Методические указания

Урок 15.1. Основы ООП

Задачи урока:

- Переопределение методов
- Перегрузка операторов

0. Подготовка к уроку

До начала урока преподавателю необходимо:

1) Просмотреть, как ученики справились с домашним заданием
2) Прочитать методичку

1. Переопределение методов

Учитель: На этом занятии мы рассмотрим интересные возможности, которые предоставляет нам ООП. Начнем с переопределения методов. Что это такое и для чего нужно. Представим ситуацию, что у нас есть родительский и дочерние классы, а в каждом из них есть метод с одинаковым названием. Какой метод вызовется при обращении через экземпляр дочернего класса? Правильный ответ - метод из дочернего класса. Это и есть так называемое переопределение методов. Когда мы наследуемся от какого то класса и изменяем поведение метода так, как на требуется

Рассмотрим пример

```
class Parent:
    def say_hello():
        print('Привет я метод родительского класса')

class Children(Parent):
    def say_hello():
        print('Привет я метод дочернего класса')


child = Children
child.say_hello()
```

Результат

Привет я метод дочернего класса

Таким образом мы переопределили данный метод. Данный пример конечно же достаточно примитивен. Давайте рассмотрим более интересный вариант. Скажите а какой метод добавляет в список новый элемент? Правильно метод append. А давайте переопределим его Для этого мы будем наследоваться от класса списка(list)

```
class Test(list):
    def append(self, object) -> None:
        for i in range(len(self)):
            self[i] **= object


a = Test([1, 2, 3])
print(a)
a.append(2)
print(a)
```

Теперь наш метод не добавляет элемент в конец списка, а возводит каждый элемент списка в квадрат.

2. Перегрузка операторов

Учитель: Перегрузка операторов — один из способов реализации полиморфизма, когда мы можем задать свою реализацию какого-либо метода в своём классе.
Полиморфизм – это способность одного и того же объекта вести себя по-разному в зависимости от того, в контексте какого класса он используется.

Для начала список методов с двойным подчеркиванием, в которых мы можем изменить поведение

```
__new__(cls[, ...]) — управляет созданием экземпляра. В качестве
обязательного аргумента принимает класс (не путать с экземпляром). Должен
возвращать экземпляр класса для его последующей передачи методу __init__.
__init__(self[, ...]) - конструктор.
__del__(self) - вызывается при удалении объекта сборщиком мусора.
__repr__(self) - вызывается встроенной функцией repr; возвращает "сырые"
данные, использующиеся для внутреннего представления в python.
__str__(self) - вызывается функциями str, print и format. Возвращает
строковое представление объекта.
__format__(self, format_spec) - используется функцией format (а также методом
format у строк).
__lt__(self, other) - x < y .
__le__(self, other) - x ≤ y.
__eq__(self, other) - x == y.
__ne__(self, other) - x != y
__gt__(self, other) - x > y.
__ge__(self, other) - x >= y.
```

__hash__(self) - получение хэш-суммы объекта, например, для добавления в словарь.
__bool__(self) - вызывается при проверке истинности. Если этот метод не определён, вызывается метод __len__ (объекты, имеющие ненулевую длину, считаются истинными).
__getattr__(self, name) - вызывается, когда атрибут экземпляра класса не найден в обычных местах (например, у экземпляра нет метода с таким названием).
__setattr__(self, name, value) - назначение атрибута.
__delattr__(self, name) - удаление атрибута (del obj.name).
__call__(self[, args...]) - вызов экземпляра класса как функции.
__len__(self) - длина объекта.
__getitem__(self, key) - доступ по индексу (или ключу).
__setitem__(self, key, value) - назначение элемента по индексу.
__delitem__(self, key) - удаление элемента по индексу.
__iter__(self) - возвращает итератор для контейнера.
__reversed__(self) - итератор из элементов, следующих в обратном порядке.
__contains__(self, item) - проверка на принадлежность элемента контейнеру (item in self).

Теперь рассмотрим таблицу для перегрузки математических операторов
__add__(self, other) - сложение. x + y.
__sub__(self, other) - вычитание (x - y).
__mul__(self, other) - умножение (x * y).
__truediv__(self, other) - деление (x / y).
__floordiv__(self, other) - целочисленное деление (x // y).
__mod__(self, other) - остаток от деления (x % y).
__divmod__(self, other) - частное и остаток (divmod(x, y)).
__pow__(self, other[, modulo]) - возведение в степень (x ** y, pow(x, y[, modulo])).
__iadd__(self, other) - +=.
__isub__(self, other) - -=.
__imul__(self, other) - *=.
__itruediv__(self, other) - /=.
__ifloordiv__(self, other) - //=.
__imod__(self, other) - %=.
__ipow__(self, other[, modulo]) - **=.

Это не полная таблица, но для примера нам подойдет. Давайте попробуем заставить + например умножать

```
class Test(int):
    def __init__(self, num) -> None:
        super().__init__()
        self.num  = num
    def __add__(self, num2):
        return self.num * num2


a = Test(5)
print(a + 10)
```

Рассмотрим еще один пример переопределения методов на примере точки

```
class Point2D:
  """Точка на плоскости."""

  # Поле класса (доступна без создания экземпляра)
  # Хранит количество экземпляров класса и является общей (!)
  # для всех объектов этого класса
  instances_count = 0

  def __init__(self, x, y):
    self.x = x
    self.y = y

    # При инициализации нового класса увеличиваем количество
    # созданных экземпляров
    Point2D.instances_count += 1

  def __str__(self):
    """Вернуть строку в виде 'Точка 2D (x, y)'."""
    return 'Точка 2D ({}, {})'.format(self.x, self.y)

  def __add__(self, other):
    """Сложить self и other.

    Параметры:
      - other (Point2D): вернуть новый объект-сумму;
      - other (int, float): сдвинуть точку на other по x и y;
      - other (другой тип): возбудить исключение TypeError.
    """
    if isinstance(other, self.__class__):
      # Точка с точкой
      # Возвращаем новый объект!
      return Point2D(self.x + other.x, self.y + other.y)
    elif isinstance(other, (int, float)):
      # Точка и число
      # Добавим к обеим координатам self число other и вернем результат
      # Возвращаем старый, измененный, объект!
      self.x += other
      self.y += other
      return self
    else:
      # В противном случае сгенерируем исключение
      raise TypeError("Не могу добавить {1} к {0}".
              format(self.__class__, type(other)))

  def __sub__(self, other):
    """Создать новый объект как разность координат self и other."""
    return Point2D(self.x - other.x, self.y - other.y)

  def __neg__(self):
    """Вернуть новый объект, инвертировав координаты."""
    return Point2D(-self.x, -self.y)

  def __eq__(self, other):
    """Вернуть ответ, являются ли точки одинаковыми."""
```

```
        return self.x == other.x and self.y == other.y

    def __ne__(self, other):
        """Вернуть ответ, являются ли точки разными.

        Используем реализованную операцию ==."""
        return not (self == other)

    @staticmethod
    def sum(*points):
        """Вернуть сумму точек 'points' как новый объект.

        Статический метод: принадлежит классу, но ничего о нем не знает.
        """
        assert len(points) > 0, "Количество суммируемых точек = 0!"

        res = points[0]
        for point in points[1:]:
            res += point

        return res

    @classmethod
    def from_string(cls, str_value):
        """Создать экземпляр класса из строки 'str_value'.

        Классовый метод, доступен для вызова как:
            Point2D.from_string(...)

        Параметры:
            - cls: ссылка на класс (Point2D);
            - str_value: строка вида "float, float".

        Результат:
            - Экземпляр класса cls (Point2D).
        """
        values = [float(x) for x in str_value.split(',')]
        assert len(values) == 2

        return cls(*values)

if __name__ == "__main__":

    p1 = Point2D(0, 5)
    p2 = Point2D(-5, 10)

    # Создаем 3-ю точку через метод класса
    p3 = Point2D.from_string("5, 6")

    print(p1 + p3)  # Точка 2D (5.0, 11.0)

    # Отображаем количество созданных точек через переменную класса
    print(Point2D.instances_count)  # 4 (p1, p2, p3, p1 + p2)

    # Сложение точек через статический метод
    p4 = Point2D.sum(p1, p2, p3, Point2D(0, -21))
```

```
print(p4)  # Точка 2D (0.0, 0.0)
```

3. Решение задач

Задача 1

Переопределить метод len для нахождения длины списка/строки, чтобы он выводил всегда число 25

```
class Test(str):
    def __init__(self, word) -> None:
        super().__init__()
    def __len__(self) -> int:
        return 25


a = Test('hello')
print(len(a))
```

Дополнительно

Если на уроке остается время, то ученикам можно предложить начать прорешивать домашнее задание.

Домашняя работа

Задача 1

Реализовать класс и переопределить магические методы базовых математических операции(сложение, вычитание, умножение, деление), добавив туда выводы в консоль текущего действия. Например: при умножении выводится сообщение, что происходит умножение.