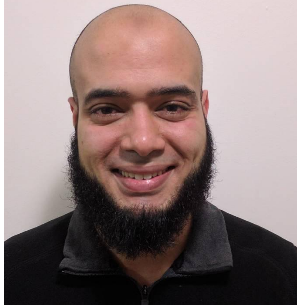
Python Programming

Special Methods: Arithmetic, Compound, Comparison, Unary

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

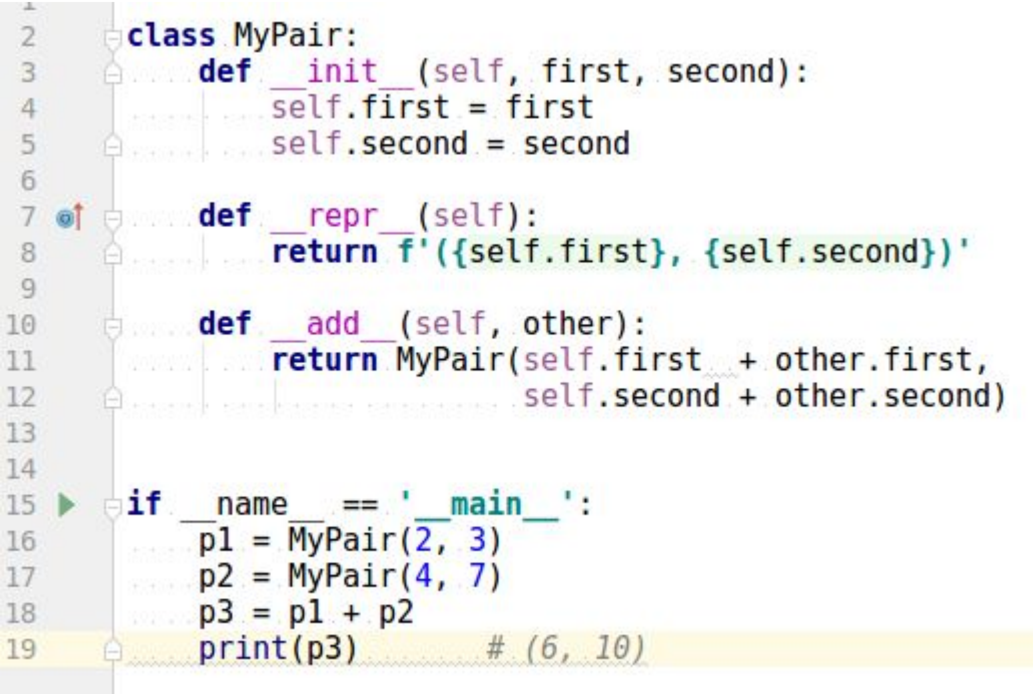


Special Methods

- We know we can do A + B
  - Both can be strings or integers
- What if I have my user-defined class and want to support such behaviour?
  - E.g. Create vector or matrix class
- We do that through overriding specific dunder methods
  - We already studied some of them

Arithmetic Operator +

- By adding **__add__** dunder, we can allow addition of our class’s object and something else (whatever class, no restriction)



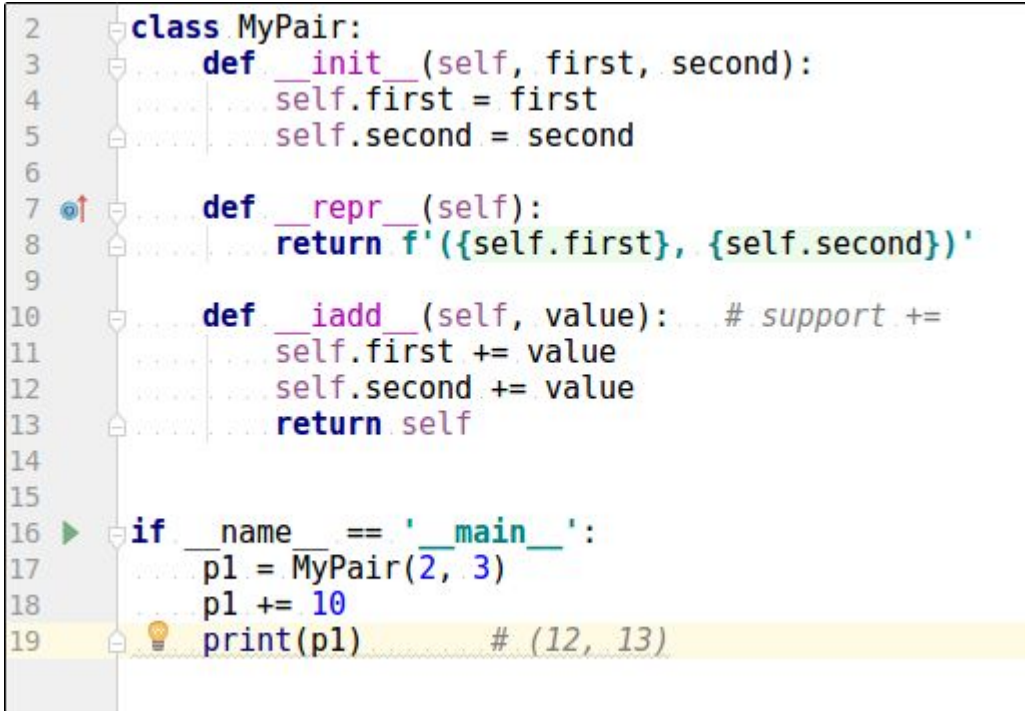
```python
class MyPair:
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def __repr__(self):
        return f'({self.first}, {self.second})'

    def __add__(self, other):
        return MyPair(self.first + other.first,
                      self.second + other.second)


if __name__ == '__main__':
    p1 = MyPair(2, 3)
    p2 = MyPair(4, 7)
    p3 = p1 + p2
    print(p3)        # (6, 10)
```

Arithmetic Operators

- You can do the same logic with the other operators
- - ⇒ __sub__
- * ⇒ __mul__
- / ⇒ __truediv__
- // ⇒ __floordiv__
- % ⇒ __mod__
- ** ⇒ __pow__
- @ ⇒ __matmul__ (matrix multiplication, as in numpy)

Compound Operator +=

- In this operator, we change the object itself NOT create a new one
- You should return self to be assigned to the caller object again
- Tip: iadd = in-place add



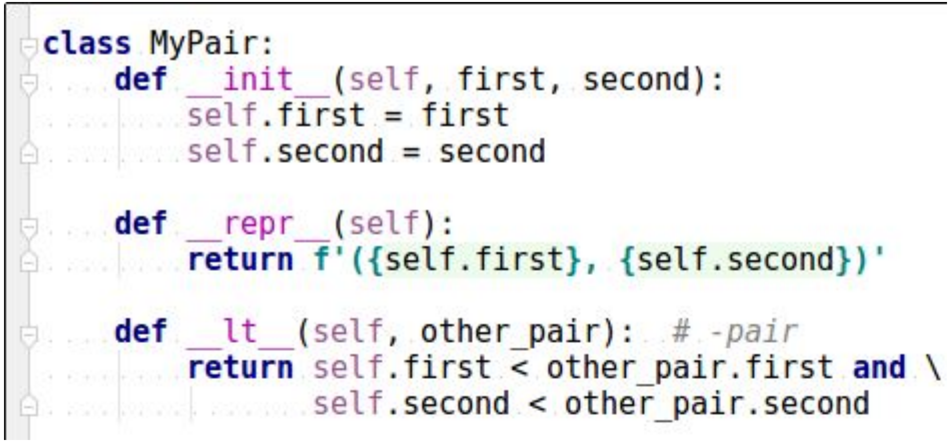
```
class MyPair:
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def __repr__(self):
        return f'({self.first}, {self.second})'

    def __iadd__(self, value):   # support +=
        self.first += value
        self.second += value
        return self


if __name__ == '__main__':
    p1 = MyPair(2, 3)
    p1 += 10
    print(p1)        # (12, 13)
```

Compound Operators

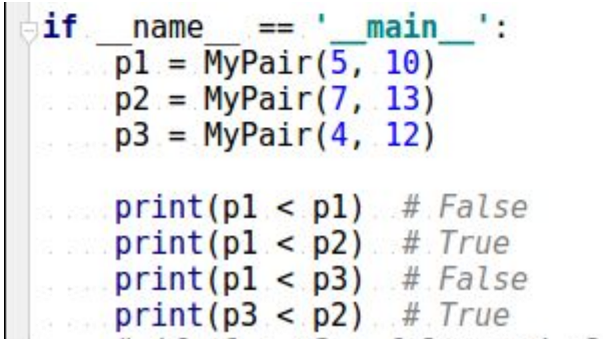
- You can do the same logic with the other operators
- -= ⇒ __isub__
- *= ⇒ __imul__
- /= ⇒ __itruediv__
- //= ⇒ __ifloordiv__
- %= ⇒ __imod__
- **= ⇒ __ipow__
- @= ⇒ __imatmul__ (matrix multiplication, as in numpy)

Comparison operator <

- With __lt__ we can support less than between 2 objects
- This allows us to sort list of employees e.g. based on age & salary



```python
class MyPair:
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def __repr__(self):
        return f'({self.first}, {self.second})'

    def __lt__(self, other_pair):  # -pair
        return self.first < other_pair.first and \
               self.second < other_pair.second
```



```python
if __name__ == '__main__':
    p1 = MyPair(5, 10)
    p2 = MyPair(7, 13)
    p3 = MyPair(4, 12)

    print(p1 < p1)  # False
    print(p1 < p2)  # True
    print(p1 < p3)  # False
    print(p3 < p2)  # True
```

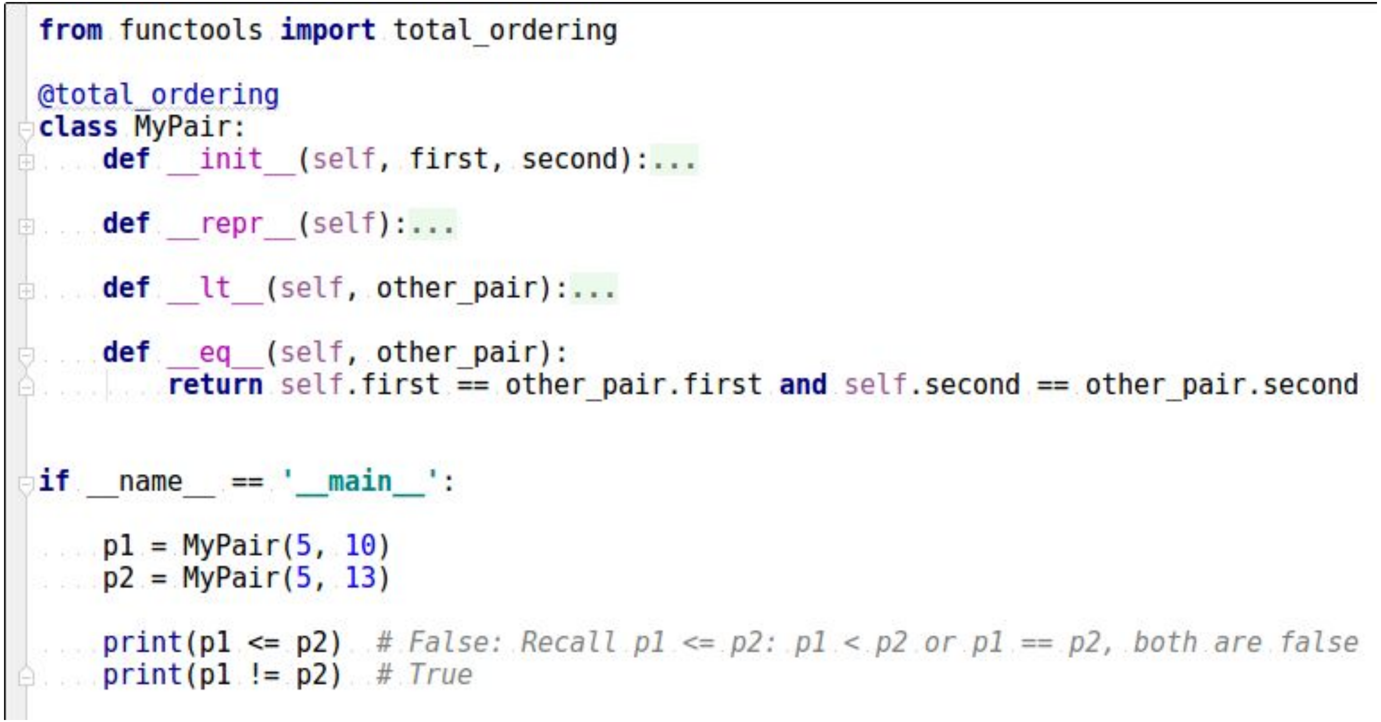
Comparison Operators

- You can do the same logic with the other operators
    - If you tried to compare without defining, you **may** get error
- <= ⇒ __le__
- == ⇒ __eq__
- != ⇒ __ne__
- > ⇒ __gt__
- >= ⇒ __ge__

Comparison operator: lt and eq is enough

- Mathematically, with **only < operator and eq**, we can know for the other comparisons over objects
  - p1 != p2 is same as not (p1 == p2)
  - p1 > p2 is same as p2 < p1 and so on
- The functools module is for higher-order functions: functions that act on or return other functions. From it we have **total_ordering**
  - Class decorator that fills in missing ordering methods
  - That is you define a few, and all others are DONE for you
  - You can only support le. But **default eq** depends on membership (p1 is p2)
  - Practically: **providing both lt and eq** is enough to avoid mistakes!

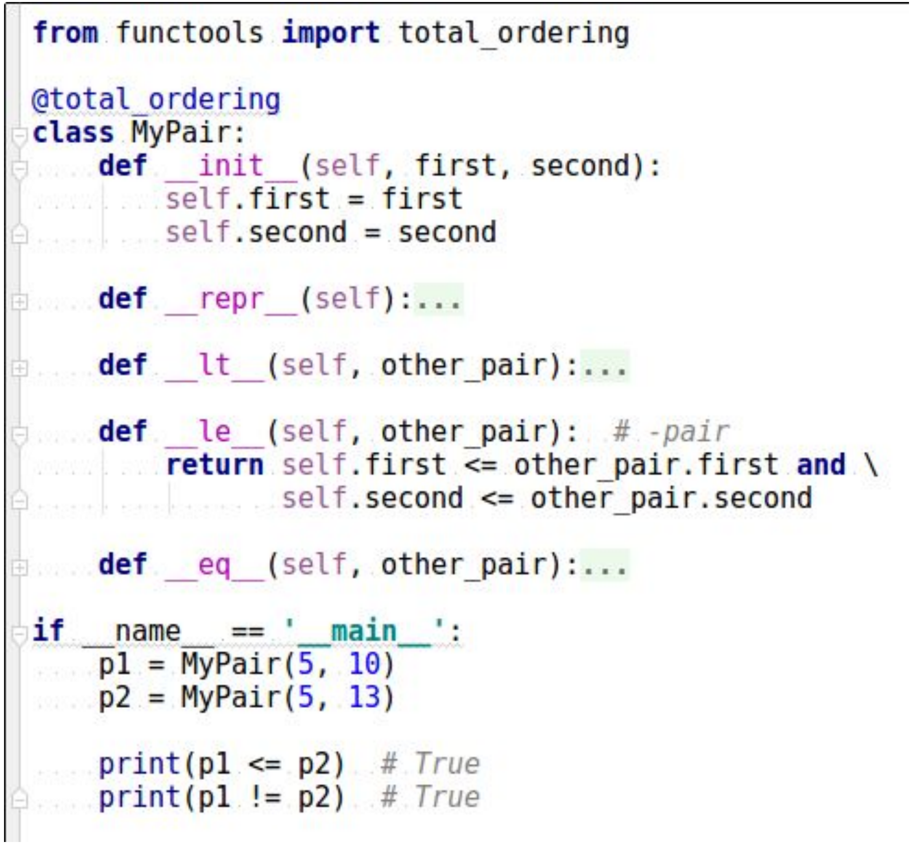
Comparison operator: Total Ordering Decorator



```python
from functools import total_ordering

@total_ordering
class MyPair:
    def __init__(self, first, second):...

    def __repr__(self):...

    def __lt__(self, other_pair):...

    def __eq__(self, other_pair):
        return self.first == other_pair.first and self.second == other_pair.second


if __name__ == '__main__':

    p1 = MyPair(5, 10)
    p2 = MyPair(5, 13)

    print(p1 <= p2)  # False: Recall p1 <= p2: p1 < p2 or p1 == p2, both are false
    print(p1 != p2)  # True
```

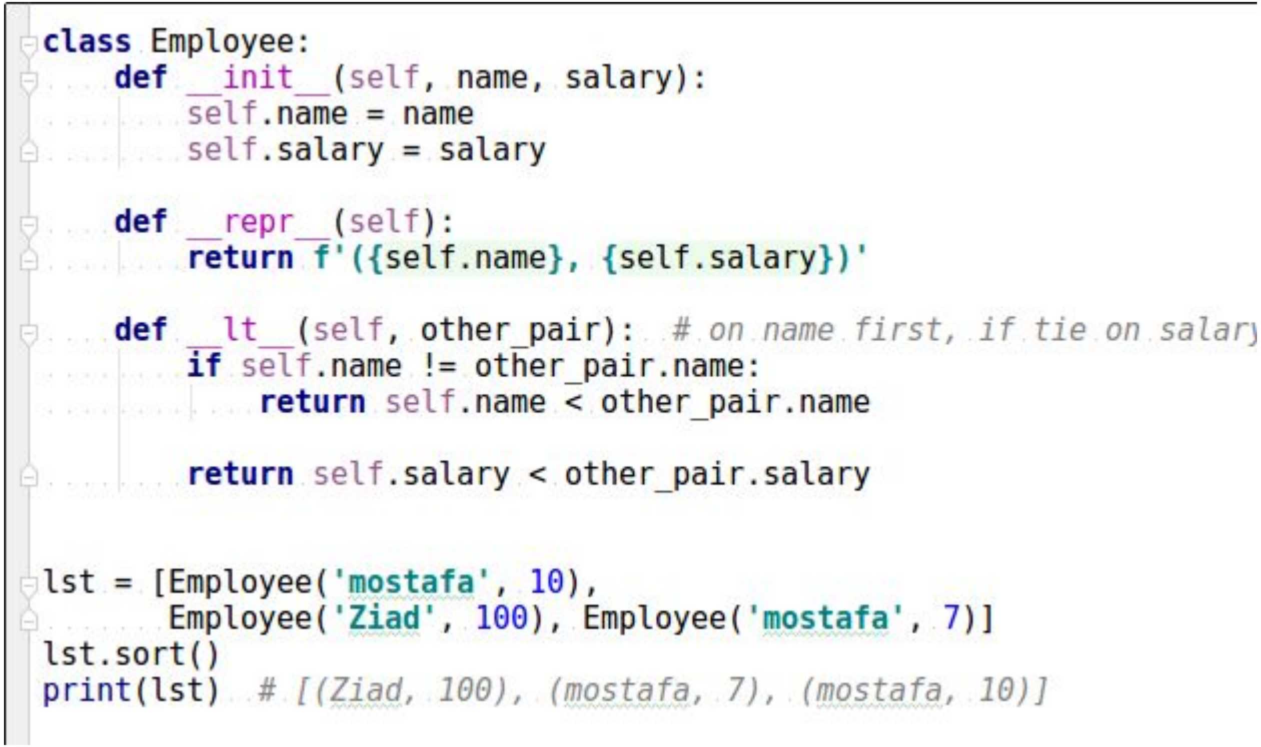
Override what u need

- If generating missing functions may break your semantic, just overwrite yours



```python
from functools import total_ordering

@total_ordering
class MyPair:
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def __repr__(self):...

    def __lt__(self, other_pair):...

    def __le__(self, other_pair):  # -pair
        return self.first <= other_pair.first and \
               self.second <= other_pair.second

    def __eq__(self, other_pair):...

if __name__ == '__main__':
    p1 = MyPair(5, 10)
    p2 = MyPair(5, 13)

    print(p1 <= p2)  # True
    print(p1 != p2)  # True
```

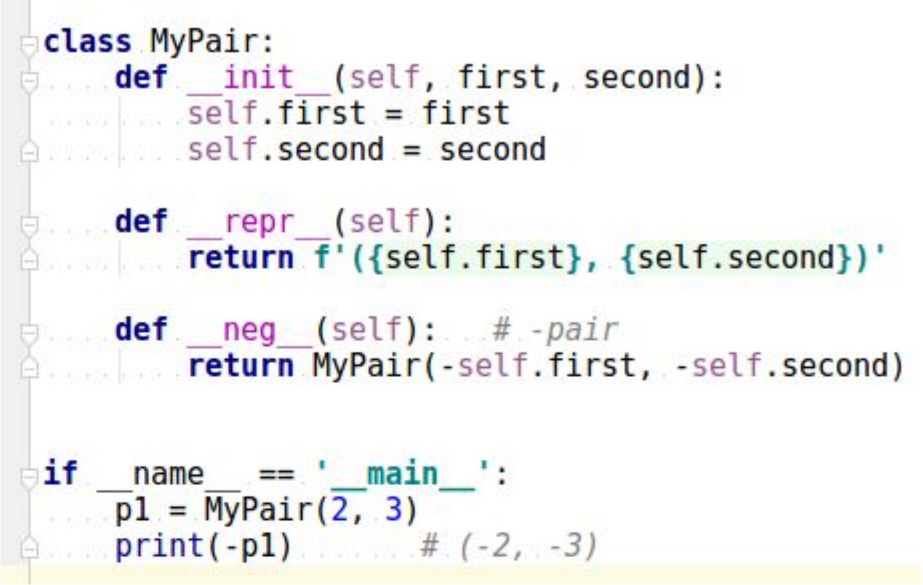
Sorting list of objects!



```python
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def __repr__(self):
        return f'({self.name}, {self.salary})'

    def __lt__(self, other_pair):  # on name first, if tie on salary
        if self.name != other_pair.name:
            return self.name < other_pair.name

        return self.salary < other_pair.salary


lst = [Employee('mostafa', 10),
       Employee('Ziad', 100), Employee('mostafa', 7)]
lst.sort()
print(lst)  # [(Ziad, 100), (mostafa, 7), (mostafa, 10)]
```

Unary Operators

- E.g. -p
- Other cases:
- +p ⇒ __pos__
- abs(p) ⇒ __abs__
- ~p ⇒ __invert__
  - bitwise inverse of p
  - *We did not study bitwise operators*



```python
class MyPair:
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def __repr__(self):
        return f'({self.first}, {self.second})'

    def __neg__(self):   # -pair
        return MyPair(-self.first, -self.second)


if __name__ == '__main__':
    p1 = MyPair(2, 3)
    print(-p1)       # (-2, -3)
```

“Acquire knowledge and impart it to the people.”

“Seek knowledge from the Cradle to the Grave.”