Функция range() в Python

Встроенная функция Python под названием **range** может быть очень полезной, если вам нужно выполнить действие определенное количество раз.

К концу данного руководства вы будете:

- Понимать, как работает функция Python range;
- Знать, как отличаются реализации для Python 2 и Python 3;
- Увидите ряд наглядных примеров работы с range();
- Сможете работать с учетом ограничений range().

Содержание:

Приступим к делу!

История range()

Несмотря на то, что range() в **Python 2** и range() в Python 3 носят одинаковое название, они кардинально отличаются между собой. Фактически, **range**() в Python 3 – это просто переименованная версия функции под названием **xrange** в Python 2.

Изначально, **range**() и **xrange**() приводили числа, которые можно повторить при помощи цикла for, однако первая функция генерировала список этих чисел, учитывая все за раз, в то время как вторая делала это более лениво, т. е. Числа возвращались по одному каждый раз, когда они нужны.

Наличие огромных списков занимает память, так что нет ничего удивительного в том, что xrange() заменила range(), ее имя и все остальное. Вы можете прочитать больше об этом решении и предыстории xrange() и range() в **PEP 3100**.

> **Обратите внимание**: PEP означает Python Enhancement Proposal. Это документы, которые покрывают большое количество тем, включая недавно предоставленные новые функции, стили, философию и руководства.

Приступим!

Циклы

Перед тем, как мы ознакомимся с тем, как работает **range**(), нам нужно взглянуть на то, как работают циклы. **Циклы** — это ключевая концепция компьютерных наук. Если вы хотите стать хорошим программистом, умение обращаться с циклами — это важнейший навык, который стоит освоить.

Рассмотрим пример **цикла for** в Python:

```python
captains = ['Janeway', 'Picard', 'Sisko']

for captain in captains:
    print(captain)
```

Выдача выглядит следующим образом:

```python
Janeway
Picard
Sisko
```

Как вы видите, **цикл for** позволяет вам выполнять определенные части кода, столько раз, сколько вам угодно. В данном случае, мы зациклили список капитанов и вывели имена каждого из них.

Хотя **Star Trek** — отличная тема и все такое, вам может быть нужен более сложный цикл, чем список капитанов. Иногда вам нужно просто выполнить часть кода определенное количество раз. Циклы могут помочь вам с этим.

Попробуйте запустить следующий код с числами, кратными трем:

```python
numbers_divisible_by_three = [3, 6, 9, 12, 15]

for num in numbers_divisible_by_three:
    quotient = num / 3
    print(f"{num} делится на 3, результат {int(quotient)}.")
```

Выдача цикла будет выглядеть следующим образом:

```
3 делится на 3, результат 1.
6 делится на 3, результат 2.
9 делится на 3, результат 3.
12 делится на 3, результат 4.
15 делится на 3, результат 5.
```

Это выдача, которая нам нужна, так что можем сказать, что цикл выполнил работу адекватно, однако есть еще один способ получения аналогично результата: использование **range**().

> **Обратите внимание**: Последний пример кода содержит определенное форматирование строк. Чтобы узнать больше об этой теме, перейдите на статью F-Строки в новой версии Python.

Теперь, когда вы знакомы с циклами поближе, посмотрим, как вы можете использовать **range**() для упрощения жизни.

Введение в range()

Итак, как работает функция Python под названием range? Простыми словами, **range**() позволяет вам генерировать ряд чисел в рамках заданного диапазона. В зависимости от того, как много аргументов вы передаете функции, вы можете решить, где этот ряд чисел начнется и закончится, а также насколько велика разница будет между двумя числами.

Вот небольшой пример **range**() в действии:

```python
for i in range(3, 16, 3):
    quotient = i / 3
    print(f"{i} делится на 3, результат {int(quotient)}.")
```

В этом цикле вы просто можете создать ряд чисел, кратных трем, так что вам не нужно вводить каждое из них лично.

> **Обратите внимание**: хотя в этом примере показано надлежащее использование range(), его слишком часто приводят для использования в циклах.

Например, следующее использование range() едва ли можно назвать Питоническим (**это плохой пример**):

```python
captains = ['Janeway', 'Picard', 'Sisko']

for i in range(len(captains)):
    print(captains[i])
```

range() отлично подходит для создания повторяющихся чисел, но это не самый лучший выбор, если вам нужно перебрать данные, которые могут быть зациклены с помощью **оператора in**.

Есть три способа вызова range():

1. range(**стоп**) берет один аргумент
2. range(**старт**, **стоп**) берет два аргумента
3. range(**старт**, **стоп**, **шаг**) берет три аргумента

Вызывая range() с одним аргументом, вы получите ряд чисел, начинающихся с 0 и включающих каждое число до, но не включая число, которое вы обозначили как конечное (**стоп**).

Как это выглядит на практике:

```python
for i in range(3):
    print(i)
```

Выдача вашего цикла будет выглядеть так:

```python
0
1
2
```

Проверим: у нас есть все числа от 0 до, но не включая 3 — числа, которое вы указали как конечное.

range(старт, стоп)

Вызывая **range**() с двумя аргументами, вам нужно решить не только, где ряд чисел должен остановиться, но и где он должен начаться, так что вам не придется начинать с нуля каждый раз. Вы можете использовать **range**() для генерации ряда чисел, начиная с А до Б, используя диапазон (А, Б). Давайте узнаем, как генерировать диапазон, начинающийся с 1.

Попробуем вызывать **range**() с двумя аргументами:

```python
for i in range(1, 8):
    print(i)
```

Ваша выдача будет выглядеть следующим образом:

```python
1
2
3
4
5
6
7
```

Отлично: у вас есть все числа от 1 (число, которые вы определили как стартовое), до, но не включая, 8 (число, которые вы определили как конечное).

Но если вы добавите еще один аргумент, то вы сможете воспроизвести ранее полученный результат, когда пользуетесь списком под названием **numbers_divisible_by_three**.

range(старт, стоп, шаг)

Вызывая **range**() с тремя аргументами, вы можете выбрать не только то, где ряд чисел начнется и остановится, но также то, на сколько велика будет разница между одним числом и следующим. Если вы не задаете этот «шаг», то **range**() автоматически будет вести себя так, как если бы шаг был бы равен 1.

Обратите внимание: шаг может быть положительным, или отрицательным числом, но он не может равняться нулю:

```
>>> range(1, 4, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: range() arg 3 must not be zero
```

Если вы попробуете использовать 0 как шаг, вы получите ошибку ValueError.

Теперь, так как вы знаете, как использовать шаг, вы можете снова использовать цикл, который мы видели ранее, с числами, кратными 3.

Попробуйте лично:

```
for i in range(3, 16, 3):
    quotient = i / 3
    print(f"{i} делится на 3, результат {int(quotient)}.")
```

Ваша выдача будет выглядеть абсолютно так же, как выдача для **цикла for**, которую мы видели ранее в данном руководстве, когда мы использовали список numbers_divisible_by_three:

```
3 делится на 3, результат 1.
6 делится на 3, результат 2.
9 делится на 3, результат 3.
12 делится на 3, результат 4.
15 делится на 3, результат 5.
```

Как вы видите в этом примере, вы можете использовать аргумент **шаг** для увеличения в сторону больших чисел. Это называется **инкрементация**.

Инкрементация с range()

Если вы хотите выполнить инкрементацию, то вам нужно, чтобы шаг был положительным числом. Чтобы понять, что под этим имеется ввиду, введите следующий код:

```
for i in range(3, 100, 25):
    print(i)
```

Если ваш шаг равен 25, то выдача вашего цикла будет выглядеть вот так:

```
3
28
```

```
53
78
```

Вы получили ряд чисел, каждое из которых больше предыдущего на 25, т.е., на заданный вами шаг.

Теперь, так как вы увидели то, как именно вы можете продвигаться вперед по диапазону, настало время узнать, как двигаться в обратную сторону.

Декрементация с range()

Если ваш шаг положительный — то вы двигаетесь по ряду увеличивающихся чисел, это называется **инкрементация**. Если ваш шаг отрицательный, то вы двигаетесь по ряду убывающих чисел, это называется декрементация. Это позволяет вам идти вспять.

В следующем примере ваш шаг будет **-2**. Это значит, что декрементация будет равна 2 для каждого цикла:

Python

```python
for i in range(10, -6, -2):
    print(i)
```

Выдача вашего декременирующего цикла будет выглядеть следующим образом:

Python

```
10
8
6
4
2
0
-2
-4
```

У вас есть ряд чисел, каждое из которое меньше предшествующего на 2, т. е., на абсолютное значение предоставленного вами шага.

Самый правильный способ создание диапазона декрементации, это использовать range(**старт**, **стоп**, **шаг**). При этом в Python есть встроенная обратная функция. Если вы завернете **range**() в **reversed**(), то вы сможете выводить целые числа в обратном порядке.

Давайте попробуем:

Python

```python
for i in reversed(range(5)):
    print(i)
```

Вы получите следующее:

Python

```
4
3
2
1
0
```

range() позволяет итерировать по декрементирующей последовательности чисел, где **reversed**() обычно используется для циклического преобразования последовательности в обратном порядке.

> **Обратите внимание**: reversed() также работает со строками.

Углубляемся в range()

Теперь, когда вы ознакомились с основами использования range(), настало время спуститься немного глубже.

Как правило, **range**() используется в двух случаях:

1. Выполнении тела цикла определенное количество раз;
2. Создание более эффективных итераций целых чисел, которое может быть выполнено при помощи списков или кортежей.

Первое использование можно назвать самым простым, и вы можете сделать так, чтобы itertools дал вам более эффективный способ построения итераций, чем это может сделать **range**().

Вот еще несколько моментов, которые стоит учитывать при использовании **range**.

Python
```
>>> type(range(3))
<class 'range'>
```

Вы можете получить доступ к объектам в **range**() по индексу, как если бы вы имели дело со списком:

Python
```
print(range(3)[1])
# Результат: 1

print(range(3)[2])
# Результат: 2
```

Вы даже можете использовать срез в **range**(), но выдача в REPL может показаться немного странной, на первый взгляд:

Python
```
print(range(6)[2:5])
# Результат: range(2, 5)
```

Хотя эта выдача может выглядеть необычно, **range**() просто возвращает еще одну **range**().

Тот факт, что вы можете использовать элементы range() по индексу и фрагменту range() указывает на важный момент: range() весьма ленивый, в отличие от списка, но не является итератором.

float и range()

Вы могли обратить внимание на то, что все числа, с которыми мы имели дело, являлись целыми числами. Это связано с тем, что range() может принимать только целые числа в качестве аргументов.

float

В Python, если число не является целым, оно является десятичным. Есть несколько различий между целыми и десятичными числами.

Целое число (тип данных **int**):

- Является целым числом;
- Не содержит десятичной точки;
- Может быть положительным, отрицательными или нулем;

Десятичное число (тип данных **float**):

- Может быть любым числом, которое включает десятичную точку;
- Может быть положительным и отрицательным;

Попробуйте вызвать **range**() с десятичным числом и увидите, что будет:

```python
for i in range(3.3):
    print(i)
```

Вы увидите следующее уведомление об ошибке TypeError:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

Если вам нужен обходной путь, который позволит вам использовать десятичные числа, вы можете использовать NumPy.

Использование NumPy

NumPy – это сторонняя библиотека Python. Если вы собираетесь ее использовать, сначала вам нужно убедиться в том, что она установлена.

Как это сделать при помощи REPL:

```python
import numpy
```

Если вы получите ошибку ModuleNotFoundError, то вам нужно провести **установку numpy**. Чтобы сделать это, перейдите в командную строку и введите:

```python
pip install numpy
```

После установки, внесите следующее:

```python
import numpy as np

np.arange(0.3, 1.6, 0.3)
```

Результат:

```python
array([0.3, 0.6, 0.9, 1.2, 1.5])
```

Если вы хотите вывести каждое число на свою строку, вы можете сделать следующее:

```python
import numpy as np

for i in np.arange(0.3, 1.6, 0.3):
    print(i)
```

Выдача будет следующей:

```python
0.3
0.6
0.8999999999999999
1.2
1.5
```

Но откуда взялось число **0.8999999999999999**?

У компьютеров есть проблемы с сохранением десятичных чисел с запятой в двоичные числа с запятой. Это приводит к разным неожиданным представлениям этих чисел.

> Возможно, вы захотите взглянуть на библиотеку **decimal**, которая немного отстает в контексте производительности и читаемости, но позволяет вам выражать десятичные числа в точном виде.
>
> Еще один вариант – использовать **round**(). Помните, что round() содержит собственные нюансы, которые могут приводить к неожиданным результатам!

Так или иначе, эти ошибки связанные с **плавающей запятой** являются проблемой, в зависимости от того, над какой задачей вы работаете. Ошибки могут быть выражены в виде, например, шестнадцатеричного десятичного числа, что не является критичной проблемой, в большинстве случаев. Они настолько маленькие, что, если вы только не работаете над расчетами орбитальной траектории спутников, вам не стоит беспокоиться.

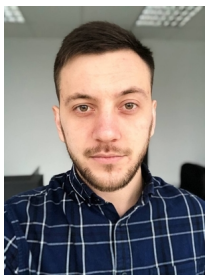
В качестве альтернативы, вы можете использовать **np.linspace**(). Он делает в целом то же самое, но с использованием других параметров. С **np.linspace**() вы определяете начало и конец (оба включительно), а также длину и массив (за исключением шага).

Например, **np.linspace**(1, 4, 20) выдает 20 одинаково разделенных чисел: .0, ..., 4.0. В другом случае, **np.linspace**(0, 0.5, 51) задает 0.00, 0.01, 0.02, 0.03, ..., 0.49, 0.50.

Итоги

Теперь вы понимаете, как использовать **range**() и работать в обход его ограничений. Также вы понимаете, как эта важная функция развивалась между Python 2 и Python 3.

Счастливого программирования!



Vasile Buldumac

Являюсь администратором нескольких порталов по обучению языков программирования Python, Golang и Kotlin. В составе небольшой команды единомышленников, мы занимаемся популяризацией языков программирования на русскоязычную аудиторию. Большая часть статей была адаптирована нами на русский язык и распространяется бесплатно.

E-mail: vasile.buldumac@ati.utm.md

Образование
Universitatea Tehnică a Moldovei (*utm.md*)

2014 — 2018 Технический Университет Молдовы, ИТ-Инженер. Тема дипломной работы «*Автоматизация покупки и продажи криптовалюты используя технический анализ*»

2018 — 2020 Технический Университет Молдовы, Магистр, Магистерская диссертация «*Идентификация человека в киберпространстве по фотографии лица*»

in