Пошаговое выполнение программ

Сегодня мы поговорим о том, как эффективнее находить ошибки в коду программы. Пусть у вас есть тест, на котором программа не работает, и вы хотите найти ошибки в вашем коде. В такой ситуации пристальное изучение кода помогает не всегда.

1 «Метод утенка»

Существует так называемый «метод утенка», который иногда помогает находить ошибки в программе, даже если нет теста, на котором программа не работает. Часто, когда вы объясняете вашему другу, родителю или преподавателю вашу программу строку за строкой, рассказываете что вы хотели сделать, вы сами находите ошибку в логике или коде, вашему собеседнику даже не нужно ничего говорить или понимать. Отсюда родился метод утенка, про него даже есть соответствующая статья в Википедии. Метод заключается в том, чтобы посадить игрушку рядом с собой (в оригинале, желтого утенка), и рассказать ему весь ваш код, что вы делаете и как вы это делаете. И этот утенок, зачастую, помогает найти ошибку.

2 Дебагер в Pycharm

Давайте вернемся к компьютеру как помощнику в решении задач. Если вы используете среду разработки Pycharm, то вы можете воспользоваться встроенным *дебагером* – утилитой для пошагового выполнения вашей программы. В других средах разработки тоже часто есть подобные средства. Мы же рекомендуем пользоваться Pycharm, как одной из самых удобных сред разработки для начинающих, поэтому покажем на её примере.

Пусть мы написали код, который работает не совсем правильно. Задача, которую мы решаем, заключается в том, чтобы для всех чисел от a до b вывести все делители в отсортированном порядке; делители каждого числа должны идти на своей строке:

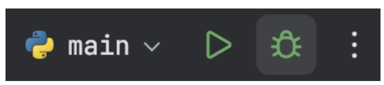
```python
from math import sqrt

def find_divisors(x):
    divisors = []
    for d in range(2, int(sqrt(x))):
        if x % d == 0:
            divisors.append(d)
            if d != x / d:
                divisors.append(x / d)
    return divisors

a, b = map(int, input().split())
for x in range(a, b+1):
    divs = find_divisors(x)
    print(*sorted(divs))
```

Продемонстрируем некоторые возможности Pycharm для поиска ошибок.

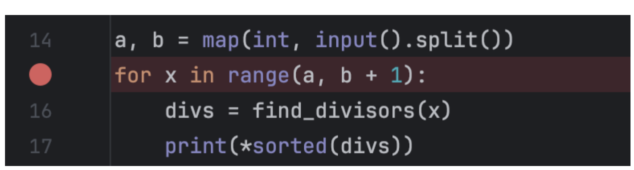
Раньше вы запускали вашу программу через треугольничек, или комбинацию клавиш `shift + f10`, но рядом есть еще один способ запуска программы, запуск в дебаге, который обозначен зеленым жуком:



Кнопка запуска с использованием дебагера обозначается жуком

Для начала запустим код как обычно, через треугольник, попросим вывести делители чисел от 1 до 100. Если вы запустите этот код, то увидите, что пока не всё работает правильно – первые строки вообще пустые, хотя делители у первых чисел, очевидно, есть.

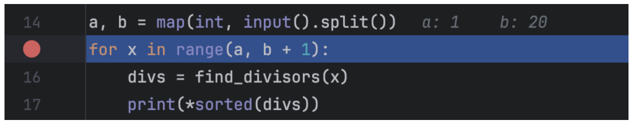
Нажмем на жука, запустим программу в дебаг-режиме. Мы хотим выполнять нашу программу построчно, но прямо сейчас это сделать не получится. Программа ждет ввода чисел, введем их ей, и увидим, что программа снова завершилась, как и ранее. Для того, чтобы программа остановилась в некоторый момент, и мы бы дальше продолжили пошаговое её выполнение, нужно поставить брейкпоинт (*breakpoint*), или точку остановки программы. Для этого, мы можем нажать на номер строки, в которой мы хотим, чтобы наша программа останавливалась. Появится красный круг, и каждый раз, когда программа будет доходить до этой строки, она будет останавливаться и ждать команду от вас:



Точка остановки в 15й строке

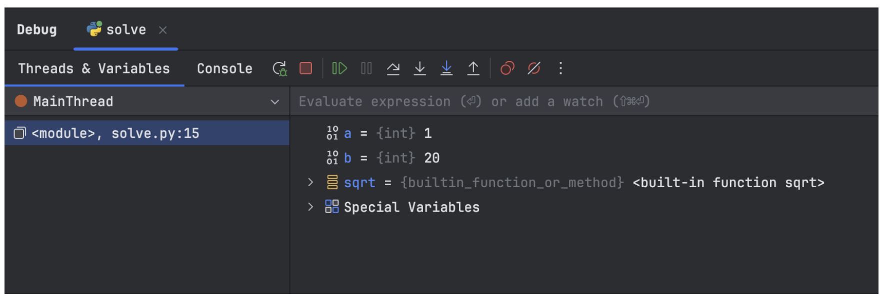
В примере выше точка остановки поставлена на 15ую строку. В следующий раз, когда мы запустим программу через дебагер (жука), она остановится на этой строке. Запустим, введем числа от 1 до 20, и посмотрим что получилось.

Синяя строка обозначает, что программа сейчас находится на этой строке:



Программа находится на строке 15

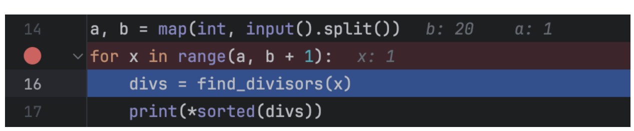
Какую информацию мы уже с вами можем получить? Как минимум, мы уже видим, что значения a и b равны 1 и 20 соответственно, значит считывание работает правильно. Значения видны в двух местах: в месте объявления в коде и в нижнем окне:



В окне дебагера отображаются значения переменных

Дальше пока ничего нет: `x` еще не создался, потому что синяя строка пока не выполнилась. Чтобы выполнить следующую строку, можно воспользоваться либо кнопками на панели («step over» и «step into»), либо соответствующими горячими клавишами. Клавиша F7, или «step into», переходит на следующую строку, которую будет выполнять ваша программа, и если в текущей строке вызывается некоторая функция, то она перейдет в эту функцию.

В нашем случае нажатие любой клавиши приведет нас на шестнадцатую строку, потому что мы в пятнадцатой строке не вызываем никакую функцию. Давайте так и сделаем:

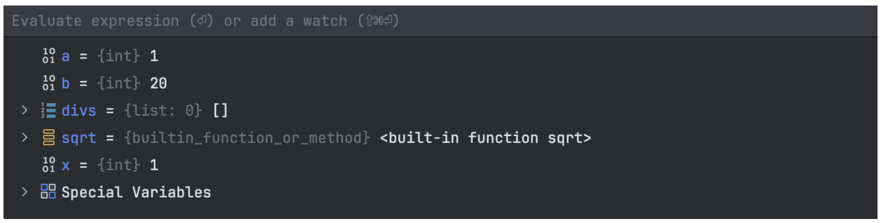


Мы перешли на следующую строку, нажав F7

Теперь мы можем как заходить в функцию `find_divisors`, так и не заходить. Если мы нажмем F8, то мы не зайдем в функцию, и сразу перейдем на семнадцатую строку.

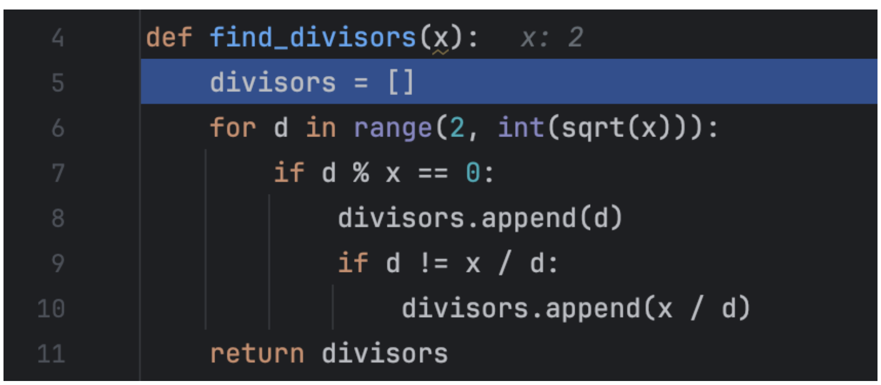
Видим, что список делителей в данный момент пустой, то есть функция вернула пустой список для `x = 1`:

Мы готовы выводить список, сделаем это и перейдем на следующую строку. Можно в ходе выполнения переключаться на консоль, смотреть что вывелось – пока что ничего. Вернемся на вкладку с переменными. Там появилось и `x`, и `divs`, который остался пустым с прошлой итерации цикла:

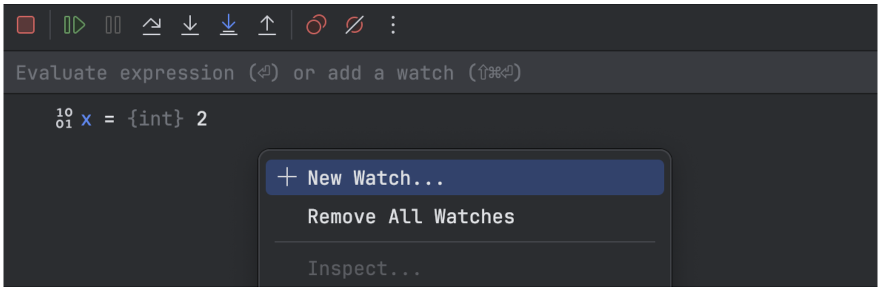


Список `divs` пустой после перехода к 17 строке

Теперь у нас `x` равен двум. Давайте теперь зайдем в функцию `find_divisors`, нажав клавишу `F7` (step into). Внутри функции виден `x`, он равен двум:

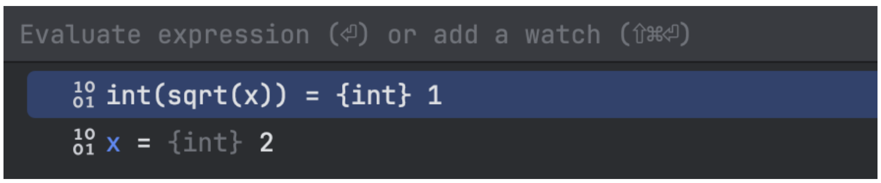


Продолжим выполнять функцию. Создадим список, и увидим, что выполнение функции сразу перескочило к строке `return`, поскольку ни одной итерации цикла у нас не произошло. Давайте разберемся почему. Для этого научимся смотреть на значение некоторого выражения, например посмотрим на значение границы диапазона `range` – `int(sqrt(x))`. Скопируем выражение и добавим в нижнюю панель так называемый *watch*, или наблюдение. Для этого нажмем правой кнопкой мыши на нижнее окошко, и добавим новый *watch*:



Добавление нового наблюдения

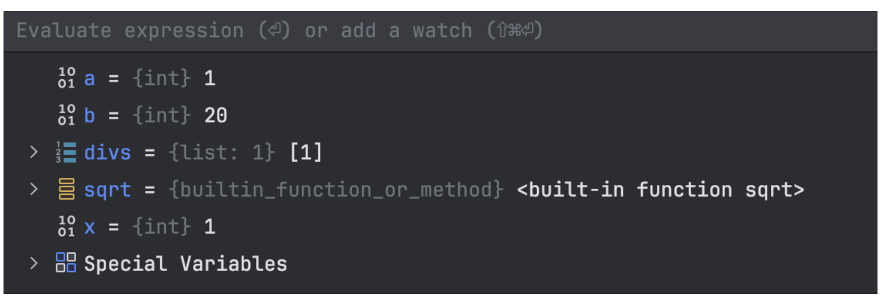
Вставим наше выражение, которое хотим посмотреть. Видим, что выражение имеет тип `int` (это указано в фигурных скобках), и оно равно единице:



Получается, что наш цикл работал от двух до единицы, что некорректно. У двойки есть делители – единица и сама двойка.

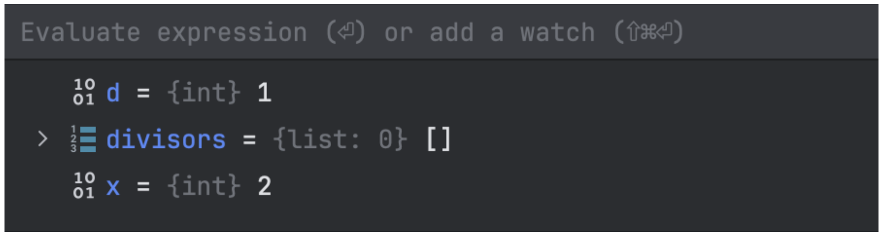
Мы нашли первую ошибку – давайте исправим границы в цикле. Мы хотим перебирать границы от единицы, до $\sqrt{x}$ включительно. Поскольку `range` перебирает до второго аргумента не включительно, прибавим к правой границе единицу, получив `range(1, int(sqrt(x)) + 1)`.

Теперь программу нужно перезапустить, потому что мы её исправили. Снова запустим её через жучка и посмотрим, что происходит. Введем данные, выполним функцию. Сейчас мы видим, что для единицы один делитель, равный единице, мы выписали:



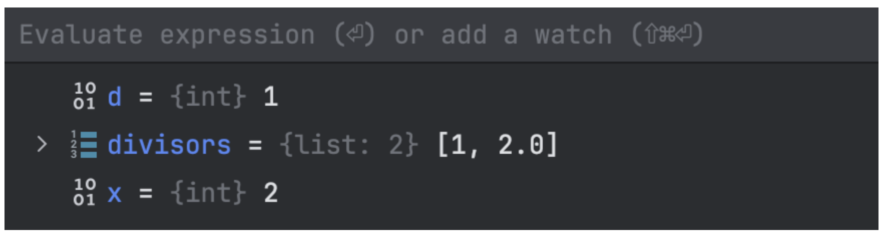
После вызова от `x = 1` список `divs` содержит 1 делитель

Пойдем дальше, зайдем в функцию для `x = 2`. Теперь видно, что в `for` программа зашла, и что текущее `d` равно единице:



Зайдя внутрь цикла в `find_divisors`, видим, что `d = 1`

Можем посмотреть, что происходит дальше. Мы зашли в `if`, ведь действительно, у нас `x = 2`, а `d = 1` – остаток равен нулю. Дальше мы зайдем в следующий `if`, потому что `d != x/d`. Действительно, `d` равно единице, а `x/d` равно двум. Идем дальше и кладем значение в список. Видим, что в список у нас поместилось значение `2.0`:

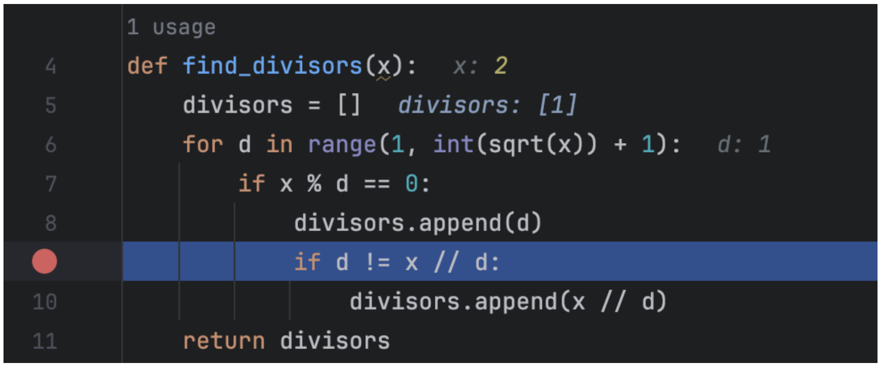


Значение переменных после первой итерации в `find_divisors` при `x=2`

Это не очень корректно, так как мы хотели сохранить целое число. Мы нашли еще одну ошибку – использовали `/` вместо `//`, причем сразу в двух местах. Давайте устраним эти ошибки, и перезапустим нашу программу. Также давайте теперь переставим брейкпоинт – поставим его сразу на `if` в 9 строке, чтобы остановиться сразу там (при желании мы можем расставить и несколько брейкпоинтов).

Также мы можем продолжать выполнение программы до того момента, пока она снова не придет в брейкпоинт. Давайте посмотрим, как это делать. Запустим программу. Видим, что мы остановились в 9 строке в первый раз, когда `x` и `d` равны единице. Мы нашли один делитель, а в `if` не зашли, потому что `x = 1` – полный квадрат. Пошагово выполняем дальше, и допустим, вот мы уже зашли в функцию при `x = 2`:

Давайте продолжим нашу программу сразу до следующего брейкпоинта. Для этого нажмем на «resume program», или `F9`. Нажимаем, и оказываемся снова на девятой строке, когда пришли туда в следующий раз:



В общем случае мы остановимся тогда, когда встретим самый первый по ходу выполнения брейкпоинт. В нашем случае все работает верно для `x = 2`, идем дальше.

Переходим к `x = 3` и пробуем пропустить выполнение функции через `F8`, но нас останавливают на девятой строке, потому что там есть брейкпоинт. Мы пытались перескочить функцию, но не получилось. Мы можем на ходу выполнения программы убирать и добавлять брейкпоинты, если они потеряли свою актуальность или, наоборот, понадобились. Давайте теперь продолжим нашу программу и перескочим сразу к выводу делителей числа 3.

Мы можем в любой момент записывать произвольные выражения в список наблюдений. Давайте перейдем к девятой строке, и посмотрим, чему же равен `x // d`. Скопируем это выражение и добавим через *new watch*:

Теперь мы видим значение `x // d` в этом месте кода. Мы также можем и удалять *watch*-и, которые нам больше не нужны. Если убрать все брейкпоинты, можно закончить выполнение программы через *resume*. Теперь в консоли видно, что у нас все делители находятся верно.

В режиме дебага нет цветового разделения на то, что ввели мы, а что вывела программа, но мы всегда можем запустить нашу программу и в обычном режиме, чтобы разделить визуально потоки ввода и вывода.