Практическое занятие №1

П.Н. Советов, РТУ МИРЭА

Запустите IDLE. Проверьте работу интерпретатора в режиме REPL на простых арифметических примерах. Научитесь создавать новые программы в редакторе IDLE, сохранять их с расширением .py, запускать программы.

1. Знакомство с языком

1.1. (уровень сложности: низкий)

Запишите число 42 с помощью 10 различных способов, чем разнообразнее, тем лучше. Использовать литеральную запись, без арифметики и функций. В любом варианте должно соблюдаться == 42.

1.2. (уровень сложности: низкий)

Есть ли ограничения на диапазон чисел в Питоне? Как насчет чисел с плавающей запятой? Какое максимальное значение вы сумели получить до переполнения?

1.3. (уровень сложности: низкий)

Найдите в документации встроенную (builtin) арифметическую функцию, которая возвращает не одно, а два значения разом.

1.4. (уровень сложности: низкий)

С приведенным ниже циклом что-то не так. Как это исправить?

Что на самом деле представляет собой 0.1 внутри интерпретатора? Можно ли увидеть его настоящее значение? Поэкспериментируйте.

In [4]:
```
a = 10
while a != 0:
    a -= 0.1
```

1.5. (уровень сложности: низкий)

А вот совсем уже безобидный код. Циклов нет, но программа зависает. Почему?

In [ ]:
```
z = 1
z <<= 40
2 ** z
```

1.6. (уровень сложности: низкий)

Да-да, и этот код зацикливается! А тут-то что не так?

In [ ]:
```
i = 0
while i < 10:
    print(i)
    ++i
```

1.7. (уровень сложности: низкий)

Что за странное выражение и странный результат?

In [3]:
```
(True * 2 + False) * -True
```

Out[3]:
```
-2
```

1.8. (уровень сложности: низкий)

В Питоне можно использовать цепочки операций сравнения. Рассмотрите следующие примеры и попробуйте объяснить код:

In [1]:
```
x = 5
1 < x < 10
```

Out[1]:
```
True
```

In [2]:
```
x = 5
1 < (x < 10)
```

Out[2]:
```
False
```

2. Сообщения об ошибках

К сообщениям об ошибках Питона нужно привыкать, в них нет ничего страшного. Давайте специально напишем некорректный код для того, чтобы получить каждое из указанных ниже сообщений об ошибках.

2.1. (уровень сложности: простейший)

SyntaxError: invalid syntax

2.2. (уровень сложности: простейший)

SyntaxError: cannot assign to literal

2.3. (уровень сложности: простейший)

NameError: name ... is not defined

2.4. (уровень сложности: простейший)

SyntaxError: unterminated string literal

2.5. (уровень сложности: простейший)

TypeError: unsupported operand type(s) for ...

2.6. (уровень сложности: простейший)

IndentationError: expected an indented block

2.7. (уровень сложности: простейший)

IndentationError: unindent does not match any outer indentation level

2.8. (уровень сложности: простейший)

ValueError: math domain error

2.9. (уровень сложности: простейший)

OverflowError: math range error

3. Арифметика

Мир микропроцессоров не ограничивается только большими чипами для настольных применений. Маломощные микроконтроллеры могут выступать в роли умных датчиков в задачах, связанных с Интернетом вещей. Слабенькие 8-битные микропроцессоры являлись «сердцем» многих классических игровых приставок. Типичный 8-битный процессор не имеет аппаратной поддержки умножения. Как же выкручиваются программисты в этой ситуации?

Представим, что в Питоне тоже отсутствует операция умножения. Ее можно заменить сложением. Если мы хотим умножить какое-то число x на 12, то нам понадобится 11 сложений, правильно? Это довольно много, но, оказывается, можно обойтись и меньшим числом сложений.

Из арифметических операций разрешается использовать только явно указанные и в указанном количестве. Входным аргументом является переменная x. Унарный минус использовать нельзя. Тело программы должно состоять из линейной последовательности присваиваний. Оформите линейный код решения в виде функции.

3.1. (уровень сложности: низкий)

Умножение на 12. Используйте 4 сложения.

3.2. (уровень сложности: низкий)

Умножение на 16. Используйте 4 сложения.

3.3. (уровень сложности: средний)

Умножение на 15. Используйте 3 сложения и 2 вычитания.

3.4. (уровень сложности: низкий)

Некто попытался реализовать «наивную» функцию умножения с помощью сложений. К сожалению, в коде много ошибок. Сможете ли вы их исправить?

Добавьте к naive_mul автоматическое тестирование на случайных данных. Сравнивайте с встроенным умножением, используя конструкцию assert.

In [ ]:
```
def naive_mul(x, y):
   r = 1;
    for i in range(0, y - 1)
    x = x + r;
    end
```

Существует старинный метод умножения по методу русского крестьянина. Разобрать его проще на примере.

Предположим, мы хотим перемножить 10 и 15:

x	y
10	15
5	30
2	60
1	120

В первом столбце последовательно записывают результаты деления на 2 с отбрасыванием остатка. Во втором столбце находятся результаты умножения на 2. Отмечаем нечетные числа в первом столбце. Складываем те числа в правом столбце, которые стоят напротив отмеченных ранее чисел. То есть 30 + 120 = 150. В нашем случае мы не учитывали исходные числа при сложении, но учитывать их, вообще говоря, может быть нужно.

В этом алгоритме используются лишь простейшие операции: умножение на 2, целочисленное деление на 2, проверка на нечетность и сложение. Эти операции соответствуют тем элементарным действиям, которые эффективно выполняет любой процессор. Сам же алгоритм, несмотря на кажущуюся необычность, является вариантом умножения в столбик при использовании двоичного представления чисел:

```
   1111
   1010 *
   ----
   0000
  1111
 0000
1111
```

3.5. (уровень сложности: средний)

Реализуйте функцию fast_mul в соответствии с алгоритмом двоичного умножения в столбик (без рекурсии!). Добавьте автоматическое тестирование, как в случае с naive_mul.

3.6. (уровень сложности: средний)

Реализуйте аналогичную функцию fast_pow для возведения в степень. Решение необходимо получить только с помощью небольших модификаций предыдущего решения.

3.7. (уровень сложности: средний)

Предположим, что стоит задача перемножить два 16-битных числа без знака. Однако в наличии у нас имеется только операция умножения 8-битных чисел. Оказывается, можно применить четыре 8-битных умножения, если разбить исходные числа на 8-битные половинки. Побитовые сдвиги умножениями не считаются.

Выведите формулу для умножения «половинок» и реализуйте функцию mul16(x, y), используя приведенную ниже функцию mul_bits, которая имитирует умножение с ограниченной числом bits разрядностью аргументов.

In [1]:
```
def mul_bits(x, y, bits):
    x &= (2 ** bits - 1)
    y &= (2 ** bits - 1)
    return x * y
```

3.8. (уровень сложности: средний)

Четыре умножения из предыдущей задачи – не предел. Советский ученый Анатолий Алексеевич Карацуба в 1960 г. предложил формулу (а для общего случая – рекурсивный алгоритм), требующую лишь трех умножений. Выведите эту формулу с подсказками от преподавателя и реализуйте функцию mul16k(x, y).

3.9. (уровень сложности: высокий)

Иногда возникает необходимость в создании программы, которая в качестве результата выдаст другую программу. В этом нет ничего необычного, так устроен, к примеру, компилятор. Но если задача стоит узкоспециальная, то речь идет о создании генератора программ по заданным параметрам.

Реализуйте генератор программ fast_mul_gen(y) для задач 2.1-2.3. Воспользуйтесь ранее полученным кодом fast_mul. Ваша функция должна выдать текст функции f(x) (умножение на ранее заданный y), тело которой состоит из некоторого числа присваиваний. Для вывода функции используйте print. Добавьте автоматическое тестирование. Объясните, почему в общем случае у вас получается большее количество сложений, чем в задачах 2.1-2.3.

3.10. (уровень сложности: низкий)

Реализуйте на основе fast_mul_gen генератор программ для возведения в степень.

4. Пиксельные шейдеры

Шейдеры представляют собой небольшие программы, обычно предназначенные для исполнения на графической карте. Шейдеры могут работать параллельно и не запоминают свое состояние. Это просто функции, переводящие координаты экрана в цвет. Шейдеры широко используются для создания специальных эффектов, а также в играх. Обычно шейдеры программируют на C-подобных языках. Попробуем имитировать работу шейдеров прямо в Питоне!

Итак, вся работа должна производится в теле функции func(x, y). Координаты заданы в диапазоне [0, 1). Функция возвращает три цветовых компонента, каждый из которых также находится в диапазоне [0, 1]. При решении задач старайтесь не использовать ветвлений и, тем более, циклов. Не забывайте, что шейдеры не имеют доступа к глобальным данным и поэтому даже модуль random использовать нельзя.

Для решения задач понадобится приведенная ниже заготовка программы.

Дополнительная информация

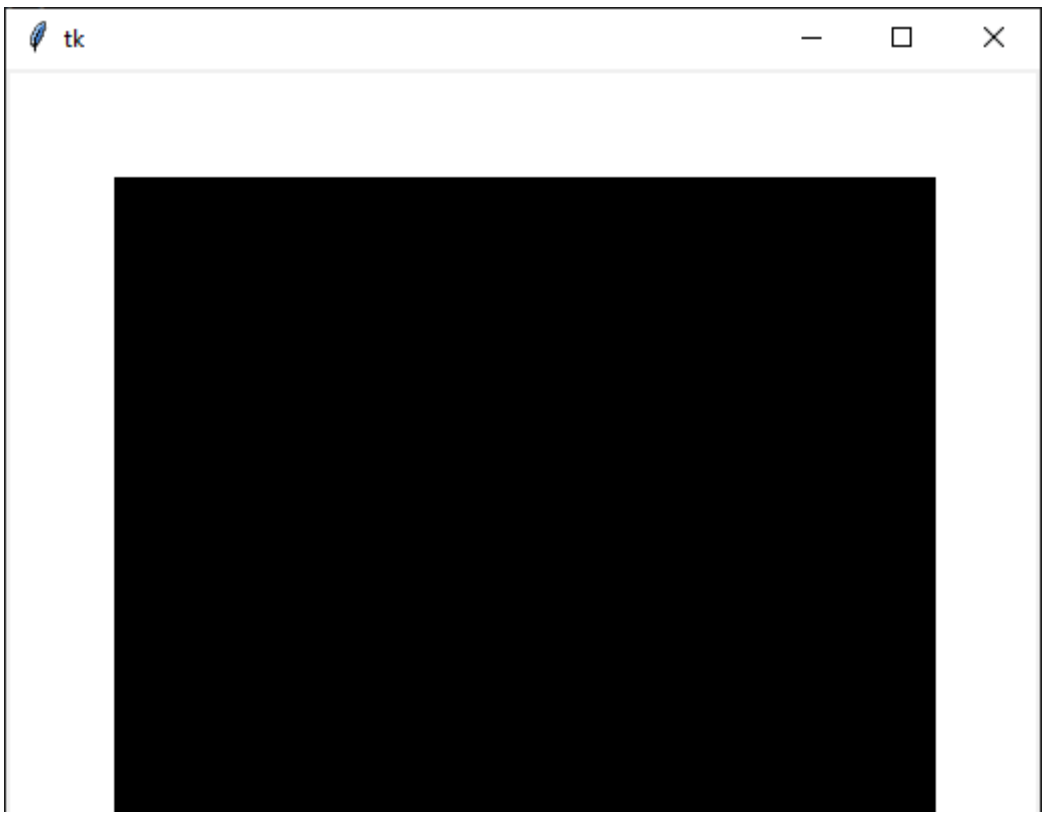
1. Книга The Book of Shaders.
2. Сайт известного специалиста в области процедурной графики Inigo Quilez.
3. О Value Noise.

In [1]:

```
import math
import tkinter as tk


def draw(shader, width, height):
    image = bytearray((0, 0, 0) * width * height)
    for y in range(height):
        for x in range(width):
            pos = (width * y + x) * 3
            color = shader(x / width, y / height)
            normalized = [max(min(int(c * 255), 255), 0) for c in color]
            image[pos:pos + 3] = normalized
    header = bytes(f'P6\n{width} {height}\n255\n', 'ascii')
    return header + image


def main(shader):
    label = tk.Label()
    img = tk.PhotoImage(data=draw(shader, 256, 256)).zoom(2, 2)
    label.pack()
    label.config(image=img)
    tk.mainloop()


def shader(x, y):
    # Ваш код здесь:
    return x, y, 0


main(shader)
```

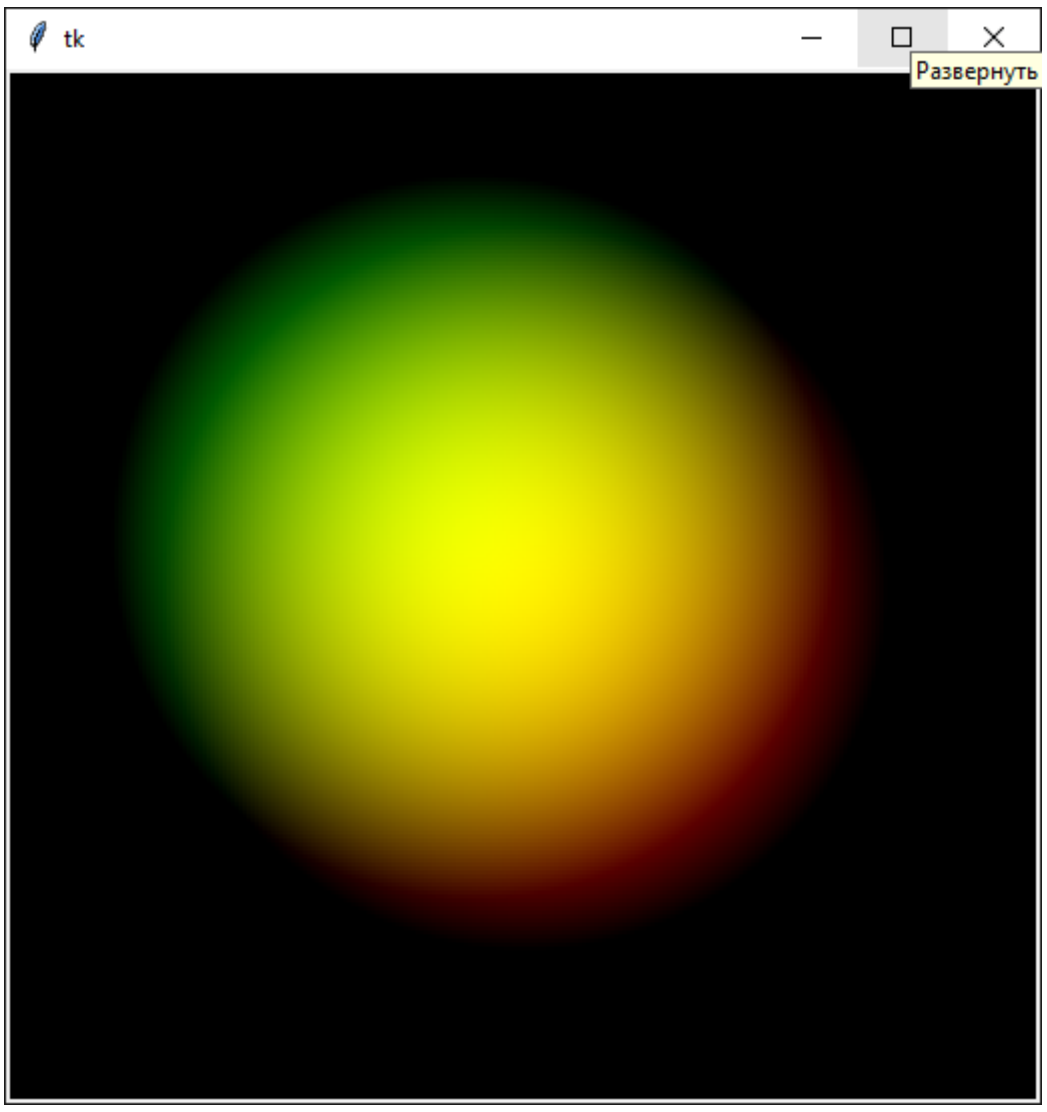
4.1. (уровень сложности: низкий)

Изобразите свою версию знаменитого «Черного квадрата».



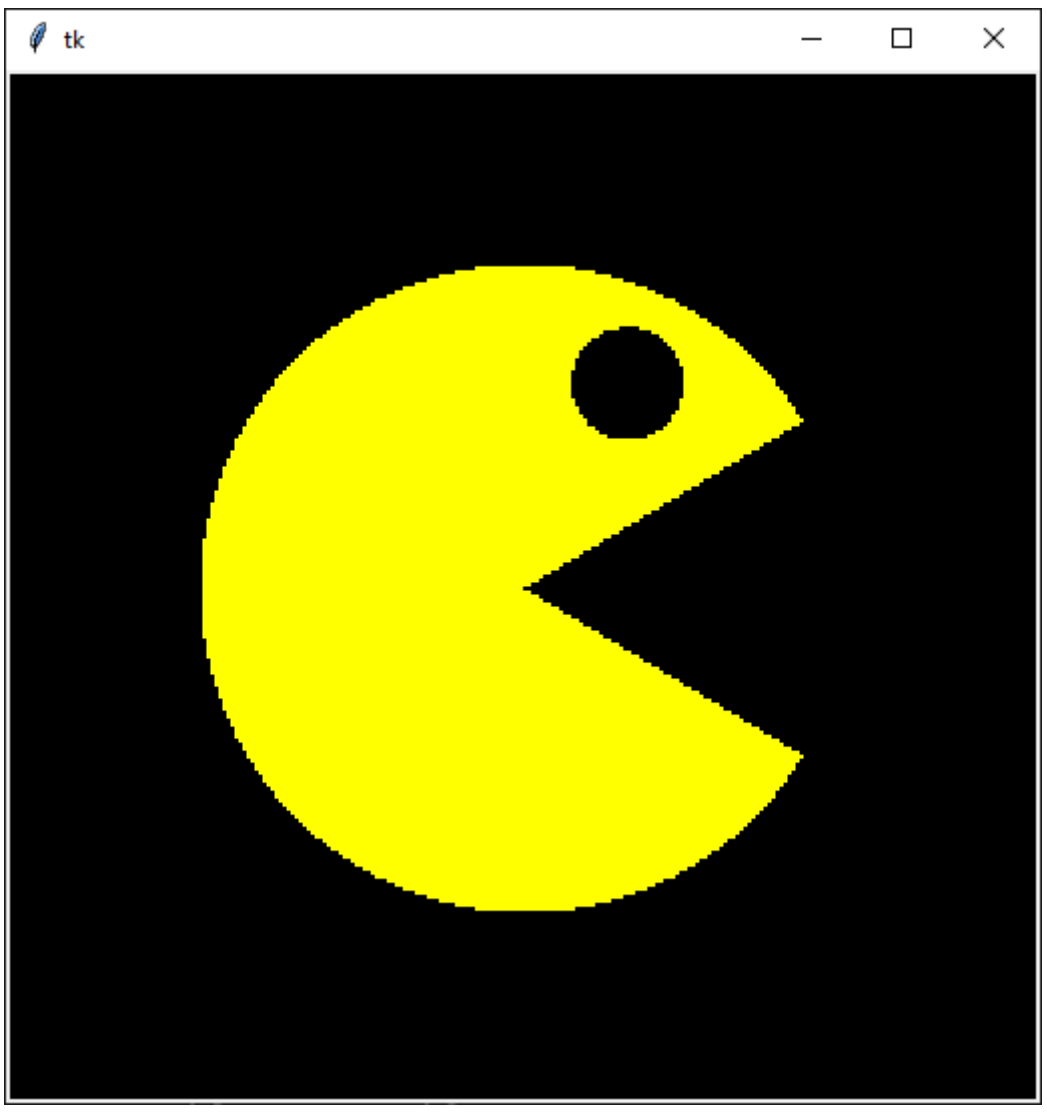
4.2. (уровень сложности: низкий)

Изобразите шар, показанный на примере ниже.



4.3. (уровень сложности: средний)

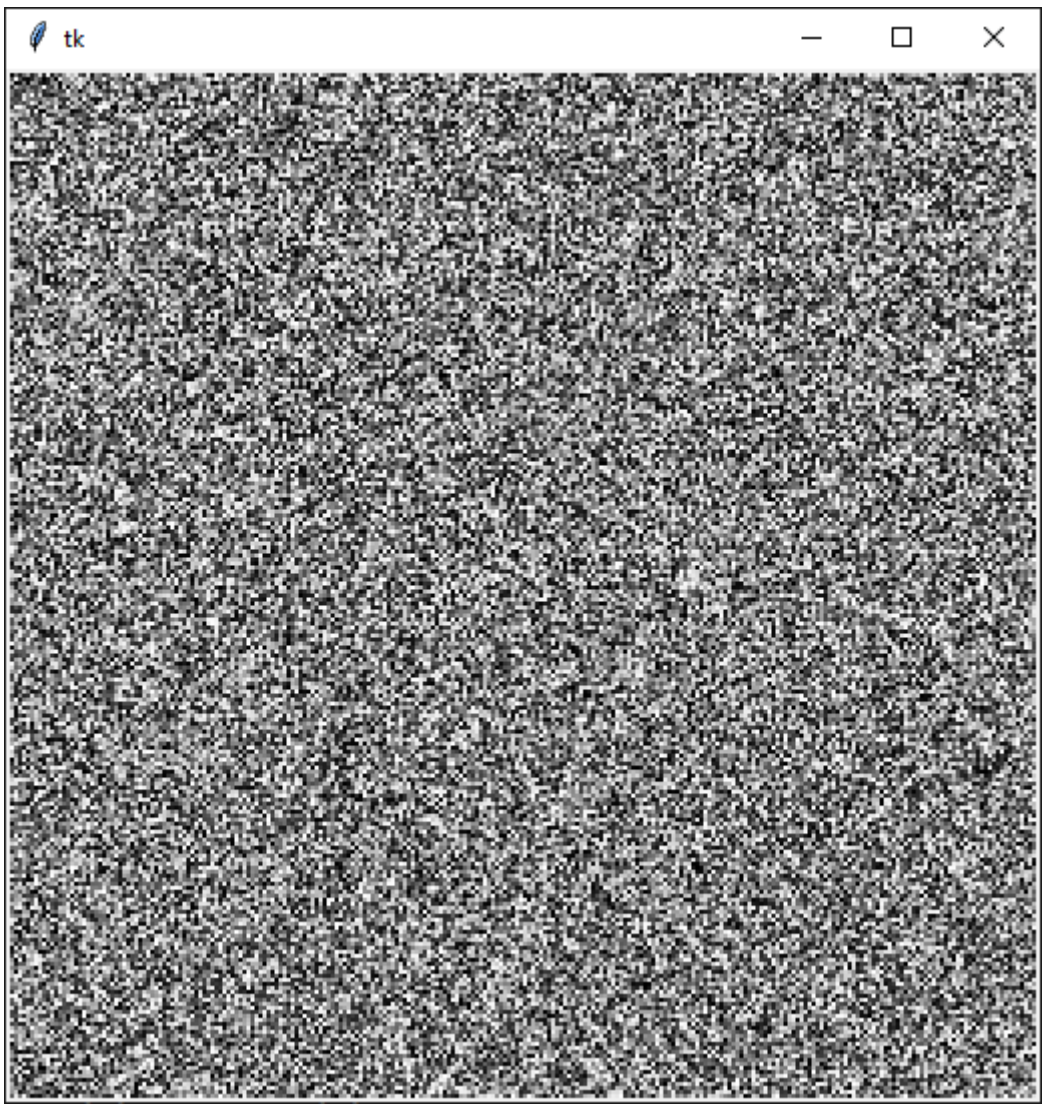
Изобразите знаменитого персонажа компьютерной игры.



4.4. (уровень сложности: низкий)

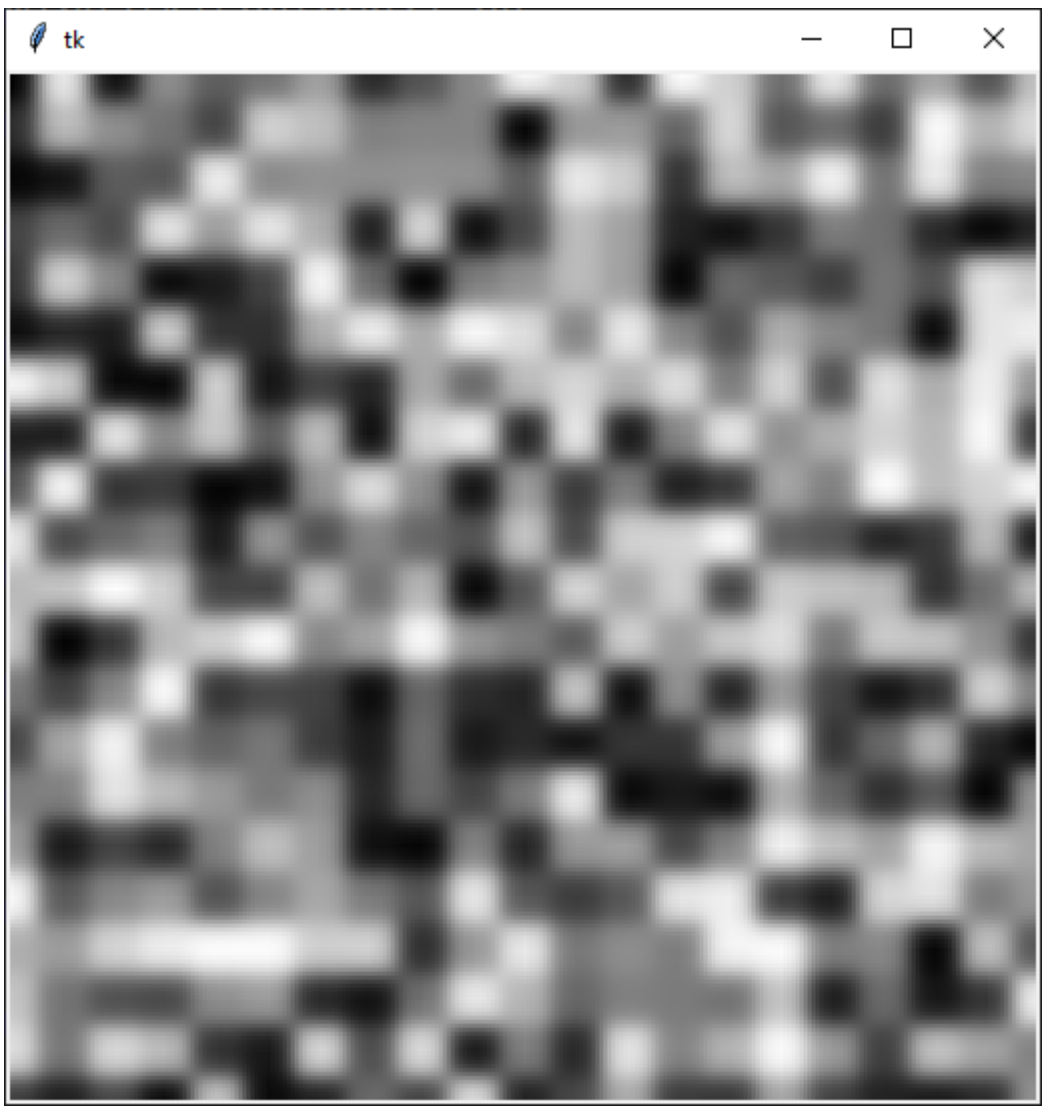
Реализуйте функцию шума noise, которая по заданным координатам выдает случайное значение. Не используйте модуль random. Не

забывайте, что глобальным состоянием пользоваться нельзя.



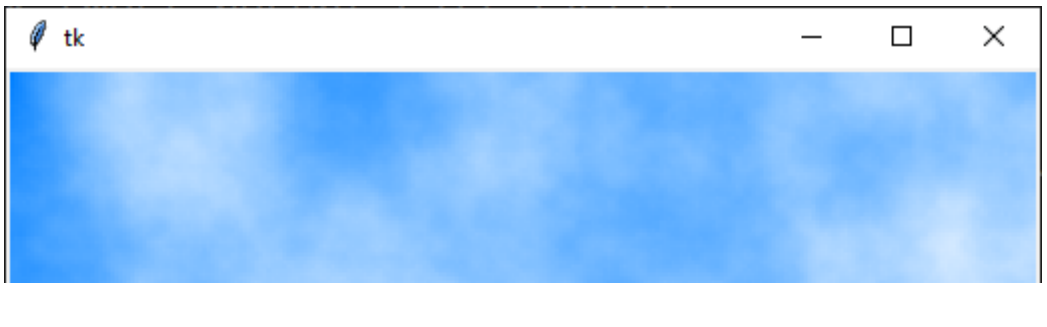
4.5. (уровень сложности: средний)

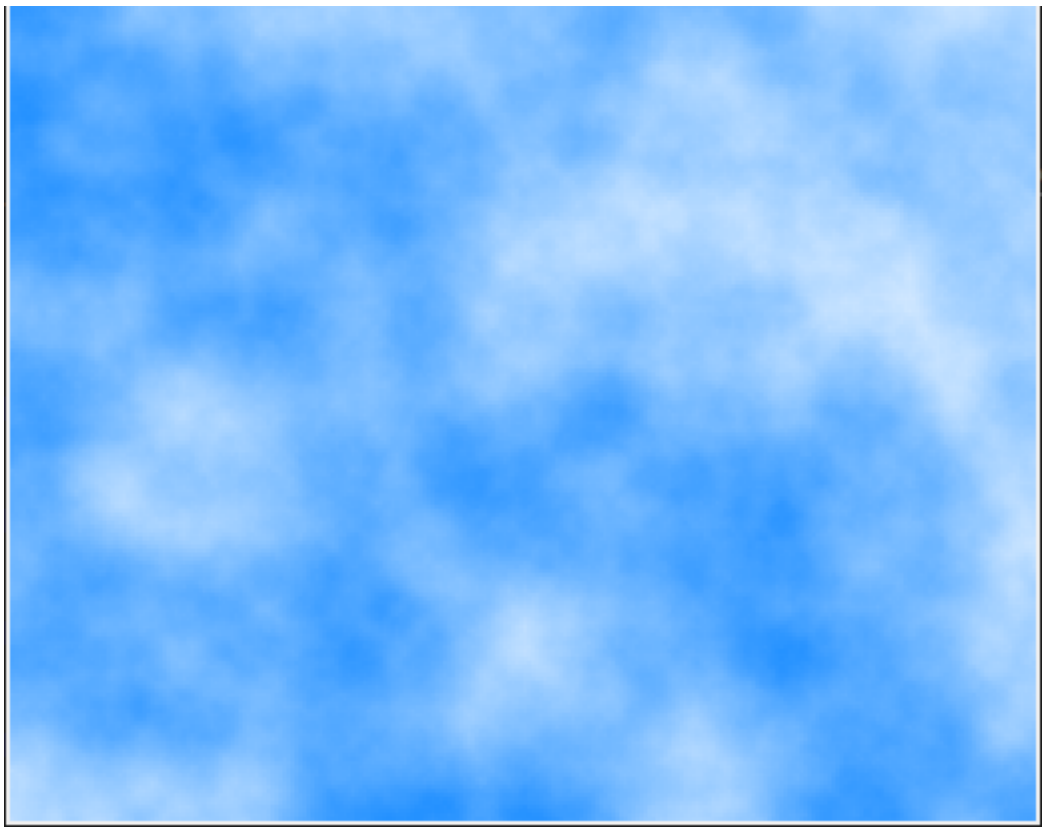
Реализуйте функцию интерполяционного шума val_noise, используя алгоритм value noise (см. источники выше), а также ранее разработанную функцию noise.



4.6. (уровень сложности: средний)

Изобразите облака с помощью алгоритма фрактального шума (fBm noise, см. источники выше) и с использованием ранее разработанной функции val_noise.





4.7. (уровень сложности: хакер)

Реализуйте более серьезную и производительную систему для работы с шейдерами в Питоне. Вот несколько идей для улучшения:

1. Добавьте к координатам переменную t. Это позволит делать анимацию.
2. Улучшите быстродействие системы. Что лучше использовать – модуль multiprocessing, JIT-компилятор Numba или что-то еще?
3. Замените работу с Tkinter на систему с удобным редактором и просмотром, используя PySDL, PyQt или что-то иное.

5. SDF-шейдеры

Этот вариант шейдеров использует функции знаковых расстояний (SDF, Signed Distance Function) и операции конструктивной сплошной геометрии (CSG, Constructive Solid Geometry). Вот простой пример использования на практике связки SDF+CSG для создания моделей и их печати на 3d принтере.

SDF возвращает кратчайшее расстояние от заданной аргументом точки до границы фигуры. При этом положительный (отрицательный) знак расстояния определяет, что точка находится снаружи (внутри) фигуры.

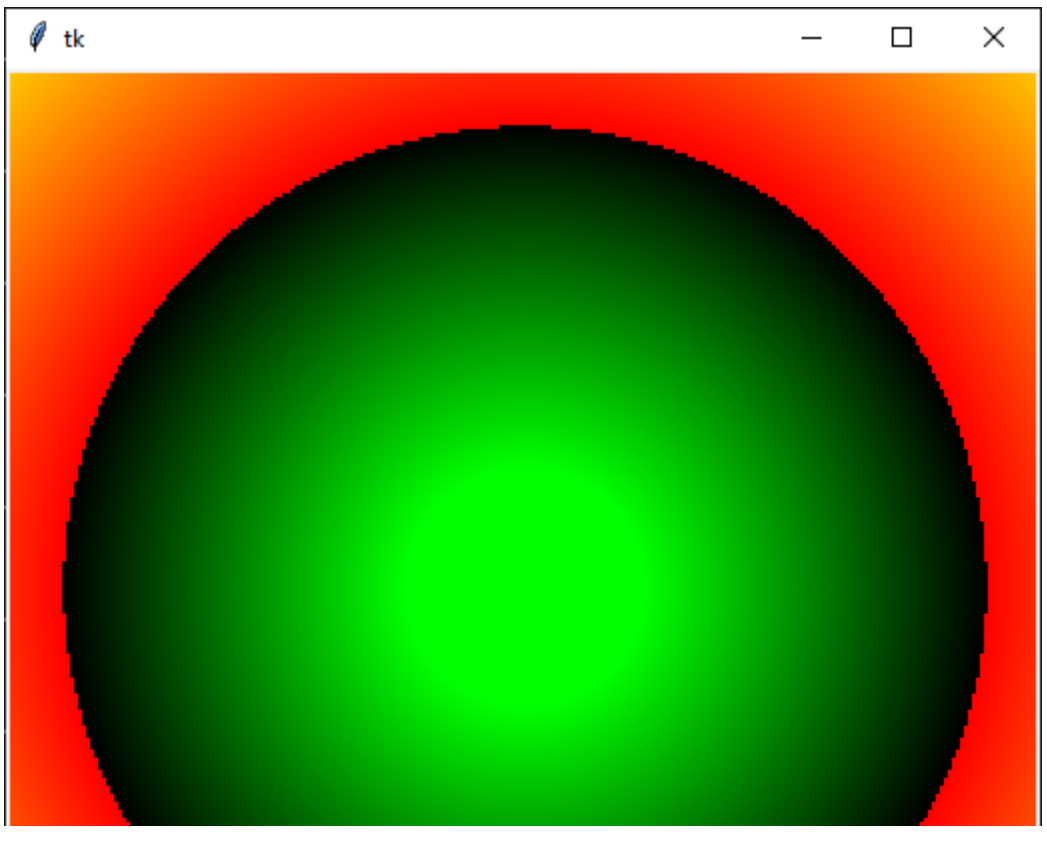
Для решения задач этого раздела необходимо внести изменения в код визуализации шейдеров:

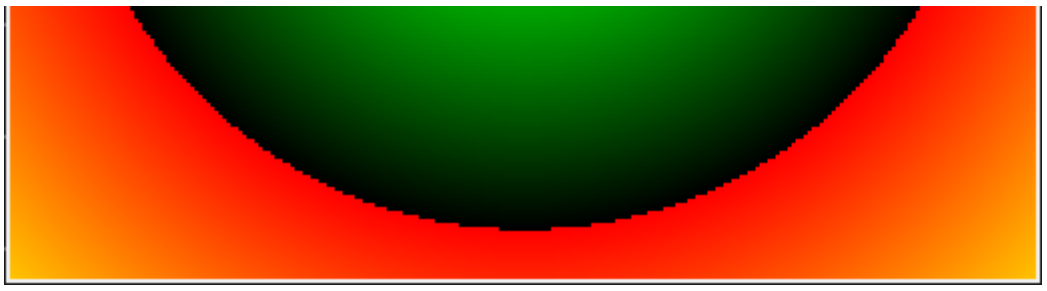
```
def sdf_func(x, y):
    # Ваш код здесь:
    return x


def shader(x, y):
    d = sdf_func(x - 0.5, y - 0.5)
    return d > 0, abs(d) * 3, 0
```

5.1. (уровень сложности: низкий)

Реализуйте функцию circle(x, y, r) и проверьте ее работу на примере:

```
def sdf_func(x, y):
    return circle(x, y, 0.45)
```

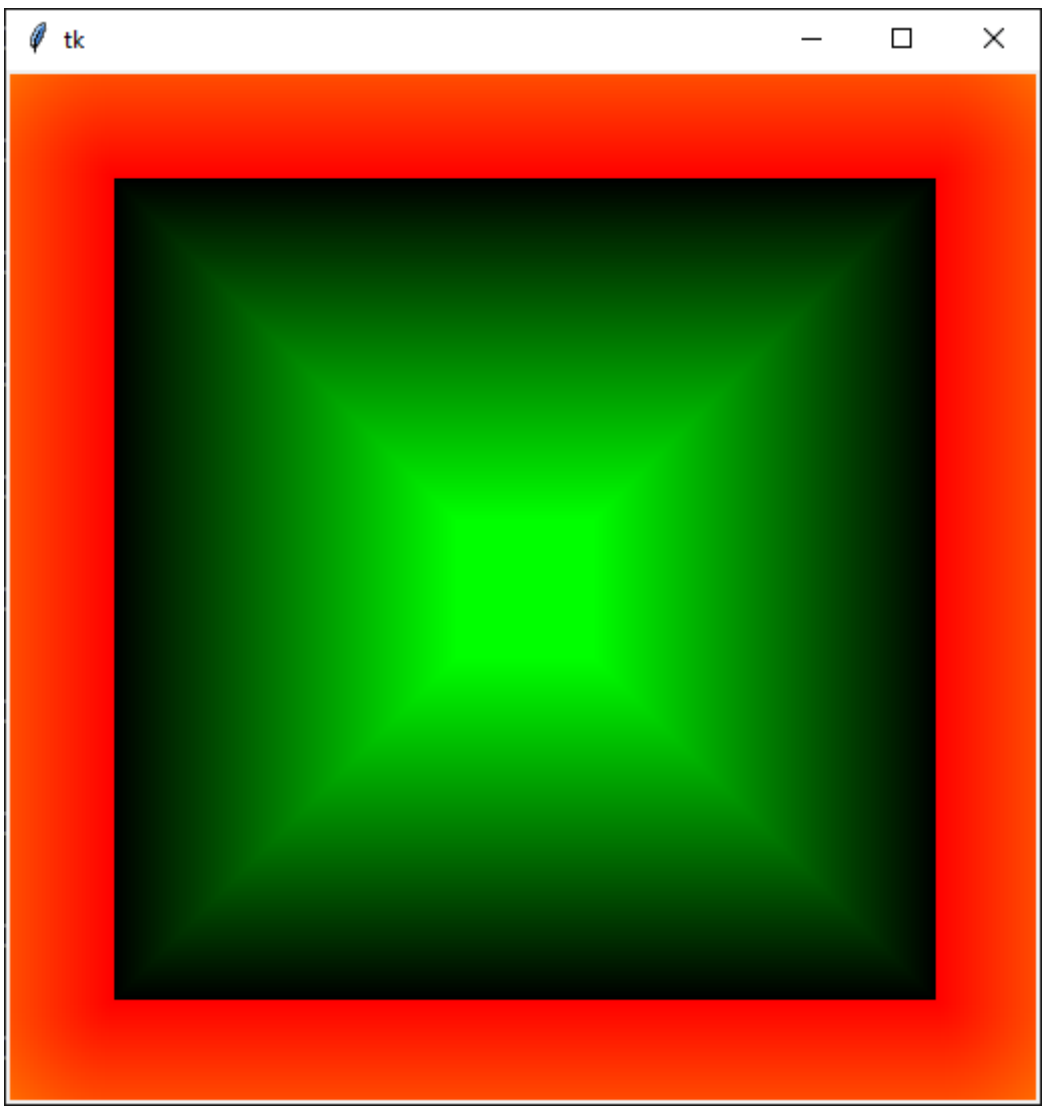




5.2. (уровень сложности: средний)

Реализуйте функцию box(x, y, size) и проверьте ее работу на примере:

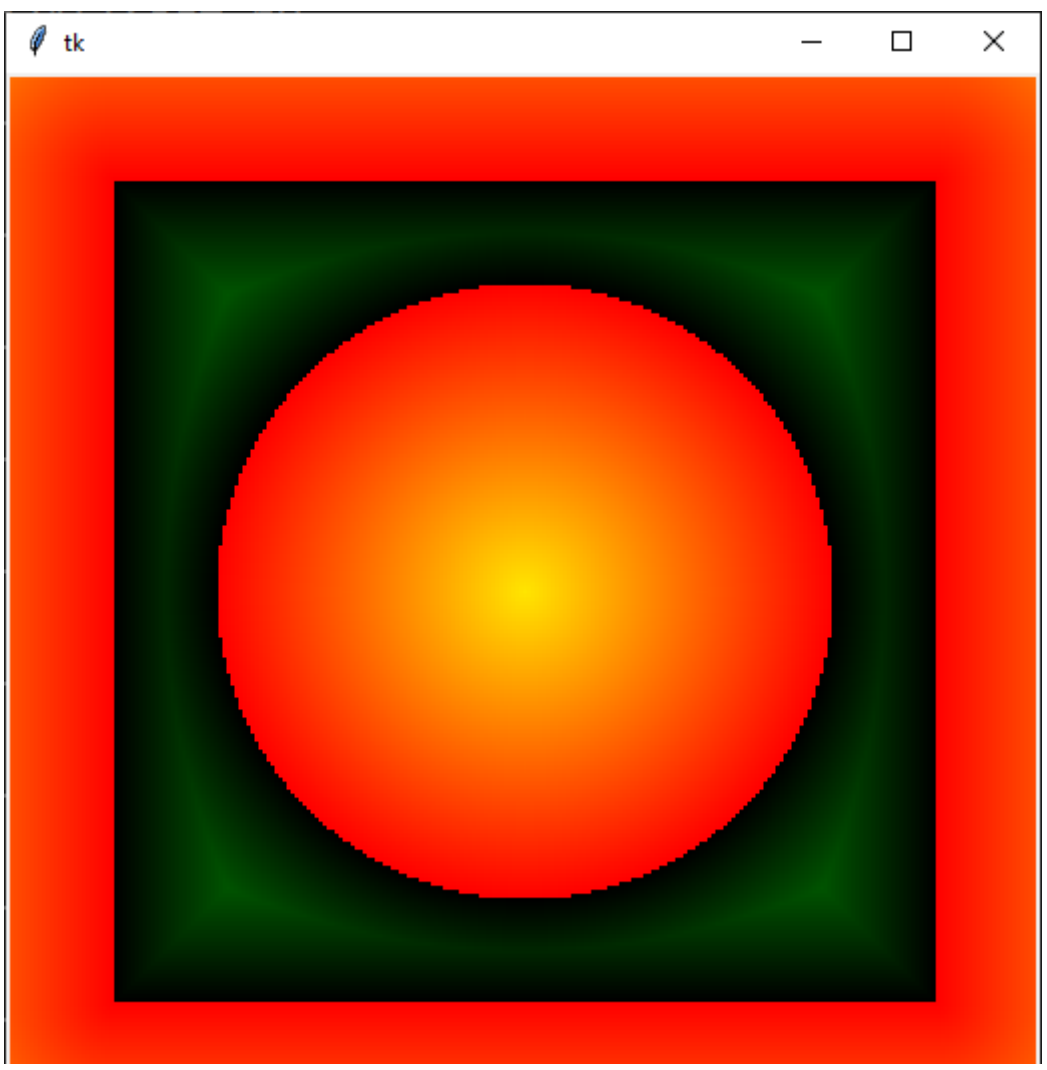
```
def sdf_func(x, y):
    return box(x, y, 0.4)
```



5.3. (уровень сложности: средний)

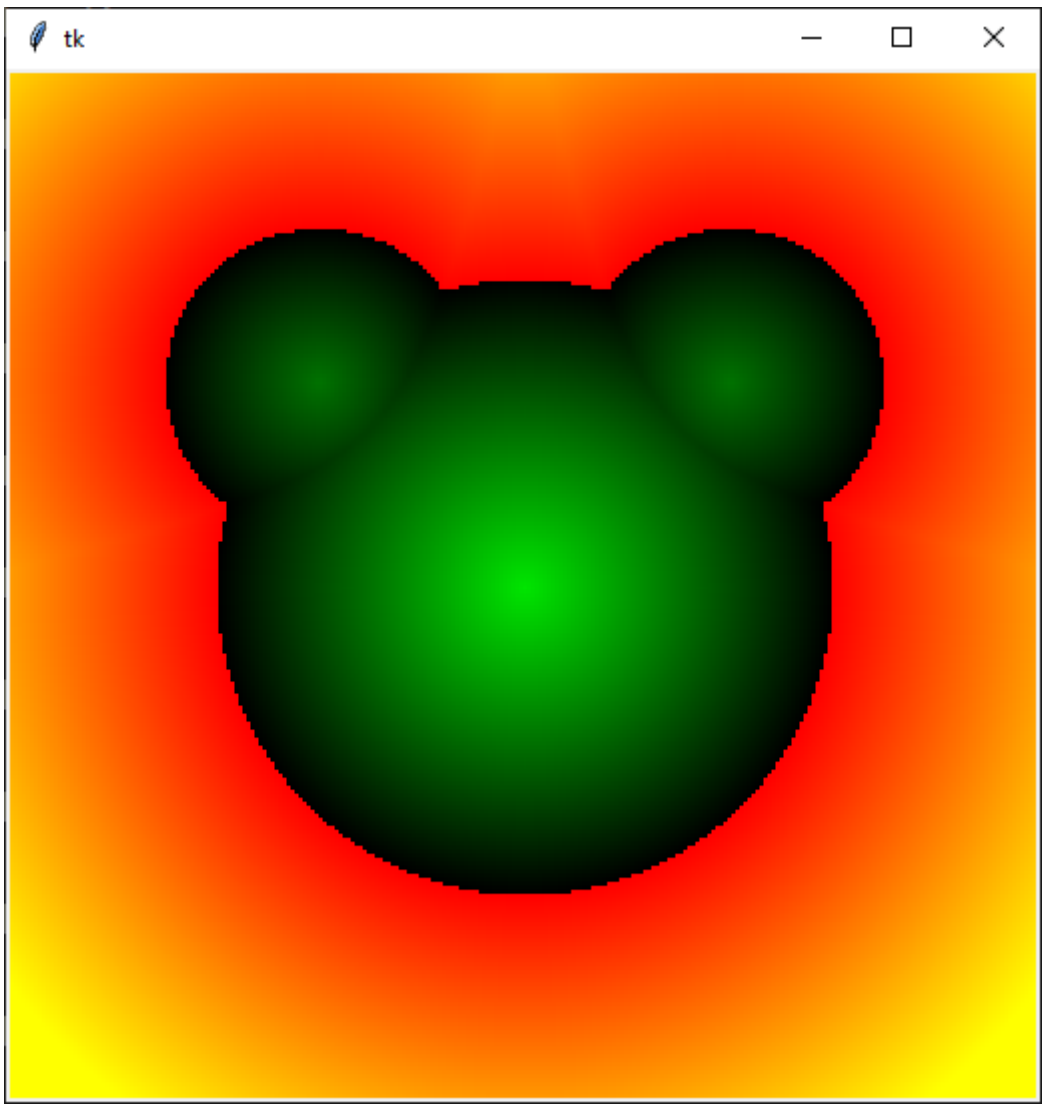
Реализуйте операции над графическими объектами, аналогичные операциям над множествами: union(a, b), intersect(a, b), difference(a, b). Проверьте реализацию на приведенном ниже примере:

```
def sdf_func(x, y):
    return difference(box(x, y, 0.4), circle(x, y, 0.3))
```



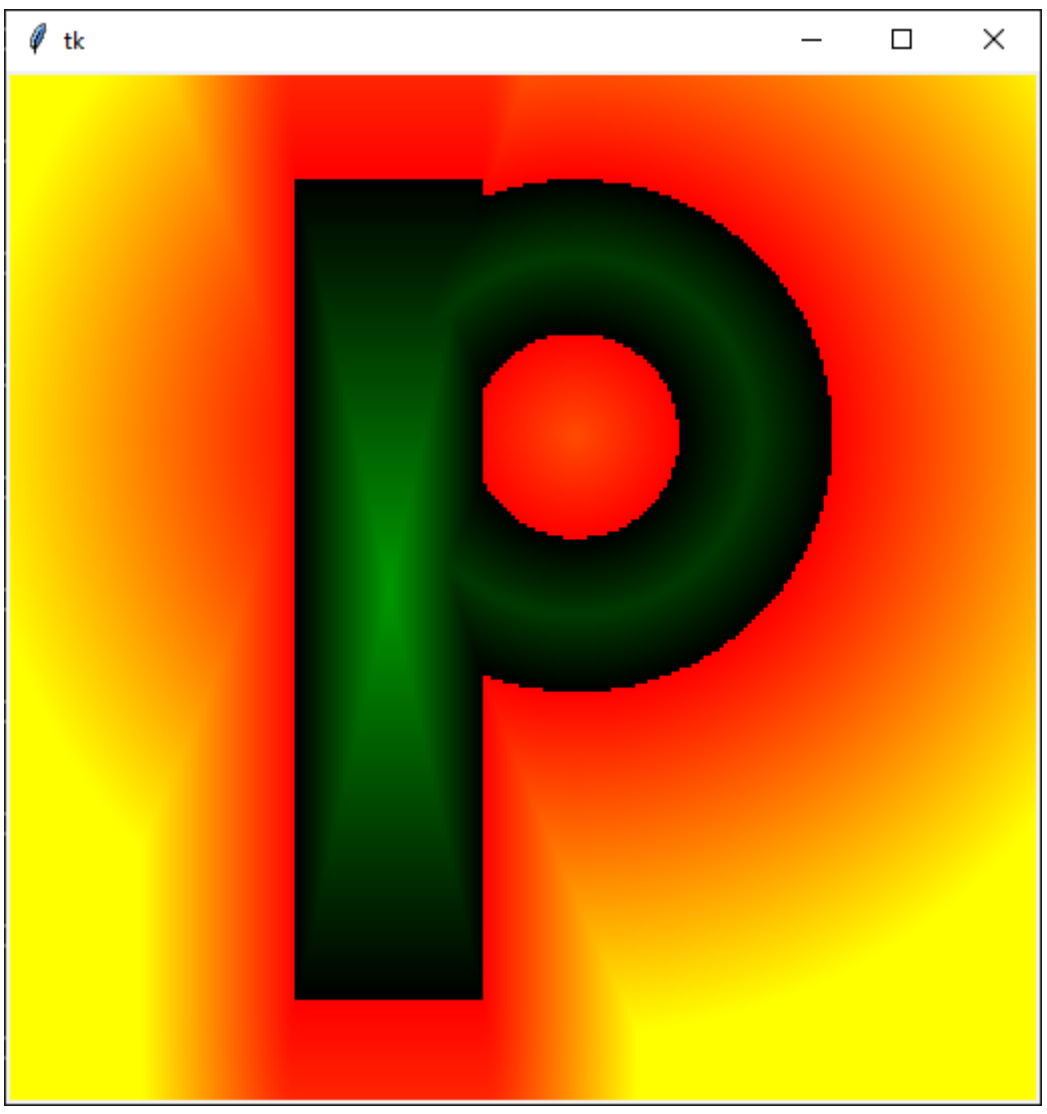
5.3. (уровень сложности: средний)

Изобразите кодом sdf_func следующую картинку:



5.4. (уровень сложности: средний)

Изобразите кодом sdf_func следующую картинку:



6. Алгоритмическая игра DandyBot

(уровень сложности: низкий)

Скачайте DandyBot. Код для своего игрока записывается в файле user_bot.py. Игра запускается с помощью main.py.

Вот простой пример содержимого user_bot.py:

```
def script(check, x, y):
```

```
def script(check, x, y):
    return 'right'
```

Игровая логика записывается исключительно в теле функции script. В нашем случае игрок будет постоянно двигаться вправо.

Полный список действий, которые можно возвращать из функции script, задающей «интеллект» игрока:

- 'up'. Двигаться вверх на клетку.
- 'down'. Двигаться вниз на клетку.
- 'left'. Двигаться влево на клетку.
- 'right'. Двигаться вправо на клетку.
- 'pass'. Ничего не делать.
- 'take'. Взять золото.

Для изучения среды есть функция check:

- check('player', x, y). True, если какой-то игрок в позиции (x, y).
- check('gold', x , y). Если золото в позиции (x, y), то вернуть его количество, иначе вернуть 0.
- check('wall', x, y). True, если стена в позиции (x, y).
- check('level'). Вернуть номер текущего уровня.

Ваша задача — пройти все уровни. Дополнительно устанавливаемыми библиотеками и глобальными данными пользоваться нельзя.

Если игра Вам показалась слишком простой, то попробуйте ее улучшить: упростите код, добавьте новые функции, уровни.