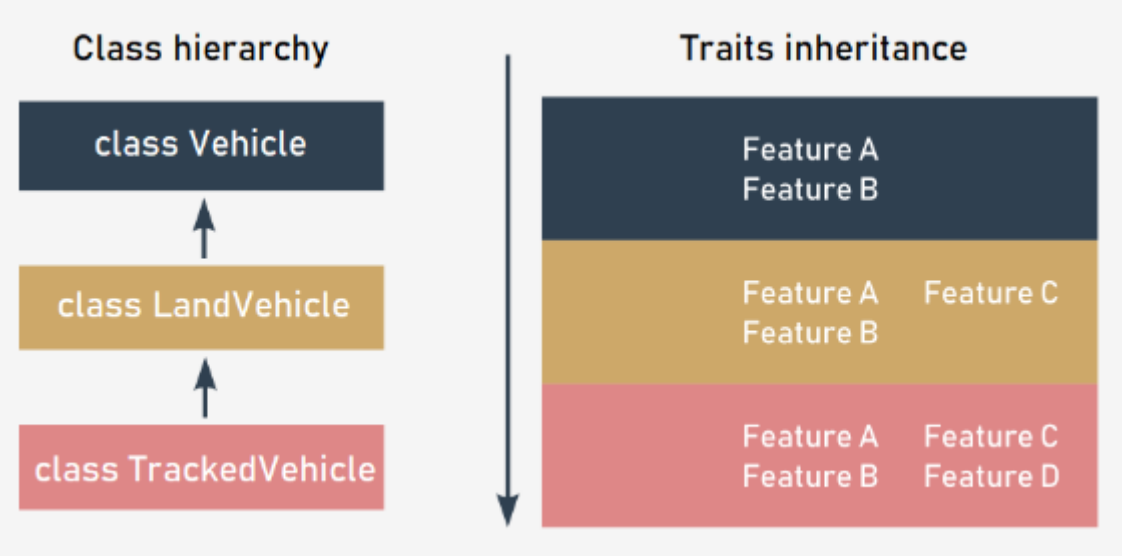
2.2 Inheritance and polymorphism — Inheritance as a pillar of OOP

Inheritance is one of the fundamental concepts of object oriented programming, and expresses the fundamental relationships between classes: superclasses (parents) and their subclasses (descendants). Inheritance creates a class hierarchy. Any object bound to a specific level of class hierarchy inherits all the traits (methods and attributes) defined inside any of the superclasses.

This means that inheritance is a way of building a new class, not from scratch, but by using an already defined repertoire of traits. The new class inherits (and this is the key) all the already existing equipment, but is able to add some new features if needed.

Each subclass is more specialized (or more specific) than its superclass. Conversely, each superclass is more general (more abstract) than any of its subclasses. Note that we've presumed that a class may only have one superclass — this is not always true, but we'll discuss this issue more a bit later.



A very simple example of two-level inheritance is presented here:

```
class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass
```

All the presented classes are empty for now, as we're going to show you how the mutual relations between the super- and subclasses work.

We can say that:

- the **Vehicle** class is the superclass for both the **LandVehicle** and **TrackedVehicle** classes;
- the **LandVehicle** class is a subclass of **Vehicle** and a superclass of **TrackedVehicle** at the same time;
- the **TrackedVehicle** class is a subclass of both the **Vehicle** and **LandVehicle** classes.

Inheritance and polymorphism — Single inheritance vs. multiple inheritance

There are no obstacles to using multiple inheritance in Python. You can derive any new class from more than one previously defined class.

But multiple inheritance should be used with more prudence than single inheritance because:

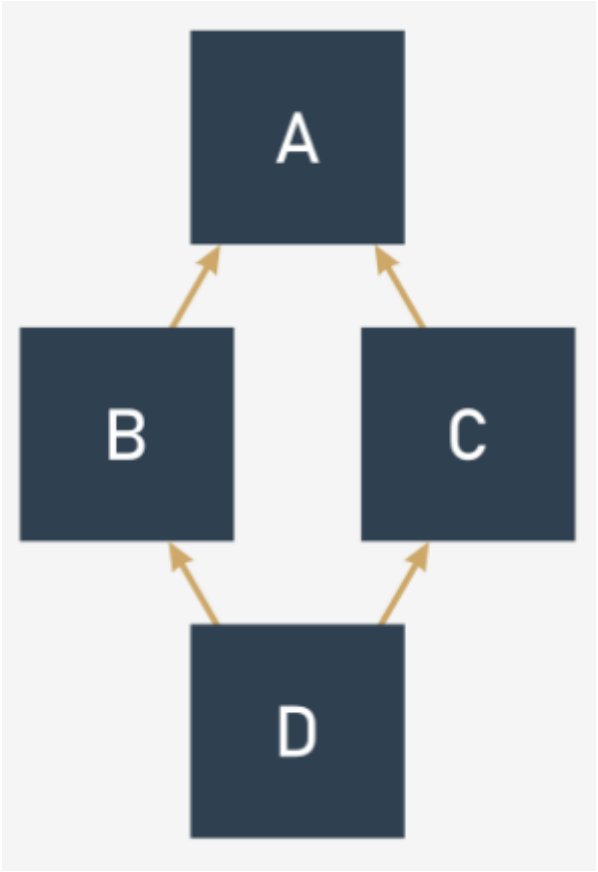
- a single inheritance class is always simpler, safer, and easier to understand and maintain;
- multiple inheritance may make method overriding somewhat tricky; moreover, using the super() function can lead to ambiguity;
- it is highly probable that by implementing multiple inheritance you are violating the single responsibility principle;

If your solution tends to require multiple inheritance, it might be a good idea to think about implementing composition.

MRO - Method Resolution Order

The spectrum of issues possibly coming from multiple inheritance is illustrated by a classical problem named the diamond problem, or even the deadly diamond of death. The name reflects the shape of the inheritance diagram — take a look at the picture.

There is the top-most superclass named A; there are two subclasses derived from A — B and C; and there is also the bottom-most subclass named D, derived from B and C (or C and B, as these two variants mean different things in Python) Can you see the diamond there?



The ambiguity that arises here is caused by the fact that class B and class C are inherited from superclass A,

and class D is inherited from both classes B and C. If you want to call the method info(), which part of the code would be executed then?

Python lets you implement such a class hierarchy. Can you guess the output of the code?

```
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(B, C):
    pass

D().info() # Class B
```

In the multiple inheritance scenario, any specified attribute is searched for first in the current class. If it is not found, the search continues into the direct parent classes in depth-first level (the first level above), from the left to the right, according to the class definition. This is the result of the MRO algorithm.

In our case:

- **class D** does not define the method `info()`, so Python has to look for it in the class hierarchy;
- **class D** is constructed in this order:
    - the definition of **class B** is fetched;
    - the definition of **class C** is fetched;
- Python finds the requested method in the **class B** definition and stops searching;
- Python executes the method.

Possible pitfalls - MRO inconsistency

MRO can report definition inconsistencies when a subtle change in the **class D** definition is introduced, which is possible when you work with complex class hierarchies.

Imagine that you have changed the **class D** definition from:

```
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(B, C):
```

```
    pass

D().info()
    pass
```

```
class D(A, C):
    pass
```

output

```
Traceback (most recent call last):
  File "diamond.py", line 13, in
    class D(A, C):
TypeError: Cannot create a consistent method resolution order (MRO) for bases
A, C
```

This message informs us that the MRO algorithm had problems determining which method (originating from the A or C classes) should be called.

Due to MRO, you should knowingly list the superclasses in the subclass definition. In the following example, class D is based on classes B and C, whereas class E is based on classes C and B (the order matters!).

```
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(B, C):
    pass

class E(C, B):
    pass

D().info()
E().info()
```

output

```
Class B
Class C
```

LAB

Objectives

- improving the student's skills in operating with multiple inheritance;
- pointing out the nature of multiple inheritance problems.

Scenario

Your task is to build a multifunction device (MFD) class consisting of methods responsible for document scanning, printing, and sending via fax. The methods are delivered by the following classes:

- scan(), delivered by the Scanner class;
- print(), delivered by the Printer class;
- send() and print(), delivered by the Fax class.

Each method should print a message indicating its purpose and origin, like:

- 'print() method from Printer class'
- 'send() method from Fax class'

- create an MFD_SPF class ('SPF' means 'Scanner', 'Printer', 'Fax'), then instantiate it;
- create an MFD_SFP class ('SFP' means 'Scanner', 'Fax', 'Printer'), then instantiate it;
- on each object call the methods: scan(), print(), send();
- observe the output differences. Was the Printer class utilized each time?

reposta

```
#!/usr/bin/env python
# -*- coding: utf-8 -*


class Scanner:
    def scan(self):
        print("scan() method from class Scanner")

class Printer:
    def print(self):
        print("print() method from Printer class")

class Fax:
    def send(self):
        print("send() method from Fax class")
    def print(self):
        print("print() method from Fax class")

class MFD_SPF(Scanner,Fax,Printer):
    pass
class MFD_SFP(Scanner,Printer,Fax):
    pass

oUn=MFD_SFP()
oDos=MFD_SPF()

oUn.scan()
oUn.print()
```

```
oUn.send()

oDos.scan()
oDos.print()
oDos.send()
```

Inheritance and polymorphism — Inheritance as a pillar of OOP

In Python, polymorphism is the provision of a single interface to objects of different types. In other words, it is the ability to create abstract methods from specific types in order to treat those types in a uniform way.

Imagine that you have to print a string or an integer — it is more convenient when a function is called simply **print**, not **print_string** or **print_integer**.

However, the string must be handled differently than the integer, so there will be two implementations of the function that lead to printing, but naming them with a common name creates a convenient abstract interface independent of the type of value to be printed.

The same rule applies to the operation of addition. We know that addition is expressed with the '+' operator, and we can apply that operator when we add two integers or concatenate two strings or two lists.

Let's see what methods are present for both built-in types (string and integer) responsible for handling the '+' operator. Once again, we'll use the dir(object) function, which returns a list of object attributes.

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__',
'__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__',
'__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
>>> dir('a')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```

As you can see, there are many attributes available for the string and integer types, many of them carrying the same names. The first name common to both lists is __add__, which is a special method responsible for handling addition, as you may remember from the previous slides.

To briefly demonstrate polymorphism on integers and strings, execute the following code in the Python interpreter:

```
a = 10
print(a.__add__(20))
b = 'abc'
print(b.__add__('def'))
```

output

```
30
abcdef
```

By the way, if you look for a method that is used when you print a value associated with an object, the **__str__** method is called to prepare a string that is used in turn for printing.

One way to carry out polymorphism is inheritance, when subclasses make use of base class methods, or override them. By combining both approaches, the programmer is given a very convenient way of creating applications, as:

- most of the code could be reused and only specific methods are implemented, which saves a lot of development time and improves code quality;
- the code is clearly structured;
- there is a uniform way of calling methods responsible for the same operations, implemented accordingly for the types.

Remember You can use inheritance to create polymorphic behavior, and usually that's what you do, but that's not what polymorphism is about.

In the right pane, there is a code implementing both inheritance and polymorphism:

- **inheritance**: class Radio inherits the t**urn_on()** method from its superclass — that is why we see The device was turned on string twice. Other subclasses override that method and as a result we see different lines being printed;
- **polymorphism**: all class instances allow the calling of the turn_on() method, even when you refer to the objects using the arbitrary variable element.

```
class Device:
    def turn_on(self):
        print('The device was turned on')

class Radio(Device):
    pass

class PortableRadio(Device):
    def turn_on(self):
        print('PortableRadio type object was turned on')

class TvSet(Device):
    def turn_on(self):
        print('TvSet type object was turned on')

device = Device()
radio = Radio()
portableRadio = PortableRadio()
tvset = TvSet()
```

```
for element in (device, radio, portableRadio, tvset):
    element.turn_on()
```

output

```
The device was turned on
The device was turned on
PortableRadio type object was turned on
TvSet type object was turned on
```

Duck typing is a fancy name for the term describing an application of the duck test: «If it walks like a duck and it quacks like a duck, then it must be a duck», which determines whether an object can be used for a particular purpose. An object's suitability is determined by the presence of certain attributes, rather than by the type of the object itself.

Duck typing is another way of achieving polymorphism, and represents a more general approach than polymorphism achieved by inheritance. When we talk about inheritance, all subclasses are equipped with methods named the same way as the methods present in the superclass.

In duck typing, we believe that objects own the methods that are called. If they do not own them, then we should be prepared to handle exceptions.

Let's talk about two things that share conceptually similar methods, but represent totally different things, like cheese and wax. Both can melt, and we use the melted forms for different purposes.

When you run the code you should receive the following output:

```
class Wax:
    def melt(self):
        print("Wax can be used to form a tool")

class Cheese:
    def melt(self):
        print("Cheese can be eaten")

class Wood:
    def fire(self):
        print("A fire has been started!")

for element in Wax(), Cheese(), Wood():
    try:
        element.melt()
    except AttributeError:
        print('No melt() method')
```

output

```
Wax can be used to form a tool
Cheese can be eaten
No melt() method
```

Both the Wax and Cheese classes own melt() methods, but there is no relation between the two. Thanks to duck typing, we can call the melt() method. Unfortunatelly, the Wood class is not equipped with this method, so an

AttributeError exception occurs.

Summary:

- polymorphism is used when different class objects share conceptually similar methods (but are not always inherited)
- polymorphism leverages clarity and expressiveness of the application design and development;
- when polymorphism is assumed, it is wise to handle exceptions that could pop up.