

Технологии программирования на Python



7. ООП



python

ОСНОВНЫЕ ПРИЧИНЫ ИСПОЛЬЗОВАТЬ ООП:

- Объект может служить пространством имен и хранилищем для данных
- Объект может обладать заданным поведением
- Программа может легче читаться, если будет состоять из взаимодействующих объектов, отражающий реальный мир
- Использование шаблонов проектирования
- Наследование помогает переиспользовать код

Объектно-ориентированное программирование (ООП) — это подход, при котором программа рассматривается как набор объектов, взаимодействующих друг с другом. У каждого есть свойства и поведение. Если постараться объяснить простыми словами, то ООП ускоряет написание кода и делает его более читаемым.

Структура ООП

Объекты и классы

Класс — это «шаблон» для объекта, который описывает его свойства.

Объект — это набор переменных и функций, как в традиционном функциональном программировании. Переменные и функции и есть его свойства.

Атрибуты и методы

• **Атрибуты** — это переменные, конкретные характеристики объекта, такие как цвет поля или имя пользователя.

• **Методы** — это функции, которые описаны внутри объекта или класса. Они относятся к определенному объекту и позволяют взаимодействовать с ними или другими частями кода.

Принципы ООП

Объектно-ориентированное программирование определяют через четыре принципа, по которым можно понять основы работы. Иногда количество сокращают до трех — опускают понятие абстракции.

Абстракция

Абстрагирование — это способ выделить набор наиболее важных атрибутов и методов и исключить незначимые.

Инкапсуляция

Каждый объект — независимая структура. Все, что ему нужно для работы, уже есть у него внутри. Если он пользуется какой-то переменной, она будет описана в теле объекта, а не снаружи в коде. Это делает объекты более гибкими. Даже если внешний код перепишут, логика работы не изменится.

Наследование

Можно создавать классы и объекты, которые похожи друг на друга, но немного отличаются — имеют дополнительные атрибуты и методы. Более общее понятие в таком случае становится «родителем», а более специфичное и подробное — «наследником».

Полиморфизм

Одинаковые методы разных объектов могут выполнять задачи разными способами.

Классы в Python

Как известно, в python всё является объектами - и строки, и списки, и словари, и всё остальное.

Но возможности ООП в python этим не ограничены. При необходимости, программист может написать свой тип данных (класс), определить в нём свои методы и атрибуты.

Создание класса и объекта

```
class A:  
    pass
```

Теперь, когда он определен в коде, мы можем создать несколько экземпляров этого класса. Запишем их в переменные `a` и `b`:

```
a = A()  
b = A()
```

Атрибуты класса

Этим объектам можно задать атрибуты — персональные для каждого объекта переменные. Создадим для каждого из них атрибут `value`:

```
a.value = 1  
b.value = 2
```

```
c = A()  
print(c.value)  
# Traceback (most recent call last):  
#   File "<stdin>", line 1, in <module>  
# AttributeError: 'A' object has no attribute 'value'
```

Важно понимать, что изначально **класс A** ничего в себе не содержал, мы добавили атрибуты `value` в процессе работы с его экземплярами. При этом, в самом классе этот атрибут не появился, так как **class A** — это всего лишь шаблон.

Методы класса

Методы класса — это персональные функции класса. Все методы должны определяться внутри класса и принимать аргумент, который принято называть *self*.

self — обязательный аргумент, содержащий в себе экземпляр класса, автоматически передающийся при вызове метода у объекта. По сути, это тоже самое, что и *this* в C#.

```
class A:  
    value = 1  
  
    def method(self):  
        print(self.value)
```

```
a = A()  
a.method() # 1  
a.value = 5  
a.method() # 5
```

Метод также можно вызвать не только у экземпляра, но и у самого объекта класса A, но тогда аргумент *self*, т.е. экземпляр, придется передавать вручную:

```
a = A()  
a.value += 1  
A.method(a) # 2
```


Конструктор класса

Чтобы не приходилось каждый раз задавать значение атрибута после создания экземпляра класса, это можно сделать сразу же при создании экземпляра, при помощи **конструктора класса**.

Конструктор класса — метод для создания объекта класса. Когда мы создавали объекты класса A, мы использовали конструктор по умолчанию, который не принимает параметров и который неявно имеют все классы.

Для реализации конструктора, в классе нужно определить метод с названием `__init__`:

```
class A:  
    def __init__(self, value):  
        self.value = value
```

```
a = A(5)  
b = A(6)  
print(a.value) # 5  
print(b.value) # 6
```

Атрибуты класса

```
class A:  
    value = 1
```

```
a = A()  
a.value = 5
```

```
print(a.value) # 5  
print(A.value) # 1
```

```
b = A()  
  
print(a.value) # 5  
print(A.value) # 1  
print(b.value) # 1
```

Классы, которые вообще не подразумевают создание экземпляров, а просто содержат в себе какие-то общие методы и атрибуты, также называются **статическими**.

Методы класса

У класса можно также определить метод, который будет иметь отношение не к конкретному экземпляру, а только к самому классу.

Такой метод помечается встроенным декоратором `@classmethod` и должен принимать в качестве аргумента объект класса, а не экземпляра. Этот аргумент принято называть `cls`, сокращенно от `class`.

```
class A:  
    value = 1  
  
    @classmethod  
    def class_method(cls):  
        print(cls.value)
```

Статические методы

Статические методы ничего не знают о классе, в котором они содержатся.

Такой метод помечается декоратором `@staticmethod` и не принимает никаких обязательных аргументов.

Статический метод также можно вызвать и у класса и у экземпляра. Смысл создания статических методов в том, чтобы объединить в одном классе все однотипные функции.

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width
        self.area = self.make_area(length, width)

    @staticmethod
    def make_area(length, width):
        return length * width

rect = Rectangle(5, 10)
print(rect.area) # 50
print(Rectangle.make_area(2, 3)) # 6
```

Интроспекция

Многие языки программирования поддерживают интроспекцию, и Python не является исключением. В общем, в контексте объектно-ориентированных языков программирования, **интроспекция** — это способность объекта во время выполнения получить тип, доступные атрибуты и методы, а также другую информацию, необходимую для выполнения дополнительных операций с объектом.

dir()

Встроенная функция **dir()** предоставляет список атрибутов и методов, доступных для указанного объекта, который может быть объявленной переменной или функцией.

```
>>> dir([1,2])
['_add_', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'_format_', '_ge_', '_getattribute_', '_getitem_', '_getstate_', '_gt_', '_hash_', '_ia_', '_init_', '_init_subclass_',
'_iter_', '_le_', '_len_', '_lt_', '_mul_', '_ne_', '_or_', '_reduce_ex_', '_repr_', '_reversed_', '_rmul_', '_setattr_', '_setitem_', '_sizeof_',
'_subclasshook_', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse']
```

При вызове функции **dir()** без аргумента она возвращает имена, доступные в локальной области видимости.

```
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> dir()
['_annotations_', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>>
```

help()

```
>>> help([1,2].sort)
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
    order of two equal elements is maintained).

    If a key function is given, apply it once to each list item and sort them,
    ascending or descending, according to their function values.

    The reverse flag can be set to sort in descending order.
```

.__doc__

```
>>> [1,2].sort.__doc__
'Sort the list in ascending order and return None.\n\nThe sort is in-place (i.e. the list itself is modified) and stable (i.e. the\norder of two equal elements is maintained).\n\nIf a key function is given,\napply it once to each list item and sort them,\nascending or descending, according to their function values.\n\nThe reverse flag can be set to sort in descending order.'
>>>
```

```
def get_full_help(obj):  
    for name in dir(obj):  
        attr = getattr(obj, name)  
        if callable(attr) and "_" not in name:  
            doc = attr.__doc__  
            small_help = doc.split('\n')[0]  
            print(f'{name:<15}{small_help}')
```

```
get_full_help('abc')
```

```

>>> get_full_help('abc')
capitalize    Return a capitalized version of the string.
casefold      Return a version of the string suitable for caseless comparisons.
center        Return a centered string of length width.
count         S.count(sub[, start[, end]]) -> int
encode        Encode the string using the codec registered for encoding.
endswith      S.endswith(suffix[, start[, end]]) -> bool
expandtabs    Return a copy where all tab characters are expanded using spaces.
find          S.find(sub[, start[, end]]) -> int
format        S.format(*args, **kwargs) -> str
index         S.index(sub[, start[, end]]) -> int
isalnum       Return True if the string is an alpha-numeric string, False otherwise.
isalpha       Return True if the string is an alphabetic string, False otherwise.
isascii       Return True if all characters in the string are ASCII, False otherwise.
isdecimal     Return True if the string is a decimal string, False otherwise.
isdigit       Return True if the string is a digit string, False otherwise.
isidentifier  Return True if the string is a valid Python identifier, False otherwise.
islower       Return True if the string is a lowercase string, False otherwise.
isnumeric     Return True if the string is a numeric string, False otherwise.
isprintable   Return True if the string is printable, False otherwise.
isspace       Return True if the string is a whitespace string, False otherwise.
istitle       Return True if the string is a title-cased string, False otherwise.
isupper       Return True if the string is an uppercase string, False otherwise.
join          Concatenate any number of strings.
ljust         Return a left-justified string of length width.
lower         Return a copy of the string converted to lowercase.
lstrip        Return a copy of the string with leading whitespace removed.
maketrans     Return a translation table usable for str.translate().
partition     Partition the string into three parts using the given separator.
removeprefix  Return a str with the given prefix string removed if present.
removesuffix  Return a str with the given suffix string removed if present.
replace       Return a copy with all occurrences of substring old replaced by new.
rfind         S.rfind(sub[, start[, end]]) -> int
rindex        S.rindex(sub[, start[, end]]) -> int
rjust         Return a right-justified string of length width.
rpartition    Partition the string into three parts using the given separator.
rsplit        Return a list of the substrings in the string, using sep as the separator string.
rstrip        Return a copy of the string with trailing whitespace removed.
split         Return a list of the substrings in the string, using sep as the separator string.
splitlines    Return a list of the lines in the string, breaking at line boundaries.
startswith    S.startswith(prefix[, start[, end]]) -> bool
strip         Return a copy of the string with leading and trailing whitespace removed.
swapcase      Convert uppercase characters to lowercase and lowercase characters to uppercase.
title         Return a version of the string where each word is titlecased.
translate     Replace each character in the string using the given translation table.
upper         Return a copy of the string converted to uppercase.
zfill         Pad a numeric string with zeros on the left, to fill a field of the given width.
>>>

```


type()

Другой часто используемой функцией интроспекции является функция `type()`. Как видно из названия, эта функция возвращает тип объекта, который может быть примитивным типом данных, объектом, классом или модулем.

isinstance()

Еще одной из функций интроспекции, которая особенно полезна, является функция `isinstance()`. Используя эту функцию, мы можем определить, является ли определенный объект экземпляром указанного класса.

`hasattr()`, `delattr()`, `setattr()`, `getattr()`

`hasattr()` — имеет ли объект указанный атрибут

`delattr()` — удалить у объекта указанный атрибут

`setattr()` — задать объекту указанный атрибут

`getattr()` — получить значение указанного атрибута

Инкапсуляция. Модификаторы доступа

Модификаторы доступа используются для модификации области видимости переменных по умолчанию. Есть три типа модификаторов доступов в ООП:

- публичный — public
- приватный — private
- защищенный — protected

Доступ к переменным с модификаторами **публичного** доступа открыт из любой точки вне класса, доступ к **приватным** переменным открыт только внутри класса, и в случае с **защищенными** переменными, доступ открыт только в внутри класса и его наследниках.

Public

Это самые обычные переменная и функция внутри класса.

Protected

Защищенные атрибуты и методы в Python защищены только условно. К ним также можно обращаться где угодно, но их смысл в том, чтобы они использовались внутри класса или внутри наследуемых от него классов.

Защищенные атрибуты в Python принято обозначать нижним подчеркиванием перед названием атрибута. Это даст другим программистам понять, что с такой переменной не стоит работать напрямую и скорее всего она используется где-то внутри методов класса.

```
class A:  
    _attribute = 1  
  
    def _method(self):  
        return 1
```

Private

Приватные атрибуты и методы не наследуются в производных классах и используются только внутри конкретного класса.

Приватные атрибуты в Python нужно обозначать двойным нижним подчеркиванием __ перед названием атрибута.

```
class A:
    __attribute = 1

    def __method(self):
        return 1
```

Но вот получить к ним доступ за пределами класса уже не получится

```
a = A()
print(a.__attribute) # AttributeError: 'A' object has no attribute '__attribute'
print(a.__method()) # AttributeError: 'A' object has no attribute '__method'
```

Это происходит потому что Python изменяет названия этих атрибутов при хранении объекта. Если применить к классу функцию `dir()`, то можно увидеть, что приватные атрибуты хранятся в классе с видоизмененным названием. И при большом желании к ним все таки можно получить доступ, но делать это крайне не рекомендуется.

```
a = A()
print(a._A__attribute) # 1
print(a._A__method)    # 1
```

Инкапсуляция. Свойства (Properties)

По умолчанию атрибуты в классах являются общедоступными, а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его.

Касательно инкапсуляции непосредственно в языке программирования Python скрыть атрибуты класса можно сделав их **приватными** и ограничив доступ к ним через специальные методы, которые еще называются **свойствами**.

```
class Person:
    def __init__(self, name, age):
        self.__name = name # устанавливаем имя
        self.__age = age   # устанавливаем возраст

    def set_age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустимый возраст")

    def get_age(self):
        return self.__age

    def get_name(self):
        return self.__name

    def display_info(self):
        print(f"Имя: {self.__name}\tВозраст: {self.__age}")
```

Декораторы свойств

Для создания свойства-геттера над свойством ставится декоратор **@property**.

Для создания свойства-сеттера над свойством устанавливается декоратор

@имя_свойства_геттера.setter.

```
class Person:
    def __init__(self, name, age):
        self.__name = name # устанавливаем имя
        self.__age = age   # устанавливаем возраст

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустимый возраст")

    @property
    def name(self):
        return self.__name

    def display_info(self):
        print(f'Имя: {self.__name}\tВозраст: {self.__age}')
```

Во-первых, стоит обратить внимание, что свойство-сеттер определяется после свойства-геттера.

Во-вторых, и сеттер, и геттер называются одинаково - age. И поскольку геттер называется age, то над сеттером устанавливается аннотация @age.setter.

После этого, что к геттеру, что к сеттеру, мы обращаемся через выражение person.age.

Полиморфизм. Перегрузки операторов

Можно переписать логику сложения. Например, можно дополнительно предусмотреть создание вектора с одинаковыми координатами, если была выполнена попытка сложения вектора с числом

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, vector):
        if isinstance(vector, (int, float)):
            vector = Vector(vector, vector)
        return Vector(self.x + vector.x, self.y +
vector.y)
```


Другие операторы

Помимо оператора сложения можно перегрузить и другие операторы. Для них тоже есть соответствующие методы:

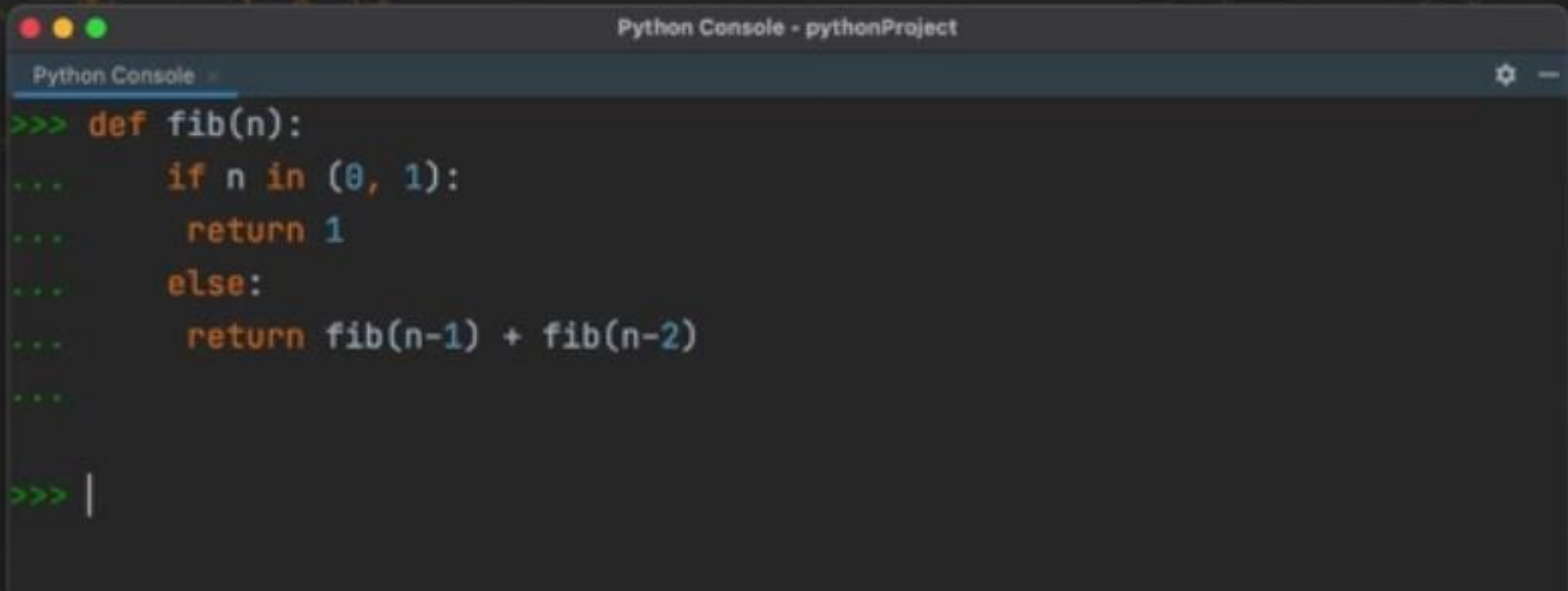
`__sub__`: Вычитание — $x - y$
`__mul__`: Умножение — $x * y$
`__truediv__`: Деление — x / y
`__floordiv__`: Целочисленное деление — $x // y$
`__mod__`: Остаток от деления — $x \% y$
`__pow__`: Возведение в степень — $x ** y$

`__lt__`: Меньше — $x < y$
`__le__`: Меньше или равно — $x \leq y$
`__gt__`: Больше — $x > y$
`__ge__`: Больше или равно — $x \geq y$
`__eq__`: Равно — $x == y$
`__ne__`: Не равно — $x != y$

`__len__`: длина объекта — `len()`
`__bool__`: приведение к типу `bool` — `bool()`
`__iter__`: итератор объекта — `iter()`

Мы можем превратить любой объект сделать функционально-подобным, благодаря методу `__call__`.

`__call__`



```
Python Console - pythonProject

Python Console
>>> def fib(n):
...     if n in (0, 1):
...         return 1
...     else:
...         return fib(n-1) + fib(n-2)
...
>>> |
```

```
>>> class Fib:
...     cache = {}
...     def fib(self, n):
...         if n not in self.cache:
...             if n in (0, 1):
...                 res = 1
...             else:
...                 res = self.fib(n-1) + self.fib(n-2)
...                 self.cache[n] = res
...         return self.cache[n]
...     def __call__(self, n):
...         return self.fib(n)
```

```
>>> f = Fib()
... print(f(10))
...
```

Наследование

Наследование позволяет создавать новый класс на основе уже существующего класса. Наряду с инкапсуляцией наследование является одним из краеугольных камней объектно-ориентированного программирования.

```
class Person:
    def __init__(self, name):
        self.__name = name # имя человека

    @property
    def name(self):
        return self.__name

    def print_info(self):
        print(f'Имя: {self.name}')
```

```
class Employee(Person):
    def work(self):
        print(f'{self.name} начал работу')

employee = Employee('Иван')
print(employee.name) # Иван
employee.print_info() # Имя: Иван
employee.work()      # Иван начал работу
```

На самом деле, даже пустой класс А тоже неявно наследуется от базового типа `object`, т.к. в Python все является объектом.

Множественное наследование

Одной из отличительных особенностей языка Python является поддержка множественного наследования.

```
# класс работника
class Employee(Person):
    def work(self):
        print(f'{self.name} начал работу')

# класс студента
class Student(Person):
    def study(self):
        print(f'{self.name} начал обучение')

# класс работающего студента
# наследование от классов Employee и Student
class WorkingStudent(Employee, Student):
    def __str__(self):
        return f'<{self.name} — работающий студент. Возраст: {self.age}>'

working_student = WorkingStudent('Иван', 19)
working_student.work() # Иван начал работу
working_student.study() # Иван начал обучение
print(working_student) # <Иван — работающий студент. Возраст: 19>
```

Множественное наследование

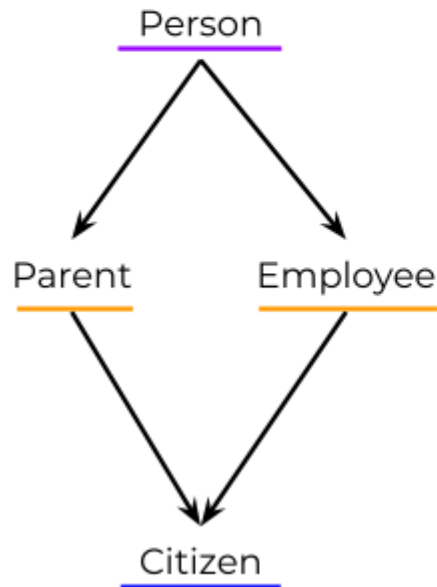
Одна из самых спорных вещей в ООП: стоит ли это использовать?

Плюс: позволяет сокращать затраты на разработку класса и избегать повторного использования кода.

Минус: повышает сложность создания и модификации системы классов. Увеличивает связь между классами, а значит, изменения в базовом классе могут повлечь серьёзные проблемы в дочерних.

Итого: обычно не используют, но есть ситуации, когда оно необходимо.

Diamond problem

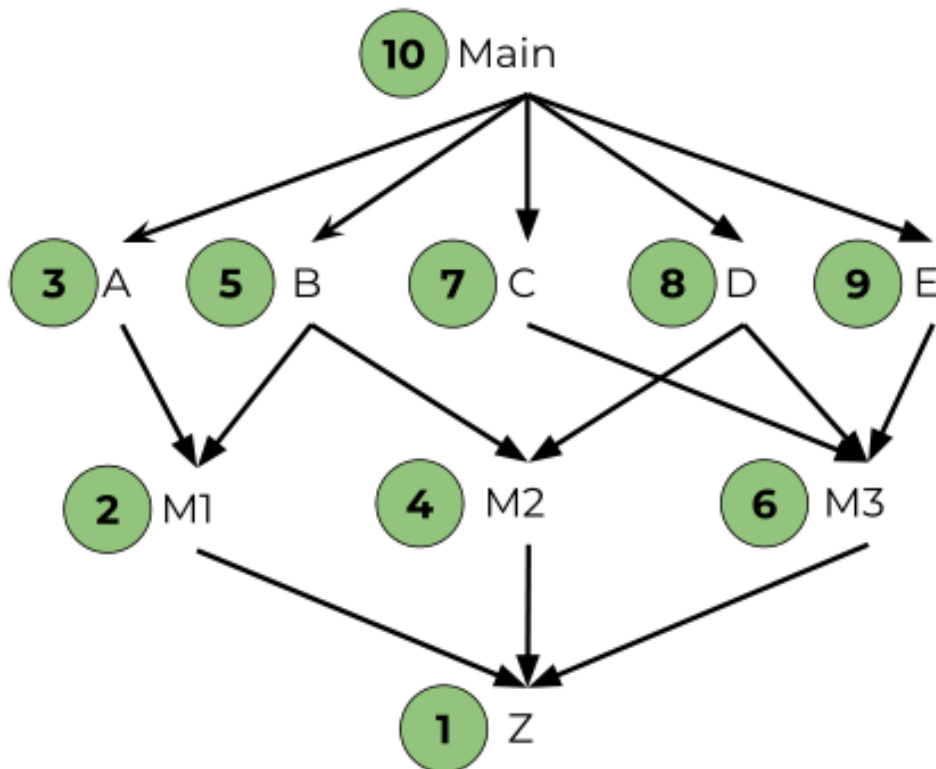


Method Resolution Order (MRO) —
порядок разрешения методов.

1. Citizen
2. Parent
3. Employee
4. Person

Diamond problem

```
class Main: ...  
  
class A(Main): ...  
class B(Main): ...  
class C(Main): ...  
class D(Main): ...  
class E(Main): ...  
  
class M1(A, B): ...  
class M2(B, D): ...  
class M3(C, D, E): ...  
  
class Z(M1, M2, M3): ...
```



~~MRO: Z > M1 > M2 > M3 > A > B > C > ...~~

MRO: Z -> M1 -> A -> M2 -> B -> M3 -> C -> D -> E -> Main

Правила алгоритма C3-линеаризации

- Сохранять локальный порядок методов: если класс А указан перед классом В, все методы класса А должны рассматриваться перед методами класса В.
- Поддерживать порядок в родительских классах: если класс А является родителем класса В, то все методы класса А должны рассматриваться перед методами класса В.
- Учитывать порядок наследования: если класс С является родителем для двух или более классов, порядок методов класса С должен сохраняться в MRO всех этих классов.

Абстрактные классы

```
from abc import ABC, abstractmethod

class Figure(ABC):
    """
    Абстрактный базовый класс Фигура

    Args and attrs:
        pos_x (int): координата X
        pos_y (int): координата Y
        length (int): длина фигуры
        width (int): ширина фигуры
    """
    def __init__(self, pos_x: int, pos_y: int, length: int, width: int) -> None:
        self.pos_x = pos_x
        self.pos_y = pos_y
        self.length = length
        self.width = width

    @abstractmethod
    def move(self, pos_x: int, pos_y: int) -> None:
        self.pos_x = pos_x
        self.pos_y = pos_y
```

Абстрактный метод обязательно должен быть переопределён в дочернем классе!

