# УРОК 33. УГЛУБЛЕННАЯ РАБОТА С КЛАССАМИ И ВСТРОЕННЫЕ ДЕКОРАТОРЫ


# МЕТОДЫ КЛАССА И СТАТИЧЕСКИЕ МЕТОДЫ

Методы класса определяются с помощью декоратора @classmethod и принимают первым аргументом класс, а не экземпляр класса. Они позволяют работать с атрибутами класса, вызывать другие методы класса и выполнять другие операции, связанные с классом.

```python
class MyClass:
  class_attr = "Class attribute"

  @classmethod
  def class_method(cls):
    print(cls.class_attr)

MyClass.class_method() # Class attribute
```

Статические методы определяются с помощью декоратора @staticmethod и не принимают дополнительных аргументов (класс или экземпляр класса). Они являются простыми функциями внутри класса и используются для группировки связанных операций, которые не требуют доступа к атрибутам класса или экземпляра.

```python
class MyClass:
  @staticmethod
  def static_method():
    print("Static method")

MyClass.static_method() # Static method
```

# ЗАДАНИЕ ДЛЯ ЗАКРЕПЛЕНИЯ

Объясните, что происходит в этом фрагменте кода и какой будет вывод:

```python
class Person:
  age = 25

  def printAge(cls):
    print('The age is:', cls.age)

Person.printAge = classmethod(Person.printAge)
Person.printAge()
```

# ЗАЩИЩЕННЫЕ И ПРИВАТНЫЕ АТРИБУТЫ

В Python существует соглашение об использовании нотации с одинарным подчеркиванием _ для обозначения защищенных атрибутов и с двойным подчеркиванием __ для обозначения приватных атрибутов.

Защищенные атрибуты предназначены для внутреннего использования внутри класса и его наследников. Хотя доступ к этим атрибутам не ограничен, их использование снаружи класса считается не рекомендуемым.

Приватные атрибуты предназначены для полной инкапсуляции и не предназначены для доступа извне класса. Имена приватных атрибутов автоматически изменяются с добавлением имени класса в начало, чтобы предотвратить их переопределение или случайное использование в подклассах.

```python
class MyClass:
  def __init__(self):
    self._protected_attr = "Protected attribute"
    self.__private_attr = "Private attribute"

obj = MyClass()
print(obj._protected_attr) # Protected attribute
print(obj._MyClass__private_attr) # Private attribute
```

# ЗАДАНИЕ ДЛЯ ЗАКРЕПЛЕНИЯ

Какая аналогия может быть у этих картинок и внутренних и внешних интерфейсов?


# ПОЛИМОРФИЗМ И НАСЛЕДОВАНИЕ

Полиморфизм в Python позволяет объектам различных классов иметь одинаковый интерфейс, то есть поддерживать одни и те же методы или операции. Это позволяет обрабатывать объекты разных классов с использованием общих функций и методов, что упрощает и унифицирует код.

Python

```python
class Shape:
  def area(self):
    raise NotImplementedError("Method 'area' must be implemented")

class Rectangle(Shape):
  def __init__(self, width, height):
    self.width = width
    self.height = height

  def area(self):
    return self.width * self.height

class Circle(Shape):
  def __init__(self, radius):
    self.radius = radius

  def area(self):
    return 3.14 * self.radius ** 2

shapes = [Rectangle(5, 10), Circle(3)]
for shape in shapes:
  print(shape.area())
```

Наследование позволяет создавать новые классы на основе уже существующих. Класс, от которого наследуется новый класс, называется базовым классом или родительским классом, а класс, который наследует базовый класс, называется производным классом или дочерним классом.


Проблема ромбовидного наследования возникает, когда производный класс наследует от нескольких классов, которые имеют общего предка. Для разрешения этой проблемы используется MRO (Method Resolution Order) - порядок разрешения методов.

```python
class A:
  def method(self):
    print("A method")

class B(A):
  def method(self):
    print("B method")

class C(A):
  def method(self):
    print("C method")

class D(B, C):
  pass

obj = D()
obj.method() # B method
```

Для создания пользовательского исключения необходимо определить новый класс, наследующийся от класса Exception или его подклассов.

```python
class CustomException(Exception):
  pass

try:
  raise CustomException("This is a custom exception")
except CustomException as e:
  print(e) # This is a custom exception
```

# ЗАДАНИЕ ДЛЯ ЗАКРЕПЛЕНИЯ

Как вы объясните фрагмент, который представлен ниже, в рамках объектно-ориентированного подхода?

```python
1 + 1
2
"1" + "1"
'11'
```

# ВЫЗОВ КОНСТРУКТОРА ТЕКУЩЕГО КЛАССА

Иногда требуется вызвать конструктор текущего класса вместо конструктора родительского класса. Это может быть полезно, например, при создании подкласса с расширенным функционалом, но с сохранением базового состояния.

super() используется для вызова методов родительского класса. Он позволяет обращаться к функциональности родительского класса без необходимости указывать его имя явно. super() применяется не только в конструкторах, но и в любых других методах класса.

Python

```python
class MyBaseClass:
  def __init__(self, x):
    self.x = x

class MySubClass(MyBaseClass):
  def __init__(self, x, y):
    super().__init__(x)
    self.y = y

obj = MySubClass(1, 2)
print(obj.x) # 1
print(obj.y) # 2
```

# ЗАДАНИЕ ДЛЯ ЗАКРЕПЛЕНИЯ

Объясните, что происходит в этом фрагменте кода и какой будет вывод:

Python

```python
class Animal:
  def __init__(self, name):
    self.name = name
  def make_sound(self):
    print("The animal makes a sound")
class Dog(Animal):
  def __init__(self, name):
    super().__init__(name)
  def make_sound(self):
    super().make_sound()
    print("The dog barks")
my_dog = Dog("Buddy")
my_dog.make_sound()
```

# АБСТРАКТНЫЕ КЛАССЫ

Абстрактные классы в Python предоставляют механизм для определения интерфейсов, которые должны быть реализованы в производных классах. Абстрактные классы не могут быть инициализированы напрямую, они служат только для наследования.

Модуль abc (Abstract Base Classes) предоставляет базовый класс ABC и декораторы для определения абстрактных методов и свойств.

```python
from abc import ABC, abstractmethod

class AbstractClass(ABC):
  @abstractmethod
  def method(self):
    pass

class ConcreteClass(AbstractClass):
  def method(self):
    print("ConcreteClass method")

obj = ConcreteClass()
obj.method() # ConcreteClass method
```

# ПРАКТИЧЕСКАЯ РАБОТА

Создать систему наследования для геометрических фигур: фигура, прямоугольник, квадрат, круг (можно добавить и другие плоские фигуры - например, параллелограмм, треугольник, прямоугольный треугольник).

У всего определить площадь и периметр, а также __str__ (дополнительно можно добавить __repr__).

Важно: где-то нужно переопределять методы, а площадь абстрактной фигуры не определена.

# ПОЛЕЗНЫЕ МАТЕРИАЛЫ

1. Python
2. Python | Наследование