множинне наслідування: введення

Python - одна з небагатьох мов програмування, яка має в своєму арсеналі такий інструмент як **множинне наслідування** - тобто спроможність класу наслідувати не якийсь один клас, а будь-яку кількість базових класів. При цьому потенційно похідний клас набуває доступу до атрибутів всіх класів, від яких він наслідується.

```
>>> class A:
...     a_name = "from class A"
...
>>> class B:
...     b_name = "from class B"
...
>>> class C(A, B):
...     c_name = "from class C"
...
>>> c = C()
>>> c.a_name
'from class A'
>>> c.b_name
'from class B'
>>> c.c_name
'from class C'

>>>
```

В наведеному прикладі ми створюємо класи A і B з своїми атрибутами (які саме це атрибути - методи або властивості не має значення, так як це історія про те як шукаються імена і вона виглядає однаково для будь-яких атрибутів).
Клас C наслідується від A і B і має свій, визначений в тілі класу, атрибут. Як Ви бачите далі з прикладу - екземпляр класу C має доступ до атрибутів “свого” класу і всіх атрибутів батьківських класів.

Це виглядає як потужний інструмент (так і є), але в багатьох мовах програмування він не доданий, тому що вважається що множинне наслідування може створювати проблеми, вирішення яких може нівелювати переваги. Подумайте яку ви очікуєте поведінку якщо в класах A і B будуть атрибути з однаковими іменами? А тепер ускладнить ситуацію - і уявіть що ланцюжок наслідувань більш довгий і однакові імена зустрічаються “десь раніше”.
Розберемо на прикладі використання множинного наслідування.

множинне наслідування: завдання

Згадайте приклад коду з минулого матеріалу де ми створювали ієрархію класів співробітників компанії і додавали методи розрахунку платежів і складання платіжної відомості.
Уявимо, що з часом ми отримали нове завдання: у зв'язку з розвитком компанії і більш глибокою автоматизацією всіх процесів з'явилась нова посада - фахівець з бухгалтерського обліку який до того ж вміє писати код в межах спеціалізованої бухгалтерської платформи, яку використовує наша компанія.
Тобто: людина на цій посаді вміє розраховувати податки як бухгалтер і в той же час вміє супроводжувати і розвивати код.
Так як ця людина має відношення до бухгалтерії - вона працює в штаті компанії (не ФОП), і повинна ініціюватись як бухгалтер (без визначення мови програмування).

```
class AccountTechSupport(Programmer, Accountant):
    pass
```

Спочатку все здається доволі простим, враховуючи множинне наслідування: достатньо успадкувати клас **AccountTechSupport** від класів **Programmer** і **Account** - вони мають всі необхідні методи.

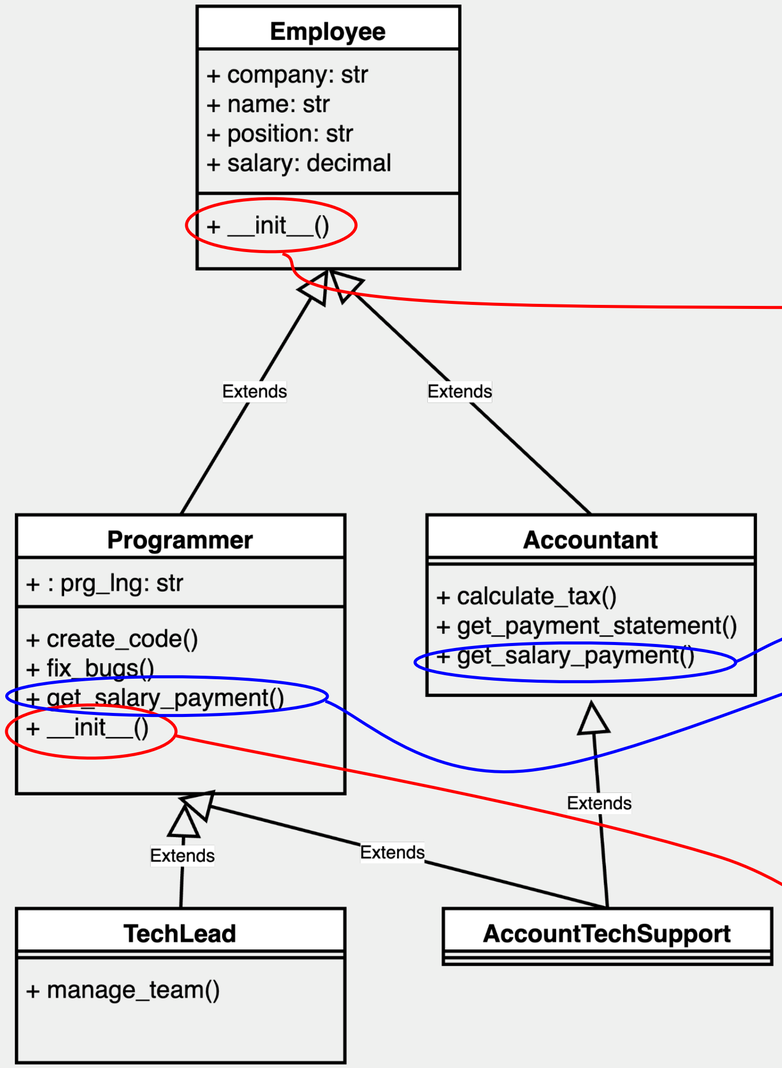
```
...
acc_support = AccountTechSupport("Ivan", 40000, "accountant support")
                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: Programmer.__init__() missing 1 required positional argument:
'prg_lng'
```

Але при намаганні створити екземпляр - виникає помилка. Клас очікує введення мови програмування (тобто використовується метод **__init__()** класу **Programmer**).

Ми можемо змінити черговість батьківських класів при визначенні класу (це змінює шлях пошуку атрибуту, а ми розуміємо що при ініціалізації екземпляру Python знаходить метод __init__() класу Programmer). Але це не допомагає - виникає та ж помилка.

```
class AccountTechSupport(Accountant, Programmer):
    pass
```

множинне наслідування: MRO



Тут графічно зображена побудована нами структура класів, доповнена усіма атрибутами які визначаються у відповідних класах.

Головною новиною для нас тут є клас **AccountantTechSupport**, який наслідує два класи і це створює два потенційних маршрути для пошуку відповідних атрибутів при зверненні до них. Це утворює потенціальну неоднозначність.

Подумайте - якщо екземпляр класу **AccountantTechSupport** буде звертатись до атрибуту **__init__()** - то яка з двох потенційно доступних реалізацій цього методу в дереві наслідування буде викликана? А якщо ми будемо звертатись до методу **get_salary_payment()** - який присутній і в **Accountant** і в **Programmer** - то яка з реалізацій буде викликана?

Для того щоб сформувати однозначний шлях пошуку Python використовує MRO - method resolution order - який використовує спеціальний алгоритм лінеаризації C3. В результаті застосування алгоритму створюється лінійний шлях пошуку - тобто список - який включає послідовність класів для пошуку атрибуту. При пошуку атрибуту Python спочатку дивиться простір імен екземпляру, якщо там атрибуту немає, то далі послідовно перебираються простори імен класі з цього списку і коли знайдено відповідний атрибут - він використовується.

множинне наслідування: керування пошуком

Щоб побачити послідовність пошуку, яка буде використана, необхідно звернутись до методу mro() класу.
Як Ви бачите послідовність починається з самого класу, закінчується класом object.
Помилка, яка виникала на етапі ініціалізації екземпляру класу **AccountTechSupport**, стає зрозумілою: по шляху пошуку першим зустрічається клас **Programer**, метод **__init__()** якого потребує параметр **prg_lng** (а не **Employee** - метод **__init__()** якого нам потрібен).

```
>>> AccountTechSupport.mro()
[
 <class 'inheritance_example_1.AccountTechSupport'>,
 <class 'inheritance_example_1.Accountant'>,
 <class 'inheritance_example_1.Programmer'>,
 <class 'inheritance_example_1.Employee'>,
 <class 'abc.ABC'>,
 <class 'object'>
]
```

```
class AccountTechSupport(Accountant, Programmer):
    def __init__(self, name, salary, position):
        super(Programmer, self).__init__(name, salary, position)
```

Якщо ми хочемо впливати на послідовність пошуку, ми можемо використати вже знайому нам функцію **super()**, але з аргументами: в якості першого аргументу передаємо клас з дерева наслідування, з суперкласу якого треба починати пошук (в нашому випадку передаємо **Programmer**, тоді пошук почнеться з **Employee** - як батьківського для **Programmer**). Другий аргумент - безпосередньо екземпляр.

Таким чином множинне наслідування може бути складним для використання зважаючи на дві причини:
- доволі “непрозорий” шлях пошуку атрибутів при декількох потенційних шляхах при множинному наслідуванні.
- “сильний зв’язок” між класами при наслідуванні - зміни в класах в дереві наслідування можуть непрямо зважати на похідні класи. Це може бути вирішено використанням композиції класів замість успадкування.

Незважаючи на наведені складнощі множинне успадкування є потужним інструментом і активно використовується. Далі ми познайомимось з прийомом множинного успадкування який знімає наведені складнощі.

множинне наслідування: класи “домішки”

В межах нашої задачі ми можемо виділити одну сутність - **Employee**, два набора скілів - **Programming** і **Accounting**, і дві політики взаємодії з співробітником стосовно взаємовідносин - **ContractWorker** (ФОП) і **OfficeWorker** (в штаті). Ми можемо визначити наступні класи(для зручності згруповані в окремих модулях):

```python
# module hr.py
class Employee:
    company = "Web factory"

    def __init__(self, name, salary, position):
        self.name = name
        self.salary = salary
        self.position = position

    def __str__(self):
        return self.name
```

```python
# module cooperation.py
class ContractWorker:
    def get_salary_payment(self):
        return f"{(self.salary / 0.95):.2f} UAH"


class OfficeWorker:
    def get_salary_payment(self):
        return f"{self.salary} UAH"
```

```python
# module skills.py
class Programming:
    def create_code(self):
        return f"Programmer {self} writes code"

    def fix_bug(self):
        return f"Programmer {self} fixes bug"


class Accounting:
    def calculate_tax(self):
        return f"Accountant {self} calculates tax"

    def get_payment_statement(self, employees):
        statement = f"pay by company '{self.company}': \n"
        for employee in employees:
            statement += (f"{employee.name:>20}, "
                          f"{employee.position:>20} - gets "
                          f"{employee.get_salary_payment():<15}\n")
        return statement
```

множинне наслідування: класи “домішки”

```
import hr, skills, cooperation


class Programmer(
    hr.Employee,
    skills.Programming,
    cooperation.ContractWorker
):
    def __init__(self, name, salary, position, prg_lng):
        super().__init__(name, salary, position)
        self.prg_lng = prg_lng


class Accountant(
    hr.Employee,
    skills.Accounting,
    cooperation.OfficeWorker
):
    pass


class AccountantTechSupport(
    hr.Employee,
    skills.Accounting,
    skills.Programming,
    cooperation.OfficeWorker
):
    pass
```

В окремому модулі, імпортувавши все необхідне, ми можемо “зібрати” необхідних нам співробітників з відповідними скілами і відповідними відносинами з класів, які і називають “класи-домішки”, **mixin-класи**.

Кожен з визначених на попередньому кроці класів (**mixin-класів**) визначає атрибути притаманні лише йому і не пов’язаний з іншими.

Ми збираємо необхідні нам для роботи з конкретними співробітниками класи немов з блоків конструктора - набираючи необхідний набор скілів і обираючи тип взаємовідносин.

При необхідності додати в якийсь клас додаткові атрибути (наприклад клас **Programmer**, атрибут екземпляра **prg_lng**) - ми перевизначаємо в тілі класу відповідний метод.