Информационные технологии

Ввод-вывод списков

Превращение строки в список

Возьмём строку, состоящую из слов, разделённых пробелами (может быть не одним) и символами конца строки.

In [22]:

```
s = "hello world    this is    a\ntest"
print(s)
```

```
hello world    this is    a
test
```

Если мы хотим работать с отдельными словами, входящими в эту строку, то её нужно разделить на отдельные слова — то есть получить список, состоящий из слов, которые входят в эту строку. Для этого можно использовать метод `split()`.

In [23]:

```
s.split()
```

Out[23]:

```
['hello', 'world', 'this', 'is', 'a', 'test']
```

Итак, мы получили ровно то, что хотели. Заметим, что в качестве разделителя в этом случае использовались пробелы (один или несколько), а также любые *пробельные символы*, в число которых входит символ табуляции (у нас его не было) и символ перевода строки `\n`. Разделители в элементы получающегося списка не попадают.

Важно! Метод `split()` не меняет строку (вообще говоря строку в принципе нельзя поменять):

In [24]:

```
s = "hello world    this is    a\ntest"
s.split()
print(s)
```

```
hello world    this is    a
test
```

В коде выше вторая строка не приводит ни к какому эффекту — вы создали список и мгновенно его забыли, а строка осталась неизменной. Если вы хотите что-то дальше делать со списком, полученным в результате применения `split()`, а не просто на него полюбоваться и забыть навсегда, то нужно, например, сохранить его в виде какой-то переменной:

In [25]:

```
words = s.split()
# теперь результат выполнения s.split() сохранён в переменную words
for word in words:
    print(word)
```

```
hello
world
this
is
a
test
```

У метода `split()` есть необязательный параметр — разделитель. Если он указан, то строчка разбивается по тому разделителю, который ему передан.

In [26]:

```
s.split("    ")
```

Out[26]:

```
['hello world', 'this is', 'a\ntest']
```

Мы передали в качестве разделителя четыре пробела и теперь строка разделяется по ним, а один пробел или символ перевода строки не воспринимается как разделитель.

Допустим, мы хотим ввести несколько чисел через запятую и вывести каждое из них, увеличенное на 1. Можно попробовать такой код:

In [27]:

```
s = input("Введите несколько чисел, разделённых запятыми: ")
elements = s.split(",")
for n in elements:
    print(n+1)
```

```
Введите несколько чисел, разделённых запятыми: 1, 2, 3
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-27-d86c54cf4e17> in <module>()
      2 elements = s.split(",")
      3 for n in elements:
----> 4     print(n+1)

TypeError: must be str, not int
```

Как нетрудно догадаться, он не работает: после разделения строки получается список, состоящий из строк. Чтобы превратить эти строки в числа, нужно использовать `int` . Вообще говоря, есть методы, которые позволяют довольно просто превратить все элементы списка в числа, но пока мы до них не дошли, будем обрабатывать в цикле каждый элемент в отдельности.

In [28]:

```
s = input("Введите несколько чисел, разделённых запятыми: ")
elements = s.split(",")
for n in elements:
    n = int(n)
    print(n+1)
```

```
Введите несколько чисел, разделённых запятыми: 1, 2, 3
2
3
4
```

Теперь работает!

Вывод списка в строчку

Решим теперь обратную задачу: есть список, мы хотим его вывести в каком-то виде. Есть разные способы это сделать. Например, можно просто распечатать:

In [35]:

```
elements = ["one", "two", "three"]
print(elements)
```

```
['one', 'two', 'three']
```

Тут элементы списка заключатся в скобки, а каждая строчка дополнительно ещё и в кавычки. Можно распечатать поэлементно в цикле, как в примере выше:

In [36]:

```
for el in elements:
    print(el)
```

```
one
two
three
```

Так он выводится в столбик, потому что `print()` по умолчанию добавляет символ перевода строки `\n` в конце каждой выводимой строчки. Если мы хотим вывести элементы списка как-то иначе, например, в строчку, но без скобок, запятых и кавычек, то нужно использовать другие способы.

Первым из них является применение метода `join()`.

In [37]:

```
print(" ".join(elements))
```

```
one two three
```

In [38]:

```
print(":".join(elements))
```

```
one:two:three
```

In [39]:

```
print("----".join(elements))
```

```
one----two----three
```

Это удобный метод, но у него есть ограничение: он требует, чтобы список, который ему подали на вход, состоял из строк. Если среди его элементов есть другие объекты, он ломается.

In [40]:

```
numbers = [6, 9, 10]
", ".join(numbers)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-40-639bcad789c7> in <module>()
      1 numbers = [6, 9, 10]
----> 2 ", ".join(numbers)

TypeError: sequence item 0: expected str instance, int found
```

Эту проблему можно обойти (мы позже будем говорить про `map()` и списочные включения (list comprehensions)), но сейчас обсудим другой подход к выводу списков, основнный на функции `print()`.

Напомним, что если передать функции `print` список, то он его распечатает в квадратных скобках через запятую.

In [41]:

```
print(numbers)
```

```
[6, 9, 10]
```

А если передать отдельные элементы списка, то он их распечатает без скобок и разделяя пробелом (вообще говоря, любым символом — это настраивается с помощью параметра `sep`).

In [42]:

```
print(numbers[0], numbers[1], numbers[2])
```

```
6 9 10
```

Строчка выше даёт тот результат, который мы хотели, но она будет работать только в том случае, когда в списке ровно три элемента: если элементов больше, выведены будут только три, если меньше, то возникнет ошибка. К счастью, в Python есть конструкция, которая позволяет «раздеть» список — передать все его элементы какой-то функции через запятую. Это делается с помощью звёздочки (*).

In [43]:

```
print(*numbers)
```

```
6 9 10
```

Звёздочка как бы убирает квадратные скобки вокруг элементов списка. Сравните:

In [44]:

```
print(*[5, 8, 9, 11, "Hello"])
```

```
5 8 9 11 Hello
```

In [45]:

```
print(5, 8, 9, 11, "Hello")
```

```
5 8 9 11 Hello
```

Если вы хотите использовать не пробел, а какой-нибудь разделитель, то это тоже возможно:

In [46]:

```
print(*numbers, sep = ', ')
```

```
6, 9, 10
```

Строки и срезы

Строки могут вести себя почти как списки. Например, можно обращаться к отдельным элементам строки (к отдельным символам) или делать срезы.

In [47]:

```
s = "This is a test"
```

In [48]:

```
s[6]
```

Out[48]:

```
's'
```

In [49]:

```
s[5]
```

Out[49]:

```
'i'
```

In [50]:

```
s[3:8]
```

Out[50]:

```
's is '
```

Позже мы ещё поговорим про строки (это отдельная большая история).

Алгоритмы с циклами

Рассмотрим пример алгоритма, в котором используется цикл.

Напомним, как в прошлый раз мы искали числа Фибоначчи.

Сначала мы выполняем *инициализацию* — устанавливаем начальные значения переменных, которые нам понадобятся в дальнейшем. Исходно мы знаем значения первых двух чисел Фибоначчи (это единицы); мы запишем их в переменные a и b и в дальнейшем будем хранить очередные два найденных числа Фибоначчи, необходимые для нахождения следующего числа.

In [51]:

```
# инициализация (выполняется только один раз в начале алгоритма)
a = 1 # первое число
b = 1 # второе число
```

Затем мы пишем код, осуществляющий переход к следующему числу. Мы будем выполнять его несколько раз, каждый раз получая новое число Фибоначчи.

In [52]:

```
c = a + b #нашли следующее число
a = b # значение a нам уже не нужно, а вот значение b ещё пригодится
b = c # запомнили вычисленное значение
print(b)
```

```
2
```

Выполнив эту ячейку несколько раз, мы будем получать последовательные числа Фибоначчи.

Этот подход работает довольно неплохо, но если бы нам нужно было найти 115-е число Фибоначчи, мы бы замучались перезапускать ячейку. Вместо этого используем цикл `for`, который будет автоматически выполнять фрагмент кода, вычисляющий следующее число, столько раз, сколько нам нужно. Например, 10:

In [53]:

```
# инициализация (выполняется только один раз в начале алгоритма)
a = 1 # первое число
b = 1 # второе число
for i in range(10):
    # тело цикла: выполнится 10 раз
    c = a + b
    a = b
    b = c
    print(i, c)
```

```
0 2
1 3
2 5
3 8
4 13
5 21
6 34
7 55
8 89
9 144
```

Другой пример: допустим, мы бы хотели не выводить найденные числа Фибоначчи на экран, а записать их в некоторый список. Для этого немного модифицируем код выше: вместо команды `print()` нужно подставить команду, которая добавит найденное число в некоторый список. Однако, чтобы было, куда добавлять, этот список должен быть создан заранее. Изначально он может быть и пустым — такой список обозначается пустыми квадратными скобками.

In [54]:

```
a = 1 # первое число
b = 1 # второе число
fib = []
# создали пустой список fib

for i in range(25):
    c = a + b
    a = b
    b = c
    fib.append(b) # записали элемент в конец списка fib

print(fib)
```

```
[2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 418
1, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418]
```

Проверка условий

В ходе выполнения программы иногда требуется в зависимости от каких-то условий выполнять тот или иной фрагмент кода. Например, если пользователь ввёл не те данные, которые от него просили (хотели положительное число, а получили отрицательное), то надо вывести ошибку и попросить ввести данные снова. Решение этой задачи разбивается на несколько шагов: сначала нужно проверить некоторое условие, а потом в зависимости от результата этой проверки выбрать, какой код выполнять. Давайте начнём с проверки условий.

In [55]:

```
6 < 8
```

Out[55]:

```
True
```

Здесь мы спросили «Правда ли, что 6 меньше 8?». «Воистину так» — ответил Python на своём заморском языке. Слово `True`, которое он выдал — это не просто слово, означающее «истина», а специальное логическое значение. Его ещё называют «булевским» (по имени одного из основателей математической логики Джоджа Буля (https://ru.wikipedia.org/wiki/%D0%91%D1%83%D0%BB%D1%8C,_%D0%94%D0%B6%D0%BE%D1%80%D
Оно бывает всего двух видов: либо истина (`True`), либо ложь (`False`). Третьего не дано.

In [56]:

```
8 > 9
```

Out[56]:

```
False
```

Результат проверки можно записать в переменную.

In [57]:

```
condition = 6 < 8
```

In [58]:

```
condition
```

Out[58]:

```
True
```

Говорят, что переменная `condition` теперь булевская (`bool`).

In [59]:

```
type(condition)
```

Out[59]:

```
bool
```

Можно проверять равенство двух величин. Правда ли, что 7 равно 7?

In [60]:

```
7 == 7
```

Out[60]:

```
True
```

Обратите внимание: здесь нужно написать символ равенства два раза, потому что один знак равно — это операция присвоения («присвоить то, что справа, тому, что слева»), а операция проверки равенства — это совсем другая штука. Например.

In [61]:

```
a = 5
```

Положили в `a` число `5`. Такая операция ничего не вернула.

In [62]:

```
a = 7
```

Теперь положили в `a` число 7.

In [63]:

```
a == 5
```

Out[63]:

```
False
```

Теперь спросили, правда ли, что `a` равняется пяти. Получили `False`.

Надо сказать, что сравнение работает достаточно разумным образом. Например, число `7` и число `7.0` — это, строго говоря, разные объекты (первое — это целое число, второе — число с плавающей запятой), но понятно, что как числа это один и тот же объект. Поэтому сравнение выдаст `True`.

In [64]:

```
7 == 7.0
```

Out[64]:

```
True
```

Оператор if

In [66]:

```
a = int(input("Введите положительное число: "))
if a < 0:
    print("Ошибка!")
    print("Число не является положительным!")
print("Вы ввели", a)
```

```
Введите положительное число: 4
Вы ввели 4
```

Нужно обратить внимание на несколько вещей: во-первых, после `if` указывается условие, а после условия обязательно ставится двоеточие (как и в циклах), дальше идёт блок команд, которые выполняются в том случае, если условие верно (то есть является `True`). Как и в циклах, этот блок команд должен быть выделен отступом. Команды, не входящие в блок (в данном случае это последняя строчка) выполняются в любом случае.

Допустим, мы хотим обработать отдельно обе ситуации: когда условие выполняется и когда оно не выполняется. Для этого нужно использовать ключевое слово `else`.

In [ ]:

```
a = int(input("Введите положительное число: "))
if a < 0:
    print("Ошибка!")
    print("Число не является положительным!")
else:
    print("Как хорошо!")
    print("Вы ввели положительное число!")
print("Вы ввели", a)
```

Конструкция `if-else` работает как альтернатива: выполняется либо один фрагмент кода (после `if` — если условие верно), либо другой (после `else` — если неверно). Иногда нужно проверить несколько условий подряд.

In [ ]:

```
a = int(input("Введите какое-нибудь число: "))
if a > 100:
    print("Это очень большое число")
elif a > 10:
    print("Это больше число")
else:
    print("Это маленькое число")
```

Здесь используется ключевое слово `elif`, являющееся объединением слов `else` и `if`. Условия проверяются по очереди, начиная от первого; как только какое-то из условий оказывается верным, выполняется соответствующий блок и проверка остальных условий не производится.

Сложные условия

Допустим, нам нужно проверить выполнение нескольких условий. Скажем, мы хотим получить число от 0 до 100 — числа меньше 0 или больше 100 нас не устраивают. Это можно было бы сделать с помощью нескольких вложенных операторов `if` примерно так.

In [ ]:

```
a = int(input("Пожалуйста, введите число от 0 до 100: "))
if a <= 100:
    if a >= 0:
        print("Спасибо, мне нравится ваше число")
    else:
        print("Вы ошиблись, это не число от 0 до 100")
else:
    print("Вы ошиблись, это не число от 0 до 100")
```

Этот код довольно громоздок, строчку с сообщением об ошибке пришлось скопировать дважды. Не очень хорошо. Оказывается, можно реализовать тот же функционал проще.

In [ ]:

```
a = int(input("Пожалуйста, введите число от 0 до 100: "))
if a <= 100 and a >= 0:
    print("Спасибо, мне нравится ваше число")
else:
    print("Вы ошиблись, это не число от 0 до 100")
```

Здесь используется ключевое слово `and`, обозначающее операцию *логического И*. Оно делает следующее: проверяет левое условие (в данном случае `a <= 100`), проверяет правое условие (`a >= 100`) и если оба этих условия выполняются (то есть имеют значение `True`), то и результат выполнения `and` оказывается `True`; если же хотя бы одно из них не выполняется (то есть имеет значение `False`), то и результат выполнения `and` является `False`. Таким образом мы можем проверить в точности интересующее нас условие.

> Строго говоря, если левый аргумент `and` оказывается ложью, то правый даже не вычисляется: зачем тратить время, если уже понятно, что возвращать надо ложь?

Можно было бы переписать этот код другим способом, используя логическое ИЛИ (`or`):

In [ ]:

```
a = int(input("Пожалуйста, введите число от 0 до 100: "))
if a > 100 or a < 0:
    print("Вы ошиблись, это не число от 0 до 100")
else:
    print("Спасибо, мне нравится ваше число")
```

Результат выполнения `or` является истиной в том случае, если хотя бы один аргумент является истиной. Наконец, есть третий логический оператор — это отрицание (`not`). Он имеет всего один аргумент и возвращает истину, если этот аргумент является ложью, и наоборот.

In [ ]:

```
a = int(input("Пожалуйста, введите число от 0 до 100: "))
if not (a <= 100 and a >= 0):
    print("Вы ошиблись, это не число от 0 до 100")
else:
    print("Спасибо, мне нравится ваше число")
```

Можно проверить, как работают логические команды, просто подставляя в качестве аргументов `True` или `False` :

In [67]:

```
True or False
```

Out[67]:

```
True
```

In [68]:

```
False and True
```

Out[68]:

```
False
```

Быть или не быть?

In [69]:

```
to_be = False
to_be or not to_be
```

Out[69]:

```
True
```

Что будет, если `to_be` сделать равным `True` ?

Цикл while

In [ ]:

```
a = int(input("Введите число от 0 до 100: "))
while a>100 or a<0:
    print("Неверно! Это не число от 0 до 100")
    a = int(input("Введите число от 0 до 100: "))
print("Ну хорошо")
```

Другой пример: проверка пароля.

In [ ]:

```
correct_passwd = ';ugliugliug'
passwd = input("Please, enter password: ")
while passwd != correct_passwd:
    print("Access denied")
    passwd = input("Please, enter password: ")
print("Access granted")
```

Этот код не очень изящный, потому что нам приходится дважды писать строку с `input()`. Ситуация, при которой нам приходится копировать какие-то строчки кода, обычно означает, что допущена ошибка в проектировании. Можно сделать это более изящно с помощью команды `break` — она позволяет выйти из цикла. Следующий пример также демонстрирует бесконечный цикл: по идее `while True:` должен выполняться до тех пор, пока `True` это `True`, то есть вечно. Но мы выйдем раньше с помощью `break`.

In [ ]:

```
correct_passwd = ';ugliugliug'
while True:
    passwd = input("Please, enter password: ")
    if passwd == correct_passwd:
        print("Access granted")
        break
    else:
        print("Access denied")
```

Команду `break` можно использовать для выхода из любого цикла. Вот пример для цикла `for`:

In [ ]:

```
numbers = [6, 8, 9, 6, -7, 9]
for i in numbers:
    if i<0:
        print("Negative number detected!!!!111111odin")
        break
    print(i+1)
```

Нумерация элементов списка

В качестве небольшого отступления обсудим такую задачу: есть список, нужно вывести его элементы и их индексы. Эту задачу можно было бы решать так:

In [ ]:

```
numbers = [7, 8, 9, 43]
i = 0 # здесь будет храниться индекс
for n in numbers:
    print(i, n)
    i += 1
    # эта строчка эквивалентна i = i + 1
```

Не самое изящное решение — приходится вводить какую-то переменную `i`, инициализировать её до входа в цикл и ещё не забывать прибавить к ней единицу внутри цикла.

Другой вариант:

In [ ]:

```
numbers = [7, 8, 9, 43]
for i in range(len(numbers)):
    print (i, numbers[i])
```

Тоже не является верхом изящества: здесь нам приходится каждый раз писать `numbers[i]` и вообще понятность страдает: глядя на цикл `for` неясно, что мы собираемся перебирать элементы списка `numbers`.

Правильный Python'овский подход выглядит вот так:

In [ ]:

```
numbers = [7, 8, 9, 43]
for i, n in enumerate(numbers, 2):
    print(i,n)
```

Что здесь произошло. Главная магия кроется в команде `enumerate()`. Давайте посмотрим, как она работает:

In [ ]:

```
enum = list(enumerate(numbers))
```

In [ ]:

```
enum
```

В результате выполнения `enumerate` возвращается штука, которая ведёт себя как список, элементами которого являются пары чисел (вообще-то это кортежи, `tuple`, то есть неизменяемые списки). В каждой паре первое число — это индекс, а второе — элемент исходного списка.

Дальше, при выполнении цикла `for` используется механизм списочного присваивания. Работает он так.

In [ ]:

```
a, b = (7, 9)
```

In [ ]:

```
print(a)
print(b)
```

Если в левой части от знака равенства стоит несколько переменных, разделённых запятыми, а в правой — какой-нибудь объект, ведущий себя как список, причём число его элементов равно число переменных, то присваивание идёт поэлементно: первый элемент списка в первую переменную, второй во вторую и т.д.

In [ ]:

```
a, b, c = (1, 2, 3)
print(c)
```

Теперь обратите внимание, что в цикле `for` у нас в качестве переменной цикла указаны две переменные (через запятую). Что произойдёт, когда мы попадём в цикл в первый раз? `for` возьмёт первый элемент от `enumerate(numbers)`. Это пара чисел:

In [ ]:

```
enum[0]
```

Теперь он приравняет эту пару чисел к паре переменных: `i` и `n`:

In [ ]:

```
i, n = enum[0]
```

Теперь в `i` лежит `0` (индекс первого элемента `numbers`, а в `n` — сам первый элемент `numbers`. И так далее, на каждом шаге в `i` будет лежать соответствующий индекс, а в `n` соответствующее число.

При этом результат получается гораздо более изящным, чем предыдущие: нет необходимости как-то явно описывать происходящее с `i`, и смысл происходящего понятен при взгляде на строчку с `for`.

На сегодня всё.