Advanced Python Programming

adrian.copie@e-uvt.ro

Lecture Attendance

- Not mandatory but highly encouraged

Laboratories Attendance

- Minimum 5 attendances for participating on Advanced Python Programming exam in winter

Final grade

(65% written exam, 35% homework)

IDEs

- JetBrains PyCharm (https://www.jetbrains.com/pycharm/)
- Microsoft Visual Studio Code (https://code.visualstudio.com/)
- Others

Big chapters

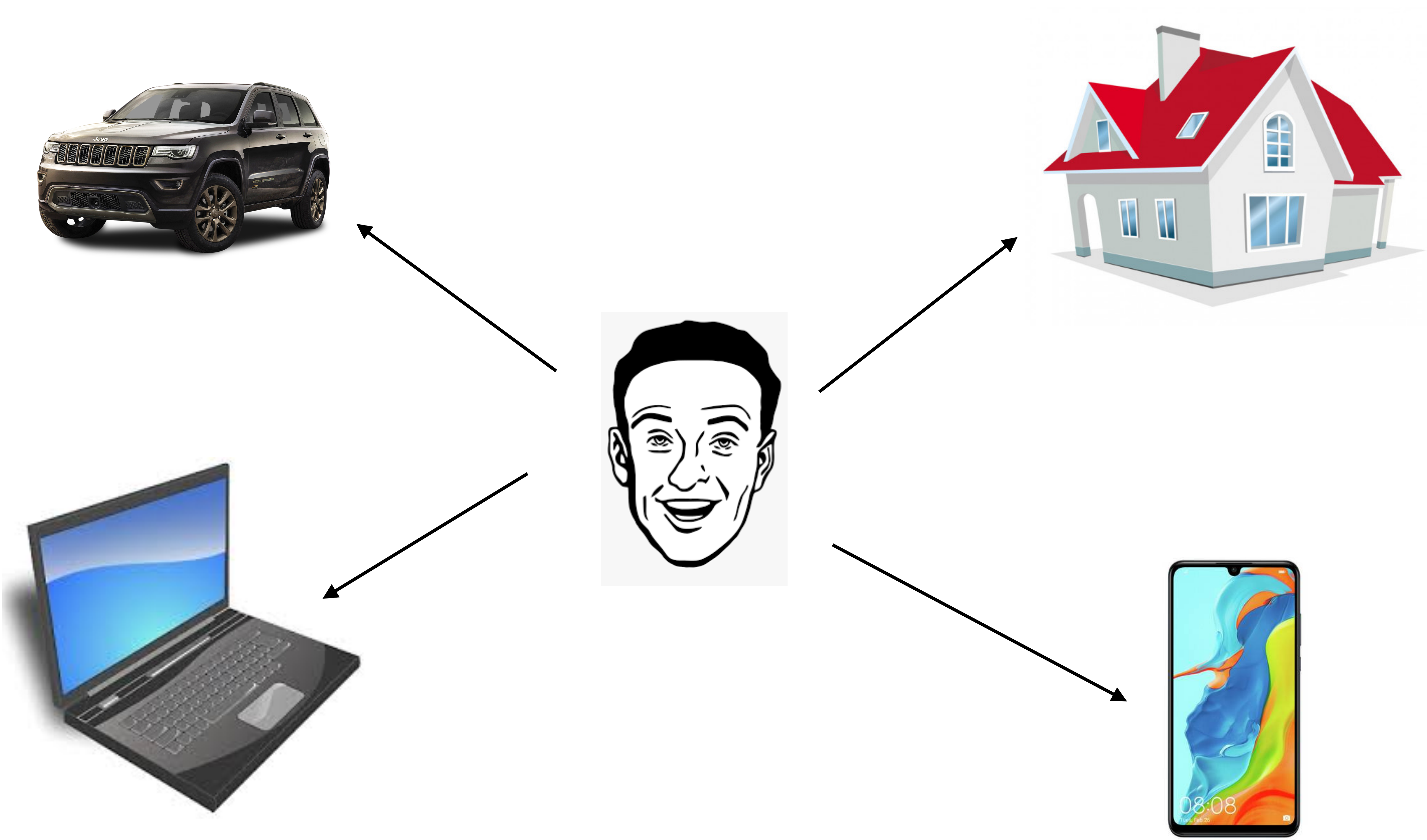
- Object Oriented Programming
- Concurrent Programming
- Functional Programming
- Asynchronous Programming
- Network Programming
- Parallel Programming
- Design Patterns implemented in Python
- Elements of computer graphics using Python’s libraries

Object Oriented Programming with Python

OOP

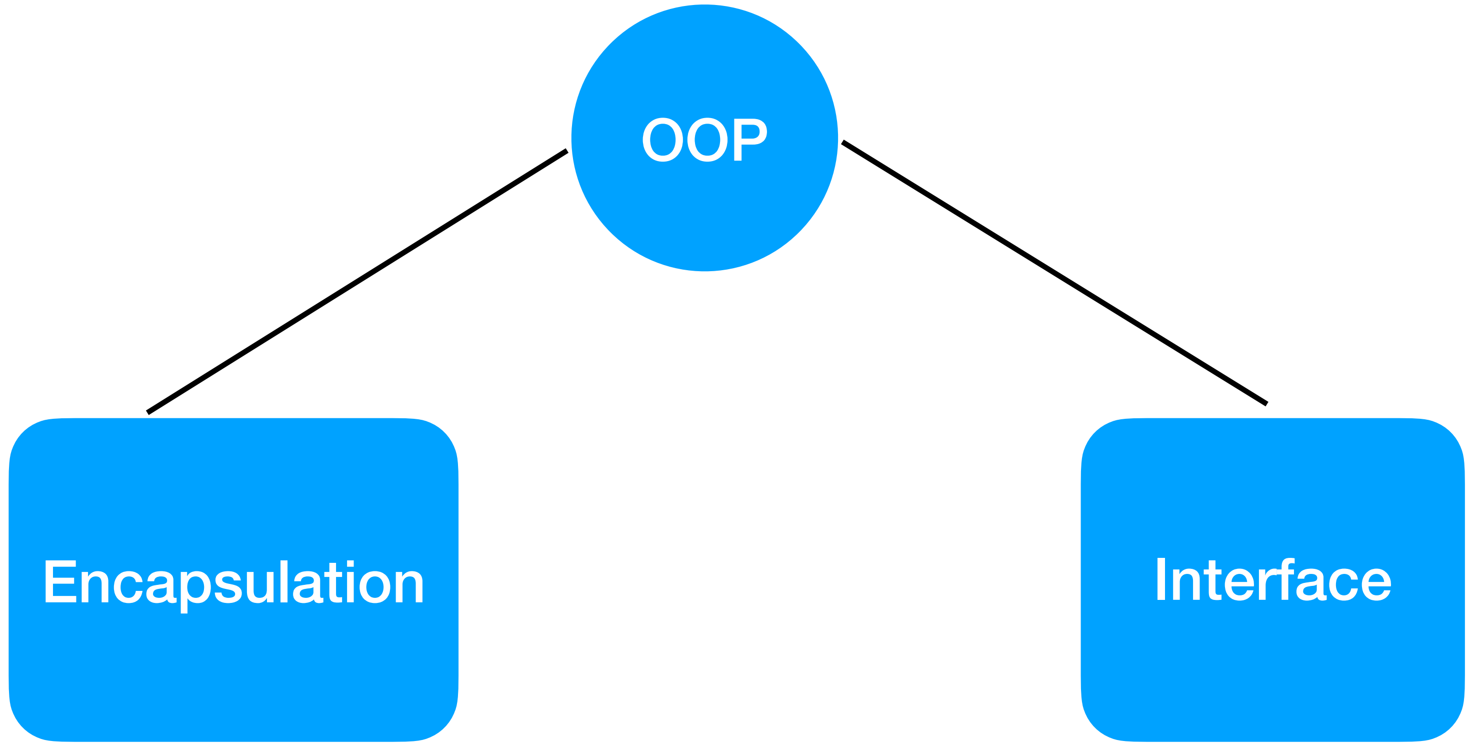
- Classes
- Constructors
- Methods
- Instance variables vs. Class variables
- Class methods and Static methods
- Initializers vs. Constructors
- Inheritance
- Multiple Inheritance
- Metaclasses

Everything around us is an object



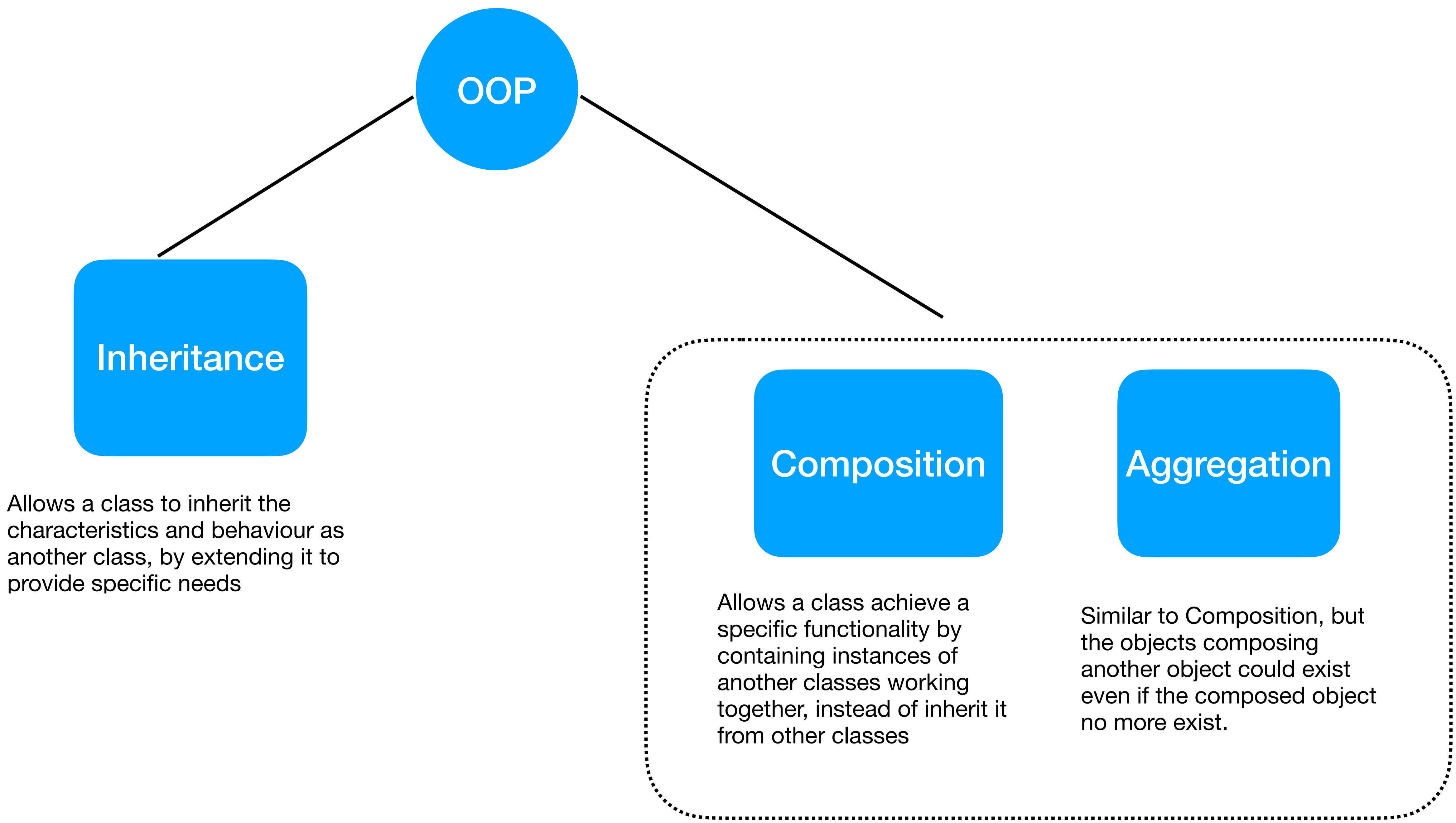
Why use OOP?

- Modularity. Better management. Better debugging (https://www.visualcapitalist.com/millions-lines-of-code)
- Code reusing by inheritance
- Increased productivity
- Polymorphism. Better flexibility
- Great for modelling and solving real life problems
- Security by data hiding and abstraction

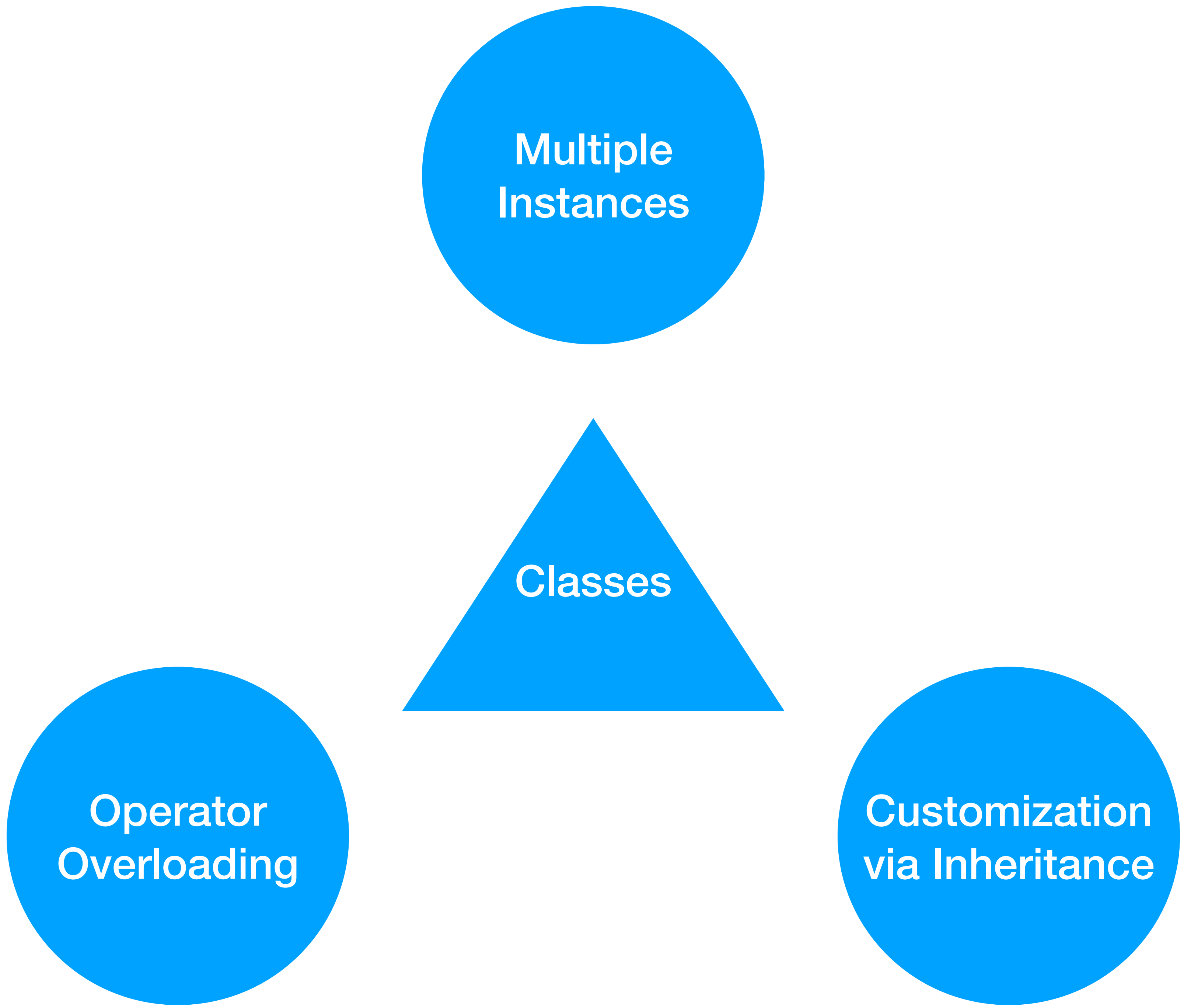


Allows a class to hide the attributes which are not necessary to be seen by other classes.

Allows a class to provide a facade to other classes, providing only the meaningful methods and attributes.


OOP
Inheritance
Allows a class to inherit the characteristics and behaviour as another class, by extending it to provide specific needs
Composition
Allows a class achieve a specific functionality by containing instances of another classes working together, instead of inherit it from other classes
Aggregation
Similar to Composition, but the objects composing another object could exist even if the composed object no more exist.

Classes are factories for
generating multiple objects
with distinct namespaces.



Classes can define objects
that respond differently to
some sort of operations

We can extend a class by
redefining its attributes
outside the class itself.
Classes can build up
namespaces hierarchies.

Design a strategy game

Game characters

OOP in Python

- Native support for OOP
- Everything is an object

```
x = 1
y = 2
z = x + y
```

Python does not support native types like C/C++

Classes

- Blueprint for creating objects
- Creating a class in Python is pretty straightforward, it is based on the reserved word `class` which will instruct the interpreter to create the class object in the memory and use it as a pattern for creating new instances of the class.

Design a strategy game

```python
class Soldier:
    pass
```

Class definition: used the `class` reserved word followed by the name of the class

To create a few instances of the `Soldier` class, we will write:

```python
soldier_one = Soldier()
soldier_two = Soldier()
```

`soldier_one` and `soldier_two` are now instances of the `Soldier` class.

To check that this is the case we can write:

```
print(soldier_one)
print(soldier_two)
```

and the output is:

```
<__main__.Soldier object at 0x10b3d7a10>
<__main__.Soldier object at 0x10b3d7ad0>
```

Namespaces

```
class Soldier:
    pass

soldier_1 = Soldier()

print(soldier_1.__dict__)

print(Soldier.__dict__)
```

instance namespace

```
{}
```

class namespace

```
{'__module__': '__main__', '__dict__': <attribute
'__dict__' of 'Soldier' objects>, '__weakref__':
<attribute '__weakref__' of 'Soldier' objects>,
'__doc__': None}
```

Namespaces

```python
class Officer:
    pass

officer_1 = Officer()
officer_1.rank = "colonel"

print(officer_1.__dict__)
print(Officer.__dict__)
```

instance namespace

```
{'rank': 'colonel'}
```

class namespace

```
{'__module__': '__main__', '__dict__': <attribute
'__dict__' of 'Officer' objects>, '__weakref__':
<attribute '__weakref__' of 'Officer' objects>,
'__doc__': None}
```

See the namespace of these instances:

```
print(soldier_one.__dict__)
print(soldier_two.__dict__)
```

and the output is:

```
{}
{}
```

All the `Soldier` instances are empty, no fields defined

Let's **add** a `name` for every instance of the `Soldier` object:

```
soldier_one.name = "John"
soldier_two.name = “Richard"
```

then display the value of the name field for every instance:

```
print(soldier_one.name)
print(soldier_two.name)
```

the result being :

```
John
Richard
```

Now, if we display the namespace of the objects

```
print(soldier_one.__dict__)
print(soldier_two.__dict__)
```

we see this:

```
{'name': 'John'}
{'name': ‘Richard'}
```

Add **new** information about our soldiers, the `weapon` they use to fight:

```
soldier_one.weapon = “sword"
soldier_two.weapon = "spear"
```

and display the values of the fields:

```
print(soldier_one.weapon)
print(soldier_two.weapon)
```

the result is:

```
sword
spear
```

To see the namespace of our two instances we'll proceed to write:

```
print(soldier_one.__dict__)
print(soldier_two.__dict__)
```

and this time the fields of the objects are:

```
{'name': 'John', 'weapon': 'sword'}
{'name': 'Richard', 'weapon': ‘spear'}
```

Constructors

```
class Soldier:
    def __init__(self, name, weapon):
        self.name = name
        self.weapon = weapon
```

`self` – the instance itself

instance variable initialization

Python vs Java constructors

Python

```
class Soldier:
    def __init__(self, name, weapon):
        self.name = name
        self.weapon = weapon
```

Java

```
public class Soldier {

    private String name;
    private String weapon:

    public Soldier(name, weapon){
        this.name = name;
        this.name = weapon;
    }
}
```

```
soldier_one = Soldier("John", "sword")
soldier_two = Soldier("Richard", “spear”)
```

There is no need to provide the instance itself in the parentheses, Python will do it automatically, so we will have only two arguments for the `Soldier` instance creation.

To display the values in the `Soldier` instances, we can run:

```
print(f'{soldier_one.name}, {soldier_one.weapon}')
print(f'{soldier_two.name}, {soldier_two.weapon}')
```

the output is:

```
John, sword
Richard, spear
```

Methods

```
class Soldier:
    def __init__(self, name, weapon):
        self.name = name
        self.weapon = weapon

soldier_one = Soldier("John", "sword")
soldier_two = Soldier("Richard", "spear")

print(f'I am {soldier_one.name} and I fight with a {soldier_one.weapon}')
print(f'I am {soldier_two.name} and I fight with a {soldier_two.weapon}’)
```

And the output will be:

```
I am John and I fight with a sword
I am Richard and I fight with a spear
```

```python
class Soldier:
    def __init__(self, name, weapon):
        self.name = name
        self.weapon = weapon

    def who_is(self):
        print(f'I am {self.name} and I fight with a {self.weapon}’)

soldier_one = Soldier("John", "sword")
soldier_two = Soldier("Richard", “spear")
```

We can then call that new function like:

```python
soldier_one.who_is()
soldier_two.who_is()
```

And the output will be again:

```
I am John and I fight with a sword
I am Richard and I fight with a spear
```

We have invoked the `who_is()` method on the `soldier_one` and `soldier_two` instances like:

```
soldier_one.who_is()
soldier_two.who_is()
```

But we can also invoke them through the class itself, in this case we need to manually pass th instance to the w`ho_is()` method:

```
Soldier.who_is(soldier_one)
Soldier.who_is(soldier_two)
```

the output being the same:

```
I am John and I fight with a sword
I am Richard and I fight with a spear
```

Instance variables vs. Class variables

We need a new property for our `Soldier` class called `defense`

```
class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense
```

`name`, `weapon` and `defense` are three variables that belong to the instance itself

What if we have some variables which are common for every instance of `Soldier` class?

Let's instantiate the `Soldier` class and then display the values for the defense attribute:

```
soldier_one = Soldier("John", "sword", 30)
soldier_two = Soldier("Richard", "spear", 40)

print(soldier_one.defense)
print(soldier_two.defense)
```

the output of our code is:

```
30
40
```

Now, we add a new method `increase_defense()` like it follows:

```
class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def increase_defense(self):
        return self.defense * 1.10
```

Basically, we increase the `defense` attribute's value by 10 percent

If we do the initializations and display the attributes, we have:

```
soldier_one = Soldier("John", "sword", 30)
soldier_two = Soldier("Richard", "spear", 40)

print(soldier_one.defense)
print(soldier_two.defense)


print(soldier_one.increase_defense())
print(soldier_two.increase_defense())
```

With the output:

```
30
40
33.0
44.0
```

Notice that the result is floating point after multiplication

this approach is not so convenient, since the increase percent is hardcoded in the method and also we want to have it shared between classes not instances

```python
class Soldier:

    increase_defense_ratio = 1.10

    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def increase_defense(self):
        return self.defense * increase_defense_ratio
```

Error

(Python does not know about the `increase_defense_ratio` variable)

```python
class Soldier:

    increase_defense_ratio = 1.10

    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def increase_defense(self):
        return self.defense * Soldier.increase_defense_ratio
```

So in the main program we can display it like:

```python
print(Soldier.increase_defense_ratio)
```

with the result:

```
1.10
```

But we also could write:

```
print(soldier_one.increase_defense_ratio)
```

with the same result:

```
1.10
```

It is time to ask how this is possible? We can notice that `increase_defense_ratio` is not an instance variable for `Soldier`, but it still displays the correct result.

Answer: **Python Method Resolution Order** (MRO)

```python
class Soldier:

    increase_defense_ratio = 1.10

    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def increase_defense(self):
        return self.defense * Soldier.increase_defense_ratio

soldier_one = Soldier("John", "sword", 30)
soldier_two = Soldier("Richard", "spear", 40)

print(f"soldier_one.defense = {soldier_one.defense}")
print(f"soldier_two.defense = {soldier_two.defense}")

Soldier.increase_defense_ratio = 1.5

print(f"soldier_one.defense = {soldier_one.defense}")
print(f"soldier_two.defense = {soldier_two.defense}")

print(f"soldier_one.increase_defense() = {soldier_one.increase_defense()}")
print(f"soldier_two.increase_defense() = {soldier_two.increase_defense()}")

print(f"soldier_one.increase_defense_ratio = {soldier_one.increase_defense_ratio}")
print(f"soldier_two.increase_defense_ratio = {soldier_two.increase_defense_ratio}")
```

```
soldier_one.defense = 30
soldier_two.defense = 40

soldier_one.defense = 30
soldier_two.defense = 40

soldier_one.increase_defense() = 45.0
soldier_two.increase_defense() = 60.0

soldier_one.increase_defense_ratio = 1.5
soldier_two.increase_defense_ratio = 1.5
```

let’s print the namespaces for both Soldier class and its instance:

```
print(soldier_one.__dict__)
```

The result of the call is below:

```
{'name': 'John', 'weapon': 'sword', 'defense': 30}
```

```
print(Soldier.__dict__)
```

The result of the call is below:

```
{'__module__': '__main__', 'increase_defense_ratio': 1.5,
'__init__': <function Soldier.__init__ at 0x1079bb9e0>,
'increase_defense': <function Soldier.increase_defense at
0x1079bbf80>, '__dict__': <attribute '__dict__' of 'Soldier'
objects>, '__weakref__': <attribute '__weakref__' of 'Soldier'
objects>, '__doc__': None}
```

Another application of class variables is to keep track of the number of created instances from a class

```
class Soldier:

    total_number_of_soldiers = 0

    def __init__(self, name, weapon):
        self.name = name
        self.weapon = weapon
        Soldier.total_number_of_soldiers += 1

soldier_one = Soldier("John", "sword")
soldier_two = Soldier("Richard", "spear")

print(Soldier.total_number_of_soldiers)
```

The result is below:

```
2
```

Class methods and static methods

```
class Soldier:
    increase_defense_ratio = 1.10

    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def increase_defense(self):
        return self.defense * Soldier.increase_defense_ratio

    @classmethod
    def set_defense_ratio(cls, new_ratio):
        cls.increase_defense_ratio = new_ratio

soldier_one = Soldier("John", "sword", 30)
soldier_two = Soldier("Richard", "spear", 40)

Soldier.set_defense_ratio(1.5)

print(soldier_one.defense)
print(soldier_two.defense)

print(soldier_one.increase_defense())
print(soldier_two.increase_defense())
```

The output of this code is:

```
30
40
45.0
60.0
```

Class methods are used to provide alternative constructors for the `__init__()` method

CSV format: →

```
John,sword,30
Richard,spear,40
```

Approach 1

```
class Soldier:
    increase_defense_ratio = 1.10

    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def increase_defense(self):
        return self.defense * Soldier.increase_defense_ratio

    @classmethod
    def set_defense_ratio(cls, new_ratio):
        cls.increase_defense_ratio = new_ratio

# prepare a collection to hold the deserialized soldiers
army = []

with open("game.sav") as f:
    for line in f:
        name, weapon, defense = line.split(",")
        army.append(Soldier(name, weapon, defense))
```

```python
class Soldier:
    increase_defense_ratio = 1.10

    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def increase_defense(self):
        return self.defense * Soldier.increase_defense_ratio

    @classmethod
    def set_defense_ratio(cls, new_ratio):
        cls.increase_defense_ratio = new_ratio

    @classmethod
    def from_csv(cls, line):
        name, weapon, defense = line.split(",")
        return cls(name, weapon, defense)

# prepare a collection to hold the deserialized soldiers
army = []

with open("game.sav") as f:
    for line in f:
        army.append(Soldier.from_csv(line))
```

Approach 2

```python
class Soldier:
    increase_defense_ratio = 1.10

    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def increase_defense(self):
        return self.defense * Soldier.increase_defense_ratio

    @classmethod
    def set_defense_ratio(cls, new_ratio):
        cls.increase_defense_ratio = new_ratio

    @classmethod
    def from_csv(cls, line):
        name, weapon, defense = line.split(",")
        return cls(name, weapon, defense)

    @staticmethod
    def make_noise():
        print("whoosh")


soldier_one = Soldier("John", "sword", 30)
soldier_one.make_noise()
```

Properties

```python
# first approach to define the Officer class

class Officer:
    def __init__(self, rank):
        self.rank = rank

# create an instance of an officer
officer_1 = Officer("major")
print(officer_1.rank)

# new officer with an inexistent rank
officer_2 = Officer("private")
print(officer_2.rank)

# upgrade officer's rank
officer_1.rank = "colonel"

print(officer_2.__dict__)
del officer_2.rank
print(officer_2.__dict__)
```

```
major
private
{'rank': 'private'}
{}
```

We have one attribute called rank, which is public and we can modify it or delete it through instance attributes

! This is not an officer rank

This is how we use the Officer instance from our code

Properties

```
#second approach to define the Officer class

ranks = ["lieutenant", "captain", "major", "colonel", "general"]

class Officer:
    def __init__(self, rank="lieutenant"):
        if rank not in ranks:
            raise ValueError("The given rank is not among the accepted ranks")
        # practically it is not hidden, but we have changed the attribute name to differ
        # from "rank"
        self._rank = rank

    def set_rank(self, new_rank):
        if new_rank not in ranks:
            raise ValueError("The given rank is not among the accepted ranks")
        self._rank = new_rank

    def get_rank(self):
        return self._rank

    def del_rank(self):
        del self._rank

officer = Officer("major")
print(officer.get_rank())

officer.set_rank("colonel")
print(officer.get_rank())

officer.set_rank("private")
print(officer.get_rank())
```

Properties

When an attempt to set a rank which was different that the preset collection, an exception was raised

```
major
colonel
Traceback (most recent call last):
  File "/Users/adriancopie/Projects/Personal/advanced/oop_3.py", line 30, in
<module>
    officer.set_rank("private")
  File "/Users/adriancopie/Projects/Personal/advanced/oop_3.py", line 15, in
set_rank
    raise ValueError("The given rank is not among the accepted ranks")
ValueError: The given rank is not among the accepted ranks
```

Properties

The problem with the two previous approaches is that we have different ways to deal with the `rank` attribute

In the first case, to set the rank property we used:

```
officer.rank = “major”
```

In the second case we used:

```
officer.set_rank(“major”)
```

which can cause lots of troubles if the `Officer` class is instantiated in many places

Properties

```
# third approach to define the Officer class

ranks = ["lieutenant", "captain", "major", "colonel", "general"]

class Officer:
    def __init__(self, rank="lieutenant"):
        if rank not in ranks:
            raise ValueError("The given rank is not among the accepted ranks")
        self._rank = rank

    def set_rank(self, new_rank):
        print("set_rank called...")
        if new_rank not in ranks:
            raise ValueError("The given rank is not among the accepted ranks")
        self._rank = new_rank

    def get_rank(self):
        print("get_rank called...")
        return self._rank

    def del_rank(self):
        print("del_rank called...")
        del self._rank

    rank = property(get_rank, set_rank, del_rank, "This is the Officer class")

officer_1 = Officer()
officer_1.rank = "major"
print(officer_1.rank)
```

```
set_rank called...
get_rank called...
major
```

One can see that the way in which we interact with the rank attribute is similar to the one in the first approach

Properties

```
# fourth approach to define the Officer class

ranks = ["lieutenant", "captain", "major", "colonel", "general"]

class Officer:
    def __init__(self, rank="lieutenant"):
        if rank not in ranks:
            raise ValueError("The given rank is not between the accepted ranks")
        self._rank = rank

    @property
    def rank(self):
        print("get_rank called...")
        return self._rank

    @rank.setter
    def rank(self, new_rank):
        print("set_rank called...")
        if new_rank not in ranks:
            raise ValueError("The given rank is not between the accepted ranks")
        self._rank = new_rank

    @rank.deleter
    def rank(self):
        print("del_rank called...")
        del self._rank

officer_1 = Officer()
officer_1.rank = "major"
print(officer_1.rank)
```

In this approach we have used the `@property` decorator instead of creating a property object

```
set_rank called...
get_rank called...
major
```

One can see that the way in which we interact with the rank attribute is similar to the one in the first approach

Initializer vs. Constructor

```
class Soldier:
    def __new__(cls):
        print("__new__() method called")
        obj = super().__new__(cls)
        print(obj)
        return obj

    def __init__(self):
        print("__init__() method called")
        print(self)

soldier = Soldier()
```

```
__new__() method called
<__main__.Soldier object at 0x104b4dbd0>
__init__() method called
<__main__.Soldier object at 0x104b4dbd0>
```

```python
class Soldier:
    def __new__(cls):
        print("__new__ method called")

    def __init__(self):
        print("__init__ method called")
        print(self)

soldier = Soldier()
```

call to super() is missing

```
__new__ method called
```

__init__ method is never called since no instance is created

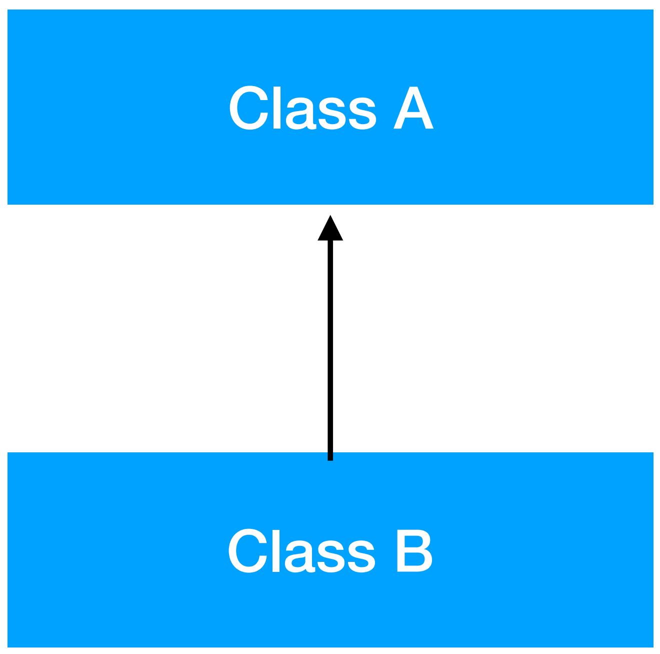
Limiting the number of instances that could be created

```
class Soldier:
    max_allowed_soldier_objects = 3
    curr_soldier_objects = 0

    def __new__(cls):
        if cls.curr_soldier_objects >= cls.max_allowed_soldier_objects:
            raise ValueError(f"Cannot create more than {cls.max_allowed_soldier_objects}")
        cls.curr_soldier_objects += 1
        return super().__new__(cls)

soldier_one = Soldier()
soldier_two = Soldier()
soldier_three = Soldier()
soldier_four = Soldier()
```

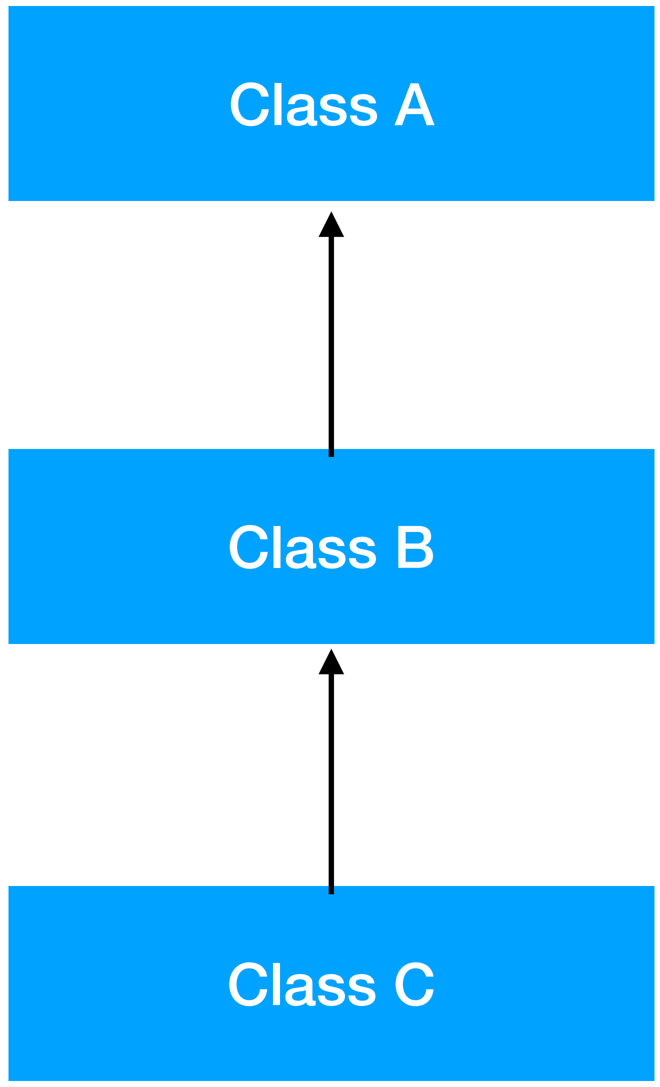
The result of our code is below:

```
Traceback (most recent call last):
  File "/advanced/soldier_3.py", line 15, in <module>
    soldier_four = Soldier()
  File "/advanced/soldier_3.py", line 7, in __new__
    raise ValueError(f"Cannot create more than {cls.max_allowed_soldier_objects}")
ValueError: Cannot create more than 3
```

Inheritance - Types

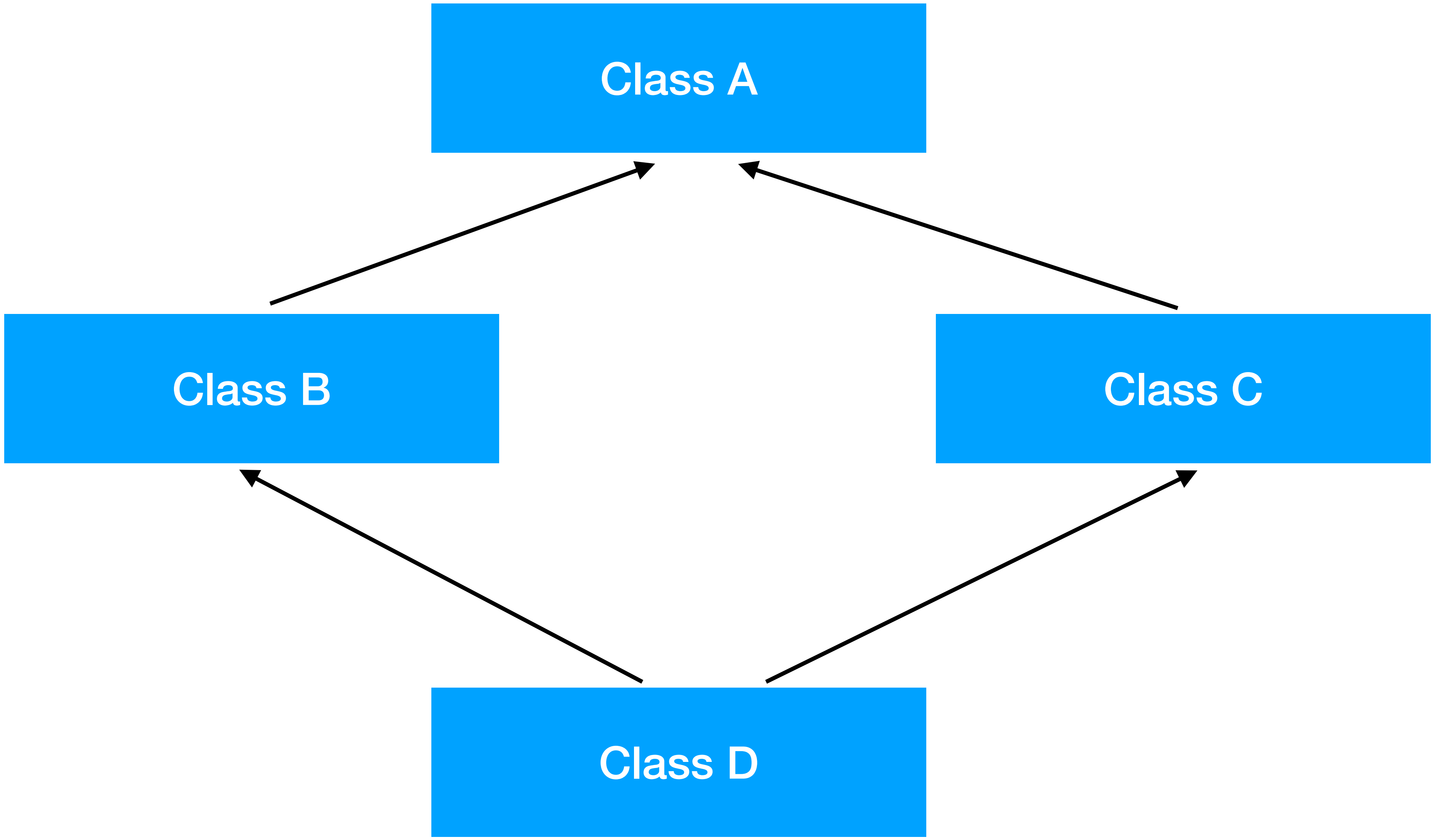


Single Inheritance



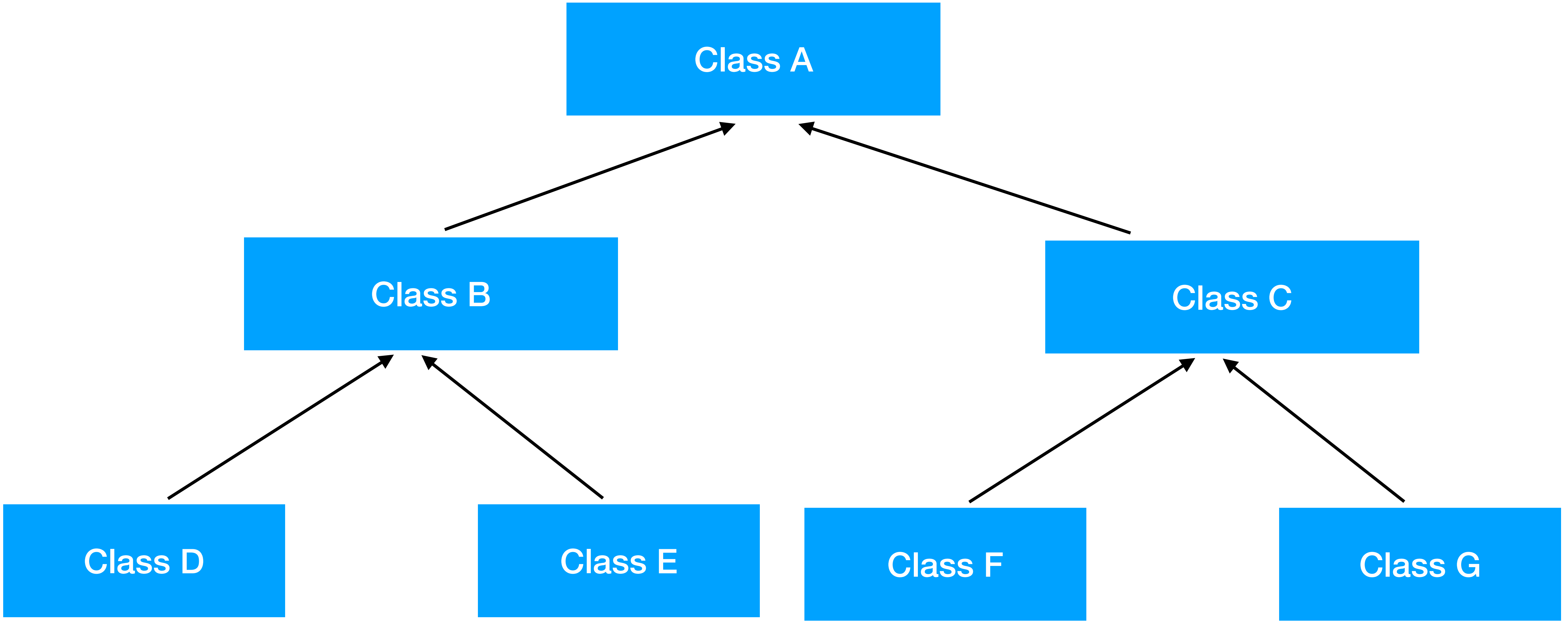
Multi-Level Inheritance

Inheritance - Types



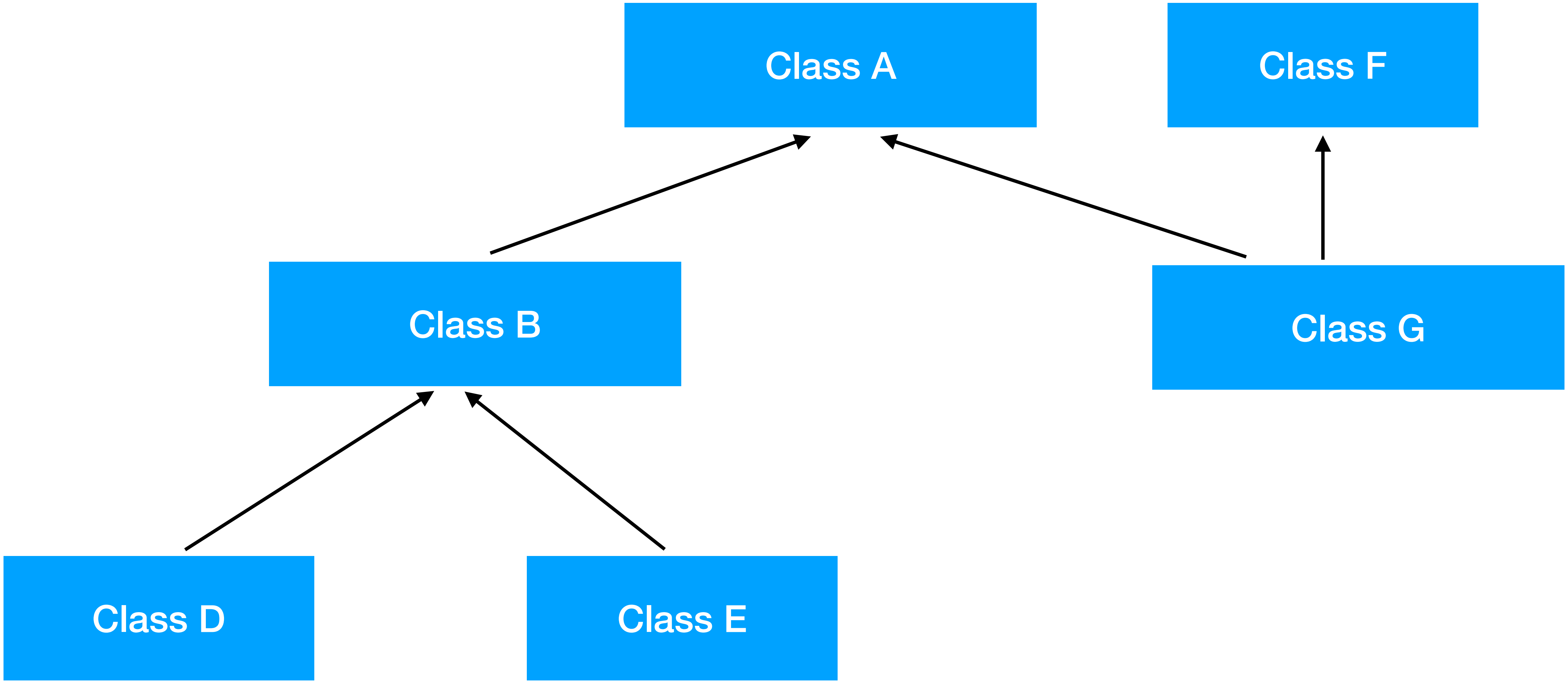
Multipath Inheritance

Inheritance - Types



Hierarchical Inheritance

Inheritance - Types



Hybrid Inheritance

Inheritance

```
class Soldier:
    pass
```

which is equivalent with:

```
class Soldier(object):
    pass
```

if a class needs to inherit something from another class, the **base** class will be written in parentheses

A swordsman has a `name`, wear a `weapon` and also is wearing an armour which protects him against the enemy weapons, this protection being quantified by a number, the `defense` attribute.

```
class Swordsman:
    increase_defense_ratio = 1.10

    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def increase_defense(self):
        return self.defense * Swordsman.increase_defense_ratio

    @classmethod
    def set_defense_ratio(cls, new_ratio):
        cls.increase_defense_ratio = new_ratio
```

We can simplify as:

```
class Swordsman(Soldier):
    pass
```

then we instantiate two `Swordsman` objects:

```
swordsman_one = Swordsman("John", "sword", 30)
swordsman_two = Swordsman("Richard", "sword", 30)

print(swordsman_one)
print(swordsman_two)

print(swordsman_one.name)
print(swordsman_two.name)
```

MRO again!

The Swordsman object does not have the name attribute but its parent, the Soldier class, has it.

the result being:

```
<__main__.Swordsman object at 0x10af559d0>
<__main__.Swordsman object at 0x10af55a10>
John
Richard
```

```
print(help(swordsman_one))
```

and it will show:

```
Help on Swordsman in module __main__ object:

class Swordsman(Soldier)
 |  Swordsman(name, weapon, defense)
 |
 |  Method resolution order:
 |      Swordsman
 |      Soldier
 |      builtins.object
 |
 |  Methods inherited from Soldier:
 |
 |  __init__(self, name, weapon, defense)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  increase_defense(self)
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from Soldier:
 |
 |  set_defense_ratio(new_ratio) from builtins.type
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Soldier:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from Soldier:
 |
 |  increase_defense_ratio = 1.1

None
```

Let’s consider that swordsmen could wear additional `gear`, like shields and knives, so we should add this additional attribute to the `Swordsman` class

```
class Swordman(Soldier):
    def __init__(self, name, weapon, defense, gear):
        self.name = name
        self.weapon = weapon
        self.defense = defense
        self.gear = gear
```

Duplicate code in `Swordsman`

A better approach:

```
class Swordsman(Soldier):
    def __init__(self, name, weapon, defense, gears=None):
        super().__init__(name, weapon, defense)
        if gears is None:
            self.gears = []
        else:
            self.gears = gear
```

Also notice this construction when it comes to use lists as parameters for functions, and they have a default value

A possible initialization of the `Swordsman` class could be:

```
swordsman_one = Swordsman("John", "sword", 30, ["knife", "shield"])
swordsman_two = Swordsman("Richard", "sword", 30, [“shield”])

print(swordsman_one.gear)
print(swordsman_two.gear)
```

The displayed result is:

```
['knife', 'shield']
shield
```

In order to make this class more useful, we add a method called `add_gear` which will add additional gear to this swordsman and a method called `remove_gear` which will do the opposite, removing gear from the swordsman

```
class Swordsman(Soldier):
    def __init__(self, name, weapon, defense, gears=None):
        super().__init__(name, weapon, defense)
        if gears is None:
            self.gears = []
        else:
            self.gears = gears

    def add_gear(self, gear):
        if gear not in self.gears:
            self.gears.append(gear)

    def remove_gear(self, gear):
        if gear in self.gears:
            self.gears.remove(gear)
```

```
swordsman_one.add_gear("stick")
print(swordsman_one.gears)
```

with the output:

```
['knife', 'shield', ‘stick']
```

and then

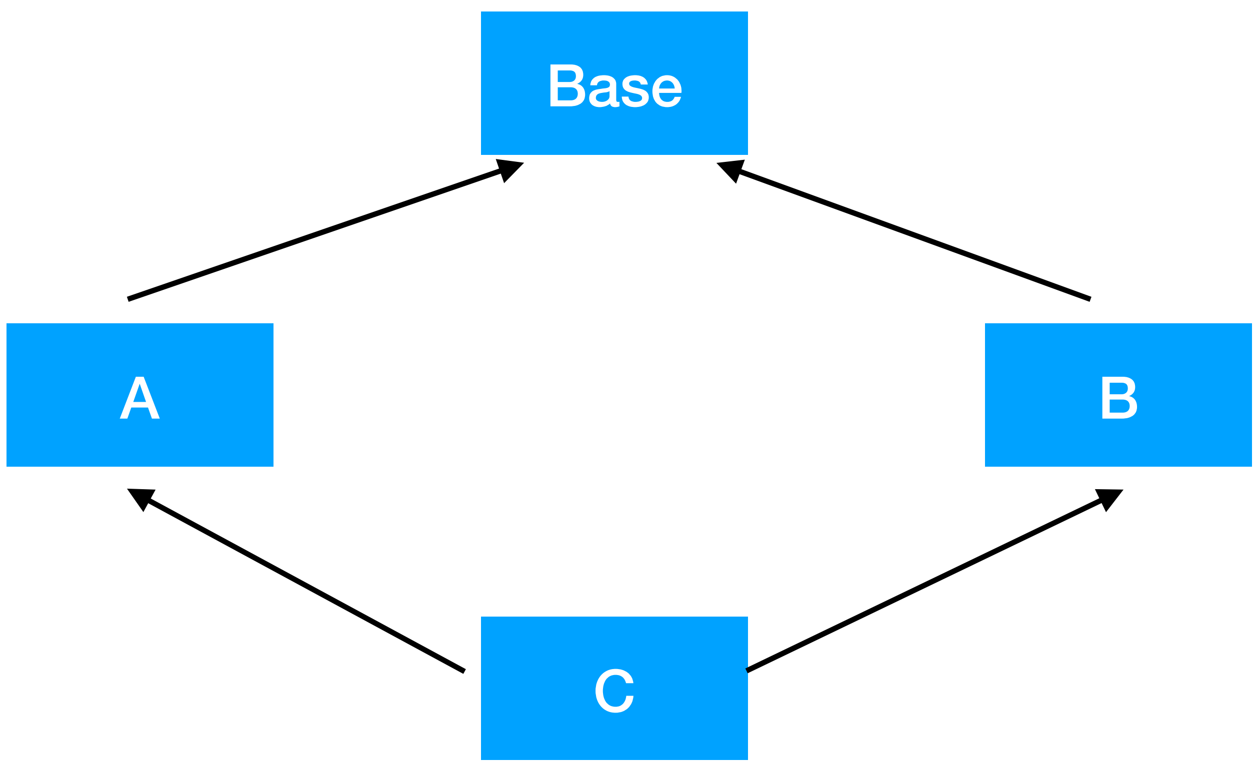
```
swordsman_one.remove_gear("knife")
print(swordsman_one.gears)
```

with the output:

```
['shield', ‘stick’]
```

Multiple Inheritance

(Discussion)



First approach

```
class Base:
    def __init__(self):
        print("Base.__init__")

class A(Base):
    def __init__(self):
        print("A.__init__")
        Base.__init__(self)

class B(Base):
    def __init__(self):
        print("B.__init__")
        Base.__init__(self)

class C(A, B):
    def __init__(self):
        print("C.__init__")
        A.__init__(self)
        B.__init__(self)


c = C()
```

Output:

```
C.__init__
A.__init__
Base.__init__
B.__init__
Base.__init__
```

Base constructor is called twice

(MRO is not used)

Second approach

```
class Base:
    def __init__(self):
        print("Base.__init__")

class A(Base):
    def __init__(self):
        print("A.__init__")
        super().__init__()

class B(Base):
    def __init__(self):
        print("B.__init__")
        super().__init__()

class C(A, B):
    def __init__(self):
        print("C.__init__")
        A.__init__(self)
        B.__init__(self)

c = C()
```

Output:

```
C.__init__
A.__init__
B.__init__
Base.__init__
B.__init__
Base.__init__
```

Base constructor is called twice

(MRO is partially used)

Third approach

```
class Base:
    def __init__(self):
        print("Base.__init__")

class A(Base):
    def __init__(self):
        print("A.__init__")
        super().__init__()

class B(Base):
    def __init__(self):
        print("B.__init__")
        super().__init__()

class C(A, B):
    def __init__(self):
        print("C.__init__")
        super().__init__()


c = C()
```

Output:

```
C.__init__
A.__init__
B.__init__
Base.__init__
```

Base constructor is called
Only once

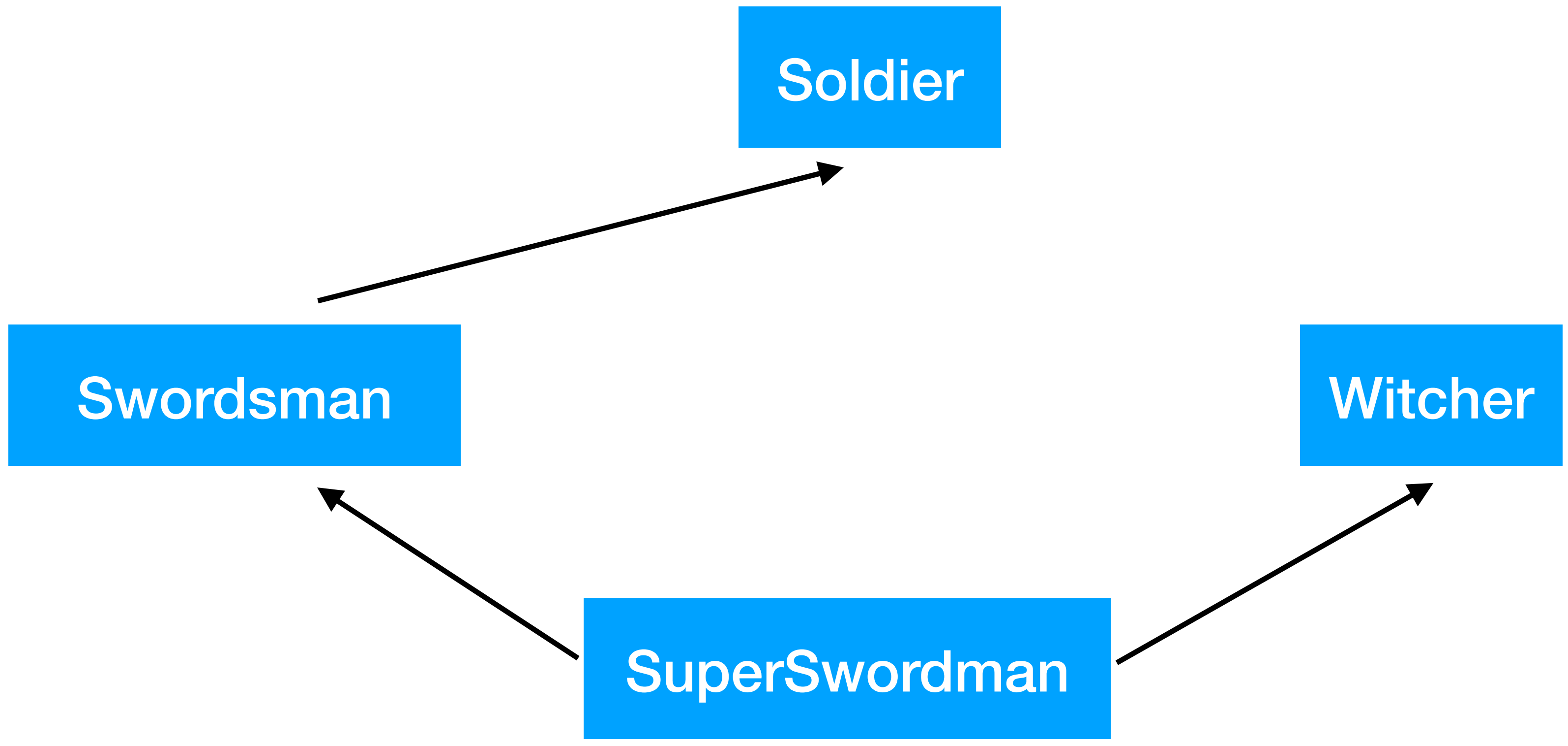
(MRO is used)

Multiple Inheritance

The plot:

Our game tends to develop and at some point, let’s assume that our Swordsman finds an artefact which allows him to gain magical powers, and will act like a Witcher, he will be able to cast various spells.

Multiple Inheritance



```python
class Witcher:
    def __init__(self, spells=None):
        if spells is None:
            self.spells = []
        else:
            self.spells = spells

    def cast_spell(self, spell):
        if spell in self.spells:
            print(f'spell {spell} was casted!')

witcher = Witcher(["bless", "destruction", "fireball"])
witcher.cast_spell("fireball")
```

```
spell fireball was casted!
```

```python
class SuperSwordsman(Swordsman, Witcher):
    def __init__(self, name, weapon, defense, gears, spells):
        Swordsman.__init__(self, name, weapon, defense, gears)
        Witcher.__init__(self, spells)


super_swordsman = SuperSwordsman("John", "sword", 30, ["knife",
"shield"], ["fireball"])

super_swordsman.cast_spell("fireball")
```

the result of the call being:

```
spell fireball was casted!
```

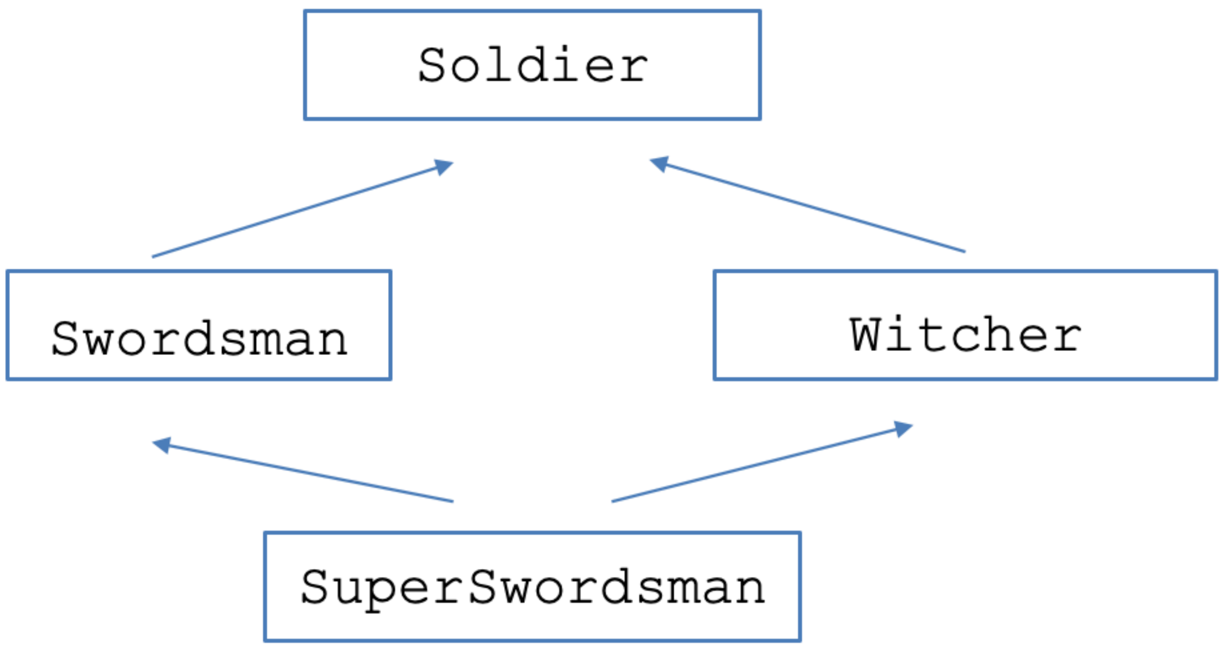
Notice that we didn't use `super()` for calling the constructor of the superclass for Witcher because it is not related to Soldier

MRO not used
For constructors

MRO

The diamond problem:

for a class derived from other classes that could be a level of ambiguity in determining which method or attribute was invoked.



We need to modify a little out `Witcher` class, to support a `name` attribute. Also, we derive it from Soldier too, in order to exemplify the diamond problem.

```
class Witcher(Soldier):
    def __init__(self, name, spells=None):
        self.name = name
        if spells is None:
            self.spells = []
        else:
            self.spells = spells

    def cast_spell(self, spell):
        if spell in self.spells:
            print(f'spell {spell} was casted!')
```

Because of this change, the `SuperSwordsman` class has to change too:

```
class SuperSwordsman(Swordsman, Witcher):
    def __init__(self, name, weapon, defense, gears, wname, spells):
        Swordsman.__init__(self, name, weapon, defense, gears)
        Witcher.__init__(self, wname, spells)
```

And now we instantiate a `SuperSwordsman` object:

```
super_swordsman = SuperSwordsman("John", "sword", 30,
["knife", "shield"], "Merlin", ["fireball"])
print(super_swordsman.name)
```

`name` is present in the base class `Soldier` but also in the class `Witcher`, we have a legitimate question: which one will be displayed when we invoke the `super_swordsman.name`?

From the figure with the diamond problem, we can see the following inheritance chains:

```
SuperSwordsman -> Swordsman -> Soldier
SuperSwordsman -> Witcher ->Soldier
```

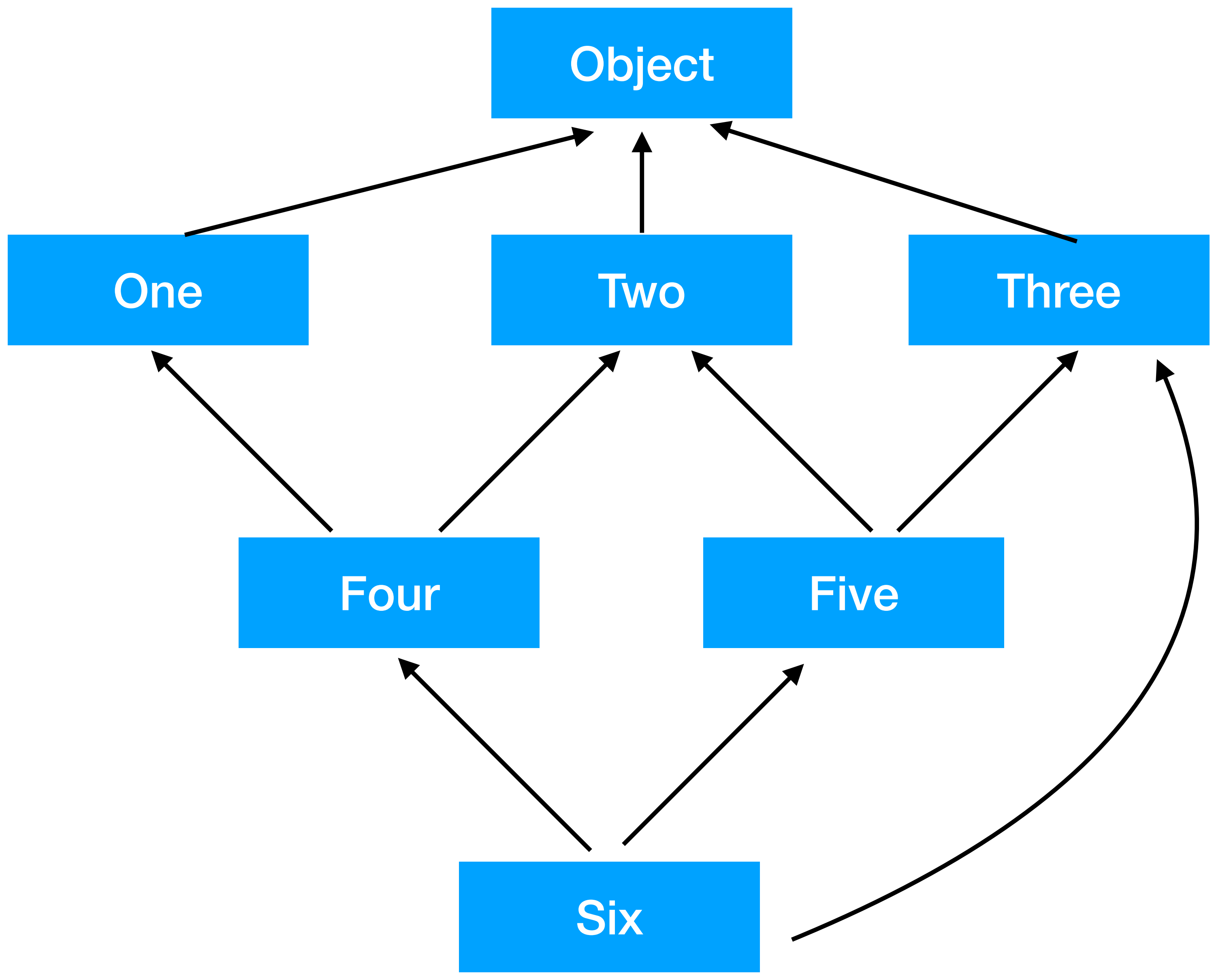
`Soldier` class is in the top of the chain, and should be the last place in which we are looking for an attribute or a method

`Swordsman` and `Witcher` are on the same level

`SuperSwordsman` class is at the beginning of the chain

Method Resolution Order

(more complex example)



Method Resolution Order

(more complex example)

```
class One:
    pass

class Two:
    pass

class Three:
    pass

class Four(One, Two):
    pass

class Five(Two, Three):
   pass

class Six(Five, Four, Three):
    pass


print(Six.__mro__)
```

```
(<class '__main__.Six'>,
<class '__main__.Five'>,
<class ‘__main__.Four'>,
<class '__main__.One'>,
<class '__main__.Two'>,
<class '__main__.Three'>,
<class 'object'>)
```

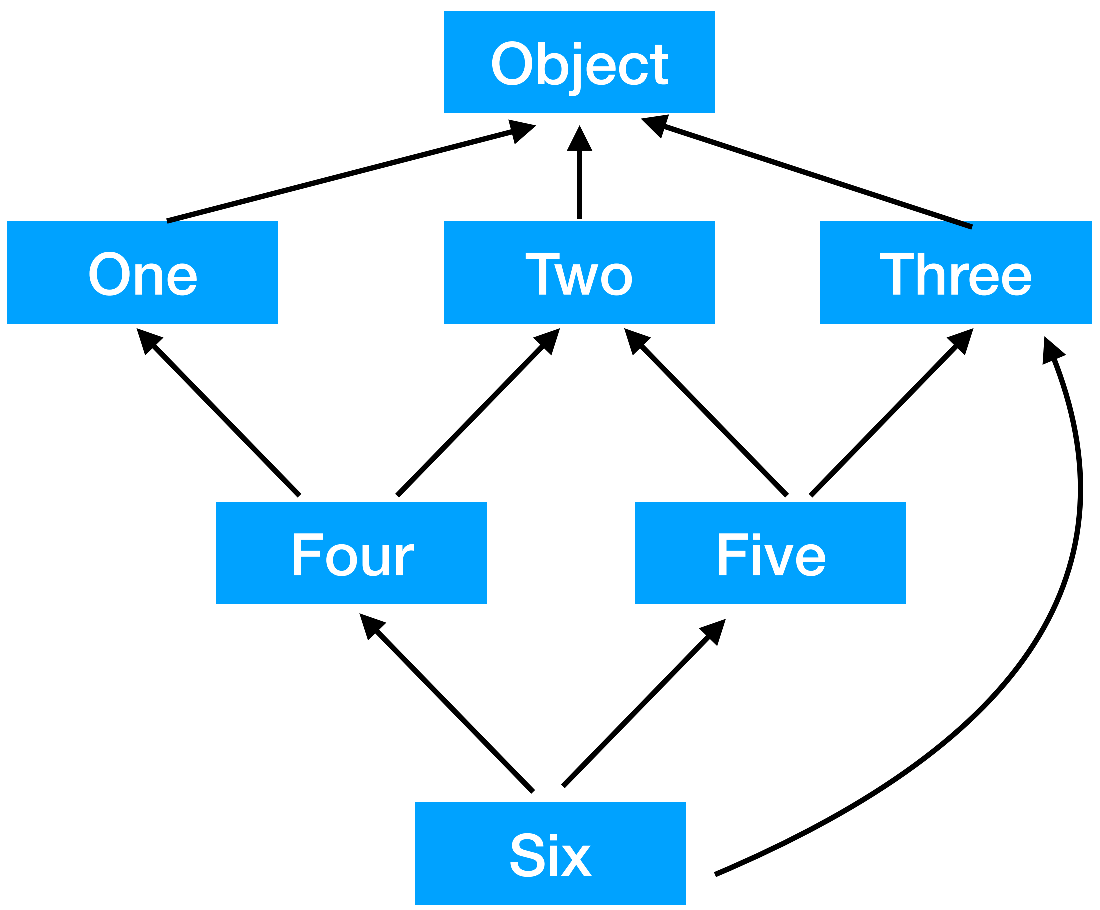
C3 Linearization Algorithm for MRO

Linearization of a class is defined as:

- The class itself
- Followed by the MROs of its parent classes, combined in a way that respects the inheritance hierarchy

Consistency rules:

- If a class has multiple parents, MRO is computed in a way that ensures:
  - **Local precedence order**: A class is checked before its parents.
  - **Monotonicity**: If one class is before another in one place, it will remain so throughout the entire MRO.
  - **No circular dependencies.**



MRO for individual classes

```
One:      [One, object]
Two:      [Two, object]
Three:    [Three, object]
Four:     [Four, One, Two, object]
Five:     [Five, Two, Three, object]
Six: computed by merging the MROs of Five, Four, and Three
```

Phases of merging MROs for all the classes

Step 1: Start with Six

- The MRO of Six begins with the class itself:

```
MRO = [Six]
```

Step 2: List the MROs of the Parent Classes

The MRO of Six must be a merge of the MROs of its parents: Five, Four, and Three.

```
• MRO of Five:   [Five, Two, Three, object]
• MRO of Four:   [Four, One, Two, object]
• MRO of Three:  [Three, object]
```

We need to merge these lists **while respecting the inheritance hierarchy**.

Step 3: C3 Linearization Algorithm (Merging the MROs)

The C3 algorithm merges MROs by repeatedly picking the first element of each list that:

- Is not present later in any other list (i.e., the class isn't a second or later choice in any other list).

Round 1:

Current lists:

- `Five:  [Five, Two, Three, object]`
- `Four:  [Four, One, Two, object]`
- `Three: [Three, object]`

Look at the first element of each list:

- The first candidates are Five, Four, and Three.

Choose Five because:

- Five is the first in its list and doesn't appear anywhere else as a second choice or beyond.

Add Five to the MRO:

```
MRO = [Six, Five]
```

Round 2:

Updated lists:

- `Five (after removing Five): [Two, Three, object]`
- `Four:                        [Four, One, Two, object]`
- `Three:                       [Three, object]`

`The first candidates are now` Two, Four, and Three.

Choose Four because:

- Four is the first in its list and isn't a second choice in any other list.

Add Four to the MRO:

`MRO = [Six, Five, Four]`

Round 3:

Updated lists:

- `Five:` `[Two, Three, object]`
- `Four: (after removing Four):` `[One, Two, object]`
- `Three:` `[Three, object]`

The first candidates are `Two, One, and Three.`

Choose One because:

- One is the first in its list and doesn't appear anywhere else as a second choice.

Add One to the MRO:

```
MRO = [Six, Five, Four, One]
```

Round 4:

Updated lists:

- `Five:                        [Two, Three, object]`
- `Four (after removing One):   [Two, object]`
- `Three:                       [Three, object]`

The first candidates are `Two, Two, and Three.`

Choose Two because:

- Two is the first in multiple lists, but that's okay as long as it's not a second choice in another list.

Add Two to the MRO:

```
MRO = [Six, Five, Four, One, Two]
```

Round 5:

Updated lists:

- `Five (after removing Two):` `[Three, object]`
- `Four (after removing Two):` `[object]`
- `Three:` `[Three, object]`

The first candidates are `Three and Three.`

Choose Three because:

- Three is the first valid candidate, and it appears in both Five's and Three's MRO.

Add Three to the MRO:

```
MRO = [Six, Five, Four, One, Two, Three]
```

Round 6:

Updated lists:

- `Five after removing Three:      [object]`
- `Four after removing Three:      [object]`
- `Three after removing Three:     [object]`

The first and only candidate left is `object`.

Choose object because:

- object is the final base class that Python uses for all classes.

Add object to the MRO:

```
MRO = [Six, Five, Four, One, Two, Three, object]
```

Final MRO for class Six

Abstract classes

Abstract Base Class

```
from abc import ABC

class Soldier(ABC):
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    @abstractmethod
    def what_am_i(self):
        raise NotImplementedError
```

Now, let’s try to instantiate an object from the `Soldier` class:

```
soldier = Soldier("John", "sword", 30)
```

The result of the code execution is:

```
Traceback (most recent call last):
File "/Users/adriancopie/Projects/Personal/advanced/
soldier_13.py",

line 23, in <module>
    soldier = Soldier("John", "sword", 30)
TypeError: Can't instantiate abstract class Soldier with abstract
methods what_am_i
```

This is not possible to instantiate an abstract class

Subclass the `Soldier` class:

```
class Swordsman(Soldier):
    def __init__(self, name, weapon, defense, gear):
        super().__init__(name, weapon, defense)
        self.gear = gear

    def what_am_i(self):
        print("I am a swordsman”)


swordsman = Swordsman(“John”, “sword”, 20, “knife”)
swordsman.who_am_i()
```

The result of method invocation will be:

```
I am a swordsman
```

Abstract classes could have implementations for the abstract methods

```
class Soldier(ABC):
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    @abstractmethod
    def what_am_i(self):
        print("I am a basic soldier!”)
```

Again, we try to create an instance of the class, this time with the implemented abstract method:

```
soldier = Soldier("John", "sword", 30)
```

The result of the code execution is:

```
Traceback (most recent call last):
   File "/Users/adriancopie/Projects/Personal/advanced/
soldier_13.py", line 23, in <module>
    soldier = Soldier("John", "sword", 30)
TypeError: Can't instantiate abstract class Soldier
with abstract methods what_am_i
```

Even there is an implementation for an abstract method in an abstract class, the abstract cannot be instantiated.

More than this, it is possible to even call an abstract method inside the base class from the derived class using `super()`:

```
class Swordsman(Soldier):
    def __init__(self, name, weapon, defense, gear):
        super().__init__(name, weapon, defense)
        self.gear = gear

    def what_am_i(self):
        super().what_am_i()
        print("I am a swordsman")

swordsman = Swordsman("John", "sword", 30, "belt")
swordsman.what_am_i()
```

The result is:

```
I am a basic soldier!
I am a swordsman
```

Why are abstract classes useful?

- They offer a semantic contract between clients/callers and classes implementation
- Protection level for missing implementation of mandatory methods in subclasses
- Very common when an Application Program Interface (API) is delivered and the implementation is left for third parties

Data hiding

Refers to cover up the implementation details for a class

Python philosophy: “we are all consenting adults”

Everything is public inside a class

Weakly-private mechanism

Let’s consider a Python module called soldier.py having the following content:

```
secret_magic = "ice"
_secret_magic = "fireball"

def not_secret_magic():
    print("I am not a secret magic")
```

Then, in another Python file we write:

```python
from soldier import *

print(secret_magic)
print(_secret_magic)
```

The result of the execution is:

```
Traceback (most recent call last):
   File "/Users/adriancopie/Projects/Personal/advanced/
soldier_17.py", line 5, in <module>
    print(_secret_magic)
NameError: name '_secret_magic' is not defined
ice
```

Note that omission will take place only we are using the syntax

```
from <module_name> import *
```

If, for example, we would write:

```
from soldier import secret_magic, _secret_magic

print(secret_magic)
print(_secret_magic)
```

The result is:

```
ice
fireball
```

This mechanism works with methods too, in the same `soldier.py` module we have:

```
def _secret_magic():
    print('I am a secret magic')

def not_secret_magic():
    print("I am not a secret magic”)
```

and in a separate Python file we write:

```
from soldier import *

not_secret_magic()
_secret_magic()
```

The result of these calls being:

```
I am not a secret magic
Traceback (most recent call last):
    File "/Users/adriancopie/Projects/Personal/advanced/
soldier_17.py", line 8, in <module>
    _secret_magic()
NameError: name '_secret_magic' is not defined
```

if we take rid of the importing everything from that module and choose individual functions, the protection mechanism is eluded:

```
from soldier import _secret_magic, not_secret_magic

not_secret_magic()
_secret_magic()
```

the result is now:

```
I am not a secret magic
I am a secret magic
```

Strongly-private mechanism

This is a very poor protection mechanism in Python, but we have a better one. Let's look at the code below:

```
class Soldier():

    def __init__(self, name, secret_weapon):
        self.name = name
        self.__secret_weapon = secret_weapon

soldier = Soldier("John", "knife")
print(soldier.__secret_weapon)
```

with the following outcome:

```
Traceback (most recent call last):
    File "/Users/adriancopie/Projects/Personal/advanced/
soldier_16.py", line 24, in <module>
    print(soldier.__secret_weapon)
AttributeError: 'Soldier' object has no attribute
'__secret_weapon'
```

a **mangling** operation comes into stage, the name of the variable is dynamically changed into something like :

```
_<class_name>__<variable_name>
```

which in our case will be

```
_Soldier__secret_weapon.
```

Because, however, everything in Python is publicly accessible, we still can access our `__secret_weapon` attribute like:

```
soldier._Soldier__secret_weapon = "bow"
print(soldier._Soldier__secret_weapon)
```

the execution result is :

```
bow
```

Conclusion:

Even this technique is possible, this should be highly discouraged. The creator of the class had a precise reason to prefix the name of the variables with the double underscore, willing to announce the users of the class about the private character of the variables, so it would be a good idea to not try to elude this convention.

Polymorphism

It is possible for two or more classes to have methods with the same name but with different implementations. In this case we can call those methods on the instances of their classes like there is no difference between classes. Those functions act different, they have different forms on different classes, or in other words they are *polymorphic*. The word polymorph comes from Greek and means “many forms”. Polymorphism is a very important attribute of OOP and Python supports is.

Polymorphism with functions

Polymorphism is an endemic Python attribute, since it has native support for OOP. Considering the following lines of code:

```
len_1 = len("Polymorphism")
len_2 = len([1, 2, 3, 4, 5])

print(len_1)
print(len_2)
```

The output of these execution is:

```
12
5
```

Polymorphism with class methods

```python
class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def what_am_i(self):
        print("I am a soldier")


class Citizen:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def what_am_i(self):
        print("I am a citizen")
```

These classes are unrelated

```python
obj_1 = Soldier("John", "sword", 30)
obj_2 = Citizen("Richard", 30)

persons = [obj_1, obj_2]
for person in persons:
    person.what_am_i()
```

The result of running this code is:

```
I am a soldier
I am a citizen
```

Polymorphism with inheritance

```
class Soldier():
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def what_am_i(self):
        print("I am a soldier")


class Swordsman(Soldier):
    def __init__(self, name, weapon, defense, gear):
        super().__init__(name, weapon, defense)
        self.gear = gear

    def what_am_i(self):
        print("I am a swordsman")
```

```python
class Spearman(Soldier):
    def __init__(self, name, weapon, defense, spear_type):
        super().__init__(name, weapon, defense)
        self.spear_type = spear_type

    def what_am_i(self):
        print("I am a spearman")


soldier = Soldier("John", "sword", 30)
swordsman = Swordsman("Richard", "sword", 40, "knife")
spearman = Spearman("Harry", "spear", 20, "halberd")

army = [soldier, swordsman, spearman]

for s in army:
    s.what_am_i()
```

The result of the code execution is:

```
I am a soldier
I am a swordsman
I am a spearman
```

which shows us that for every object the correct `what_am_i()` method was called.

Polymorphism with functions and objects

```
def say_my_occupation(obj):
    obj.what_am_i()
```

We are passing an object as parameter

```
soldier = Soldier("John", "sword", 30)
swordsman = Swordsman("Richard", "sword", 40, "knife")
spearman = Spearman("Harry", "spear", 20, "halberd")

army = [soldier, swordsman, spearman]

def say_my_occupation(obj):
    obj.what_am_i()

for s in army:
    say_my_occupation(s)
```

Same classes definition like before

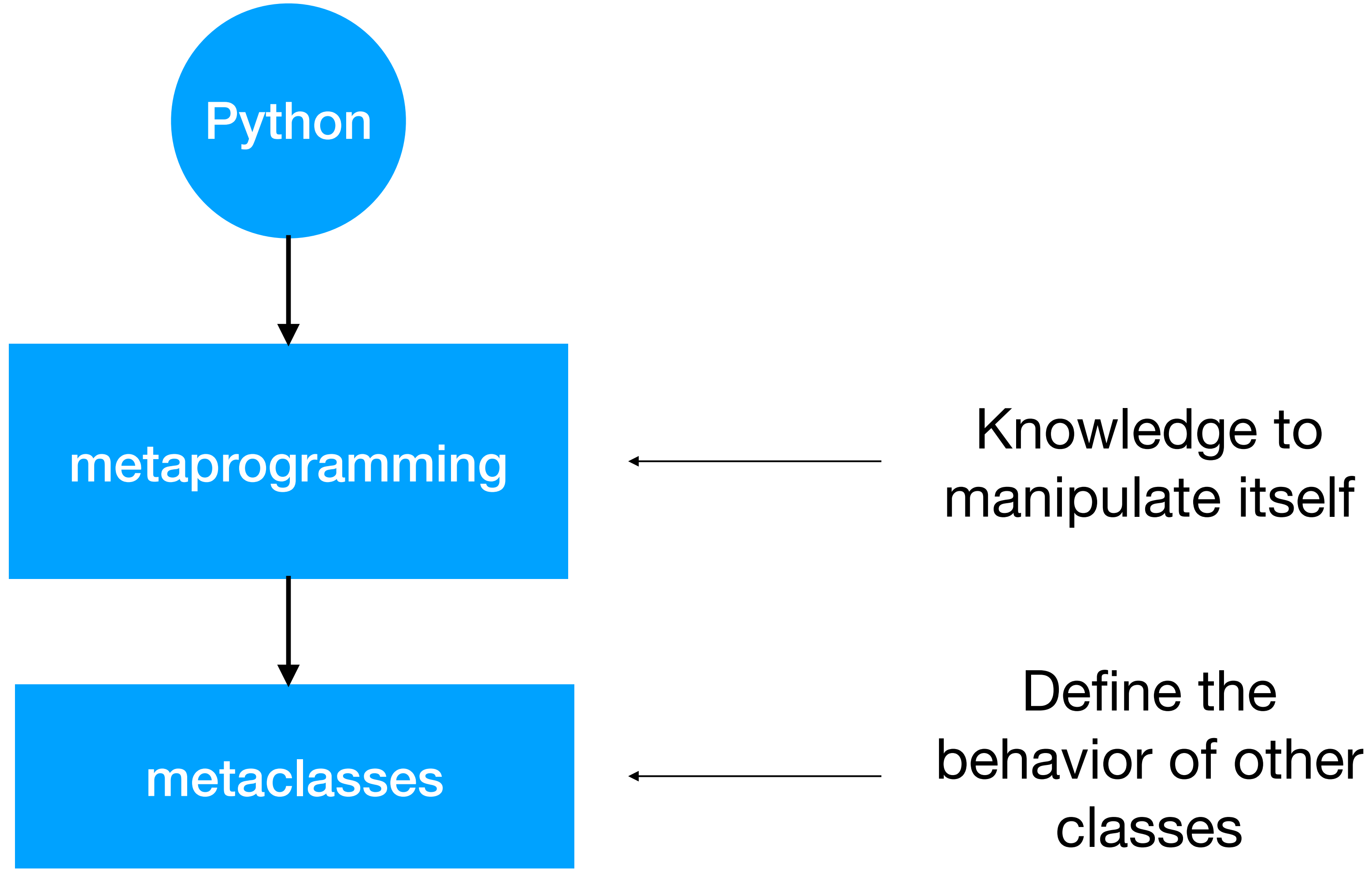
The result of the call is again:

```
I am a soldier
I am a swordsman
I am a spearman
```

Conclusion

By using polymorphism, which in Python is achieved through *method overriding*, we can created interfaces which perform similar tasks in many different ways and provide a convenient way to maintain the code in a flexible way.

Metaclasses



```
class Soldier:
    pass

soldier = Soldier()
print(soldier)
```

We have defined a class `Soldier` which actually does nothing and then we instantiate it. Displaying the created object leads to something that we expect:

```
<__main__.Soldier object at 0x106e8a3d0>
```

In Python, everything is an object, so they are the classes too and it is legitimate for us to ask ourselves what is their type?

```
print(type(soldier))
```

```
<class ‘__main__.Soldier'>
```

We expected this because `soldier` is an instance of the `Soldier` class

Now, we go one step further and run:

```
print(type(Soldier))
```

with the result:

```
<class ‘type'>
```

Avoid the confusion between function `type()` and the type `type`

We expected the type of `Soldier` to be `class`, but instead is `type` so we can deduce the fact the type of the `Soldier` class is `type`.

Now, let’s try to see what is the type of `type`

```
print(type(type))
```

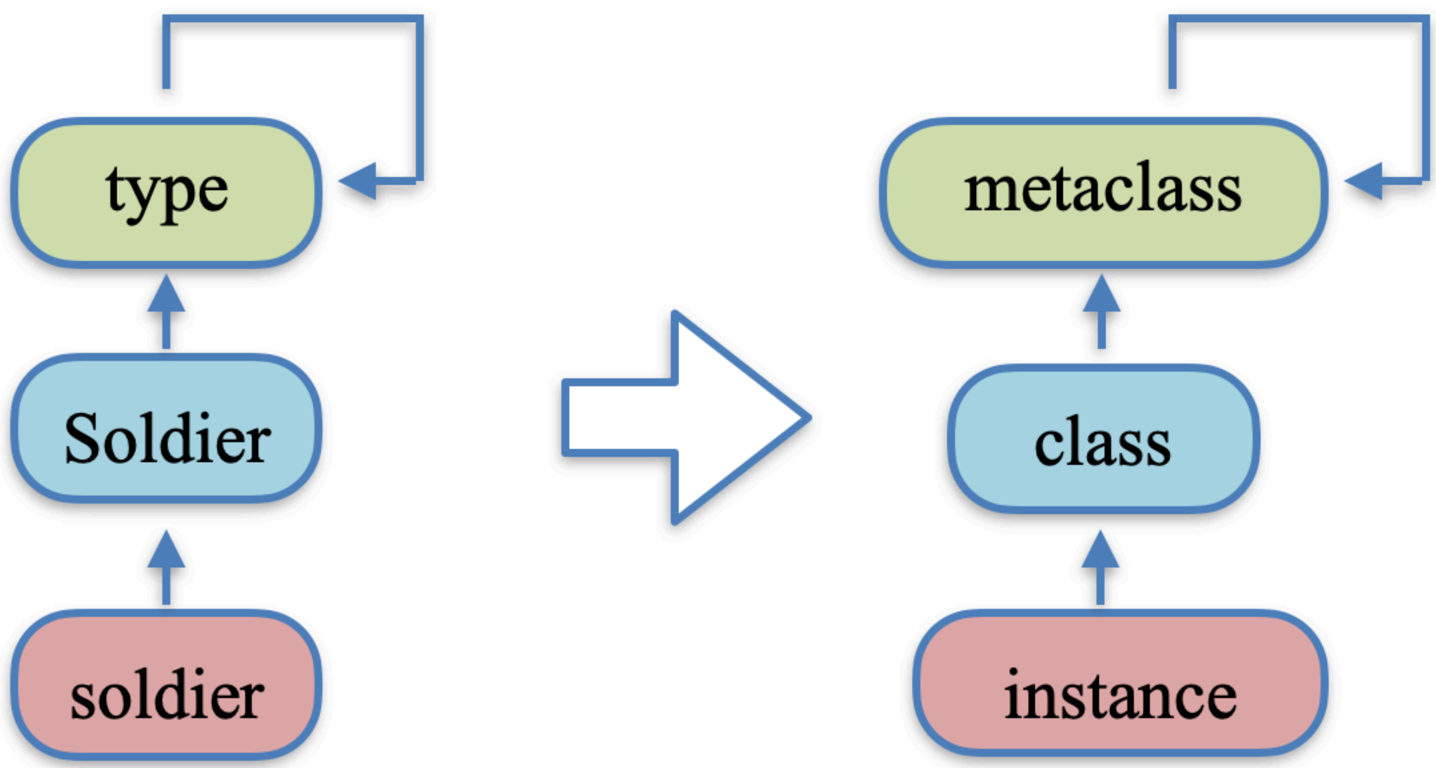
The result being:

```
<class ‘type’>
```

interesting, isn’t it?

The type of `type` is `type` again!

`type` is the root of all classes



now, call `type()` with three parameters

```
SoldierClass = type('Soldier', (), {})

print(SoldierClass)
print(type(SoldierClass))
```

name of the class

reference to the class

The outcome of this little piece of code is:

```
<class '__main__.Soldier'>
<class ‘type'>
```

Equivalent

```
class Soldier:
    pass
```

```
print(Soldier)
```

```
<class '__main__.Soldier'>
```

After that when we can instantiate an object like as we are used to:

```
soldier = SoldierClass()
```

and if we print its type we see

```
<__main__.Soldier object at 0x100f32690>
```

Even we used a `SoldierClass()` call, we notice that the type of the `soldier` instance is `Soldier`, which is correct, because ‘`Soldier`’ was passed as the first parameter in `type()` function

In the `type()` function, we can provide a dictionary of attributes as the third parameter, which will become the namespace of the class

```
Soldier = type('Soldier', (), {‘increase_defense_ratio’:1.1})

print(Soldier.__dict__)
```

```
{'increase_defense_ratio': 1.1, '__module__': '__main__', '__dict__': <attribute '__dict__' of 'Soldier' objects>, '__weakref__': <attribute '__weakref__' of 'Soldier' objects>, '__doc__': None}
```

Similar to

```
class Soldier:
    increase_defense_ratio = 1.1

    def __init__(self, name, weapon, defense):
        …
```

If we need that our class must inherit one or more classes, we have to provide them in a tuple, which will be the second parameter of the `type()` function.

```
Soldier = type('Soldier', (), {'increase_defense_ratio': 1.10})
Swordsman = type('Swordsman', (Soldier,), {})

print(help(Swordsman))
```

```
class Swordsman(Soldier)
 |  Method resolution order:
 |      Swordsman
 |      Soldier
 |      builtins.object
```

Notice the way in which we write a tuple let’s reload our previous example about multiple inheritance and make it work with metaclasses. If you remember, we have built a `SuperSwordsman` class which inherited data and behavior from `Soldier` and `Witcher` classes.

```
Soldier = type('Soldier', (), {'increase_defense_ratio': 1.10})
Witcher = type('Witcher', (), {})
SuperSwordsman = type('SuperSwordsman', (Soldier,Witcher), {})

print(help(SuperSwordsman))
```

```
class SuperSwordsman(Soldier, Witcher)
 |  Method resolution order:
 |      SuperSwordsman
 |      Soldier
 |      Witcher
 |      builtins.object
```

See the multiple inheritance

In the next examples we'll make use of the `__class__` attribute of a class, which returns the type of the object on which is applied.

```
print(SuperSwordsman.__class__)
```

will display:

```
<class 'type'>
```

Similar to
`type(SuperSwordsman)`

In the same way, let's create a string object using `str`

```
weapon = str("sword")
print(weapon.__class__)
```

and the result is:

```
<class 'str'>
```

because weapon is an instance of a class string, but if we write

```
print(weapon.__class__.__class__)
```

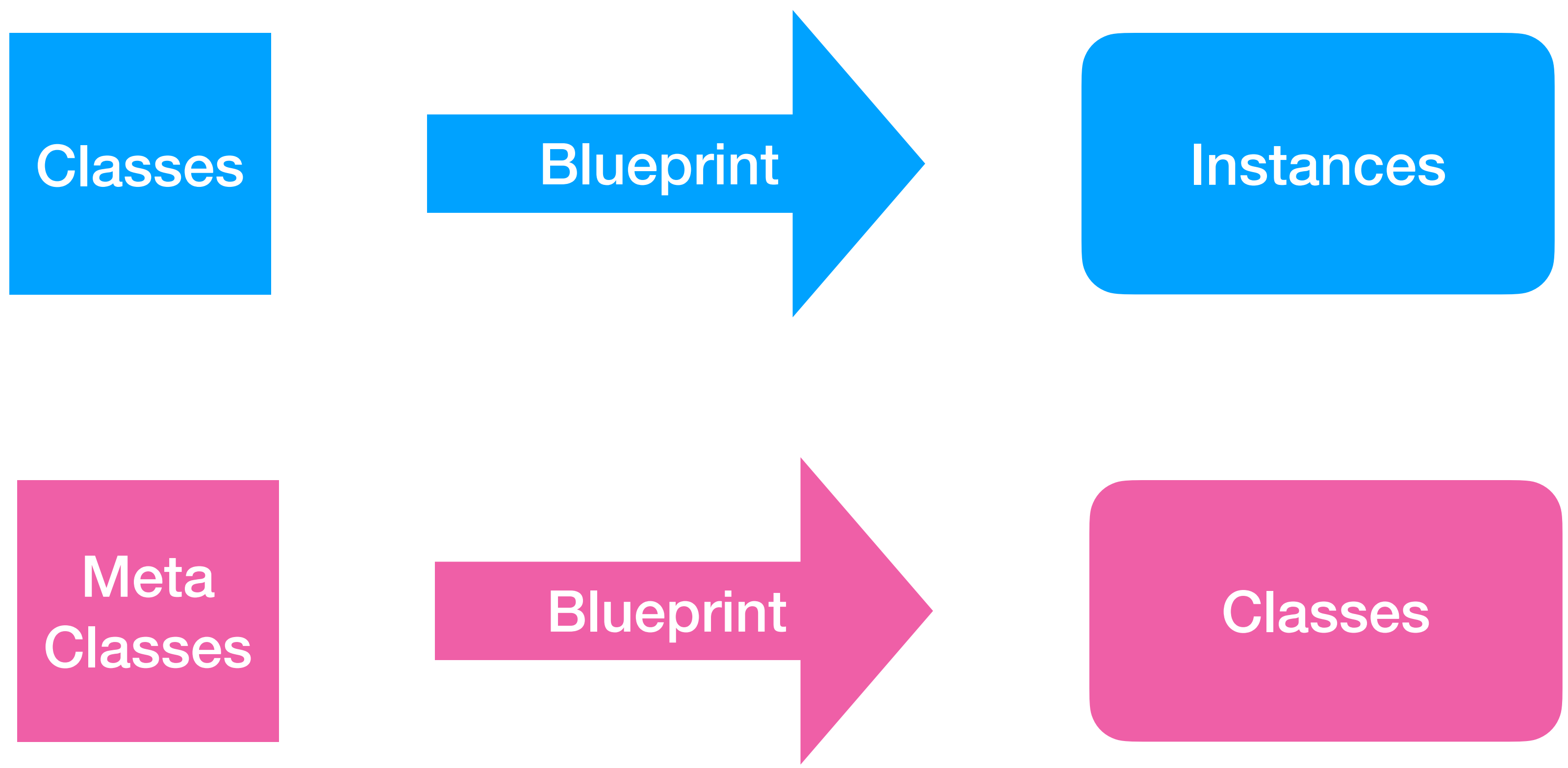
the result will be

```
<class 'type'>
```

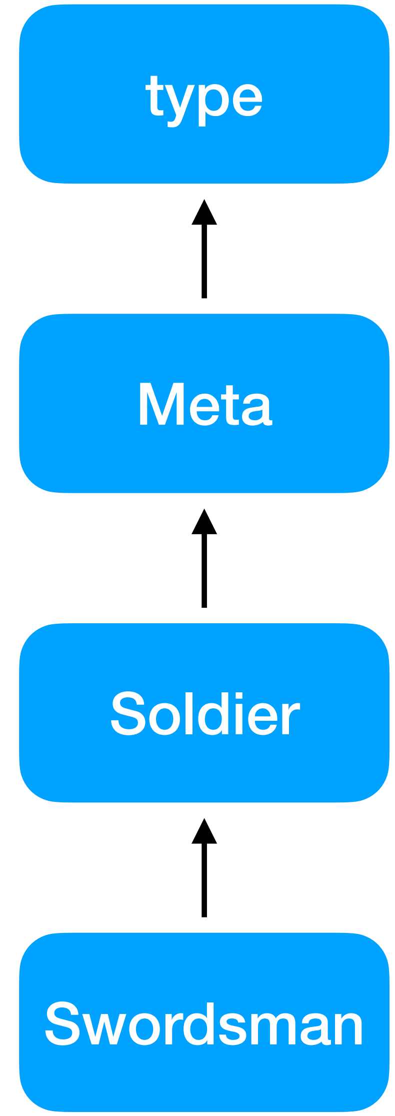
because we have applied the last `__class__` attribute over a `str` type (see previous example), so `type` is the metaclass even for those "basic" data types.

Custom Metaclasses

OOP



```python
class Meta(type):
    pass

class Soldier(metaclass=Meta):
    pass

class Swordsman(Soldier):
    pass


print(type(Meta))
print(type(Soldier))
print(type(Swordsman))
```




```
<class 'type'>
<class '__main__.Meta'>
<class ‘__main__.Meta'>
```

Notice that the parent class for Soldier and Swordsman is Meta

```
class Meta(type):
    def __new__(cls, classname, parent, attributes):
        print("__new__ in Meta")
        print(f"classname = {classname}")
        print(f"parent = {parent}")
        print(f"attributes = {attributes}")
        return type.__new__(cls, classname, parent, attributes)


class Soldier(metaclass=Meta):
    def __new__(cls, classname, parent, attributes):
        print("__new__ in Soldier")
        return type.__new__(cls, classname, parent, attributes)
```

If we run the code above just like that, only with class definitions, without instantiating any object from these classes we obtain:

```
__new__ in Meta
classname = Soldier
parent = ()
attributes = {'__module__': '__main__', '__qualname__':
'Soldier', '__new__': <function Soldier.__new__ at
0x10330f200>}
```

After the class statement ends, the parent metaclass is called automatically

Modifying the attributes of a class

```
class Meta(type):
    def __new__(cls, cls_name, parents, attrs):
        print(attrs)

        new_dict = {}
        for key, val in attrs.items():
            if key.startswith("__"):
                new_dict[key] = val
            else:
                new_dict[key.upper()] = val

        return type(cls_name, parents, new_dict)

class Soldier(metaclass=Meta):
    increase_defense_ratio = 1.1

    def show_name(self):
        pass
```

Every custom attribute is transformed to uppercase

```
print(Soldier.increase_defense_ratio)
print(Soldier.show_name)
```

```
Traceback (most recent call last):
  File "/Users/adriancopie/Projects/Personal/advanced/
metaclass_7.py", line 20, in <module>
    print(Soldier.increase_defense_ratio)
AttributeError: type object 'Soldier' has no attribute
'increase_defense_ratio'
{'__module__': '__main__', '__qualname__': 'Soldier',
'increase_defense_ratio': 1.1, 'show_name': <function
Soldier.show_name at 0x10a9d9160>}
```

```
print(Soldier.INCREASE_DEFENSE_RATIO)
print(Soldier.SHOW_NAME)
```

```
{'__module__': '__main__', '__qualname__': 'Soldier',
'increase_defense_ratio': 1.1, 'show_name': <function
Soldier.show_name at 0x10ce14160>}
1.1
<function Soldier.show_name at 0x10ce14160>
```

Registering classes in a system

```
class GameRegistry(type):
    registry = []

    def __new__(cls, classname, parent, attributes):
        cls.registry.append(classname)
        return type.__new__(cls, classname, parent, attributes)

class Soldier(metaclass=GameRegistry):
    def __new__(cls, classname, parent, attributes):
        return type.__new__(cls, classname, parent, attributes)

class Swordsman(Soldier):
    pass

print(Soldier.registry)
```

the result of the call is

```
['Soldier', ‘Swordsman’]
```

When to use metaclasses

- Dynamically add attributes or methods to a class
- Registering classes or plugins in a system registry
- Code generators
- API development
- …

Operator overloading

By overloading Python built in operators, we are giving another meaning to the original operations

```
a = 2 + 3
print(a)
```

5

```
b = 'abc' + 'def'
print(b)
```

abcdef

```
c = 4 * 5
print(c)
```

20

```
d = 'abc' * 3
print(d)
```

abcabcabc

```
e = 8 - 5
print(e)
```

3

```
s1 = {1, 2, 3}
s2 = {1}
print(s1 - s2)
```

{2,3}

__str__ and __repr__

```
class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense


soldier = Soldier("John", "sword", 30)
print(soldier)
print(str(soldier))
```

```
<__main__.Soldier object at 0x100ea61d0>
<__main__.Soldier object at 0x100ea61d0>
```

The output does not give us too much info

```python
class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def __str__(self):
        return f'name = {self.name}, weapon = {self.weapon}, 
defense = {self.defense}'


soldier = Soldier("John", "sword", 30)
print(soldier)
print(str(soldier))
```

```
name = John, weapon = sword, defense = 30
name = John, weapon = sword, defense = 30
```

better output

```
# overloading the __str__ and __repr__ methods

class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def __str__(self):
        return f'name = {self.name}, weapon = {self.weapon},
defense = {self.defense}'

    def __repr__(self):
        return f'Soldier("{self.name}", "{self.weapon}",
"{self.defense}")'
```

```python
soldier = Soldier("John", "sword", 30)
print(soldier)
print(str(soldier))
print(repr(soldier))

other = eval(repr(soldier))
print(type(other))
```

```
name = John, weapon = sword, defense = 30
name = John, weapon = sword, defense = 30

Soldier("John", "sword", "30")
<class '__main__.Soldier'>
```

`__repr__` goal is to be unambiguous
`__str__` goal is to be readable

Overloading the “+” operator

Let’s assume that, through a spell, we can “melt” two soldiers together

```
# overloading the addition operator "+"

class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def __add__(self, other):
        return Soldier(self.name, self.weapon, self.defense +
other.defense)

soldier_one = Soldier("John", "sword", 30)
soldier_two = Soldier("Richard", "spear", 40)

print((soldier_one + soldier_two).defense)
```

```
70
```

Overloading the comparison “<” operator

```
class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def __lt__(self, other):
        if self.defense < other.defense:
            return True
        else:
            return False
```

```python
soldier_one = Soldier("John", "sword", 30)
soldier_two = Soldier("Richard", "spear", 40)

if soldier_one < soldier_two:
    print(f'soldier {soldier_one.name} is weaker than 
{soldier_two.name}')
else:
    print(f'soldier {soldier_one.name} is stronger than 
{soldier_two.name}')
```

```
soldier John is weaker than Richard
```

Overloading the comparison “>” operator

```
# overloading the comparison operator ">"

class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def __gt__(self, other):
        if self.defense > other.defense:
            return True
        else:
            return False
```

```
soldier_one = Soldier("John", "sword", 30)
soldier_two = Soldier("Richard", "spear", 40)

if soldier_one > soldier_two:
    print(f'soldier {soldier_one.name} is stronger than
{soldier_two.name}')
else:
    print(f'soldier {soldier_one.name} is weaker than
{soldier_two.name}')
```

```
soldier John is weaker than Richard
```

Overloading the equivalence "=" operator

```
class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def __eq__(self, other):
        if self.name == other.name and \
                self.weapon == other.weapon and \
                self.defense == other.defense:
            return True
        else:
            return False
```

```
soldier_one = Soldier("John", "sword", 30)
soldier_two = Soldier("Richard", "spear", 40)
soldier_three = Soldier("John", "sword", 30)

if soldier_one == soldier_two:
    print(f'soldier {soldier_one.name} is the same as
{soldier_two.name}')
elif soldier_one == soldier_three:
    print(f'soldier {soldier_one.name} is the same as
{soldier_three.name}')
elif soldier_two == soldier_three:
    print(f'soldier {soldier_two.name} is the same as
{soldier_three.name}')
```

```
soldier John is the same as John
```

Call an instance like calling a function

```
class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def __call__(self):
        print(f"Hi, I am {self.name}, and I carry a 
{self.weapon}")


soldier = Soldier("John", "sword", 30)
soldier()
```

```
Hi, I am John, and I carry a sword
```

even `soldier` is an instance of the `Soldier` class, it can be invoked like a method

Call an instance like calling a function with variable number of arguments

```
class Soldier:
    def __init__(self, name, weapon, defense):
        self.name = name
        self.weapon = weapon
        self.defense = defense

    def __call__(self, *args, **kwargs):
        print(args)
        print(kwargs)


soldier = Soldier("John", "sword", 30)
soldier(1, "john", weapon="sword")
soldier(7, 9, "spear", defense=30, name="John")
```

```
(1, 'john')
{'weapon': 'sword'}
(7, 9, 'spear')
{'defense': 30, 'name': 'John'}
```

Creating an iterator

Iterators are objects that allow you to traverse through all the elements of a collection, regardless of its specific implementation.

An **iterable** object is an object that implements `__iter__`, which is expected to return an **iterator** object.

An **iterator** is an object that implements `next`, which is expected to return the next element of the iterable object that returned it, and raise a `StopIteration` exception when no more elements are available.

Creating an iterator for numbers

```
class NumberIterable:

    def __iter__(self):
        self.current = 0
        return self

    def __next__(self):
        self.current += 1
        return self.current


num_it = NumberIterable()
i = iter(num_it)

print(next(i))
print(next(i))
print(next(i))
```

Call the iterator for a finite number of iterations

```
1
2
3
```

Creating an iterator for numbers

```
class NumberIterable:

    def __iter__(self):
        self.current = 0
        return self

    def __next__(self):
        self.current += 1
        return self.current

iterator = NumberIterable()

for i in iterator:
    print(i)
```

Approach 1
Infinite number of generated elements

```
1607692
1607693
1607694Traceback (most recent call last):
  File "/Users/adriancopie/Projects/Personal/advanced/
overloaded_8.py", line 15, in <module>
    print(i)
KeyboardInterrupt
```

Creating an iterator for numbers

```
class NumberIterator:

    def __init__(self, max_val):
        self.max_val = max_val

    def __iter__(self):
        self.current = 0
        return self

    def __next__(self):
        if self.current < self.max_val:
            self.current += 1
            return self.current
        else:
            raise StopIteration

iterator = NumberIterator(3)

for i in iterator:
    print(i)
```

Approach 2
Limited number of generated elements

```
1
2
3
```

Creating an iterator for Objects

Suppose that we need to model a quiver of arrows for our Bowman fighter. We implement it as an iterator, so we can then loop over its content.

```
class Arrow:
    pass

class Quiver:
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.count = 0
        return self

    def __next__(self):
        if self.count < self.max:
            x = Arrow()
            self.count += 1
            return x
        else:
            raise StopIteration
```

```
quiver = Quiver(3)
iterator = iter(quiver)

print(next(iterator))
print(next(iterator))
print(next(iterator))
print(next(iterator))
```

first approach of looping using the next() function

```
<__main__.Arrow object at 0x10af77250>
<__main__.Arrow object at 0x10af77250>
<__main__.Arrow object at 0x10af77250>
Traceback (most recent call last):
  File "/Users/adriancopie/Projects/Personal/advanced/
overloaded_8.py", line 30, in <module>
    print(next(iterator))
  File "/Users/adriancopie/Projects/Personal/advanced/
overloaded_8.py", line 22, in __next__
    raise StopIteration
StopIteration
```

second approach of looping using the for statement

```
quiver = Quiver(3)
iterator = iter(quiver)

for i in iterator:
    print(i)
```

```
<__main__.Arrow object at 0x107a79210>
<__main__.Arrow object at 0x107a792d0>
<__main__.Arrow object at 0x107a79210>
```

Decorators

Functions in Python

Block of reusable code which is used to perform a well defined, single action

Functions provide better modularity and code reusability

First class objects (citizens) - they can passed as arguments to other functions and used like any other object in Python (string, float, list, …)

function definition

argument

```python
def increase_defense(defense):
    return defense + 1


print(defense(30))
```

returned result

body of the function

```
31
```

Functions in Python always return something

```
def increase_defense(current_defense):
    return current_defense + 1


def show_defense(defense):
    print(str(defense))


increased_defense = increase_defense(30)
print(type(increased_defense))

res = show_defense(30)
print(type(res))
```

```
<class 'int'>
30
<class 'NoneType'>
```

Even the `show_defense` function has not a `return` statement, it returns `None`

Side effects of the functions

```
def show_defense(defense):
    print(str(defense))
```

`show_defense` function has sides effects, even if it only prints a message:

- It returns `None`
- It outputs something at the console

```python
defense = 30


def set_defense(new_defense):
    global defense
    defense = 35

def show_defense():
    print(defense)

set_defense(50)
show_defense()
set_defense(60)
show_defense()
```

Modify value outside its body

Rely on value outside its body

```
35
35
```

Pure functions

- Functions that do not cause any side effect, neither rely on any
- Their output depends only on their input
- In what will follow we'll consider that a function will return a value based on a given argument

Assign a function to a variable

```
def increase_defense(defense):
    return defense + 1


increment_fn = increase_defense
new_defense = increment_fn(5)
print(new_defense)
```

```
6
```

we call a function by the variable's name increase_defense() is defined inside more_power()

Define a function inside another function

```
def more_power(defense):
    def increase_defense(defense):
        return defense + 1

    new_power = increase_defense(defense)
    return new_power

res = more_power(7)
print(res)
```

```
8
```

Pass a function as an argument to another function

increase_defense is an argument for super_power()

```python
def increase_defense(defense):
    return defense + 1

def super_power(func):
    extra_defense = 10
    return func(extra_defense)

new_defense = super_power(increase_defense)
print(new_defense)
```

```
11
```

Functions returning other functions

```
def strenght_up():
    def increase_defense(defense):
        return defense + 1
    return increase_defense

strenght_func = strenght_up()
print(strenght_func(5))
```

`increase_defense` is returned from `strenght_up()` function

```
6
```

Nested functions having access to enclosing function's variables

```
def say_something(speech):
    def say_this():
        print(speech)
    say_this()

say_something("Greetings from Timisoara!")
```

```
Greetings from Timisoara!
```

`speech` argument is defined outside the `say_this()` function

`speech` is not sent as parameter for `say_this()` function

Closures

```
def say_something(greetings):
    def say_this(where_from):
        return f'{greetings} {where_from} !'
    return say_this

say_smth_func = say_something("Greetings from ")
print(say_smth_func("Timisoara"))

print(dir())
del(say_something)
print(dir())
print(say_smth_func("West University!"))
```

get the inner function

run the inner function with parameter

Even after the function deletion, the inner function has access to the enclosing param

```
Greetings from  Timisoara !
['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'say_smth_func', 'say_something']
['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'say_smth_func']
Greetings from  West University!
```

We say that the say_this() function is a closure.

A `Closure` is a function object that remembers values in enclosing scopes even if they are not present in memory

We can do some introspection for the previous functions:

```
print(say_something.__closure__)
print(say_smth_func.__closure__)
```

```
None
(<cell at 0x109b887d0: str object at 0x109c0a0b0>,)
```

One can see that for `say_something()` function, the `__closure__` attribute is set to `None`, while for `say_smth_func()` the `__closure__` attribute has a value, so `say_smth_func()` is a closure.

Decorators

Decorator

```
def upper_decorator(func):
    def wrapper():
        f = func()
        to_upper = f.upper()
        return to_upper

    return wrapper
```

The function to be decorated

```
def greetings():
    return "Greetings from Timisoara!"

decorator = upper_decorator(greetings)
print(decorator())
```

```
GREETINGS FROM TIMISOARA!
```

Decorators

Decorator

```
def upper_decorator(func):
    def wrapper():
        f = func()
        to_upper = f.upper()
        return to_upper

    return wrapper

@upper_decorator
def greetings():
    return "Greetings from Timisoara!"

print(greetings())
```

The function to be decorated

```
GREETINGS FROM TIMISOARA!
```

Decorators augment the work of an original function without altering it

Applying multiple decorators

```
def upper_phrase(func):
    def wrapper():
        f = func()
        to_upper = f.upper()
        return to_upper

    return wrapper


def hyphen_phrase(func):
    def wrapper():
        f = func()
        hyphened = f.replace(' ', '-')
        return hyphened

    return  wrapper

@hyphen_phrase
@upper_phrase
def greetings():
    return "Greetings from Timisoara!"
```

First decorator

Second decorator

`d2 = hyphen_phrase(upper_decorator(greetings))`

```
GREETINGS-FROM-TIMISOARA!
```

Arguments in decorators

```
def custom_decorator(func):
    def wrapper(arg1, arg2):
        func(arg1, arg2)

    return wrapper

@custom_decorator
def greetings(name, city):
    print(f'Hello {name} from {city}!')


greetings("John", "Timisoara")
```

Arguments are in `wrapper` function

```
Hello John from Timisoara!
```

Decorating functions with an arbitrary number of arguments

Generic way of passing arguments

```
def complex_decorator(func):
    def complex_wrapper(*args, **kwargs):
        func(*args, **kwargs)
    return complex_wrapper

@complex_decorator
def greetings(name, city, country, planet):
    print(f"Hello {name} from {city} in the {country} country, on
planet {planet}!")


greetings("John", "Timisoara", country="Romania", planet="Earth")
```

```
Hello John from Timisoara in the Romania country, on the
planet Earth!
```

Decorators with parameters

One more nesting level

```
def augmented_decorator(dec_arg1, dec_arg2):
    def decorator(func):
        def wrapper(func_arg1, func_arg2):
            print(func(func_arg1, func_arg2) + f" in the {dec_arg1}
country, on the planet {dec_arg2}!")
        return wrapper
    return decorator

@augmented_decorator("Romania", "Earth")
def greetings(name, city):
    return f"Hello {name} from {city}"

greetings("John", "Timisoara")
```

wrapper function has access to all the arguments

```
Hello John from Timisoara in the Romania country, on the
planet Earth!
```