100 строк Питона, которые решат любое задание ЕГЭ

- `int(x)`

  Переводит значения переменных в целочисленный тип данных.

- `str(x)`

  Переводит значение переменных в строковый тип данных.

- `bool(x)`

  Находит истинность (True) или ложность (False) аргумента в скобках.

- `float(x)`

  Переводит значения переменных в дробный (с плавающей точкой) тип данных.

- Функция `int` может быть расширена до перевода в десятичную систему счисления из любой другой. Для этого в качестве аргументов необходимо указать число в строковой форме и через запятую систему счисления указать основание системы счисления, из которой осуществляется переход. Например, `int('10100101', 2)` переведет число 10100101 из двоичной системы счисления в десятичную.

- `hex(x)`

  Перевод из 10-ой в 16-ую. В качестве аргумента тип данных int.

- `oct(x)`

  Перевод из 10-ой в 8-ую. В качестве аргумента тип данных int.

- `bin(x)`

  Перевод из 10-ой в 2-ую. В качестве аргумента тип данных int.

- `x = input()`

  Функция, необходимая для ввода данных с клавиатуры и записи в переменную. По умолчанию все введенные значения имеют тип данных str.

- `x = int(input())`

  Комбинация 2 функций, которое позволяет считывать число с клавиатуры и работать с ним как с целочисленным типом данных, а не как со строкой по умолчанию.

- `arr = []` Создание пустого списка.

- `arr = [x, 'y', [z, w]]`

  Объявление списка и заполнение конкретными значениями. Значения списка могут быть любых типов данных.

- `arr = [input(), input(), input()....]`

  Объявление списка и заполнение его значениями, введенными с клавиатуры. Сколько `input()` – столько и значений. Каждое из них по умолчанию `str`, но можно преобразовать в любой другой тип данных.

- `print (x,y,...)`

  Выводит заданные аргументы любого типа на экран. По умолчанию все аргументы выводятся через пробел.

- `print (x,y, ... , sep = "")`

  Вывод значений на экран с настраиваемым разделителем. sep (разделитель между элементами) задается вручную последним аргументом. Для этого необходимо приравнять его к символу или строке, которую хотите видеть между выводимыми значениями. Например `print('a', 'b', sep = '@')` выведет `a@b`. Он просто заменяет пробел между элементами на то, что вы указываете в кавычках.

- `print (x,y, ... , sep = "\n")` Выводит значения с каждой новой строки.

- `a+b, a-b, a*b, a**b`

  Математические действия сложения, вычитания и возведения в степень.

- `a/b` Классическое деление, результатом обязательно должно быть число типа float.

- `a//b`

  Нахождение целой части от деления числа a на b. Может использоваться для избавления от последней цифры числа с основанием системы счисления b.

- `a%b`

  Нахождение остатка от деления числа a на b. Может использоваться для поиска последней цифры в системе счисления с основанием b.

- `a>b, a<b, a>=b, a<=b` Математические сравнения 2 чисел

- `a>b>c`

  Двойные неравенства поддерживаются в питоне, true будет если все они выполняются.

- `a+=b, a*=b, a-=b, a/=b, a//=b, a%=b`

  Сокращенные варианты для перезаписывания переменной a путем прибавления, умножения, вычитания и различных делений на число b. Полная форма, например, сложения выглядела бы `a = a + b`

- `a==b`

  Проверка на равенство 2 значений переменных (может применяться для сравнивания строк, и логических переменных и любых других типов данных)

- `a!=b` Проверка на неравенство 2 значений элементов.

- `a and b` Логическое умножение, конъюнкция.

- `a or b` Логическое сложение, дизъюнкция.

- `a<=b` Логическая операция импликации, следствия.

- `a==b` Эквивалентность

- `not(x)` Отрицание

- `If условие:`

  Условный оператор, который выполнит все, что будет написано ниже через tab, если условие будет True. В конце строки с условием обязательно стоит двоеточие.

  Пример:

  ```
  if a>b:
      print(a)
      print(b)
  ```

  Если a>b, на экран выведется сначала a, после чего на следующей строке появится b

  Условий if может быть сколько угодно, однако стоит помнить, что питон проверит каждое из этих условий, независимо от того, было ли предыдущее ложным или нет.

else:

Противоположное условие if и может существовать только при условии, наличия if до него. Выполнит все, что написано ниже через tab только в том случае, если все условия if оказались ложными.

Пример:

```
if a>b:
    print(a)
    print(b)
else:
    print(0)
```

Выводит 0, если условие a>b, оказалось False.

elif условие:

Дополнительное условие связка, которую можно использовать для 3 и более условий. Так же как и второй if обязательно должен содержать условие, которое проверится только если предыдущее if или elif оказались ложными. Используется для ускорения программы с большим количеством разных проверок.

Пример:

```
if a>b:
    print(a)
    print(b)
elif a==b:
    print(b-a)
else:
    print(0)
```

Сначала произойдет проверка a>b, только если она окажется ложной, произойдет проверка на равенство, и, наконец, во всех остальных случаях перейдет к else.

for i in range (a,b,c):

Цикл с известным количеством повторений. Поочередно присваивает i значения из полуинтервала [a,b) с шагом c. Все действия, написанные ниже через табуляцию будут выполнять столько раз, сколько потребуется для перебора всех возможных i.

Пример:

```
for i in range (10,20,2):
    print(i)
```

Выведет числа 10, 12, 14, 16, 18, каждое с новой строки. Обратите внимание, что 20 берется НЕ включительно.

for i in range (a,b):

Сокращенная версия цикла фор, которая выполняет все те же функции, но шаг по умолчанию стоит 1.

for i in range(b):

Самая упрощенная версия цикла фор, в ней мы указываем только финальное значение, а начало и шаг ставятся по умолчанию 0 и 1.

for x in s:

Перебирает все элементы s. Если s это список, то i поочередно присваиваются значения элементов. Если s строка, то i поочередно присваиваются значения символов (включая пунктуацию, пробелы и прочее).

while условие:

Старший брат цикла for, в котором количество повторений определяется не конкретным числом, а выполнением определённого условия. Пока это условие выполняется, цикл будет работать по кругу.

Пример:

```
a=0
b=10
while a<b:
    a+=2
```

Произойдет сравнение 0<10, после чего a увеличится на 2. Затем цикл вернется обратно к сравнению теперь уже проверит 2<10. После очередного увеличения a примет значение 10, и вот теперь уже 10<10 примет значение False, то завершит работу цикла.

continue

Служебное слово, которое позволяет пропустить действия в цикле, написанные после него и сразу же перейти к следующей итерации цикла.

Пример:

```
for i in range (10,20,1):
    if i == 15:
        continue
    print(i)
```

В этом случае значение 15 не будет выведено, поскольку все, что написано после continue пропустится

break

Насильное завершение цикла. Даже если еще не достигнут конец цикла, break автоматически завершит его выполнение.

Объединяем несколько строк в одну минипрограмму:

```
a=int(input())
k=0
while a>0:
    if a%10 == 1:
        k+=1
    a//=10
print(k)
```

Это программа подсчитывает количество единиц по следующему принципу. Она рассматривает последнюю цифру числа (a%10) и проверяет равна ли она 1. Если это утверждение верное, то счетчик к увеличивается на 1. После этого независимо от результата последняя цифра отбрасывается. Число стало короче на 1 цифру, поэтому идет очередная проверка и повторение цикла. Как только мы выполним последнее деление 1 цифры на 10, мы увидим, что число стало равно 0. Цифр больше не осталось и на этом все закончится. В результате мы определим количество единиц, ответив на вопрос «сколько раз последняя цифра числа была равна 1?»

- `arr = [x for x in s]`

  Простейший генератор, создающий массив и заполняющий его значениями из s. В случае, если s строка – x становится каждым отдельным символом.

- `len(arr)`

  Функция, которая находит количество элементов в списке.

- `sum(arr)`

  Функция, которая находит сумму элементов в списке.

- `max(arr)`

  Функция, которая находит максимальный элемент в списке.

- `min(arr)`

  Функция, которая находит минимальный элемент в списке.

- `set(arr)`

  Функция, которая превращает список в множество, убирая все повторяющиеся элементы из списка. `len(set(arr))` – если внимательно посмотреть, найдет количество уникальных элементов в списке.

- `arr.append(x)`

  Метод, который добавляет в конец списка элемент x.

- `arr[i]`

  Обращение к i-тому с начала (считая с 0) элементу массива.

- `arr[-i]`

  Обращение к i-тому с конца (считая с 1) элементу массива.

- `arr.pop(i)` Удаление i-го элемента массива.

- `arr1.extend(arr2)`

  Дописывает в конец списка arr1 все элементы списка arr2.

- `arr.remove(x)`

  Удаляет первый элемент в списке со значением x.

- `arr.index(x,a,b)`

  Находит индекс первого элемента со значением x, в диапазоне [a,b). Значения a,b можно не указывать, если вам нужно осуществить поиск по всему списку.

- `arr.count(x)` Считает количество элементов, равных x.

- `arr.sort()`

  Сортирует значения в порядке возрастания. Если вы захотите вывести отсортированный список в виде `print(arr.sort())`, то получится none потому что вы попытаетесь как бы вывести процесс сортировки, а не финальный результат. Так что сначала написали строчку сортировки, а только потом вывод.

```
arr.sort()
print(arr)
```

- `arr.reverse()` Разворачивает список задом наперед.

- Последние 2 строки можно комбинировать: `arr.sort(revers = True)`

- `str1+str2`

  Склеивание двух строк. Одна записывается за другой без разделителей.

- `int*str` Дублирование строки str int раз.

- `str[i]`

  Обращение к i-му символу строки сначала, начиная от 0

- `str[a:b:c]`

  Срез, часть строки, получаемая в результате записывания друг за другом символов с индексом в диапазоне [a,b) с шагом c. Любой из аргументов можно пропустить с сохранением двоеточий. Тогда по умолчанию будут взяты начало и конец строки с шагом 1. `str[:5:]` возьмет срез от 0 до 4 включительно с шагом 1.

- `str[::-1]` Срез, позволяющий перевернуть строку.

- `len(str)`

  Позволяет определить количество символов в строке, включая пробелы и знаки пунктуации. Очень многие методы и функции строк наследуются от списков, поэтому вы увидите повторяющиеся строки, однако я покажу на примерах, что есть альтернативы, решающие кое-какие проблемы.

- `str.index("x")`

  Находит индекс первого вхождения подстроки x в строку str. Пришло от списков, однако, если таких подстрок нет, выдает ошибку.

- `str.find("x")`

  Метод, применяемый только для строк, выполняющий ту же функцию, что и предыдущий в этом списке, однако если подстрок x нет, то вместо ошибки выведется -1 и программа спокойно будет работать дальше.

- `str.replace ("str1","str2",a)`

  Заменяет a первых подстрок str1 на подстроки str2. Если вы хотите изменить вашу строку таким образом, ее придется перезаписать:

  `str=str.replace ("str1","str2",a)`

- `str.split("x")`

  Разбивает строку на 2 строки по разделителю x. x не запишется ни в одну из этих строк. Эти части необходимо сохранить где-то в отдельных переменных

- `a,b,c,d...= str.split("x")`

  Записывает в переменные a,b,c,d части строки str, полученные в результате разделения. Удобно, если вы точно знаете, сколько у вас частей и их не слишком много.

- `arr=[s.split("x")]`

  Записывает части строки, полученные в результате разбиения, в виде элементов массива. Удобно, если вы не знаете сколько частей получится и вас не особо волнует каждая из этих частей.

- `s.isalpha()`

  Выдает True, если строка состоит только из букв русского или латинского алфавита в нижнем и верхнем регистре.

- `s.isdigit()`

  Выдает True, если строка состоит только из цифр. Используется для проверки перед переводом в int, дабы не вылетела ошибка о невозможности перевода типа данных str в int

- `s.upper()` Переводит все буквы нижнего регистра в верхний.

- `s.lower()` Переводит все буквы из верхнего регистра в нижний.

- `s.count("x")` Подсчитывает количество подстрок x в строке str.

- `def f(a,b,c...):`

  Объявление функции. Необходимо для составления рекурсий и прописывания своих собственных команд, которые составляются под конкретную задачу.

  `f` – название функции, `a,b,c...` аргументы, которые обязаны быть переданы в функцию, чтобы она работала. Функция выполнит все, что написано после нее с табуляцией. Прописывается в программе до того, как у вас появится необходимость ее использовать.

- `return a`

  Служебное слово, возвращающее в качестве результата выполнения функции некоторое значение a. Например для встроенной функции `len(str)`, если бы мы прописывали ее сами, существовало бы `return k`, где `k` – это количество символов в строке, полученное в результате выполнения функции. Проще говоря, `return` – это результат, который вы хотите увидеть по окончании выполнения вашей функции. При этом не все функции содержат return (пример будет чуть позже)

- `return f(n-1)`

  Один из вариантов задания простейшей рекурсии. Результатом выполнения функции f(n) будет та же самая функция, но от другого аргумента. При этом стоит понимать, что значения вложенных функций будут подставляться в внешние функции и до вас вернется именно f(n), который внутри кода использует другие f-ки, для нахождения конечного результата.

- ```
  if n == 1:
      return 1
  ```

  Если не прописать в рекурсии определенные начальные условия, т.е. конкретные значения функции, для конкретных аргументов, она уйдет в бесконечность, пытаясь найти хоть что-то уже известное. Эти 2 строки наглядно показывают, что при n=1 значение равно 1. Другими словами f(1)=1

- ```
  def f(n):
      if n == 1 or n== 0:
          return 1
      else:
          return f(n-1)+f(n-2)
  ```

  Это рекурсивный алгоритм поиска n-го числа Фибоначчи. Это такая последовательность, где каждое следующее число равно сумме 2 предыдущих. Соответственно, чтобы найти любое вообще число, нам нужно знать 2 первых подряд идущих числа, от которых мы сможем поочередно найти все последующие.

Это и происходит в алгоритме. Нам изначально даны f(0) и f(1) = 1. Пусть нужно найти f(3). n=3 не равна 0 и 1, поэтому отправляемся в else. Для нахождения f(3) нужно найти f(2) + f(1). f(1) уже известно, а f(2) снова придется найти по формуле, спустившись на уровни ниже: f(2)=f(1)+f(0) = 1+1=2. Отлично, питон нашел еще одно значение, теперь он просто подставляет его выше. И находит, что f(3) = 2+1 = 3.

`from functools import lru_cache`

Есть маленькая проблема с предыдущим пунктом. Как только мы находим какое-то новое значение, нам не составит труда запомнить его, или куда-то записать. Питон так не умеет по умолчанию. Каждый раз, когда ему нужно будет найти f(n), где n != 1 или 0, он заново просчитывает все эти цепочки. Для банальный чисел Фибоначчи ему нужно 2^n операций, чтобы найти конкретное число n-е число. Поэтому для нахождения уже 45-50 члена последовательности придется потратить 10 минут своего времени. Дабы этого избежать, можно заставить его запоминать новые значения, как это делаем мы, чтобы их не пришлось заново считать.

Для этого потребуется подключить встроенную библиотеку. А точнее одну конкретную функцию оттуда.

`functools` это та самая библиотека. Запомнить легко: `func` от `function`, `tools` – инструменты. Это огромная встроенная библиотека (а значит, ее можно использовать на ЕГЭ). `LRU` – это алгоритм упаковки кэша. Кэш – сохраненные данные, быстрая память, почти как буфер обмена. Поэтому из библиотеки инструментов для функций вы импортируете одну конкретную. Просто пропишите строку в самом начале вашего кода, и вы сможете использовать функцию для ускорения работы вашей рекурсии. А вообще советую самостоятельно поизучать все возможности библиотеки, потому что там много полезного.

`@lru_cache(maxsize = None)`

Строка которая прописывается для активации функции кэширования перед той функцией, значение которой вы хотите сохранять. Если у вас несколько функций, и для каждой вы хотите запомнить значения, значит строку с @ необходимо прописать перед каждой.

`@lru_cache()`

Теоретически функцию можно сократить на новых версиях питона, потому что все, что вы делает в скобках это снимаете ограничения с объема запоминаемых данных, однако лучше не рисковать, потому что кто знает, насколько древний питон попадется на ЕГЭ.

```
from functools import lru_cache
@lru_cache(maxsize = None)
def f(n):
    if n == 1 or n== 0:
        return 1
    else:
        return f(n-1)+f(n-2)
print(f(100))
```

Теперь ваш алгоритм поиска n-го числа Фибоначчи полностью завершен. И даже 100-е число он мгновенно находит. Кстати, это 573147844013817084101.

`global k`

Служебное слово, позволяющее, рассматривать и изменять k за рамками функции. Существуют 2 вида переменных: локальные и глобальные. Локальные существуют внутри экосистемы функций. Мы не можем обратиться к ним после выполнения функции. Единственный способ что-то вытащить оттуда, это написать `return`. А что, если мне нужно, отслеживать вне функции определенные вспомогательные данные, на которые я смогу повлиять? Или я задал какой-то счетчик выполнения циклов вне функции? Да, я мог бы вывести его через `return`, но зачем перегружать систему лишними переменными, если я могу просто сообщить функции, что данные она может напрямую передать в заранее подготовленную переменную. Тогда мне достаточно внутри функции прописать `global k`, что покажет, что из функции может быть произведено прямое обращение к переменной в любой момент выполнения, а не только в конце с помощью `return`.

`return f(n) *g(n)`

Иногда встречаются задания, где даны сразу 2 функции. В этом случае мы просто прописываем 2 функции через def и спокойно пользуемся одной в другой

```
def f(n):
    if k<5:
        arr.append(k)
        f(n+1)
        f(n+2)

arr=[]
f(1)
print(arr)
```

Это тот самый пример, где я составил функцию без `return`. Потому что цель этой функции не найти конкретное значение, а заполнить список числами. Обратите внимание, если я напишу `print(f(1))`, он выведет мне None, потому что вы попросите его вывести процесс заполнения массива. Соответственно функции в данном рекурсивном алгоритме запускаются просто строкой f(n+1)

- `f = open ("file_name", "file_access")`

  Для взаимодействия с файлами вам необходимо сначала открыть файл (как бы подключить его для дальнейшего взаимодействия) и записать его в отдельную переменную

  Существует много разных видов доступа. Для ЕГЭ понадобится только чтение. По крайней мере пока.

- `f = open('example.txt','r')`

  Текстовый файл открыт только для считывания из него данных.

- `f.close()`

  После открытия файла, дабы он не грузил процессор и не болтался в памяти его необходимо закрыть.

- `with open('example.txt') as f:`

  Упрощает работу с файлом. Теперь мы не открываем его, а как бы просто записываем все данные в f. Теперь не придется закрывать, что обычно забывают сделать люди.

- `x=f.readline()`

  Позволяет считать строку и записать в переменную x.

- `arr = f.readlines()`

  Позволяет считать все строки и записать каждую в виде отдельного элемента массива.

- `arr = [x for x in f]`

  Позволяет считать все строки и записать каждую в виде отдельного элемента массива. Обратите внимание, что подобный генератор уже встречался, но там был x in s. S выступала строкой, поэтому x становился условной единицей сроки – символом. Здесь же, x становится условной единицей файла, что эквивалентно строке.

- `arr = [int(x) for x in f]`

  Позволяет считать и записать в список отдельные строки сразу в целочисленном типе данных. Удобно, если в задании дается файл с столбцом чисел.

- `map()`

  Функция, позволяющая быстро и оперативно работать со списками. Применяет действие ко всем элементам списка.

  Пример: `map(int(arr))` преобразует все элементы списка `arr` в целочисленный тип данных.

- `list()`

  Главный помощник map, преобразующий `map` в читаемый тип данных. Map – это процесс преобразование. Некий код, который позволяет переделать одно в другое, но не конечный продукт. Для того, чтобы увидеть, каким станет список после взаимодействия с ним с помощью map необходимо добавить ко всему функцию `list()`

- ```
  arr2 = map(int(arr))
  print(list(arr2))
  ```

  Если я попытаюсь вывести просто arr2, то получу <map object at 0x00000158843ED550>. Это отдельный объект и тип данных, который, как в этом примере, удобно конвертируется в список.

- `float('inf')` Бесконечность. Просто бесконечность.

- ```
  x= "информатику"
  y="математику"
  print("я люблю {x} больше, чем {y}")
  ```

  Форматирование строк – то, что может упростить вам жизнь в 25 номере при сборе чисел. В любое место строки, помеченное {} будет записан любой текст, элемент или значение, которое принимает записанное в фигурных скобках переменная.