Последовательности и циклы

[1. Строка как последовательность. Индексы, срезы(slice). 1](#_Toc149423270)

[Срезы 3](#_Toc149423271)

[Изменение строк 4](#_Toc149423272)

[Практика 1 Строки (срезы) 4](#_Toc149423273)

[Основные методы строк 6](#_Toc149423274)

[Практика 2 Строки (методы) 13](#_Toc149423275)

[2. Кодирование и декодирование строк 16](#_Toc149423276)

[Байтовая строка 19](#_Toc149423277)

[Набор символов ASCII 20](#_Toc149423278)

[Спецсимволы, экранирование символов, raw-строки 20](#_Toc149423279)

[Практика 3 Строки (спецсимволы, экранирование, raw) 23](#_Toc149423280)

[3. Оператор цикла while 25](#_Toc149423281)

[Бесконечный цикл 25](#_Toc149423282)

[Конструкция while - else 26](#_Toc149423283)

[Конструкция while – break-else 26](#_Toc149423284)

[Моржовый оператор 27](#_Toc149423285)

[Практика 4 Цикл while 29](#_Toc149423286)

[Оператор continue 30](#_Toc149423287)

[Разбор игры Угадай число 34](#_Toc149423288)

[Практика 4 Цикл while и строки 34](#_Toc149423289)

[Практика 5 while + break 35](#_Toc149423290)

[4. Тип данных range 37](#_Toc149423291)

[Примеры использования: 38](#_Toc149423292)

[5. Оператор цикла for. 39](#_Toc149423293)

[Практика 6 for 44](#_Toc149423294)

[Примеры работы оператора цикла for. Функция enumerate() 48](#_Toc149423295)

[6. Вложенные циклы 51](#_Toc149423296)

[Практика 7 вложенные циклы 56](#_Toc149423297)

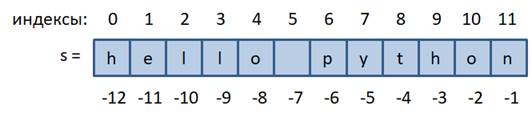
[Итератор и итерируемые объекты. Функции iter() и next() 58](#_Toc149423298)

[Тип данных NoneType 62](#_Toc149423299)

# Строка как последовательность. Индексы, срезы(slice).

Как вы уже знаете, в Python строку можно задавать, например, так:

s = "hello python"



и она представляет собой упорядоченный набор символов. Что значит упорядоченный?

Смотрите, каждый символ строки имеет свой уникальный, порядковый номер. Эти номера называются индексами.

Первый – 0, второй – 1 и так до конца. Мы можем использовать эти индексы для обращения к отдельному символу строки. Для этого, записывается имя переменной и в квадратных скобках указывается номер символа:

s[0]

s[1]

Но будьте внимательны, если указать несуществующий индекс:

s[12]

то получим ошибку выхода за пределы диапазона. В нашем случае последний индекс равен 11, а мы записали 12. Чтобы взять последний символ нам следовало бы записать:

s[11]

То есть, последний индекс равен длине строки минус один:

len(s) - 1

И, по идее, мы могли бы его так и указать в квадратных скобках:

s[len(s) - 1]

Эта конструкция работает для строк произвольной длины (кроме нулевой, когда символов нет). Но в Python то же самое можно записать проще, используя отрицательные индексы:

s[-1]

То есть, длину строки len(s) можно опускать и записывать отрицательные значения. Получаем способ индексации с конца строки.

Причем индексацию можно выполнять также и непосредственно над строками, например:

"panda"[3]

вернет 4-й символ этой строки. Иногда это тоже может быть полезно.

## Срезы

Фактически, когда мы выполняем индексацию, то возвращается новая строка из одного выделенного символа. Но мы можем выделять и сразу несколько символов, используя следующий синтаксис:

строка[start:stop)

Выделенная последовательность символов называется срезом строки.

Например, можно сделать так:

s[1:3]

вернет два символа с индексами 1 и 2. Последнее значение 3 не включается в срез. В срезах можно не указывать последнее значение:

s[4:]

или первое:

s[:5]

или оба:

s[:]

В последнем случае возвращается та же самая строка. Убедиться в этом можно, следующим образом:

a = s[:]

id(a)

id(s)

В срезах также можно использовать отрицательные индексы, например:

s[2:-2]

Но если записать:

s[-2:2]

то получим пустую строку, так как эти индексы не образуют диапазон значений.

Наконец, в срезах дополнительно можно указывать еще и шаг перебора символов, согласно синтаксису:

строка[start:stop:step)

Например:

s[2:10:2]

s[2::3]

s[:5:3]

s[::2]

И использовать отрицательный шаг:

s[::-1]

тогда все символы будут перебираться в обратном порядке, начиная с последнего. Если же указать:

s[::-2]

то будут выбираться символы с конца через один.

## Изменение строк

При работе со строками следует иметь в виду, что она относится к неизменяемым типам данных, то есть, существующую строку изменить нельзя. В частности, из-за этого попытка присвоить строке какой-либо символ:

s[0] = 'H'

приведет к ошибке. Чтобы изменить строку, нужно создать новую с другим содержимым:

s2 = 'H' + s[1:]

И так происходит каждый раз, когда нужно изменить что-то в уже существующей строке.

Итак, из этого материала вы должны хорошо себе представлять:

* строка – упорядоченный набор символов;
* как выполняется индексация к отдельным символам строки str[index];
* как выделять из строк наборы символов – срезы;
* строка – неизменяемый объект;
* способ модификации (изменения) строк через индексы и срезы.

Для закрепления этого материала пройдем практические задания и перейдем к следующему материалу

## Практика 1 Строки (срезы)

1. Напишите программу отображения первых четырех символов из введенной строки. Будем полагать, что строка гарантированно длиной не менее четырех символов.

Входные данные:

panda

Выходные данные:

pand

2. Напишите программу отображения последних трех символов из введенной строки. Будем полагать, что строка гарантированно длиной не менее трех символов.

Входные данные:

Пушкин

Выходные данные:

кин

3. Напишите программу отображения всех символов с нечетными индексами из введенной строки.

Входные данные:

Pushkin

Выходные данные:

uhi

4. Вводятся две строки (каждая с новой строчки). Из первой строки выделить все символы с четными индексами, а из второй - с нечетными. Объединить строки через пробел и вывести на экран.

Входные данные:

Hello

Python

Выходные данные:

Hlo yhn

5. Из введенной строки отобразить первые пять символов в обратном порядке. Полагается, что введенная строка имеет минимум пять символов.

Входные данные:

abrakadabra

Выходные данные:

karba

6. Вводятся два слова (через пробел в одной строке). Длина первого слова меньше второго. Необходимо обрезать второе слово до длины первого и отобразить обрезанное слово на экране.

Входные данные:

Hello Pushkin

Выходные данные:

Pushki

7. Напишите программу на Python для форматирования числа в процентах.



x = 0.25  
y = -0.25  
print("\nOriginal Number: ", x)  
print(f'{x:.2%}') # print(f'{x = :.2%}')  
print("Formatted Number with percentage: "+"{:.2%}".format(x));  
print("Original Number: ", y)  
print("Formatted Number with percentage: "+"{:.2%}".format(y));

## Основные методы строк

Познакомимся с основными методами для строк. Что такое методы? Смотрите, когда мы объявляем какую-либо строку:

s = "python"

то в памяти устройства автоматически создается объект, содержащий указанные символы. Тип данных этого объекта – строка:

type(s)

Так вот, каждый такой объект связан с набором стандартных функций по работе со строками. Эти функции и называются методами.



Чтобы вызвать метод для конкретной строки, необходимо указать объект, поставить точку, записать имя метода и в круглых скобках список аргументов, если они необходимы:

объект.метод(аргументы)

Давайте рассмотрим все на конкретных примерах. Итак, у нас есть объект-строка, на который ссылается переменная s. Через эту переменную можно вызывать все методы строк. Чтобы увидеть их полный список, можно записать:

s.

и Pycharm отобразит полный их список. Мы с вами сейчас увидим работу наиболее употребительных из них. Первый метод:

s.upper()

возвращает новую строку (новый объект) со всеми заглавными буквами. При этом сама строка остается без изменений. И это логично, так как строки относятся к неизменяемым типам данных.

Обратите внимание, для вызова этого метода после его имени обязательно нужно поставить круглые скобки. Без них мы получим просто ссылку на объект-функцию:

s.upper

но запущена она не будет. Для запуска необходимы круглые скобки в конце – это оператор для выполнения функций и методов.

Если мы хотим сохранить результат преобразования строки в какой-либо переменной, то это делается так:

res = s.upper()

И теперь res ссылается на строку 'PYTHON'.

Второй метод:

res.lower()

наоборот, переводит все буквенные символы в нижний регистр и возвращает соответствующую строку.

Следующий метод

String.count(sub[, start[, end]])

возвращает число повторений подстроки sub в строке String. Два необязательных аргумента:

* start – индекс, с которого начинается поиск;
* end – индекс, которым заканчивается поиск.

В самом простом случае, мы можем для строки

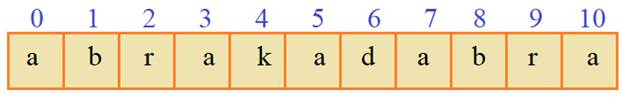
msg = "abrakadabra"

определить число повторений сочетаний «ra»:

msg.count("ra")

получим значение 2 – именно столько данная подстрока встречается в нашей строке.

Теперь предположим,  что мы хотим начинать поиск с буквы k, имеющей индекс 4.



Тогда метод следует записать со значением start=4:

msg.count("ra", 4)

и мы получим значение 1. Далее, укажем третий аргумент – индекс, до которого будет осуществляться поиск. Предположим, что мы хотим дойти до 10-го индекса и записываем:

msg.count("ra", 4, 10)

и получаем значение 0. Почему? Ведь на индексах 9 и 10 как раз идет подстрока «ra»? Но здесь, также как и в срезах, последний индекс исключается из рассмотрения. То есть, мы говорим, что нужно дойти до 10-го, не включая его. А вот если запишем 11:

msg.count("ra", 4, 11)

то последнее включение найдется.

Следующий метод

String.find(sub[, start[, end]])

возвращает индекс первого найденного вхождения подстроки sub в строке String. А аргументы start и end работают также как и в методе count. Например:

msg.find("br")

возвращает 1, т.к. первое вхождение «br» как раз начинается с индекса 1. Поставим теперь значение start=2:

msg.find("br", 2)

и поиск начнется уже со второго индекса. Получим значение 8 – индекс следующего вхождения подстроки «br». Если мы укажем подстроку, которой нет в нашей строке:

msg.find("brr")

то метод find возвращает -1. Третий аргумент end определяет индекс, до которого осуществляется поиск и работает также как и в методе count.

Метод find ищет первое вхождение слева-направо. Если требуется делать поиск в обратном направлении: справа-налево, то для этого используется метод

String.rfind(sub[, start[, end]])

который во всем остальном работает аналогично find. Например:

msg.rfind("br")

возвратит 8 – первое вхождение справа.

Наконец, третий метод, аналогичный find – это:

String.index(sub[, start[, end]])

Он работает абсолютно также как find, но с одним отличием: если указанная подстрока sub не находится в строке String, то метод приводит к ошибке:

msg.index("brr")

тогда как find возвращает -1. Спрашивается: зачем нужен такой ущербный метод index? В действительности такие ошибки можно обрабатывать как исключения и это бывает полезно для сохранения архитектуры программы, когда неожиданные ситуации обрабатываются единым образом в блоке исключений. Но, обо всем этом позже.

Следующий метод

String.replace(old, new, count=-1)

Выполняет замену подстрок old на строку new и возвращает измененную строку. Например, в нашей строке, мы можем заменить все буквы a на o:

msg.replace("a", 'o')

на выходе получим строку «obrokodobro». Или, так:

msg.replace("ab", "AB")

Используя этот метод, можно выполнять удаление заданных фрагментов, например, так:

msg.replace("ab", "")

Третий необязательный аргумент задает максимальное количество замен. Например:

msg.replace("a", 'o', 2)

Заменит только первые две буквы a: «msg.replace("a", 'o', 2)». При значении -1 количество замен неограниченно.

Следующие методы позволяют определить, из каких символов состоит наша строка. Например, метод

String.isalpha()

возвращает True, если строка целиком состоит из букв и False в противном случае. Посмотрим, как он работает:

msg.isalpha()

вернет True, т.к. наша строка содержит только буквенные символы. А вот для такой строки:

"hello world".isalpha()

мы получим False, т.к. имеется символ пробела.

Похожий метод

String.isdigit()

возвращает True, если строка целиком состоит из цифр и False в противном случае. Например:

"5.6".isdigit()

т.к. имеется символ точки, а вот так:

"56".isdigit()

получим значение True. Такая проверка полезна, например, перед преобразованием строки в целое число. О проверках мы еще будем говорить.

Следующий метод

String.rjust(width[, fillchar = ‘ ‘])

возвращает новую строку с заданным числом символов width и при необходимости слева добавляет символы fillchar:

d="abc"

d.rjust(5)

Получаем строку «  abc» с двумя добавленными слева пробелами. А сама исходная строка как бы прижимается к правому краю. Этот метод часто используют для добавления незначащих нулей перед цифрами:

d = "12"

d.rjust(3, '0')

Получим строку «012». Причем вторым аргументом можно писать только один символ. Если записать несколько, то возникнет ошибка:

d.rjust(3, "00")

Если ширина width будет меньше длины строки:

d.rjust(1)

то вернется исходная строка. Аналогично работает метод

String.ljust(width[, fillchar = ‘ ‘])

который возвращает новую строку с заданным числом символов width, но добавляет символы fillchar уже справа:

d.ljust(10, "\*")

Следующий метод

String.split(sep=None, maxsplit=-1)

возвращает коллекцию строк, на которые разбивается исходная строка String. Разбивка осуществляется по указанному сепаратору sep. Например:

"Иванов Иван Иванович".split(" ")

Мы здесь разбиваем строку по пробелам. Получаем коллекцию из ФИО. Тот же результат будет и при вызове метода без аргументов, то есть, по умолчанию он разбивает строку по пробелам:

"Иванов Иван Иванович".split()

А теперь предположим, перед нами такая задача: получить список цифр, которые записаны через запятую. Причем, после запятой может быть пробел, а может и не быть. Программу можно реализовать так:

digs = "1, 2,3, 4,5,6"

digs.replace(" ", "").split(",")

мы сначала убираем все пробелы и для полученной строки вызываем split, получаем список цифр.

split(delimeter, num): параметр num указывает, сколько вхождений delimeter используется для разделения. Оставшаяся часть строки добавляется в список без разделения на подстроки.

text = "Это был огромный, в два обхвата дуб, с обломанными ветвями и с обломанной корой"

# разбиение по первым пяти пробелам

splitted\_text = text.split(" ", 5)

print(splitted\_text)

print(splitted\_text[5])     # обхвата дуб, с обломанными ветвями и с обломанной корой

Еще один метод - **partition()** разбивает строку по разделителю на три подстроки и возвращает кортеж из трех элементов - подстрока до разделителя, разделитель и подстрока после разделителя:

text = "Это был огромный, в два обхвата дуб, с обломанными ветвями и с обломанной корой"

text\_parts = text.partition("дуб")

print(text\_parts)

# ('Это был огромный, в два обхвата ', 'дуб', ', с обломанными ветвями и с обломанной корой')

Обратный метод

String.join(список)

возвращает строку из объединенных элементов списка, между которыми будет разделитель String. Например:

d = digs.replace(" ", "").split(",")

", ".join(d)

получаем строку «1, 2, 3, 4, 5, 6». Или так, изначально была строка:

fio = "Иванов Иван Иванович"

и мы хотим здесь вместо пробелов поставить запятые:

fio2 = ",".join(fio.split())

Теперь fio2 ссылается на строку с запятыми «Иванов,Иван,Иванович».

Вместо списка в метод join можно передать простую строку, тогда разделитель будет вставляться между символами этой строки:

word = "hello"

joined\_word = "|".join(word)

print(joined\_word)      # h|e|l|l|o

Следующий метод

String.strip()

удаляет пробелы и переносы строк в начале и конце строки. Например:

"    hello world         \n".strip()

возвращает строку «hello world». Аналогичные методы:

String.rtrip() и String.ltrip()

удаляют пробелы и переносы строк только справа или только слева.

Метод isalpha() возвращает True, если все символы в строке являются алфавитными. Если нет, возвращается False.

str.isalnum()

Возвращает значение True, если все символы в строке являются буквенно-цифровыми и имеется хотя бы один символ, False в противном случае. Символ является буквенно-цифровым, если возвращается одно из следующих значений True: c.isalpha(), c.isdecimal(), c.isdigit(), или c.isnumeric().

str.isascii()

Возврат, True если строка пуста или все символы в ней являются ASCII, False в противном случае. Символы ASCII имеют кодовые точки в диапазоне от U+0000 до U+007F.

### Подстановки

Еще один способ передачи форматируемых значений в строку представляет использование подстановок или специальных плейсхолдеров, на место которых вставляются определенные значения. Для форматирования мы можем использовать следующие плейсхолдеры:

* **s**: для вставки строк
* **d**: для вставки целых чисел
* **f**: для вставки дробных чисел. Для этого типа также можно определить через точку количество знаков в дробной части.
* **%**: умножает значение на 100 и добавляет знак процента

|  |  |
| --- | --- |
| 1  2  3 | n = 5000  source = f"{n:,d} символов"  print(source)   # 5,000 символов |

Для дробных чисел, то есть таких, которые представляют тип float, перед кодом плейсхолдера после точки можно указать, сколько знаков в дробной части мы хотим вывести:

|  |  |
| --- | --- |
| 1  2  3  4  5 | number = 23.8589578  print("{:.2f}".format(number))   # 23.86  print("{:.3f}".format(number))   # 23.859  print("{:.4f}".format(number))   # 23.8590  print("{:,.2f}".format(10001.23554))    # 10,001.24 |

Еще один параметр позволяет установить минимальную ширину форматируемого значения в символах:

|  |  |
| --- | --- |
| 1  2 | print("{:10.2f}".format(23.8589578))    #     23.86  print("{:8d}".format(25))               #      25 |

Аналогичный пример с f-строками:

|  |  |
| --- | --- |
| 1  2  3  4 | n1 = 23.8589578  print(f"{n1:10.2f}")    #     23.86  n2 = 25  print(f"{n2:8d}")       #      25 |

Для вывода процентов лучше воспользоваться кодом "%":

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | number = .12345  print("{:%}".format(number))        # 12.345000%  print("{:.0%}".format(number))      # 12%  print("{:.1%}".format(number))      # 12.3%    print(f"{number:%}")        # 12.345000%  print(f"{number:.0%}")      # 12%  print(f"{number:.1%}")      # 12.3% |

В заключение занятия я приведу список всех рассмотренных методов, которые хорошо было бы запомнить и применять по мере необходимости при работе со строками:

|  |  |
| --- | --- |
| Название | Описание |
| String.upper() | Возвращает строку с заглавными буквами |
| String.lower() | Возвращает строку с малыми буквами |
| String.count(sub[, start[, end]]) | Определяет число вхождений подстроки в строке |
| String.find(sub[, start[, end]]) | Возвращает индекс первого найденного вхождения |
| String.rfind(sub[, start[, end]]) | Возвращает индекс первого найденного вхождения при поиске справа |
| String.index(sub[, start[, end]]) | Возвращает индекс первого найденного вхождения |
| String.replace(old, new, count=-1) | Заменяет подстроку old на new |
| String.isalpha() | Определяет: состоит ли строка целиком из буквенных символов |
| String.isdigit() | Определяет: состоит ли строка целиком из цифр |
| String.rjust(width[, fillchar = ‘ ‘]) | Расширяет строку, добавляя символы слева |
| String.ljust(width[, fillchar = ‘ ‘]) | Расширяет строку, добавляя символы справа |
| String.split(sep=None, maxsplit=-1) | Разбивает строку на подстроки |
| String.join(список) | Объединяет коллекцию в строку |
| String.strip() | Удаляет пробелы и переносы строк справа и слева |
| String.rstrip() | Удаляет пробелы и переносы строк справа |
| String.ltrip() | Удаляет пробелы и переносы строк слева |

Метод format и f-строки

### Форматирование без метода format

Существует также еще один способ форматирования с помощью следующего синтаксиса:

|  |  |
| --- | --- |
| 1 | строка%(параметр1, параметр2,..параметрN) |

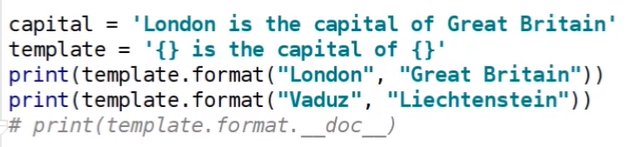
То есть в начале идет строка, которая содержит те же плейсхолдеры, которые были рассмотрены выше (за исключением плейсхолдера %), после строки ставится знак процента %, а затем список значений, которые вставляются в строку. Фактически знак процента представляют операцию, в результате которой образуется новая строка:

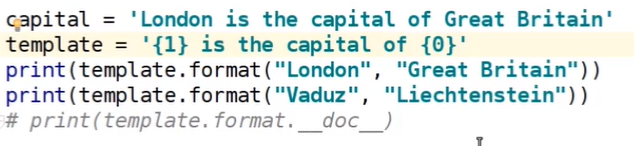
|  |  |
| --- | --- |
| 1  2 | info = "Имя: %s \t Возраст: %d" % ("Tom", 35)  print(info)   # Имя: Tom     Возраст: 35 |

Рядом с плейсхолдером указывается знак процента и в отличие от функции format здесь не требуются фигурные скобки.

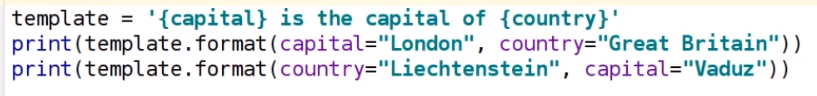
Причем способы форматирования чисел здесь также применяются:

|  |  |
| --- | --- |
| 1  2 | number = 23.8589578  print("%0.2f  - %e" % (number, number))   # 23.86  - 2.385896e+01 |





Именованные аргументы



import requests  
  
res = requests.get('https://docs.python.org/3.12/')  
print(res)  
print('Ответ от {0.url} с кодом {0.status\_code}'.format(res))  
print(f'Ответ от {res.url} с кодом {res.status\_code}')  
  
  
res = requests.get('https://docs.python.org/3.12/random')  
print(res)  
print('Ответ от {0.url} с кодом {0.status\_code}'.format(res))

Теория получилась несколько справочной. Но, что поделать, некоторые возможности языка лучше сразу выучить, чтобы потом не изобретать велосипед и подменять уже существующие методы своими фрагментами программного кода.

## Практика 2 Строки (методы)

1. Дана строка, состоящая из слов, разделенных пробелами. Определите, сколько в ней слов

print(input().count(' ') + 1)

1. Вводится строка. Необходимо определить число вхождений дефисов (-) в этой строке. На экране отобразить полученное число.

Входные данные:

osnovnye-metody-strok

Выходные данные:

2

2. Вводится слово. Необходимо первую букву этого слова сделать заглавной, а остальные - малыми. Результат отобразить на экране.

Входные данные:

HELLO

Выходные данные:

Hello

3. Вводится строка. С помощью метода String.find найдите в этой строке индекс первого вхождения фрагмента "ra". Полученное число выведите на экран.

Входные данные:

abrakadabra

Выходные данные:

2

4. Пользователь Василий очень часто злоупотребляет восклицательными знаками!

Напишите программу, которая будет заменять восклицательные знаки на точки.

Дана произвольная строка.

text = "Вася самый умный! Вася лучше всех! И ждет его успех!"

5. Вводится строка (слаг). Замените в этой строке все двойные дефисы (--) и тройные (---) на одинарные (-). Подумайте, в какой последовательности следует выполнять эти замены. Результат преобразования выведите на экран.

Входные данные:

dobavlyaem---slagi--slug-k--url---adresam

Выходные данные:

dobavlyaem-slagi-slug-k-url-adresam

6. Для повышения безопасности, пароли пользователей должны соответствовать требованиям. Проверьте, соответствует ли данный пароль предложенным требованиям.

Требования к паролю:

* Минимум 6 символов
* Должен содержать спецсимвол "#"
* Должен начинаться с большой буквы

Вывести "безопасный", если строка пароля соответствует требованиям безопасности или "не безопасный" в противоположном случае.

Дано Результат

a12345 не безопасный

F12#34 безопасный

F12$3 не безопасный

#Derpassword не безопасный

Derpassword# безопасный

Подсказка. Учитель забыл рассказать про метод проверки строки, начинающейся с большой буквы!

7. Вводятся три целых положительных числа (максимум трехзначные) через пробел в одну строчку. Для двухзначных и однозначных чисел нужно добавить слева незначащие нули так, чтобы все числа содержали по три цифры. Вывести на экран полученные числа в столбик.

Входные данные:

8 11 123

Выходные данные:

008

011

123

num\_1, num\_2, num\_3 = map(str, input().split())

# rjust(3, '0') возвращает выровненную по левому краю строку заданной минимальной ширины

# может вставлять необходимые символы переданные вторым аргументом

print(num\_1.rjust(3, '0'))

9. Вводится строка, состоящая из слов, разделенных пробелом. Необходимо подсчитать число слов в этой строке и результат (число) отобразить на экране.

Входные данные:

I love Python

Выходные данные:

3

a = input()

a = a.strip() # удаляем пробелы в начале и в конце строки

while a.count(' ') > 0: # считаем количество двойных пробелов

a.replace(' ', ' ') # меняем двойные пробелы на одинарные

a = a.count(' ') + 1 # считаем кол-во слов (кол-во пробелов между словами + 1)

print(a)

10. Вводится строка, состоящая из названий городов, разделенных пробелом. Необходимо преобразовать эту строку, чтобы названия городов шли через точку с запятой. Результат отобразить на экране.

Входные данные:

Москва Тверь Казань

Выходные данные:

Москва;Тверь;Казань

Метод startswith()

Метод startswith() возвращает True, если строка начинается с указанного префикса (строки). Если нет, возвращается False.

str.startswith(prefix[, start[, end]])

Параметры Метод в Python принимает не более трех параметров:

prefix ‒ строка или кортеж проверяемых строк;

start (необязательно) ‒ начальная позиция, в которой должен быть проверен префикс в строке; end (необязательно) ‒ конечная позиция, в которой необходимо проверить префикс в строке.

Метод startswith() возвращает логическое значение:

Он возвращает True, если строка начинается с указанного префикса.

Он возвращает False, если строка не начинается с указанного префикса.

Пример 1: Без параметров start и end

text = "Python is easy to learn."   
result = text.startswith('is easy') *# returns False*print(result)  
result = text.startswith('Python is ') *# returns True*print(result)  
result = text.startswith('Python is easy to learn.') *# returns True*print(result)

text = "programming is easy"  
result = text.startswith(('python', 'programming')) *# prints True*print(result)  
result = text.startswith(('is', 'easy', 'java')) *# prints False*print(result) *# With start and end parameter # 'is easy' string is checked*result = text.startswith(('programming', 'easy'), 12, 19) *# prints False*print(result)

Заключение

Методы startswith() и endswith() позволяют удобно проверять, начинается или заканчивается ли строка определенной подстрокой. Эти методы могут быть полезны при работе с файлами, при парсинге данных или при обработке строк в общем. Они просты в использовании и широко используются в Python.

# 2. Кодирование и декодирование строк

Если бы вас спросили, «Чем UTF-8 отличается от Unicode?», вы бы с уверенностью дали чёткий ответ? В наши дни интернационализации все разработчики должны это сделать.

Я думаю, что многие из нас не различают эти понятия должным образом. Изучим короткое введение в наборы символов и кодировки.

Итак, сначала было слово! И слово называлось Юникод



Пользователям Python повезло. Все символы и документы заранее приводятся к кодировке **UTF-8**.



И если где-то в коде вы напишете строку **«Ёжик в тумане»** или **«אֱלִיעֶזֶר וְהַגֶזֶר»**, то будьте уверены, они не превратятся в абракадабру и не приведут к ошибкам.

'Ёжик'.encode()

b'\xd0\x81\xd0\xb6\xd0\xb8\xd0\xba'

'Ёжик в тумане'.encode().decode()

'Ёжик в тумане'

'Ёжик в тумане'.encode().decode('utf-16')

'臐뛐룐뫐퀠₲苑菑볐냐뷐뗐'

Тем не менее, строковые методы **decode** / **encode** не потеряли свою актуальность. По умолчанию все преобразования осуществляются в кодировке **UTF-8**, но никто не мешает задать нужную вам.

Задача **encode()** – представить строку в виде объекта типа **bytes** (предваряется литералом **b**). Если знак относится к **ASCII**, то его байтовое представление будет выглядеть как оригинальный символ. В случае когда он выходит за пределы ASCII, то заменяется байтовым представлением (**\x** - эскейп-последовательность для обозначения 16-ричных чисел в языке Python).

Пример

'Python'.encode()

b'Python'

'Пайтон'.encode()

B'\xd0\x9f\xd0\xb0\xd0\xb9\xd1\x82\xd0\xbe\xd0\xbd'

Метод **decode()** преобразует последовательность байтов в привычную нам строку. Если задать другую кодировку, то можем получить либо неожиданный результат (как в последнем примере, где русские слова превратились в набор корейских иероглифов), либо ошибку **UnicodeDecodeError**, если в требуемой кодировке нет такого сочетания байтов или они выходят за ее границы.

Т.е. на самом деле, сравнивать UTF-8 и Unicode — это как сравнивать яблоки и апельсины:

UTF-8 — это кодировка; Unicode — это набор символов.

Набор символов — это список символов с уникальными номерами (эти номера иногда называют «code points»). Например, в наборе символов Unicode, номер «41» соответствует английской букве «A». Кодировка — это алгоритм, который преобразует числа (номера в наборе символов) в двоичный код, понятный машине.

Последний пример:

print ('Вася'.encode('utf-8'))  
print ('Вася'.encode('utf-16'))  
  
s = list ('Вася'.encode('utf-8'))  
print(s)

Результат

b'\xd0\x92\xd0\xb0\xd1\x81\xd1\x8f'

b'\xff\xfe\x12\x040\x04A\x04O\x04'

[208, 146, 208, 176, 209, 129, 209, 143]

Пример с эмодзи:

print('\u25A0')  
print('\u2660')  
print('\u2665')

Результат:

■

♠

♥

<https://emojinarium.com/ru/>

Теперь, когда кто-нибудь спросит Вас, «Чем UTF-8 отличается от Unicode?», вы сможете с уверенностью ответить:

UTF-8 и Unicode нельзя сравнивать. UTF-8 — это кодировка, которая используются для перевода двоичных данных в числа. Unicode — это набор символов, который используется для преобразования чисел в символы.

## Байтовая строка

В примере выше мы уже разобрали такие строки

Байт - минимальная единица хранения и обработки цифровой информации. Последовательность байт представляет собой какую-либо информацию (текст, картинку, мелодию...). !!! Файл pdf, img, jpeg

Байтовая строка – это неизменяемая последовательность чисел от 0 до 255. b - литерал для объявления байтовой строки:

<class 'bytes'>

b1 = b"hello"

*b2 = b"привет"*

Однако, если мы, например, попытаемся преобразовать строку "привет" в байты, мы получим ошибку. Здесь строка состоит из юникодных символов, которые преобразуются в байты при помощи кодировок.

b4 = b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'  
print(b1)  
print(b4.decode())

Самая известная кодировка - это UTF-8. С помощью Python мы можем преобразовать строку в байтовую строку с использованием кодировки UTF-8.

s = 'Привет'  
encoded\_string = s.encode(encoding="utf-8")  
print(encoded\_string)

# \xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'  
print(encoded\_string.decode())

## Набор символов ASCII

import string  
  
all = string.printable  
  
print(all)  
print(string.ascii\_letters)  
print(string.digits)  
print(string.punctuation)

Откуда взять русский алфавит в Python

cyrillic\_lower = [chr(i) for i in range(1072, 1104)]

print(cyrillic\_lower)

или a = ord('а')  
''.join([chr(i) for i in range(a, a + 32)])  
*# 'абвгдежзийклмнопрстуфхцчшщъыьэюя'*

## Спецсимволы, экранирование символов, raw-строки

Теперь, когда познакомились со строками и их методами, пришло время узнать, какие специальные символы могут содержать строки в Питоне. С одним из них мы уже сталкивались – это символ перевода строки:

‘\n’

Я напомню, например, когда задается многострочная строка:

text = """hello

python"""

то в ней автоматически добавляет этот символ перевода между строками:

'hello\npython'

Причем, это один символ, хотя он и выглядит как два символа: обратный слеш и n. Мы в этом легко можем убедиться, если воспользоваться функцией:

len(text)

Получим значения 12 = 5 + 6 + 1 – как раз число символов в двух строках плюс один символ перевода строки.

Мало того, мы можем его явно прописывать в любой строке, формируя многострочный текст, например, так:

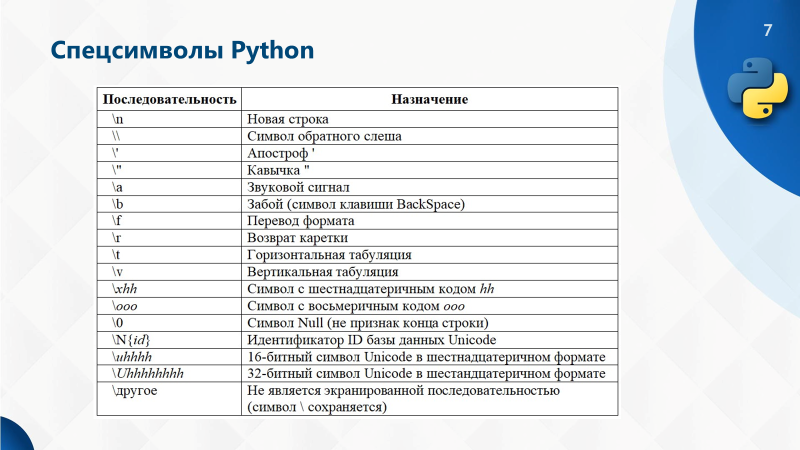
t = "panda needs\npython"

и, выводя эту строку с помощью функции:

print(t)

увидим две строки.

Вообще, в строках языка Python можно прописывать следующие спецсимволы:



Все их запоминать совсем не обязательно, на практике используются, в основном:

\n, \\, \’, \", \t

Значительно реже другие варианты. И, обратите внимание, перед каждым спецсимволом записан символ обратного слеша. Это, своего рода, маркер начала спецсимвола. И если после слеша идет одно из обозначений таблицы, то оно будет восприниматься как некая управляющая последовательность.

Давайте я все это продемонстрирую на примерах. Добавим в строку символ табуляции:

t = "\tpanda needs\npython"

Теперь функция print() интерпретирует его, как особый горизонтальный отступ:

print(t)

Если же мы уберем букву t:

t = "\panda needs\npython"

то при печати увидим просто обратный слеш. В действительности, здесь сработала последняя строчка таблицы: когда не подходит ни одна последовательность, то просто печатается обратный слеш.

Но здесь нужно быть осторожным. Предположим, что мы слово needs хотим заключить в обратные слеши:

t = "panda \needs\ python"

Однако, при печати:

print(t)

первый слеш пропадет, так как он будет восприниматься началом спецпоследовательности символа переноса строки. Поэтому, для добавления символа обратного слеша в строку, следует записывать два обратных слеша подряд:

t = "panda \\needs\\ python"

Тогда в строке они будут автоматически заменены на один символ слеша и при выводе мы это и видим. Это называется экранированием, когда мы символы с двойным назначением записываем, добавляя перед ними обратный слеш. В данном случае получаем двойной слеш.

Часто такие символы следует прописывать при определении путей к файлам. Как мы знаем, в ОС Windows маршруты имеют вид:

D:\Python\Projects\folder\tex1.py

Здесь фигурируют обратные слеши для разделения каталогов. Чтобы правильно описать такой путь, слеши следует экранировать:

path = "D:\\Python\\Projects\\folder\\tex1.py"

и при печати:

print(path)

видим, что маршрут определен верно. Если бы слеши не были экранированы, то получили бы неверный путь к файлу:

path = "D:\Python\Projects\folder\tex1.py"

Вот этот момент следует хорошо запомнить.

Кроме обратного слеша экранировать также следует и кавычки. Например, мы хотим сформировать строку:

s = "Марка вина "Ягодка""

Внутри этой строки имеются кавычки. Но эти же самые кавычки определяют начало и конец строки в Python. Поэтому такая запись приведет к синтаксической ошибке. Чтобы все работало корректно, нужно выполнить экранирование кавычек:

s = "Марка вина \"Ягодка\""

Или, в данном случае, можно было бы использовать одинарные кавычки для определения строки, а внутри нее записать двойные:

s = 'Марка вина "Ягодка"'

Но на практике рекомендуется всегда выполнять экранирование таких символов, чтобы избежать случайных ошибок. Например, в этой строке уже нельзя просто так записать одинарные кавычки:

s = 'Марка вина 'Ягодка''

Снова получим синтаксическую ошибку, их нужно экранировать:

s = 'Марка вина \'Ягодка\''

В завершение этого занятия отмечу, что в Python можно задавать, так называемые, сырые (row) строки. Это строки, в которых игнорируются спецпоследовательности и все символы воспринимаются буквально так, как записаны. Например, если взять строку с путем к файлу:

path = "D:\\Python\\Projects\\folder\\tex1.py"

то сейчас, при отображении, мы видим по одному слешу:

print(path)

Но, если определить эту же строку, как сырую, добавив букву r перед ней:

path = r"D:\\Python\\Projects\\folder\\tex1.py"

то при печати увидим по два слеша, именно так, как прописали. Поэтому, в таких строках можно убрать спецопределения и записывать строку буквально так, как она должна выглядеть:

path = r"D:\Python\Projects\folder\tex1.py"

На этом мы завершим наше очередное занятие по Python. Из него вы должны хорошо себе представлять, что такое спецсимволы и экранирование символов. Какие основные спецсимволы для строк существуют и как определяются сырые строки.

## Практика 3 Строки (спецсимволы, экранирование, raw)

1. Необходимо задать строку со следующим содержимым: Тема занятия "спецсимволы". И отобразить ее на экране (кавычки у слова спецсимволы также должны быть отображены).

2. Вводится два слова в одну строку через пробел. Поставьте между этими словами символ обратного слеша (вместо пробела). Результирующую строку отобразите на экране.

P. S. Задачу реализовать без применения F-строк.

Входные данные:

Hello Pushkin!

Выходные данные:

Hello\Pushkin!

3. Используя raw-строки, задайте строку, содержащую этот путь к файлу: C:\WINDOWS\System32\drivers\etc\hosts. Результат отобразите на экране.

Входные данные:

Выходные данные:

C:\WINDOWS\System32\drivers\etc\hosts

4. Вводится слово. Необходимо сформировать новую строку, где введенное слово будет заключено в двойные кавычки. Результат выведите на экран.

Входные данные:

language

Выходные данные:

"language"

5. Вводится строка. Необходимо найти все индексы фрагмента "ра" во введенной строке. Вывести в строку через пробелы найденные индексы. Если этот фрагмент ни разу не будет найден, то вывести значение -1.

Входные данные:

Барабанщик бил бой в барабан

Выходные данные:

2 23

text = 'Барабанщик бил бой в барабан'  
res\_count = 0  
  
for i in range(0, text.count('ра')):  
 res\_count = text.find('ра', res\_count)  
 print(res\_count, end=' ')  
 res\_count += 1

# 3. Оператор цикла while

На этом занятии начнем знакомиться с еще одним ключевым элементом компьютерных программ – циклами. Вначале, давайте я на простом примере покажу, о чем идет речь. Представим, что нам нужно вычислить сумму целых чисел от 1 до N.

## Бесконечный цикл

Простой пример

import time  
  
while 1:  
 print('working ... ')  
 time.sleep(1)

далее

x = 1  
while 1:  
 print(x, 'working ... ')  
 time.sleep(1)

Результат

1 working ...

1 working ...

1 working

Инкрементируем

while 1:  
 print(x, 'working ... ')  
 time.sleep(1)  
 x += 1

1 working ...

2 working ...

3 working ..

## Конструкция while - else

Далее

x = 1  
while x < 5:  
 print(x, 'Working ... ')  
 time.sleep(1)  
 x += 1  
else:  
 print ('Happy End')  
print ('Total End')

и так далее

## Конструкция while – break-else

x = 1  
while x < 5:  
 print(x, 'Working ... ')  
 time.sleep(1)  
 x += 1  
 if x==3:  
 print('Bad end')  
 break  
  
else:  
 print ('Happy end')  
print ('Total end')

Применение как условного зацикленного оператора

password = ''  
  
while password != '2023':  
 password = input('Введи пароль ->')  
  
print('Авторизация успешна')

А теперь в одну строчку

password = ''  
x = 0  
while password != '2023': password = input(f'попытка: {x} \nВведи пароль ->');x += 1  
print('Авторизация успешна')

Вначале мы задаем пароль для проверки, затем, переменную ps, которая хранит введенный пользователем пароль и делаем цикл, пока пароли не совпадают. Обратите внимание, условие цикла – пока пароли не совпадают. Опять же, всегда следует помнить, что мы прописываем условие работы, а не остановки цикла. Поэтому здесь нам нужно проверять на несовпадение паролей и запрашивать у пользователя пароль, пока они не совпадут.

Моржовый оператор

Добавленный в Python 3.8 моржовый оператор (:=), формально известен как оператор присваивания выражения. Он дает возможность присвоить переменные в выражении, включая переменные, которых еще не существует. Как было сказано выше, с помощью простого оператора присваивания (=) мы назначили *num* равным 15 в контексте отдельного оператора.

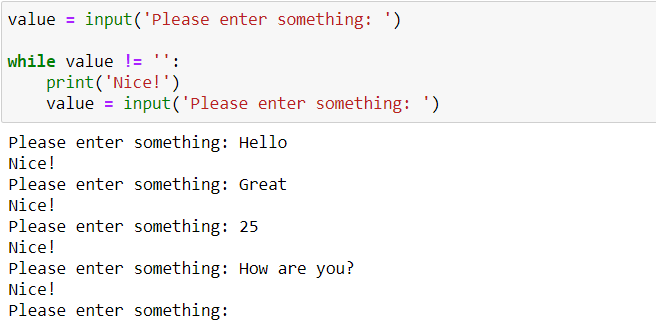
Лучший способ понять, как работает моржовый оператор – этот простой пример. Как и выше, мы хотим присвоить 15 переменной num, а затем вывести значение num. Мы можем выполнить обе эти задачи в одной строке кода, используя новый оператор следующим образом:

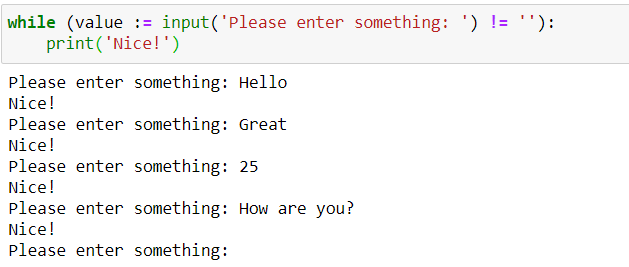
print(num := 15)

# 15

print(num = 15)

# TypeError





Функция divmod

a = divmod(23,10)  
print(a) #(2,3)

Сумма чисел в числе

x = 12345  
s = 0  
while x > 0:  
 x,b = divmod(x, 10)  
 s+=b  
print(s)

Перевод в другие системы счисления

x = 7  
s = ''  
while x > 0:  
 x, b = divmod(x, 2)  
 s = str(b) + s  
  
print(s)

s= [1,2,3]  
  
print(f'{s = }')

## Практика 4 Цикл while

1. "Обратный отсчет". Выведите все целые числа от n до 0 включительно.

Формат входных данных

Дано целое положительно число n.

Формат выходных данных

Выведите все числа, требуемые по условию задачи.

2. Задание. Даны два целые числа a и b.

Выведите на экран все целые четные числа от a до b включительно.

Формат входных данных

Даны два целые числа. Гарантируется, что a < b.

3. Задание. "Лесенка символами"

Вывести лесенку из n ступенек символами \*.

Примеры

n = 3

\*

\*\*

\*\*\*

n = 5

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

Часто возникает вопрос: а чем блок else отличается от блока операторов, просто идущих после блока while? Ведь когда цикл while завершается, мы так и так переходим к последующим операторам! Но обратите внимание вот на эту на фразу «штатное завершение цикла». Штатное завершение – это когда условие цикла стало равно False и оператор while прекращает свою работу. Только в этом случае мы перейдем в блок else. Давайте я покажу это на конкретном примере. Предположим, что мы вычисляем сумму вида:

S = 1/2 + 1/3 + 1/4 + 1/10 + … + 1/0

И если в этом сумме встречается деление на ноль, то вычисления следует прервать. Реализуем эту программу, следующим образом:

S=0

i=-10

while i < 100:

    if i == 0:

        break

    S += 1/i

    i += 1

else:

    print("Сумма вычислена корректно")

print(S)

Смотрите, если здесь при вычислении суммы встречается деление на 0, то срабатывает break и цикл досрочно прерывается, то есть, завершается в нештатном режиме. В этом случае блок else пропускается и мы не видим сообщения, что сумма вычислена корректно. Если же все проходит штатно и цикл завершается по своему условию, то в консоли появляется сообщение «Сумма вычислена корректно», означающее выполнение блока else.

## Оператор continue

Оператор continue используется для указания Python, что необходимо пропустить все оставшиеся команды в текущем блоке цикла и продолжить со следующей итерации цикла.

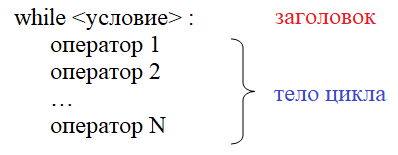
while True:  
 s = input('Введите что-нибудь : ')  
 if s == 'выход':  
 break  
 if len(s) < 3:  
 print('Слишком мало')  
 continue  
 print('Введённая строка достаточной длины')

В этой программе мы запрашиваем ввод со стороны пользователя, но обрабатываем введённую строку только если она имеет длину хотя бы в 3 символа. Итак, мы используем встроенную функцию len для получения длины строки, и если длина менее 3, мы пропускаем остальные действия в блоке при помощи оператора continue. В противном случае все остальные команды в цикле выполняются, производя любые манипуляции, которые нам нужны

Циклы позволяют реализовывать некие повторяющиеся действия. Например, предположим, что маленькие панды прыгают с горки в течение часа, пока мама-панда не позовет всех к столу – кушать. На уровне текста это можно записать, так:

цикл (пока не прошел час):  
      прыгаем с горки

То есть, пока истинно условие, цикл работает, как только условие становится ложным – прошел час, цикл завершается. Ровно так работает цикл while, о котором и пойдет речь на нашем занятии. Он имеет, следующее определение (синтаксис):



Вначале записывается ключевое слово while, затем, условие работы цикла, ставится двоеточие для определения блока операторов, работающих внутри этого цикла. Такой блок еще называют телом цикла, а ключевое слово while с условием – заголовком цикла.

Обратите внимание на форматирование. Здесь также, как и в условных операторах, набор операторов внутри тела цикла должны иметь одинаковые отступы относительно ключевого слова while.

Давайте вернемся к исходной задаче – вычисления суммы чисел от 1 до N и посмотрим, как здесь нам поможет цикл while. Вначале определим три вспомогательные переменные: N – значение последнего слагаемого; s – для хранения вычисленной суммы (начальное значение 0); i – значение текущего слагаемого (начинается с 1):

N = 1000

s = 0

i = 1

Далее, так как сумму нужно вычислять, пока слагаемое i не достигнет значения N, то условие цикла можно определить, следующим образом:

while i <= N:

А внутри цикла будем выполнять следующие действия:

s += i

i += 1

Вначале i равна 1 и эта единица прибавляется к сумме s. После чего i увеличивается на 1 и становится равной 2. Затем, выполняется проверка условия цикла. Пока оно у нас истинно, поэтому снова попадаем внутрь тела цикла и к s прибавляется уже значение 2, а i опять увеличиваем на 1 и оно становится равным 3. И так до тех пор пока i не станет больше N. К этому моменты мы просуммируем все числа и результат будет храниться в переменной s. Вот принцип работы циклов, причем, во всех языках программирования, не только в Python.



Также однократное выполнение тела цикла в программировании называют итерацией. Я буду часто использовать этот термин, поэтому привел его, чтобы вы меня правильно понимали.

Давайте реализуем теперь эту программу на Python и посмотрим, как она сработает.

Возможно, у вас возник вопрос: а какие условия можно прописывать в циклах? В действительности, все те же самые, что и в условных операторах. В том числе и составные. Например, давайте будем вычислять сумму пока не дойдем до слагаемого N или до значения 50. Так как цикл работает, пока истинно условие, то его следует записать, так:

while i <= N and i <= 50:

...

Смотрите, мы здесь указали делать цикл пока i меньше или равна N и меньше или равна 50. Если хотя бы одно из этих подусловий окажется ложным, то и все составное условие также станет ложным и цикл завершится. В результате, i достигнет или N или 50. Вот это нужно хорошо себе представлять: в циклах прописываются условия их работы, а не завершения.

Используя этот же цикл, мы легко можем поменять условие задачи и вычислить сумму чисел, которые меняются через один:

1, 3, 5, 7, …

Для этого достаточно счетчик i увеличивать не на 1, а сразу на два:

i += 2

При этом, условие цикла можно оставить прежним. Он завершится, как только i превысит N или 50. Здесь уже нет гарантии, что последнее слагаемое будет именно N или 50, но нам это и не нужно, мы лишь указываем завершить цикл, когда превысим одно из этих значений.

Конечно, внутри тела цикла можно записывать любые операторы, в том числе и функцию print(). Давайте, например, отобразим в консоли цифры от 0 до 9:

i = 0

while i < 10:

    print(i)

    i += 1

Я здесь указал условие i меньше 10, а не i <= 9, так как оператор < работает несколько быстрее оператора <=. Поэтому на практике предпочтительно, по возможности, применять не составные, а простые операторы: меньше, больше, равно и не равно. Хотя использование <= или >= не критично и вполне допустимо. Но, все же, по возможности, лучше прописывать простые операторы в условиях циклов.

Если нам нужно реализовать убывающую последовательность чисел, например:

-1, -2, -3, …, -N

то это делается аналогично, только с уменьшающимся счетчиком:

N = -10

i = -1

while i >= N:

    print(i)

    i -= 1

Вообще, счетчик в цикле можно менять произвольным образом, например, умножая на два:

i \*= 2

Здесь нет никаких ограничений.

Наконец, внутри цикла while можно прописывать, например, и условные операторы. Давайте выведем все числа, кратные 3, которые нам встретятся при переборе целых значений от 1 до N:

N = 20

i = 1

while i <= N:

    if i % 3 == 0:

        print(i)

    i += 1

Как видите, все достаточно просто и очевидно. Здесь главное не забывать о правильном форматировании текста программы: функция print() находится внутри условия if, поэтому перед ней необходимы отступы.

А сам блок условия имеет отступы относительно оператора while. Так определяются вложенные блоки операторов в языке Python. И об этом всегда следует помнить.

На этом мы завершим первое знакомство с оператором цикла while. На следующем уроке продолжим эту тему. А для закрепления текущего материала не забудьте выполнить несколько практических заданий.

## Разбор игры Угадай число

Частный случай линейного поиска

<https://www.cs.usfca.edu/~galles/visualization/Search.html>

## Практика 4 Цикл while и строки

1. Строка состоит из символов *X*, *Y* и *Z*. Определите длину самой длинной последовательности, состоящей из символов *X*.

Входные данные:



Выходные данные

9

**2.** Вводятся два целых положительных числа n и m, причем, n < m. Вывести в строку через пробел квадраты целых чисел в диапазоне [n; m]. Программу реализовать при помощи цикла while.

**Входные данные:**

2 4

**Выходные данные:**

4 9 16

**3.** Вводится стоимость одной книги x рублей (вещественное число). Необходимо вывести на экран в строчку через пробел стоимости 2, 3, ... 10 таких книг с точностью до десятых. Программу реализовать при помощи цикла while.

**Входные данные:**

34.6

**Выходные данные:**

69.2 103.8 138.4 173.0 207.6 242.2 276.8 311.4 346.0

**3.** Вводится целое положительное число n. Вычислить и вывести на экран сумму: 1 + 1/2 + 1/3 + ... + 1/n с точностью до тысячных (три знака после запятой). Программу реализовать при помощи цикла while.

**Входные данные:**

8

**Выходные данные:**

2.718

Задание 5. Подсчитать количество гласных(русских) букв во введенной строке без учета регистра.

text = "В теории, теория и практика неразделимы. На практике это не так."  
vowels = "ауоыэяюёие"

Задание 6. "Подсчет длинных слов"

Определить в предоставленном сообщении количество слов длиной больше, чем 5.

Дана строка текста, слова разделены пробелами, знаки препинания отсутствуют.

text = "Lorem ipsum dolor sit amet consectetur adipiscing elit Integer porttitor bibendum nisi ut convallis ante"

## Практика 5 while + break

**1.** На каждой итерации цикла пользователь вводит целое число. Цикл продолжается, пока не встретится число 0. Необходимо вычислить сумму введенных в цикле чисел и вывести результат на экран. Программу реализовать при помощи цикла while.

**Входные данные:**

8

11

2

-4

0

**Выходные данные:**

17

**2.** Последовательность Фибоначчи образуется так: первые два числа равны 1 и 1, а каждое последующее равно сумме двух предыдущих. Имеем такую последовательность чисел: 1, 1, 2, 3, 5, 8, 13, ... Постройте последовательность Фибоначчи длиной n (n вводится с клавиатуры). Результат отобразите в виде строки полученных чисел, записанных через пробел. Программу реализовать при помощи цикла while.

**Входные данные:**

8

**Выходные данные:**

1 1 2 3 5 8 13 21

**3.** Одноклеточная амеба каждые 3 часа делится на 2 клетки. Определить, сколько клеток будет через n часов (n - целое положительное число, вводимое с клавиатуры). Считать, что изначально была одна амеба. Результат вывести на экран. Задачу необходимо решить с использованием цикла while.

**Входные данные:**

11

**Выходные данные:**

8

**4.** Гражданин 1 января открыл счет в банке, вложив 1000 руб. Каждый год размер вклада увеличивается на 5% от имеющейся суммы. Определить сумму вклада через n лет (n - целое положительное число, вводимое с клавиатуры). Результат округлить до сотых и вывести на экран. Программу реализовать при помощи цикла while.

**Входные данные:**

5

**Выходные данные:**

1276.28

**5.** Составить программу поиска всех трехзначных чисел, которые при делении на 47 дают в остатке 43 и кратны 3. Вывести найденные числа в строчку через пробел. Программу реализовать при помощи цикла while.

**Входные данные:**

**Выходные данные:**

231 372 513 654 795 936

**6.** Вводится список названий городов в одну строчку через пробел. Определить, что в этом списке все города имеют длину более 5 символов. Реализовать программу с использованием цикла while и оператора break. Вывести ДА, если условие выполняется и НЕТ - в противном случае.

**Входные данные:**

Самара Ульяновск Новгород Воронеж

**Выходные данные:**

ДА

**7.** Вводится список имен студентов в одну строчку через пробел. Определить, что хотя бы одно имя в этом списке начинается и заканчивается на ту же самую букву (без учета регистра). Реализовать программу с использованием цикла while и оператора break. Вывести ДА, если условие выполняется и НЕТ - в противном случае.

**Входные данные:**

Петр Анна Иван Сергей Михаил Федор

**Выходные данные:**

ДА

# 4. Тип данных range

Тип range - диапазон, представляет собой неизменяемую последовательность чисел и обычно используется для выполнения определенного количества раз циклов for.

Для диапазонов доступны общие операции с последовательностями, кроме конкатенации и повторения, поскольку объекты диапазона могут представлять только последовательности, которые следуют строгому шаблону, а повторение и конкатенация обычно нарушают этот шаблон.

## Примеры использования:

>>> r = range(0, 20, 2)

>>> r

*# range(0, 20, 2)*

>>> 11 **in** r

*# False*

>>> 10 **in** r

*# True*

>>> r.index(10)

*# 5*

>>> r[5]

*# 10*

>>> r[:5]

*#range(0, 10, 2)*

>>> r[-1]

*# 18*

**Длина диапазона**: len(r)

data = range(10)

print(len(data))

#10

**Минимальный элемент диапазона**: min(r)

data = range(10)

print(min(data))

#0

**Максимальный элемент диапазона**: max(r)

data = range(10)

print(max(data))

#9

**Индекс числа в диапазоне**: range.index(x[, start[, end]])  
Необязательные аргументы start и end, указывают на промежуток поиска:

data = range(10)

print(data.index(5))

#5

data = range(2, 10)

print(data.index(5))

#3

Таким образом, тип range (диапазон) имеет массу путей применения, поскольку позволяет легко автоматизировать процесс заполнения различных наборов чисел. Как правило, для этой цели применяется функция range, в которой указан начальный элемент, граница и шаг последовательности. Благодаря реализации интерфейса коллекции, диапазоны обеспечивают проверку вхождения, поиск по индексу, срез и отрицательную индексацию элементов.

# 5. Оператор цикла for.

На предыдущих занятиях мы с вами познакомились с оператором цикла while, а также вспомогательными операторами break, continue и else. На этом занятии вы узнаете о втором операторе цикла for, который довольно часто используется в Python.

Он имеет следующий синтаксис:

for <переменная> in <итерируемый объект>:  
      оператор 1  
      оператор 2  
       …  
      оператор N

Пример

for i in range(1, 5):  
 print(i)  
else:  
 print('Цикл for закончен штатно')  
print ('Конец программы')

С его помощью очень легко реализовывать перебор, так называемых, итерированных объектов. Что это такое, мы будем говорить на одном из следующих занятий, а сейчас, вам достаточно знать, что это объекты, состоящие из множества элементов, которые можно перебирать. Например, списки или строки.

Как всегда, постичь магию работу этого оператора лучше всего на конкретных примерах. Пусть у нас имеется список:

d = [1, 2, 3, 4, 5]

И мы хотим перебрать все его элементы. Через оператор цикла for сделать это можно, следующим образом:

for x in d:

    print(x)

Или, вместо списка, можно взять строку:

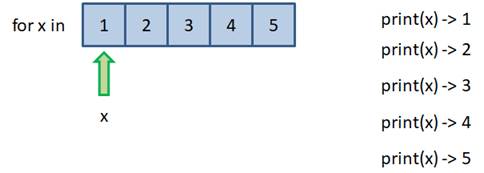
for x in "python":  
 print(x)

Тогда переменная x на каждой итерации будет ссылаться на очередной символ этой строки.

Пронумеруем

for x, y in enumerate("python"):  
 print(x, y, sep='=')

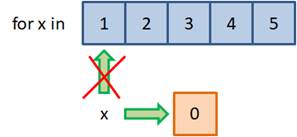
Но давайте посмотрим, как все это работает в деталях. Вернемся к примеру со списком. На первой итерации переменная x будет ссылаться на первый элемент со значением 1. Соответственно, функция print() выводит это значение в консоль. На следующей итерации переменная x ссылается уже на второй элемент и print() выводит значение 2. И так до тех пор, пока не будет достигнут конец списка.



В этой демонстрации ключевое, что переменная x ссылается на элемент списка. То есть, если мы захотим изменить значение в списке, используя переменную x, например, вот так:

for x in [1, 2, 3, 4, 5]:

    x = 0



то ничего не получится. Здесь x просто будет ссылаться на другой объект со значением 0, но элементы списка это никак не затронет. То есть, в такой реализации оператора цикла for мы можем лишь перебирать значения элементов и что-то с ними делать, например, вычислять их произведение:

d = [5, 2, 4, 3, 1]

p = 1

for x in d:

    p \*= x

print(p)

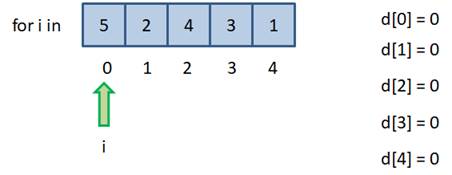
А как тогда менять значения элементов в списке с помощью for? Для этого к элементам списка нужно обращаться по индексу. То есть, цикл должен перебирать не элементы списка, а его индексы:

d = [5, 2, 4, 3, 1]

for i in [0, 1, 2, 3, 4]:

    d[i] = 0

print(d)



В этом случае, мы на каждой итерации цикла, обращаемся сначала к первому элементу списка, присваиваем ему ноль, затем, ко второму элементу, присваиваем ему ноль и так для всех остальных элементов. В итоге меняется сам список.

Однако описывать индексы через еще один список, далеко не лучшая практика. Для подобных целей в Python существует специальная функция range(), которая генерирует арифметическую последовательность чисел с параметрами:

Range(start, stop, step)

Например, для генерации последовательности от 0 до 4, функцию range() можно записать в виде:

range(5)

range(0, 5)

range(0,5,1)

То есть, последнее значение 5 не включается в диапазон. Чтобы увидеть сгенерированные числа, преобразуем их последовательность в список с помощью известной нам функции list():

list(range(5))

list(range(0, 5))

list(range(0, 5, 1))

Если же мы запишем:

list(range(0))

то получим пустой список, так как значения здесь начинаются с нуля и заканчиваются нулем, при этом, ноль в интервал не входит.

То же самое произойдет при указании любого отрицательного значения, например:

list(range(-5))

list(range(0, -5))

Но как нам тогда формировать последовательность отрицательных значений? Очень просто, в качестве стартового значения нужно указать число меньше конечного значения:

list(range(-10, -5))

Или с шагом:

list(range(-10, -5, 2))

Но если указать отрицательный шаг:

list(range(-10, -5, -2))

то снова увидим пустой список. Я, думаю, вы догадались почему? Теперь, мы начинаем двигаться от -10 в меньшую сторону и значение -5 становится недостижимым. В этом случае функция range() не выдает никаких значений. Чтобы поправить ситуацию, в качестве конечного значения нужно записать число меньше -10, например, -20:

list(range(-10, -20, -2))

Вот так генерируются последовательности в обратном порядке. То же самое можно проделать и в положительной области:

list(range(5, 0, -1))

Обратите внимание, мы начинаем с 5 и заканчиваем 1. Здесь конечное значение 0 не включается в диапазон. Если нам нужно дойти до нуля, то в данном случае следует указать -1 в качестве конечного значения:

list(range(5, -1, -1))

Вот это всегда следует помнить при работе с функцией range() – конечное значение не включается в диапазон.

Итак, теперь, когда мы с вами узнали, как работает функция range(), перепишем нашу программу с перебором элементов списка по их индексам, следующим образом:

d = [5, 2, 4, 3, 1]

for i in range(5):

    d[i] = 0

print(d)

Здесь нам не нужно функцию range() превращать в список, оператор цикла for умеет перебирать любые итерируемые объекты, а range(), как раз возвращает такой объект, поэтому переменная i будет принимать значения от 0 до 4 включительно и мы видим, что все значения списка стали равны нулю.

И последний штрих в этой программе. Число 5 лучше заменить вызовом функции определения длины списка:

len(d)

Тогда получим универсальную программу, работающую со списком любой длины:

for i in range(len(d)):

    d[i] = 0

В заключение этого занятия приведу еще один пример использования цикла for для вычисления суммы ряда:

S = 1/2 + 1/3 + 1/4 + … + 1/1000

Программу можно реализовать, следующим образом:

S = 0

for i in range(2, 1001):

    S += 1 / i

print(S)

На этом мы завершим наше первое знакомство с оператором цикла for и функцией range(). На данный момент вам нужно хорошо понимать, как перебираются списки и строки, как формировать арифметические последовательности функцией range() и как ее можно использовать совместно с оператором цикла for.

## Практика 6 for

**3.** С помощью функции range() сформируйте следующую последовательность чисел:

-10, -8, -6, -4, -2

Результат выведите в виде последовательности чисел, записанных через пробел в одну строчку.

**Входные данные:**

**Выходные данные:**

-10 -8 -6 -4 -2

**4.** С помощью функции range() сформируйте следующую последовательность чисел:

1, 4, 7, 10, 13, 16, 19

Результат выведите в виде последовательности чисел, записанных через пробел в одну строчку.

**Входные данные:**

**Выходные данные:**

1 4 7 10 13 16 19

**5.** Вводятся целые числа в одну строчку через пробел. Необходимо преобразовать эти данные в список целых чисел. Затем, перебрать этот список в цикле for и просуммировать все нечетные значения. Результат вывести на экран.

**Входные данные:**

8 11 -2 4 0 13 19 12 7

**Выходные данные:**

50

**6.** Вводятся названия городов в одну строчку через пробел. Необходимо преобразовать входные данные в список. Затем, перебрать его циклом for и заменить значения элементов на длину названия соответствующего города. Результат вывести на экран в виде последовательности чисел через пробел в одну строчку.

**Входные данные:**

Москва Уфа Караганда Тверь Минск Казань

**Выходные данные:**

6 3 9 5 5 6

7. Автомат получает на вход трёхзначное число. По этому числу строится новое число по следующим правилам.

1.  Складываются первая и вторая, а также вторая и третья цифры исходного числа.

2.  Полученные два числа записываются друг за другом в порядке убывания (без разделителей).

Пример. Исходное число: 348. Суммы: 3 + 4 = 7; 4 + 8 = 12. Результат: 127. Укажите наименьшее число, в результате обработки которого автомат выдаст число 1412.

Ответ: 395.

**for** i **in** range(100, 1000):

s = str(i)

k1 = int(s[0]) + int(s[1])

k2 = int(s[1]) + int(s[2])

first = str(max(k1, k2))

second = str((min(k1, k2)))

s1 = first + second

**if** s1 == '1412':

**print**(i)

**break**

**9.** Вводится натуральное число n. С помощью цикла for найти все делители этого числа. Найденные делители выводить сразу в столбик без формирования списка.

**Входные данные:**

12

**Выходные данные:**

1

2

3

4

6

12

**10.** Вводится натуральное число n. С помощью цикла for определить, является ли оно простым (то есть, делится нацело только на само себя и на 1). Вывести на экран ДА, если n простое и НЕТ - в противном случае.

**Входные данные:**

11

**Выходные данные:**

ДА

11. Напишите программу, которая ищет среди целых чисел, принадлежащих числовому отрезку [2422000; 2422080], простые числа. Выведите все найденные простые числа в порядке возрастания, слева от каждого числа выведите его номер по порядку, считая, что первое найденное число имеет номер «1», второе  — «2», и т. д.

k = 0  
**for** n **in** range(2422000,2422080):  
 **if** all(n%i !=0 **for** i **in** range(2,int(n\*\*0.5))):  
 k+=1  
 print(k,n)

**9.** Вводится список названий городов в одну строчку через пробел. Перебрать все эти названия с помощью цикла for и определить, начинается ли название следующего города на последнюю букву предыдущего города в списке. Если последними встречаются буквы 'ь', 'ъ', 'ы', то берется следующая с конца буква. Вывести на экран ДА, если последовательность удовлетворяет этому правилу и НЕТ - в противном случае.

**Входные данные:**

Москва Астрахань Новгород Димитровград Душанбе

**Выходные данные:**

ДА

**10.** Вводится натуральное число n. Вычислить сумму всех натуральных чисел меньше n, которые кратны или 3 или 5. Результат (сумму) вывести на экран. Пример: n = 10, имеем числа: 3, 5, 6, 9. Их сумма равна 23.

**Входные данные:**

21

**Выходные данные:**

98

11. Вводится список в виде целых чисел в одну строку через пробел. Необходимо сначала сформировать список на основе введенной строки, а затем, каждое значение этого списка изменить, возведя в квадрат. Отобразить результат на экране в виде строки полученных чисел, записанных через пробел..

Входные данные:

8 -11 4 3 6

Выходные данные:

64 121 16 9 36

**12.** Вводится список в виде целых чисел в одну строку через пробел. Сначала нужно сформировать список из введенной строки. Затем, каждый элемент этого списка продублировать один раз. Результат отобразить на экране в виде строки полученных чисел, записанных через пробел.

**Входные данные:**

8 11 2

**Выходные данные:**

8 8 11 11 2 2

## Примеры работы оператора цикла for. Функция enumerate()

Мы продолжаем знакомиться с оператором цикла for языка Python. Сегодня мы с вами рассмотрим несколько примеров использования цикла for при решении разных задач. И первая задача – вычислить факториал от натурального числа n. Пусть пользователь вводит натуральное целое число, а мы будем выдавать значение его факториала:

n = int(input("Введите натуральное число не более 100: "))

if n < 1 or n > 100:

    print("Неверно введено натуральное число")

else:

    p = 1

    for i in range(1, n+1):

        p \*= i

    print(f"Факториал {n}! = {p}")

Обратите внимание на запись функции range(). Чтобы она выдавала последовательность целых чисел от 1 до n включительно, необходимо записать n+1, так как последнее значение не включается в диапазон.

Следующий пример – отображение символов \* в виде елочки. Сделать это можно очень просто:

for i in range(1, 7):

    print('\*' \* i)



Здесь счетчик i будет принимать значения 1, 2, 3, 4, 5 и 6. Соответственно, функция print() будет дублировать символ \* i раз. В итоге, в первой строке будет одна звездочка, во второй – две и так до шести звездочек. Видите, как просто можно реализовать такую программу, используя оператор цикла и оператор работы со строками. Именно поэтому, все, что мы проходим, нужно хорошо запоминать и применять по мере необходимости.

Давайте, теперь, соединим все слова списка в одно предложение. Предположим, что у нас имеется список со словами:

words = ["Python", "дай", "мне", "силы", "пройти", "этот", "курс", "до", "конца"]

Которые объединим через цикл for:

s = ''

for w in words:

    s += ' ' + w

print(s)

У нас здесь получается вначале строки пробел. Убрать его можно двумя способами. Либо вызвать метод строки:

s = s.lstrip()

Либо, сделать так, чтобы он и не появлялся:

s = ''

flFirst = True

for w in words:

    s += ('' if flFirst else ' ') + w

    flFirst = False

Но, вот этот второй вариант получается более корявым и медленным по скорости работы. Поэтому, лучше воспользоваться методом lstrip().

А вот внимательный ученик с упреком заявил бы, что все это можно сделать еще проще – с помощью уже знакомого нам метода join():

print(" ".join(words))

И будет абсолютно прав! Те из вас, кто думал также – просто красавчики! Так держать!

Следующая задача. В списке:

digs = [4, 3, 100, -53, -30, 1, 34, -8]

все двузначные числа заменить нулями. Используя те знания, что у нас сейчас есть, это можно сделать так:

for i in range(len(digs)):

    if 10 <= abs(digs[i]) <= 99:

        digs[i] = 0

print(digs)

Обратите внимание, что мы здесь в цикле перебираем индексы элементов списка, а не сами элементы. Благодаря этому, в цикле получаем возможность изменять значение i-го элемента.

Как видите, в этой задаче нам потребовалось в цикле знать и индекс элемента и его значение. Именно для таких случаев в Python существует специальная функция:

индекс, значение = enumerate(объект)

которая возвращает пару (индекс, значение). Используя эту функцию, можно переписать наш пример, следующим образом:

for i, d in enumerate(digs):

    if 10 <= abs(d) <= 99:

        digs[i] = 0

Для меня такой вариант выглядит более читабельным и удобным. Во всем остальном эта программа подобна предыдущему варианту.

Последний пример, который я приведу на этом занятии – преобразование кириллицы в латиницу. Такая задача иногда возникает при создании сайтов. Вначале определим список замен для соответствующих русских букв (по алфавиту от а до я):

t = ['a', 'b', 'v', 'g', 'd', 'e', 'zh',

     'z', 'i', 'y', 'k', 'l', 'm', 'n', 'o', 'p',

     'r', 's', 't', 'u', 'f', 'h', 'c', 'ch', 'sh',

     'shch', '', 'y', '', 'e', 'yu', 'ya'

]

Затем, определим кодовое значение для первой буквы этого списка:

start\_index = ord('а')

Далее, зададим строку и переменную slug, где будем формировать строку на латинице:

title = "Программирование на Python - лучший курс"

slug = ''

Затем, само преобразование в цикле, перебирая каждый символ исходной строки (причем, предварительно, строку мы переводим в нижний регистр – все буквы становятся малыми):

for s in title.lower():

    if 'а' <= s <= 'я':

        slug += t[ord(s) - start\_index]

    elif s == 'ё':

        slug += 'yo'

    elif s in ' !?:,.':

        slug += '-'

    else:

        slug += s

После преобразования следует удалить все подряд идущие символы дефиса:

while slug.count('--'):

    slug = slug.replace('--', '-')

print(slug)

Конечно, это не самая оптимальная реализация данной задачи. Лучше ее делать с использованием, так называемых, регулярных выражений. Но, используя только наши текущие знания и как пример она очень хорошо вписывается в этот урок.

Если все примеры вам были понятны, то смело переходите к следующему материалу!

## 6. Вложенные циклы

Мы с вами в деталях познакомились с работой операторов циклов for и while. На этом занятии сделаем следующий важный шаг и узнаем, как реализуются и работают вложенные циклы.

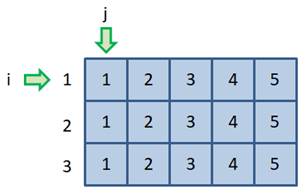
Само это название уже говорит, что один оператор цикла можно вложить в другой. Это могут быть два цикла for или два цикла while или смешанные варианты. Давайте вначале разберемся, как работают эти конструкции. Принцип у всех един, поэтому, для простоты, я возьму два цикла for (один вложен в другой). Эти циклы будут просто пробегать диапазоны чисел, первый от 1 до 3, а второй – от 1 до 5:

for i in range(1, 4):

    for j in range(1, 6):

        print(f"i = {i}, j = {j}", end=' ')

    print()



Во втором вложенном цикле мы будем выводить значения i и j в строку без перехода на новую строчку. А после завершения работы вложенного цикла вызовем функцию print(), как раз, для перевода курсора на новую строку. В результате выполнения этой программы, мы получим таблицу значений переменных i и j.

Почему получились именно такие значения? Вначале у нас счетчик i принимает значение 1, а счетчик j пробегает числа от 1 до 5, в итоге получаем первую строку. После завершения вложенного цикла, срабатывает функция print() и курсор переходит на новую строку. После этого переходим ко второй итерации первого цикла и i = 2. Счетчик j снова проходит значения от 1 до 5 и получаем вторую строку. На следующей итерации первого цикла i = 3, j проходит от 1 до 5 и получаем третью строку. То есть, у нас вложенный цикл for трижды запускался заново и каждый раз j изменялось от 1 до 5. Это и есть принцип работы вложенного цикла – на каждой итерации он отрабатывает снова и снова, пока не завершится первый цикл.

Теперь второй вопрос – зачем все это нужно? Давайте представим, что у нас есть вложенный (двумерный) список:

a = [[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]

(О таких списках мы с вами уже говорили и вам здесь все должно быть понятно). Так вот, если мы будем перебирать его элементы с помощью одного оператора цикла for:

for row in a:

    print(row, type(row))

То переменная row будет ссылаться сначала на первый вложенный список, затем, на второй и потом на третий. Но, так как row ссылается на список, то есть, на итерируемый объект, то нам ничто не мешает перебрать его элементы с помощью второго, вложенного цикла for:

for row in a:

    for x in row:

        print(x, type(x), end=' ')

    print()

Как видите, теперь в консоль выводятся числа типа int, то есть, мы обращаемся непосредственно к элементам этого двумерного списка.

Ну, хорошо, а все-таки, зачем это может быть нужно? Например, так можно выполнить сложение значений из двух одинаковых двумерных списков:

a = [[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]

b = [[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3]]

И сформировать на их основе третий список:

c = []

следующим образом:

for i, row in enumerate(a):

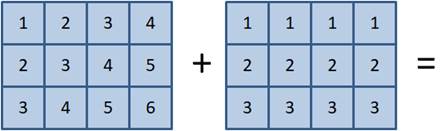
    r = []

    for j, x in enumerate(row):

        r.append(x + b[i][j])

    c.append(r)

print(c)



Я здесь воспользовался еще одной уже знакомой нам функцией enumerate(), которая возвращает индекс и значение текущего элемента. Это удобно для реализации данной программы. Внутри первого цикла мы каждый раз создаем новый пустой список и с помощью метода append() добавляем в его конец новый элемент как сумму значений из списков a и b. Полученную строку (список r) мы, затем, добавляем в основной список c. Так вычисляется сумма значений элементов двух одинаковых списков a и b.

Как видите, для реализации данной программы нам потребовался вложенный оператор цикла for. И это лишь один маленький пример. Другой пример, пусть у нас имеется текст, представленный в виде списка:

t = ["– Скажи-ка,  дядя, ведь не даром",

    "Я Python выучил с   каналом",

    "Специалист что    раздавал?",

    "Ведь были  ж заданья боевые,",

    "Да, говорят,  еще какие!",

    "Недаром помнит    вся Россия",

    "Как мы рубили   их тогда!"

]

Здесь в строках присутствуют два и более пробелов. Наша задача удалить их и оставить только один. Сделаем это с помощью вложенных циклов. В первом цикле for будем перебирать строки – элементы списка, а во втором (вложенном) цикле while удалять лишние пробелы:

for i, line in enumerate(t):

    while line.count('  '):

        line = line.replace('  ', ' ')

    t[i] = line

print(t)

В качестве условия цикла мы здесь вызываем метод count(), который подсчитывает число фрагментов из двух пробелов подряд. Как только их станет 0 – это будет означать False и цикл завершится. Преобразованная строка становится новым i-м элементом списка и в конце результат выводим в консоль.

Следующий пример. Предположим, вначале мы формируем вложенный список размером M x N элементов. Причем, M, N вводим с клавиатуры. Вначале сформируем список, состоящий из всех нулей:

M, N = list(map(int, input("Введите M и N: ").split()))

zeros = []

for i in range(M):

    zeros.append([0]\*N)

print(zeros)

А после этого все элементы заменим на единицы, используя вложенные циклы:

for i in range(M):

    for j in range(N):

        zeros[i][j] = 1

Как видите, в целом, все достаточно просто.

И последний пример. Пусть у нас имеется квадратный список (размерности совпадают):

A = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]

Необходимо поменять строки на столбцы и получить новое представление этого же списка. Для этого достаточно поменять местами элементы, стоящие выше главной диагонали с элементами, стоящими ниже главной диагонали. То есть, у нас счетчик i будет меняться от 0 до 3, а счетчик j от i+1 до 3. Затем, соответствующие элементы будем менять между собой:

for i in range(len(A)):

    for j in range(i+1, len(A)):

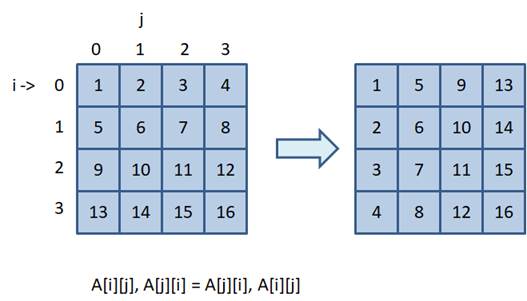
        A[i][j], A[j][i] = A[j][i], A[i][j]

for r in A:

    for x in r:

        print(x, end='\t')

    print()



Как видите, у нас получилось нужно преобразование. В математике это называется транспонированием матрицы.

Конечно, уровень вложенности операторов циклов может быть еще больше и два и три и четыре. Однако, на практике слишком большого вложения следует избегать, так как программа становится менее читаемой и гибкой для дальнейшего изменения. Лучше ограничиваться тремя подряд идущими во вложении операторами циклов, не более.

Для закрепления, как всегда, пройдите практические задания и, затем, переходите к следующему уроку.

## Практика 7 вложенные циклы

1. Напишите программу, которая ищет среди целых чисел, принадлежащих числовому отрезку [2422000; 2422080], простые числа. Выведите все найденные простые числа в порядке возрастания, слева от каждого числа выведите его номер по порядку, считая, что первое найденное число имеет номер «1», второе  — «2», и т. д.

2. Задание "Совершенное число"

Число совершенно, если оно равно сумме всех своих делителей, кроме самого себя. Пример: 6 = 1 + 2 + 3. Проверить, является ли данное число совершенным.

Дано целое положительно число.

Формат выходных данных

Вывести "Да", если число совершенно и "Нет" в противоположном случае.

Дано Результат

4 Нет

6 Да

14 Нет

22 Нет

28 Да

100 Нет

496 Да

for n in range(1, 10000):  
 s = 0  
 for i in range(1, n // 2 + 1):  
 if n % i == 0:  
 s += i  
 if n == s:  
 print(n, 'ДА')

2. В строке содержится последовательность целых чисел. Определите и запишите в ответе сначала количество пар элементов последовательности, в которых хотя бы одно число делится на 3, затем максимальную из сумм элементов таких пар. В данной задаче под парой подразумевается два идущих подряд элемента последовательности. Например, для последовательности из пяти элементов: 6; 2; 9; –3; 6  – ответ: 4 11.

Входные данные:

s = '-67 -694 835 786 -457 495 -699 -592 -405 -745 656 501 48 856 417 -600 521 221 -548 -421 -139 -293 -313 662 995 -855'

Выходные:

14 1621

s = **'-67 -694 835 786 -457 495 -699 -592 -405 -745 656 501 48 856 417 -600 521 221 -548 -421 -139 -293 -313 662 995 -855'**s = list(map(int, s.split()))  
sumMax = 0  
countPairs = 0  
**for** i **in** range(len(s) - 1):  
 **if** s[i] % 3 == 0 **or** s[i + 1] % 3 == 0:  
 sumMax = max(sumMax, s[i] + s[i + 1])  
 countPairs+=1  
  
print(countPairs, sumMax)

3. Решить задачу 2 со следующим условием: данной задаче под парой подразумевается два различных элемента последовательности.

Выходные:

189 1781

**3\*.** Вводится натуральное число N (то есть, положительное, целое). Требуется создать двумерный (вложенный) список размером N x N элементов, состоящий из всех единиц, а затем, в последний столбец записать пятерки. Вывести этот список на экран в виде таблицы чисел, как показано в примере ниже.

P.S. Будьте внимательны в конце строк пробелов быть не должно!

**Sample Input:**

4

**Sample Output:**

1 1 1 5

1 1 1 5

1 1 1 5

1 1 1 5

Вариант 1

N = int(input())

for i in range(N):

print('1 ' \* (N - 1) + '5')

n = int(input())

lst = [[1] \* n] \* n

lst[0][n-1] = 5

for i in lst:

print(\*i)

Вариант 2

N = int(input())

M = [[1]\*(N-1) + [5] for \_ in range(N)]

[print (\*i) for i in M]

n = int(input())

a = []

for i in range(n):

a.append([1] \* n)

for i in range(n):

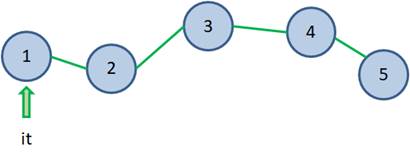
a[i][-1] = 5

for row in a:

print(\*row)

# Итератор и итерируемые объекты. Функции iter() и next()

На этом занятии мы узнаем, что такое итерируемые объекты и познакомимся со способами их перебора. Мы уже знаем, что в Python существуют объекты, содержащие последовательность некоторых элементов. Например, строки и списки. Так вот, существует универсальный механизм для перебора элементов этих и других подобных им объектов. Реализуется он через специальную конструкцию под названием итератор. То есть, каждый итерируемый объект предоставляет доступ к своим элементам через итератор. С помощью этого итератора можно один раз пройтись по всем элементам коллекции от начала до конца.



Чтобы получить доступ к итератору объекта, например, списка, нужно вызвать специальную функцию iter() и первым аргументом указать итерируемый объект. Например, возьмем список:

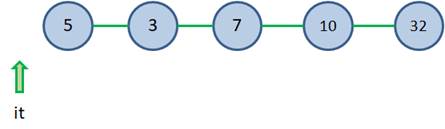
d = [5, 3, 7, 10, 32]

и для него вызовем функцию iter():

iter(d)

Смотрите, нам был возвращен объект-итератор для списка. Сохраним его через переменную it:

it = iter(d)



Все, теперь у нас есть итератор для однократного перебора элементов списка d. Далее, чтобы перебрать значения итерируемого объекта, используется функция next(), которой следует передать объект-итератор:

next(it)

Она осуществляет переход к следующему элементу (при первом вызове переходим к первому элементу) и возвращает значение этого элемента. Видим значение 5. Вызовем эту функцию еще раз:

next(it)

Итератор был перемещен на следующий элемент и возвратилось значение 3. И так можно пройтись до конца нашего списка:

next(it)

next(it)

next(it)

Если для последнего элемента вызвать функцию еще раз:

next(it)

то получим ошибку StopIteration.

Когда итератор дошел до конца коллекции, его уже нельзя вернуть назад и пройти все элементы заново. Для этого придется создавать новый итератор с помощью функции iter():

it = iter(d)

и заново перебирать все элементы:

next(it)

Так работает этот механизм перебора элементов итерируемого объекта. Причем, он универсален и не зависит от типа объекта: это может быть и список и строка и любой другой перебираемый объект. Давайте я покажу, как это будет работать со строкой:

s = "python"

Также создаем итератор:

it\_s = iter(s)

и перебираем символы функцией next():

next(it\_s)

next(it\_s)

Как видите, итератору все равно что перебирать, главное, чтобы сам объект поддерживал этот механизм, то есть, был итерируемым. Также следует иметь в виду, что доступ к элементам через итератор и по индексу:

s[2]

это совершенно разные способы обращения к элементам. К тому же не у всех итерируемых объектов есть возможность указания индексов.

Помимо строк и списков к итерируемым объектам также относится ранее рассмотренная функция:

r = range(5)

Для нее мы можем получить итератор:

it = iter(r)

и перебирать значения функцией next():

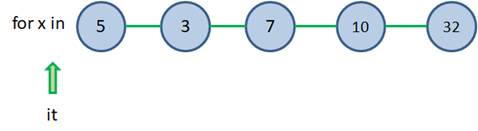
next(it)

next(it)

Когда я вам все это рассказываю, у вас, наверное, постоянно крутится вопрос: а зачем это надо? Мы и без всяких итераторов можем обращаться к элементам строк, списков и даже объекта range()? Да, все верно, если нам нужно извлечь какое-либо значение, скажем, из списка, то следует использовать индексы или срезы. То же самое и со строками. Но если нам в программе нужно перебирать итерируемые объекты самых разных типов, то единственный универсальный и безопасный способ это сделать – использовать итераторы. Например, так происходит в операторе цикла for. Мы можем ему указать перебрать любой итерируемый объект и он должен «уметь» это делать вне зависимости от типа этого объекта. Поэтому он обращается к итератору и перебирает элементы через этот универсальный механизм, пока не возникнет исключение StopIteration. Именно поэтому мы в цикле можем с легкостью перебирать и списки:

for x in [5, 3, 7, 10, 32]:

    print(x)



И строки:

for x in "python":

    print(x)

И объект range():

for x in range(1, 6):

    print(x)

И вообще любой другой тип объектов, которые поддерживают итератор, то есть, являются итерируемыми.

А что будет, если мы попробуем получить итератор для не итерированного объекта, например, числа:

iter(5)

В этом случае получим ошибку, что объект типа int не является итерируемым, то есть, не поддерживает итератор и перебор элементов. Действительно, число – это одно значение и перебирать тут нечего, поэтому оно относится к неитерируемым объектам. Соответственно, и в цикле for мы не можем указывать такие элементы:

for x in 5:

    print(x)

Перебрать одно число не получится. Здесь можно указывать только итерируемые объекты.

Надеюсь, из этого занятия вам стало понятно, что такое итератор и итерируемые объекты, как получать итератор с помощью функции iter() и как перебирать элементы функцией next(). Закрепите этот материал практическими заданиями, после чего жду всех вас на следующем уроке.

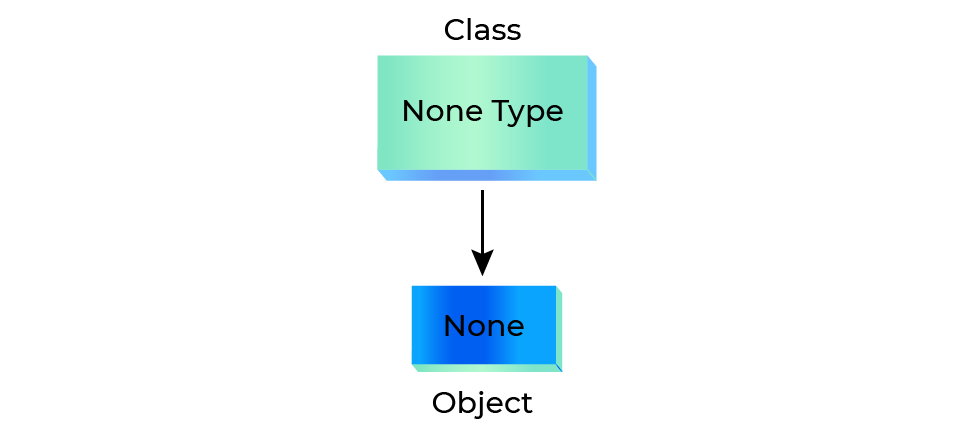
# Тип данных NoneType

Во многих языках программирования (Java, C++, C#, JavaScript и т.д.) существует ключевое слово null, которое можно присвоить переменным. Концепция ключевого слова null заключается в том, что оно дает переменной нейтральное или "нулевое" поведение.

В языке Python, слово null заменено на None, поскольку слово null звучит не очень дружелюбно, а None относится именно к требуемой функциональности – это ничего и не имеет поведения.

Литерал None

Литерал None в Python позволяет представить null переменную, то есть переменную, которая не содержит какого-либо значения. Другими словами, None – это специальная константа, означающая пустоту. Если более точно, то None – это объект специального типа данных NoneType.



Следующий программный код:

var = None

print(type(var))

выведет:

<class 'NoneType'>

Мы можем присвоить значение None любой переменной, однако мы не можем самостоятельно создать другой NoneType объект.

Все переменные, которым присвоено значение None, ссылаются на один и тот же объект типа NoneType. Создание собственных экземпляров типа NoneType недопустимо. Объекты, существующие в единственном экземпляре, называются синглтонами.

Проверка на None

Для того, чтобы проверить значение переменной на None, мы используем либо оператор is, либо оператор проверки на равенство ==.

Следующий программный код:

var = None

if var is None: # используем оператор is

  print('None')

else:

  print('Not None')

выведет:

None

Следующий программный код:

var = None

if var == None: # используем оператор ==

  print('None')

else:

  print('Not None')

выведет:

None

Для сравнения переменной с None всегда используйте оператор is. Для встроенных типов поведение is и == абсолютно одинаково, однако с пользовательскими типами могут возникнуть проблемы, так как в Python есть возможность переопределения операторов сравнения в пользовательских типах.

Сравнение None с другими типами данных

Сравнение None с любым объектом, отличным от None, дает значение False.

Следующий программный код:

print(None == None)

выведет:

True

Следующий программный код:

print(None == 17)

print(None == 3.14)

print(None == True)

print(None == [1, 2, 3])

print(None == 'Beegeek')

выведет:

False

False

False

False

False

Важно понимать, что следующий программный код:

print(None == 0)

print(None == False)

print(None == '')

выведет:

False

False

False

   Значение None не отождествляется с значениями 0, False, ''.

Сравнивать None с другими типами данных можно только на равенство.

Следующий программный код:

print(None > 0)

print(None <= False)

приводит к ошибке:

TypeError: '>' not supported between instances of 'NoneType' and 'int' ('bool')

Примечания

Примечание 1. Обратите внимание, что функции, не возвращающие значений, на самом деле, в Python возвращают значение None.

def print\_message() :

print('Я - Тимур,')

print('король матана. ')

Мы можем вызвать функцию print\_message() так:

print\_message()

 или так:

res = print\_message()

В переменной res хранится значение None.