

6.04.2020 | МИРЭК | 4 модуль

Автор: Татьяна Рогович

Основы программирования в Python

Лекция 1. Тема 1

Python Refresher: основные типы данных (int, float, bool)

Функция print()

С помощью Python можно решать огромное количество задач. Мы начнем с очень простых и постепенно будем их усложнять, закончив наш курс небольшим проектом. Если вы уже сталкивались с программированием, то вы помните, что обычно самой первой программой становится вывод "Hello, world". Попробуем сделать это в Python.

```
In [1]: print('Hello, world!')  
       print(1)  
  
Hello, world!  
1
```

Обратите внимание, что "Hello, world!" мы написали в кавычках, а единицу - нет. Это связано с тем, что в программировании мы имеем дело с разными типами данных. И Python будет воспринимать текст как текст (строковую переменную), только в том случае, если мы его возьмем в кавычки (неважно, одинарные или двойные). А при выводе эти кавычки отображаться уже не будут (они служат знаком для Python, что внутри них - текст).

print() - это функция, которая выводит то, что мы ей передадим. В других IDE это был бы вывод в терминал, а в Jupyter вывод напечатается под запускаемой ячейкой. Распознать функцию в питоне можно по скобкам после слова, внутри которых мы передаем аргумент, к которому эту функцию нужно применить (текст "Hello, world" или 1 в нашем случае).

```
In [2]: print(Hello, world)
```

```
-----
NameError Traceback (most recent call last)
<ipython-input-2-65c2bc1cadaf> in <module>()
----> 1 print(Hello, world)

NameError: name 'Hello' is not defined
```

Написали без кавычек - получили ошибку. Кстати, обратите внимание, что очень часто в тексте ошибки есть указание на то, что именно произошло, и можно попробовать догадаться, что же нужно исправить. Текст без кавычек Python воспринимает как название переменной, которую еще не задали. Кстати, если забыть закрыть или открыть кавычку (или поставить разные), то тоже поймаем ошибку.

Иногда мы хотим комментировать наш код, чтобы я-будущий или наши коллеги поменьше задавались вопросами, а что вообще имелось ввиду. Комментарии можно писать прямо в коде, они не влияют на работу программы, если их правильно оформить.

```
In [5]: # Обратите внимание: так выглядит комментарий – часть скрипта, которая не будет исполнена
# при запуске программы.
# Каждую строку комментария мы начинаем со знака хэштега.
```

```
'''
Это тоже комментарий – обычно выделение тремя апострофами мы используем для тех случаев,
когда хотим написать длинный, развернутый текст.
'''
```

```
print('Hello, world')
```

```
Hello, world
```

Обратите внимание, что в отличие от других IDE (например, PyCharm) в Jupyter Notebook не всегда обязательно использовать print(), чтобы что-то вывести. Но не относитесь к этому как к стандарту для любых IDE.

```
In [1]: "Hello, world"
```

```
Out[1]: 'Hello, world'
```

Выше рядом с выводом появилась подпись Out[]. В данном случае Jupyter показывает нам последнее значение, лежащее в буфере ячейки. Например, в PyCharm такой вывод всегда будет скрыт, пока мы его не "напечатаем" с помощью print(). Но эта особенность Jupyter помогает нам быстро проверить, что, например, находится внутри переменной, когда мы отлаживаем наш код.

Следующая вещь, которую нужно знать про язык программирования - как в нем задаются переменные. Переменные - это контейнеры, которые хранят в себе информацию (текстовую, числовую, какие-то более сложные виды данных). В Python знаком присвоения является знак =.

```
In [1]: x = 'Hello, world!'
print(x)      # Обратите внимание, что результат вызова этой функции
              # такой же, как выше,
              # только текст теперь хранится внутри переменной.
```

Hello, world!

Python - язык чувствительный к регистру. Поэтому, когда создаете/вызываете переменные или функции, обязательно используйте правильный регистр. Так, следующая строка выдаст ошибку.

```
In [2]: print(X) # мы создали переменную x, а X не существует
```

```
-----
NameError: name 'X' is not defined
-----
```

Traceback (most recent call last)
<ipython-input-2-f9979d7199c5> in <module>()
----> 1 print(X) # мы создали переменную x, а X не существует

NameError: name 'X' is not defined

Еще раз обратим внимание на ошибку. *NameError: name 'X' is not defined* означает, что переменная с таким названием не была создана в этой программе. Кстати, обратите внимание, что переменные в Jupyter хранятся на протяжении всей сессии (пока вы работаете с блокнотом и не закрыли его), и могут быть созданы в одной ячейке, а вызваны в другой. Давайте опять попробуем обратиться к x.

```
In [11]: print(x) # работает!
```

Hello, world!

Типы данных: целочисленные переменные (integer)

Знакомство с типа данных мы начнем с целых чисел. Если вы вдруг знакомы с другими языками программирования, то стоит отметить, что типизация в Python - динамическая. Это значит, что вам не нужно говорить какой тип данных вы хотите положить в переменную - Python сам все определит. Проверить тип данных можно с помощью функции type(), передав ей в качестве аргумента сами данные или переменную.

ЦЕЛЫЕ ЧИСЛА (INT, INTEGER): 1, 2, 592, 1030523235 - любое целое число без дробной части.

```
In [3]: y = 2
print(type(2))
print(type(y))
```

```
<class 'int'>
<class 'int'>
```

Обратите внимание - выше мы вызвали функцию внутри функции. type(2) возвращает скрытое значение типа переменной (int для integer). Чтобы вывести это скрытое значение - мы должны его "напечатать".

Самое элементарное, что можно делать в Python - использовать его как калькулятор. Давайте посмотрим, как он вычитает, складывает и умножает.

```
In [14]: print(2 + 2)
print(18 - 9)
print(4 * 3)
```

```
4
9
12
```

С делением нужно быть немного осторожней. Существует два типа деления - привычное нам, которое даст в ответе дробь при делении 5 на 2, и целочисленное деление, в результате которого мы получим только целую часть частного.

```
In [4]: print(5 / 2) # в результате такого деления получается другой тип данных (float), подробнее о нем поговорим позже.
print(5 // 2)
```

```
2.5
2
```

А если нам надо как раз найти остаток от деления, то мы можем воспользоваться оператором модуло %

```
In [17]: print(5 % 2)
```

```
1
```

Еще одна математическая операция, которую мы можем выполнять без загрузки специальных математических библиотек - это возведение в степень.

```
In [5]: print(5**2)
```

```
25
```

Все то же самое работает и с переменными, содержащими числа.

```
In [19]: a = 2
b = 3
print(a ** b)
# изменится ли результат, если мы перезапишем переменную a?
a = 5
print(a ** b)
```

```
8
```

```
125
```

Часто возникает ситуация, что мы считали какие-то данные в формате текста, и у нас не работают арифметические операции. Тогда мы можем с помощью функции int() преобразовать строковую переменную (о них ниже) в число, если эта строка может быть переведена в число в десятичной системе.

```
In [6]: print(2 + '5') # ошибка, не сможем сложить целое число и строку
```

```
-----
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-6-f28303544e9d> in <module>()
----> 1 print(2 + '5') # ошибка, не сможем сложить целое число и строку
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
In [7]: print(2 + int('5')) # преобразовали строку в число и все заработало
```

7

```
In [8]: int('текст') # здесь тоже поймаем ошибку, потому что строка не представляет собой число
```

```
-----  
ValueError Traceback (most recent call last)  
<ipython-input-8-5879db2a084c> in <module>()  
----> 1 int('текст') # здесь тоже поймаем ошибку, потому что строка не представляет собой число
```

```
ValueError: invalid literal for int() with base 10: 'текст'
```

(□ ` -)□—☆° . * ° Задача

Сумма цифр трехзначного числа

Дано трехзначное число 179. Найдите сумму его цифр.

Формат вывода

Выведите ответ на задачу.

Ответ

Вывод программы:

17

```
In [9]: # (□ ` - )□—☆° . * °
```

```
x = 179
x_1 = x // 100
x_2 = x // 10 % 10
x_3 = x % 10
print(x_1, x_2, x_3) # тестовый вывод, проверяем, что правильно "до
# стали" цифры из числа
print(x_1 + x_2 + x_3) # ответ на задачу
```

1 7 9

17

(□ ` -)◦—☆°.*..° Задача

Электронные часы

Дано число N. С начала суток прошло N минут. Определите, сколько часов и минут будут показывать электронные часы в этот момент.

Формат ввода

Вводится число N — целое, положительное, не превышает 10^7 .

Формат вывода

Программа должна вывести два числа: количество часов (от 0 до 23) и количество минут (от 0 до 59).

Учтите, что число N может быть больше, чем количество минут в сутках.

Примеры

Тест 1

Входные данные:

150

Вывод программы:

2 30

```
In [2]: # (□ ` - )◦—☆°.*..°  
minutes = 150  
print(minutes // 60 % 24, minutes % 60)
```

2 30

Типы данных: логические или булевые переменные (boolean)

Следующий тип данных - это логические переменные. Логические переменные могут принимать всего два значения - **истина (True)** или **ложь (False)**. В Python тип обозначается **bool**.

```
In [12]: print(type(True), type(False))
```

```
<class 'bool'> <class 'bool'>
```

Логические переменные чаще всего используется в условных операторах if-else и в цикле с остановкой по условию while. В части по анализу данных еще увидим одно частое применение - создание булевых масок для фильтрации данных (например, вывести только те данные, где возраст больше 20).

Обратите внимание, что True и False обязательно пишутся с большой буквы и без кавычек, иначе можно получить ошибку.

```
In [22]: print(type('True')) # тип str - строковая переменная
print(true) # ошибка, Python думает, что это название переменной

<class 'str'>
-----
NameError Traceback (most recent call last)
all last)
<ipython-input-22-7bacab652a4cb> in <module>()
      1 print(type('True')) # тип str - строковая переменная
----> 2 print(true) # ошибка, Python думает, что это название переменной
NameError: name 'true' is not defined
```

Как и в случае чисел и строк, с логическими переменными работает преобразование типов. Превратить что-либо в логическую переменную можно с помощью функции bool().

Преобразование чисел работает следующим образом - 0 превращается в False, все остальное в True.

```
In [33]: print(bool(0))
print(bool(23))
print(bool(-10))

False
True
True
```

Пустая строка преобразуется в False, все остальные строки в True.

```
In [34]: print(bool(''))
print(bool('Hello'))
print(bool(' ')) # даже строка из одного пробела - это True
print(bool('False')) # и даже строка 'False' - это True
```

```
False
True
True
True
```

И при необходимости булеву переменную можно конвертировать в int. Тут все без сюрпризов - ноль и единица.

```
In [23]: print(int(True))
print(int(False))
```

```
1
0
```

Логические выражения

Давайте посмотрим, где используется новый тип данных.

Мы поработаем с логическими выражениями, результат которых - булевые переменные.

Логические выражения выполняют проверку на истинность, то есть выражение равно True, если оно истинно, и False, если ложно.

В логических выражениях используются операторы сравнения:

- == (равно)
- != (не равно)
- > (больше)
- < (меньше)
- >= (больше или равно)
- <= (меньше или равно)

```
In [24]: print(1 == 1)
print(1 != '1')
c = 1 > 3
print(c)
print(type(c))
x = 5
print(1 < x <= 5) # МОЖНО ПИСАТЬ СВЯЗКИ ЦЕПОЧКОЙ
```

```
True
True
False
<class 'bool'>
True
```

Логические выражения можно комбинировать с помощью следующих логических операторов:

- логическое И (and) - выражение истинно, только когда обе части истинны, иначе оно ложно
- логическое ИЛИ (or) - выражение ложно, только когда обе части ложны, иначе оно истинно
- логическое отрицание (not) - превращает True в False и наоборот

```
In [37]: print((1 == 1) and ('1' == 1))
print((1 == 1) or ('1' == 1))
print(not(1 == 1))
print(((1 == 1) or ('1' == 1)) and (2 == 2))
```

```
False
True
False
True
```

(∩｀-˘)⊃━☆ﾟ.*◦◦ Задача

Вася в Италии

Вася уехал учиться по обмену на один семестр в Италию. Единственный магазин в городе открыт с 6 до 8 утра и с 16 до 17 вечера (включительно). Вася не мог попасть в магазин уже несколько дней и страдает от голода. Он может прийти в магазин в X часов. Если магазин открыт в X часов, то выведите True, а если закрыт - выведите False.

В единственной строке входных данных вводится целое число X, число находится в пределах от 0 до 23

Формат ввода

Целое число X, число находится в пределах от 0 до 23

Формат вывода

True или False

Примеры

Тест 1

Входные данные:

16

Вывод программы:

True

```
In [39]: ## (∩｀-˘)⊃━☆ﾟ.*◦◦
time = 16
can_visit = 6 <= time <= 8
can_visit2 = 16 <= time <= 17
print(can_visit or can_visit2)
```

True

Типы данных: вещественные числа (float)

По сути, вещественные числа это десятичные дроби, записанные через точку. Вещественные числа в питоне обозначаются словом float (от "плавающей" точки в них). Также могут быть представлены в виде инженерной записи: $1/10000 = 1e-05$

ВЕЩЕСТВЕННЫЕ ЧИСЛА (FLOAT): 3.42, 2.323212, 3.0, 1e-05 - число с дробной частью (в том числе целое с дробной частью равной 0)

```
In [10]: 4.5 + 5
```

```
Out[10]: 9.5
```

Если у нас было действие с участием целого и вещественного числа, то ответ всегда будет в виде вещественного числа (см. выше).

Также давайте вспомним про "обычное" деление, в результате которого получается вещественное число.

```
In [11]: print(11 / 2)
print(type(11 / 2))
print(11 // 2)
print(type(11 // 2))
```

```
5.5
<class 'float'>
5
<class 'int'>
```

С вещественными числами нужно быть осторожными со сравнениями. В связи с особенностями устройства памяти компьютера дробные числа хранятся там весьма хитро и не всегда условные 0.2 то, чем кажутся. Это связано с проблемой точности представления чисел.

Подробнее можно прочитать [здесь \(https://pythoner.name/documentation/tutorial/floatingpoint\)](https://pythoner.name/documentation/tutorial/floatingpoint).

```
In [7]: 0.2 + 0.1 == 0.3
```

```
Out[7]: False
```

Наверняка, от такого равенства мы ожидали результат True, но нет. Поэтому будьте аккуратны и старайтесь не "заязыывать" работу вашей программы на условия, связанные с вычислением вещественных чисел. Давайте посмотрим, как на самом деле выглядят эти числа в памяти компьютера.

```
In [62]: print(0.2 + 0.1)
print(0.3)
```

```
0.30000000000000004
0.3
```

Числа с плавающей точкой представлены в компьютерном железе как дроби с основанием 2 (двоичная система счисления). Например, десятичная дробь

0.125

имеет значение $1/10 + 2/100 + 5/1000$, и таким же образом двоичная дробь

0.001

имеет значение $0/2 + 0/4 + 1/8$. Эти две дроби имеют одинаковые значения, отличаются только тем, что первая записана в дробной нотации по основанию 10, а вторая по основанию 2.

К сожалению, большинство десятичных дробей не могут быть точно представлены в двоичной записи. Следствием этого является то, что в основном десятичные дробные числа вы вводите только приближенными к двоичным, которые и сохраняются в компьютере.

Если вам совсем не обойтись без такого сравнения, то можно сделать так: сравнивать не результат сложения и числа, а разность этих двух чисел с каким-то очень маленьким числом (с таким, размер которого будет точно не критичен для нашего вычисления). Например, порог это числа будет разным для каких-то физических вычислений, где важна высокая точность, и сравнения доходов граждан.

```
In [46]: 0.2 + 0.1 - 0.3 < 0.000001
```

Out[46]: True

Следующая проблема, с которой можно столкнуться - вместо результата вычисления получить ошибку 'Result too large'. Связано это с ограничением выделяемой памяти на хранение вещественного числа.

```
In [21]: 1.5 ** 100000
```

```
-----
-----
OverflowError
all last)
<ipython-input-21-449e5e88bdf9> in <module>()
----> 1 1.5 ** 100000
```

Traceback (most recent c

OverflowError: (34, 'Result too large')

```
In [23]: 1.5 ** 1000
```

```
Out[23]: 1.2338405969061735e+176
```

А если все получилось, то ответ еще может выглядеть вот так. Такая запись числа называется научной и экономит место - она хранит целую часть числа (мантилла) и степень десятки на которую это число нужно умножить (экспонента). Здесь результатом возведения 1.5 в степень 1000 будет число 1.2338405969061735, умноженное на 10 в степень 176. Понятно, что это число очень большое. Если бы вместо знака + стоял -, то и степень десятки была бы отрицательная (10^{-176} степени), и такое число было бы очень, очень маленьким.

Как и в случае с целыми числами, вы можете перевести строку в вещественное число, если это возможно. Сделать это можно функцией float()

```
In [63]: print(2.5 + float('2.4'))
```

```
4.9
```

Округление вещественных чисел

У нас часто возникает необходимость превратить вещественное число в целое ("округлить"). В питоне есть несколько способов это сделать, но, к сожалению, ни один из них не работает как наше привычное округление и про это всегда нужно помнить.

Большинство этих функций не реализованы в базовом наборе команд питона и для того, чтобы ими пользоваться, нам придется загрузить дополнительную библиотеку math, которая содержит всякие специальные функции для математических вычислений.

```
In [51]: import math # команда import загружает модуль под названием math
```

Модуль math устанавливается в рамках дистрибутива Anaconda, который мы использовали, чтобы установить Jupyter Notebook, поэтому его не нужно отдельно скачивать, а можно просто импортировать (загрузить в оперативную память текущей сессии). Иногда нужную библиотеку придется сначала установить на компьютер с помощью команды !pip install -название модуля- и только потом импортировать.

Самый простой способ округлить число - применить к нему функцию int.

```
In [47]: int(2.6)
```

```
Out[47]: 2
```

Обратите внимание, что такой метод просто обрубает дробную часть (значения выше 0.5 не округляются в сторону большего числа).

```
In [49]: print(int(2.6))
print(int(-2.6))
```

```
2
-2
```

Округление "в пол" из модуля math округляет до ближайшего меньшего целого числа.

```
In [52]: print(math.floor(2.6)) # чтобы использовать функцию из дополнительно
го модуля -
# нужно сначала написать название этого мод
уля и через точку название функции
print(math.floor(-2.6))
```

```
2
-3
```

Округление "в потолок" работает ровно наоборот - округляет до ближайшего большего числа, независимо от значения дробной части.

```
In [53]: print(math.ceil(2.6))
print(math.ceil(-2.6))
```

```
3
-2
```

В самом питоне есть еще функция round(). Вот она работает почти привычно нам, если бы не одно "но"...

```
In [54]: print(round(2.2))
print(round(2.7))
print(round(2.5)) # внимание на эту строку
print(round(3.5))
```

```
2
3
2
4
```

Неожиданно? Тут дело в том, что в питоне реализованы не совсем привычные нам правила округления чисел с вещественной частью 0.5 - такое число округляется до ближайшего четного числа: 2 для 2.5 и 4 для 3.5.

Замечание по импорту функций

Иногда нам не нужна вся библиотека, а только одна функция из-за нее. Скажите, странно же хранить в оперативной памяти всю "Войну и мир", если нас интересует только пятое предложение на восьмой странице. Для этого можно воспользоваться импортом функции из библиотеки и тогда не нужно будет писать ее название через точку. Подводный камень здесь только тот, что если среди базовых команд питона есть функция с таким же именем, то она перезапишется и придется перезапускать свой блокнот, чтобы вернуть все как было.

```
In [32]: from math import ceil  
ceil(2.6) # теперь работает без math.
```

```
Out[32]: 3
```

(□ ` -)◦—☆° . * · ° Задача

Дробная часть

Дано вещественное число. Выведите его дробную часть.

Формат ввода

Вещественное число

Формат вывода

Вещественное число (ответ на задачу)

Примеры

Тест 1

Входные данные:

4.0

Вывод программы:

0.0

```
In [58]: # (0 ` -` )⇒-★°.*·°  
x = 4.0  
print(x - int(x))  
  
x = 5.2  
print(x - int(x))
```

```
0.0  
0.20000000000000018
```

6.04.2020 | МИРЭК | 4 модуль

Автор: Татьяна Рогович, Алла Тамбовцева

Основы программирования в Python

Лекция 1. Тема 2

Python Refresher: строки, ввод, форматирование вывода.

ТЕКСТ (СТРОКИ) (STR, STRING): любой текст внутри одинарных или двойных кавычек. Важно: целое число в кавычках - это тоже строка, не целое число.

```
In [1]: x = 'text'  
print(type(x))  
print(type('Hello, world!'))  
print(type('2'))
```

```
<class 'str'>  
<class 'str'>  
<class 'str'>
```

Если попробовать сложить число-строку и число-число, то получим ошибку.

```
In [2]: print('2' + 3)
```

```
-----  
-----  
TypeError                                     Traceback (most recent c  
all last)  
<ipython-input-2-81acf9e7aa0> in <module>()  
----> 1 print('2' + 3)  
  
TypeError: must be str, not int
```

Еще раз обратим внимание на текст ошибки. *TypeError: must be str, not int* - TypeError означает, что данная команда не может быть исполнена из-за типа какой-то из переменных. В данном случае видим, что что-то должно быть "str", когда оно типа "int". Давайте попробуем сделать 3 тоже строкой. Кстати, неважно какие вы используете кавычки (только если это не текст с кавычками внутри), главное, чтобы открывающая и закрывающая кавычки были одинаковые.

```
In [3]: print('2' + "3")
```

```
23
```

Такая операция называется конкатенацией (слиянием) строк.

Можно и умножить строку на число можно. Такая операция повторит нам строку заданное количество раз.

```
In [4]: '2' * 3
```

```
Out[4]: '222'
```

Операции со строками тоже работают, если строки лежат внутри переменных.

```
In [5]: word1 = 'John'  
word2 = ' Brown'  
print(word1 + word2)  
word3 = word1 + word2 # можем результат конкатенации положить в новую переменную  
print(word3)
```

```
John Brown  
John Brown
```

В прошлом блокноте мы преобразовывали строки в числа с помощью функции int().

```
In [6]: print(2 + int('2512')) # нет ошибки!
```

```
2514
```

С помощью функции str() мы можем превращать другие типы данных в строки.

```
In [7]: print('abs' + str(123) + str(True))
```

```
abs123True
```

Обратите внимание: эти функции (str, int, float и другие) не меняют тип самих данных или переменных, в которых они хранятся. Если мы не перезапишем значение, то строка станет числом только в конкретной строчке команды, а ее тип не изменится.

```
In [8]: a = '2342123'  
print(type(a))  
print(2 + int(a))  
print(type(a))
```

```
<class 'str'>  
2342125  
<class 'str'>
```

```
In [9]: a = int(a) # перезаписываем значение переменной  
print(type(a)) # теперь изменился и тип
```

```
<class 'int'>
```

Ввод данных. input()

Познакомимся с функцией `input()` - ввод. Это функция позволяет нам просить пользователя что-нибудь ввести или принять на ввод какие-то данные.

```
In [12]: x = input('Как вас зовут? ')  
print(x)
```

```
Иван
```

'Как вас зовут?' - аргумент функции `input()`, который будет отображаться в строке ввода. Он необязателен, но он может быть полезен для вас (особенно, если в задаче требуется ввести больше одной строки данных и нужно не запутаться).

Обратите внимание, что функция `input()` все сохраняет в строковом формате данных. Но мы можем сделать из числа-строки число-число с помощью функции `int()`. Введите "1" в обоих случаях и сравните результат.

```
In [6]: print(type(input()))  
print(type(int(input())))
```

```
<class 'str'>  
<class 'int'>
```

Нужно следить за тем, какие данные вы считываете и менять их при необходимости. Сравните два примера ниже.

```
In [2]: x = input('Input x value: ')
y = input('Input y value: ')
print(x + y) # произошла склейка строк, по умолчанию input() сохраняет значение типа str
```

23

```
In [3]: x = int(input('Input x value: '))
y = int(input('Input y value: '))
print(x + y) # вот теперь произошло сложение чисел
```

5

(Ո՝ -՛) Ե—☆°.*•。° Задача

Pig Latin 1

Pig Latin - вымышленный язык, который в конец каждого слова подставляет 'уау' (на самом деле оригинальные условия задачи несколько хитрее, но мы будем к ней возвращаться по ходу курса.

Формат ввода

Пользователь вводит слова.

Формат вывода Слово на Pig Latin.

Примеры

Тест 1

Ввод:

apple

Вывод:

appleyay

```
In [2]: # (Ո՝ -՛) Ե—☆°.*•。°
word = input('Введите слово: ')
print(word+'yay')
```

appleyay

(□ ` -)◦—☆°.*..° Задача

Х ПОВТОРЕННОЕ Х раз в квадрате

Вводится целое число X. Повторите его X раз и возведите полученное число в квадрат.
Напечатайте получившееся.

Формат ввода

Целое число

Формат вывод

Ответ на задачу

Примеры

Тест 1

Ввод:

2

Вывод:

484

```
In [6]: x = input()
print((int(x)*int(x)))**2)
```

484

(Ω ` -)◦—☆°.*◦° Задача

Хитрости умножения

Для умножения двузначного числа на 11 есть интересная хитрость: результат произведения можно получить если между первой и второй цифрой исходного числа вписать сумму этих цифр. Например, $15 \cdot 11 = 1 \cdot 1 + 5 \cdot 5 = 165$ или $34 \cdot 11 = 3 \cdot 3 + 4 \cdot 4 = 374$. Реализуйте программу, которая умножала бы двузначные числа на 11, используя эту хитрость. Пользоваться оператором умножения нельзя.

Формат ввода:

Вводится двузначное число.

Формат вывода:

Программа должна вывести ответ на задачу.

Тест 1**Пример ввода:**

15

Вывод:

165

Тест 2**Пример ввода:**

66

Вывод:

726

```
In [7]: # Ваше решение здесь
x = input()

n1 = int(x[0])
n2 = int(x[1])

if n1 + n2 < 10:
    print(str(n1) + str(n1+n2) + str(n2))
else:
    print(str(n1 + 1) + str((n1 + n2) % 10) + str(n2))
```

726

Форматирование строк (string formatting)

А теперь посмотрим на то, как подставлять значения в уже имеющийся текстовый шаблон, то есть форматировать строки. Чтобы понять, о чём идет речь, можно представить, что у нас есть электронная анкета, которую заполняет пользователь, и мы должны написать программу, которая выводит на экран введенные данные, чтобы пользователь мог их проверить.

Пусть для начала пользователь вводит свое имя и возраст.

```
In [3]: name = input("Введите Ваше имя: ")
age = int(input("Введите Ваш возраст: ")) # возраст будет целочисле-
нным
```

Теперь выведем на экран сообщение вида

Ваше имя: имя . Ваш возраст: возраст .

Но прежде, чем это сделать, поймем, какого типа будут значения, которые мы будем подставлять в шаблон. Имя (переменная name) – это строка (string), а возраст (переменная age) – это целое число (integer).

```
In [4]: result = "Ваше имя: %s. Ваш возраст: %i." % (name, age)
print(result)
```

Ваше имя: Олег. Ваш возраст: 32.

Что за таинственные %s и %i? Все просто: оператор % в строке указывает место, на которое будет подставляться значение, а буква сразу после процента – сокращённое название типа данных (s – от string и i – от integer). Осталось только сообщить Python, что именно нужно подставлять – после кавычек поставить % и в скобках перечислить названия переменных, значения которых мы будем подставлять.

Для тех, кто работал в R: форматирование строк с помощью оператора % – аналог форматирования с помощью функции sprintf() в R.

Примечание: не теряйте часть с переменными после самой строки. Иначе получится нечто странное:

```
In [5]: print("Ваше имя: %s. Ваш возраст: %i.")
```

Ваше имя: %s. Ваш возраст: %i.

Важно помнить, что если мы забудем указать какую-то из переменных, мы получим ошибку (точнее, исключение): Python не будет знать, откуда брать нужные значения.

```
In [6]: print("Ваше имя: %s. Ваш возраст: %i." % (name))
```

```
-----  
-----  
TypeError Traceback (most recent call last)  
<ipython-input-6-2343aa81e176> in <module>()  
----> 1 print("Ваше имя: %s. Ваш возраст: %i." % (name))  
  
TypeError: not enough arguments for format string
```

Кроме того, создавая такие текстовые шаблоны, нужно обращать внимание на типы переменных, значения которых мы подставляем.

```
In [7]: print("Ваше имя: %s. Ваш возраст: %s." % (name, age)) # так сработает
```

```
Ваше имя: Олег. Ваш возраст: 32.
```

```
In [8]: print("Ваше имя: %i. Ваш возраст: %s." % (name, age)) # а так нет
```

```
-----  
-----  
TypeError Traceback (most recent call last)  
<ipython-input-8-9c083ea32449> in <module>()  
----> 1 print("Ваше имя: %i. Ваш возраст: %s." % (name, age)) # а так нет  
  
TypeError: %i format: a number is required, not str
```

В первом случае код сработал: Python не очень строго относится к типам данных, и поэтому он легко может превратить целочисленный возраст в строку (два %s вместо %s и %i не является помехой). Во втором случае все иначе. Превратить строку, которая состоит из букв (name) в целое число никак не получится, поэтому Python справедливо ругается.

А что будет, если мы будем подставлять не целое число, а дробное, с плавающей точкой?
Попробуем!

```
In [9]: height = float(input("Введите Ваш рост (в метрах): "))  
height
```

```
Out[9]: 1.78
```

```
In [10]: print("Ваш рост: %f м." % height) # f - от float
```

Ваш рост: 1.780000 м.

По умолчанию при подстановке значений типа float Python выводит число с шестью знаками после запятой. Но это можно исправить. Перед f нужно поставить точку и указать число знаков после запятой, которое мы хотим:

```
In [11]: print("Ваш рост: %.2f м." % height) # например, два
```

Ваш рост: 1.78 м.

```
In [12]: print("Ваш рост: %.1f м. " % height) # или один
```

Ваш рост: 1.8 м.

В случае, если указанное число знаков после запятой меньше, чем есть на самом деле (как в ячейке выше), происходит обычное арифметическое округление.

Рассмотренный выше способ форматирования строк – не единственный. Он довольно стандартный, но при этом немного устаревший. В Python 3 есть другой способ – форматирование с помощью метода .format(). Кроме того, в Python 3.6 и более поздних версиях появился еще более продвинутый способ форматирования строк - f-strings (formatted string literals).

F-strings очень удобны и просты в использовании: вместо % и сокращённого названия типа в фигурных скобках внутри текстового шаблона нужно указать название переменной, из которой должно подставляться значение, а перед всей строкой добавить f, чтобы Python знал, что нам нужна именно f-string.

```
In [16]: print(f"Ваше имя: {name}. Ваш возраст: {age}. Рост: {height:.2f}")
```

Ваше имя: Олег. Ваш возраст: 32. Рост: 1.78

Альтернативой такого кода будет следующий синтаксис. Здесь переменные вставляются в порядке "упоминания" в строке.

```
In [14]: print("Ваше имя: {}. Ваш возраст: {}".format(name, age))
```

Ваше имя: Олег. Ваш возраст: 32

Форматирование дробей добавляем так.

```
In [15]: print("Ваше имя: {}. Ваш возраст: {}. Рост: {:.2f}".format(name, age, height))
```

```
Ваше имя: Олег. Ваш возраст: 32. Рост: 1.78
```

Если указали формат переменной (.2f, например, ожидает float), то следим за порядком.

```
In [17]: print("Ваше имя: {}. Ваш возраст: {}. Рост: {:.2f}".format(age, height, name))
```

```
-----
-----
ValueError                                     Traceback (most recent call last)
<ipython-input-17-f42c355e5798> in <module>()
----> 1 print("Ваше имя: {}. Ваш возраст: {}. Рост: {:.2f}".format(age, height, name))

ValueError: Unknown format code 'f' for object of type 'str'
```

Порядок заполнения можно указывать с помощью нумерации. Одну и тоже переменную можно использовать несколько раз.

```
In [32]: print("Ваше имя: {2}. Ваш возраст: {0}. Рост: {1:.2f}. Все верно, {2}?).format(age, height, name))
```

```
Ваше имя: Олег. Ваш возраст: 32. Рост: 1.78. Все верно, Олег?
```

Также при форматировании строк можно использовать результаты вычислений и результаты работы методах (о них немного позже). `name.upper()` в примере делает все символы строки `name` заглавными.

```
In [36]: print(f"Ваше имя: {name.upper()}. Ваш возраст: {2020-1988}. Рост в футах: {height * 100 / 30.48 :.2f}")
```

```
Ваше имя: ОЛЕГ. Ваш возраст: 32. Рост в футах: 5.84
```

(Π ` -)◦—☆°.*..° Задача

Pi

Из модуля math импортируйте переменную pi. С помощью %f (первое предложение) и format (второе предложение) форматирования выведите строки из примера. Везде, где встречается число pi - это должна быть переменная pi, округленная до определенного знака после точки.

Формат вывода:

Значение 22/7 (3.14) является приближением числа pi (3.1416)

Значение 22/7 3.142 является приближением числа pi 3.141592653589793

```
In [25]: import math
print(math.pi)
print('Значение 22/7 (%.2f) является приближением числа pi (%.4f)'
% (math.pi, math.pi))
print(f'Значение 22/7 {math.pi:.3f} является приближением числа pi
{math.pi}' )
```

3.141592653589793

Значение 22/7 (3.14) является приближением числа pi (3.1416)

Значение 22/7 3.142 является приближением числа pi 3.1415926535897
93

Функция print(): аргументы и параметры

И еще немного поговорим про форматирование вывода. Это вам пригодится, если будете решать много задач на разных сайтах вроде hackerrank, где ответ нужно выводить каким-то хитрым образом. На самом деле функция print() немного сложнее, чем то, что мы уже видели. Например, мы можем печатать одной функцией больше чем один аргумент и использовать разделители вместо пробелов.

```
In [26]: print('2 + 3 =', 2 + 3)
z = 5
print('2 + 3 =', z)
```

2 + 3 = 5
2 + 3 = 5

У функции print есть параметры. Параметры - это такие свойства функций, которые задают значение невидимых нам аргументов внутри функции. Так существует параметр sep (separator), благодаря которому мы можем менять тип разделителя между аргументами print. В качестве разделителя может выступать любая строка.

Сравните:

```
In [29]: print('1', '2', '3')
      print('1', '2', '3', sep='.')
      print('1', '2', '3', sep=' ')
      print('1', '2', '3', sep='\n')
```

```
1 2 3
1.2.3
123
1
2
3
```

Параметр end задает то, что будет выведено в конце исполнения функции print. По умолчанию там стоит невидимый символ '\n', который осуществляет переход на новую строку. Мы можем заменить его на любую строку. Если мы хотим сохранить переход на новую строку - то обязательно прописываем наш невидимый символ внутри выражения.

```
In [28]: print('1', '2', '3', sep='.', end='!')
      print('2') # строки слились

      print('1', '2', '3', sep='.', end='!\n')
      print('2') # вывод на новой строке
```

```
1.2.3!2
1.2.3!
2
```

6.04.2020 | МИРЭК | 4 модуль

Автор: Татьяна Рогович

Основы программирования в Python

Лекция 1. Тема 4

Python Refresher: списки и кортежи

Списки (list)

Давайте представим, что при написании программы нам нужно работать, например, с большой базой данных студентов университета.

Если студентов несколько сотен, нет смысла создавать для каждого отдельную переменную - нам нужно научиться сохранять их всех в одной переменной. Начиная с этого занятия, мы будем изучать типы данных, которые позволяют это делать.

Начнем мы со **списков**. Если вы изучали другие языки программирования, то наверняка знакомы с аналогичным типом данных - массивами. Как и строки, списки - это последовательности, упорядоченные данные.

Давайте для начала попробуем создать список из 3 студентов.

```
In [1]: students = ['Ivan Ivanov', 'Tatiana Sidorova', 'Maria Smirnova']
print(students)
print(type(students))
```



```
[ 'Ivan Ivanov', 'Tatiana Sidorova', 'Maria Smirnova' ]
<class 'list'>
```

Пустой список можно создать двумя способами - оператором [] и функцией list().

```
In [2]: print([])
print(list())
```



```
[]
[]
```

Список может содержать любые данные - например, числа.

Давайте создадим список оценок студента.

```
In [3]: notes = [6, 5, 7, 5, 8]
print(notes)
```

```
[6, 5, 7, 5, 8]
```

Список может быть даже смешанным. Например, давайте сохраним в одном списке имя студента, его год рождения, средний балл, и логическую переменную, которая будет равна True, если студент учится на бюджете.

```
In [4]: student1 = ['Ivan Ivanov', 1987, 7.5, True]
print(student1)
```

```
['Ivan Ivanov', 1987, 7.5, True]
```

Список может даже содержать другие списки.

Давайте создадим еще одного студента по аналогии со student1 и положим этих двух студентов в еще один список.

```
In [5]: student2 = ['Maria Smirnova', 1991, 7.9, False]
students = [student1, student2]
print(students)
```

```
[['Ivan Ivanov', 1987, 7.5, True], ['Maria Smirnova', 1991, 7.9, False]]
```

Элементы списков нумеруются, начиная с 0. Мы можем получить доступ к элементу списка по его индексу.

```
In [6]: students = ['Ivan Ivanov', 'Tatiana Sidorova', 'Maria Smirnova']
print(students[0]) # первый элемент
print(students[1]) # второй элемент
print(students[-1]) # последний элемент
```

```
Ivan Ivanov
Tatiana Sidorova
Maria Smirnova
```

Кстати, индексация работает и в строках. Там отдельными элементами являются символы.

```
In [7]: x = 'СЛОВО'
print(x[0])
print(x[-1])
```

C
O

Мы можем узнать длину списка с помощью функции len() (работает и для строк).

```
In [7]: print(len(students)) # количество элементов в списке students
print(len(x)) # количество символов в строке x
```

3
5

Но, в отличие от строк, список можно изменить.

```
In [10]: students[1] = 'Petr Petrov' # заменяем второго студента в списке на
Петра
print(students)
```

['Ivan Ivanov', 'Petr Petrov', 'Maria Smirnova']

```
In [9]: x[1] = 'a'
```

```
-----
-----
TypeError                                     Traceback (most recent call last)
all last)
<ipython-input-9-195a27d758c7> in <module>()
----> 1 x[1] = 'a'

TypeError: 'str' object does not support item assignment
```

Строки - неизменяемый тип данных, поэтому присвоение символа по индексу не сработает.

Еще одна операция, которая работает со всеми последовательностями - проверка на наличие элемента in и not in. Возвращает True или False.

```
In [8]: print('Ivan Ivanov' in students)
print('Petr' not in students[2])
print(2 in students)
```

True
True
False

Способ расширить список - метод .append(), который добавляет аргумент в список в качестве последнего элемента.

```
In [9]: lst = [0, 1, 2, 3, 4, 'cat', 'dog']

lst.append(5)
print(lst)

lst += [6] # эквивалентное append выражение
print(lst)

[0, 1, 2, 3, 4, 'cat', 'dog', 5]
[0, 1, 2, 3, 4, 'cat', 'dog', 5, 6]
```

Удалить элемент из списка можно с помощью метода .remove() (без возвращения удаленного элемента) или .pop (с возвращением удаленного элемента).

```
In [10]: lst.remove('cat') # аргумент - объект, который хотим удалить
print(lst)

[0, 1, 2, 3, 4, 'dog', 5, 6]
```

```
In [11]: x = lst.pop(-1) # аргумент - индекс объекта. Результат операции может
# сохранить в переменную
print(lst)
print(x)

[0, 1, 2, 3, 4, 'dog', 5]
6
```

Поиском в списках занимается метод .index(), который вернет индекс объекта, переданного в качестве аргумента.

```
In [15]: print(lst.index('dog')) # находим индекс объекта 'dog'
print(lst[lst.index('dog')]) # используем метод, возвращающий индекс,
# для обращения к объекту

5
dog
```

Если говорить еще о полезных методах, то это .count(), который подсчитывает количество элементов и .reverse(), который разворачивает список. Ниже еще отдельно поговорим о сортировке.

```
In [108]: print(lst.count('dog')) # считаем количество 'dog' в списке
lst.reverse() # разворачиваем список. Осторожно - метод меняет список
OK!
print(lst)

1
[5, 'dog', 4, 3, 2, 1, 0]
```

Все методы списков [здесь \(https://docs.python.org/3/tutorial/datastructures.html\)](https://docs.python.org/3/tutorial/datastructures.html).

Сортировки

Отдельно следует рассказать про метод **sort()**. Метод производит сортировку списка. Задачи сортировки - очень распространены в программировании. В общем случае, они сводятся к выстроению элементов списка в заданном порядке. В Python есть встроенные методы для сортировки объектов для того, чтобы программист мог не усложнять себе задачу написанием алгоритма сортировки. Метод **list.sort()** - как раз, один из таких случаев.

```
In [18]: test_list = [5, 8, 1, 4, 3, 7, 2]
print(test_list) # Элементы списка расположены в хаотичном порядке
test_list.sort()
print(test_list) # Теперь элементы списка теперь расположены по возрастанию

[5, 8, 1, 4, 3, 7, 2]
[1, 2, 3, 4, 5, 7, 8]
```

Таким образом, метод **list.sort()** упорядочил элементы списка `test_list`. Если нужно отсортировать элементы в обратном порядке, то можно использовать именнованный параметр `reverse`.

```
In [19]: test_list.sort(reverse=True) # параметр reverse указывает на то, что нужно отсортировать список в обратном порядке
print(test_list)

[8, 7, 5, 4, 3, 2, 1]
```

Следует обратить внимание, что метод **list.sort()** изменяет сам список, на котором его вызвали. Таким образом, при каждом вызове метода "**sort()**", наш список "`test_list`" изменяется. Это может быть удобно, если нам не нужно держать в памяти исходный список. Однако, в противном случае, или же - в случае неизменяемого типа данных (например, кортежа или строки) - этот метод не сработает. В таком случае, на помощь приходит встроенная в питон функция **sorted()**.

```
In [111]: print(sorted(test_list)) # Сам список при сортировке не изменяется
[1, 2, 3, 4, 5, 7, 8]
```

Так как sorted() функция, а не метод, то будет работать и с другими типами данных.

```
In [121]: print(sorted('something')) # отсортирует буквы в строке, но выведет список
print(''.join(sorted('something'))) # с помощью метода join можно с обрать отсортированную строку, чуть ниже подробнее про него
['e', 'g', 'h', 'i', 'm', 'n', 'o', 's', 't']
eghimnost
```

У функции sorted(), как и у метода list.sort() есть параметр key, с помощью которого можно указать функцию, которая будет применена к каждому элементу последовательности при сортировке.

```
In [20]: test_string = 'A string With upper AND lower cases'
print(sorted(test_string.split())) # заглавные буквы получили приоритет над строчными
print(sorted(test_string.split(), key=str.upper)) # все буквы были приведены к верхнему регистру, сортировка получилась по алфавиту
['A', 'AND', 'With', 'cases', 'lower', 'string', 'upper']
['A', 'AND', 'cases', 'lower', 'string', 'upper', 'With']
```

Кортежи (tuple)

Кортежи очень похожи на списки.

```
In [11]: student = ('Ivan Ivanov', 2001, 7.5, True)
print(student)
print(type(student))

('Ivan Ivanov', 2001, 7.5, True)
<class 'tuple'>
```

Пустой кортеж можно создать с помощью оператора () либо функции tuple.

```
In [12]: print()
print(tuple())
()
()
```

Основное отличие кортежей от списков состоит в том, что кортежи нельзя изменять (да-да, прямо как строки).

```
In [13]: student[1] = 2002
```

```
-----
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-13-cf841071c98c> in <module>()
      1 student[1] = 2002
TypeError: 'tuple' object does not support item assignment
```

Списки и кортежи могут быть вложены друг в друга.

Например, пусть в информации о студенте у нас будет храниться не его средний балл, а список всех его оценок.

```
In [14]: student = ('Ivan Ivanov', 2001, [8, 7, 7, 9, 6], True)
print(student)

('Ivan Ivanov', 2001, [8, 7, 7, 9, 6], True)
```

Мы можем обратиться к элементу вложенного списка или кортежа с помощью двойной индексации.

```
In [15]: print(student[2][1]) # получили вторую оценку
```

7

Иногда бывает полезно обезопасить себя от изменений массивов данных и использовать кортежи, но чаще всего мы все-таки будем работать со списками.

Опасность работы с изменяемыми типами данных

В работе со списками есть важный момент, на который нужно обращать внимание. Давайте рассмотрим такой код:

```
In [16]: a = [1, 2, 3]
b = a
b[0] = 4
print(a, b) # изменили b, но a после этого тоже изменился!

[4, 2, 3] [4, 2, 3]
```

Почему так происходит?

Дело в том, что переменная `a` ссылается на место в памяти, где хранится список `[1, 2, 3]`. И когда мы пишем, что `b = a`, `b` начинает указывать на то же самое место. То есть образуется два имени для одного и того же кусочка данных. И после изменения этого кусочка через переменную `b`, значение переменной `a` тоже меняется!

Как это исправить? Нужно создать копию списка `a`! В этом нам поможет метод `.copy()`

```
In [17]: a = [1, 2, 3]
b = a.copy() # теперь переменная b указывает на другой список, который хранится в другом кусочке памяти
b[0] = 4
print(a, b)

[1, 2, 3] [1, 2, 3]
```

Копию можно создавать и с помощью пустого среза

```
In [18]: a = [1, 2, 3]
b = a[:] # по умолчанию берется срез от первого элемента до последнего, то есть копируется весь список a
b[0] = 4
print(a, b)

[1, 2, 3] [1, 2, 3]
```

Но не путайте изменение с присваиванием!

```
In [19]: a = [1, 2, 3]
b = a
a = [4, 5, 6]
print(a, b)

[4, 5, 6] [1, 2, 3]
```

В примере выше переменная `a` была не изменена, а перезаписана, она начала указывать на другой список, хранящийся в другом месте памяти, поэтому переменная `b` осталась нетронута.

Конкатенация списков и кортежей

На списках и кортежах определен оператор `+`. По аналогии со строками, он будет склеивать две части выражения.

Но складывать можно только данные одного типа, список с кортежем склеить нельзя.

```
In [20]: print([1, 2] + [3, 4])
print((1, 2) + (3, 4))
```

```
[1, 2, 3, 4]
(1, 2, 3, 4)
```

```
In [21]: print((1, 2) + [3, 4]) # а так нельзя
```

```
-----
-----  
TypeError                                     Traceback (most recent c
all last)
<ipython-input-21-dbea205bf98e> in <module>()
----> 1 print((1, 2) + [3, 4]) # а так нельзя  
  
TypeError: can only concatenate tuple (not "list") to tuple
```

Но можно превратить список в кортеж, а потом сложить (или наоборот).

```
In [22]: print(list((1, 2)) + [3, 4])
print((1, 2) + tuple([3, 4]))
```

```
[1, 2, 3, 4]
(1, 2, 3, 4)
```

Методы `.split()`, `.join()`, функция `map()`, вывод и ввод списков

При работе со списками довольно часто нам придется вводить их и выводить в отформатированном виде. Сейчас мы рассмотрим несколько методов, которые помогут сделать это в одну строку.

Метод .split()

Метод строки .split() получает на вход строку-разделитель и возвращает список строк, разбитый по этому разделителю.

По умолчанию метод разбивает строку по пробелу

```
In [21]: print('Hello darkness my old friend'.split())
print('Hello    darkness    my old    friend'.split()) # несколько разделителей подряд? не беда, они будут прочитаны как один
```

```
['Hello', 'darkness', 'my', 'old', 'friend']
['Hello', 'darkness', 'my', 'old', 'friend']
```

```
In [22]: print('Ночь. Улица. Фонарь. Аптека'.split('. '))
```

```
['Ночь', 'Улица', 'Фонарь', 'Аптека']
```

Метод .join()

Метод .join() ведет себя с точностью до наоборот - он склеивает массив в строку, вставляя между элементами строку-разделитель.

```
In [23]: print('-'.join(['8', '800', '555', '35', '35']))
```

```
8-800-555-35-35
```

Функция map()

Функция map() берет функцию и последовательность и применяет эту функцию ко всем ее элементам (map() всегда будет ожидать от вас два аргумента).

Обратите внимание, чтобы увидеть результат работы этой функции надо дополнитель но вручную преобразовать в список (или в кортеж, в зависимости от ваших целей).

```
In [119]: print(map(bool, [9, 0, 8, -288, 998, 0])) # не совсем то, что надо
print(list(map(bool, [9, 0, 8, -288, 998, 0]))) # а теперь работает
, каждое число преобразовалось в логическую переменную
```

```
<map object at 0x000001D0FD4C7C50>
[True, False, True, True, True, False]
```

Ввод и вывод списков

Теперь мы можем быстро вводить и выводить списки, содержащие разные данные через разные разделители.

```
In [24]: phone = list(map(int, input().split())) # получаем строку, разбивае
м по пробелу, сразу преобразуем все элементы в числа
print(phone)
print(''.join(list(map(str, phone)))) # преобразуем массив чисел в
массив строк и склеиваем через дефис
```

```
[8, 913, 899, 99, 99]
8-913-899-99-99
```

Также, если нет задачи сохранить результат в переменную в виде строки, вместо join() можно использовать распаковку. Мы ставим оператор * перед списком, и, например, функция print() будет воспринимать его не как список, а как последовательность объектов.

```
In [51]: print(phone, sep='-') # не работает, получили список
print(*phone, sep='-') # теперь сработало. По сути, распакованный с
писок питон видит как в примере ниже
print(phone[0], phone[1], phone[2], phone[3], phone[4], sep='-')
```

```
[8, 913, 899, 99, 99]
8-913-899-99-99
8-913-899-99-99
```

10.04.2020 | МИРЭК | 4 модуль

Автор: Татьяна Рогович

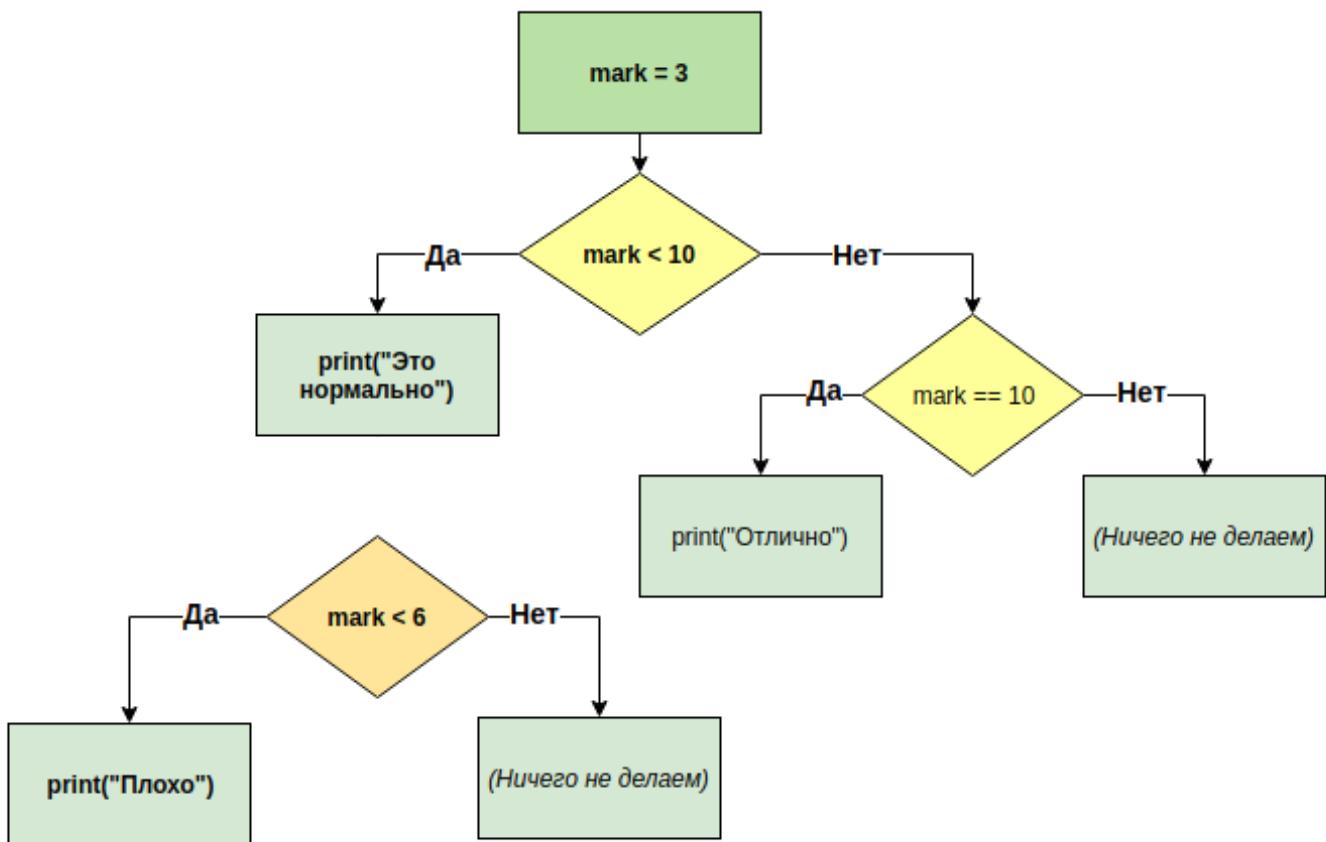
Основы программирования в Python

Лекция 2. Тема 1

Python Refresher: условия 

Условия

Условия это, наверное, душа программирования. Мы очень часто хотим, чтобы наша программа работала по разному в зависимости от происходящего. Алгоритмы, использующие условия, называют разветвляющимися, и для их понимания очень удобно рисовать блок-схемы (нет, они не остались в школе, если в коде много условий и разных действий по-прежнему очень помогает нарисовать на бумаге дерево).



Условия (if) позволяют выполнять код только в случае истинности какого-то логического выражения.

Проще говоря, "если верно, что..., то сделать ...".

Самый простой пример использования if - это вывод какой-то фразы по условию.

```
In [3]: x = 1
if x == 1: # Выражение равно True, это условие истинное
    print('That is true!') # Фраза выводится
```

That is true!

```
In [4]: if x != 1: # Выражение равно False, это условие ложное
    print('That is true!') # Фраза не выводится
```

Обратите внимание, что код, который находится внутри условия, выделяется отступом в 4 пробела или табуляцией (работает не во всех IDE, но в Jupyter все будет хорошо).

Иначе программа не поймет, что он относится к условию.

```
In [5]: if x == 1:  
    print('That is true!')  
  
File "<ipython-input-5-b1fb9bc19953>", line 2  
    print('That is true!')  
          ^  
  
IndentationError: expected an indented block
```

А что делать, если в том случае, когда условие не истинное, мы тоже хотим совершать какое-то действие? Для этого у нас есть ключевое слово **else** ("то").

```
In [6]: if x != 1:  
    print('That is true!')  
else:  
    print('That is false!')
```

That is false!

Мы разобрались, как поступать, если у нас два варианта действий, но их может быть и больше.

Для примера давайте решим простую задачу - найти минимум из двух введенных чисел. Пока ничего нового:

```
In [7]: a = input('Введите первое число: ')  
b = input('Введите второе число: ')  
if a < b:  
    min = a  
else:  
    min = b  
print('Минимум равен', min)
```

Минимум равен 3

А теперь усложним задание, добавив третий вариант развития событий - если числа равны, будем печатать 'Равные числа'.

Можно решить эту задачу с помощью вложенных условий:

```
In [9]: a = input('Введите первое число: ')
b = input('Введите второе число: ')
if a < b:
    print(a)
else:
    if a > b: # обратите внимание, здесь одно условие находится вну
    три другого, и код ниже будет писаться после двойного отступа
        print(b)
    else:
        print('Равные числа:', a)
```

Равные числа: 3

Неплохо, но можно упростить это решение с помощью конструкции **else if (или elif)**, которая позволяет в случае ложности условия сразу же написать еще одну проверку.

Вот как будет выглядеть решение нашей задачи с помощью elif:

```
In [23]: a = input('Введите первое число: ')
b = input('Введите второе число: ')
if a < b:
    print(a)
elif a > b:
    print(b)
else:
    print('Равные числа:', a)
```

Введите первое число: 3
Введите второе число: 3
Равные числа: 3

Задачи для тренировки

Распродажа

В магазине проходит акция:

- На все товары дешевле 1000 рублей скидка 15%
- На все товары дороже 1000, но дешевле 5000 рублей скидка 20%
- На все товары дороже 5000 рублей скидка 25%

Ввод

Целое неотрицательное число - цена товара в рублях

Вывод

Целое неотрицательное число - скидка на товар в рублях

```
In [12]: # место для решения
price = int(input())

if price > 5000:
    print(price * 0.25)
elif price > 1000:
    print(price * 0.20)
else:
    print(price * 0.15)
```

149.85

Цикл while

Довольно часто задачи требуют от нас несколько раз выполнить однотипный код.

Если писать несколько раз одни и те же строки, это загромождает программу. Иногда несколько раз превращается в много (100 или 10000). А иногда это число вообще зависит от параметров ввода.

Справиться с этим помогают **циклы**. На этом семинаре мы поработаем с циклом **while (пока)**

Прицип использования цикла while: записываем логическое выражение и некоторый код. Код будет выполняться до тех пор, пока логическое выражение верно.

Например, давайте напечатаем все целые числа от 1 до 10.

```
In [10]: i = 1
while i <= 10:
    print(i)
    i += 1
```

```
1
2
3
4
5
6
7
8
9
10
```

Здесь мы использовали запись $i += 1$. Она эквивалентна $i = i + 1$.

Аналогично можно записывать и другие арифметические операции: например, $-=$

Обратите внимание, что код внутри цикла (тот, который мы хотим повторно выполнять), выделяется отступом.

Операторы **break** и **continue**.

Циклами можно управлять с помощью операторов **break**, **continue**.

Break внутри цикла позволяет прервать его выполнение и сразу же перейти к коду, который идет после цикла (либо завершить программу).

В этом случае мы можем написать сразу после цикла секцию **else** (синтаксис при этом такой же, как и в условиях).

Код, написанный после **else**, будет выполняться, если цикл завершился "естественным путем" (т.е. не был прерван с помощью **break**).

Задача

Рассмотрим пример задачи, которую можно решить с использованием break.

Пусть студент сдал 5 предметов во время сессии и мы хотим узнать, есть ли у него пересдачи

Формат ввода

До пяти оценок от 1 до 10

Формат вывода

Если хотя бы одна из оценок меньше 4, завершаем программу и печатаем 'YES' (пересдачи есть)

Если все пять оценок больше 3, печатаем 'NO' (студент закрыл сессию без пересдач)

```
In [27]: i = 1
while i <= 5:
    note = int(input("Введите оценку: "))
    if note < 4:
        print('YES')
        break
    i += 1
else: # else находится на том же уровне отступа, что и while, поэтому относится именно к циклу, а не к условию внутри цикла
    print ('NO')
```

Введите оценку: 6
 Введите оценку: 7
 Введите оценку: 3
 YES

Оператор **continue** позволяет сразу же перейти на новую итерацию цикла, не выполняя код, который написан внутри цикла ниже его.

Изменим условие задачи - теперь будем считать количество пересдач у студента

```
In [11]: i = 1
retakes = 0
while i <= 5:
    note = int(input("Введите оценку: "))
    i += 1
    if note >= 4: # если пересдачи нет, сразу же идем проверять переменную i, без увеличения переменной retakes
        continue
    retakes += 1
print("Итого пересдач:", retakes)
```

Итого пересдач: 2

Операторами break и continue не стоит злоупотреблять, это может ухудшить читаемость кода.

Например, в предыдущем примере мы бы справились и без continue:

```
In [35]: i = 1
retakes = 0
while i <= 5:
    note = int(input("Введите оценку: "))
    i += 1
    if note < 4:
        retakes += 1
print("Итого пересдач:", retakes)
```

```
Введите оценку: 5
Введите оценку: 6
Введите оценку: 7
Введите оценку: 1
Введите оценку: 7
Итого пересдач: 1
```

(Ω ` -)◦—☆°.*◦° Задача

Вася начал бегать и в первый день он пробежал X километров и выдохся. Вася поставил себе цель Y километров и решил узнать, когда он ее достигнет, если каждый день будет бегать дистанцию на 10% больше, чем в предыдущий.

Формат ввода

Программа получает на вход целые числа X, Y

Формат вывода

Одно целое число (день, когда Вася пробежит свою цель)

Примеры

Ввод:

10

21

Вывод:

9

```
In [13]: x = int(input())
y = int(input())

day_count = 1
while x < y:
    x += x*0.1
    day_count += 1

print(day_count)
```

9

(Π ` -)◦—☆°.*..° Задача

Сложные проценты

Процентная ставка по вкладу составляет Р процентов годовых, которые прибавляются к сумме вклада через год. Вклад составляет X рублей Y копеек. Дробное число копеек по истечении года отбрасывается. Выведите величину вклада в рубл

Формат ввода

Программа получает на вход целые числа Р, Х, Y, K.

Формат вывода

Программа должна вывести два числа: величину вклада через K лет в рублях и копейках. .
Перерасчет суммы вклада (с отбрасыванием дробных частей копеек) происходит ежегодно.

Примеры

Тест 1

Входные данные:

12
179
0
5

Вывод программы:

315 43

Тест 2

Входные данные:

13
179
0
100

Вывод программы:

36360285 50

Тест 3

Входные данные:1
1
0
1000**Вывод программы:**

11881 92

```
In [61]: # (n ` - `)□=☆°.*..°

p = 12
x = 179
y = 0
k = 5

year = 0 # заводим счетчик годов
end_year_amount = x * 100 + y # считаем начальную сумму в копейках

while year < k:
    percent = end_year_amount * p / 100 # считаем процент в этот год
    end_year_amount = int(end_year_amount + percent) # считаем итог в этот год
    year += 1 # Обновляем счетчик года

print(end_year_amount // 100, end_year_amount % 100)
```

315 43

10.04.2020 | МИРЭК | 4 модуль

Автор: Татьяна Рогович

Основы программирования в Python

Лекция 2. Тема 2 ¶

Python Refresher | Срезы. Метод строк find()

Строки, кортежи и списки представляют собой последовательности, а это значит, что мы можем обратиться к любому их элементу по индексу.

Для выполнения такой операции в питоне используются квадратные скобки [] после объекта. В квадратных скобках указывается желаемый индекс. Индексирование начинается с 0.

```
In [1]: s = 'Welcome to "Brasil!"' # заодно обратите внимание на кавычки вН  
утри кавычек (используем разные типы)  
print(s)
```

```
print(s[0]) # первый элемент  
print(s[1]) # второй  
print(s[2]) # третий  
print(s[-1]) # последний  
print(s[-2]) # второй с конца
```

```
#Предыдущие действия никак не изменили строку  
print(s)
```

```
Welcome to "Brasil!"  
W  
e  
l  
"  
!  
Welcome to "Brasil!"
```

Кроме выбора одного элемента с помощью индексирования можно получить подстроку. Для этого надо указать индексы границ подстроки через двоеточие.

Первое число - от какого индекса начинаем (если здесь ничего не написать, то начнем сначала). Второе число (после первого двоеточия) - каким индексом заканчивается срез (если ничего не написать, то питон возьмет последний символ). Третье число (необязательное, после второго двоеточия) - шаг, по умолчанию там стоит 1 (каждая буква).

Таким образом, использовав одно число без двоеточий, мы получим один символ. Использовав два числа через двоеточие - срез строки, включая первый индекс и не включая второй (первое число обязательно меньше второго). Использовав три числа через два двоеточия - срез строки с определенным шагом, заданным третьим числом.

```
In [25]: print(s[1:])
print(s[:4]) # четыре первых символа до порядкового номера 4
print(s[:]) # копия строки
print(s[:-1]) # вся строка кроме последнего символа
print(s[::2]) # также можно выбирать символы из строки с каким-то шагом
print(s[::-1]) # например, с помощью шага -1 можно получить строку наоборот
```

```
elcome to "Brasil!"
Welc
Welcome to "Brasil!"
Welcome to "Brasil!
Wloet Bai!
"!lisarB" ot emocleW
```

По аналогии со строками, у списков и кортежей тоже можно брать срезы.

```
In [1]: myList = [0, 1, 2, 3, 4, 5]
myTuple = (0, 1, 2, 3, 4, 5)
```

```
In [2]: print(myList[1:]) # берем все, начиная с элемента с первым индексом
print(myTuple[2:])
```

```
[1, 2, 3, 4, 5]
(2, 3, 4, 5)
```

```
In [3]: print(myList[:3]) # берем все до элемента с третьим индексом (невключительно)
print(myTuple[:-1])
```

```
[0, 1, 2]
(0, 1, 2, 3, 4)
```

```
In [4]: print(myList[::2]) # берем все четные элементы  
print(myTuple[1::2]) # берем все нечетные элементы  
  
[0, 2, 4]  
(1, 3, 5)
```

Метод .find()

Когда мы работаем со строками, у нас часто стоит задача брать срез не с конкретного индекса, а привязывать его к поиску определенного символа. У нас есть специальный метод строковых переменных .find().

Методы - это методы классы. Грубо говоря, это функции, которые будут работать только с определенным типом данных. Синтаксис метода следующий: {название переменной или данные}.{название метода()}.

Например, давайте проверим содержит ли строка упоминание университета.

```
In [5]: 'В ВШЭ стартовала новая программа по Data Science'.find('ВШЭ')  
Out[5]: 2
```

Метод .find() берет один аргумент - подстроку, которую ищет в строке. Возвращает метод индекс первого символа подстроки, если ее удалось найти.

Если под строка не была найдена, метод вернет -1 (на следующем занятии мы будем использовать это свойство, когда разберемся с условным оператором).

```
In [32]: 'В ВШЭ стартовала новая программа по Data Science'.find('ВШЭ') # обратите внимание, метод чувствителен к регистру  
Out[32]: -1
```

.find() иногда используется в парсинге веб-страниц. Зная индекс первого элемента, мы можем достать интересующую нас информацию.

Например, мы скачали с сайта информацию о цене нового планшета и хотим достать оттуда собственно цену. Мы знаем, что цена идет после подстроки "ЦЕНА:" и что после самой цены идет постфикс "руб.". Давайте попробуем достать цену и посчитать, сколько стоит два таких планшета.

```
In [41]: info = 'iPad 64 GB ЦЕНА: 39 990 руб. Скидка: 5%'  
print(info.find('ЦЕНА:')) # нашли индекс Ц - начала подстроки "ЦЕНА":  
print(info.find('руб.')) # нашли индекс р  
price = info[info.find('ЦЕНА:')+6:info.find('руб.'):1] # вывели срез от от начала до конца цены (с помощью  
# слогаемых б и -1 о ткорректировали индексы до начала и конца цены)  
print(price)
```

```
11  
24  
39 990
```

Почти готово, но теперь мешается пробел. Кстати, это очень частая проблема, что числа в интернете оформлены с разделителями и перед конвертацией их приходится еще и приводить к стандартному виду, который можно скормить функции int(). Пока мы не знаем метода, который может заменять символы, поэтому давайте попробуем почистить цену с помощью .find() и срезов.

```
In [42]: price_clean = price[:price.find(' ')]+price[price.find(' ')+1:]  
print(price_clean)
```

```
39990
```

Теперь с этим можно работать!

```
In [44]: print(int(price_clean) * 2)
```

```
79980
```

Если подстрока входит в строку несколько раз, то find() вернет индекс только для первого вхождения.

```
In [48]: price.find('9')
```

```
Out[48]: 3
```

Есть модификация метода find(): rfind(substring) - возвращает позицию самого правого вхождения подстроки substring в строку string или -1, если подстрока не найдена.

```
In [47]: price.rfind('9')
```

```
Out[47]: 4
```

10.04.2020 | МИРЭК | 4 модуль

Автор: Татьяна Рогович

Основы программирования в Python 1

Лекция 2. Тема 3

Python Refresher | Множества и словари

Множества (set)

Мы уже знаем списки и кортежи - упорядоченные структуры, которые могут хранить в себе объекты любых типов, к которым мы можем обратиться по индексу. Теперь поговорим о структурах неупорядоченных - множествах и словарях.

Множества хранят некоторое количество объектов, но, в отличие от списка, один объект может храниться в множестве не более одного раза. Кроме того, порядок элементов множества произволен, им нельзя управлять.

Тип называется `set`, это же является конструктором типа, т.е. в функцию `set` можно передать произвольную последовательность, и из этой последовательности будет построено множество:

```
In [1]: print(set([10, 20, 30])) # передаем список
print(set((4, 5, 6))) # передаем tuple
print(set(range(10)))
print(set()) # пустое множество
```

```
{10, 20, 30}
{4, 5, 6}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
set()
```

Другой способ создать множество - это перечислить его элементы в фигурных скобках (список - в квадратных, кортеж в круглых, а множество - в фигурных)

```
In [2]: primes = {2, 3, 5, 7}
animals = {"cat", "dog", "horse", 'cat'}
print(primes)
print(animals)

{2, 3, 5, 7}
{'horse', 'dog', 'cat'}
```

Кстати, обратите внимание, что множество может состоять только из уникальных объектов. Выше множество animals включает в себя только одну кошку несмотря на то, что в конструктор мы передали 'cat' два раза. Преобразовать в список в множество - самый простой способ узнать количество уникальных объектов.

Со множествами работает почти всё, что работает с последовательностями (но не работают индексы, потому что элементы не хранятся упорядоченно).

```
In [3]: print(len(primes)) # длина
print(11 in primes) # проверка на наличие элемента in хорошо и быстро
# работает для множеств
print("cow" in animals)

4
False
False
```

Все возможные операции с множествами: <https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset> (<https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset>)

Отдельно мы посмотрим на так называемые операции над множествами. Если вы знаете круги Эйлера, то помните как различают объекты множеств - пересечение, объекты, которые принадлежат множеству a, но не принадлежат b и так далее. Давайте посмотрим, как эти операции реализованы в питоне.

```
In [4]: a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
c = {2, 3}

print(c <= a) # проверка на подмножество (с подмножество a)
print(c <= b) # не подмножество, т.к. в b нет 2
print(a >= c)
print(a | b) # объединение
print(a & b) # пересечение
print(a - b) # разность множеств (все что в a, кроме b)
print(a ^ b) # симметрическая разность множеств (объединение без пересечения)

c = a.copy() # копирование множества, или set(a)
print(c)
```

```
True
False
True
{1, 2, 3, 4, 5, 6}
{3, 4}
{1, 2}
{1, 2, 5, 6}
{1, 2, 3, 4}
```

Предыдущие операции не меняли множества, создавали новые. А как менять множество:

```
In [5]: s = {1, 2, 3}
s.add(10) # добавить
print(s) # обратите внимание, что порядок элементов непредсказуем
s.remove(1) # удаление элемента
s.discard(1) # аналогично, но не будет ошибки, если вдруг такого элемента нет в множестве
print(s)
x = s.pop() # удаляет и возвращает один произвольный элемент множества (можем сохранить его в переменную)
print(s)
print(x)
s.clear() # очистить
print(s)

{10, 1, 2, 3}
{10, 2, 3}
{2, 3}
10
set()
```

Как мы сокращали арифметические операции раньше (например, `+=`), так же можно сокращать операции над множествами.

```
In [6]: s |= {10, 20} # s = s | {10, 20} # объединение множества s с {10, 20}
print(s)
# s ^=, s &= и т.п.
```

{10, 20}

Словари (dict)

Обычный массив (в питоне это список) можно понимать как функцию, которая сопоставляет начальному отрезку натурального ряда какие-то значения.

Давайте посмотрим на списки непривычным способом. Списки - это функции (отображения), которые отображают начальный ряд натуральных чисел в объекты (проще говоря - преводят число 0,1,2,3... во что-то):

```
In [7]: l = [10, 20, 30, 'a']
print(l[0])
print(l[1])
print(l[2])
print(l[3])
```

10
20
30
a

В словарях отображать можно не только начала натурального ряда, а произвольные объекты. Представьте себе настоящий словарь или телефонную книжку. Имени человека соответствует номер телефона.

Классическое использование словарей в анализе данных: хранить частоту слова в тексте.

кот → 10

и → 100

Тейлора → 2

Словарь состоит из набора ключей и соответствующих им значений. Значения могут быть любыми объектами (также как и в списке, хранить можно произвольные объекты). А ключи могут быть почти любыми объектами, но только неизменяемыми. В частности числами, строками, кортежами. Список или множество не могут быть ключом.

Одному ключу соответствует ровно одно значение. Но одно и то же значение, в принципе, можно сопоставить разным ключам.

```
In [8]: a = dict()
a[(2,3)] = [2,3] # кортеж может быть ключом, потому что он неизменя
Мый
a
```

```
Out[8]: {(2, 3): [2, 3]}
```

```
In [9]: b = dict()
b[[2,3]] = [2,3] # а список уже нет, получим ошибку
print(b)
```

```
TypeError                                     Traceback (most recent c
all last)
<ipython-input-9-c056762b57aa> in <module>()
    1 b = dict()
----> 2 b[[2,3]] = [2,3] # а список уже нет, получим ошибку
    3 print(b)

TypeError: unhashable type: 'list'
```

Создание словаря

В фигурных скобках (как множество), через двоеточие ключ:значение

```
In [10]: d1 = {"кот": 10, "и": 100, "Тейлора": 2}
print(d1)
```

```
{'кот': 10, 'и': 100, 'Тейлора': 2}
```

Через функцию dict(). Обратите внимание, что тогда ключ-значение задаются не через двоеточие, а через знак присваивания. А строковые ключи пишем без кавычек - по сути мы создаем переменные с такими названиями и присваиваем им значения (а потом функция dict() уже превратит их в строки).

```
In [11]: d2 = dict(кот=10, и=100, Тейлора=2)
print(d2) # получили тот же результат, что выше

{'кот': 10, 'и': 100, 'Тейлора': 2}
```

И третий способ - передаем функции dict() список списков или кортежей с парами ключ-значение.

```
In [12]: d3 = dict([('кот', 10), ('и', 100), ('Тейлора', 2)]) # перечисление
          (например, список) tuple
print(d3)

{'кот': 10, 'и': 100, 'Тейлора': 2}
```

Помните, когда мы говорили про списки, мы обсуждали проблему того, что важно создавать именно копию объекта, чтобы сохранять исходный список. Копию словаря можно сделать так

```
In [13]: d4 = dict(d3) # фактически, копируем dict который строчкой выше
print(d4)

{'кот': 10, 'и': 100, 'Тейлора': 2}
```

```
In [14]: d1 == d2 == d3 == d4 # Содержание всех словарей одинаковое
```

```
Out[14]: True
```

Пустой словарь можно создать двумя способами.

```
In [15]: d2 = {} # это пустой словарь (но не пустое множество)
d4 = dict()
print(d2, d4)

{} {}
```

Операции со словарями

Как мы уже говорили, словари неупорядоченные структуры и обратиться по индексу к объекту уже больше не удастся.

```
In [16]: d1[1] # выдаст ошибку во всех случаях кроме того, если в вашем словаре вдруг есть ключ 1
```

```
-----  
-----  
KeyError Traceback (most recent call last)  
<ipython-input-16-627b04325301> in <module>()  
----> 1 d1[1] # выдаст ошибку во всех случаях кроме того, если в вашем словаре вдруг есть ключ 1  
  
KeyError: 1
```

Но можно обращаться к значению по ключу.

```
In [17]: print(d1['КОТ'])
```

```
10
```

Можно создать новую пару ключ-значение. Для этого просто указываем в квадратных скобках название нового ключа.

```
In [18]: d1[1] = 'test'  
print(d1[1]) # теперь работает!
```

```
test
```

Внимание: если элемент с указанным ключом уже существует, новый с таким же ключом не добавится! Ключ – это уникальный идентификатор элемента. Если мы добавим в словарь новый элемент с уже существующим ключом, мы просто изменим старый – словари являются изменяемыми объектами.

```
In [19]: d1["КОТ"] = 11 # так же как в списке по индексу – можно присвоить новое значение по ключу  
print(d1['КОТ'])  
d1["КОТ"] += 1 # или даже изменить его за счет арифметической операции  
print(d1['КОТ'])
```

```
11  
12
```

А вот одинаковые значения в словаре могут быть.

```
In [20]: d1['собака'] = 12
print(d1)

{'кот': 12, 'и': 100, 'Тейлора': 2, 1: 'test', 'собака': 12}
```

Кроме обращения по ключу, можно достать значение с помощью метода `.get()`. Отличие работы метода в том, что если ключа еще нет в словаре, он не генерирует ошибку, а возвращает объект типа `None` ("ничего"). Это очень полезно в решении некоторых задач.

```
In [21]: print(d1.get("кот")) # вернул значение 11
print(d1.get("ктоо")) # вернут None
print(d1['ктоо']) # ошибка
```

```
12
None

-----
-----
KeyError                                     Traceback (most recent call last)
KeyError: 'ктоо'
```

Удобство метода `.get()` заключается в том, что мы сами можем установить, какое значение будет возвращено, в случае, если пары с выбранным ключом нет в словаре. Так, вместо `None` мы можем вернуть строку `Not found`, и ломаться ничего не будет:

```
In [22]: print(d1.get("ктоо", 'Not found')) # передаем вторым аргументом, что
          о возвращать
print(d1.get("ктоо", False)) # передаем вторым аргументом, что возврашать
```

```
Not found
False
```

Также со словарями работают уже знакомые нам операции - проверка количества элементов, проверка на наличие объектов.

```
In [23]: print(d1)
print("КОТ" in d1) # проверка на наличие ключа
print("КТОО" not in d1) # проверка на отсутствие ключа

{'кот': 12, 'и': 100, 'Тейлора': 2, 1: 'test', 'собака': 12}
True
True
```

Удалить отдельный ключ или же очистить весь словарь можно специальными операциями.

```
In [24]: del d1["КОТ"] # удалить ключ со своим значением
print(d1)
d1.clear() # удалить все
print(d1)

{'и': 100, 'Тейлора': 2, 1: 'test', 'собака': 12}
{}
```

У словарей есть три метода, с помощью которых мы можем сгенерировать список только ключей, только значений и список пар ключ-значения (на самом деле там несколько другая структура, но ведет себя она очень похоже на список).

```
In [25]: print(d1.values()) # только значения
print(d1.keys()) # только ключи
print(d1.items()) # только значения

dict_values([])
dict_keys([])
dict_items([])
```

Ну, и раз уж питоновские словари так похожи на обычные, давайте представим, что у нас есть словарь, где все слова многозначные. Ключом будет слово, а значением - целый список.

```
In [26]: my_dict = {'swear' : ['клясться', 'ругаться'], 'dream' : ['спать', 'мечтать']}
```

По ключу мы получим значение в виде списка:

```
In [27]: my_dict['swear']

Out[27]: ['клясться', 'ругаться']
```

Так как значением является список, можем отдельно обращаться к его элементам:

```
In [28]: my_dict['swear'][0] # первый элемент
```

```
Out[28]: 'клясться'
```

Можем пойти дальше и создать словарь, где значениями являются словари! Например, представим, что в некотором сообществе проходят выборы, и каждый участник может проголосовать за любое число кандидатов. Данные сохраняются в виде словаря, где ключами являются имена пользователей, а значениями – пары *кандидат-голос*.

```
In [29]: votes = {'user1': {'cand1': '+', 'cand2': '-'},
               'user2': {'cand1': 0, 'cand3': '+'}} # '+' - за, '-' - против, 0 - воздержался
```

```
In [30]: votes
```

```
Out[30]: {'user1': {'cand1': '+', 'cand2': '-'}, 'user2': {'cand1': 0, 'cand3': '+'}}
```

По аналогии с вложенными списками по ключам мы сможем обратиться к значению в словаре, который сам является значением в `votes` (да, эту фразу нужно осмыслить):

```
In [31]: votes['user1']['cand1'] # берем значение, соответствующее ключу user1, в нем - ключу cand1
```

```
Out[31]: '+'
```

"Объединять словари можно через метод `update()`. Обратите внимание, что если в словарях есть одинаковые ключи, то они перезапишутся ключами того словаря, который добавляем."

```
In [32]: votes.update(my_dict)
votes
```

```
Out[32]: {'user1': {'cand1': '+', 'cand2': '-'},
           'user2': {'cand1': 0, 'cand3': '+'},
           'swear': ['клясться', 'ругаться'],
           'dream': ['спать', 'мечтать']}
```

```
In [33]: votes['swear'] = 1 # добавим в словарь votes ключ swear, чтобы проверить, что произойдет, когда объединим с my_dict, в котором есть такой ключ\n,
votes.update(my_dict)
print(votes) # swear в votes перезависался swear из mydict
```

```
{'user1': {'cand1': '+', 'cand2': '-'}, 'user2': {'cand1': 0, 'cand3': '+'}, 'swear': ['клясться', 'ругаться'], 'dream': ['спать', 'мечтать']}
```

Через занятие мы с вами начнем работать с условными операторами и циклом for, и тогда мы поговорим об особенностях обращения к значениям в цикле и сортировок словарей.

10.04.2020 | МИРЭК | 4 модуль

Автор: Татьяна Рогович

Основы программирования в Python

Лекция 2. Тема 4 1

Python Refresher | Методы строк

Методы строк позволяют нам обрабатывать текст и осуществлять поиск по условиям. В этом блокноте познакомимся с самыми популярными методами, которые часто пригождаются в работе.

Например news.upper() - метод upper() вызывается от строковой переменной news. По сути методы, это функции, которые применимы только к особому типу данных. Так, например, функция print() напечатает все, что бы мы ей не передали, а перевод к верхнему регистру (а именно это делает метод upper()) ни с одним типом данных кроме строки уже не сработает.

```
In [1]: news = 'Samsung releases new device'
print(news.upper()) # приводит строку к верхнему регистру
print(news.lower()) # приводит строку к нижнему регистру
```

```
SAMSUNG RELEASES NEW DEVICE
samsung releases new device
```

Обратите внимание, что строковый метод, как правило, не изменяет объект и наша строка осталась такой как была.

```
In [2]: news
```

```
Out[2]: 'Samsung releases new device'
```

У строк есть множество методов, которые позволяют искать паттерны и как-то их редактировать. Методы **startswith()** и **endswith()** проверяют, стоит ли искомая построка в начале или в конце строки.

Пусть у нас есть отзыв посетителя о кафе. Мы заранее знаем, что пользователь для отзыва выбирает только последнее слово из предложенных двух: 'good', 'bad'. Попробуем оценить, остался ли доволен клиент.

```
In [3]: feedback = 'This place was bad.' # сам отзыв

if feedback.endswith('bad.'): # если строка заканчивается на 'bad'
    print('Client was disappointed') # то клиент расстроен
else:
    print('Client was satisfied') # иначе – клиенту все понравилось

Client was disappointed
```

Теперь усложним задачу. Что если слово находится не в конце предложения? Попробуем его найти!

```
In [4]: feedback2 = 'This place was bad enough'

if feedback2.find('bad') != -1:
    print('Client was disappointed') # то клиент расстроен
else:
    print('Client was satisfied') # иначе – клиенту все понравилось

Client was disappointed
```

Метод **strip()** (и его собратья **lstrip()** и **rstrip()**), работающие только с одной стороны строки, удаляет незначимые символы (пробелы, табуляцию и т.д.) с краев строк. Очень полезный метод, когда мы собираем информацию из интернета. Если этим методам передать аргумент, то они удалят подстроку.

```
In [5]: print(' 135133      '.strip()) # удалили пробельные символы слева и справа
print('ruhse.ru'.strip('ru')) # удалили ru с обеих сторон
print('ruhse.ru'.lstrip('ru')) # удалили ru слева
print('ruhse.ru'.rstrip('ru')) # удалили ru справа
```

```
135133
hse.
hse.ru
ruhse.
```

В реальности **strip** часто используется для нормализации email'ов или других идентификаторов

```
In [6]: print('pileyan@gmail.com'.strip())
pileyan@gmail.com
```

Также мы можем заменять символы в строке, с помощью метода `.replace()`. `replace()` берет два аргумента - что заменяем и на что заменяем. Есть еще optionalный третий аргумент - количество замен, которые нужно сделать.

```
In [7]: print('39 000'.replace('0', '9')) # заменяем все нули на девятки
print('39 000'.replace('0', '9', 1)) # заменяем только первый найденный нуль
```

```
39 999
39 900
```

Теперь научимся считать количество вхождений подстроки в строку с помощью метода `count()`

```
In [8]: s = "Mushroooom soup" # исходная строка
print(s.count("O")) # ищем заглавную букву O, не находим
print(s.count("o")) # ищем строчную букву o, находим 5 штук
print(s.count("oo")) # ищем две буквы o подряд, находим две таких подстроки
print(s.count("ooo")) # ищем три букв o подряд, находим одно такое вхождение
print(s.count("push")) # ищем подстроку 'push', не находим
print(s.count("o", 4, 7)) # ищем букву o в s[4:7]
print(s.count("o", 7)) # ищем букву o в s[7:]
```

```
0
5
2
1
0
2
3
```

Отдельное семейство методов строк отвечает за проверку на соответствие условиям.

```
In [9]: # isalpha - проверяет, что все символы строки являются буквами.
print('Ask me a question!'.isalpha())
print('Ask'.isalpha())

# isdigit - проверяет, что все символы строки являются цифрами.
print('13242'.isdigit())

# isalnum - проверяет, что все символы строки являются буквами или
цифрами.
print('Ask me a question!'.isalnum())
print('Ask232'.isalnum())

# islower - проверяет, что все символы строки являются маленькими (строчными) буквами.
print('ask me a question!'.islower())

# isupper - проверяет, что все символы строки являются большими (за
главными, прописными) буквами.
print('Ask me a question!'.isupper())
```

```
False
True
True
False
True
True
False
```

На этом методы строк не заканчиваются) Например, можно менять регистр букв или переводить в верхний регистр первые буквы слов

```
In [10]: # title - переводит первую букву всех слов к верхнему регистру
print('ask me a question!'.title())

# swapcase - меняет регистр на противоположный
print('ask me a question!'.swapcase())
```

```
Ask Me A Question!
ASK ME A QUESTION!
```

Может возникнуть вопрос: а зачем нам проверять, из каких символов состоит строка? Ведь даже, если строка состоит из цифр, числом она автоматически не станет. Давайте рассмотрим две ситуации, в которых очень полезно знать, какие символы входят в нашу строку.

Ситуация

Пользователь должен придумать пароль для своей учетной записи. Пароль должен состоять только из цифр и букв.

```
In [ ]: password = input("Введите пароль: ")  
  
if password.isalnum() == False:  
    print("Пароль должен состоять только из букв и цифр!")
```

А теперь давайте усложним и добавим проверку на то, что хотя бы одна буква должна быть заглавная.

```
In [11]: password = input("Введите пароль: ")  
  
if password.isalnum() == False:  
    print("Пароль должен состоять только из букв и цифр!")  
if password.islower() == True:  
    print("Пароль должен содержать как минимум одну заглавную букву!" )
```

Пароль должен содержать как минимум одну заглавную букву!

(∩｀-’efined—☆°.*°.° Задачи

Пароль должен состоять ровно из 10 символов и состоять только из букв. Вам даны два пароля, напишите программу, которая проверит их на правильность и выведет True для правильного пароля и False для неправильного.

```
In [12]: pass1 = 'qweABCjoHn'  
pass2 = 'asdfPassword1'  
  
print(len(pass1) == 10 and pass1.isalpha())  
print(len(pass2) == 10 and pass2.isalpha())
```


True
False

10.04.2020 | МИРЭК | 4 модуль

Автор: Валентина Лебедева

Основы программирования в Python

Лекция 2. Тема 5 1

Python Refresher | Функция Range. Цикл for

Функция range

Давайте решим задачу - на вход подается число N, нужно сгенерировать список от 1 до N.

```
In [53]: i = 1
myList = []
N = int(input())

while i <= N:
    myList.append(i)
    i += 1

print(myList)
```

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

Неплохо, но python предлагает нам готовое решение - встроенную функцию range().

range(N) возвращает диапазон чисел от 0 до N - 1.

```
In [55]: myRange = range(10)
print(myRange[0], myRange[8], myRange[-1])
```

0 8 9

Есть одно но - range возвращает не список и не кортеж, это становится понятно, если попробовать напечатать результат.

```
In [56]: print(myRange)
print(type(myRange))
```

```
range(0, 10)
<class 'range'>
```

Выручает старое доброе преобразование типов.

```
In [57]: print(list(myRange))
print(tuple(myRange))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Функция range ведет себя очень похоже на срезы.

range(N, M) вернет диапазон от N до M - 1.

range(N, M, i) вернет диапазон от N до M - 1 с шагом i.

```
In [60]: print(list(range(1, 5)))
print(list(range(1, 10, 2)))
print(list(range(-5, -1)))
print(list(range(-5, -10, -2)))
```

```
[1, 2, 3, 4]
[1, 3, 5, 7, 9]
[-5, -4, -3, -2]
[-5, -7, -9]
```

Цикл for

Мы уже умеем писать циклы с помощью оператора while, но теперь мы готовы познакомиться с еще одним способом.

Давайте вспомним про наш список студентов и выведем их имена по очереди. Для начала уже знакомым способом.

```
In [62]: students = ['Ivan Ivanov', 'Tatiana Sidorova', 'Maria Smirnova']

i = 0
while i < len(students):
    print(students[i])
    i += 1
```

```
Ivan Ivanov
Tatiana Sidorova
Maria Smirnova
```

А теперь напишем то же самое с помощью цикла for.

Теперь мы будем перебирать список students, доставая из него по одному элементу.

```
In [63]: students = ['Ivan Ivanov', 'Tatiana Sidorova', 'Maria Smirnova']

for student in students:
    print(student)
```

```
Ivan Ivanov
Tatiana Sidorova
Maria Smirnova
```

В примере выше на каждом шаге цикл for достает очередной элемент из списка students и сохраняет его в переменную student.

Обратите внимание, что переменную student нам не надо создавать заранее, она создается прямо во время работы цикла.

Получившийся код гораздо короче и проще варианта с использованием while.

Цикл for может перебирать и элементы range, это дает нам еще один способ решения задачи.

```
In [65]: students = ['Ivan Ivanov', 'Tatiana Sidorova', 'Maria Smirnova']

for i in range(len(students)): # здесь не надо преобразовывать range в список или кортеж
    print(students[i])
```

```
Ivan Ivanov
Tatiana Sidorova
Maria Smirnova
```

Цикл for может перебирать и кортежи, и даже строки.

```
In [66]: myStr = 'Hello'

for char in myStr:
    print(char)
```

```
Н
е
л
л
о
```

Задачи для тренировки

Часть из этих задач мы решим в классе. Но если мы даже не успеем - попытайтесь сделать их дома сами.

Повышенная стипендия

Студент может подать документы на повышенную стипендию, если его средний балл по итогам сессии не ниже 8.

Формат ввода

Вводятся оценки студента за сессию (целые числа от 1 до 10) в одну строку через пробел

Формат вывода

Выведите YES, если студент может подать документы на повышенную стипендию, и NO, если нет.

Примеры

Тест 1

Входные данные:

6 8 7 6

Вывод программы:

NO

```
In [ ]: # решение здесь
```

Несовершеннолетние студенты

Студсовет факультета организует вечеринку, но после 23 часов им нужно отвезти в общежитие всех несовершеннолетних студентов. Несовершеннолетними считаются все студенты, родившиеся после 2001 года.

Формат ввода

Вводится целое неотрицательное число $N \leq 1000$

Далее, вводится N строк формата "Имя Фамилия; Год рождения"

Формат вывода

Вывести список всех несовершеннолетних студентов

Примеры

Тест 1

Входные данные:

5

Ivan Ivanov; 2001

Petr Petrov; 2002

Maria Smirnova; 2002

Makar Makarov; 2002

Tatiana Kuznetsova; 2000

Вывод программы:

Petr Petrov

Maria Smirnova

Makar Makarov

In [1]: # решение здесь

10.04.2020 | МИРЭК | 4 модуль

Автор: Валентина Лебедева

Основы программирования в Python

Лекция 2. Тема 6

Python Refresher | Функции

Функции

Начиная с первого семинара, мы активно используем встроенные функции python (такие, как `print`, `int` и другие).

Настало время научиться создавать свои.

Когда нам может понадобиться создать функцию?

- Когда какой-то блок кода потребуется переиспользовать (и не подряд, как в случае цикла, а из разных мест программы)
- Когда код становится очень большим, мы можем какие-то его части оформлять в виде функций, чтобы улучшить его читаемость

Давайте научимся создавать функции на примере функции для подсчета факториала числа.

Напомним, что факториал целого числа n - это произведение целых чисел от 1 до n :

$$4! = 1 \cdot 2 \cdot 3 \cdot 4$$

Сначала решим эту задачу без создания функции.

```
In [82]: n = int(input('Number here: '))
fact = 1 # 0! и 1! равны 1, так что наше стартовое значение факториала будет равно 1
for i in range(2, n + 1):
    fact *= i
print(fact)
```

Number here: 4
24

Теперь вынесем подсчет факториала в функцию:

```
In [83]: def factorial(n): # factorial - название функции, n - ее аргумент, параметр, от которого зависит результат
    fact = 1
    for i in range(2, n + 1):
        fact *= i
    return fact # когда результат получен, его надо вернуть с помощью ключевого слова return
```

Теперь мы можем вызывать нашу функцию и пользоваться ей, как если бы она была встроенной.

```
In [85]: n = int(input('Number here: '))
print(factorial(n))
```

Number here: 5
120

Функции могут иметь много аргументов/параметров. В качестве аргументов могут выступать:

- константные значения
- переменные
- результаты вычисления выражений и исполнения других функций

Команда `return` завершает исполнение функции, возвращая ее значение.

Можно прописать возвращение нескольких значений в разных местах. Например, вернемся снова к поиску минимума на двух числах:

```
In [86]: def min1(a, b):
    if a < b:
        return a
    else:
        return b
```

```
In [87]: a = int(input())
b = int(input())
print(min1(a, b))
```

```
3
5
3
```

Теперь с использованием этой функции можно написать поиск минимума на трех числах.

```
In [88]: def min2(a, b, c):
    return min1(a, min1(b, c))
```

```
In [89]: a = int(input())
b = int(input())
c = int(input())
print(min2(a, b, c))
```

```
4
3
5
3
```

Можно вернуть и несколько значений в одном месте. Например, отсортируем два числа в порядке возрастания:

```
In [90]: def sort2(a, b):
    if a < b:
        return a, b
    else:
        return b, a
```

```
In [93]: a = int(input())
b = int(input())
print(sort2(a, b))
```

```
5
3
(3, 5)
```

Глобальные и локальные переменные

Если переменная задается вне тела функции - это **глобальная переменная**, она "видна" и может использоваться в любом месте программы.

Если переменная задается внутри тела функции - это **локальная переменная**, которая существует только внутри функции.

```
In [108]: def f():
    x = 1

    f()
    print(x) # программа видит переменную x только внутри функции
```

```
-----
NameError                                 Traceback (most recent call last)
<ipython-input-108-39cacd033cd4> in <module>
      4
      5 f()
----> 6 print(x) # программа видит переменную x только внутри функции

NameError: name 'x' is not defined
```

Переменную, созданную внутри функции, можно объявить как глобальную с помощью ключевого слова `global` (лучше этим не злоупотреблять).

```
In [109]: def f():
    global x # объявляем переменную x - глобальной, она сохранится
              и вне функции.
    x = 1

    f()
    print(x)
```

1

Распаковка и оператор * (раздел повышенного уровня сложности)

В примере выше мы увидели, что при возвращении из функции нескольких значений эти значения оборачиваются в кортеж.

Мы можем "распаковать" этот кортеж и извлечь эти значения в отдельные переменные.

```
In [94]: minimum, maximum = sort2(a, b)
print(minimum, maximum)
```

```
3 5
```

Распаковка работает не только с результатами вызова функции, но и с любыми кортежами и списками.

```
In [95]: first, second, third = (1, 2, 3)
print(first, second, third)
```

```
1 2 3
```

```
In [96]: first, second, third = [4, 5, 6]
print(first, second, third)
```

```
4 5 6
```

Мы можем извлечь из списка или кортежа несколько переменных, а остаток сохранить в новый список с помощью оператора *.

```
In [101]: first, second, *newList = list(range(10))
print(first, second, newList)
```

```
0 1 [2, 3, 4, 5, 6, 7, 8, 9]
```

Что делает это оператор *? Он "раскрывает" список newList и заставляет программу видеть его так, будто это не список, а просто все его элементы, записанные через запятую.

Таким образом мы можем выводить элементы списка через пробел без использования .join() и map().

```
In [103]: print(*list(range(10)))
```

```
0 1 2 3 4 5 6 7 8 9
```

Еще один интересный эффект оператора * - возможность создания функций с неограниченным числом параметров.

В качестве примера можно рассмотреть функцию, которая суммирует свои аргументы.

```
In [104]: def mySum(*args):  
    numSum = 0  
    for number in args:  
        numSum += number  
    return numSum
```

```
In [105]: print(mySum(1, 2))  
print(mySum(1, 2, 3, 4))
```

3

10

Задачи для тренировки

Часть из этих задач мы решим в классе. Но если мы даже не успеем - попытайтесь сделать их дома сами.

Округление списка чисел

На вход подается список чисел с плавающей точкой через пробел. Верните список, содержащий результаты округления этих чисел.

Решите задание двумя способами - через цикл for или через функцию map.

Вариант полегче

Решите задачу с помощью встроенной функции round

Вариант посложнее

Решите задачу, написав свою функцию, округляющую числа по российским правилам (дробная часть, равная 0.5, округляется вверх)

Суммы цифр

На вход подается список целых чисел. Верните список, содержащий суммы цифр этих чисел.
Решите задание двумя способами - через цикл `for` или через функцию `map`.

Пример ввода

765 98 7 5555 13 **

In [1]: *# решение здесь*

10.04.2020 | МИРЭК | 4 модуль

Автор: Татьяна Рогович

Основы программирования в Python

Лекция 2. Тема 7

Python Refresher | Чтение файлов [1](#)

Файловый ввод-вывод

Мы начинаем работать с файлами. Сейчас будем обсуждать только чтение и запись. О том, как запускать файлы на исполнение, отдельная история. Также для начала речь пойдёт о текстовых файлах или похожих на текстовые (например, код на Python или CSV-файл будет текстовым).

Как правило, если указать в Python не полный путь к файлу, а только его название, то он будет искать файл в рабочей директории. Как узнать, где это?

```
In [2]: import os  
os.getcwd()
```

```
Out[2]: 'C:\\@Rogovich\\@PythonData\\2019_2020_HSE\\2020_DPO_PythonProg\\6  
_Sorting_Files'
```

Функция `getcwd()` из модуля `os` возвращает нам путь к вашей рабочей папке. Так, например, в Windows по умолчанию Anaconda делает рабочей папкой для Jupyter папку пользователя в `Users`. если вы создали блокнот в какой-то другой папке - по умолчанию его директория и будет рабочей. Это можно изменить или глобально, прописав путь к вашей папке в свойствах, или локально в рамках сессии.

Функция `chdir()` принимает в качестве аргумента путь к папке и меняет рабочую директорию. Теперь к файлам, хранящимся в ней вы сможете обращаться без полного пути. Также все новые файлы будут сохраняться туда же.

```
In [3]: os.chdir('C:\\@Rogovich\\@PythonData\\2019_2020_HSE\\2020_DPO_Pytho  
nProg\\6_Sorting_Files') # здесь нужно вставить путь к папке, с кот  
орой хотите работать
```

```
Out[3]: 'C:\\@Rogovich\\@PythonData\\2019_2020_HSE\\2020_DPO_PythonProg\\6  
_Sorting_Files'
```

.listdir() вернет нам список содержимого директории. Очень полезная функция – можно запустить цикл, если нужно обработать все файлы в папке.

```
In [4]: os.listdir()
```

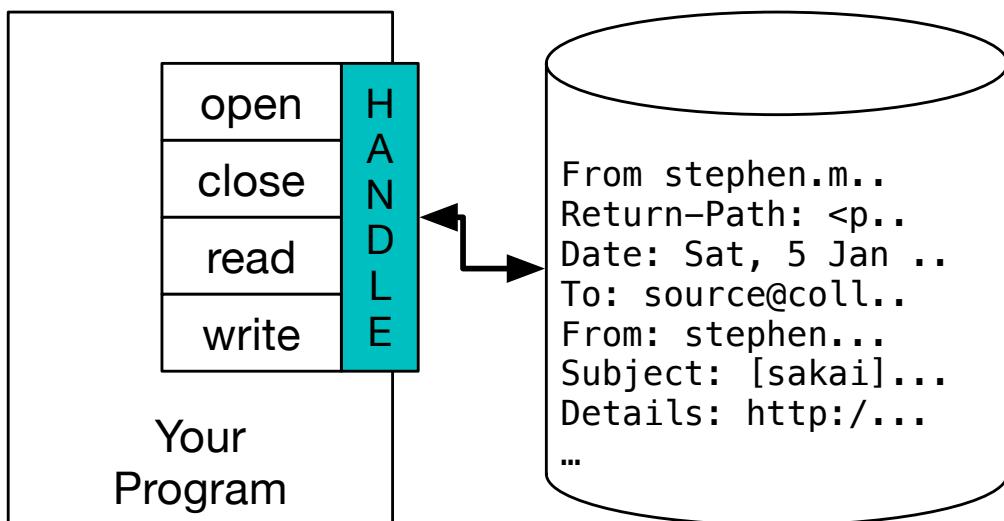
```
Out[4]: ['.ipynb_checkpoints',
          '2020_DPO_6_1_O_notation_sorting.ipynb',
          '2020_DPO_6_2_Files.ipynb',
          '2020_DPO_6_3_Html_Intro.ipynb',
          '2020_DPO_6_4_BS_Tables.ipynb',
          'html11.png',
          'html12.png',
          'html13.png',
          'html14.png',
          'html15.png',
          'mbox.txt',
          'nuclear.csv',
          'simple_table.html',
          'table.csv',
          'test.txt']
```

Давайте попробуем создать файл, записать в него что-нибудь и сохранить.

```
In [13]: f = open("test.txt", 'w', encoding='utf8')
```

Функция `open()` возвращает файловый объект и мы используем ее обычно с двумя аргументами - имя файла и режим (например, запись или чтение). Выше мы открыли файл `test.txt` в режиме записи '`w`' (если такого файла не существовало, он будет создан).

Такой объект называется `file handle` или дескриптор файла.



Source: <https://www.py4e.com/html3/07-files> (<https://www.py4e.com/html3/07-files>)

Какие могут быть режимы открытия файла (mode):

- '`r`' - read, только чтение
- '`w`' - write, только запись (если файл с таким именем существовал, он будет удален).
- '`a`' - append, новые данные будут записаны в конец файла
- '`r+`' - чтение+запись.

Если не передать второй аргумент, то файл автоматически открывается в режиме чтения.

Encoding - именнованный параметр, если работаете с кириллицей или языками со спецсимволами, то лучше задать `utf8`.

```
In [14]: f.write('Hello, world!')
f.close()
```

Метод `write` записал данные в наш файл. После этого файл нужно закрыть, чтобы он выгрузился из оперативной памяти. Если этого не сделать, то в какой-то момент питоновский сборщик мусора все равно до него доберется и закроет файл, но большие файлы могут съедать достаточно много ресурсов, поэтому лучше за этим следить.

Теперь дававайте попробуем открыть в режиме чтения.

```
In [15]: f = open("test.txt", 'r', encoding='utf8')
print(f.read())
f.close()
```

Hello, world!

После того, как мы закрыли файл, обратиться к нему больше нельзя.

```
In [16]: f.read()
```

```
-----
-----
ValueError                                Traceback (most recent call last)
<ipython-input-16-571e9fb02258> in <module>()
      1 f.read()

ValueError: I/O operation on closed file.
```

Хорошим тоном при работе с файлами считается открывать их с помощью ключевого слова `with`. Преимущество этого способа в том, что файл закроется автоматически, когда закончатся вложенные операции.

```
In [17]: with open('test.txt') as f: # открыли файл, не указали режим, по умолчанию - чтение
    read_data = f.read() # считали данные из файла в переменную
    # операции закончились, файл сам закрылся
```

```
In [18]: print(read_data)
```

Hello, world!

А теперь попробуем записать в файл новые строки.

```
In [20]: with open('test.txt', 'a') as f: # открыли файл
    f.write('\n Is this an african swallow?\n Or an european swallow?') # дозаписали строки в файл
```

Еще один вариант записать данные в файл вот так:

```
In [21]: with open('test.txt', 'a') as f:
    print("\nAnd another string", file = f)
```

```
In [22]: with open('test.txt') as f:  
    print(f.read())
```

```
Hello, world!  
Is this an african swallow?  
Or an european swallow?  
Is this an african swallow?  
Or an european swallow?  
And another string
```

Мы выше уже видели два метода файла .write() и .read(). Еще один метод, который очень часто используется - это readline. Он позволяет не загружать файл целиком в память, а считывать его построчно. Знаком остановки здесь будет выступать \n

```
In [23]: f = open('test.txt', 'r')  
f.readline()
```

```
Out[23]: 'Hello, world!\n'
```

```
In [24]: f.readline()
```

```
Out[24]: ' Is this an african swallow?\n'
```

.readline() - генератор. При обращении он выдает нам новую строку.

```
In [25]: f.readline() # вызвали третий раз
```

```
Out[25]: ' Or an european swallow?\n'
```

```
In [26]: f.close()
```

Также, чтобы прочитать все строки поочередно, можно запустить цикл. Тут не стоит забывать, что переменная f, хоть и прикапывается списком строк, когда мы её итерируем, на самом деле таковым не является. В действительности при открытии файла мы запоминаем позицию, на которой мы этот файл читаем. Изначально она указывает на самое начало файла, но с каждой итерацией сдвигается. Когда мы прочитаем файл целиком, дальнейшие попытки из него что-то прочитать ни к чему не приведут: указатель текущей позиции сдвинулся до самого конца и файл закончился.

```
In [27]: f = open('test.txt', 'r')

for line in f:
    print(line, end=' ')

for line in f: # обратите внимание, что этот цикл не выполняется
    print(line, end=' ')

f.close()
```

```
Hello, world!
Is this an african swallow?
Or an european swallow?
Is this an african swallow?
Or an european swallow?
And another string
```

Файл можно перемотать на начало, если воспользоваться методом .seek(), который возвращаетя к символу на этой позиции.

```
In [28]: f = open('test.txt', 'r')

for line in f:
    print(line, end=' ')

f.seek(0) # вернули файл на начало

for line in f: # теперь печатает!
    print(line, end=' ')

f.close()
```

```
Hello, world!
Is this an african swallow?
Or an european swallow?
Is this an african swallow?
Or an european swallow?
And another string
Hello, world!
Is this an african swallow?
Or an european swallow?
Is this an african swallow?
Or an european swallow?
And another string
```

Если методу read() передать целое число, то питон прочитает только заданное количество символов или битов, если информация в файле записана в бинарном формате.

```
In [58]: with open('test.txt') as f:
    print(f.read(6))
```

Hello,

Чтобы считать все строки файла в список, можно вызвать список от файлового объекта или использовать метод .readlines().

```
In [59]: f = open('test.txt', 'r')
print(list(f))
f.close()
```

```
['Hello, world!\n', ' Is this an african swallow?\n', ' Or an euro
pean swallow?\n', 'And another string\n']
```

```
In [60]: f = open('test.txt', 'r')
print(f.readlines())
f.close()
```

```
['Hello, world!\n', ' Is this an african swallow?\n', ' Or an euro
pean swallow?\n', 'And another string\n']
```

Пример: чтение файла построчно

Мы с вами уже работали с файлом mbox.txt (тот самый с метаданными переписки). В прошлый раз мы его забирали из интернета и полностью грузили в память. Давайте теперь попробуем прочитать его построчно, тем самым загружая в память по одной строке.

В файле есть строки формата "Date: Sat, 5 Jan 2008 09:12:18 -0500" - время, когда ушло письмо. Давайте создадим словарик, в который будем сохранять, в каком часу люди пишут письма (час от 0 до 23 - ключ, количество писем, написанных в это время, - значение).

```
In [29]: with open('mbox.txt') as f:
    for line in f:
        if line.startswith('Date: '):
            print(line)
            break # закомментируйте или удалите break, если хотите
                  # увидеть весь вывод.
```

```
Date: Sat, 5 Jan 2008 09:12:18 -0500
```

Увидели, что у каждого письма на самом деле две строки, начинающихся с 'Date: '. Нужно придумать еще одну эвристику. Например, давайте забирать только те, которые заканчиваются скобкой.

```
In [30]: with open('mbox.txt') as f:  
    for line in f:  
        if line.startswith('Date: ') and line.endswith(')\\n'):  
            print(line)  
            break
```

```
Date: 2008-01-05 09:12:07 -0500 (Sat, 05 Jan 2008)
```

Ок, будем работать с этим форматом. Тут можно обойтись без регулярок и доставать данные через двойной split() - сначала по пробелу, а потом по ':'.

```
In [31]: with open('mbox.txt') as f:  
    for line in f:  
        if line.startswith('Date: ') and line.endswith(')\\n'):  
            print(line.split()[2].split(':')[0])  
            break
```

09

И, наконец, собираем в словарь.

```
In [32]: hours = {}

with open('mbox.txt') as f:
    for line in f:
        if line.startswith('Date: ') and line.endswith('\n'):
            hours[line.split()[2].split(':')[0]] = hours.get(line.split()[2].split(':')[0], 0) + 1

for hour in sorted(hours):
    print(f'В {hour} часов было отправлено {hours[hour]} писем')
```

В 00 часов было отправлено 24 писем
 В 01 часов было отправлено 10 писем
 В 02 часов было отправлено 13 писем
 В 03 часов было отправлено 17 писем
 В 04 часов было отправлено 25 писем
 В 05 часов было отправлено 11 писем
 В 06 часов было отправлено 45 писем
 В 07 часов было отправлено 41 писем
 В 08 часов было отправлено 78 писем
 В 09 часов было отправлено 164 писем
 В 10 часов было отправлено 183 писем
 В 11 часов было отправлено 149 писем
 В 12 часов было отправлено 109 писем
 В 13 часов было отправлено 119 писем
 В 14 часов было отправлено 152 писем
 В 15 часов было отправлено 178 писем
 В 16 часов было отправлено 165 писем
 В 17 часов было отправлено 96 писем
 В 18 часов было отправлено 52 писем
 В 19 часов было отправлено 48 писем
 В 20 часов было отправлено 29 писем
 В 21 часов было отправлено 37 писем
 В 22 часов было отправлено 29 писем
 В 23 часов было отправлено 16 писем

Мы будем проходить графики отдельным блоком, но давайте быстренько построим распределение отправки писем по времени суток.

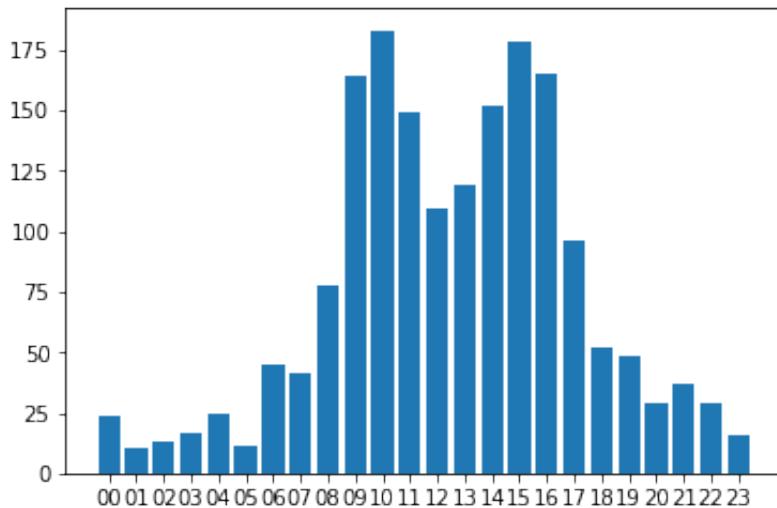
Для начала создадим список отсортированных по ключам значений. Функция построения графика будет брать два аргумента - данные для шкал x и у. На x положим отсортированные ключи, а на у, соответствующие им значения.

```
In [35]: sorted_values = []
for key in sorted(hours.keys()):
    sorted_values.append(hours[key])
```

```
In [36]: import matplotlib.pyplot as plt # импортировали библиотеку для построения графиков
%matplotlib inline
# запустили "магическую" функцию, которая будет отображать графики прямо в блокноте

plt.bar(sorted(hours.keys()), sorted_values)
```

Out[36]: <BarContainer object of 24 artists>



Немного про исключения

Почти наверняка вы уже делали что-то, что приводило к сообщению об ошибке. Сегодня мы научимся их обрабатывать и писать собственные исключения - наши инструкции для Python, чтобы код не ломался, как только что-то пойдет не так.

Например, мы хотим открывать файл по запросу от пользователя, но хотим обработать случай, когда пользователь введет неправильное название.

```
In [37]: fname = input('Введите название файла: ')
fhand = open(fname) # введем название несуществующего файла
```

```
-----
-----
FileNotFoundException                                Traceback (most recent c
all last)
<ipython-input-37-667881827a46> in <module>()
      1 fname = input('Введите название файла: ')
----> 2 fhand = open(fname) # введем название несуществующего файл
a

FileNotFoundException: [Errno 2] No such file or directory: 'f'
```

Напишем блок try/except. Try будет исполняться до тех пор, пока что-то не сломается. Как только возникнет ошибка, ваша программа перейдет в часть except и выполнит действие, описанное в ней. Сообщения об ошибке выведено не будет.

```
In [38]: fname = input('Enter the file name: ') # передадим валидное название файла

try:
    fhand = open(fname)
    count = 0
    for line in fhand:
        if line.startswith('From '):
            count = count + 1
    fhand.close()
    print('There were', count, 'from lines in', fname)

except:
    print('File cannot be opened:', fname)

print('Программа работает')
```

There were 1797 from lines in mbox.txt
Программа работает

```
In [39]: fname = input('Enter the file name: ') # передадим название файла с ошибкой

try:
    fhand = open(fname)
    count = 0
    for line in fhand:
        if line.startswith('From '):
            count = count + 1
    fhand.close()
    print('There were', count, 'from lines in', fname)

except:
    print('File cannot be opened:', fname)

print('Программа работает')
```

File cannot be opened: 2
Программа работает

Except позволил нам избежать ошибки и остановки работы программы, следующая часть исполнена.

Естественно, try-except можно использовать не только с файлами.

```
In [85]: try:  
    print(y) # переменной у пока не существует  
except:  
    print("An exception occurred") # нет ошибки
```

```
An exception occurred
```

Но что делать, если мы хотим пропускать только определенный вид ошибок, но видеть сообщения об остальных? Try/except хороший инструмент для отладки кода. Давайте посмотрим, как называется ошибка при попытке вызова неопределенной переменной.