Наследование и Исключения

Видимость элементов в Наследовании:
1 - Поля объекта
2 - Поля класса
3 - Поля родительского Класса (Рекурсивно)

```
>>> class A:
...     F = 100500
...     def meth(self):
...             print('qq')
...
>>> class B(A):
...     pass
...
>>> dir(B)
['F', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'meth']
>>> B.__dict__
mappingproxy({'__module__': '__main__', '__doc__': None})
>>> # В самом В Полей и методов нет

>>> b = B()
>>> dir(b)
['F', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'meth']
>>> b.__dict__
{}

>>> class B(A):
...     C = 42
...
>>> b = B()
>>> dir(b)
['C', 'F', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'meth']
```

```
>>> b.__dict__
{}
>>> B.__dict__
mappingproxy({'__module__': '__main__', 'C': 42, '__doc__': None})
>>>
>>> b.F
100500
>>> B.C
42
>>> b.C
42
>>> b.meth()
qq
```

Тк всё логически является наслаиваемыми друг на друга пространствами имён, поля и методы базовых классов мы можем переопределять

```
>>> class B(A):
...     F = 42
...
>>> b = B()
>>> b.F
42
>>> b.__class__.F
42
>>> A.F
100500
```

```
>>> class A:
...     def __add__(self, other):
...             return 100500
...
```

```
>>> class B(A):
...     def __str__(self):
...             return "CLASS B"
...
```

```
>>> a = A()
>>> a + a
100500
>>> a + 214214
100500
>>> a + None
100500
>>>
>>> print(a)
<__main__.A object at 0x7f3cc92b53d0>

>>> b = B()
>>> print(b)
CLASS B
>>> type(b)
<class '__main__.B'>
>>> b + b
100500
```

С наследованием есть свои приколы. Рассмотрим такой класс, который умеет в сложение

```
>>> class A:
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         return self.val + other.val
...
>>> A(9) + A(10)
19
```

Получаем на выходе сложения не объект класса A, а число.Хотим получать объект, поэтому переделаем __add__, а заодно добавим формат вывода, чтобы явно отличать объекты

```
>>> class A:
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         return A(self.val + other.val)
...     def __str__(self):
...         return f'<{self.val}>'
...
>>>
>>> print(A(9) + A(10))
<19>
```

Так делать удобно ровно до момента наследования, потому что явно указали преобразование в класс A

```
>>> class B(A):
...     def __str__(self):
...         return f'<<<{self.val}>>>'
...
>>> print(B(9))
<<<9>>>
>>> print(B(9) + B(10))
<19>
```

Мы хотели бы, чтобы у нас возвращался объект того класса, в котором мы находимся. Для решения этой задачи придумано спецполе ***__class__***, точно знающее, кто ты по жизни: база или производка

```
>>> class A:
...     def __init__(self, val):
...             self.val = val
...     def __add__(self, other):
...             return self.__class__(self.val + other.val)
...     def __str__(self):
...             return f'<{self.val}>'
...
>>> class B(A):
...     def __str__(self):
...             return f'<<<{self.val}>>>'
...
>>> print(B(9) + B(10))
<<<19>>>
```

Вспоминаем метод type
type не просто выводит строку-информацию о классе; он возвращает объект, являющийся конструктором своего параметра - его классом

```
>>> b = B(9)
>>> type(b)
<class '__main__.B'>
>>>
>>>
>>> CB = type(b)
>>> CB
<class '__main__.B'>
>>> B
<class '__main__.B'>
>>> CB is B
True
>>> b.__class__
<class '__main__.B'>
```

Ещё один интересный вопрос про работу с базовыми классами
Вот мы хотим выводить иерархию классов, при этом чтобы каждый последующий класс возвращал всё с предыдущего + инфо о себе

```
>>> class A:
...     def fun(self):
...             return "A"
...
>>> class B(A):
...     def fun(self):
...             return A.fun(self) + "B"
...
>>> a = A()
>>> a.fun()
'A'
>>> print(a.fun())
A
>>> b = B()
>>> print(b.fun())
AB
```

Не нравится то же самое, что и с пунктом выше - Мы руками указываем базовый класс. Более того, мы жёстко зависим от родительских классов:

```
>>> del A
>>> b = B()
>>> b.fun()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in fun
NameError: name 'A' is not defined. Did you mean: 'a'?
```

Решение - НЕ добираться до родительского класса Вообще.
Вместо этого используется прокси-объект super(), он возвращает пространство имён, содержащее атрибуты родительского класса

```
>>> class A:
...     def fun(self):
...             return "A"
...
>>> class B(A):
...     def fun(self):
...             return super().fun() + "B"
...
>>> b = B()
>>> b.fun()
'AB'
```

При этом Суперу не нужен объект, в котором он применяется. Он, тк функция, просто смотрит, в каком контексте он запущен

В питончике из-за того, что мы можем создавать и менять при наследовании поля и методы как и где угодно, должна быть и есть защита от коллизий имён.
В частости, это Приватные атрибуты

```
>>> class A:
...     __F = 100500
...
>>> dir(A)
['_A__F', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__']
```

По сути, мы просто подменяем в системе имя переменной с просто имени с двумя подчёркиваниями на его же с добавлением в начале имени класса. Защиты, справедливости ради, ноль, просто вероятность случайно назвать переменную так же, тем более предумышленно - почти наверное ноль

```
>>> class A:
...     __F = 100500
...     def funA(self):
...         print(self.__F)
...
>>> class B(A):
...     __F = 42
...
>>> b = B()
>>> b.funA()
100500
>>> class B(A):
...     __F = 42
...     def funB(self):
...         print(self.__F, self._A__F)
...
>>> b = B()
>>> b.funB()
42 100500
>>> dir(B)
['_A__F', '_B__F', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'funA', 'funB']
```

Мы без проблем можем даже перебить значение

```
>>> class B(A):
...     __F = 42
...     _A__F = 43
...     def funB(self):
...             print(self.__F, self._A__F)
...
>>> b = B()
>>> b.funB()
42 43
```

Теперь поговорим про множественное наследование

```
>>> class A:
...     F = 100500
...
>>> class B:
...     G = 42
...
>>> class C(A, B): pass
...
>>> dir(C)
['F', 'G', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__']
>>> c = C()
>>> c.F
100500
>>> c.G
42
```

Тип графа наследования - Сеть. Отсюда появляется проблема Method Resolution Ordek (MRO)

Наследование у нас рекурсивно ищет “ближайшие” по иерархии методы и поля и работает с ними. А как понять, что ближайшее, в сети?

```
>>> class A:
...     def fun(self):
...         print('A')
...
>>> class B:
...     def fun(self):
...         print('B')
...
>>> class C(A, B): pass
...
>>> c = C()
>>> c.fun()
A
```

Тк мы указали родителей в каком-то порядке, мы задали тот самый MRO: Сначала ищется в A, потом в B; Где первее нашлось, то и используется

```
>>> class A:
...     def fun(self):
...         print("A")
...
>>> class B(A): pass
...
>>> class C(A):
...     def fun(self):
...         print("C")
...
>>> class D(B, C): pass
...
>>> d = D()
>>> d.fun()
C
```

Решение заданной проблемы - Линеаризация графа наследования и проход по списку классов
Надо соблюдать два свойства: Монотонность - соблюдение порядка наследования - и Соблюдение порядка объявления

- Монотонность: соблюдение порядка *наследования*
    - Если для класса ***class B(…, A, …)***: порядок поиска полей такой:
      B: [B, …, A, …]
    - ⇒ то для класса ***C(…, B, …)***: порядок должен быть таким:
      [C, …, B, …, A, …]
- Соблюдение порядка *объявления*:
    - ***class C(D, E):***

    - → ***[C, …, D, …, E, …]***
- Два разных порядка *могут конфликтовать*

```
>>> class A: pass
...
>>> class B(A): pass
...
>>> class Y(B, A): pass
...
>>> class X(A, B): pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, B
>>> Y.mro()
[<class '__main__.Y'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

mro() возвращает список классов, и этим можно и нужно пользоваться

super() Во множественном наследовании - Проходит по mro и возвращает первый пришедший метод

```
>>> class A:
...     def __str__(self):
...         return f"<{self.val}>"
...
>>> # A - абстрактный, у него нет поля val

>>> class B:
...     def __init__(self, val):
...         self.val = val
...
>>> class C(A, B):
...     pass
...
>>> c = C(123)
>>> print(c)
<123>
>>>
>>> class C(A, B):
...     def __init__(self, val):
...             v = f'[{val}]'
...             super().__init__(v)
...
>>> c = C(123)
>>> print(c)
<[123]>
>>> # Мы хз, в каком там родителе или выше был метод. А MRO может. И Супер может
```

Полиморфизм

Проверка принадлежности объекта какому-то классу осуществляется с помощью функции ***isinstance***

```
>>> class A: pass
...
>>> class B(A): pass
...
>>> a, b,  = A(), B()
>>> isinstance(a, A)
True
>>> isinstance(a, B)
False
>>> isinstance(b, A)
True
>>> isinstance(b, B)
True
```

Ещё есть такая приколюха, как Проксирование - Хранить родительский объект в виде поля, а метода нового класса делать обёрткой над методами родителя; Но это оч затратно, надо буквально перебить ВСЕ методы ручками, потому что __init__ в это не умеет

Исключения

Исключение - Объект Питона, который вбрасывается в поток вычислений и обрабатывается либо вами, либо Самим Питоном

Среди встроенных объектов Питона (поля объекта __builtins__) указаны все встроенные исключения

>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BaseExceptionGroup', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EncodingWarning', 'EnvironmentError', 'Exception', 'ExceptionGroup', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'aiter', 'all', 'anext', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

>>> raise ModuleNotFoundError("NOPE")
Traceback (most recent call last):
*** File "<stdin>", line 1, in <module>***
ModuleNotFoundError: NOPE

Блоки для обработки исключений и механизм исключений очень схожи с C++ (в конце концов, питон на плюсах и написан). Конструкция ***try-throw-catch*** меняется на ***try-raise-except***

```
>>> a, b = 1, 2
>>> try:
...     print(a/b)
... except ZeroDivisionError:
...     print("QQ")
...
0.5

>>> a, b = 1, 0
>>> try:
...     print(a/b)
... except ZeroDivisionError:
...     print("QQ")
...
QQ

>>> a, b = 1, "345"
>>> try:
...     print(a/b)
... except ZeroDivisionError:
...     print("QQ")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Такое исключение мы не обрабатывали. Чтобы отлавливать более обширные классы исключений (н-р, все), можно ловить базовые классы исключений, как в плюсах базовые будут ловить производные.

Все встроенные исключения наследуются от Exception

```
>>> a, b = 1, "345"
>>> try:
...     print(a/b)
... except Exception:
...     print("QQ")
...
QQ

>>>
>>> a, b = 1, 0
>>> try:
...     print(a/b)
... except Exception as E:
...     print(E, "Happens")
...
division by zero Happens

>>> a, b = 1, "345"
>>> try:
...     print(a/b)
... except Exception as E:
...     print(E, "Happens")
...
unsupported operand type(s) for /: 'int' and 'str' Happens
```

У исключений тоже есть доп.слова ***else*** и ещё не встречаемое нами ***finally***
else - Выполнится, если исключений не было
finally - Выполнится всегда, даже если исключение не перехвачено

```
>>> a, b = 1, "345"
>>> try:
...     print(a/b)
... except Exception as E:
...     print(E, "Happens")
... else:
...     print("QQ")
... finally:
...     print("Qwerty")
...
unsupported operand type(s) for /: 'int' and 'str' Happens
Qwerty

>>> a, b = 1, 2
>>> try:
...     print(a/b)
... except Exception as E:
...     print(E, "Happens")
... else:
...     print("QQ")
... finally:
...     print("Qwerty")
...
0.5
QQ
Qwerty
```