Урок 14. Области видимости переменных. Методические указания

План урока

1. Объекты и переменные: переменная как временный способ доступа к объекту
2. Внешние переменные (и первые отсылки к тому, что разные переменные могут ссылаться на один и тот же объект)
3. Константы
4. Видимость переменных внутри функции
5. Одноименные внешние переменные. Затенение внешних переменных при присваивании.
6. Одноименные переменные в разных функциях.
7. Функция main как способ не загрязнять пространство имён.
8. Модификатор видимости `global`
9. Использование глобальных переменных для продления времени жизни переменной (жизнь между вызовами функции)
10. Аргументы функций как локальные переменные

Аннотация

В уроке речь пойдёт про области видимости переменных. Разбиение программы на функции позволяет сильно упростить программу. Одна из причин упрощения заключается в том, что программисту приходится оперировать только небольшим набором переменных, а не всеми сразу. Для того, чтобы использовать только необходимые данные придуманы области видимости, которые скрывают от одной части программы переменные, используемые в другой части программы. Хотя это делается автоматически, крайне важно понимать, как интерпретатор это делает, иначе неизбежны ошибки. Кроме того изредка вам будет нужно "обойти" эти ограничения и сделать переменную доступную всем частям программы, и на такой случай вам придется познакомиться с глобальными переменными.

Локальные и глобальные переменные

На прошлом занятии мы начали говорить про **аргументы** функций. Пользоваться ими вроде бы просто, но за кажущейся простотой скрывается много тонкостей:

- с какими переменными может работать функция?
- что если в двух функциях переменные называются одинаково?
- может ли функция изменить значение аргумента? Есть и много других вопросов. Занятие будет посвящено в первую очередь переменным, и лишь во вторую — функциям. Эти знания нечасто нужны для написания кода, но совершенно необходимы для понимания того, как программа работает.

Это поможет вам не гадать, как ведет себя программа, и значительно сократит время, которое вы тратите на отладку.

Давайте для начала вспомним, как писать функции.

Решите простую задачу: Встаньте стройными рядами (https://lms.yandexlyceum.ru/task/view/1427)

Вернемся к уже рассмотренной на прошлом занятии функции, которая должна выводить на экран содержимое списка, печатая каждый элемент на своей строчке. Посмотрим на переданный аргумент функции.

```
def print_array(array):
    for element in array:
        print(element)


print_array(['Hello', 'world'])
print_array([123, 456, 789])
```

```
Hello
world
123
456
789
```

Переменная `array` будет **локальной**. Она существует только во время выполнения (иногда говорят "жизни") функции. Можно сказать, что имя аргумента функции — это **временное** имя для переданного в функцию значения.

Мы создали список `['Hello', 'world']` и передали его в функцию. Список существует независимо от того, есть ли у него имя, или нет, но если у объекта нет имени, мы не можем с ним работать. Поэтому у переданного списка временно появляется имя — в тот самый момент, когда функция начинает работу. Когда функция прекращает работу, это имя исчезает.

В вышеописанной программе `array` не существует вне функции.

Что интересно, у каждого объекта в программе может быть несколько имен. Например, слегка модифицируем программу:

```
def print_array(array):
    for element in array:
        print(element)


words = ['Hello', 'world']
print_array(words)
```

```
Hello
world
```

Теперь у списка есть имя `words`. Но в момент выполнения функции у этого списка существует сразу два имени: `words` (имя из «внешнего мира», из **глобальной области видимости**) и `array` — локальное имя. В функции можно использовать любое из них, но внешнее — крайне не рекомендуется. Давайте перепишем программу, используя внешнее имя, и разберемся, почему так делать не стоит:

```
def print_array(array):
    print('id(array) = ', id(array))
    print('id(words) = ', id(words))
    for element in array:
        print(element)


words = ['Hello', 'world']
print_array(words)
```

```
id(array) =  4535527944
id(words) =  4535527944
Hello
world
```

```
def print_array(array):
    for element in words:
        print(element)


words = ['Hello', 'world']
print_array(words)
```

```
Hello
world
```

Пока всё работает аналогично предыдущему примеру. А теперь выполним такую команду:

```
print_array(['abc', 'def', 'ghi'])
```

```
Hello
world
```

Мы рассчитывали, что будут распечатаны три строки: `abc`, `def` и `ghi`, но вместо этого снова распечатались `Hello` и `world`, которые находятся в списке `words`. Если ваша функция написана так, то чтобы поменять её поведение, придётся не просто передать ей список, а перезаписать используемую переменную:

```
words = ['abc', 'def', 'ghi']
```

Еще один минус: раньше вы могли прочитать три строки функции `print_array` и точно понять, как она будет работать. Теперь вам необходимо прочитать всю программу (а в большой программе могут быть тысячи строк), чтобы разобраться, как ведет себя эта функция — а всё потому что она теперь не самодостаточна, а зависит от внешней переменной.

Всё, что вы сейчас узнали про использование внешних переменных в функции, необходимо в первую очередь для поиска ошибок. Если программа ведёт себя странно, проверьте, не используете ли вы где-то внешние (глобальные) переменные.

Повторим: использовать в своих программах эту возможность языка стоит как можно реже.

К внешним переменным прибегать допустимо, если переменная — **константа**. Константы обычно пишутся большими буквами, чтобы не перепутать их с обычными переменными. Давайте напишем функцию, которая по номеру печатает соответствующий цвет радуги:

```
ENGLISH_RAINBOW_COLORS = [
    'red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'
]
RUSSIAN_RAINBOW_COLORS = [
    'красный', 'оранжевый', 'желтый', 'зеленый', 'голубой', 'синий',
    'фиолетовый'
]


def rainbow_color(index, russian_or_english):
    if russian_or_english == 'russian':
        print(RUSSIAN_RAINBOW_COLORS[index])
    elif russian_or_english == 'english':
        print(ENGLISH_RAINBOW_COLORS[index])
    else:
        print('Неверный язык')


rainbow_color(2, 'russian')
rainbow_color(2, 'english')
```

```
желтый
yellow
```

Эта функция будет работать нормально: предполагается, конечно, что в ходе работы программы никому не понадобится менять константы (на то они и константы).

Заметьте, что мы не передаем эти константы в функцию как аргументы, а просто пользуемся ими. Они есть в нашем распоряжении, потому что определены на самом верхнем уровне.

Однако, если константы надо использовать лишь в одной конкретной функции, то лучше **внести** их в эту функцию. Тогда их точно никто не испортит.

```
def rainbow_color(index, russian_or_english):
    ENGLISH_RAINBOW_COLORS = ['red', 'orange',
                              'yellow', 'green', 'blue', 'indigo', 'violet']
    RUSSIAN_RAINBOW_COLORS = ['красный', 'оранжевый',
                              'желтый', 'зеленый', 'голубой', 'синий', 'фиолетовый']
    if russian_or_english == 'russian':
        print(RUSSIAN_RAINBOW_COLORS[index])
    elif russian_or_english == 'english':
        print(ENGLISH_RAINBOW_COLORS[index])
    else:
        print('Неверный язык')

rainbow_color(2, 'russian')
```

```
желтый
```

Области видимости

Давайте поговорим про область видимости переменных немного подробнее.

Сформулируем общее правило, касающееся видимости переменных:

1. Внутри функции видны все переменные этой функции (локальные переменные и аргументы функции).
2. Также внутри функции видны переменные, которые определены снаружи этой функции.
3. Снаружи не видны **никакие** переменные, которые определены внутри функции.

Принцип такой же, как в машине с тонированными стеклами: изнутри видны все наружние объекты (и внутренние, конечно же), а вот снаружи посмотреть в машину не получается.

Но вот вопрос: что же произойдет, если имена переменных в функции и во внешней программе совпадут?

Оказывается, возможны два разных варианта.

Первый вариант мы уже видели: это ситуация, когда внутри функции используется внешняя переменная, и это ровно та же внешняя переменная, что была снаружи. В следующем примере - это переменная `Pi`.

```
def print_circle_length(radius):
    perimeter = 2 * Pi * radius
    print('Длина окружности: ', perimeter)


Pi = 3.14
print_circle_length(10)
```

```
Длина окружности:  62.800000000000004
```

Второй вариант — это ситуация, когда внутри функции используется переменная с тем же именем, что и в основной программе, но перед использованием ей присваивается новое значение.

```
area = 'Красная площадь'


def print_square_area(length, width):
    area = length * width
    print('Площадь площади: ', area)


print('Место встречи: ', area)
print_square_area(330, 75)
print('Повторяю, место встречи: ', area)
```

```
Место встречи:  Красная площадь
Площадь площади:  24750
Повторяю, место встречи:  Красная площадь
```

Если попытаться мысленно выполнить программу, кажется, что функция должна **испортить** внешнюю переменную `area`, записав в неё вместо строки «Красная площадь» число, равное площади прямоугольника со сторонами `length` и `area`. Но этого не происходит. Когда после выполнения функции мы повторяем печать, то оказывается, что переменная `area` все так же содержит строку «Красная площадь», хотя внутри функции там определенно было число.

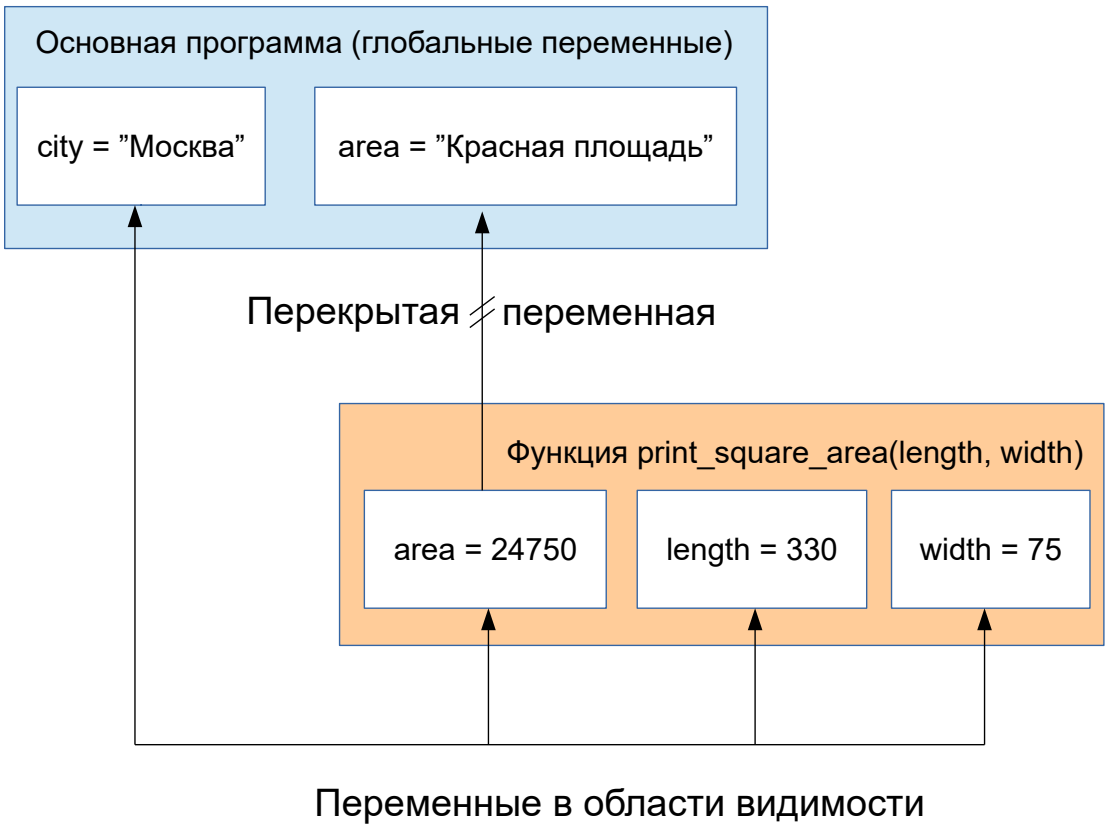
Что же произошло?

Оказывается, что `area` внутри функции и снаружи — это две **совершенно разные** переменные. Если внутри функции переменной что-то присваивается (в любом месте функции), то интерпретатор не позволит вам работать с внешней переменной.

Вместо этого он создает **новую** переменную с тем же именем и присваивает значение уже ей. При этом к внешней переменной вы обратиться уже не можете: внутри функции она потеряла своё имя.

Как иногда говорят, внутренняя переменная с тем же именем её перекрыла или затенила (на английском языке это называется `variable shadowing`).

На рисунке представлена визуализация описанного принципа.



Переменные в области видимости

Это сделано для того, чтобы обезопасить вас от ошибок. Если бы функция могла менять внешние переменные, то в программе происходило бы много странных и непонятных вещей.

Например, представьте, что вы написали функцию, которая вычисляет площадь и записывает что-то в переменную `area`. При этом в основной программе переменной `area` у вас не было, поэтому всё работало хорошо. Потом вы в какой-то момент ввели переменную `area` — теперь функция, которая работала хорошо и безопасно, стала менять значение `area` во внешней программе. Вам бы пришлось долго искать в программе ошибку, из-за которой вы встречаетесь на площади 24750.

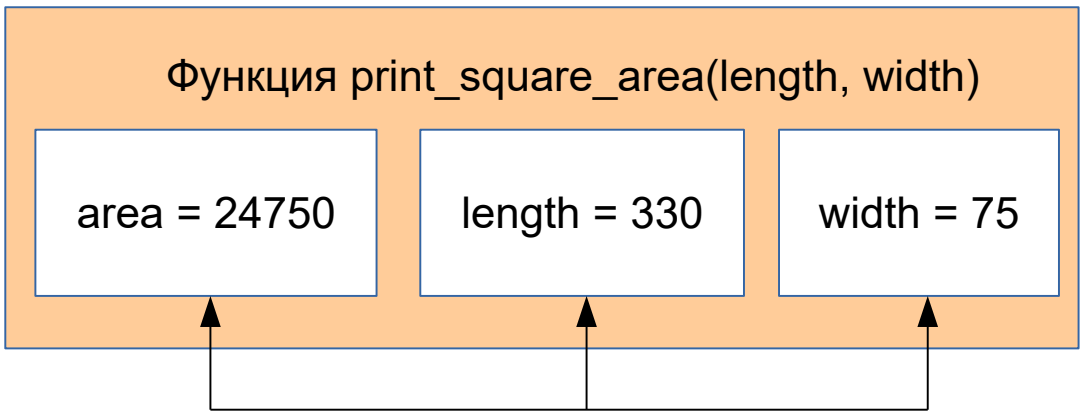
Именно поэтому области видимости так важны. Они позволяют изолировать небольшой кусочек программы, в котором ваши изменения на что-то влияют. Благодаря областям видимости не окажется так, что вы поменяли строку в одной функции, а сломалась другая.

Для того, чтобы не вносить в программу лишние глобальные переменные, часто весь код со внешнего уровня программы (т.е. всё, что не находится внутри функции) переносят в отдельную функцию `main`. После этого остается только вызвать её. Все переменные, которые были глобальными, таким образом, становятся локальными для функции `main` и не загрязняют область видимости других функций.

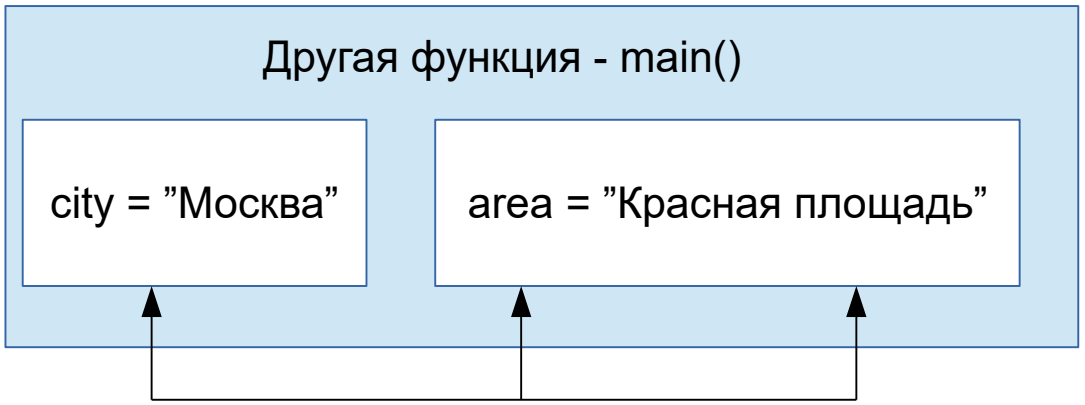
```
def print_square_area(length, width):
    area = length * width
    print('Площадь площади: ', area)


def main():
    area = 'Красная площадь'
    print('Место встречи: ', area)
    print_square_area(330, 75)
    print('Повторяю, место встречи: ', area)


main()
```

```
Место встречи:  Красная площадь
Площадь площади:  24750
Повторяю, место встречи:  Красная площадь
```

Снова посмотрим на визуализацию прицнипа выделения функции `main`:



Переменные в области видимости



Отдельная область видимости

Использование глобальных переменных

Впрочем, если у вас есть глобальная переменная и вам зачем-то очень нужно повлиять на эту внешнюю переменную, это можно сделать. Для этого перед тем как присваивать внутри функции этой переменной какое-то значение, нужно указать, что она глобальная, т.е. относится к внешней области видимости.

Например, напишем функцию, которая при вызове будет обновлять счетчик, считающий, сколько раз вызвана эта функция.

```
ask_number = 0


def ask_again():
    global ask_number
    ask_number = ask_number + 1
    print('Ты спрашиваешь меня уже в ', ask_number, '-й раз', sep='')


ask_again()
ask_again()
ask_again()
```

```
Ты спрашиваешь меня уже в 1-й раз
Ты спрашиваешь меня уже в 2-й раз
Ты спрашиваешь меня уже в 3-й раз
```

Хотя переменной `ask_number` и присваивается значение, она является внешней переменной за счет строчки `global ask_number`.

Мы уже не раз говорили, но повторим еще раз: про глобальные переменные нужно помнить и знать, но использовать их в своих программах крайне не рекомендуется.

Менять внутри функции значение внешней переменной еще хуже, чем использовать внешние переменные, не изменяя их. Старайтесь писать функции которые минимально зависят от всего, что происходит снаружи.

Но давайте теперь немного поэкспериментируем с нашей программой. Уберем строку, которая делала переменную глобальной. Когда мы запустим программу, интерпретатор выдаст ошибку:

```
ask_number = 0


def ask_again():
    ask_number = ask_number + 1
    print('Ты спрашиваешь меня уже в ', ask_number, '-й раз', sep='')


ask_again()
ask_again()
ask_again()
```

```
---------------------------------------------------------------------------
-
UnboundLocalError                         Traceback (most recent call las
t)
<ipython-input-15-1733b3dfd088> in <module>()
      7
      8
----> 9 ask_again()
     10 ask_again()
     11 ask_again()

<ipython-input-15-1733b3dfd088> in ask_again()
      3
      4 def ask_again():
----> 5     ask_number = ask_number + 1
      6     print('Ты спрашиваешь меня уже в ', ask_number, '-й раз', sep=
'')
      7

UnboundLocalError: local variable 'ask_number' referenced before assignmen
t
```

В переводе на русский язык это означает, что вы ссылаетесь на локальную переменную `ask_number`, прежде чем ей присвоено значение. Это происходит потому, что присваивание сделало переменную локальной.

Но в конструкции `ask_number = ask_number + 1` сначала выполняется правая часть, в которой мы пытаемся эту локальную переменную использовать.

Однако использовать её пока нельзя, так как она к этому моменту еще не существует. Как вы должны помнить, локальные переменные создаются в момент присваивания, и до этого использовать их нельзя.

Одна из немногих причин использовать глобальные переменные — возможность сохранить какие-то данные между вызовами функции. Функция не может ничего сохранить «на будущее» в локальных переменных поскольку они исчезнут после того как интерпретатор вернется из функции. Зато, если функция при первом вызове сохранит что-либо в глобальную переменную, то при следующих вызовах эти данные будут доступны для чтения (и для изменения). Но не забудьте, что глобальная переменная может быть изменена не только из этой функции, но и в любом другом месте программы, так что вы должны внимательно следить за тем, чтобы значение вашей переменной **не испортилось** между вызовами функции.

Задачи:

- Заикание (https://lms.yandexlyceum.ru/task/view/1421),
- Несвежие анекдоты (необязательная задача) (https://lms.yandexlyceum.ru/task/view/1428)

Домашние задачи:

- НРЗБРЧВ (https://lms.yandexlyceum.ru/task/view/1423),
- Я вас знаю (https://lms.yandexlyceum.ru/task/view/1424)

Аргументы функций как локальные переменные

Внимательный читатель мог заметить, что мы не рассмотрели еще один случай (а можно сказать, даже два) использования переменных с одинаковыми именами. Это случай, когда имя локальной переменной совпадает с именем **аргумента функции** и случай, когда имя внешней переменной совпадает с именем аргумента.

Первая программа:

```
def greet(name):
    print("Привет,", name)
    name = 'товарищ'
    print("Здравствуй,", name)


greet('Вася')
```

```
Привет, Вася
Здравствуй, товарищ
```

Вторая программа:

```
name = 'Петя'


def greet(name):
    print("Привет,", name)


greet('Вася')
```

```
Привет, Вася
```

Работу обеих программ очень легко объяснить. Они выполняются именно так, потому что аргумент функции — по сути обыкновенная локальная переменная. В момент вызова функции происходит присваивание значения «Вася» этой локальной переменной. А дальше программист волен её использовать и изменять, как ему вздумается.

Аргумент функции всегда является локальной переменной, а значит будет иметь приоритет над внешней переменной с тем же именем.

Дополнительная задача: Простое перемешивание (https://lms.yandexlyceum.ru/task/view/1429)

Рассмотрим еще одну дополнительную задачу "Физическое моделирование".

Физики часто используют компьютерное моделирование, чтобы понять, как протекает физический процесс. Для этого они описывают физическую систему несколькими переменными, которые изменяются во времени по некоторому закону. Мы будем моделировать очень простую систему: движение камня, который просили под углом 45° к горизонту. На камень вдоль оси Y действует только сила тяжести. Конечно, из механики вам известно, что движение будет по параболе, но мы смоделируем этот процесс напрямую из законов движения по методу Эйлера.

Мы знаем, что на камень действует ускорение свободного падения вдоль вертикальной оси.

$a_x = 0$; $a_y = -g = -9.81$ м/с^2;

Ускорение остается постоянным в течение все времени, пока камень не столкнется с землей. Оно будет изменять скорость. За время Δt скорости изменятся на величины:

$\Delta v_x = a_x \Delta t$; $\Delta v_y = a_y \Delta t$

Давайте теперь предположим, что за небольшой промежуток времени Δt скорости не успевают измениться. Тогда на этом коротеньком промежутке тело двигается равномерно прямолинейно, а координаты изменятся на величины:

$\Delta v_x = v_x \Delta t$; $\Delta v_y = v_y \Delta t$

Получается, что мы разбили движение на отрезки, которые тело проходит с постоянной скоростью и по прямой. Но если отрезки сделать очень маленькими, то окажется, что на каждом отрезке камень действительно не успевает сильно изменить ни скорость, ни направление. В итоге отрезки выстраиваются в траекторию, которая очень близка к тому, как тело будет двигаться на самом деле.

Чтобы окончательно решить задачу нам остается указать координаты и скорости в начальный момент. Мы зададим координаты нулевыми, а проекции скорости вдоль осей x и y — равными 10 м/с^2 (бросок под углом 45°).

Пересчитывать скорости и координаты мы будем каждые 0,05 с. Чтобы получить еще более плавное движение можно взять интервал времени поменьше.

Вот простенькая программа для вычисления траектории камня:

```
dt = 0.05  # Шаг по времени
ax, ay = 0, -9.81  # Ускорения вдоль осей
x, y = 0, 0  # Начальные координаты
vx, vy = 10, 10  # Начальные скорости

while y >= 0:
    # Обновляем скорости
    vx += ax * dt
    vy += ay * dt
    # Обновляем координаты
    x += vx * dt
    y += vy * dt
    # Точки траектории печатаем округленными до 3 знака
    print(round(x, 3), round(y, 3))
```

```
0.5 0.475
1.0 0.926
1.5 1.353
2.0 1.755
2.5 2.132
3.0 2.485
3.5 2.813
4.0 3.117
4.5 3.396
5.0 3.651
5.5 3.881
6.0 4.087
6.5 4.268
7.0 4.425
7.5 4.557
8.0 4.665
8.5 4.748
9.0 4.806
9.5 4.84
10.0 4.85
10.5 4.835
11.0 4.795
11.5 4.731
12.0 4.642
12.5 4.529
13.0 4.392
13.5 4.23
14.0 4.043
14.5 3.832
15.0 3.596
15.5 3.336
16.0 3.051
16.5 2.741
17.0 2.408
17.5 2.049
18.0 1.666
18.5 1.259
19.0 0.827
19.5 0.37
20.0 -0.111
```

Ваша задача — завести отдельную функцию `updateCoordinates()`, которая будет обновлять значения координат и скоростей. Перепишите эту программу в трех разных вариантах:

- В первом варианте x и y, vx и vy должны быть отдельными переменными, как это сделано сейчас. Однако они должны обновляться внутри функции.
- Во втором варианте координаты и скорости должны быть записаны в два списка, а функция должна обновлять эти списки:

```
coords = [x, y]
velocities = [vx, vy]
```

Напомним, что кортежи отличаются от списков тем, что они неизменяемы.

- В третьем варианте x и y, vx и vy должны быть записаны в два кортежа, а функция должна обновлять эти кортежи:

```
coords = (x, y)
velocities = (vx, vy)
```

Объясните, чем третий вариант принципиально отличается от первых двух.</td>

Задача: Длинный чек (https://lms.yandexlyceum.ru/task/view/1422)

Дополнительная задача: Азбука Морзе (https://lms.yandexlyceum.ru/task/view/1430)

На следующем занятии нам придется еще немного поговорить о переменных. Вы узнаете, что переменная и её значение — это не одно и то же. А также поймёте, что происходит с результатами вычислений, которые больше не нужны. Через занятие мы начнем изучать функции с возвращаемыми значениями. Это даст вам мощнейший инструмент для написания сложных программ из очень простых кусочков.