**Содержание**

1. **Понятия и определения парадигм программирования. Объективно-ориентированное программирование**
2. **Понятие абстракции, инкапсуляции, наследования и полиморфизма как ключевых принципов объективно-ориентированного программирования**
3. **Структурные элементы: классы, объекты, методы**
4. **Реализация программы «Крестики-нолики»**
5. **Описание этапов создания программы «Крестики-нолики»**

**Список использованной литературы**

1. **Понятия и определения парадигм программирования. Объективно-ориентированное программирование**

**Парадигма программирования** – этосовокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Парадигма программирования не определяется однозначно языком программирования; практически все современные языки программирования в той или иной мере допускают использование различных парадигм.

Парадигма разработки – это набор правил и критериев, соблюдаемых разработчиками, чтобы выдержать конкретную стилистику и модель написания кода.

Единая парадигма помогает избегать ошибок, упрощает работу в команде и ускоряет разработку. Ориентируясь на одну парадигму, можно корректно структурировать код приложения, зная четкие правила, выбранные командой, которая работает над конкретным проектом.

Существуют различные типы парадигм, например процедурный, ориентированный на работу с функциями, или логический, подразумевающий решение базовых логических задач. Но есть и более интересный подход к решению задач разработки, и это – объектно-ориентированная парадигма.  Также существующие парадигмы очень часто пересекаются друг с другом в деталях (например, модульное и объектно-ориентированное программирование), поэтому можно встретить ситуации, когда разные авторы употребляют названия из разных парадигм, говоря при этом, по сути, об одном и том же явлении.

Диомидис Спинеллис даёт следующее определение. Слово «парадигма» используется в программировании для определения семейства обозначений (нотаций), разделяющих общий способ (методику) реализаций программ.

В статье Дэниела Боброва парадигма определяется как «стиль программирования как описания намерений программиста». Брюс Шрайвер определяет парадигму программирования как «модель или подход к решению проблемы», Линда Фридман – как «подход к решению проблем программирования».

Памела Зейв даёт определение парадигмы как «способа размышления о компьютерных системах».

Питер Вегнер предлагает другой подход к определению термина парадигмы программирования. В его работе «Concepts and paradigms of object-oriented programming» парадигмы определяются как «правила классификации языков программирования в соответствии с некоторыми условиями, которые могут быть проверены».

Тимоти Бадд предлагает понимать термин «парадигма» как «способ концептуализации того, что значит „производить вычисления“, и как задачи, подлежащие решению на компьютере, должны быть структурированы и организованы».

Парадигма программирования как исходная концептуальная схема постановки проблем и их решения является инструментом грамматического описания фактов, событий, явлений и процессов, возможно, не существующих одновременно, но интуитивно объединяемых в общее понятие.

Программирование возникло и развивалось как процедурное программирование, которое предполагает, что основой программы является алгоритм, процедура обработки данных. Объектно-ориентированное программирование – это методика разработки программ, в основе которой лежит понятие объекта как некоторой структуры, описывающей объект реального мира, его поведение. Задача, решаемая с использованием методики объектно-ориентированного программирования, описывается в терминах объектов и операций над ними, а программа при таком подходе представляет собой набор объектов и связей между ними. Другими словами, можно сказать, что объектно-ориентированное программирование представляет собой метод программирования, который весьма близко напоминает наше поведение. Оно является естественной эволюцией более ранних нововведений в разработке языков программирования. Объектно-ориентированное программирование является более структурным, чем все предыдущие разработки, касающиеся структурного программирования. Оно также является более модульным и более абстрактным, чем предыдущие попытки абстрагирования данных и переноса деталей программирования на внутренний уровень.

Язык программирования Python поддерживает несколько парадигм программирования: структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное.

**Объектно-ориентированное программирование (ООП)**

Объектно-ориентированное программирование – это одна из парадигм разработки, подразумевающая организацию программного кода, ориентируясь на данные и объекты, а не на функции и логические структуры.

Крупные проекты часто реализуют, используя объектно-ориентированное программирование. Как мы знаем, в реальном мире любой объект как-либо взаимодействует с окружающим миром, будь то целенаправленное действие или просто предоставление информации о себе. И конечно же, это взаимодействие программисты перенесли в виртуальный мир.

Объектно-ориентированное программирование заставляет разработчиков фокусироваться на объектах, которыми нужно манипулировать, а не на той логике, что позволяет изменять данные и как-то с ними взаимодействовать. Такой подход хорошо работает в случае с комплексными программными решениями, требующими постоянной поддержки со стороны большого числа программистов.

Идеологически объектно-ориентированное программирование – подход к программированию как к моделированию информационных объектов, решающий на новом уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости, что существенно улучшает управляемость самим процессом моделирования, что, в свою очередь, особенно важно при реализации крупных проектов.

Управляемость для иерархических систем предполагает минимизацию избыточности данных и их целостность, поэтому созданное удобно управляемым – будет и удобно пониматься. Таким образом, через тактическую задачу управляемости решается стратегическая задача — транслировать понимание задачи программистом в наиболее удобную для дальнейшего использования форму.

Объектно-ориентированное программирование – это методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого объекта, а классы образуют иерархию наследования.

1. **Понятие абстракции, инкапсуляции, наследования и полиморфизма как ключевых принципов объективно-ориентированного программирования**

Основные принципы структурирования в случае ООП связаны с различными аспектами базового понимания предметной задачи, которое требуется для оптимального управления соответствующей моделью:

1. **абстракция** для выделения в моделируемом предмете важного для решения конкретной задачи по предмету, в конечном счёте — контекстное понимание предмета, формализуемое в виде класса;
2. **инкапсуляция** для быстрой и безопасной организации собственно иерархической управляемости: чтобы было достаточно простой команды «что делать», без одновременного уточнения как именно делать, так как это уже другой уровень управления;
3. **наследование** для быстрой и безопасной организации родственных понятий: чтобы было достаточно на каждом иерархическом шаге учитывать только изменения, не дублируя всё остальное, учтённое на предыдущих шагах;
4. **полиморфизм** для определения точки, в которой единое управление лучше распараллелить или наоборот — собрать воедино.

Обычный человеческий язык в целом отражает идеологию объектно-ориентированного программирования, начиная с инкапсуляции представления о предмете в виде его имени и заканчивая полиморфизмом использования слова в переносном смысле, что в итоге развивает выражение представления через имя предмета до полноценного понятия-класса.

**Абстракция данных**

Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой. В объектно-ориентированном программировании рассматривают лишь абстракцию данных (нередко называя её просто «абстракцией»), подразумевая набор наиболее значимых характеристик объекта, доступных остальной программе.

Каждый верхний слой над объектом (классы) более абстрактный, чем его «младшая версия». Это позволяет не переписывать по 10 раз один и тот же объект, указывая одни и те же атрибуты и методы. Напротив, абстрактные классы позволяют создавать все более конкретные классы и вытекающие из них объекты, не описывая реализацию функций заранее (в этом и суть абстракции), а оставляя исключительно базовый шаблон для дальнейших надстроек.

**Инкапсуляция**

Инкапсуляция – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе. Одни языки отождествляют инкапсуляцию с сокрытием данных, как в Python, но некоторые языки различают эти понятия.

В конкретном языке Python мы лишь имитируем сокрытие атрибутов. В итоге все соглашение здесь сводится к тому, что наличие двух подчеркиваний перед атрибутом должно сигнализировать, что трогать его вне класса не стоит, даже через такой способ, разве что при крайней необходимости.

Таким образом, инкапсуляция обеспечивается следующими средствами:

**А) Контроль доступа**

Поскольку методы класса могут быть как чисто внутренними, обеспечивающими логику функционирования объекта, так и внешними, с помощью которых взаимодействуют объекты, необходимо обеспечить скрытость первых при доступности извне вторых. Для этого в языки вводятся специальные синтаксические конструкции, явно задающие область видимости каждого члена класса.

Традиционно это модификаторы public, protected и private, обозначающие, соответственно, открытые члены класса, члены класса, доступные внутри класса и из классов-потомков, и скрытые, доступные только внутри класса.

**Б) Методы доступа**

Поля класса в общем случае не должны быть доступны извне, поскольку такой доступ позволил бы произвольным образом менять внутреннее состояние объектов. Поэтому поля обычно объявляются скрытыми (либо язык в принципе не позволяет обращаться к полям класса извне), а для доступа к находящимся в полях данным используются специальные методы, называемые методами доступа. Такие методы либо возвращают значение того или иного поля, либо производят запись в это поле нового значения. При записи метод доступа может проконтролировать допустимость записываемого значения и, при необходимости, произвести другие манипуляции с данными объекта, чтобы они остались корректными (внутренне согласованными). Методы доступа называют ещё аксессорами, а по отдельности — геттерами и сеттерами.

Этот принцип гласит, что вся важная информация, необходимая для работы объекта, в нем же и хранится. И только определенные данные доступны для внешних функций и объектов.

Данные конкретного объекта или класса хранятся в пределах этого объекта или класса. Вносить в них изменения, используя другие классы, нельзя. У окружения есть право только запрашивать «публичные» методы и атрибуты. Такой подход обеспечивает повышенный уровень безопасности, а также сокращает шансы на случайное повреждение данных внутри какого-то класса или объекта со стороны.

Таким образом, чтобы защитить атрибуты класса от случайных внешних изменений в Python используется сокрытие данных. Для этого перед атрибутом ставится двойное подчеркивание. Сокрытие данных входит в определение инкапсуляции. Оно является одним из основных принципов объективно-ориентированного программирования.

Двойное подчеркивание перед именем – это лишь соглашение между программистами, это не синтаксическое правило языка и при желании его можно нарушить. Согласно этому соглашению, если поле или метод имеют два знака подчеркивания перед именем, но нет после, то это атрибут или метод разработан исключительно для внутреннего использования.

Таким образом, приватное значение атрибутов можно изменять и получать только через вызовы геттеров и сеттеров. Это позволяет защитить данные от ненужных изменений.

Инкапсуляция – это механизм языка, который позволяет объединить данные методы, работающие с этими данными в единый объект и скрыть детали реализации от пользователя. Чтобы изменять или получать значения приватных атрибутов, используются специальные методы, которые называются сеттеры и геттеры. В программировании серьезных классов они используются довольно часто. В Python сокрытие данных лишь имитируется. При желании со значением приватного атрибута можно работать напрямую, но так делать крайне не рекомендуется. Приватными могут быть не только атрибуты, но и методы класса, если этот метод используется как некая шестеренка для полноценной работы класса.

**Наследование**

В программировании, как и в реальной жизни у разных объектов бывают общие действия или атрибуты. И без грамотного использования объектно-ориентированного программирования в таком случае получится не очень чистый код.

Еще один принцип объектно-ориентированного программирования называется наследование. При наследовании дочерний класс принимает весь функционал и атрибуты родительского класса и при этом может добавлять еще что-то свое. Таким образом, наследование применяется там, где можно выделить общие свойства и/или поведение объектов класса.

Для создания нового класса на основе существующего в объектно-ориентированном программировании используется принцип наследования классов. Для этого в скобках указывается класс, у которого мы наследуем. Класс, от которого мы наследуем, называется базовым, или также родительским классом, или же суперклассом. Наследника правильно называть подклассом, наследником, производным или дочерним классом. При наследовании дочерний класс получает весь функционал от родительского класса, т.е. атрибуты и методы, кроме, конечно же, приватных данных. При необходимости атрибуты можно переопределить для конкретного дочернего класса. Если при инициализации дочернего класса нам нужно дополнить новыми атрибутами, то чтобы не повторять код внутри метода init, нужно сначала использовать функцию super, чтобы обратиться к методу init суперкласса, а затем уже дополнять своими атрибутами.

Наследование — свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствованной функциональностью.

Это как раз основная суть взаимоотношений между классами и объектами, описанная выше. Чтобы не создавать кучу одинаковых объектов или классов, можно создать класс над классами с более общими характеристики и функциями, а потом постепенно наследовать от него те или иные возможности. Используя специальную конструкцию, программист может забрать из класса ряд атрибутов или методов, оставить их в прежнем виде и дополнить новыми или же слегка переосмыслить на свое усмотрение, а потом создать из них уникальный объект или подкласс для дальнейшего наследования опций.  Не нужно каждый раз создавать новый класс или объект с полным набором опций.

**Полиморфизм**

Полиморфизм – это свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Необходимость совместного использования объектов разных классов, способных обрабатывать однотипные сообщения, требует поддержки полиморфизма – возможности записывать разные объекты в переменные одного и того же типа. В таких условиях объект, отправляя сообщение, может не знать в точности, к какому классу относится адресат, и одни и те же сообщения, отправленные переменным одного типа, содержащим объекты разных классов, вызовут различную реакцию.

Полиморфизм реализуется путём введения в язык правил, согласно которым переменной типа «класс» может быть присвоен объект любого класса-потомка её класса.

Это один из ключевых принципов объектно-ориентированного программирования, позволяющий использовать одни и те же методы для обработки различных типов данных. Полиморфизм в разных языках программирования отличается: есть строго типизированные языки в духе C++, где задействуется «перегрузка», а есть такие языки, как JavaScript, где по умолчанию функции могут обрабатывать разные типы информации без необходимости указывать тип заранее.

Полиморфизм позволяет с помощью идентичных методов обрабатывать разные типы данных, например двузначные числа и числа с плавающей точкой. Также полиморфизмом считается возможность переопределять методы в дочерних классах для обработки других видов данных или выполнения дополнительных действий при вызове аналогичного метода.

Полиморфизм заключается в изменении функционала, который наследуется от базового класса. Это изменение в свою очередь, заключается в переопределении методов и атрибутов в дочернем классе так, как нам это нужно.

Объектно-ориентированное программирование исповедует ряд принципов, лежащих в основе правил создания и использования всех структурных элементов, включая классы, объекты, методы и прочие компоненты.

1. **Структурные элементы: классы, объекты, методы**

Работа с классами – это и есть объектно-ориентированное программирование.

**Классы**

Класс – универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей). В частности, в классах широко используются специальные блоки из одного или чаще двух спаренных методов, отвечающих за элементарные операции с определённым полем (интерфейс присваивания и считывания значения, геттеры-сеттеры), которые имитируют непосредственный доступ к полю. Эти блоки называются «свойствами» и почти совпадают по конкретному имени со своим полем (например, имя поля может начинаться со строчной, а имя свойств – с заглавной буквы). Другим проявлением интерфейсной природы класса является то, что при копировании соответствующей переменной через присваивание копируется только интерфейс, но не сами данные, то есть класс – это ссылочный тип данных. Переменная – объект, относящаяся к заданному классом типу, называется экземпляром этого класса. При этом в некоторых исполняющих системах класс также может представляться некоторым объектом при выполнении программы посредством динамической идентификации типа данных. Обычно классы разрабатывают таким образом, чтобы обеспечить отвечающие природе объекта и решаемой задаче целостность данных объекта, а также удобный и простой интерфейс. В свою очередь, целостность предметной области объектов и их интерфейсов, а также удобство их проектирования, обеспечивается наследованием.

Появление в объектно-ориентированной парадигме отдельного понятия класса закономерно вытекает из желания иметь множество объектов со сходным поведением. Класс в объектно-ориентированной парадигме – это в чистом виде абстрактный тип данных, создаваемый программистом. С этой точки зрения объекты являются значениями данного абстрактного типа, а определение класса задаёт внутреннюю структуру значений и набор операций, которые над этими значениями могут быть выполнены. Желательность иерархии классов (а значит, наследования) вытекает из требований к повторному использованию кода — если несколько классов имеют сходное поведение, нет смысла дублировать их описание, лучше выделить общую часть в общий родительский класс, а в описании самих этих классов оставить только различающиеся элементы.

В языке программирования Python, впрочем, как и в других языках, чтобы создать и работать с объектом, его нужно сначала описать. И это формальное описание объекта называется классом. Слова в названии класса пишутся с большой буквы. Такой стиль называется CamelCase и используется преимущественно для имен класса.

Как уже говорилось выше, в современных объектно-ориентированных языках программирования каждый объект является значением, относящимся к определённому классу. Класс представляет собой объявленный программистом составной тип данных, имеющий в составе:

**Поля данных**

Параметры объекта (конечно, не все, а только необходимые в программе), задающие его состояние (свойства объекта предметной области). Иногда поля данных объекта называют свойствами объекта, из-за чего возможна путаница. Фактически поля представляют собой значения (переменные, константы), объявленные как принадлежащие классу.

Это наиболее абстрактная и обобщенная форма в объектно-ориентированной парадигме. Что-то в духе шаблона, на базе которого строятся другие элементы структуры кода.

Описывать каждый элемент по отдельности сложно и нецелесообразно, ведь у них много общих характеристик и методов. Поэтому мы можем создать класс – то есть объект, способный стать базой для других объектов. Класс – это пример абстракции и наследования, упрощающий генерацию новых объектов.

**Объекты**

Сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

Объект в программировании – это некоторая сущность в цифровом пространстве, обладающая определённым состоянием и поведением, имеющая определённые свойства (атрибуты) и операции над ними (методы). Как правило, при рассмотрении объектов выделяется то, что объекты принадлежат одному или нескольким классам, которые определяют поведение (являются моделью) объекта. Термины «экземпляр класса» и «объект» взаимозаменяемы.

**Свойства объекта**

Псевдополя, доступные для чтения и/или записи. Свойства внешне выглядят как поля и используются аналогично доступным полям (с некоторыми исключениями), однако фактически при обращении к ним происходит вызов методов доступа. Таким образом, свойства можно рассматривать как «умные» поля данных, сопровождающие доступ к внутренним данным объекта какими-либо дополнительными действиями (например, когда изменение координаты объекта сопровождается его перерисовкой на новом месте). Свойства, по сути, не более чем синтаксический сахар, поскольку никаких новых возможностей они не добавляют, а лишь скрывают вызов методов доступа. Конкретная языковая реализация свойств может быть разной. Например, в C# объявление свойства непосредственно содержит код методов доступа, который вызывается только при работе со свойствами, то есть не требует отдельных методов доступа, доступных для непосредственного вызова. В Delphi объявление свойства содержит лишь имена методов доступа, которые должны вызываться при обращении к полю. Сами методы доступа представляют собой обычные методы с некоторыми дополнительными требованиями к сигнатуре.

В языке Python все значения являются объектами, даже классы. В этом языке можно построить класс, экземплярами которого будут классы. Такие классы называются метаклассами.

И хотя в структуре объектно-ориентированного программирования объекты находятся не на первом месте, мы начнем с них, так как это упрощает общее понимание парадигмы.

Объект – это кусок кода, описывающий элемент с конкретным набором характеристик и функций.

**Методы**

Методы – это функции, описанные внутри объекта или класса. Они относятся к конкретному объекту и позволяют взаимодействовать с ними или другими частями кода. Это процедуры и функции, связанные с классом. Они определяют действия, которые можно выполнять над объектом такого типа, и которые сам объект может выполнять.

Классы могут наследоваться друг от друга. Класс-потомок получает все поля и методы класса-родителя, но может дополнять их собственными либо переопределять уже имеющиеся. Большинство языков программирования поддерживает только единичное наследование (класс может иметь только один класс-родитель), лишь в некоторых допускается множественное наследование – порождение класса от двух или более классов-родителей.

Множественное наследование создаёт целый ряд проблем, как логических, так и чисто реализационных, поэтому в полном объёме его поддержка не распространена. Вместо этого в 1990-е годы появилось и стало активно вводиться в объектно-ориентированные языки понятие интерфейса.

Интерфейс – это класс без полей и без реализации, включающий только заголовки методов. Если некий класс наследует (или, как говорят, реализует) интерфейс, он должен реализовать все входящие в него методы. Использование интерфейсов предоставляет относительно дешёвую альтернативу множественному наследованию.

Взаимодействие объектов в абсолютном большинстве случаев обеспечивается вызовом ими методов друг друга.

Важно подчеркнуть, что в языке программированbя Python методы любого класса всегда должны принимать в качестве первого параметра именно слово self. Оно означает ссылку на текущий объект. Через эту ссылку внутри класса мы можем обратиться к атрибутам или даже методам этого же класса. Собственно, это отличительная особенность методов объектов от простых функций. Название self – это лишь условность языка Python. В ряде языков программирования есть своего рода аналог – ключевое слово this. Конечно же, методы могут принимать разное количество аргументов. Чтобы все было правильно, мы можем внутри метода прописывать проверку с помощью функции isinstance. В наших методах также могут использоваться именованные аргументы, и, конечно же, также могут возвращаться значения оператором return.

Таким образом, внутри классов можно прописывать свои функции, они называются методами класса. Первым параметром любого метода является слово self. Self означает ссылку на сам объект. С помощью этой ссылки мы получаем доступ к нужным нам атрибутам. Также как и обычные функции методы класса могут содержать именованные аргументы, а также возвращать значение с помощью оператора return.

**Конструктор \_\_init\_\_ и работа с несколькими классами**

Как мы уже говорили, экземпляров класса может быть много, и часто с каждым из них приходится работать индивидуально. И, конечно же, также часто их приходится инициализировать с какими-то индивидуальными входными данными. При работе с классами почти всегда используется специальный метод, который прописан в языке Python и который отвечает за инициализацию. Это конструкция с \_\_init\_\_. В языке программирования Python этот метод называется конструктором класса. Он называется так, потому что в этом методе мы своего рода конструируем наш объект. Передаем данные в качестве аргумента, через ключевое слово self инициализируем нужные нам атрибуты.

**Атрибуты**

Атрибуты – это конкретные характеристики объекта. Если вы хоть немного знакомы с программированием, то атрибуты можно представить в виде переменных с данными. В качестве атрибутов можно представить характеристики в духе уровня выносливости, скорости и других статических показателей.

Также при создании своих классов, методов и больших модулей в целом нередко пишется документация к ним, в которой описывается каждая структура, что принимается на вход, что возвращается, какие ошибки могут возникнуть и т.д. В программе необходимо описывать все классы и методы, используемые в программе.

Ранее мы уже говорили о том, что в реальном мире одни объекты постоянно взаимодействуют с другими, при работе с классами такое взаимодействие используется постоянно. Причем довольно часто один класс состоит из объектов другого класса, при этом методы у каждого свои.

Обычно для инициализации объекта в классе прописывается магический метод \_\_init\_\_. В качестве аргументов метод принимает self и все необходимые атрибуты, которыми характеризуется объект. Затем в теле метода объявляются сами атрибуты объекта и им присваиваются переданные значения. Таким образом, при создании экземпляра класса в скобках мы передаем необходимые значения аргумента с нужными значениями. В методе init нередко используются именованные аргументы, чтобы задать значение по умолчанию. А также в теле можно использовать и другие атрибуты, которые в класс не передаются. В атрибутах класса хранится состояние атрибута класса.

Имея в арсенале все вышеуказанные знания и понимание этих понятий, мы можем написать программу, реализующую игу «Крестики-нолики» в объектно-ориентированной парадигме, использовав понятия методов, классов и при соблюдении всех правил и норм языка программирования Python.

1. **Реализация программы «Крестики-нолики»**

import random

class Cell:

def \_\_init\_\_(self, number, value=' '\*5):

self.number = number

self.value = value

class Board:

def \_\_init\_\_(self):

self.dict\_cells = {}

for number\_cell in range(1, 10):

cell = Cell(number\_cell)

self.dict\_cells[number\_cell] = cell

def view\_board(self):

line = "-" \* 5 + "|" + "-" \* 5 + "|" + "-" \* 5

print(f'\n{self.dict\_cells[1].value}|{self.dict\_cells[2].value}|{self.dict\_cells[3].value}\n{line}\n'

f'{self.dict\_cells[4].value}|{self.dict\_cells[5].value}|{self.dict\_cells[6].value}\n{line}\n'

f'{self.dict\_cells[7].value}|{self.dict\_cells[8].value}|{self.dict\_cells[9].value}\n')

def win\_lines(self):

win\_lines = {'row\_1': [self.dict\_cells[1], self.dict\_cells[2], self.dict\_cells[3]],

'row\_2': [self.dict\_cells[4], self.dict\_cells[5], self.dict\_cells[6]],

'row\_3': [self.dict\_cells[7], self.dict\_cells[8], self.dict\_cells[9]],

'column\_1': [self.dict\_cells[1], self.dict\_cells[4], self.dict\_cells[7]],

'column\_2': [self.dict\_cells[2], self.dict\_cells[5], self.dict\_cells[8]],

'column\_3': [self.dict\_cells[3], self.dict\_cells[6], self.dict\_cells[9]],

'diagonal\_1': [self.dict\_cells[1], self.dict\_cells[5], self.dict\_cells[9]],

'diagonal\_2': [self.dict\_cells[3], self.dict\_cells[5], self.dict\_cells[7]],

}

return win\_lines

class Player:

def \_\_init\_\_(self, name):

self.name = name

self.board = Board()

self.num\_step = 0

def game(self):

print(f'Привет, {self.name}, первый ход мой!')

self.board.dict\_cells[self.machine\_logic()].value = ' 0 '

self.board.view\_board()

self.num\_step += 1

while True:

while True:

answer = int(input(f'В какую клетку пойдёте, {self.name}? '))

if self.board.dict\_cells[answer].value.isspace():

self.board.dict\_cells[answer].value = ' x '

if self.result() == 'stop game':

return

break

else:

print(f'Клетка занята, {self.name} выберите другую клетку.')

comp\_move = self.machine\_logic()

print(f'А я пойду в {comp\_move}-ую клетку.')

self.board.dict\_cells[comp\_move].value = ' 0 '

if self.result() == 'stop game':

return

self.board.view\_board()

def machine\_logic(self):

if self.num\_step == 0:

return random.choice(range(1, 10))

for i\_num in range(4):

for name\_win\_line, list\_cells in self.board.win\_lines().items():

count\_x, count\_0, empty\_cell = 0, 0, []

for i\_cell in list\_cells:

if i\_cell.value == ' 0 ':

count\_0 += 1

elif i\_cell.value == ' x ':

count\_x += 1

else:

empty\_cell.append(i\_cell)

if i\_num == 0 and count\_0 == 2 and empty\_cell:

return empty\_cell[0].number

elif i\_num == 1 and count\_x == 2 and empty\_cell:

return empty\_cell[0].number

elif i\_num == 2 and count\_0 == 1 and count\_x == 0 and empty\_cell:

return random.choice(empty\_cell).number

elif i\_num == 3 and count\_0 == 1 and count\_x == 1 and empty\_cell:

return empty\_cell[0].number

def result(self):

for name\_win\_line, list\_cells in self.board.win\_lines().items():

cnt\_x, cnt\_0 = 0, 0

for i\_cell in list\_cells:

if i\_cell.value == ' 0 ':

cnt\_0 += 1

elif i\_cell.value == ' x ':

cnt\_x += 1

if cnt\_x == 3:

print(f'Поздравляю с победой {self.name}!')

for new\_cell in list\_cells:

new\_cell.value = ' \033[34mx\033[0m '

self.board.view\_board()

return 'stop game'

elif cnt\_0 == 3:

print(f'Вы проиграли, {self.name}.')

for new\_cell in list\_cells:

new\_cell.value = ' \033[31m0\033[0m '

self.board.view\_board()

return 'stop game'

if not [val for val in self.board.dict\_cells.values() if val.value == ' '\*5]:

self.board.view\_board()

print('Игра окончена - победила дружба!')

return 'stop game'

my\_player = Player('Сергей')

my\_player.game()

1. **«Описание этапов создания программы «Крестики-нолики»**

Импортируем библиотеку random для случайного выбора клетки первого хода компьютера.

Создаем **класс Cell** (клетка), описывающий клетку игрового поля. **Метод \_\_init\_\_** принимает параметры number (номер клетки) и value (значение клетки), по умолчанию – 5 пробелов.

Создаем **класс Board** (доска), описывающий игровое поле. В методе \_\_init\_\_ инициализируется словарь dict\_cells. Далее в цикле наполняем этот словарь 9 клетками, инициализируя 9 объектов **класса Cell**. Ключом словаря является номер клетки, значением – **объект сell**.

**Метод view\_board** выводит игровое поле в консоль python при помощи функции print. Принимает объект **класса Board**.

**Метод vin\_lines** записывает выигрышные линии, колонки и диагонали игрового поля в переменную vin\_lines и возвращает ее.

Создаем **класс Player** (игрок), описывающий игрока.

**Метод \_\_init**\_\_ принимает аргумент name – имя игрока, инициализирует переменную board, записывая в нее объект класса Board. Также инициализирует переменную num\_step (номер хода), при инициализации объекта класса Player – по умолчанию 0.

**Метод machine\_logic.**

Осуществляем проверку: если переменная num\_step **класса Player** равна 0, вызываем оператор return и возвращаем случайное значение в диапазоне от 1 до 9, тем самым задаем номер клетки, в которую будет осуществляться первый ход компьютера.

Во всех остальных случаях:

1. Инициализируем цикл for в диапазоне 4, записывая значение в переменную i\_num.
2. Внутри цикла инициализируем цикл for в диапазоне значений, возвращающихся из **метода vin\_lines Класса Board** через переменную board и записываем эти значения в переменные name\_vin\_line и list\_cells.

Инициализируем переменные: count\_x со значением 0, count\_0 со значением 0 и empty\_cell, равную пустому списку.

1. Инициализируем цикл for в диапазоне значений, содержащихся в переменной list\_cells и записываем в переменную i\_cells.

Осуществляем проверку: если значение i\_cell равно 0, значит прибавляем в переменную count\_0 +1. Иначе если значение переменной i\_cell равно x, прибавляем переменную count­\_x +1.

Во всех остальных случаях добавляем в конец списка, содержащегося в переменной empty\_cell, переменную i\_cell.

Далее во втором цикле осуществляем проверку: если значение переменной i\_num равно 0 и значение переменной count\_0 равно 2 и переменная empty\_cell содержит какое-либо значение, отличное от 0, вызываем оператор return и возвращаем первое значение из списка, содержащегося в переменной empty\_cell, т.е. номер клетки.

Иначе если значение переменной i\_num равно 2 и значение переменной count\_0 равно 1 и значение переменной count\_x равно 0 и переменная empty\_cell содержит какое-либо значение, отличное от 0, вызываем оператор return и возвращаем случайное значение из списка, записанного в переменной empty\_cell, т.е. номер клетки.

Иначе если значение переменной i\_num равно 3 и значение переменной count\_0 равно 1 и значение переменной count\_X равно 1 и переменная empty\_cell содержит какое-либо значение, отличное от 0, то вызываем оператор return и возвращаем первое значение из списка, записанного в переменной empty\_cell, т.е. номер клетки.

**Метод result**.

* Инициализируем цикл for в диапазоне значений, возвращающихся из **метода vin\_lines класса Board** через переменную Board и записываем эти значения в переменные name\_vin\_line и list\_cells.
  + Инициализируем переменные cnt\_х, равную 0 и cnt\_0, равную 0.
  + Инициализируем цикл for в диапазоне значений, содержащихся в переменной list\_cells и записываем значение в переменную i\_cell. Внутри цикла осуществляем проверки:
    - если значение переменной i\_cell равно 0, то прибавляем к переменной cnt\_0 +1
    - иначе если значение переменной i\_cell равно 0, то прибавляем к переменной cnt\_x +1
    - если значение переменной cnt\_x равно 3, выводим в консоль сообщение «Поздравляю с победой!» и инициализируемый цикл for в диапазоне значений, содержащихся в переменной list\_cells. Записываем значение в переменную new\_cell и устанавливаем значение этой переменной х, окрашенный в зеленый цвет, тем самым окрашивая линию, диагональ или столбец, по которому победил пользователь в зеленый цвет.

Вызываем **метод view\_board класса Board** через переменную board, тем самым обновляем игровое поле в консоли. Вызываем оператор return, возвращая значение «stop game».

* + - Иначе если значение переменной cnt\_0 равно 3, выводим в консоль сообщение «Вы проиграли!» и инициализируемый цикл for в диапазоне значений, содержащихся в переменной list\_cells. Записываем значение в переменную new\_cell и устанавливаем значение этой переменной 0, окрашенный в красный цвет, тем самым окрашивая линию, диагональ или столбец, по которому проиграл пользователь, в красный цвет.

Вызываем **метод view\_board класса Board** через переменную board, тем самым обновляем игровое поле в консоли. Вызываем оператор return, возвращая значение «stop game».

* При помощи list comprehentions, инициализируем список циклом for в диапазоне значений словаря dict\_cells класса Board через переменную board, записывая значение в переменную val.
  + - Осуществляем проверку: если значение переменной val равно строке из 5 пробелов, тогда записываем переменную val в инициализированный список.
    - Если инициализированный список оказался пустым, то выводим в консоль сообщение «Игра окончена! Победила дружба!». Вызываем оператор return и возвращаем значение «stop game».

**Метод game** описывает игру:

* Выводит приветственное сообщение «Привет, пользователь! Первый ход мой.»
* Обращается через переменную board к словарю dict\_cells **объекта класса Board**, устанавливая значение 0 по ключу, сгенерированному **методом machine\_logic класса Player**, тем самым осуществляя первый ход компьютера.
* Обращаемся к методу view\_board **класса Board** через переменную board, тем самым вводим в консоль обновленное игровое поле.
* Прибавляем к переменной num\_step + 1.
* Инициализируем бесконечный цикл.
  + Инициализируем вложенный бесконечный цикл и на нижнем уровне вложенности осуществляем следующие действия:
    - Инициализируем переменную answer через функцию input с приглашением «В какую клетку пойдете?»
    - Осуществляем проверку: если номер клетки, введенной пользователем, соответствует пустому значению клетки в словаре dict\_cells класса Board, тогда записываем в эту клетку значение х.
    - Осуществляем проверку: обращаемся к **методу result класса Player**. Если вернувшееся значение равно «stop game», вызываем оператор return, тем самым заканчивая игру.
* Во всех остальных случаях вызываем оператор break, тем самым завершая вложенный бесконечный цикл.
* Иначе выводим в консоль сообщение «Клетка занята. Выберете другую клетку.»
* В бесконечном цикле верхнего уровня осуществляем следующие действия:
* Инициализируем переменную comp\_move, записывая в нее значение, вернувшееся из **метода machine\_logic**, тем самым определяя номер клетки, в которую будет осуществлен следующий ход компьютера. И выводим в консоль сообщение «А я пойду в клетку ...».
* Записываем в словарь dict\_cells **класса Board** через переменную Board **класса Player** значение 0, по ключу, записанному в переменную comp\_move.
* Осуществляем проверку: если значение, вернувшееся из метода result **класса Player** равно «stop game», вызываем оператор return, тем самым заканчивая игру. Во всех остальных случаях вызываем **метод view\_board класса Board** через переменную board **класса** **Player**, таким образом выводим в консоль обновленное игровое поле.

В основном коде модуля инициализируем переменную my\_player и записываем в нее значение инстанса **класса** **Player**, указав значение name, таким образом инициализируем игрока.

Вызываем **метод game класса Player** через переменную my\_player, таким образом запуская игру игрока, записанного в переменной my\_player с данным скриптом.

Таким образом, используя парадигму объектно-ориентированного программирования, применяя знания о таких понятиях как методы, классы, объекты и ключевые принципы объектно-ориентированного программирования, мы смогли реализовать и описать игру «Крестики-нолики» на языке программирования Python.

**Список использованной литературы:**

1. *Роганов Е. А.* [Основы информатики и программирования](http://230100.msiu.ru/files/13-main.pdf). — М.: МГИУ, 2001. [Архивная копия](http://web.archive.org/web/20131008041515/http:/230100.msiu.ru/files/13-main.pdf) от 8 октября 2013 на [Wayback Machine](https://ru.wikipedia.org/wiki/Wayback_Machine" \o "Wayback Machine)
2. *D. G. Bobrow.* If Prolog is the answer, what is the question. // *Fifth Generation of Computer Systems*, pages 138—145, Tokyo, Japan, November 1984. Institute for New Generation Computer Technology (ICOT), North-Holland.
3. *B. D. Shriver.* Software paradigms. IEEE Software, 3(1):2, January 1986.
4. *L. W. Friedman.* Comparative programming languages: generalizing the programming function. Prentice Hall, 1991, page 188.
5. *P. Zave.* A compositional approach to multiparadigm programming. IEEE Software, 6(5): 15—25, September 1989.
6. *P. Wegner.* Concepts and paradigms of object-oriented programming. *{OOPS} messenger}*, 1(1): 7—87, August 1990.
7. *T. A. Budd.* Multi-Paradigm Programming in LEDA. Addison-Wesley, Reading, Massachusetts, 1995.
8. https://ru.wikipedia.org/wiki/
9. *Гради Буч.* Объектно-ориентированный анализ и проектирование с примерами приложений на С++. — Бином, 1998. — [ISBN 0-8053-5340-2](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/0805353402), [ISBN 5-7989-0067-3](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/5798900673), [ISBN 5-7940-0017-1](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/5794000171).