# **1. Присвоение по ссылке и по значению. Специфика создания объектов и присвоения в Python, особенности Python в связи с распространенностью использования неизменяемых типов.**

**Модель данных**

Рассмотрим как создаются объекты в памяти, их устройство, процесс объявления новых переменных и работу операции присваивания.

Для того, чтобы объявить и сразу инициализировать переменную необходимо написать её имя, потом поставить знак равенства и значение, с которым эта переменная будет создана. Например строка:

b = **5**

объявляет переменную b и присваивает ей значение 5.

Целочисленное значение 5 в рамках языка Python по сути своей является объектом. Объект, в данном случае – это абстракция для представления данных, данные – это числа, списки, строки и т.п. При этом, под данными следует понимать как непосредственно сами объекты, так и отношения между ними (об этом чуть позже). Каждый объект имеет три атрибута – это идентификатор, значение и тип. Идентификатор – это уникальный признак объекта, позволяющий отличать объекты друг от друга, а значение – непосредственно информация, хранящаяся в памяти, которой управляет интерпретатор.

Имя переменной не должно совпадать с ключевыми словами интерпретатора Python. Список ключевых слов можно найти [здесь](https://pythonworld.ru/osnovy/klyuchevye-slova-modul-keyword.html).(ок) Также его можно получить непосредственно в программе, для этого нужно подключить модуль *keyword* и воспользоваться командой *keyword.kwlist.*

>>> import keyword

>>> print("Python keywords: ", keyword.kwlist)

Проверить является или нет идентификатор ключевым словом можно так:

>>> keyword.iskeyword("try")

**True**

>>> keyword.iskeyword("b")

**False**

Для того, чтобы посмотреть на объект с каким идентификатором ссылается данная переменная, можно использовать функцию *id()*.

>>> a = **4**

>>> b = **5**

>>> id(a)

**1829984576**

>>> id(b)

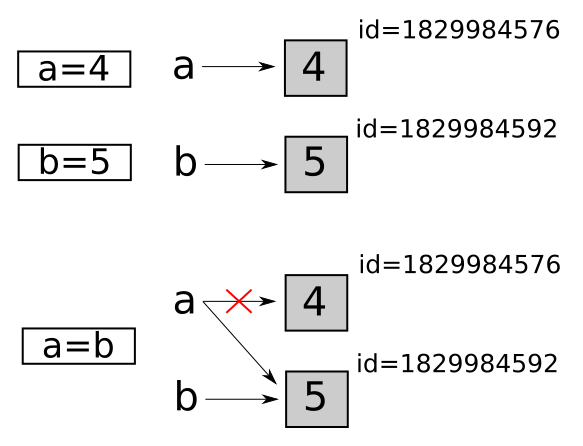
**1829984592**

>>> a = b

>>> id(a)

**1829984592**

Как видно из примера, идентификатор – это некоторое целочисленное значение, посредством которого уникально адресуется объект. Изначально переменная *a* ссылается на объект 4 с идентификатором 1829984576, переменная *b* – на объект с *id* = 1829984592. После выполнения операции присваивания *a = b*, переменная *a* стала ссылаться на тот же объект, что и *b*.



Тип переменной можно определить с помощью функции *type()*. Пример использования приведен ниже.

>>> a = **10**

>>> b = "hello"

>>> c = (**1**, **2**)

>>> type(a)

<*class* 'int'>

>>> type(b)

<*class* 'str'>

>>> type(c)

<*class* 'tuple'>

**Изменяемые и неизменяемые типы данных**

В Python существуют изменяемые и неизменяемые типы.

К **неизменяемым** (*immutable*) типам относятся: целые числа (*int*),  числа с плавающей точкой (*float*), комплексные числа (*complex*), логические переменные (*bool*), кортежи (*tuple*), строки (*str*) и неизменяемые множества (*frozen set*).

К **изменяемым** (*mutable*) типам относятся: списки (*list*), множества (*set*), словари (*dict*).

Как уже было сказано ранее, при создании переменной, вначале создается объект, который имеет уникальный идентификатор, тип и значение, после этого переменная может ссылаться на созданный объект.

Неизменяемость типа данных означает, что созданный объект больше не изменяется. Например, если мы объявим переменную*k* = 15, то будет создан объект со значением 15, типа *int* и идентификатором, который можно узнать с помощью функции *id()*.

>>> k = **15**

>>> id(k)

**1672501744**

>>> type(k)

<*class* 'int'>

Объект с *id* = 1672501744 будет иметь значение 15 и изменить его уже нельзя.

Если тип данных изменяемый, то можно менять значение объекта. Например, создадим список [1, 2], а потом заменим второй элемент на 3.

>>> a = [**1**, **2**]

>>> id(a)

**47997336**

>>> a[**1**] = **3**

>>> a

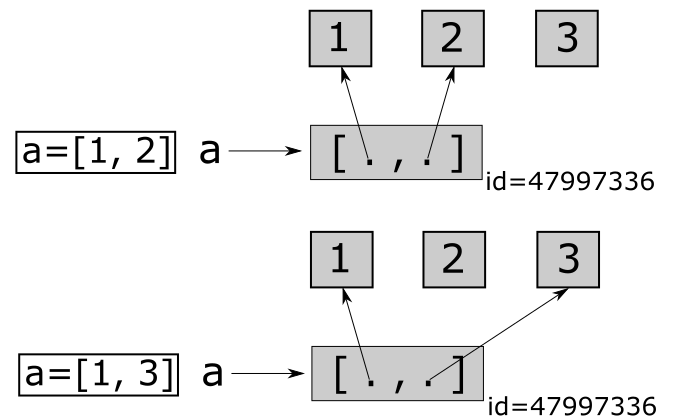
[**1**, **3**]

>>> id(a)

**47997336**

Как видно, объект на который ссылается переменная *a*, был изменен. Это можно проиллюстрировать следующим рисунком.

В рассмотренном случае, в качестве данных списка, выступают не объекты, а отношения между объектами. Т.е. в переменной *a*хранятся ссылки на объекты содержащие числа 1 и 3, а не непосредственно сами эти числа.



# **3. Циклы в Python, работа и устройство цикла for, типичное применение range и enumerate в цикле for.**

**Функция *range***

Функция*range()* создает объект, который внутри себя содержит итератор, выдающий элементы из диапазона, определяемого аргументами функции.

Создадим объект *range*.

**>>> r = range(3)**

Получим его итератор.

**>>> ri = iter(r)**

Используя *next()*, извлечем из него все элементы.

**>>> next(ri)**

**0**

**>>> next(ri)**

**1**

**>>> next(ri)**

**2**

**>>> next(ri)**

**Traceback (most recent call last):**

**File "<pyshell#18>", line 1, in <module>**

**next(ri)**

**StopIteration**

Через параметры функции *range*можно задать диапазон и шаг. Создадим *range*, генерирующий числа от 1 до 10 с шагом 3.

**>>> ri2 = iter(range(1,10,3))**

**>>> next(ri2)**

**1**

**>>> next(ri2)**

**4**

**>>> next(ri2)**

**7**

**>>> next(ri2)**

**Traceback (most recent call last):**

**File "<pyshell#25>", line 1, in <module>**

**next(ri2)**

**StopIteration**

Объект *range*идеально подходит для использования в цикле *for*. Работа цикла *for*организована следующим образом:

получаем итератор у объекта, который предоставляет данные;

используя функцию *next()*, извлекаем элементы на каждой итерации цикла;

если произошел выброс исключения *StopIteration*, то выходим из цикла.

Пример: выведем список целых чисел от 0 до 2 на экран.

**>>> for i in range(3):**

**print(i)**

**0**

**1**

**2**

Список чисел от 1 до 10 с шагом 2.

**>>> for i in range(1, 10, 2):**

**print(i)**

**1**

**3**

**5**

**7**

**9**

*Range*можно использовать в *List Comprehensions*.

**>>> [i for i in range(3)]**

**[0, 1, 2]**

Функция *range()* – мощный инструмент, позволяющий получать последовательности целых чисел в заданном диапазоне с определенным шагом, с доступом через итератор.

**Функция *enumerate***

Функция *enumerate()* конструирует генератор по переданной в нее (через аргумент) объект. Она предоставляет кортежи, состоящие из двух элементов, первый из которых – индекс, а второй – значение, извлекаемое из объекта.

Рассмотрим простой пример: найти в строке первое вхождение символа *‘o’* и вывести номер его позиции. Эта задача может быть решена так.

**>>> msg = "hello!"**

**>>> i = 0**

**>>> for sym in msg:**

**if sym == 'o':**

**print("index = ", i)**

**break**

**i += 1**

**index =  4**

В этой программе нам пришлось заводить дополнительную переменную *i*, значение которой на каждой итерации цикла мы увеличивали на единицу. Это не очень удобно! Использование функции *enumerate()*позволит упростить решение.

**>>> msg = "hello!"**

**>>> for tp in enumerate(msg):**

**if 'o' in tp:**

**print("index = ", tp[0])**

**index =  4**

В процессе работы цикла *for*из объекта, созданного функцией *enumerate()*, будут последовательно извлекаться следующие кортежи:

**(0, ‘h’)**

**(1, ‘e’)**

**(2, ‘l’)**

**(3, ‘l’)**

**(4, ‘o’)**

После извлечения кортежа *(4, ‘o’)*, на экран выведется соответствующее сообщение и цикл завершится. Выведем все кортежи, получаемые из строки*“hello!”* с помощью функции *enumerate()*.

**>>> for tp in enumerate(msg):**

**print(tp)**

**(0, 'h')**

**(1, 'e')**

**(2, 'l')**

**(3, 'l')**

**(4, 'o')**

**(5, '!')**

Функция *enumerate()* позволяет получить индексы элементов объекта при обходе его в цикле *for*без введения дополнительных переменных.

**Функция *map***

Функция *map()*предоставляет возможность применить указанную функцию к каждому элементу объекта. В результате получим список из модифицированных элементов исходного объекта.

Решим задачу возведения в квадрат всех элементов списка с использованием цикла *for*.

**>>> numbers = [1, 2, 3, 4, 5]**

**>>> sq\_nums = []**

**>>> for num in numbers:**

**sq\_nums.append(num\*\*2)**

**>>> sq\_nums**

**[1, 4, 9, 16, 25]**

Эту задачу можно также решить с использованием *List Comprehensions*.

**>>> sq\_nums = [num\*\*2 for num in numbers]**

**>>> sq\_nums**

**[1, 4, 9, 16, 25]**

Точно такой же результат получим, если воспользуемся функций *map()*.

**>>> sq\_nums = list(map(lambda x: x\*\*2, numbers))**

**>>> sq\_nums**

**[1, 4, 9, 16, 25]**

На первый взгляд такой подход может показаться сложнее предыдущего, но вся его мощь заключается в первом аргументе – функции, которая используется для модификации значений элементов объекта. Это может быть как *lambda*-функция, как в нашем примере, так функция, созданная с использованием ключевого слова *def*.

Результат работы функции *map()*можно использовать в цикле *for*.

**>>> msg = "simple"**

**>>> for sym in map(lambda x: x+'-', msg):**

**print(sym, end='')**

**s-i-m-p-l-e-**

Используйте *map()*, если вам необходимо обработать элементы объекта, используя определенную функцию .

**Функция *zip***

Функция *zip()* позволяет в одном цикле *for*производить параллельную обработку данных. Это очень мощный инструмент! *Zip*принимает в качестве аргументов объекты, элементы которых будут объединены в кортежи, полученную структуру можно превратить в список кортежей, если это необходимо.

Решим следующую задачу, демонстрирующую возможности *zip()*: в нашем распоряжении есть два списка, построим третий, каждый элемент которого будет формироваться как сумма элемента из первого списка и элемента из второго, умноженного на два.

**>>> a = [1, 3, 5, 7, 9]**

**>>> b = [2, 4, 6, 8, 10]**

**>>> c = []**

**>>> for (x, y) in zip(a, b):**

**c.append(x + 2 \* y)**

**>>> c**

**[5, 11, 17, 23, 29]**

Результат объединения списков с помощью функции *zip()* представлен ниже. Элементы исходных списков группируются попарно, образуя кортежи.

**>>> a = [1, 3, 5, 7, 9]**

**>>> b = [2, 4, 6, 8, 10]**

**>>> list(zip(a, b))**

**[(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]**

Если в функцию *zip()*передать три списка, то получим кортежи, состоящие из трех элементов.

**>>> a = [1, 3, 5, 7, 9]**

**>>> b = [2, 4, 6, 8, 10]**

**>>> c = [10, 11, 12, 13, 14]**

**>>> list(zip(a, b, c))**

**[(1, 2, 10), (3, 4, 11), (5, 6, 12), (7, 8, 13), (9, 10, 14)]**

В случае, когда исходные списки имеют разную длину, итоговое количество кортежей будет равно числу элементов в самом коротком списке.

**>>> a = [1, 3, 5, 7, 9]**

**>>> d = [20, 21]**

**>>> list(zip(a, d))**

**[(1, 20), (3, 21)]**

С помощью функции *zip()* можно создавать словари.

**>>> a = [1, 3, 5, 7, 9]**

**>>> keys = ['a', 'b', 'c', 'd', 'e']**

**>>> dict(zip(keys, a))**

**{'d': 7, 'c': 5, 'a': 1, 'b': 3, 'e': 9}**

Используйте *zip()* для параллельной обработки данных и быстрого построения структур данных. Такое решение будет функциональным и лаконичным!

***4. Списки в Python. Обращение к элементам списка и создание срезов. . Обход списка и поиск элементов в списке. Ключевые операции, приводящие к изменению списка и порождающие измененные списки, копирование списков.***

Для группировки множества элементов в питоне используется список list, который может быть записан как индексированная последовательность значений, разделенных запятыми, заключенная в квадратные скобки. Списки имеют произвольную вложенность, т.е. могут включать в себя любые вложенные списки. Физически список представляет собой массив указателей (адресов) на его элементы

Со списками, так же как и со строками, можно делать срезы. А именно:

A[i:j] срез из j-i элементов A[i], A[i+1], …, A[j-1].

A[i:j:-1] срез из i-j элементов A[i], A[i-1], …, A[j+1] (то есть меняется порядок элементов).

A[i:j:k] срез с шагом k: A[i], A[i+k], A[i+2\*k],… . Если значение k<0, то элементы идут в противоположном порядке.

Данный модуль предоставляет общие (поверхностная и глубокая) операции копирования.

copy.copy(x) - возвращает поверхностную копию x.

copy.deepcopy(x) - возвращает полную копию x.

Исключение copy.error - возникает, если объект невозможно скопировать.

Разница между поверхностным и глубоким копированием существенна только для составных объектов, содержащих изменяемые объекты (например, список списков, или словарь, в качестве значений которого - списки или словари):

Поверхностная копия создает новый составной объект, и затем (по мере возможности) вставляет в него ссылки на объекты, находящиеся в оригинале.

Глубокая копия создает новый составной объект, и затем рекурсивно вставляет в него копии объектов, находящихся в оригинале. import copy

>>> test\_1 = [1, 2, 3, [1, 2, 3]]

>>> test\_copy = copy.copy(test\_1)

>>> print(test\_1, test\_copy)

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

>>> test\_copy[3].append(4)

>>> print(test\_1, test\_copy)

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

>>> test\_1 = [1, 2, 3, [1, 2, 3]]

>>> test\_deepcopy = copy.deepcopy(test\_1)

>>> test\_deepcopy[3].append(4)

Для операции глубокого копирования часто возникают две проблемы, которых нет у операции поверхностного копирования:

Рекурсивные объекты (составные объекты, которые явно или неявно содержат ссылки на себя) могут стать причиной рекурсивного цикла;

Поскольку глубокая копия копирует всё, она может скопировать слишком много, например, административные структуры данных, которые должны быть разделяемы даже между копиями.

Функция deepcopy решает эти проблемы путем:

Хранения "memo" словаря объектов, скопированных во время текущего прохода копирования;

Позволения классам, определенным пользователем, переопределять операцию копирования или набор копируемых компонентов.

>>>

>>> r = [1, 2, 3]

>>> r.append(r)

>>> print(r)

[1, 2, 3, [...]]

>>> p = copy.deepcopy(r)

>>> print(p)

[1, 2, 3, [...]]

**5. Словари в Python. Итерирование по словарям, преобразование между словарями и списками в Python. Операции с представлениями словарей.**

Структура данных, позволяющая идентифицировать ее элементы не по числовому индексу, а по произвольному, называется *словарем* или *ассоциативным массивом*. Соответствующая структура данных в языке Питон называется dict.

Каждый элемент словаря состоит из двух объектов: ключа и значения.

Словари нужно использовать в следующих случаях:

* Подсчет числа каких-то объектов. В этом случае нужно завести словарь, в котором ключами являются объекты, а значениями — их количество.
* Хранение каких-либо данных, связанных с объектом. Ключи — объекты, значения — связанные с ними данные. Например, если нужно по названию месяца определить его порядковый номер, то это можно сделать при помощи словаря Num['January'] = 1; Num['February'] = 2; ....
* Установка соответствия между объектами (например, “родитель—потомок”). Ключ — объект, значение — соответствующий ему объект.
* Если нужен обычный массив, но при этом масимальное значение индекса элемента очень велико, но при этом будут использоваться не все возможные индексы (так называемый “разреженный массив”), то можно использовать ассоциативный массив для экономии памяти.

**Итерирование по словарям**

Можно легко организовать перебор ключей всех элементов в словаре:

for key in A:

print(key, A[key])

Следующие методы возвращают представления элементов словаря. Представления во многом похожи на множества, но они изменяются, если менять значения элементов словаря. Метод keys возвращает представление ключей всех элементов, метод values возвращает представление всех значений, а метод items возвращает представление всех пар (кортежей) из ключей и значений.

Соответственно, быстро проверить, если ли значение val среди всех значений элементов словаря A можно так: val in A.values(), а организовать цикл так, чтобы в переменной key был ключ элемента, а в переменной val было его значение можно так:

for key, val in A.items():

print(key, val)

Из списка в словарь:

countries = ["Italy", "Germany", "Spain", "USA", "Switzerland"]

dishes = ["pizza", "sauerkraut", "paella", "Hamburger"]

print (dict(zip(countries, dishes)))

**Операции с представлениями словарей**

dict.clear() - очищает словарь.

dict.copy() - возвращает копию словаря.

classmethod dict.fromkeys(seq[, value]) - создает словарь с ключами из seq и значением value (по умолчанию None).

dict.get(key[, default]) - возвращает значение ключа, но если его нет, не бросает исключение, а возвращает default (по умолчанию None).

dict.items() - возвращает пары (ключ, значение).

dict.keys() - возвращает ключи в словаре.

dict.pop(key[, default]) - удаляет ключ и возвращает значение. Если ключа нет, возвращает default (по умолчанию бросает исключение).

dict.popitem() - удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение KeyError. Помните, что словари неупорядочены.

dict.setdefault(key[, default]) - возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ с значением default (по умолчанию None).

dict.update([other]) - обновляет словарь, добавляя пары (ключ, значение) из other. Существующие ключи перезаписываются. Возвращает None (не новый словарь!).

dict.values() - возвращает значения в словаре.

# 7. Множества в Python. Основные способы создания, получения и изменения значений. Обход множеств. Выполнение основных операций с парой множеств в Python.

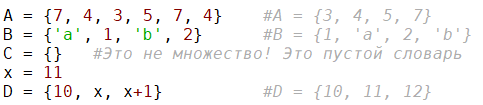
# Множество (set) – неупорядоченный набор уникальных элементов.

Множество может содержать только элементы неизменяемых типов (числа, строки, кортежи). Элементы могут быть разных типов.

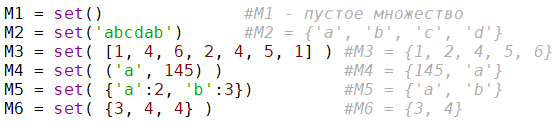
***Создание множества***

Создать множество можно несколькими способами:

* явно указав все элементы в программе. Элементы множества перечисляются в фигурных скобках через запятую. Можно использовать переменные. При перечислении значения могут повторяться, но в множестве будет сохранен только один экземпляр:



* с помощью функции set(). Функция позволяет создать пустое множество или преобразовать в множество данные других типов:



Учтите, что если в исходных данных имелись совпадающие значения, то в множестве всегда остаются только уникальные элементы. Этим эффектом можно воспользоваться для удаления повторяющихся значений из списка или кортежа:



* с помощью генератора множества.

Синтаксис генераторов множеств похож на синтаксис генераторов списков, но выражение заключается в фигурные скобки. Повторяющиеся элементы в множестве не сохраняются. Пример:

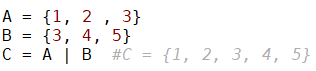


* из других множеств, используя операции над множествами, а также функции и методы множеств.

***Операции над множествами***

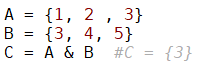
* объединение множеств ( | )

Результатом A | B является множество, содержащее все элементы A и все элементы В:



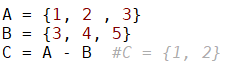
* пересечение множеств ( & )

Результатом A & B является множество, содержащее все элементы, которые одновременно входят в A и В:



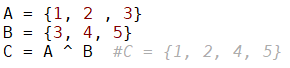
* разность множеств ( - )

Результатом A - B является множество, содержащее все элементы A, которые не входят в В:

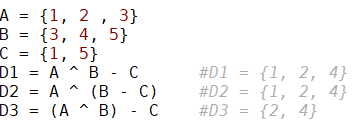


* симметричная разность множеств ( ^ )

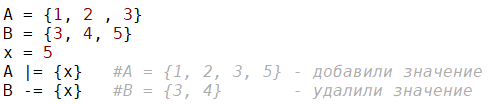
Результатом A ^ B является множество, содержащее те элементы A и В, которые не входят в пересечение этих множеств:



Используя эти операции, из множеств можно составлять выражения. Приоритет операций в порядке убывания: -, &, ^, |. При необходимости, как обычно, воспользуйтесь скобками:



Можно использовать комбинированные операторы присваивания: -=, &=, ^=, |=. Пример:



Следующая группа операций предназначена для выполнения различных проверок. Значением выражения являются значения True или False в зависимости от результатов проверки. Обычно соответствующие выражения используются в инструкциях if или while. В приведенных ниже примерах предполагается, что A = {2, 3, 4, 5}.

* проверка на наличие элемента в множестве ( in, not in ):



* проверка на равенство ( ==, != ):



* проверка, является ли одно множество подмножеством другого ( <, <=, >, >= ):





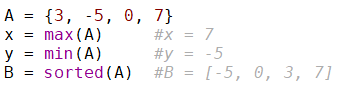
***Функции для работы с множествами***

**set()** – преобразует данные различных типов в множество (примеры см. в разделе создание множеств);

**len()** – возвращает количество элементов множества;

**max(), min()** – находят максимальное и минимальное значение элементов множества;

**sorted( )** – сортирует элементы множества. Возвращает список с отсортированными значениями. Метода sort() у множеств нет, так как множество это не последовательность и порядок элементов в множестве не определен. Примеры:



***Методы множеств***

Ниже приведены только основные методы множеств. Описание всех методов следует смотреть в справочной литературе. В частности, методы, соответствующие операциям -, &, ^, |, -=, &=, ^=, |= , здесь не описаны.

**add( )** – добавляет объект, указанный в качестве параметра, в множество. Метод изменяет текущее множество и ничего не возвращает. Пример:



**remove( )** – удаляет объект, указанный в качестве параметра, из множества. Метод изменяет текущее множество и ничего не возвращает. Если объект не найден, то возникнет ошибка (исключение KeyError). Пример:



**discard( )** – удаляет объект, указанный в качестве параметра, из множества. Метод изменяет текущее множество и ничего не возвращает. Если объект не найден, то ничего не делает. Пример:



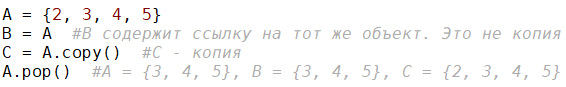
**pop( )** – удаляет произвольный элемент из множества и возвращает его. Если элементов нет, то возникнет ошибка (исключение KeyError). Пример:



**clear( )** – удаляет все элементы из множества. Метод изменяет текущее множество и ничего не возвращает:

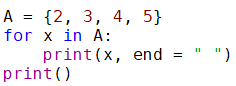


**copy( )** – создает копию множества. Пример:



***Перебор элементов множества***

Для обработки всех элементов множества используется цикл for:



**11. Объявление и вызов функции в Python. Параметры функции со значением по умолчанию и комментирование функции. Получение информации о функции. Способы передачи параметров при вызове функции.**

Функция в python - объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции **def**.

Определим простейшую функцию:

**def**add(x, y):

**return** x + y

Инструкция **return** говорит, что нужно вернуть значение. В нашем случае функция возвращает сумму x и y.

Теперь мы ее можем вызвать:

**>>>**add(1, 10)

11

**>>>**add('abc', 'def')

'abcdef'

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции!):

**>>>def**newfunc(n):

**... def**myfunc(x):

**... return** x + n

**... return**myfunc

**...**

**>>>**new =newfunc(100) *# new - этофункция*

**>>>**new(200)

300

Функция может и не заканчиваться инструкцией return, при этом функция вернет значение None:

**>>>def**func():

**... pass**

**...**

**>>>print**(func())

None

Функция может принимать произвольное количество аргументов или не принимать их вовсе. Также распространены функции с произвольным числом аргументов, обязательными и необязательными.

**>>>def**func(a, b, c=2): *# c - необязательный аргумент*

**... return** a + b + c

**...**

**>>>**func(1, 2) *# a = 1, b = 2, c = 2 (поумолчанию)*

5

**>>>**func(1, 2, 3) *# a = 1, b = 2, c = 3*

6

**>>>**func(a=1, b=3) *# a = 1, b = 3, c = 2*

6

**>>>**func(a=3, c=6) *# a = 3, c = 6, b неопределен*

Traceback (most recent call last):

File "", line 1, in

func(a=3, c=6)

TypeError: func() takes at least 2 arguments (2 given)

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции **lambda**. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией deffunc():

**>>>**func=**lambda** x, y: x + y

**>>>**func(1, 2)

3

**>>>**func('a', 'b')

'ab'

**>>>**(**lambda** x, y: x +y)(1, 2)

3

**>>>**(**lambda** x, y: x +y)('a', 'b')

'ab'

lambda функции, в отличие от обычной, не требуется инструкция return, а в остальном, ведет себя точно так же:

**>>>**func=**lambda**\*args: args

**>>>**func(1, 2, 3, 4)

(1, 2, 3, 4)

**13. Исключения (exceptions)** - ещё один тип данных в python. Исключения необходимы для того, чтобы сообщать программисту об ошибках.

Самый простейший пример исключения - деление на ноль:

>>>

**>>>** 100 / 0

Traceback (most recent call last):

File "", line 1, in

100 / 0

ZeroDivisionError: division by zero

Разберём это сообщение подробнее: интерпретатор нам сообщает о том, что он поймал исключение и напечатал информацию (**Traceback (most recent call last)**).

Далее имя файла (**File ""**). Имя пустое, потому что мы находимся в интерактивном режиме, строка в файле (**line 1**);

Выражение, в котором произошла ошибка (**100 / 0**).

Название исключения (**ZeroDivisionError**) и краткое описание исключения (**division by zero**).

**Иерархия исключений**

В языке программирования Python присутствует строгая иерархия исключений. Вершиной является BaseException, включающий в себя все существующие разновидности исключений:

* SystemExit – возникает при выходе из программы с помощью sys.exit;
* KeyboardInterrupt – указывает на прерывание программы пользователем;
* GeneratorExit – появляется при вызове метода close для объекта generator;
* Exception – представляет совокупность обычных несистемных исключений.

Перечень несистемных исключений, которые содержит в себе класс Exception приведены в следующей таблице. Все они актуальны для последней на данный момент версии Python 3.

|  |  |
| --- | --- |
| Название | Характеристика |
| ArithmeticError | Порождается арифметическими ошибками (операции с плавающей точкой, переполнение числовой переменной, деление на ноль) |
| AssertionError | Возникает при ложном выражении в функции assert |
| AttributeError | Появляется в случаях, когда нужный атрибут объекта отсутствует |
| BufferError | Указывает на невозможность выполнения буферной операции |
| EOFError | Проявляется, когда функция не смогла прочитать конец файла |
| ImportError | Сообщает о неудачном импорте модуля либо атрибута |
| LookupError | Информирует про недействительный индекс или ключ в массиве |
| MemoryError | Возникает в ситуации, когда доступной памяти недостаточно |
| NameError | Указывает на ошибку при поиске переменной с нужным именем |
| NotImplementedError | Предупреждает о том, что абстрактные методы класса должны быть обязательно переопределены в классах-наследниках |
| OSError | Включает в себя системные ошибки (отсутствие доступа к нужному файлу или директории, проблемы с поиском процессов) |
| ReferenceError | Порождается попыткой доступа к атрибуту со слабой ссылкой |
| RuntimeError | Сообщает об исключении, которое не классифицируется |
| StopIteration | Возникает во время работы функции next при отсутствии элементов |
| SyntaxError | Представляет собой совокупность синтаксических ошибок |
| SystemError | Порождается внутренними ошибками системы |
| TypeError | Указывает на то, что операция не может быть выполнена с объектом |
| UnicodeError | Сообщает о неправильной кодировке символов в программе |
| ValueError | Возникает при получении некорректного значения для переменной |
| Warning | Обозначает предупреждение |

Теперь, зная, когда и при каких обстоятельствах могут возникнуть исключения, мы можем их обрабатывать. Для обработки исключений используется конструкция **try - except**.

Первый пример применения этой конструкции:

>>>

**>>> try**:

**...**  k = 1 / 0

**... except** ZeroDivisionError:

**...**  k = 0

**...**

**>>> print**(k)

0

В блоке try мы выполняем инструкцию, которая может породить исключение, а в блоке except мы перехватываем их. При этом перехватываются как само исключение, так и его потомки. Например, перехватывая ArithmeticError, мы также перехватываем FloatingPointError, OverflowError и ZeroDivisionError. Возбуждает указанное исключение.Инструкция позволяет прервать штатный поток исполнения при помощи возбуждения исключения. Если после инструкции отсутствует выражение (например, не указывается тип исключения), то повторно поднимается отловленное исключение. Если в данной области нет активного исключения (например, raise не находится внутри блока except), возбуждается Runtime Error. Если выражение присутствует, то результатом его вычисления должен являться либо экземпляр наследника [BaseException](https://pythonz.net/references/named/baseexception/), либо сам наследник, то есть тип. Во втором случае объект исключения будет сформирован путём инстанциирования типа без аргументов.

*# Объект сформируется из класса автоматически.*  
 **raise** MyException  
   
 *# Формируем объект исключения вручную.*  
 **raise** MyException('Моё исключение')

**Объект трассировки**  
  
Объект трассировки обычно создаётся автоматически при возбуждении исключения и подвязывается к нему в атрибут \_\_traceback\_\_ (поддерживает запись). Вы можете в один приём создать исключение и подвязать к нему трассировку при помощи метода исключения with\_traceback() — метод вернёт тот же объект исключения для которого вызван:

**raise** Exception('Возбуждено исключение').with\_traceback(traceback\_obj)

 Ещё две инструкции, относящиеся к нашей проблеме, это **finally** и else. Finally выполняет блок инструкций в любом случае, было ли исключение, или нет (применима, когда нужно непременно что-то сделать, к примеру, закрыть файл). Инструкция else выполняется в том случае, если исключения не было.

>>> f = open('1.txt')

>>> ints = []

>>> **try**:

... **for** line **in** f:

... ints.append(int(line))

... **except** ValueError:

... print('Это не число. Выходим.')

... **except** Exception:

... print('Это что ещё такое?')

... **else**:

... print('Всё хорошо.')

... **finally**:

... f.close()

... print('Я закрыл файл.')

... *# Именно в таком порядке: try, группа except, затем else, и только потом finally.*

...

Это не число. Выходим.

Я закрыл файл.

14.

Взаимодействие с файлами в языке программирования Python предоставляет пользователю возможность хранить информацию, которая была обработана приложением, чтобы потом получать к ней доступ в любое удобное время. Базовые функции этого языка позволяют достаточно легко создавать, записывать и читать данные из файлов.

**Создание и открытие**

Чтобы получить возможность работать с файлом в Python 3, для начала его необходимо создать. Сделать это можно стандартными средствами операционной системы, перейдя в нужный каталог и создав новый документ с форматом txt. Однако аналогичное действие выполняется и с помощью метода open в языке программирования Python, которому надо передать в качестве параметров название файла и режим его обработки.

Следующий код демонстрирует получение переменной file ссылки на новый документ. Если запустить эту программу, она создаст текстовый файл test.txt в папке, где хранится исходный код.

file = open("test.txt", "w")

file.close()

Если же файл с указанным именем test.txt уже существует в каталоге с кодом, программа просто продолжит работу с ним, не создавая новый документ. Как можно заметить, имя файла является первым параметром метода open. Сразу за ним следует специальная буква, которая обозначает метод обработки данных. В данном случае “w” означает write, то есть запись. Подробнее обо всех доступных режимах работы будет немного дальше, а сейчас важно усвоить, что **после выполнения любых манипуляций над файлом, его обязательно следует закрыть с помощью функции close, чтобы гарантированно избежать потери информации**.

В предыдущем примере для доступа к файлу был использован относительный путь, который не содержит в себе исчерпывающих сведений о местоположении объекта на жестком диске. Для того, чтобы задать их, необходимо в качестве первого аргумента функции open прописать абсолютный путь. В данном случае документ test.txt будет находиться в корневом каталоге на диске D, а не в папке программы.

file = open(r"D:\test.txt", "w")

file.close()

Перед строковым литералом мы использовали символ r, для отключения экранирования. Иначе компилятор посчитает последовательность “\t” как символ табуляции и выдаст исключение.

**Режим открытия**

Ранее были упомянуты специальные символы, которые используются в языке Python при открытии файла. Они задают режим открытия файла. Указывают программе, как именно нужно открывать. Все они представлены в следующей таблице, которая содержит их сигнатуру и короткое описание назначения.

|  |  |
| --- | --- |
| Символ | Значение |
| “r” | открытие для чтения (по умолчанию) |
| “w” | открытие для записи, а если его не существует по заданному пути, то создается новый |
| “x” | открытие для записи, но только если его еще не существует, иначе будет выдано исключение |
| “a” | открытие на дополнительную запись, чтобы информация добавлялась в конец документа |
| “b” | открытие в двоичном режиме |
| “t” | открытие в текстовом режиме (по умолчанию) |
| “+” | открытие одновременно на чтение и запись |

Пользуясь вторым аргументом метода open, можно комбинировать различные режимы работы с файлами, указывая, к примеру, “rb” для чтения записанных данных в двоичном режиме.

Еще один пример: отличие “r+” и “w+” заключается в том, что во втором случае создастся новый файл, если такого нет. В первом же случае возникнет исключение. При использовании “r+” и “w+” файл будет открыт и на чтение и на запись. Пример обработки исключения разберем, когда будем рассматривать чтение файла.

**Методы**

Объект, который возвращает функция open, содержит ссылку на существующий файл. Также в нем имеется информация о созданном документе, представленная в виде четырех основных полей. Все они описаны в следующей таблице, которая содержит их имена и значение.

|  |  |
| --- | --- |
| Свойство | Значение |
| name | возвращает имя файла |
| mode | возвращает режим, в котором был открыт |
| closed | возвращает true, если файл закрыт и true, если открыт |
| softspace | возвращает true, если при выводе данных из файла не следует отдельно добавлять символ пробела |

Чтобы вывести на экран свойства файла, достаточно воспользоваться оператором доступа, то есть точкой, после чего передать это в качестве параметра уже знакомой функции print.

Например:

f = open(r"D:\test.txt", "w")

print(f.name)

f.close()

D:\test.txt

**Запись**

В Python 3 запись в файл осуществляется с помощью метода write. Метод вызываем у объекта, который ссылается на существующий файл. Важно помнить, что для этого следует предварительно открыть документ с помощью функции open и указать режим записи символом “w”. Метод write принимает в качестве аргумента данные, которые нужно поместить в текстовый файл. Следующий пример кода показывает запись строки “hello”.

file = open("test.txt", "w")

file.write("hello")

file.close()

Если необходимо добавить новую информацию к записанным ранее данным, следует заново вызвать функцию open, указав ей в качестве режима работы символ “a”. В противном случае все сведения из файла test.txt будут полностью удалены. В приведенном ниже примере кода текстовый документ открывается для дополнительной записи, после чего в него помещается строковый литерал “ world” с пробелом вначале. Таким образом в test.txt будет располагаться “hello world”. После всего этого не нужно забывать об обязательном закрытии файла.

file = open("test.txt", "a")

file.write(" world")

file.close()

Именно так осуществляется самая простая процедура записи данных в текстовый файл. Стоит заметить, что язык программирования Python содержит массу дополнительных средств для более продвинутой работы с документами, которые также включают и улучшенную запись.

**Чтение**

Для чтения информации из файла в Python 3, следует вызывать метод read через объект, который ссылается на существующий документ. Также необходимо не забывать указывать “r” в качестве второго параметра функции open при открытии текстового файла.

В следующем примере read возвращает информацию из test.txt в метод print, который затем выводит сведения на экран. Как и прежде, программа завершается закрытием документа при помощи метода close. Метод read также может принимать целочисленный параметр, который используется для передачи количества символов для чтения.

**with as**

Чтобы немного автоматизировать обработку текстовых файлов, рекомендуется использовать связку операторов with as. Благодаря им пропадает необходимость в вызове метода close для документа, который необходимо закрыть, поскольку это происходит автоматически. Все это демонстрируется в следующем фрагменте кода, где происходит считывание данных из test.txt. Как обычно, метод print используется для вывода строковой информации на экран.

with open('test.txt', 'r') as file:

print(file.read())

Более того, в этом случае не нужно обрабатывать исключение. Если файла с указанным именем нету, то строки с вложенным кодом в операторе with выполняться не будут.

С помощью рассмотренных возможностей языка программирования Python, пользователь в состоянии достаточно легко выполнять основные операции по работе с чтением и записью информации в файлы.

# 16. Концепция класса и объекта. Принципы и механизмы ООП.

**Объектно-ориентированное программирование** (**ООП**) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

**Класс** — тип, описывающий устройство объектов. **Объект** — это экземпляр класса. Класс можно сравнить с чертежом, по которому создаются объекты.

Объектно-ориентированная парадигма имеет **несколько принципов**:

1.Данные структурируются в виде объектов, каждый из которых имеет определенный тип, то есть принадлежит к какому-либо классу.

2.Классы – результат формализации решаемой задачи, выделения главных ее аспектов.

3.Внутри объекта инкапсулируется логика работы с относящейся к нему информацией.

4.Объекты в программе взаимодействуют друг с другом, обмениваются запросами и ответами.

5.При этом объекты одного типа сходным образом отвечают на одни и те же запросы.

6.Объекты могут организовываться с более сложные структуры, например, включать другие объекты или наследовать от одного или нескольких объектов.

Появление в ООП отдельного понятия класса закономерно вытекает из желания иметь множество объектов со сходным поведением. Класс в ООП — это в чистом виде абстрактный тип данных, создаваемый программистом. С этой точки зрения объекты являются значениями данного абстрактного типа, а определение класса задаёт внутреннюю структуру значений и набор операций, которые над этими значениями могут быть выполнены. Желательность иерархии классов (а значит, наследования) вытекает из требований к повторному использованию кода — если несколько классов имеют сходное поведение, нет смысла дублировать их описание, лучше выделить общую часть в общий родительский класс, а в описании самих этих классов оставить только различающиеся элементы.

БАЗОВЫЕ МЕХАНИЗМЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Объектно-ориентированное программирование базируется на трех основных принципах. Это инкапсуляция, полиморфизм и наследование.

**1.Инкапсуляция** - это понятие в объектно-ориентированном программирование обозначающее защиту данных (сокрытие данных) от внешнего пользователя.

Для лучшего понимания, приведу пример инкапсуляции в жизни на примере телефона:

Чтобы совершить звонок по сотовому телефону, вам необязательно знать как работают сотовые сети, где расположены вышки связи, как у них организованно хранение данных и прочее. Все что вам нужно знать, чтобы совершить звонок по сотовому телефону - это что у вас должен быть номер того абонента, кому вы хотите позвонить и деньги на счету.

**Свойство инкапсуляции в объектно-ориентированном программировании обозначает то, что нужно дать пользователю вашей программы доступ только к нужным интерфейсам (методам) и скрыть доступы к внутренним приватным методам и полям вашего класса.**

**2.Наследование** - это второе свойство объектно-ориентированного программирования, позволяющее наследовать функционал (набор полей и методов) одного класса в другой.

**3.Полиморфизм -** это свойство объектно-ориентированного программирования, позволяющее

**17. Объявление класса, конструктор, создание объектов и одиночное наследование в Python. Управление доступом к атрибутам класса в Python.**

**Создание класса**

Если функция создаётся с помощью ключевого слова def — то класс создаётся с помощью ключевого слова class.

Вот пример простейшего класса:

|  |  |  |
| --- | --- | --- |
| 1 | >>> class ThisIsClass: | |
| 2 | ...   pass |

* class — указание на то, что далее последует объявление класса;
* ThisIsClass — имя класса;
* pass — оператор в классе.

Следует запомнить, что *объект* (или *инстанс* — instance) класса, и *экземпляр* класса — это две разные сущности.

*Объект класса* создаётся во время его объявления (грубо говоря — когда интерпретатор встречает слово class в коде).

*Экземпляр класса* — отдельный, новый объект класса, который создаётся во время присваивания класса какой-то переменной и вызова с оператором ().

Например:

|  |  |  |
| --- | --- | --- |
| 1 | >>> id(ThisIsClass)    // это объект класса | |
| 2 | 140249307055928 |

|  |  |  |
| --- | --- | --- |
| 3 | >>> cl = ThisIsClass() // экземпляр класса, новый объект в памяти | |
| 4 | >>> id(cl) |

|  |  |
| --- | --- |
| 5 | 140249307072992 |
| 6 | >>> cl = ThisIsClass   // указатель на объект класса, но не экземпляр | |

|  |  |
| --- | --- |
| 7 | >>> id(cl) |
| 8 | 140249307055928 | |

После создания экземпляра класса — вы можете работать с ним, как с обычной переменной или функцией.

#### Наследование (inheriting) классов

До сих пор вы создавали класс так:

|  |  |
| --- | --- |
| 1 | class ThisIsClass: |

Однако, классы в **Python** поддерживают механизм наследования, когда дочерний класс наследует атрибуты и методы из родительского класса.

Для того, что бы указать, что у класса есть родительских класс — он указывается в скобках:

|  |  |
| --- | --- |
| 01 | >>> class ParentClass: |
| 02 | ...   parentclassattribute = 'parentclassattribute' | |

|  |  |
| --- | --- |
| 03 | ... |
| 04 | >>> class InheritingClass(ParentClass): | |

|  |  |
| --- | --- |
| 05 | ...   def method(self): |
| 06 | ...     print self.parentclassattribute | |

|  |  |
| --- | --- |
| 07 | ... |
| 08 | >>> cl = InheritingClass() | |

|  |  |
| --- | --- |
| 09 | >>> cl.method() |
| 10 | parentclassattribute | |

В этом примере InheritingClass наследует значение атрибута parentclassattribute  из родительского — ParentClass класса.

Тоже касается и методов классов:

|  |  |
| --- | --- |
| 01 | >>> class ParentClass: |
| 02 | ...   def paerntmethod(self): | |

|  |  |  |
| --- | --- | --- |
| 03 | ...     print ('This is parent') | |
| 04 | ... |

|  |  |  |
| --- | --- | --- |
| 05 | >>> class InheritingClass(ParentClass): | |
| 06 | ...   pass |

|  |  |
| --- | --- |
| 07 | ... |
| 08 | >>> cl = InheritingClass() | |

|  |  |  |
| --- | --- | --- |
| 09 | >>> cl.paerntmethod() | |
| 10 | This is parent |

Дочерний класс InheritingClass использует метод paerntmethod родительского класса.

Кроме того, дочерний класс может переопределить метод (или атрибут) родительского класса:

|  |  |
| --- | --- |
| 01 | >>> class ParentClass: |
| 02 | ...   classattribute = 'classattribute' | |

|  |  |
| --- | --- |
| 03 | ...   def method(self): |
| 04 | ...     print 'This is parent method' | |

|  |  |
| --- | --- |
| 05 | >>> class InheritingClass(ParentClass): |
| 06 | ...   classattribute = 'Inherited class attribute' | |

|  |  |
| --- | --- |
| 07 | ...   def method(self): |
| 08 | ...     print ('This is child') | |

|  |  |
| --- | --- |
| 09 | ... |
| 10 | >>> cl = InheritingClass() | |

|  |  |
| --- | --- |
| 11 | >>> cl.classattribute |
| 12 | 'Inherited class attribute' | |

|  |  |  |
| --- | --- | --- |
| 13 | >>> cl.method() | |
| 14 | This is child |

Дочерний класс InheritingClass перезаписывает данные родительского атрибута classattribute своим значением, и переопределяет поведение родительского метода method().

Управления доступом к атрибутам в Python может осуществляться несколькими способами:

* \_\_getattr\_\_ \_\_set\_\_ \_\_getattribute\_\_ \_\_delete\_\_
* property
* дескриптор

property и дескрипторы применяются к отдельным атрибутам.

### Свойства

attribute = property(fget, fset, fdel, doc)  
Все аргументы по умолчанию None, при обращение приводят к исключению

### Дескрипторы

Протокол дескрипторов позволяет передавать выполнение операций чтения и записи для определенного атрибута методам отдельного обьекта класса.  
В отличии от свойств в дескрипторе при отсутствии метода не возникает исключение.  
Если не реализован \_\_set\_\_, то атрибут будет просто переопределен

### \_\_getattr\_\_ и \_\_getattribute\_\_

\_\_getattr\_\_ -вызывается при обращение к несуществующим атрибутам  
\_\_set\_\_ -при присваивании значений  
\_\_getattribute\_\_ -вызывается при обращение к любым атрибутам  
  
Делегирования - управление доступом ко всем атрибутам встроенного объекта  
функция getattr(self, item) эквивалентна self.item

Предотвращение зацикливаний  
Если внутри метода \_\_getattribute\_\_ обратится к другому атрибуту то эта операция произведет вызов метода  \_\_getattribute\_\_ и это приведи к бесконечному циклу  
<https://germangorelkin.blogspot.com/2017/07/python_18.html>

В Python роль конструктора играет метод \_\_init\_\_().

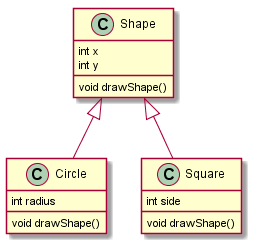
В Python наличие пар знаков подчеркивания спереди и сзади в имени метода говорит о том, что он принадлежит к группе методов перегрузки операторов. Если подобные методы определены в классе, то объекты могут участвовать в таких операциях как сложение, вычитание, вызываться как функции и др.

При этом методы перегрузки операторов не надо вызывать по имени. Вызовом для них является сам факт участия объекта в определенной операции. В случае конструктора класса – это операция создания объекта. Так как объект создается в момент вызова класса по имени, то в этот момент вызывается метод \_\_init\_\_(), если он определен в классе.

Необходимость конструкторов связана с тем, что нередко объекты должны иметь собственные свойства сразу.

**18. Полиморфизм и утиная типизация и проверка принадлежности объекта к классу в языке Python.**

**Полиморфизм** является одним базовым аспектом объектно-ориентированного программирования и предполагает способность к изменению функционала, унаследованного от базового класса. Благодаря ему мы можем добавить новые методы и переназначить методы родительского класса. Полиморфизм делает код гибким, такой код легко расширять и поддерживать.



В данном примере Shape – родительский класс(superclass). Circle и Square – наследники. Мы добавляем переменную, которая нам нужна(radius, side) и переназначаем метод drawShape()

**Утиная типизация** – идея утиной типизации заключается в том, что нас не волнует какого типа будет объект. Смысл утиной типизации заключается в ослаблении типов. Вместо того чтобы заботиться о точном классе объекта мы заботимся о том какие методы для него можно вызвать и какие операции над ним можно выполнять.

*Цитата: "Если что-то выглядит как утка, плавает как утка и крякает как утка, то, вероятно, это утка"*

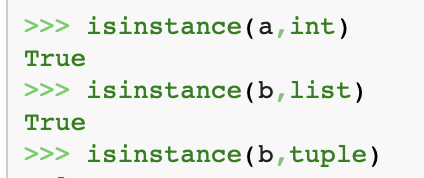
***A screenshot of a cell phone

Description automatically generated* Проверка принадлежности объекта к классу в языке Python.**

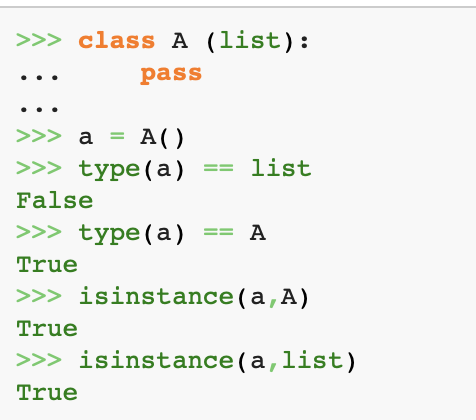
Есть два основных способа проверить к какому классу принадлежит объект: **type, isinstance**

Type и isinstance – две разные команды

Функция **isinstance()** специально создана для проверки принадлежности данных определенному классу (типу данных)

**Синтаксис:**

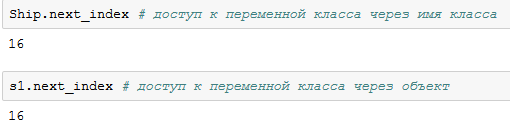
Эта функция isinstance() поддерживает наследование. Для isinstance() экземпляр производного класса также является экземпляром базового класса. Для type() это не так:

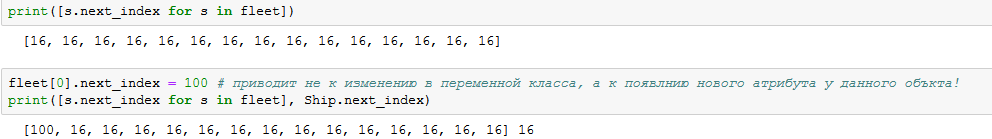
****

С помощью **issubclass** проверить, является ли один класс потомком другого.

**19. Методы классов и статические переменные и методы в Python. Специальные методы для использования пользовательских классов со стандартными операторами и функциями.**

Статическая переменная: переменная, объявленная внутри класса, обращение к которой происходит через имя класса. Если работать с переменной конкретного экземпляра (объекта) – т.е. обращаться к ней не через имя класса (статически), а через имя объекта, то это будет уже другое значение.

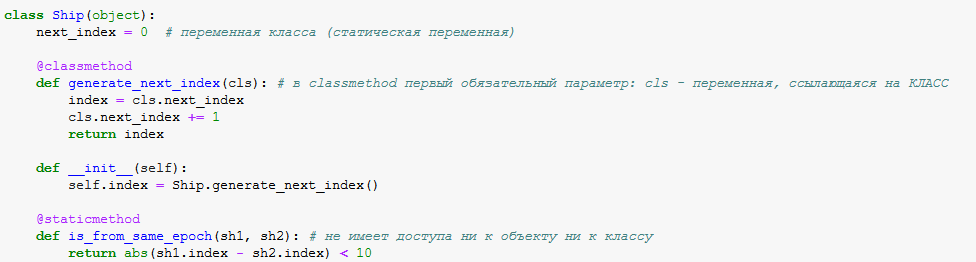


****

В метод класса первым аргументом неявным образом передаётся класс (аналогично метод экземпляра получает в первом аргументе сам экземпляр). Для удобного объявления метода класса используйте декоратор @classmethod.

Внутри класса можно создать метод, который будет доступен без создания экземпляра класса. Для этого перед определением метода внутри класса следует указать декоратор @staticmethod. Статический метод не имеет доступа к объекту.

Пример класса:

****

## 

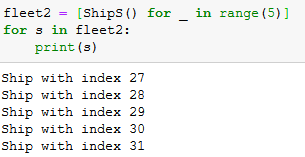
## Использование метода класса:

## 

## Использование статического метода:

## *Специальные методы*

\_Специальные зарезервированные методы имеют префикс — двойной символ подчеркивания. С их помощью реализованы такие механизмы, как конструкторы, последовательности, итераторы, и т.д. 



Экземпляры классов и специальные методы:

* \_\_dict\_\_ — словарь атрибутов класса, можно изменять этот словарь напрямую;
* \_\_class\_\_ — объект-класс, экземпляром которого является данный инстанс;
* \_\_init\_\_ — конструктор. Если в базовом классе есть конструктор, конструктор производного класса должен вызвать его;
* \_\_del\_\_ — деструктор. Если в базовом классе есть деструкор, деструктор производного класса должен вызвать его;
* \_\_cmp\_\_ — вызывается для всех операций сравнения;
* \_\_hash\_\_ — возвращает хеш-значение объекта, равное 32-битному числу;
* \_\_getattr\_\_ — возвращает атрибут, недоступный обычным способом;
* \_\_setattr\_\_ — присваивает значение атрибуту;
* \_\_delattr\_\_ — удаляет атрибут;
* \_\_call\_\_ — срабатывает при вызове экземпляра класса.

Специальные методы для поддержки операций сравнения:

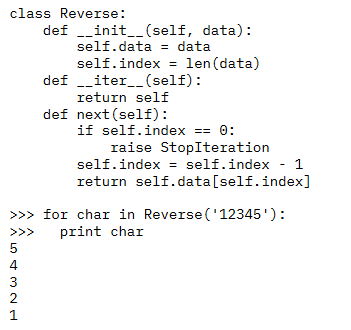
* х == у.\_*еq\_*(у);
* х != у.\_*nе\_*(у);
* х < у.\_*lt\_*(y);
* х > y.\_*gt\_*(y);
* х <= у.\_*lе\_*(у);
* х >= у.\_*ge\_*(y);
* у in х. \_*contains\_*(у).

Объекты классов можно привести к строковому или числовому типу.

* \_\_repr\_\_ — возвращает формальное строковое представление объекта;
* \_\_str\_\_ — возвращает строковое представление объекта;
* \_\_oct\_\_ , \_\_hex\_\_ , \_\_complex\_\_ , \_\_int\_\_ , \_\_long\_\_ , \_\_float\_\_ — возвращают строковое представление в соответствующей системе счисления.

Интерпретатор Python будет автоматически подставлять метод \_*ne\_*() (not equal - не равно), реализующий действие оператора неравенства (!=), если в классе присутствует реализация метода \_*eq\_*(), но отсутствует реализация метода \_*ne\_*() .

Итераторы хороши там, где списки не подходят в силу того, что занимают много памяти, а итератор возвращает его конкретное значение. В классе нужно определить два стандартных метода — \_\_iter\_\_ и next. Метод \_\_iter\_\_ будет возвращать объект через метод next:



**20. Основные возможности, поддерживаемые функциональными языками программирования. Поддержка функционального программирования в Python.**

Функциональное программирование — парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемости этого состояния.

Основано на лямбда-исчислении, базирующемся на двух операциях: абстракции (операции, позволяющей конструировать функции) и аппликации (операции вызова функции). Определяет программу как вызов функции (аппликация).

Принципы функционального программирования:

* вызов функции для одних и тех же значений параметров должен возвращать одинаковый результат (чистые (pure) функции);
* изменения контекста (внешних переменных) при вызове функции определяется как побочный эффект (нежелателен!);
* изменяемость (mutation) переменных нежелательна;
* поддержка функций высших порядков (вызов функций для функций).

### *Что делает язык функциональным или императивным?*

Функции:

* функции "граждане первого класса" (functions as first-class citizens);
* функции высшего порядка (higher-order functions);
* замыкания (closures);
* функции без побочных эффектов (pure functions);
* рекурся, хвостовая рекурсия (recursion, tail recursion).

Данные:

* неизменяемые структуры данных (immutable data structures);
* вычисление результатов функций вместо явного хранения (и изменения) состояния программы (avoid changing-state).

Идиомы:

* итераторы, последовательности, ленивые вычисления, сопоставление с образцом,монады (iterators, sequences, lazy evaluation, pattern matching, monads...)

### *Поддержка функционального программирования в Python*

PRO:

* функции "граждане первого класса" (функции можно динамически создавать и уничтожать, передавать их в другие функции, возвращать их как значения и так далее);
* лямбда-функции (анонимные функции);
* поддержка функциональных идиом в стандартной библотеке: map/filter/reduce, itertools, operator;
* генераторы могут использоваться для ленивых вычислений (вычислений по требованию).

CONTRA:

* невозможно разделить функции с побочным эффектом и без;
* изменяемые переменные;
* дорогостоящие операции копирования в памяти;
* императивный стиль для циклов;
* отсутствие оптимизации для хвостовой рекурсии;
* нет синтаксиса для проверки шаблонов;
* система типов базируется только на классах;
* нет механизма перегрузки функций;
* в стандартной библиотеке не реализован механизм построения композиции функций;
* имеется только императивный механизм обработки ошибок.

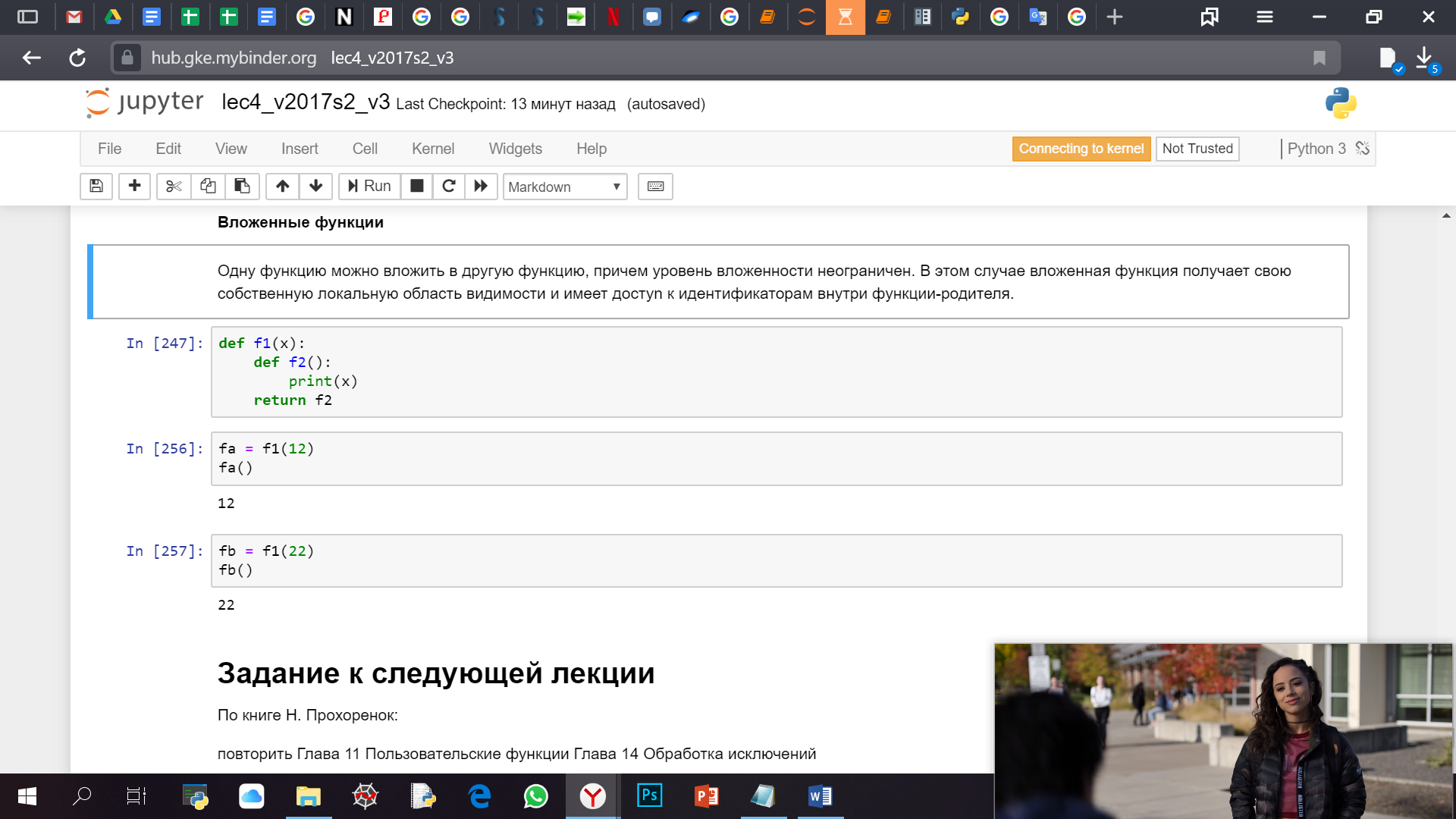
**23. Вложенные функции и замыкания, специфика реализации в Python.**

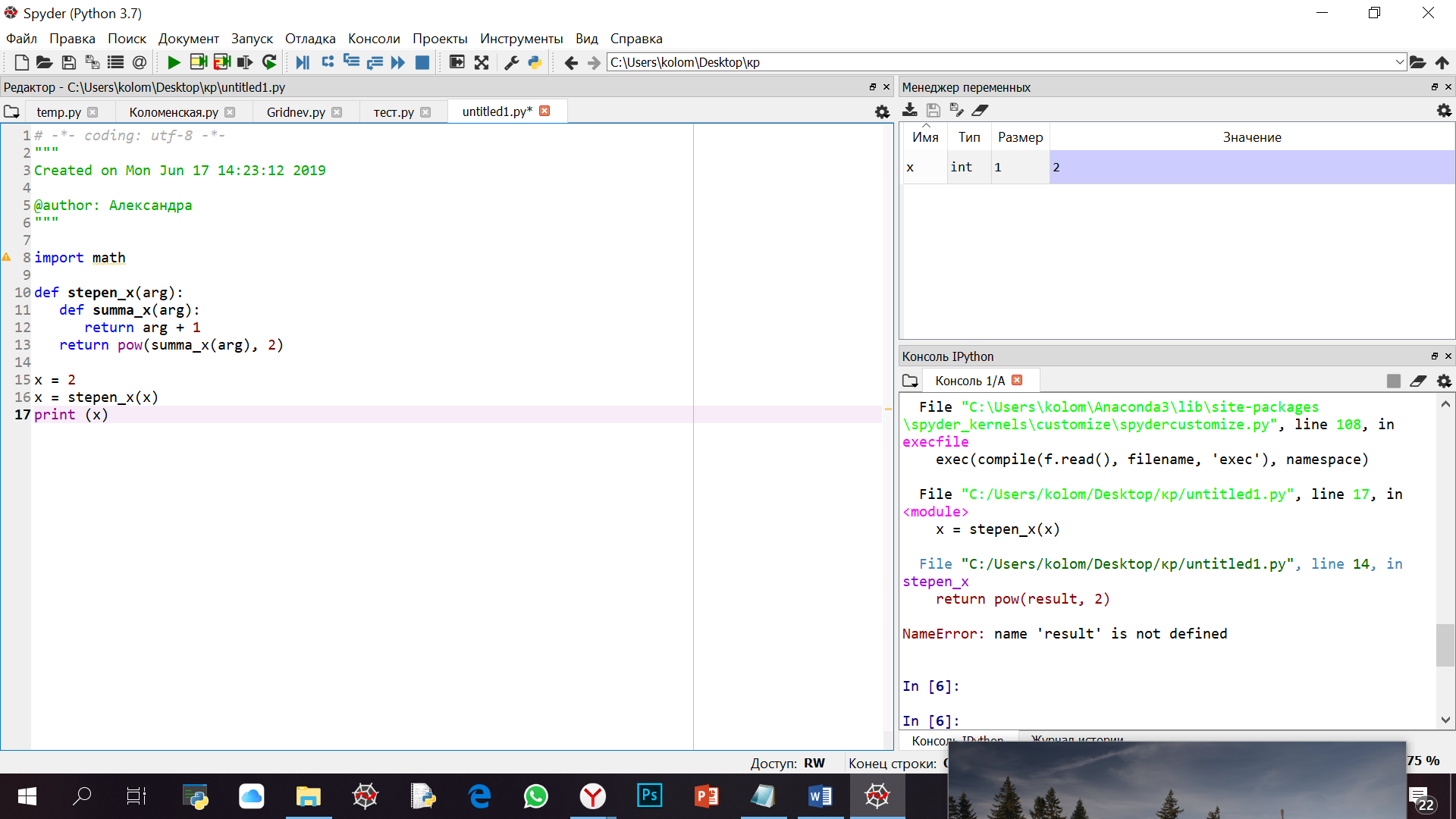
Одну функцию можно вложить в другую функцию, причем уровень вложенности неограничен. В этом случае вложенная функция получает свою собственную локальную область видимости и имеет доступ к идентификаторам внутри функции-родителя.

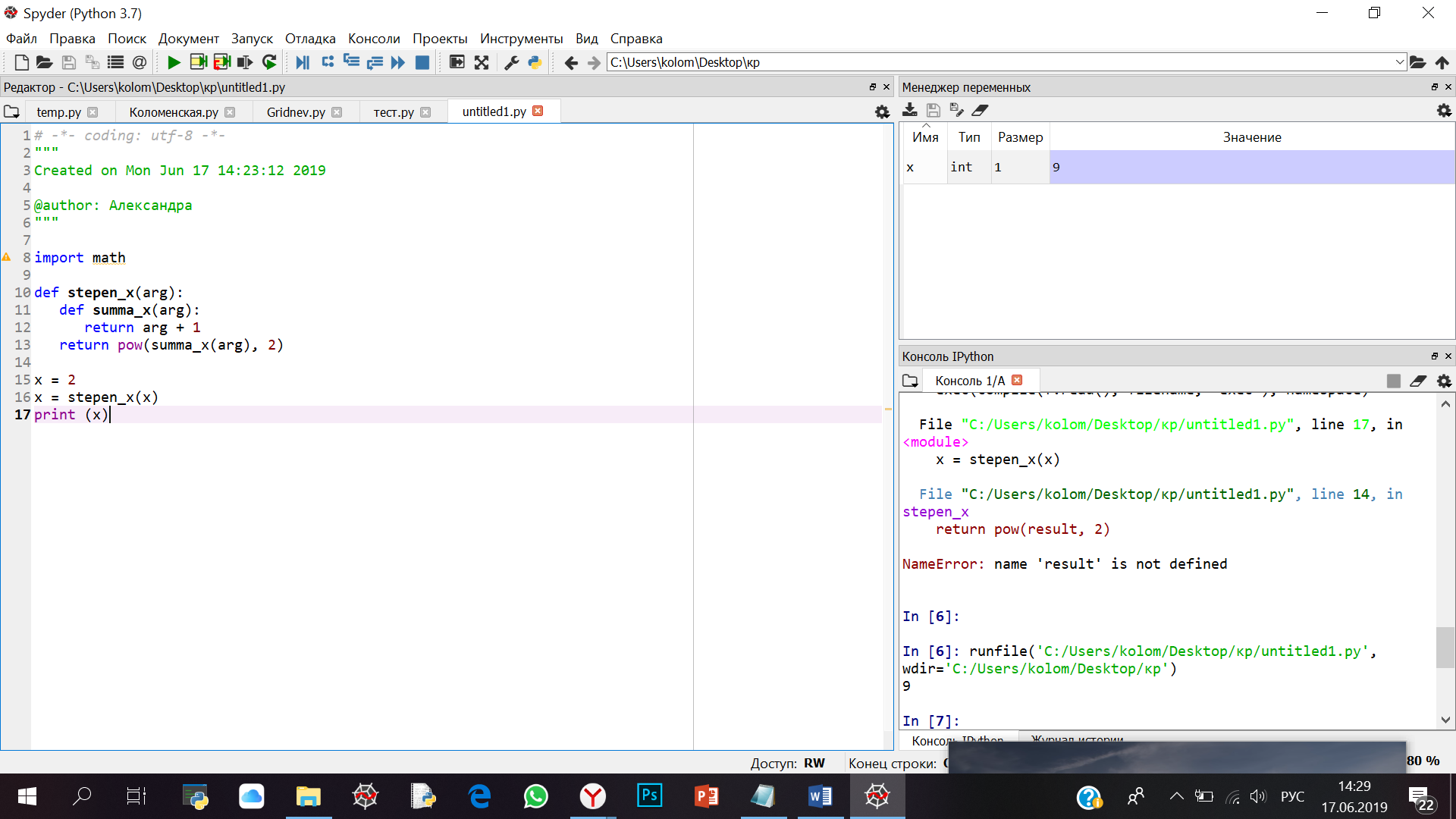
*Использование внутренних функций* для избавления от повторяющихся фрагментов кода в одной функции (применение принципа DRY).

Don’trepeatyourself, DRY (рус. не повторяйся) — это принцип разработки программного обеспечения, нацеленный на снижение повторения информации различного рода, особенно в системах со множеством слоёв абстрагирования. Принцип DRY формулируется как: «Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы»

*Пример 1:* функция f2 вложена в f1, при вызове f1 мы получаем то, что было в f2.



Пример 2: функция подсчета (х+1)^2. Во вложенной функции считается х+1, а в основной функции результат вложенной возводится в квадрат**.**



***Замыкания.***

(Умные слова, которые не всем могут быть понятны): В информатике закрытие является первоклассной функцией со свободными переменными, связанными в лексической среде. Такая функция называется «замкнутой» над ее свободными переменными.

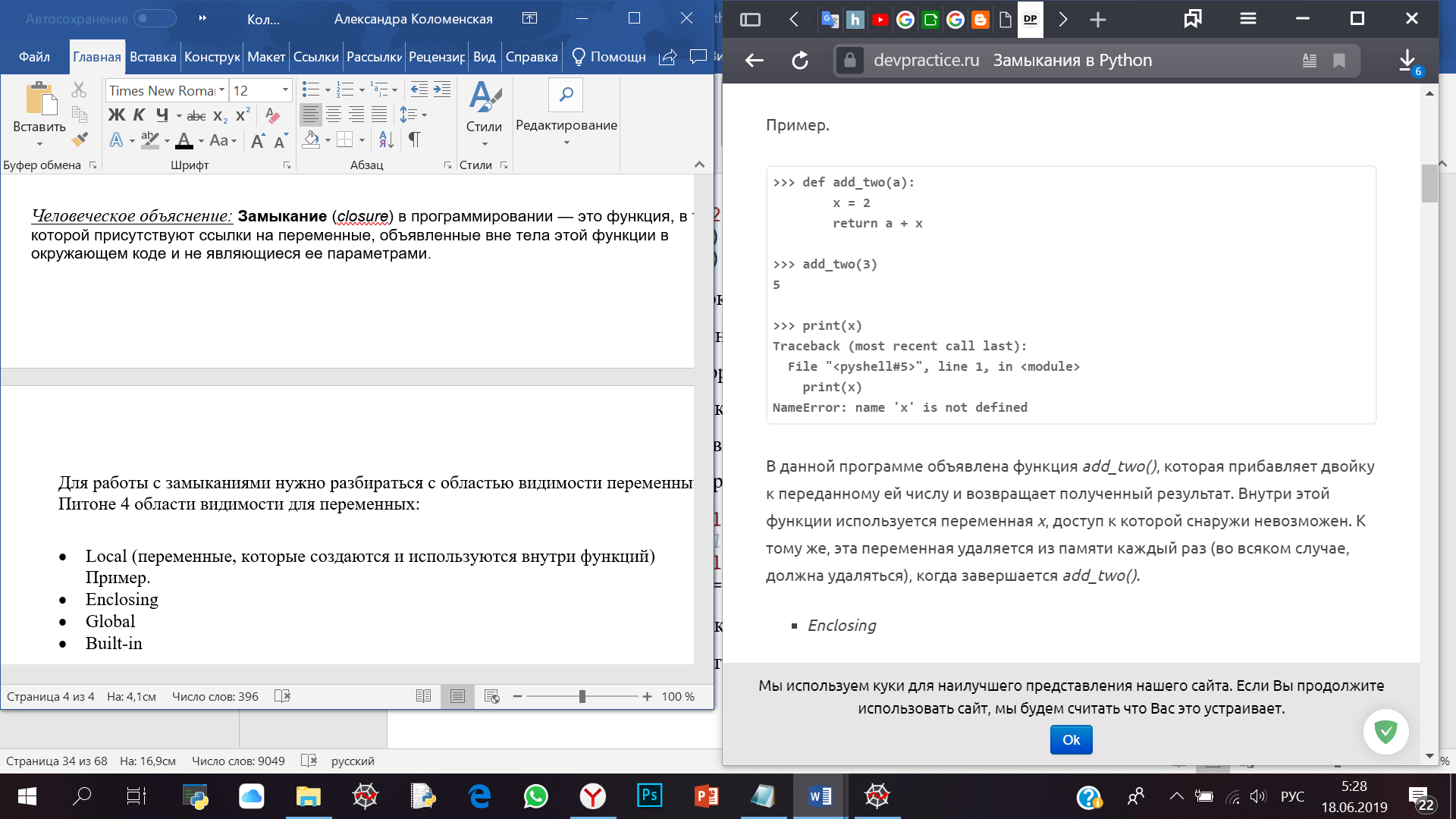
Закрытие – это просто более удобный способ предоставления функции доступа к локальному состоянию.

*Человеческое объяснение:***Замыкание** (*closure*) в программировании — это функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся ее параметрами.

Для работы с замыканиями нужно разбираться с областью видимости переменных. В Питоне 4 области видимости для переменных:

* Local (переменные, которые создаются и используются внутри функций)

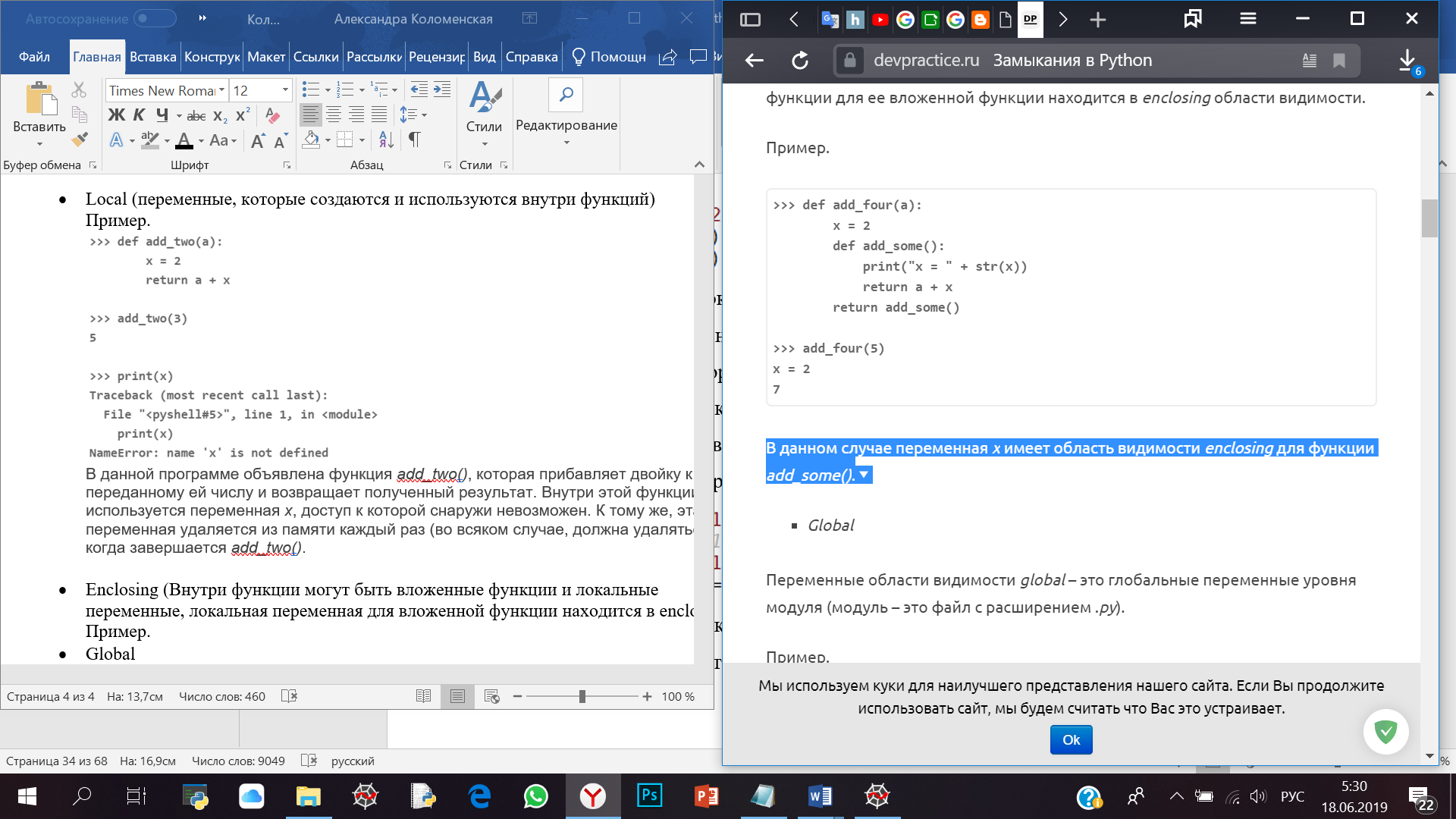
Пример.



В данной программе объявлена функция *add\_two()*, которая прибавляет двойку к переданному ей числу и возвращает полученный результат. Внутри этой функции используется переменная *x*, доступ к которой снаружи невозможен. К тому же, эта переменная удаляется из памяти каждый раз (во всяком случае, должна удаляться), когда завершается *add\_two()*.

* Enclosing (Внутри функции могут быть вложенные функции и локальные переменные, локальная переменная для вложенной функции находится в enclosing)

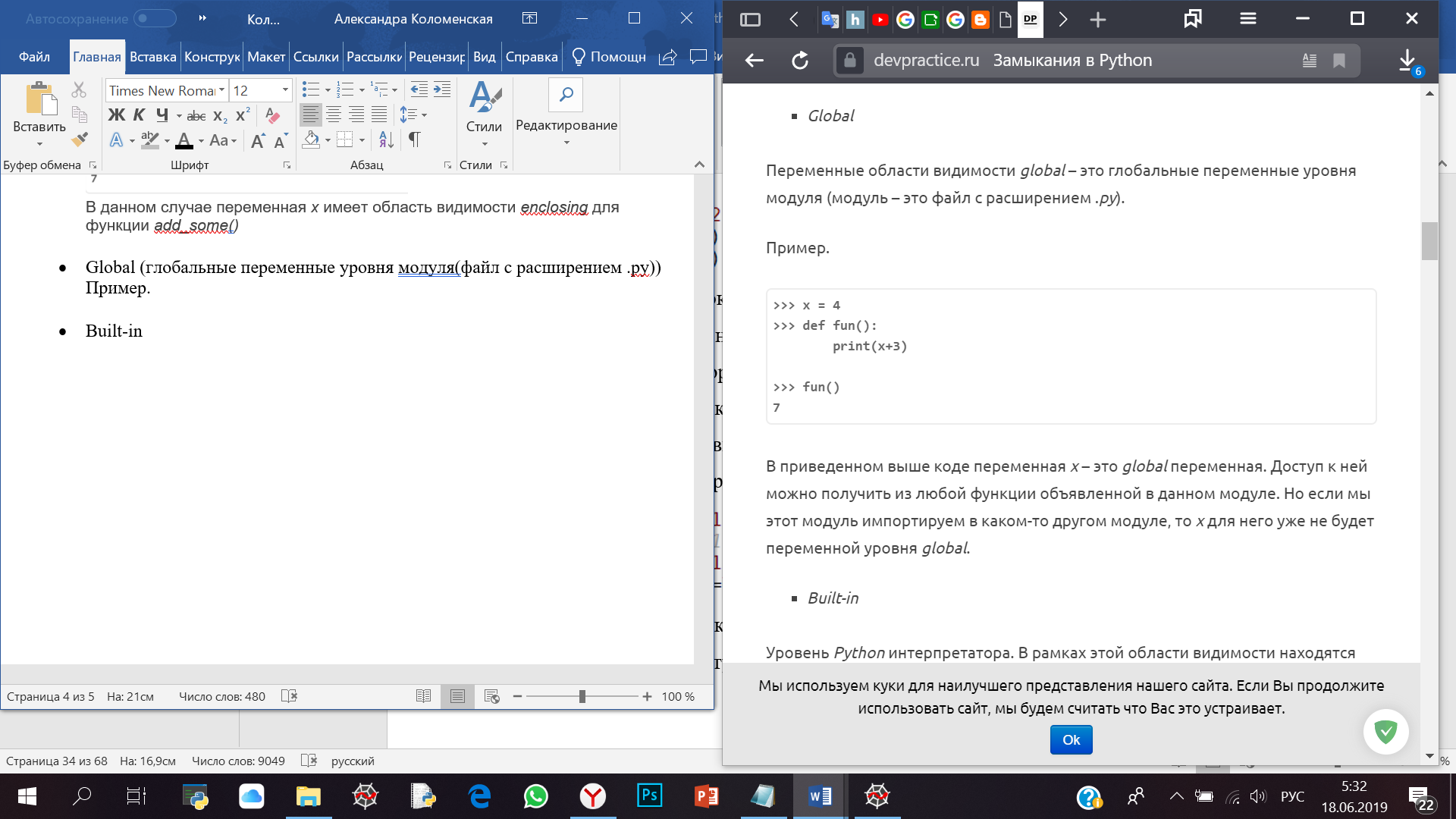
Пример.



В данном случае переменная *x* имеет область видимости *enclosing*для функции *add\_some()*

* Global (глобальные переменные уровня модуля(файл с расширением .py))

Пример.



В приведенном выше коде переменная *x* – это *global*переменная. Доступ к ней можно получить из любой функции объявленной в данном модуле. Но если мы этот модуль импортируем в каком-то другом модуле, то *x* для него уже не будет переменной уровня *global*.

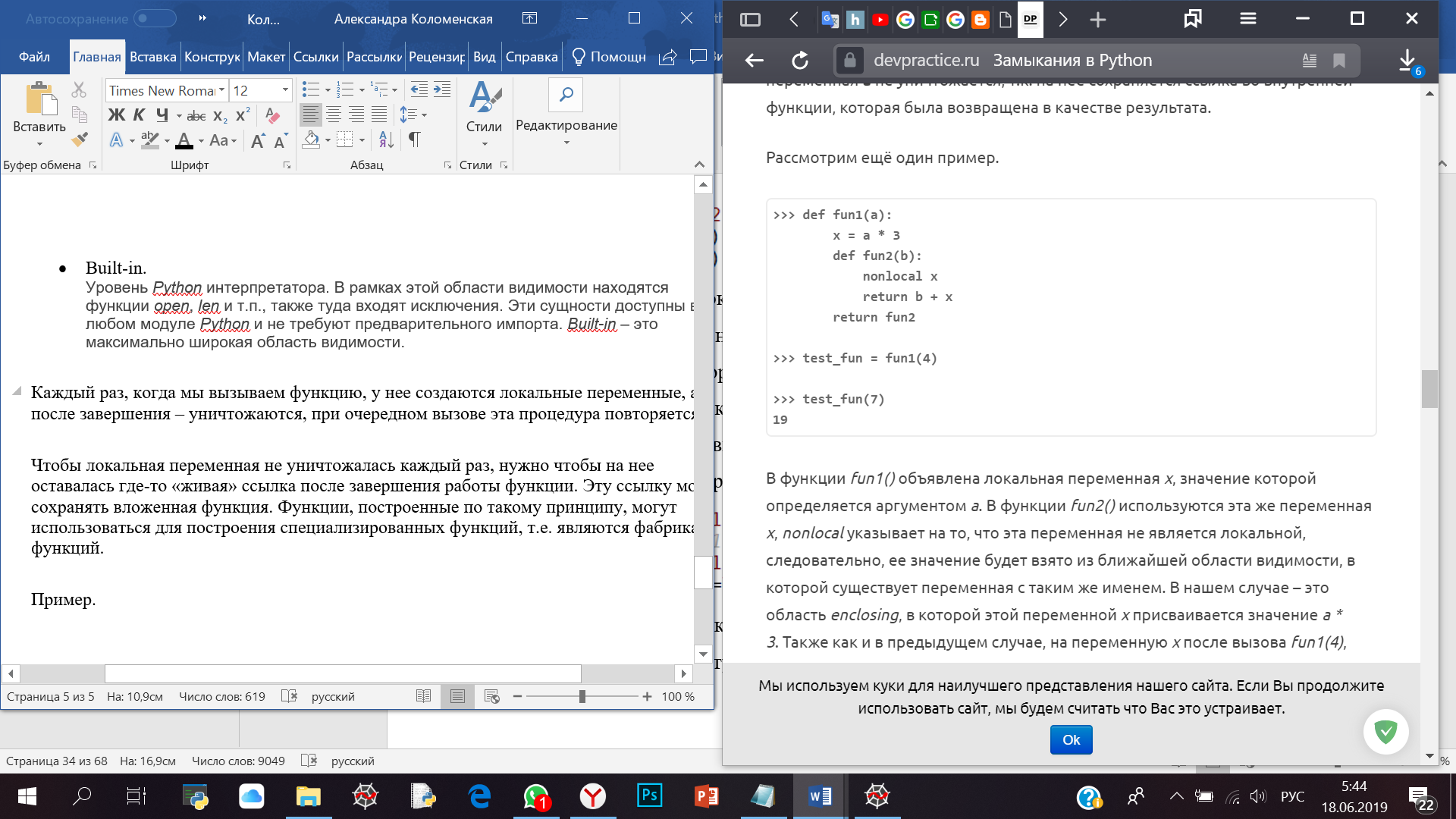
* Built-in.

Уровень *Python*интерпретатора. В рамках этой области видимости находятся функции *open*, *len*и т.п., также туда входят исключения. Эти сущности доступны в любом модуле *Python*и не требуют предварительного импорта. *Built-in*– это максимально широкая область видимости.

Каждый раз, когда мы вызываем функцию, у нее создаются локальные переменные, а после завершения – уничтожаются, при очередном вызове эта процедура повторяется.

Чтобы локальная переменная не уничтожалась каждый раз, нужно чтобы на нее оставалась где-то «живая» ссылка после завершения работы функции. Эту ссылку может сохранять вложенная функция. Функции, построенные по такому принципу, могут использоваться для построения специализированных функций, т.е. являются фабриками функций.

Пример.



В функции *fun1()* объявлена локальная переменная *x*, значение которой определяется аргументом *a*. В функции *fun2()* используются эта же переменная *x*, *nonlocal*указывает на то, что эта переменная не является локальной, следовательно, ее значение будет взято из ближайшей области видимости, в которой существует переменная с таким же именем. В нашем случае – это область *enclosing*, в которой этой переменной *x*присваивается значение *a \* 3* на переменную *x* после вызова *fun1(4)*

***Свойства замыкания***

Вот как определяется “свойство замыкания” в книге “Структура и интерпретация компьютерных программ” Айбельсона Х., Сассмана Д.Д.: “В общем случае, операция комбинирования объектов данных обладает свойством замыкания в том случае, если результаты соединения объектов с помощью этой операции сами могут соединяться этой же операцией”.

Это свойство позволяет строить иерархические структуры данных. Покажем это на примере кортежей в *Python*.

*Пример.*

Создадим функцию*tpl()*, которая на вход принимает два аргумента и возвращает кортеж. Эта функция реализует операцию “объединения элементов в кортеж”.

**>>>tpl = lambda a, b: (a, b)**

Если мы передадим в качестве аргументов числа, то, получим простой кортеж.

**>>> a = tpl(1, 2)**

**>>> a**

**(1, 2)**

Эту операцию можно производить не только над числами, но и над сущностями, ей же и порожденными.

**>>> b = tpl(3, a)**

**>>> b**

**(3, (1, 2))**

**>>> c = tpl(a, b)**

**>>> c**

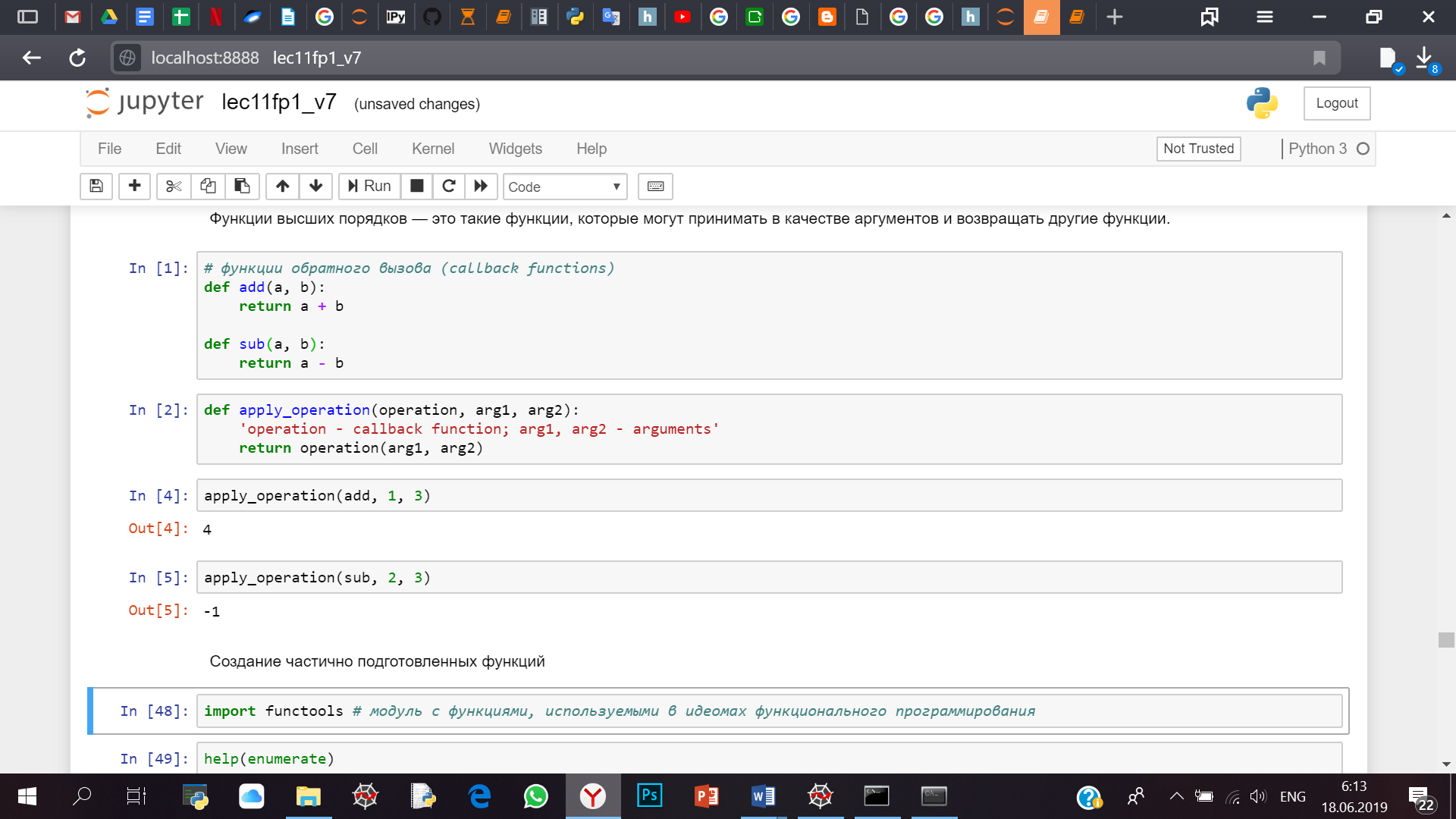
**((1, 2), (3, (1, 2)))**

Таким образом, в нашем примере кортежи оказались замкнуты относительно операции объединения *tpl*.

**24. Функции высшего порядка и декораторы в Python.**

*Функции высших порядков* — это такие функции, которые могут принимать в качестве аргументов и возвращать другие функции.

*Пример.*



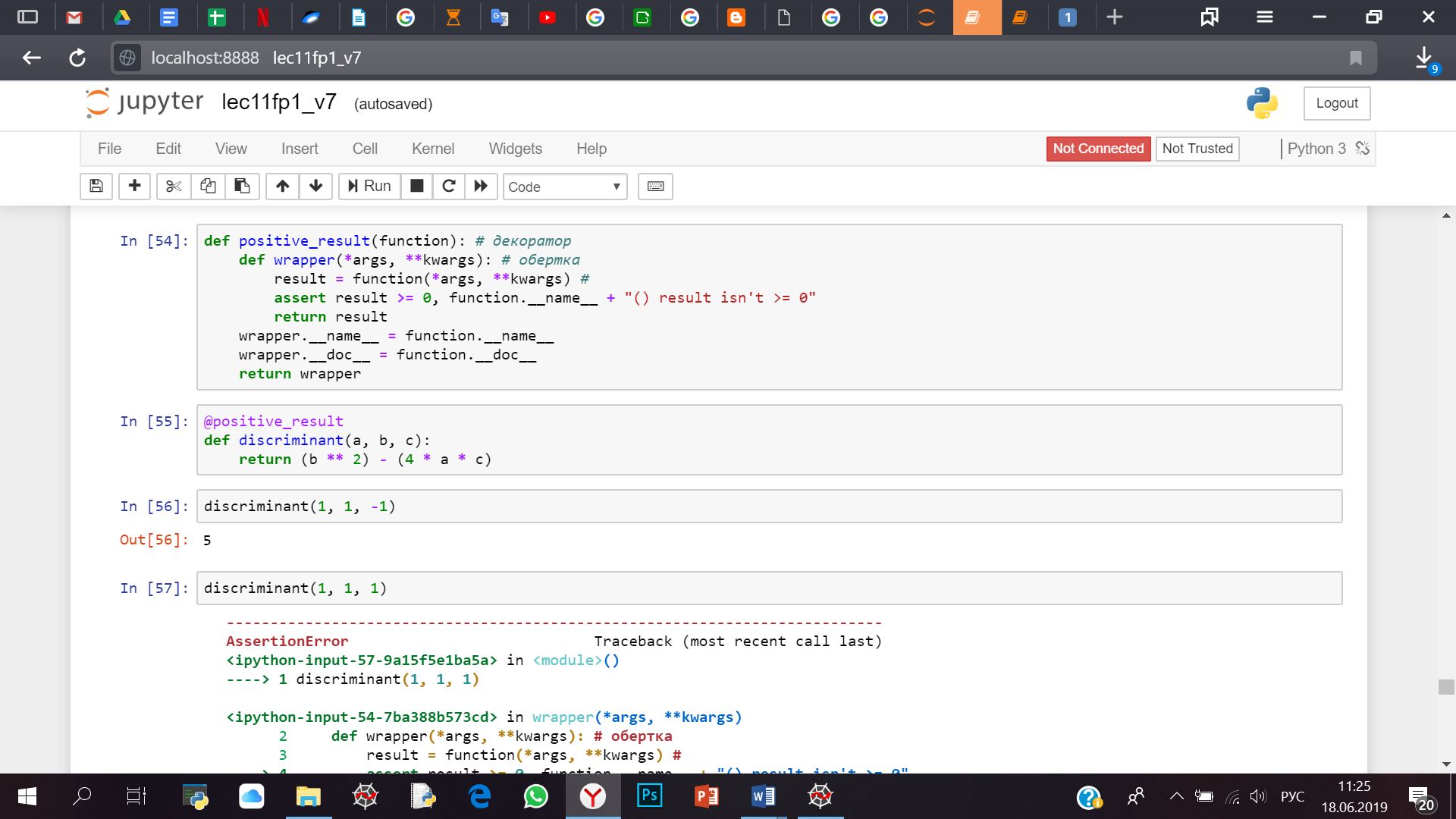
ВIn[4], in[5] функцияapply\_operationкакаргументпринимает функции сложения вычитания, созданные в in[1].

### ***Декораторы***

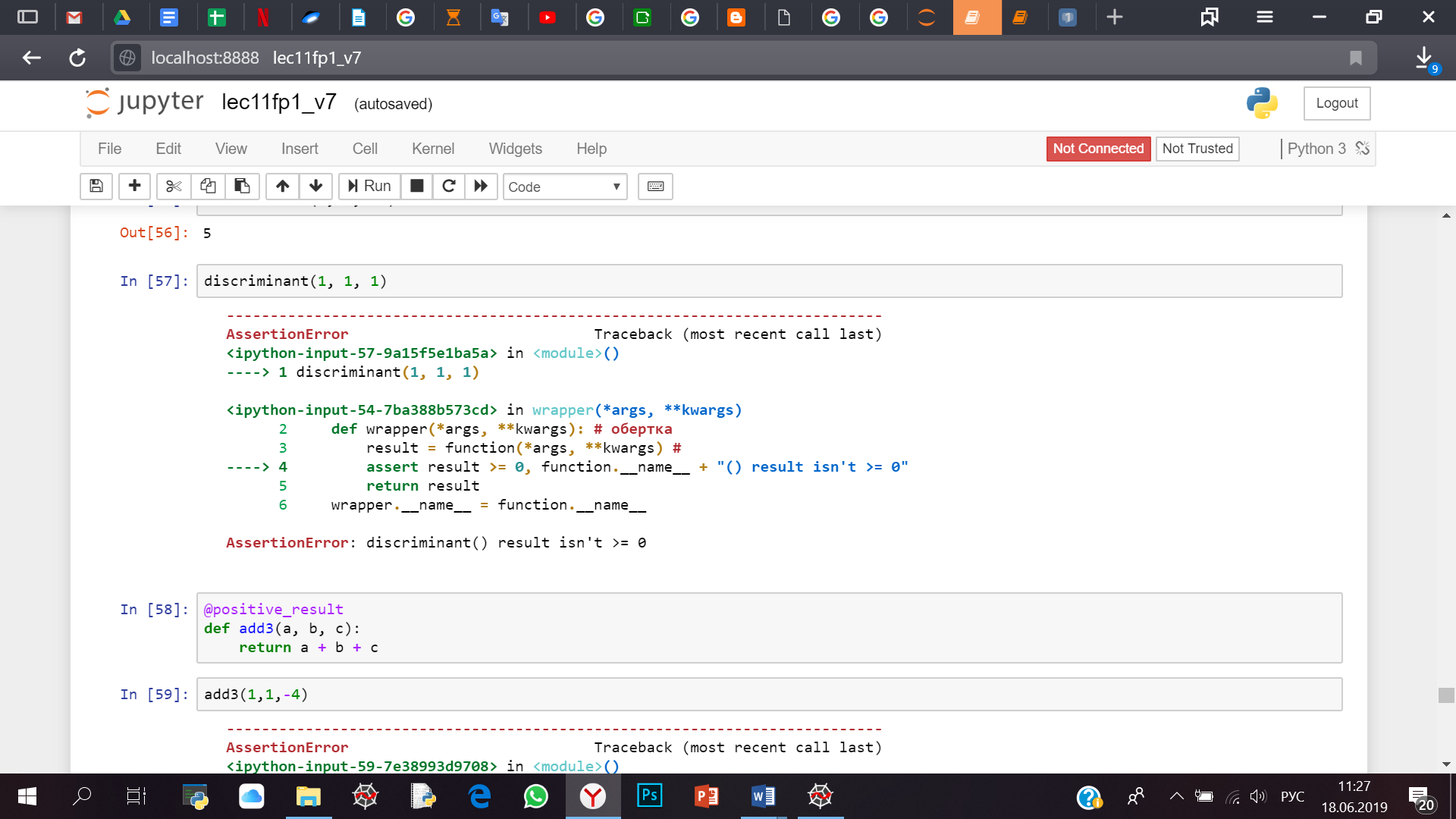
Декоратор - это функция, которая принимает функцию или метод в качестве единственного аргумента и  
возвращает новую функцию или метод, включающую декорированную функцию или метод, с дополнительными функциональными возможностями.

Существуют предопределенные декораторы, например, ©property и @classmethod.

Пример. Предположим, что у нас имеется множество функций, выполняющих вычисления, и некоторые из нихвсегда должны возвращать не отрицательный результат. Мы могли бы добавить в каждую из таких функций инструкцию assert, но использование декораторов проще и понятнее.



У нас декоратор, который считает дискриминант по 3 коэффициентам a,b и с. В случае, если он положительный, то функция срабатывает, если же он отрицательный. Например, при вводе (1,1,1), то получится следующий вывод:



Декоратор проверил результат функции, он получился меньше нуля, поэтому вывел ошибку в функции.

### **Объекты-декораторы**

Напоследок стоит упомянуть, что не только функции, а любые вызываемые объекты могут быть декоратором. Экземпляры классов/объекты с методом \_\_call\_\_() тоже можно вызывать, поэтому их можно использовать в качестве декораторов.

Доп.факты:

* Декораторы не обязательно должны быть функциями, это может быть любой вызываемый объект.
* Декораторы не обязаны возвращать функции, они могут возвращать что угодно. Но обычно мы хотим, чтобы декоратор вернул объект того же типа, что и декорируемый объект. Пример:
* >>>defdecorator(func):
* ...return'sumit'
* ...
* >>> @decorator
* ...defhello\_world():
* ...print('helloworld')
* ...
* >>>hello\_world

'sumit'

* Также декораторы могут принимать в качестве аргументов не только функции.
* Также стоит обратить внимание на [functools.wraps()](https://docs.python.org/3.7/library/functools.html#functools.wraps) — функцию, которая помогает сделать декорируемую функцию похожей на исходную, делая такие вещи, как сохранение doctstring исходной функции.

**26. Итераторы в Python: встроенные итераторы, создание собственных итераторов, типичные способы обхода итераторов и принцип их работы. Встроенные функции для работы с итераторами и возможности модуля itertools.**

Применять для обработки данных явные последовательности не всегда эффективно, так как на хранение временных данных может тратиться много оперативной памяти. Более эффективным решением представляется использование **итераторов** - специальных объектов, обеспечивающих последовательный доступ к данным контейнера. Если в выражении есть операции с итераторами вместо контейнеров, промежуточные данные не будут требовать много места для хранения - ведь они запрашиваются по мере необходимости для вычислений. При обработке данных с использованием итераторов память будет требоваться только для исходных данных и результата, да и то необязательно вся сразу - ведь данные могут читаться и записываться в файл на диске.

Итераторы можно применять вместо последовательности в операторе for. Более того, внутренне оператор for запрашивает от последовательности ее итератор. Объект файлового типа тоже (построчный) итератор, что позволяет обрабатывать большие файлы, не считывая их целиком в память.

Там, где требуется итератор, можно использовать последовательность.

Работа с итераторами рассматривается в разделе, посвященном функциональному программированию, так как итераторами удобно манипулировать именно в функциональном стиле.

Использовать итератор можно и "вручную". Любой объект, поддерживающий интерфейс итератора, имеет метод next(), который при каждом вызове выдает очередное значение итератора. Если больше значений нет, возбуждается исключение StopIteration. Для получения итератора по некоторому объекту необходимо прежде применить к этому объекту функцию iter() (цикл for делает это автоматически).

В Python имеется модуль **itertools**, который содержит набор функций, комбинируя которые, можно составлять достаточно сложные схемы обработки данных с помощью итераторов. Далее рассматриваются некоторые функции этого модуля.

**Функция iter()**

Эта функция имеет два варианта использования. В первом она принимает всего один аргумент, который должен "уметь" предоставлять свой итератор. Во втором один из аргументов - функция без аргументов, другой - стоповое значение. Итератор вызывает указанную функцию до тех пор, пока та не возвратит стоповое значение. Второй вариант встречается много реже первого и обычно внутри метода класса, так как сложно порождать значения "на пустом месте":

it1 = iter([1, 2, 3, 4, 5])

defforit(mystate=[]):

iflen(mystate) < 3:

mystate.append(" ")

return " "

it2 = iter(forit, None)

print [x for x in it1]

print [x for x in it2]

Примечание: Если функция не возвращает значения явно, она возвращает None, что и использовано в примере выше.

**Функция enumerate()**

Эта функция создает итератор, нумерующий элементы другого итератора. Результирующий итератор выдает кортежи, в которых первый элемент - номер (начиная с нуля), а второй - элемент исходной последовательности:

>>>print [x for x inenumerate("abcd")]

[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]

**Функция sorted()**

Эта функция, появившаяся в Python 2.4, позволяет создавать итератор, выполняющий сортировку:

>>>sorted('avdsdf')

['a', 'd', 'd', 'f', 's', 'v']

Далее рассматриваются функции модуля **itertools**.

**Функция itertools.chain()**

Функция chain() позволяет сделать итератор, состоящий из нескольких соединенных последовательно итераторов. Итераторы задаются в виде отдельных аргументов. Пример:

fromitertoolsimportchain

it1 = iter([1,2,3])

it2 = iter([8,9,0])

for i inchain(it1, it2):

print i,

даст в результате 1 2 3 8 9 0

**Функция itertools.repeat()**

Функция repeat() строит итератор, повторяющий некоторый объект

заданное количество раз:

for i in itertools.repeat(1, 4):

print i,

1 1 1 1

**Функцияitertools.count()**

Бесконечныйитератор, дающийцелыечисла, начинаясзаданного:

for i in itertools.count(1):

print i,

if i> 100:

break

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101

**Функцияitertools.cycle()**

Можно бесконечно повторять и некоторую последовательность (или значения другого итератора) с помощью функции cycle():

tango = [1, 2, 3]

for i in itertools.cycle(tango):

print i,

1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 . . .

**Функцииitertools.imap(), itertools.starmap() иitertools.ifilter()**

Аналогами map() и filter() в модуле itertools являются imap() и ifilter(). Отличие **imap()** от map() в том, что вместо значения от преждевременно завершившихся итераторов объект None не подставляется. Пример:

for i inmap(lambda x, y: (x,y), [1,2], [1,2,3]):

print i,

(1, 1) (2, 2) (None, 3)

fromitertoolsimportimap

for i inimap(lambda x, y: (x,y), [1,2], [1,2,3]):

print i,

(1, 1) (2, 2)

Здесь следует заметить, что обычная функция map() нормально воспринимает итераторы в любом сочетании с итерабельными (поддающимися итерациям) объектами: for i inmap(lambda x, y: (x,y), iter([1,2]), [1,2,3]): print i, (1, 1) (2, 2) (None, 3)

**Функция itertools.starmap()** подобна itertools.imap(), но имеет всего два аргумента. Второй аргумент - последовательность кортежей, каждый кортеж которой задает набор параметров для функции (первого аргумента):

>>> from itertools import starmap

>>> for i in starmap(lambda x, y: str(x) + y, [(1,'a'), (2,'b')]):

... print i,

...

1a 2b

**Функция ifilter()** работает как filter(). Кроме того, в модуле itertools есть функция ifilterfalse(), которая как бы добавляет отрицание к значению функции:

for i in ifilterfalse(lambda x: x > 0, [1, -2, 3, -3]):

print i,

-2 -3

**Функции itertools.takewhile() и itertools.dropwhile()**

Некоторую новизну вносит другой вид фильтра: takewhile() и его "отрицательный" аналог dropwhile(). Следующий пример поясняет их принцип действия:

for i in takewhile(lambda x: x > 0, [1, -2, 3, -3]):

print i,

print

for i in dropwhile(lambda x: x > 0, [1, -2, 3, -3]):

print i,

1

-2 3 -3

Таким образом, takewhile() дает значения, пока условие истинно, а остальные значения даже не берет из итератора (именно не берет, а не высасывает все до конца!). И, наоборот, dropwhile() ничего не выдает, пока выполняется условие, зато потом выдает все без остатка.

**Функция itertools.izip()**

Функция izip() аналогична встроенной zip(), но не тратит много памяти на построение списка кортежей, так как итератор выдает их строго по требованию.

**Функция itertools.groupby()**

Эта функция дебютировала в Python 2.4. Функция принимает два аргумента: итератор (обязательный) и необязательный аргумент - функцию, дающую значение ключа: groupby(iterable[, func]). Результатом является итератор, который возвращает двухэлементный кортеж: ключ и итератор по идущим подряд элементам с этим ключом. Если второй аргумент опущен, элемент итератора сам является ключом. В следующем примере группируются идущие подряд положительные и отрицательные элементы:

import itertools, math

lst = map(lambda x: math.sin(x\*.4), range(30))

for k, i in itertools.groupby(lst, lambda x: x > 0):

print k, lst(i)

**Функция itertools.tee()**

Эта функция тоже появилась в Python 2.4. Она позволяет клонировать итераторы. Первый аргумент - итератор, подлежащий клонированию. Второй ( N ) -- количество необходимых копий. Функция возвращает кортеж из N итераторов. По умолчанию N=2. Функция имеет смысл, только если итераторы задействованы более или менее параллельно. В противном случае выгоднее превратить исходный итератор в список.

# 27. Функции генераторы и выражения генераторы: создание и применение в Python.

**Функции-генераторы** - функция, возвращающая подвид итератора, генерирующий значения. Генераторы - это итераторы, вид итерации, который **вы можете повторять только один раз**.

Выглядят функции-генераторы также как и [обычные](https://pythonz.net/references/named/fynktsiya/), но содержат выражения с ключевым словом yield для последовательного генерирования значений, которые могут быть использованы в циклах [for in](https://pythonz.net/references/named/for-in-else/), либо их получения при помощи функции [next()](https://pythonz.net/references/named/next/).

На каждой yield работа функции временно приостанавливается, при этом сохраняется состояние исполнения, включая локальные переменные, указатель на текущую инструкцию, внутренний стек и состояние обработки исключения. При последующем обращении к итератору генератора (при вызовах его методов) функция продолжает своё исполнение с места, на котором была приостановлена. Этим функции-генераторы отличаются от обычных функций, при вызове которых исполнение всякий раз начинается с начала.

def generator():

for i in (1, 2, 3):

yield i

g = generator()

print(g)

for i in g:

print(i) *#123*

**Выражения-генераторы** очень похожи на списки. Основное различие заключается в том, что он не создает сразу полный набор результатов; он создает объект-генератор, который затем может быть повторен. Генераторы обычно используются, когда надо единожды пройтись по итерируемому объекту.

Пример выражения-генератора:

my\_sum = sum(i\*i **for** i **in** range(10)) *# сумма квадратов 0, 1, 4, ... 81*

my\_sum *# 285*

**28. Специфика массивов, как структур данных. Динамические массивы – специфика работы, сложность операций. Специфика работа с array в Python.**

## Массивы (списки)

Массивы (списки) в Питоне, как и в других языках программирования, — это определенное количество элементов одного типа, которые имеют общее имя, и каждый элемент имеет свой индекс — порядковый номер.

L - список целых чисел:

L = [25, 755, -40, 57, -41]

**Однако, в языке Python отсутствует такая структура, как «массив». Для работы с массивами используются *списки*.**

Списки являются упорядоченными последовательностями, которые состоят из различных объектов (значений, данных), заключающихся в квадратные скобки [ ] и отделяющиеся друг от друга с помощью запятой.

|  |
| --- |
| list('текст') *# ['т', 'е', 'к', 'с', 'т']* |

## Создание списков на Python

Создать список можно несколькими способами. Рассмотрим их.

#### 1. Получение списка через присваивание конкретных значений

* Так выглядит в коде Python пустой список:

|  |
| --- |
| s = [] *# Пустой список* |

* Примеры создания списков со значениями:

|  |
| --- |
| l = [25, 755, -40, 57, -41] *# список целых чисел*  l = [1.13, 5.34, 12.63, 4.6, 34.0, 12.8] *# список из дробных чисел*  l = ["Sveta", "Sergei", "Ivan", "Dasha"] *# список из строк*  l = ["Москва", "Иванов", 12, 124] *# смешанный список*  l = [[0, 0, 0], [1, 0, 1], [1, 1, 0]] *# список, состоящий из списков*  l = ['s', 'p', ['isok'], 2] *# список из значений и списка* |

* Списки можно складывать (конкатенировать) с помощью знака «+»:

|  |
| --- |
| l = [1, 3] + [4, 23] + [5]    *# Результат:*  *# l = [1, 3, 4, 23, 5]* |

#### 2. Списки при помощи функции List()

Получаем список при помощи функции**List()**

|  |
| --- |
| l = list ('spisok')  *# 'spisok' - строка*  **print**(l) *#['s', 'p', 'i', 's', 'o', 'k'] - результат - список* |

#### 3. Создание списка при помощи функции Split()

* Используя функцию **split** в Питон можно получить из строки список. Рассмотрим пример:

|  |
| --- |
| stroka ="Hello, world"  *# stroka - строка*  lst=stroka.split(",") *# lst - список*  lst *# ['Hello', ' world']* |

#### 4. Генераторы списков

* В python создать список можно также при помощи генераторов, — это довольно-таки новый метод:
* Первый простой способ.

Сложение одинаковых списков заменяется умножением:

|  |
| --- |
| *# список из 10 элементов, заполненный единицами*  l = [1]\*10  *# список l = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]* |

* Второй способ сложнее.

|  |
| --- |
| l = [i **for** i **in** range(10)]  *# список l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]* |

или такой пример:

|  |
| --- |
| c = [c \* 3 **for** c **in** 'list']  **print** (c) *# ['lll', 'iii', 'sss', 'ttt']* |

В Python a list представляет собой динамический массив. Вы можете создать такой способ:

lst = [] # Declares an empty list named lst

Или вы можете заполнить его пунктами:

lst = [1,2,3]

Вы можете добавлять элементы, используя "append":

lst.append('a')

Вы можете перебирать элементы списка с помощью цикла for:

for item in lst:

# Do something with item

Или, если вы хотите отслеживать текущий индекс:

for idx, item in enumerate(lst):

# idx is the current idx, while item is lst[idx]

Чтобы удалить элементы, вы можете использовать команду del или функцию удаления, как в:

del lst[0] # Deletes the first item

lst.remove(x) # Removes the first occurence of x in the list

В python динамический массив является "массивом" из модуля массива. Например.

from array import array

x = array('d') #'d' denotes an array of type double

x.append(1.1)

x.append(2.2)

x.pop() # returns 2.2

Этот тип данных по существу является перекрестком между встроенным типом "list" и типом nump "ndarray". Как и ndarray, элементы в массивах являются типами C, указанными при инициализации. Они не являются указателями на объекты python

**29. Абстрактная структура данных стек и очередь: базовые и расширенные операции, их сложность.**

Очереди – это способ более эффективной организации данных. Пример очереди из реальной жизни – вереница людей в баре. Если бы каждый самостоятельно привлекал внимание бармена, это привело бы к сплошному хаосу. Выход из ситуации – выстроить всех в один ряд.

Чтобы реализовать очередь в Python, нужно создать класс и инициализировать его пустым списком. Одним из свойств очереди является то, что первые элементы в очереди выходят первыми (FIFO – first-in, first-out). Снова вернемся к нашему примеру с баром: клиенты, попавшие в очередь раньше, будут обслужены раньше.

Метод «add», используемый для представления очереди, добавляет элемент в конец строки. Чтобы добавить элемент в начало строки, используется метод **insert()** с нулевым индексом.

|  |
| --- |
| 1 class Queue:  2 def \_\_init\_\_(self):  3 self.people = []  4 def add(self, person):  5 self.people.insert(0, person) |

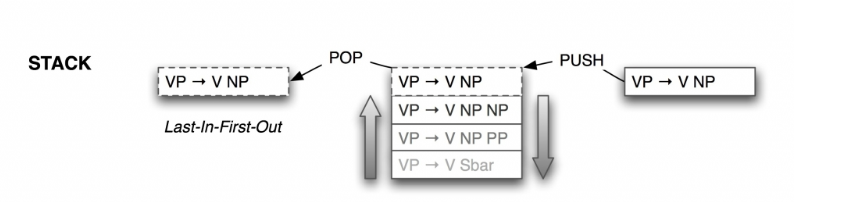
Для удаления элемента из очереди используется метод **pop()**:

|  |
| --- |
| 1 class Queue:  2 def \_\_init\_\_(self):  3 self.people = []  4 def add(self, person):  5 self.people.insert(0, person)  6 def remove(self):  7 return self.people.pop() |

Чтобы узнать размер нашей очереди, можно воспользоваться встроенным в Python методом **len()**:

|  |
| --- |
| 1 class Queue:  2 def \_\_init\_\_(self):  3 self.people = []  4 def add(self, person):  5 self.people.insert(0, person)  6 def remove(self):  7 return self.people.pop()  8 def size(self):  9 return len(self.people) |

Стек.



Основным отличием стека от очереди, является способ организации и манипулирования данными. В стеке используется алгоритм LIFO (last-in, first-out) – последний добавленный элемент извлекается первым.

В Python создать класс стека довольно просто: инициализируется пустой список и создаются методы, позволяющие добавлять и удалять элементы с одной стороны.

Метод pop() удаляет элемент с вершины стека и возвращает его.

Метод push() добавляет элемент на вершину стека.

|  |
| --- |
| 1 class Stack:  2 def \_\_init\_\_(self):  3 self.elements = []  4 def push(self, element):  5 self.elements.append(element)  6 def pop(self):  7 return self.elements.pop() |

Важно отметить, что мы можем выбрать реализацию стека через список, где вершиной считается первый, а не последний элемент. В этом случае предыдущие методы append и pop работать не будут. Мы должны будем явно использовать pop и insert для позиции с индексом 0 (первый элемент в списке).

|  |  |
| --- | --- |
| 1 | class Stack: |
| 2 | def \_\_init\_\_(self): |
| 3 | self.items = [] |
| 4 |  |
| 5 | def isEmpty(self): |
| 6 | return self.items == [] |
| 7 |  |
| 8 | def push(self, item): |
| 9 | self.items.insert(0,item) |
| 10 |  |
| 11 | def pop(self): |
| 12 | return self.items.pop(0) |
| 13 |  |
| 14 | def peek(self): |
| 15 | return self.items[0] |
| 16 |  |
| 17 | def size(self): |
| 18 | return len(self.items) |
| 19 |  |
| 20 | s = Stack() |
| 21 | s.push('hello') |
| 22 | s.push('true') |
| 23 | print(s.pop()) |

Эта возможность изменять физическое воплощение абстрактного типа данных при поддержке логических характеристик - пример того, как работает абстракция. Однако, даже если стек будет вести себя аналогично, рассмотрение производительности этих двух реализаций покажет их несомненное различие. Напомним, что операции append и pop обе являются О(1). Это означает, что первая реализация будет выполнять добавление и выталкивание за постоянное время, независимо от количества элементов в стеке. Производительность второго варианта страдает, поскольку и insert(0), и pop(0) для стека, размером n, являются O(n). Очевидно, что даже если реализации логически эквивалентны, то при тестировании они будут иметь очень разные затраты по времени.

**30. Специфика реализации и скорости основных операций в очереди  
на базе массива и связанного списка**

Очередь – это набор объектов, который поддерживает быструю семантику first-in, first-out (FIFO) для вставки и удаления.

Аналогия очереди first-in, first-out на пальцах

Новые объекты (молекулы воды, шарики для тенниса, и т.д.) помещаются в один конец трубки, и двигаются к другому, когда вы, или что-то другое пропихиваете их вдоль трубки. Пока объекты находятся в очереди (в нашем случае это трубка), вы не можете достать их. Единственный способ взаимодействия с объектами в очереди, это добавить новые объекты в конце очереди (enqueuer) или убрать существующие из начала (dequeuer) трубки.

Очереди похожи на стеки, но отличаются способом **удаления элементов**:

В случае с очередью вы удаляете объект, который был недавно добавлен (по принципу первый вошел – первым вышел first-in, first-out или **FIFO**), в случае сто стеком вы удаляете последний добавленный элемент (последним зашел – первым вышел, last-in, first-out or **LIFO**).

С точки зрения производительности, верная реализация очереди заключается в выделении О(1) времени для вставки и удаления операций. Существует две основные операции, используемые в очереди и они должны выполнять работу быстро и корректно.Очереди Python имеют широкий ряд приложений в алгоритмах, а также все необходимое для составления графиков, а также решения параллельных проблем в программировании. Короткий и удобный алгоритм под названием BFS годится для работы с такими структурами данных как tree и graph.

Алгоритмы планирования часто используют приоритетные очереди изнутри. Существуют определенные виды очередей: вместо получения следующего элемента в соответствии с указанным временем вставки, приоритетная очередь извлекает элемент с самым высоким приоритетом. Приоритет отдельных элементов определяется очередью на основании порядка применяемых к ним ключам. Однако, обычная очередь, не будет переопределять порядок объектов, которые в ней находятся. Вы получаете только то, и в том порядке, в каком вы помещали объекты в очередь (пример с трубкой). Python позволяет выполнить несколько реализаций очереди, каждая из которых имеет небольшие отличия.

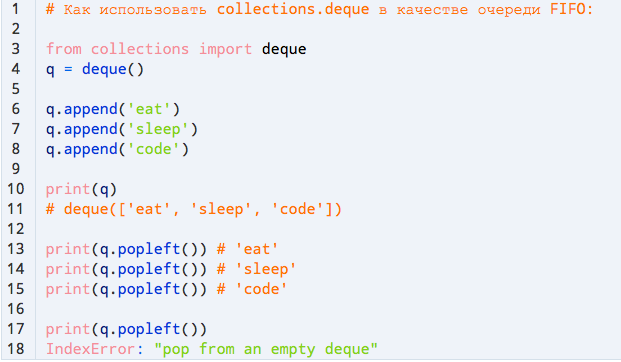
## **Встроенный список**

Мы можем использовать обычный список в качестве очереди, но это не очень эффективно с точки зрения производительности. Списки слишком медленные для этой задачи, так как вставка и удаление элемента с начала требует сдвига всех прочих элементов по одному, на это уходит *О(n)* времени. В связи с этим, я не спешу рекомендовать список в качестве временной очереди в Python (разве что вы имеете дело с небольшим количеством элементов).



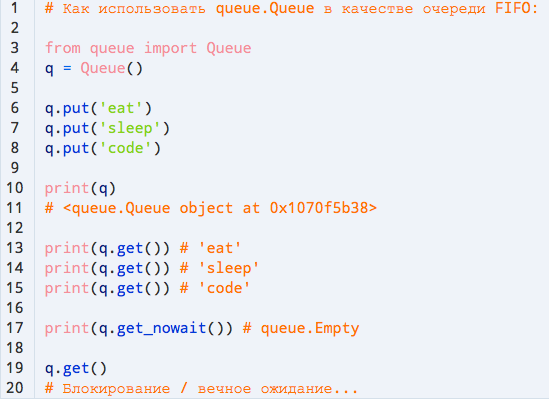
## **Класс collections.deque**

Класс **deque** реализует двухконечную очередь, которая поддерживает добавление и удаление элементов с обоих концов в течение О(1) времени. Объекты **deque** представлены в виде двусвязных списков, что дает им превосходную производительность для входящих и выходящих элементов, но при этом у него **плохая производительность** O(n) при работе со случайно принимаемыми элементами в середине очереди. В связи с тем, что **deque** поддерживает вставку и удаление элементов одинаково хорошо, они могут поддерживать и очереди и стеки. collections.deque это отличное решение, если вы ищите структуру данных очереди в Python в **стандартной библиотеке**.



## **Класс queue.Queue**

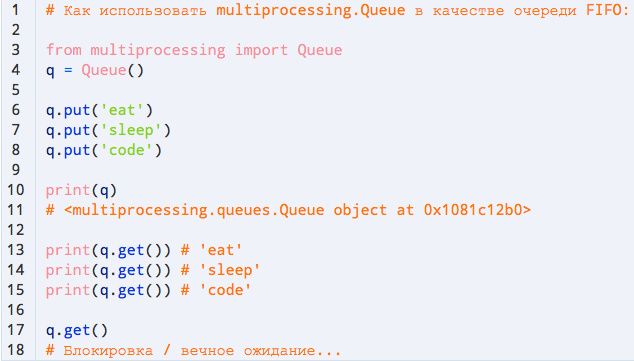
Такая реализация очереди в стандартной библиотеке является **синхронизированной** и предоставляет блокировку семантики для поддержки нескольких конкурирующих производителей и потребителей. Модуль queue содержит несколько других классов, реализующих мульти-продюсера, очереди мульти-потребители , которые полезны в параллельных вычислениях. В зависимости от причины использования, закрытие семантики может быть полезным, или просто привести к ненужному перерасходу. В данном случае, вам лучше будет использовать **collections.deque** как очередь общего назначения.



## **Класс multiprocessing.Queue**

Это реализация очереди с разделенными функциями, которые позволяет находящимся в очереди объектам быть обработанными параллельно несколькими одновременно работающими процессами. Основанная на процессах паралеллизация очень популярна в Python, из-за глобального блокиратора интерпретатора GIL. **multiprocessing.Queue**используется при разделении данных между процессами и может хранить любой пригодный объект.

Как использовать multiprocessing.Queue в качестве очереди FIFO:



**31. Связанные списки: однонаправленные и двунаправленные – принцип реализации. Сравнение скорости выполнения основных операций в связанных списках и в динамическом массиве.**

В информатике, **связный список**— [структура данных](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), состоящая из [узлов](http://ru.wikipedia.org/w/index.php?title=%D0%A3%D0%B7%D0%B5%D0%BB_%28%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29&action=edit&redlink=1), каждый из которых содержит как собственные [данные](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85_%28%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29), так и одну или две [ссылки](http://ru.wikipedia.org/wiki/%D0%A1%D1%81%D1%8B%D0%BB%D0%BA%D0%B0_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) («связки») на следующий и/или предыдущий узел списка. Принципиальным преимуществом перед [массивом](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2) является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

**Основные правила реализации связных списков**  
Список состоит из элементов, называемых узлами (**node**). Первый узел списка называется «головным» (**head**), а последний - «хвостовым» (**tail**).

На рисунке изображен двойной связный список.

****

Каждый элемент состоит из 3-х полей, два из которых являются указателями на предыдущий или следующий узел. Элемент может указывать и более чем на два узла, и в этом случае список называется многосвязным.

Динамический массив — это массив, который может менять свой размер. Если элемент добавляется в середину динамического массива, то происходит перераспределение элементов, находящихся справа от него, так как все элементы массива должны быть расположены в памяти строго по порядку. Поэтому вставка элемента в динамический массив требует дополнительных ресурсов, если элемент добавляется не в конец массива.

Преимущество связного списка в том, что не требуется перестраивать последовательность узлов, независимо от того, в какую позицию списка вставляется новый элемент. Недостаток связных списков – это последовательный доступ (**sequential access**) к элементам, тогда как для массивов время доступа постоянно и не зависит от размера - **random access**. Если приложению требуется быстрый поиск элемента по индексу, то в данном случае списки не подходят, и лучше воспользоваться массивами.

Связный список - это рекурсивная структура, так как узел всегда содержит указатель на следующий узел. Это позволяет использовать простой рекурсивный алгоритм для таких операций, как объединение двух списков или изменение порядка элементов на обратный. У односвязных списков есть важное преимущество: если к вершине такого списка добавляется новый элемент, то старый список будет по прежнему доступен по ссылке на только что добавленный узел. Этот прием называется **persistent data structure** (постоянное хранение данных). У двусвязных списков есть свои преимущества, так как они позволяют выполнять итерацию в обоих направлениях, в отличие от односвязных списков.

**34.Алгоритм сортировки вставками, и его сложность. Алгоритм быстрого перебора поиска в отсортированном массиве. Сложность поиска в отсортированном и не отсортированном массиве.**

**Сортировка вставками (Insertion Sort).**

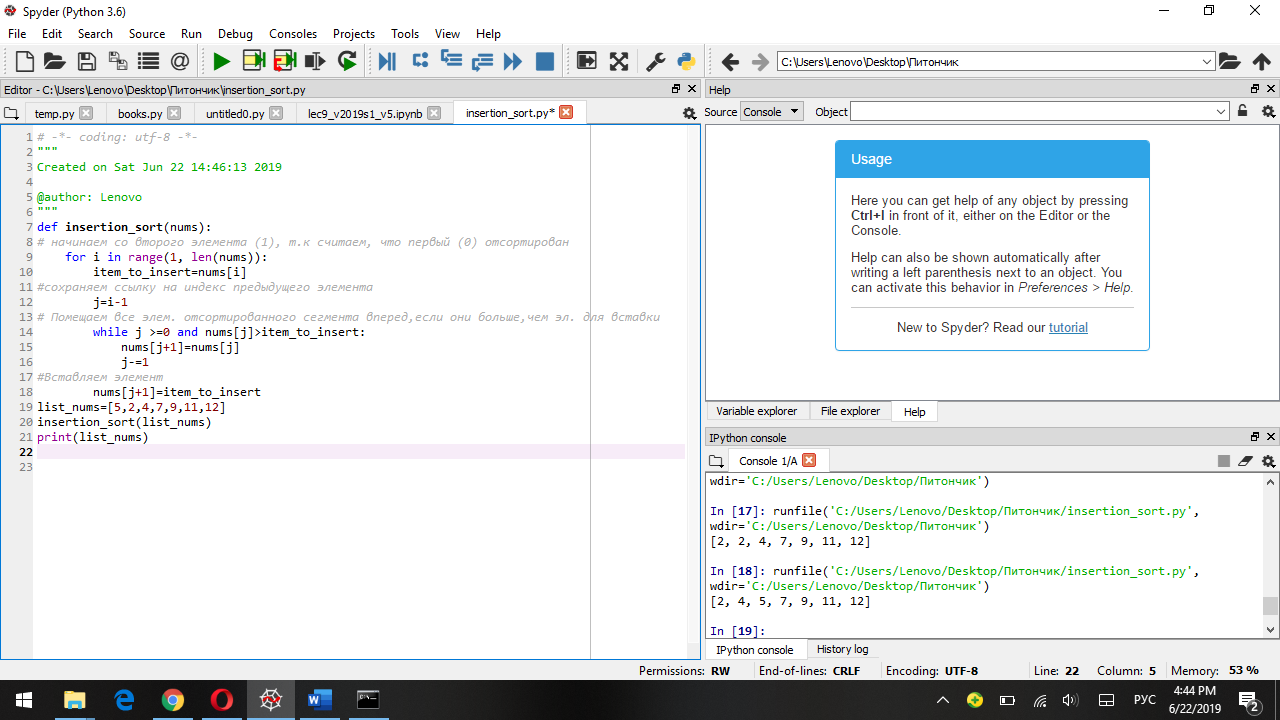
Этот алгоритм сегментирует список на отсортированные и несортированные части. Он перебирает сортированный сегмент и вставляет просматриваемый элемент в правильную позицию отсортированного списка.

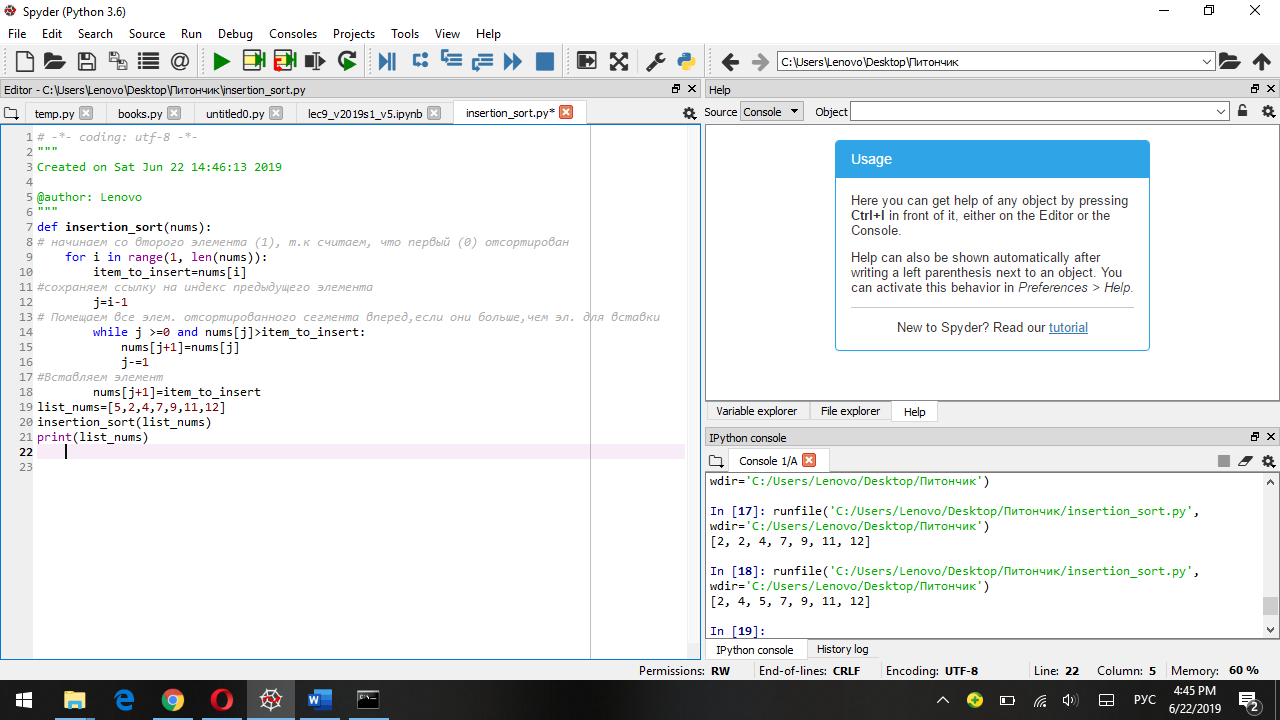
**Объяснение.**

Предполагается, что первый элемент списка отсортирован. Затем мы переходим к следующему элементу, назовем его **х**. Если **x** больше первого элемента, мы оставляем его как есть. Если **x** меньше, мы копируем значение первого элемента во вторую позицию и затем устанавливаем первый элемент в **x**.

Когда мы переходим к другим элементам несортированного сегмента, мы непрерывно перемещаем более крупные элементы в отсортированном сегменте вверх по списку, пока не встретим элемент меньше **x**, или не достигнем конца отсортированного сегмента, а затем поместим **x**в его правильное положение.

**Реализация.**





**Сложность**

Сложность сортировки вставками в среднем равна O(**n^2**).

**Алгоритм быстрого перебора поиска в отсортированном массиве.**

**(или же двоичный (бинарный) поиск элемента в массиве).**

Если у нас есть массив, содержащий упорядоченную последовательность данных, то очень эффективен двоичный поиск.  
Бинарный поиск позволяет найти данный элемент в отсортированном массиве или определить, что он не встречается в данном массиве за O(log *n*) действий, где *n* - количество элементов в массиве.   
  
**Описание алгоритма:**

1. Находится средний элемент последовательности. Для этого первый и последний элементы связываются с переменными, а средний вычисляется.
2. Средний элемент сравнивается с искомым значение. В зависимости от того, больше оно или меньше среднего элемента, дальнейший поиск будет происходить лишь в левой или правой половинах массива. Если значение среднего элемента окажется равным искомому, то поиск завершен.
3. Одна из границ исследуемой последовательности становится равной предыдущему или последующему среднему элементу из п.2.
4. Снова находится средний элемент, теперь уже в «выбранной» половине. Описанный выше алгоритм повторяется уже для данной последовательности.

Переменные i и j содержат, соответственно, левую и правую границы отрезка массива, где находится нужный нам элемент.  
**Реализация.**

def BinSearch(li, x):

i = 0

j = len(li)-1

m = int(j/2)

while li[m] != x and i < j:

if x > li[m]:

i = m+1

else:

j = m-1

m = int((i+j)/2)

if i > j:

return 'Нет такого'

else:

return m

**Двоичный, или бинарный, поиск значения** в списке или массиве используется только для упорядоченных последовательностей, то есть отсортированных по возрастанию или убыванию.

# возможности по ее улучшению.

**Быстрая сортировка**, сортировка Хоара (англ. **quicksort**), часто называемая qsort — широко известный алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром во время его работы в МГУ в 1960 году.

Общая **идея алгоритма** состоит в следующем:

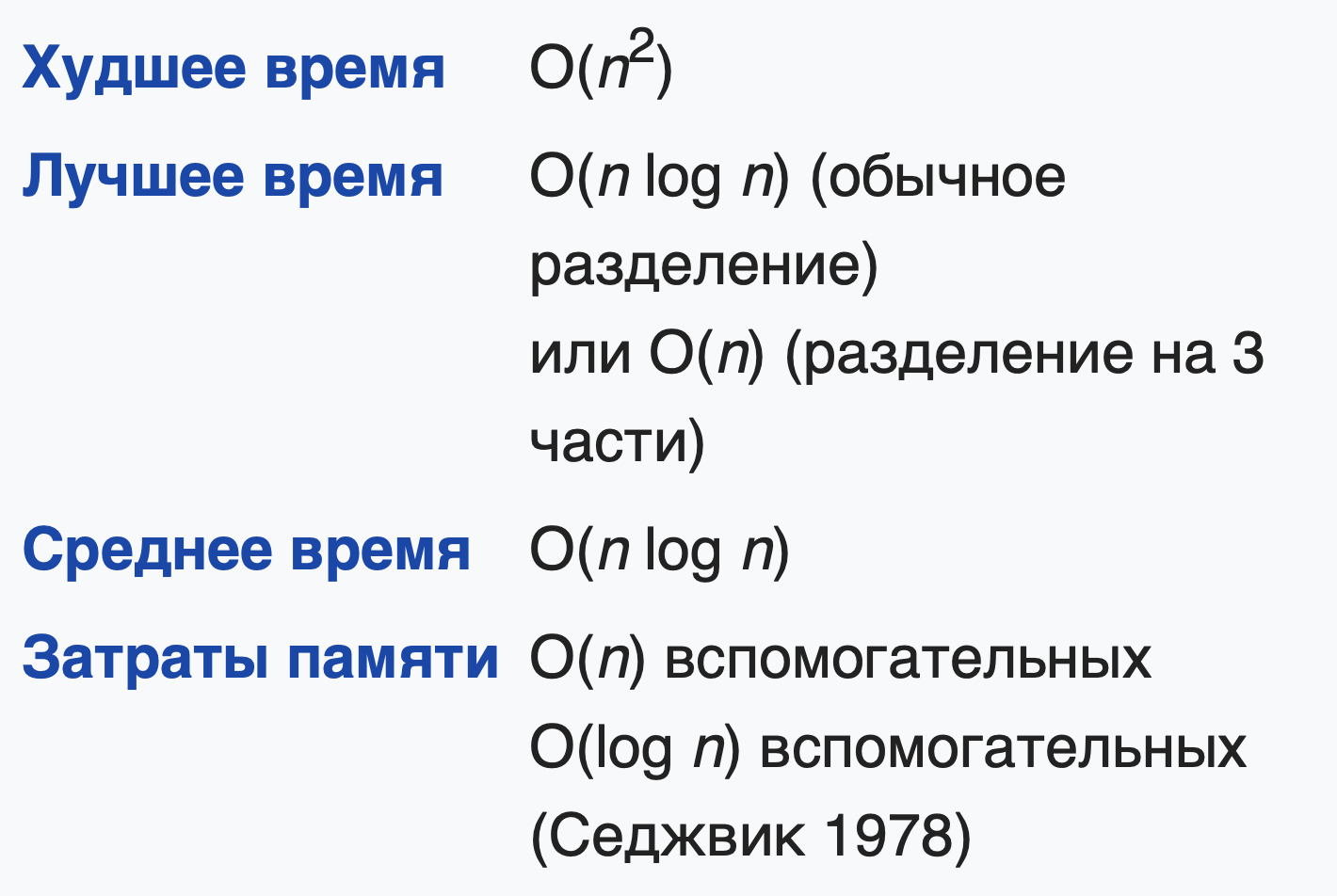
* Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. *(От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективности)*
* Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные» и «большие».
* Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

*На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения*

Варианты написания кода:

1. # Чисто питоновское решение, использует рекурсию => не запоминает границы и каждый раз их пересчитывает
2. **def** quicksort(all\_elements\_list):
3. **if** all\_elements\_list == []: **return** all\_elements\_list
4. last\_element = all\_elements\_list[0]
5. lesser\_element = quicksort([x **for** x **in** all\_elements\_list[1:] **if** x < last\_element])
6. greater\_element = quicksort([x **for** x **in** all\_elements\_list[1:] **if** x >= last\_element])
7. **return** lesser\_element + [last\_element] + greater\_element
8. **print**(quicksort([3,5,4,0,6,1,2]))
9. # В 1 строчку тоже самое
10. **def** quicksort(all\_elements\_list):
11. **return** [] **if** all\_elements\_list==[] **else** quicksort([x **for** x **in** all\_elements\_list[1:] **if** x < all\_elements\_list[0]]) + [all\_elements\_list[0]] + quicksort([x **for** x **in** all\_elements\_list[1:] **if** x >= all\_elements\_list[0]])
12. **print**(quicksort([3,5,4,0,6,1,2]))
13. # Еще одна версия
14. **def** quicksort(all\_elements\_list):
15. **if** all\_elements\_list: **return** quicksort([x **for** x **in** all\_elements\_list **if** x<all\_elements\_list[0]]) + [x **for** x **in** all\_elements\_list **if** x==all\_elements\_list[0]] + quicksort([x **for** x **in** all\_elements\_list **if** x>all\_elements\_list[0]])
16. **return** []
17. # Математическая версия
18. **def** quicksort(all\_elements\_list):
19. **if** all\_elements\_list: **return** quicksort(filter(**lambda** x: x < all\_elements\_list[0], all\_elements\_list[1:])) + all\_elements\_list[0:1] + quicksort(filter(**lambda** x: x >= all\_elements\_list[0], all\_elements\_list[1:]))
20. **return** []

**Сложность**



Производительность данного алгоритма лучше, чем у сортировки пузырьком, но при простой реализации и худшем варианте скорость примерно равна

**Сравнение скоростей:**

* Стандартная питоновcкая сортировка на C++: list.sort() 4 function calls in 0.004 CPU seconds
* Варианты, написанные выше: quicksort(list) 33000-150000 function calls in 0.432-0.708 CPU seconds

**Улучшить алгоритм можно с помощью нескольких способов:**

* Правильно разбивать список(массив) – улучшать алгоритм выбора значения
* Увеличить число разбиваний текста. Но больше 3 не эффективно
* Вероятность выбора правильной (первой приближенной) оси

**Улучшенный алгоритм быстрой сортировки до максимальной скорости**

Визуализация работы 38 сек: <https://youtu.be/kPRA0W1kECg?t=38>

Код <https://pastebin.com/80U6Eb9H>

1. **def** quicksort(all\_elements\_list):
2. **def** quicksort\_position(all\_elements\_list, left, right):
3. position = left
4. j = right - 1
5. **while** (True): # Цикл ставит разделяющий элемент на правильное место
6. **while** (all\_elements\_list[position] < all\_elements\_list[right]): position += 1
7. **while** (all\_elements\_list[j] > all\_elements\_list[right] **and** j > left): j -= 1
8. **if** (position >= j): **break**
9. all\_elements\_list[position], all\_elements\_list[j] = all\_elements\_list[j],all\_elements\_list[position]
10. all\_elements\_list[right], all\_elements\_list[position]  = all\_elements\_list[position],all\_elements\_list[right]
11. **return** position
12. stack = [x **for** x **in** range(int(4 + len(all\_elements\_list) / 2))] # Создание стека
13. k = 0 # Указатель на минимальную позицию
14. stack[0] = 0 # Указатель на позицию левой границы половины
15. stack[1] = len(all\_elements\_list) - 1 # Указатель на позицию правой границы половины
16. **while** (k >= 0):
17. position = quicksort\_position(all\_elements\_list, stack[k], stack[k + 1])
18. **if**(position != stack[k + 1]):right\_left\_border = position + 1 # Левая граница правого подъинтервала
19. **else**:right\_left\_border = stack[k + 1]
20. right\_right\_border = stack[k + 1] # Правая граница правого подъинтервала
21. left\_left\_border = stack[k] # Левая граница левого подъинтервал
22. **if**(position != stack[k]):left\_right\_border =position - 1 #Правая граница левого подъинтервал
23. **else**:left\_right\_border = stack[k]
24. k -= 2 # Удаляет текущий интервал
25. **if** (right\_left\_border != right\_right\_border):k += 2; stack[k] = right\_left\_border; stack[k + 1] = right\_right\_border
26. **if** (left\_left\_border != left\_right\_border):k += 2; stack[k] = left\_left\_border; stack[k + 1] = left\_right\_border
27. **return** all\_elements\_list
28. **print**(quicksort([3,5,4,0,6,1,2]))

**38. Реализация двоичных деревьев в виде связанных объектов. Различные реализации рекурсивного обхода двоичных деревьев.**

Дерево - связный ациклический граф.

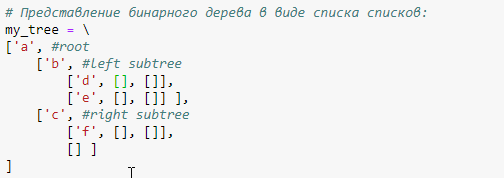
Дочерние узлы - все узлы имеющие входящие связи от некоторого узла, называются дочерними узлами данного узла.

Родитель - узел, исходящая связь которого соединена с данным узлом.

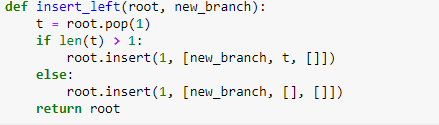
**Двоичное (бинарное) дерево** — иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Обычно, первый называется родительским узлом, а дети называются левым и правым наследниками.

Реализация двоичных деревьев:

Представление бинарного дерева в виде списка списков:



# Пример реализации функции insert\_left для представления бинарного дерева в виде списка списков:



*Прямой порядок обхода* дерева:

1. Первым просматривается корневой узел
2. Затем производится рекурсивный прямой обход левого поддерева.
3. Затем производится рекурсивный прямой обход правого поддерева.

http://localhost:8888/notebooks/t_tree_1.png

*Обратный порядок обхода* дерева:

1. Первым производится рекурсивный обратный обход левого поддерева.
2. Затем производится рекурсивный обратный обход правого поддерева.
3. Затем просматривается корневой узел

http://localhost:8888/notebooks/t_tree_2.png

*Симметричный порядок обхода* дерева:

1. Первым производится рекурсивный симметричный обход левого поддерева.
2. Затем просматривается корневой узел
3. Затем производится рекурсивный симметричный обход правого поддерева.

