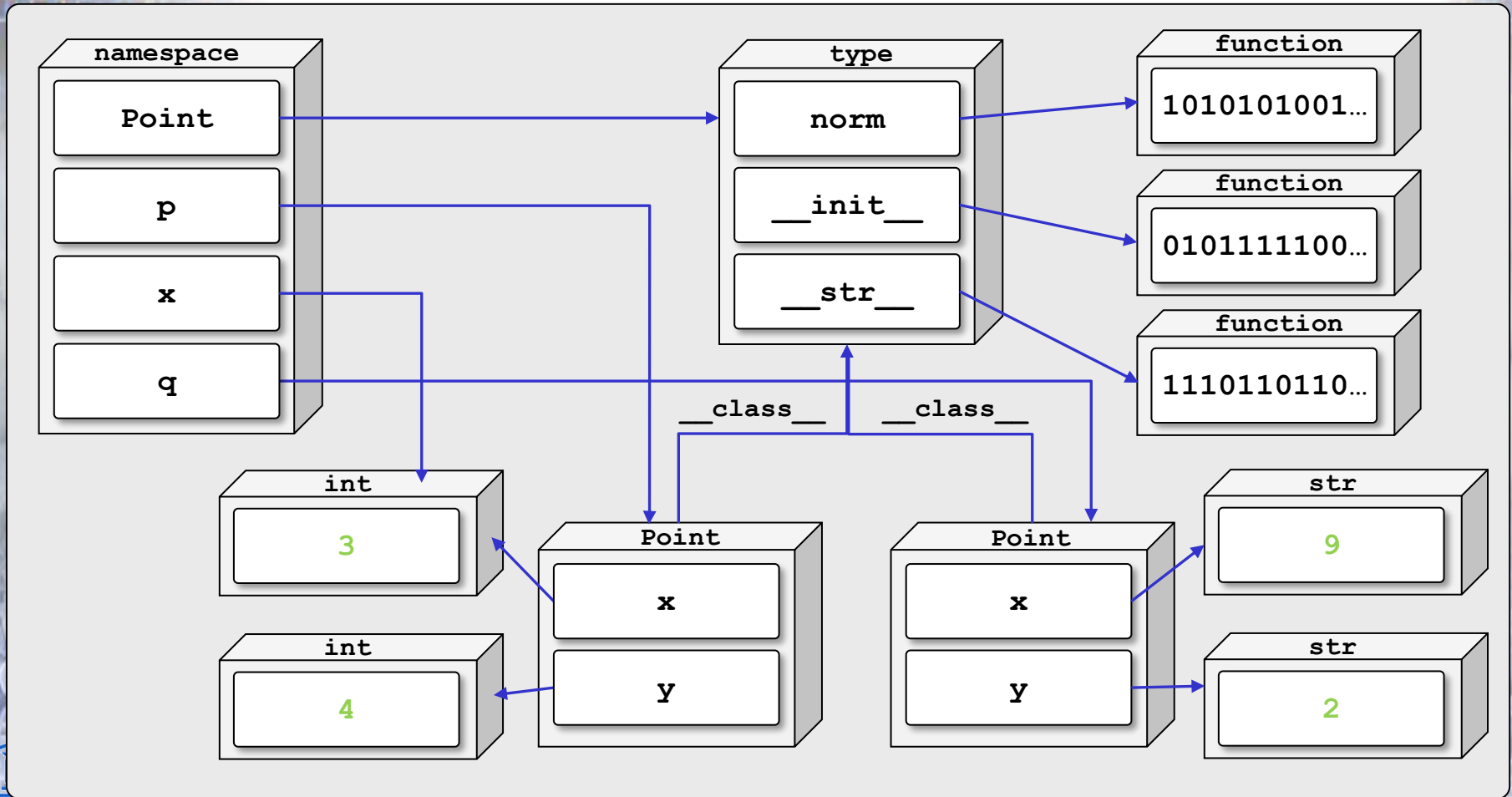
```
>>> from point1 import Point
>>> p = Point(3, 4)
>>> p.x
3
>>> p.y
4
>>> p.norm()
5.0
>>> x = p.x
>>> q = Point(9, 2)
>>> q.x
9
>>> q.y
2
>>> q.norm()
9.2195444572928871
```



# The initialisation method



*objects (main memory)*



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

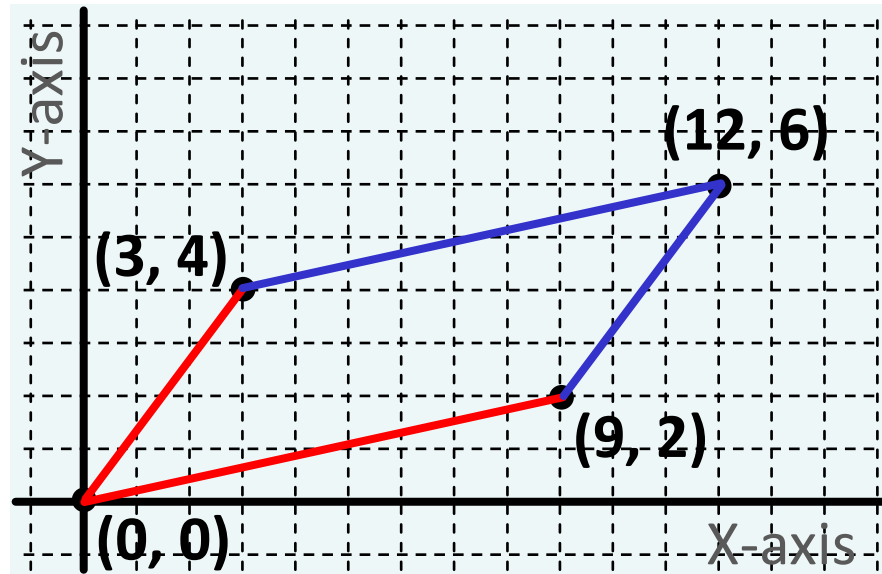
# Instances as parameters



- 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)
```

# Instances as parameters



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



# 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
```



# 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)
```



# 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

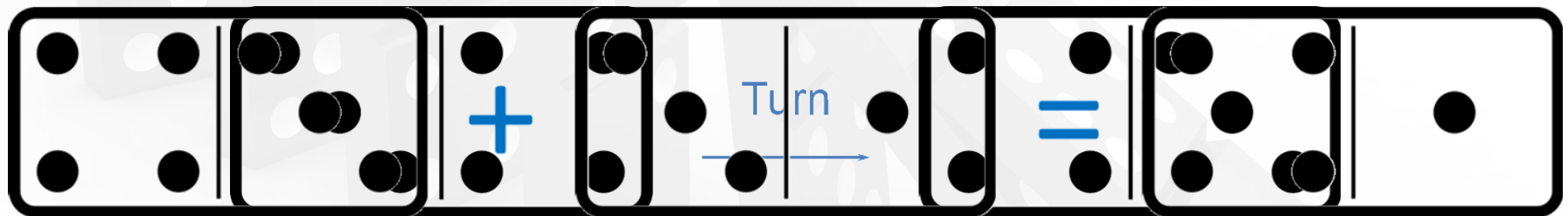
```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def norm(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5
    def __str__(self):
        return f'({self.x:.2f}, {self.y:.2f})'
    def __repr__(self):
        return f'Point({self.x}, {self.y})'
   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



# Domino tiles



|       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|
| +---+ | +---+ | +---+ | +---+ | +---+ | +---+ | +---+ |
|       |       | o     | o     | o o   | o o   | o o o |
|       | o     |       | o     |       | o     |       |
|       |       | o     | o     | o o   | o o   | o o o |
| +---+ | +---+ | +---+ | +---+ | +---+ | +---+ | +---+ |
| 0     | 1     | 2     | 3     | 4     | 5     | 6     |



# 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

I VIEW X-RAYS



LUCKY COWS  
DRINK MILK



# 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*

# OOP characteristics



E N C A P S U L A T I O N

P O L Y M O R P H I S M

I N H E R I T A N C E

# 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*



# 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

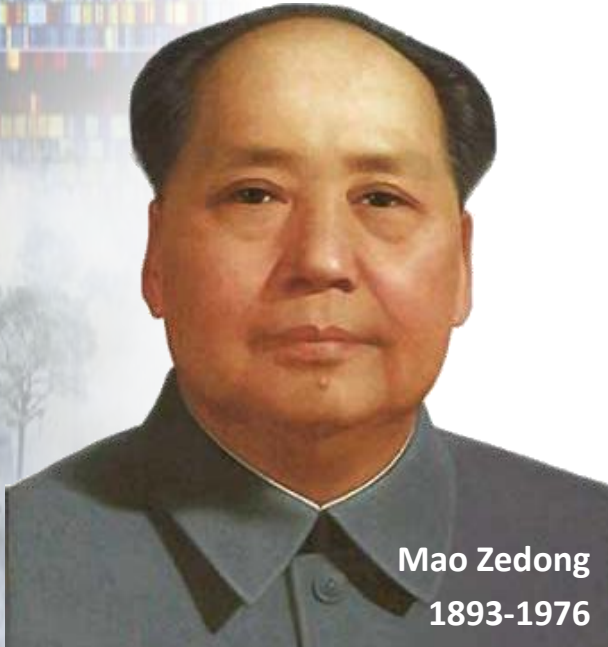


# Questions or remarks?



UNIVERSITEIT  
GENT

# The sky is the limit ...



Mao Zedong  
1893-1976

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

— Mao Zedong

