# Министерство образования Республики Беларусь

УПРАВЛЕНИЕ ОБРАЗОВАНИЯ МОГИЛЕВСКОГО ОБЛАСТНОГО ИСПОЛНИТЕЛЬНОГО КОМИТЕТА

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«Могилевский государственный политехнический колледж»

# Домашняя контрольная работа №1

# По дисциплине: «Инструментальное программное обеспечение»

Группа ПО-455

## Выполнил О.Е. Листопадова

Шифр 21

# 2021

**27 Опишите особенности наследования классов в языке Python.**

Наследование в объектно-ориентированном программировании очень похоже на наследование в реальной жизни, где ребенок наследует те или иные характеристики его родителей в дополнение к его собственным характеристикам.

В объектно-ориентированном программировании, наследование означает отношение IS-A. Например, болид — это транспорт. Наследование это одна из самых удивительных концепций объектно-ориентированного программирования, так как оно подразумевает повторное использование.

Основная идея наследования в объектно-ориентированном программировании заключается в том, что класс может наследовать характеристики другого класса. Класс, который наследует другой класс, называется дочерним классом или производным классом, и класс, который дает наследие, называется родительским, или основным.

Рассмотрим простой пример наследования. Выполним следующий скрипт:

# Создание класса Vehicle

class Vehicle:

def vehicle\_method(self):

print("Это родительский метод из класса Vehicle")

# Создание класса Car, который наследует Vehicle

class Car(Vehicle):

def car\_method(self):

print("Это метод из дочернего класса")

В скрипте создаем два класса: Vehicle и Car, которые наследуют класс Vehicle. Чтобы наследовать класс, нужно только вписать название родительского класса внутри скобок, которая следует за названием дочернего класса. Класс Vehicle содержит метод vehicle\_method(), а дочерний класс содержит метод car\_method(). Однако, так как класс Car наследует класс Vehicle, он также наследует и метод vehicle\_method().

Рассмотрим это на практике и выполним следующий скрипт:

car\_a = Car()

car\_a.vehicle\_method() # Вызываем метод родительского класса

car\_a = Car()

car\_a.vehicle\_method() # Вызываем метод родительского класса

В этом скрипте мы создали объект класса Car вызывали метод vehicle\_method() при помощи объекта класса Car. Можно обратить внимание на то, что класс Car не содержит ни одного метода vehicle\_method(), но так как он унаследовал класс Vehicle, который содержит vehicle\_method(), класс Car также будет использовать его. Выдача выглядит следующим образом:

Это родительский метод из класса Vehicle

Это родительский метод из класса Vehicle

**Множественное наследование Python**

В Python, родительский класс может иметь несколько дочерних, и, аналогично, дочерний класс может иметь несколько родительских классов. Давайте рассмотрим первый сценарий. Выполним следующий скрипт:

# создаем класс Vehicle

class Vehicle:

def vehicle\_method(self):

print("Это родительский метод из класса Vehicle")

# создаем класс Car, который наследует Vehicle

class Car(Vehicle):

def car\_method(self):

print("Это дочерний метод из класса Car")

# создаем класс Cycle, который наследует Vehicle

class Cycle(Vehicle):

def cycleMethod(self):

print("Это дочерний метод из класса Cycle")

В этом скрипте, родительский класс Vehicle наследуется двумя дочерними классами — Car и Cycle. Оба дочерних класса будут иметь доступ к vehicle\_method() родительского класса.

car\_a = Car()

car\_a.vehicle\_method() # вызов метода родительского класса

car\_b = Cycle()

car\_b.vehicle\_method() # вызов метода родительского класса

В выдаче видна выдача метода vehicle\_method() дважды, как показано ниже:

Это родительский метод из класса Vehicle

Это родительский метод из класса Vehicle

Видно, как родительский класс наследуется двумя дочерними классами. Таким же образом, дочерний класс может иметь несколько родительских. Посмотрим на пример:

Python

class Camera:

def camera\_method(self):

print("Это родительский метод из класса Camera")

class Radio:

def radio\_method(self):

print("Это родительский метод из класса Radio")

class CellPhone(Camera, Radio):

def cell\_phone\_method(self):

print("Это дочерний метод из класса CellPhone")

В скрипте выше мы создали три класса: Camera, Radio, и CellPhone. Классы Camera и Radio наследуются классом CellPhone. Это значит, что класс CellPhone будет иметь доступ к методам классов Camera и Radio. Следующий скрипт подтверждает это:

cell\_phone\_a = CellPhone()

cell\_phone\_a.camera\_method()

cell\_phone\_a.radio\_method()

Выдача будет выглядеть следующим образом:

Это родительский метод из класса Camera

Это родительский метод из класса Radio

**Полиморфизм**

Термин полиморфизм буквально означает наличие нескольких форм. В контексте объектно-ориентированного программирования, полиморфизм означает способность объекта вести себя по-разному.

Полиморфизм в программировании реализуется через перегрузку метода, либо через его переопределение.

*Перегрузка метода*

Перегрузка метода относится к свойству метода вести себя по-разному, в зависимости от количества или типа параметров. Взглянем на очень простой пример перегрузки метода. Выполним следующий скрипт:

# создаем класс Car

class Car:

def start(self, a, b=None):

if b is not None:

print (a + b)

else:

print (a)

В скрипте выше, если метод start() вызывается передачей одного аргумента, параметр будет выведен на экран. Однако, если мы передадим 2 аргумента методу start(), он внесет оба аргумента и выведет результат суммы.

Попробуем с одним аргументом для начала:

car\_a = Car()

car\_a.start(10)

В выдаче мы можем видеть 10. Теперь попробуем передать два аргумента:

car\_a.start(10, 20)

В выдаче 30.

*Переопределение метода*

Переопределение метода относится к наличию метода с одинаковым названием в дочернем и родительском классах. Определение метода отличается в родительском и дочернем классах, но название остается тем же. Простой пример переопределения метода в Python.

# создание класса Vehicle

class Vehicle:

def print\_details(self):

print("Это родительский метод из класса Vehicle")

# создание класса, который наследует Vehicle

class Car(Vehicle):

def print\_details(self):

print("Это дочерний метод из класса Car")

# создание класса Cycle, который наследует Vehicle

class Cycle(Vehicle):

def print\_details(self):

print("Это дочерний метод из класса Cycle")

В скрипте выше, классы Cycle и Car наследуют класс Vehicle. Класс Vehicle содержит метод print\_details(), который переопределен дочерним классом. Теперь, если вызвать метод print\_details(), выдача будет зависеть от объекта, через который вызывается метод. Можно выполнить следующий скрипт, чтобы понять суть на деле:

car\_a = Vehicle()

car\_a. print\_details()

car\_b = Car()

car\_b.print\_details()

car\_c = Cycle()

car\_c.print\_details()

Это родительский метод из класса Vehicle

Это дочерний метод из класса Car

Это дочерний метод из класса Cycle

Как видно, выдача отличается, к тому же метод print\_details() вызывается через производные классы одного и того же базового класса. Однако, так как дочерние классы переопределены методом родительского класса, методы ведут себя по-разному.

**49 Перечислите операции над множествами.**

## Проверка на равенство

>>> set([1, 2, 3, 2, 1]) == {3, 1, 2}

True

Можно подумать, что два множества равны, если каждый отдельный элемент одного множества содержится и во втором, и догадка была бы близка к истине. Но коллекции в Python хранят только ссылки на объекты. Так что же, множества равны, если ссылаются на одни и те же объекты? Одинаковые ссылки "равны" — это правда. Но могут быть и равны разные объекты!

Дело в том, что в Python есть специальный протокол проверки на равенство. Большинство встроенных типов данных поддерживает этот протокол. Можно проверять на равенство числа, строки, булевы значения. А ещё можно приравнивать кортежи, списки, словари. И вот тут Python поступает очень разумно: если приравнять две коллекции одного типа, то эти коллекции будут считаться равными, если их элементы попарно равны (с точки зрения протокола, опять же).

Пример:

>>> [1, 2, ["foo", "bar"]] == [1, 2, ["foo"] + ["bar"]]

True

>>> (1, True, []) == (1, True, [])

True

>>> {"a": 1, "b": 2} == {"b": 2, "a": 1}

True

Словари равны, если порядок ключей разный — лишь бы были равны значения по соответствующим ключам и сами наборы ключей были одинаковыми.

Вот и множества равны, если содержат одинаковые наборы равных попарно элементов.

## Объединение множеств

Множества в Python, по аналогии с множествами в математике, поддерживают операцию объединения (union). Эта операция не объединяет множества, как могло бы показаться, но возвращает такое множество — новый объект — которое содержит все элементы, содержащиеся хотя бы в одном из оригинальных множеств. По смыслу объединение похоже на операцию "ИЛИ" из булевой логики: элемент будет присутствовать в объединении, если он присутствует в первом исходном множестве ИЛИ во втором. Для объединения множеств в Python используется оператор |:

>>> visited\_by\_masha = {'Paris', 'London'}

>>> visited\_by\_kolya = {'Moscow', 'Paris'}

>>> visited\_by\_kolya | visited\_by\_masha

{'London', 'Moscow', 'Paris'}

## Пересечение множеств

Раз есть операция, похожая на "ИЛИ", то логично было бы предположить, что есть и "операция И". Да, есть и такая — пересечение (intersection) множеств. В пересечение входят элементы, присутствующие в первом из оригинальных множеств И во втором. Оператор пересечения — &. Пример:

>>> visited\_by\_masha = {'Paris', 'London'}

>>> visited\_by\_kolya = {'Moscow', 'Paris'}

>>> visited\_by\_kolya & visited\_by\_masha

{'Paris'}

## Разность множеств

Разность (difference) множеств — такое множество, элементы которого содержатся в первом оригинальном множестве, но не содержатся во втором. Разность представлена оператором - (уж очень смысл похож на вычитание из арифметики). Пример:

>>> visited\_by\_masha = {'Paris', 'London'}

>>> visited\_by\_kolya = {'Moscow', 'Paris'}

>>> visited\_by\_masha - visited\_by\_kolya

{'London'}

>>> visited\_by\_kolya - visited\_by\_masha

{'Moscow'}

## Симметрическая разность

Симметрическая разность (symmetric difference) — множество, в которое входят элементы, присутствующие ЛИБО в первом, ЛИБО во втором оригинальном множестве. По смыслу операция похожа на исключающее ИЛИ (xor), поэтому и представлена оператором ^. Пример:

>>> visited\_by\_masha = {'Paris', 'London'}

>>> visited\_by\_kolya = {'Moscow', 'Paris'}

>>> visited\_by\_kolya ^ visited\_by\_masha

{'London', 'Moscow'}

## Подмножества и надмножества

Одно множество является подмножеством (subset) другого, если все элементы первого входят во второе, но второе может содержать ещё и другие элементы. Второе в этом случае является надмножеством (superset) для первого. Такое соотношение множеств можно проверить с помощью методов issubset и issuperset:

>>> a = {1, 2, 3, 4}

>>> b = {3, 4}

>>> b.issubset(a)

True

>>> a.issuperset(b)

True

**62 Напишите программу, которая определяет максимальное число из введенной с клавиатуры последовательности положительных чисел (длина последовательности неограниченна).**

Блок-схема алгоритма работы программы представлена на рисунке 1

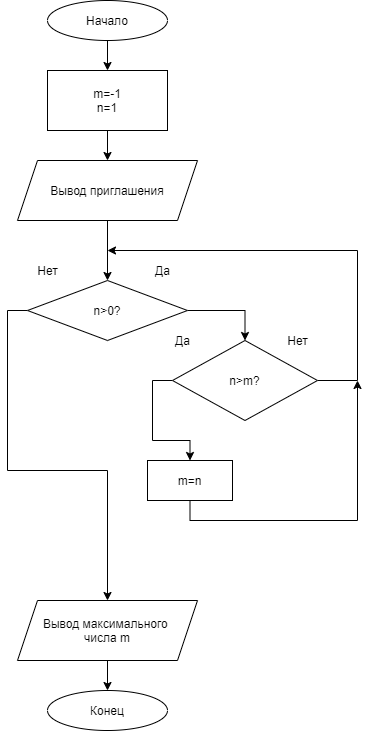


Рисунок 1 – Блок схема алгоритма работы программы.

**Текст программы**

# 62 Напишите программу, которая определяет максимальное число

# из введенной с клавиатуры последовательности положительных чисел

# (длина последовательности неограниченна).

m=-1

n=1

print('Для окончания введите 0')

while n >0: # в цикле считываем значения

n=int(input())

if n>m:

m=n

print('Работа завершена. Максимальное число - ',m)

**84 Решите задачу. Дан двумерный массив. Заполните его по строкам с клавиатуры. Отсортируйте каждый столбец по возрастанию.**

Блок-схема алгоритма работы программы представлена на рисунке 2

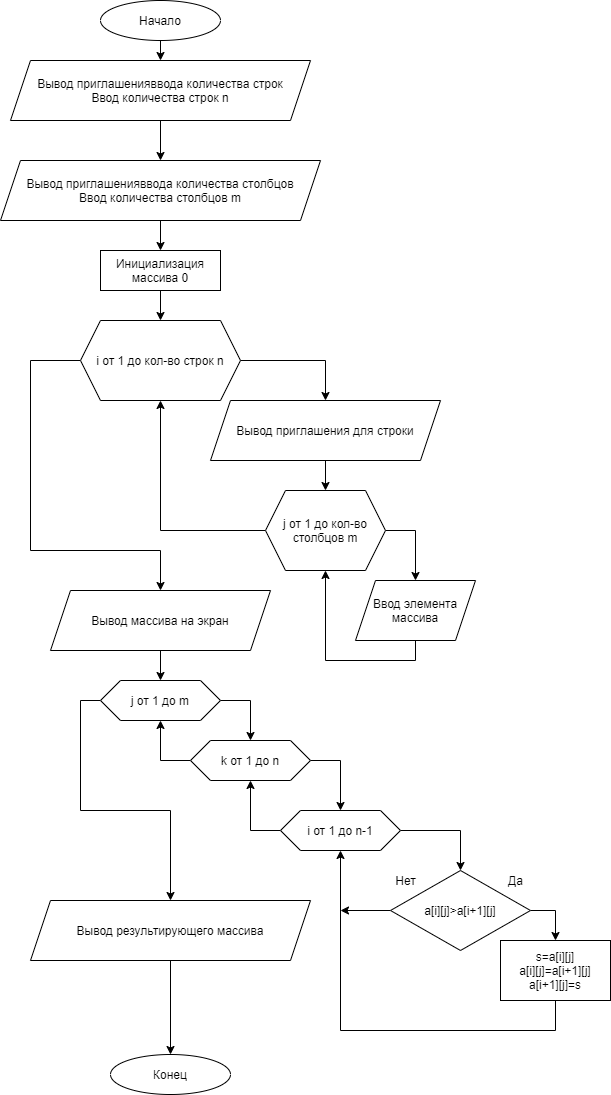


Рисунок 2 – Блок схема алгоритма работы программы

**Текст программы**

# 84 Решите задачу. Дан двумерный массив. Заполните его

# по строкам с клавиатуры. Отсортируйте каждый столбец по возрастанию.

print('Введите количество строк N') # выводим на экран приглашение

n=int(input())

print('Введите количество столбцов M') # выводим на экран приглашение

m=int(input())

a=[[0] \* m for i in range(n)]

for i in range(n):

print('Вводите строку ',i+1)

for j in range(m):

a[i][j]=int(input())

print(a) # выводим массив на экран

# сортируем каждый столбец

for j in range(m): # перебираем столбцы

for k in range(n): # повторяем несколько проходов

for i in range(n-1):# перебираем элементы столбца сравнивая 2 соседних

if a[i][j]>a[i+1][j]: # если не отсортировано

s=a[i][j] # меняем элементы местами

a[i][j]=a[i+1][j]

a[i+1][j]=s

print(a) # выводим массив на экран

**Список использованных источников**

1 Прохоренок Н. А. Python 3. Самое необходимое / Н. А. Прохоренок, В. А. Дронов. — СПб. : БХВ-Петербург, 2016. — 464 с.: ил

2 Плас, Дж. В. Python для сложных задач: наука о данных и машинное обучение / Дж. В. Плас. — СПб.: Питер, 2018. — 576 с.: ил.

3 Саммерфилд, М. Программирование на Python 3. Подробное руководство / М. Саммерфилд. - М.: Символ, 2016. - 608 c.