* [ОБЗОР КУРСА](https://lyceum.yandex.ru/courses/123/groups/568)

[Урок Классы 1](https://lyceum.yandex.ru/courses/123/groups/568/lessons/659)

**Проектирование и разработка классов. Часть №1**

**План урока**

1

[Подготовительный этап](https://lyceum.yandex.ru/courses/123/groups/568/lessons/659/materials/1726#1)

2

[Движение фигуры по пустой доске](https://lyceum.yandex.ru/courses/123/groups/568/lessons/659/materials/1726#2)

**Аннотация**

*На этом занятии мы коснёмся обширной темы проектирования программ. Чем больше и сложнее программа, тем важнее ещё до написания кода понять, что она должна делать и какова будет её внутренняя структура.*

**Проектирование**

Этап определения внутренней структуры программы, разнесения функциональности между модулями (а затем и внутри модулей — между функциями, классами и методами) называется проектированием. Чем сложнее программа, тем больше возможностей придётся рассмотреть при проектировании, и тем важнее выбрать подходящий вариант. Результат проектирования должен быть внутренне непротиворечив, хорошо отражать логику предметной области и допускать дальнейшее развитие программы без перепроектирования.

Мы разберём проектирование программы для игры в шахматы. Программа предназначена для двух игроков, которые за одним компьютером будут по очереди делать ходы. Программа должна принимать только допустимые ходы и отображать на экране положение фигур после каждого хода с помощью псевдографики. Никакого искусственного интеллекта и возможности самостоятельно делать ходы от программы не требуется.

Можно было бы просто выбрать готовый вариант проектирования и приступить к поэтапной реализации каждого класса и метода. Однако в этом случае мы не смогли бы рассмотреть варианты и увидеть, что бывает, когда программа спроектирована некорректно и её приходится перепроектировать на лету. Такое случается, когда требования к программе изменяются и уточняются в ходе её написания.

**Этапы проектирования программы**

Поэтому мы будем проектировать программу и сразу писать код, делая это в несколько этапов:

* подготовительный этап: цвет фигур, доска;
* движение фигуры по пустой доске: реализация класса доски и классов пешки и ладьи;
* движение фигуры по доске с другими фигурами: ферзь, ладья, слон и пешка (при ходе на две клетки) не должны иметь препятствий на пути; взятие фигуры другой фигурой.

Мы не будем реализовывать взятие на проходе, рокировку, превращение пешки, запрет хода королём под шах и прочие сложные правила.

**1. Подготовительный этап**

Для начала опишем цвет фигуры. От цвета нам нужно три операции: задание цвета фигуры, проверка цвета (чёрный или белый) и получение противоположного цвета.

Простейший способ описать цвет — это определить две константы WHITE и BLACK и функцию, которая будет возвращать цвет, противоположный переданному.

WHITE = 1

BLACK = 2

*# Удобная функция для вычисления цвета противника*

**def** opponent(color):

**if** color == WHITE:

**return** BLACK

**else**:

**return** WHITE

*# Инициализация цвета*

color = WHITE

*# Проверка цвета*

**if** color == BLACK:

do\_something()

*# сравнение цветов*

color == other\_color

*# Цвет противника*

opponent\_color = opponent(color)

Обратите внимание, что используются именно константы, а не строки «white» и «black». Если допустить в такой строке опечатку (например, whte вместо white), то выражение всегда будет ложным. Однако в случае color == «whte» программа просто будет тихонько работать неправильно, и ошибку придётся поискать. Если же написать color == WHTE, то интерпретатор прекратит работу, встретив неизвестное имя WHTE, и сразу укажет, в какой строке ошибка.

**Обозначение констант в Python**

Переменные BLACK и WHITE названы в верхнем регистре, чтобы показать, что они фактически будут константами, то есть мы не собираемся их изменять. Настоящих констант в Python нет, и WHITE — такая же переменная, как и любая другая, однако такие стандартные обозначения помогают писать и читать программы.

На этом варианте кода с константами WHITE и BLACK мы и остановимся в итоге. Но давайте попробуем ещё пару вариантов.

*# Не здравый пример лишнего класса для цвета:*

WHITE = 1

BLACK = 2

**class** PieceColor():

**def** \_\_init\_\_(self, color):

self.color = color

**def** opponent(self):

**if** self.color == WHITE:

**return** PieceColor(BLACK)

**else**:

**return** PieceColor(WHITE)

**def** is\_black(self):

**return** self.color == BLACK

**def** is\_white(self):

**return** self.color == WHITE

**def** \_\_eq\_\_(self, other):

**return** self.color == other.color

*# Инициализация цвета*

color = PieceColor(WHITE)

*# Проверка цвета*

**if** color.is\_black():

do\_something()

*# сравнение цветов*

color == other\_color

*# Цвет противника*

opponent\_color = color.opponent()

*# Пример с отдельным классом для каждого цвета*

**class** Black():

**def** \_\_eq\_\_(self, other):

*# истина, если другой операнд оператора ==*

*# тоже является экземпляром (англ. instance) класса Black*

**return** isinstance(other, Black)

**def** opponent(self):

**return** White()

**def** is\_black(self):

**return** True

**def** is\_white(self):

**return** False

**class** White():

**def** \_\_eq\_\_(self, other):

**return** isinstance(other, White)

**def** opponent(self):

**return** Black()

**def** is\_black(self):

**return** False

**def** is\_white(self):

**return** True

*# Инициализация цвета*

color = White()

*# Проверка цвета*

color.is\_black()

*# сравнение цветов*

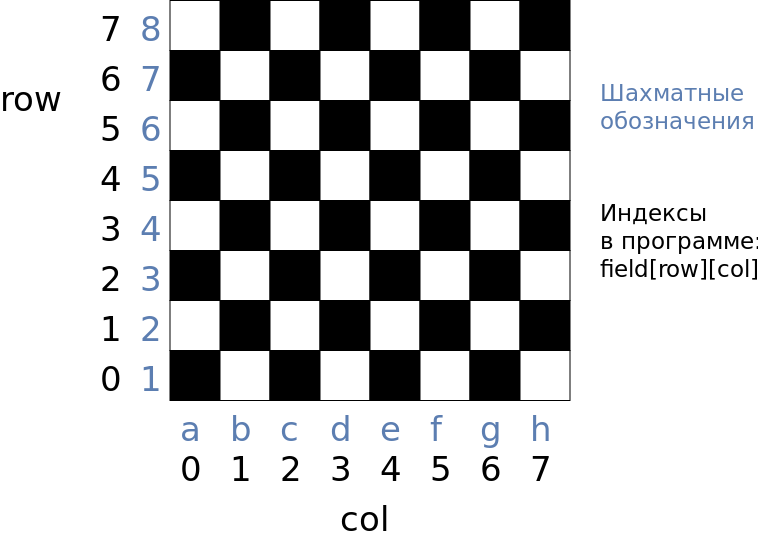
color == other\_color

*# Цвет противника*

opponent\_color = color.opponent()

Видно, что во втором варианте реализации цветов мы не смогли избавиться от констант WHITE и BLACK. В третьем вместо двух констант получились два класса, а код их использования прост и близок к человеческому языку. Однако всё портит излишняя сложность классов White и Black. Возможно, в какой-то другой программе это было бы оправдано, однако здесь мы точно знаем постановку задачи и не будем «навешивать» на цвета фигур дополнительной функциональности. Поэтому остановимся на самом первом варианте с двумя константами и функцией.

**2. Движение фигуры по пустой доске**



На картинке серым цветом показаны обозначения клеток, принятые в шахматах, а чёрным — индексы, которые будем использовать мы.

Давайте спроектируем шахматную доску и позволим одной фигуре ходить по ней. Пока будем считать, что других фигур нет, и взаимодействия с ними учитывать не будем.

Пока не очень понятно, какие у нас будут классы и методы. Очевидно, что нужно определить класс Board для доски и по классу для каждого типа фигуры. Но как распределить между ними функциональность и кто будет отвечать за ходы и взятия фигур, пока неясно.

**Проектирование «сверху вниз» и «снизу вверх»**

Здесь можно пойти двумя путями: проектировать и программировать «сверху вниз» или «снизу вверх». Проектирование «сверху вниз» идёт от общего к частному: от интерфейса к деталям реализации. При движении «снизу вверх» сначала проектируются простые независимые классы и функции, а потом, на их основе, создаются более сложные классы и функции.

В начале урока мы написали код для цвета фигур, программируя «снизу вверх». Однако сейчас мы плохо представляем, какой интерфейс должен быть у доски и фигур. Если мы попробуем реализовать эти классы сразу, то есть риск, что при объединении в интерфейс пользователя мы не сумеем совместить их оптимальным образом. Поэтому сначала напишем интерфейс пользователя, чтобы понять, какие интерфейсы нам нужны от доски и фигур:

**def** print\_board(board): *# Распечатать доску в текстовом виде (см. скриншот)*

**print**(' +----+----+----+----+----+----+----+----+')

**for** row **in** range(7, -1, -1):

**print**(' ', row, end=' ')

**for** col **in** range(8):

**print**('|', board.cell(row, col), end=' ')

**print**('|')

**print**(' +----+----+----+----+----+----+----+----+')

**print**(end=' ')

**for** col **in** range(8):

**print**(col, end=' ')

**print**()

**def** main():

*# Создаём шахматную доску*

board = Board()

*# Цикл ввода команд игроков*

**while** True:

*# Выводим положение фигур на доске*

print\_board(board)

*# Подсказка по командам*

**print**('Команды:')

**print**(' exit -- выход')

**print**(' move <row> <col> <row1> <col1> -- ход из клетки (row, col)')

**print**(' в клетку (row1, col1)')

*# Выводим приглашение игроку нужного цвета*

**if** board.current\_player\_color() == WHITE:

**print**('Ход белых:')

**else**:

**print**('Ход чёрных:')

command = input()

**if** command == 'exit':

**break**

move\_type, row, col, row1, col1 = command.split()

row, col, row1, col1 = int(row), int(col), int(row1), int(col1)

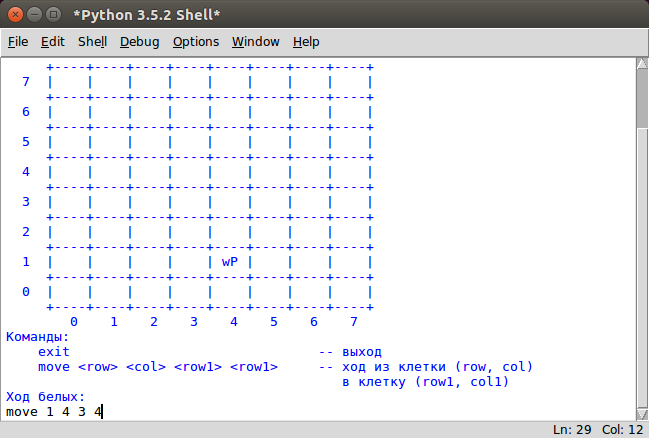
**if** board.move\_piece(row, col, row1, col1):

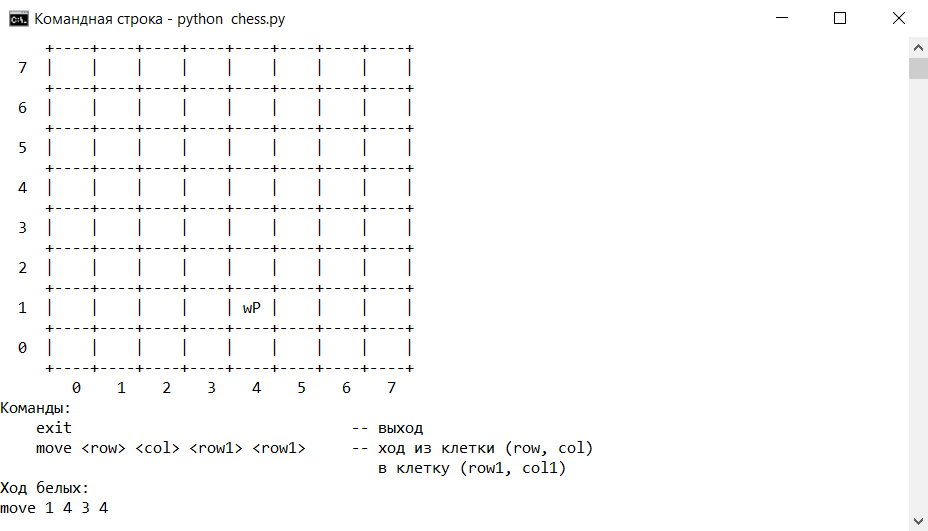
**print**('Ход успешен')

**else**:

**print**('Координаты некорректны! Попробуйте другой ход!')

Пример работы текстового интерфейса





Теперь понятно, что от доски нам понадобится достаточно простой интерфейс: инициализация без аргументов; возможность определять цвет фигур текущего игрока; метод cell, возвращающий двухбуквенное представление фигуры в клетке; и метод move\_piece, который должен перемещать фигуру из одной клетки в другую. При этом метод move\_piece должен возвращать истину, если ход сделан, и ложь, если по каким-то причинам ход невозможен.

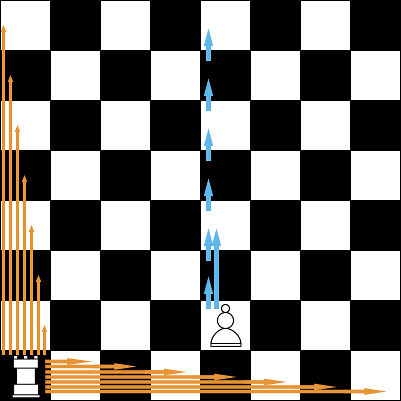


Схема движения ладьи и пешки. Ладья движется по вертикали или горизонтали на любое количество клеток. Пешка — на одну или две клетки вперёд из начального положения и на одну клетку не из начального положения.

Давайте реализуем этот класс:

**def** correct\_coords(row, col):

'''Функция проверяет, что координаты (row, col) лежат

внутри доски'''

**return** 0 <= row < 8 **and** 0 <= col < 8

**class** Board:

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

self.color = WHITE

self.field = []

**for** row **in** range(8):

self.field.append([None] \* 8)

self.field[1][4] = Pawn(1, 4, WHITE)  
 *# Пешка белого цвета в клетке E2.*

**def** current\_player\_color(self):

**return** self.color

**def** cell(self, row, col):

'''Возвращает строку из двух символов. Если в клетке (row, col)

находится фигура, символы цвета и фигуры. Если клетка пуста,

то два пробела.'''

piece = self.field[row][col]

**if** piece **is** None:

**return** ' '

color = piece.get\_color()

c = 'w' **if** color == WHITE **else** 'b'

**return** c + piece.char()

**def** move\_piece(self, row, col, row1, col1):

'''Переместить фигуру из точки (row, col) в точку (row1, col1).

Если перемещение возможно, метод выполнит его и вернёт True.

Если нет --- вернёт False'''

**if** **not** correct\_coords(row, col) **or** **not** correct\_coords(row1, col1):

**return** False

**if** row == row1 **and** col == col1:

**return** False *# нельзя пойти в ту же клетку*

piece = self.field[row][col]

**if** piece **is** None:

**return** False

**if** piece.get\_color() != self.color:

**return** False

**if** **not** piece.can\_move(row1, col1):

**return** False

self.field[row][col] = None *# Снять фигуру.*

self.field[row1][col1] = piece *# Поставить на новое место.*

piece.set\_position(row1, col1)

self.color = opponent(self.color)

**return** True

В этом фрагменте кода мы определили функцию correct\_coords, которая нужна для проверки корректности клетки, и класс Board, который отвечает за выбор фигуры и её движение. Также понятно, что от каждого класса фигур нам понадобится инициализатор с аргументом — цветом фигуры, методы current\_player\_color, can\_move и char. Метод current\_player\_color должен возвращать цвет фигур текущего игрока, can\_move — определять, может ли фигура данного класса пойти в клетку с заданными координатами, char — возвращать букву, обозначающую фигуру. У каждого класса фигуры будет своя реализация can\_move, поэтому будет задействован полиморфизм. Давайте для примера напишем реализации классов пешки и ладьи:

*# Класс Пешка*

**class** Pawn:

**def** \_\_init\_\_(self, row, col, color):

self.row = row

self.col = col

self.color = color

**def** set\_position(self, row, col):

self.row = row

self.col = col

**def** char(self):

**return** 'P'

**def** get\_color(self):

**return** self.color

**def** can\_move(self, row, col):

*# Пешка может ходить только по вертикали*

*# "взятие на проходе" не реализовано*

**if** self.col != col:

**return** False

*# Пешка может сделать из начального положения ход на 2 клетки*

*# вперёд, поэтому поместим индекс начального ряда в start\_row.*

**if** self.color == WHITE:

direction = 1

start\_row = 1

**else**:

direction = -1

start\_row = 6

*# ход на 1 клетку*

**if** self.row + direction == row:

**return** True

*# ход на 2 клетки из начального положения*

**if** self.row == start\_row **and** self.row + 2 \* direction == row:

**return** True

**return** False

*# Класс Ладья*

**class** Rook:

**def** \_\_init\_\_(self, row, col, color):

self.row = row

self.col = col

self.color = color

**def** set\_position(self, row, col):

self.row = row

self.col = col

**def** char(self):

**return** 'R'

**def** get\_color(self):

**return** self.color

**def** can\_move(self, row, col):

*# Невозможно сделать ход в клетку, которая не лежит в том же ряду*

*# или столбце клеток.*

**if** self.row != row **and** self.col != col:

**return** False

**return** True

Вы наверняка обратили внимание, что написанные нами классы для фигур, имеют очень много общего. Это должно навести нас на мысль, что можно эффективно применить механизм наследования. Попробуйте написать этот код самостоятельно.

[Справка](https://yandex.ru/support/lyceum-students)

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках проекта «Яндекс.Лицей», принадлежат АНО ДПО «ШАД». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «ШАД».

© 2018 – 2020  ООО «[Яндекс](https://yandex.ru/)»

Чаты