Теоретические вопросы для подготовки к экзамену

1. Присвоение по ссылке и по значению. Специфика создания объектов и присвоения в Python, особенности Python в связи с распространенностью использования неизменяемых типов.
2. Циклы в Python, работа и устройство цикла for, типичное применение range и enumerate в цикле for.
3. Списки в Python. Обращение к элементам списка и создание срезов. Обход списка и поиск элементов в списке. Ключевые операции, проводящие к изменению списка и порождающие измененные списки, копирование списков.
4. Словари в Python. Итерирование по словарям, преобразование между словарями и списками в Python. Операции с представлениями словарей.
5. Множества в Python. Основные способы создания, получения и изменения значений. Обход множеств. Выполнение основных операций с парой множеств в Python.
6. Кортежи в Python. Отличия кортежей от списков. Распаковка и частичная распаковка кортежей.
7. Выражения генераторы и генераторы списков в Python. Использование условий в генераторах.
8. 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.

9. Передача переменного количества параметров (именованных и не именованных) в функции Python. Вызов функции с позиционными параметрами, находящимися в списке, и именованными параметрами, находящимися в словаре.
10. Синтаксис и семантика обработки исключительных ситуаций в Python. Создание пользовательских исключений и инструкция assert.
11. Концепция класса и объекта. Принципы и механизмы ООП.
12. Объявление класса, конструктор, создание объектов и одиночное наследование в Python. Управление доступом к атрибутам класса в Python.
13. Полиморфизм и утиная типизация и проверка принадлежности объекта к классу в языке Python.
14. Методы классов и статические переменные и методы в Python. Специальные методы для использования пользовательских классов со стандартными операторами и функциями.
15. Глобальные и локальные переменные в функциях на примере Python. Побочные эффекты вызова функций и их последствия.
16. Функции высшего порядка и декораторы в Python.
17. Концепция map/filter/reduce. Реализация map/filter/reduce в Python и пример их использования.
18. 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.

19. Функции генераторы и выражения генераторы: создание и применение в Python.
20. Абстрактная структура данных стек и очередь: базовые и расширенные операции, их сложность.
21. Связные списки: однонаправленные и двунаправленные – принцип реализации. Сравнение скорости выполнения основных операций в связных списках и в динамическом массиве.
22. Алгоритм обменной сортировки, сложность сортировки и возможности по ее улучшению.
23. Алгоритм сортировки выбором, сложность сортировки и возможности по ее улучшению.
24. Алгоритм сортировки вставками, его сложность. Алгоритм быстрого поиска в отсортированном массиве. Сложность поиска в отсортированном и не отсортированном массиве.
25. Алгоритм сортировки Шелла, сложность сортировки и возможности по ее улучшению.
26. Алгоритм быстрой сортировки, сложность сортировки и возможности по ее улучшению.
27. Реализация двоичных деревьев в виде связанных объектов. Различные реализации рекурсивного обхода двоичных деревьев.
28. Абстрактный тип данных - ассоциативный массив и принцип его реализации на основе хэш-таблиц и хэш-функций.

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

Оператор "=" — это не оператор присваивания значения переменной, как в некоторых других языках программирования. Оператор "=" связывает ссылку на объект с объектом, находящимся в памяти.
Способ присвоения зависит от типа объекта:
1)Неизменяемые
Неизменяемые объекты передаются по значению. Это значит, что при изменении значения переменной будет создан новый объект. К этому типу относятся: - числовые данные (int, float, complex) - символьные строки (str) - кортежи (tuple) При инициализации переменной незменяемого типа создается объект (например, целое число), этот объект имеет некоторый идентификатор:
a=10
оператор = связывает переменную a и объект посредством ссылки. При этом вы не можете изменить сам объект, т.е. когда вы присвоите переменной новое значение, интерпретатор создаст новый объект (если до этого этот объект был создан, то переменная просто получит ссылку), а первоначальный объект удалится из памяти сбощиком мусора, если ссылок на него больше нет.

2) Изменяемые
Изменяемые объекты передаются по ссылке. Это значит, что при изменении значения переменной объект будет изменен. К этому типу относятся: - списки (list) - множества (set) - словари (dict)
Создадим список a, установим для переменной b ссылку на a, прибавим к b элемент списка и выведем их значения и идентификаторы на экран:
a=[1,2]
b=a
b.append(3)
print (a,b0
>> [1,2,3] [1,2,3]
Как мы видим, переменные имеют одинаковые id и элементы списка.
Проверить, ссылаются ли две переменные на один и тот же объект, позволяет оператор is: a is b # True
В целях эффективности интерпретатор производит кэширование малых целых чисел и небольших строк. Это означает, что независимо связанные ссылки будет ссылаться на один и тот же объект.
Для больших чисел механизм кэширования не используется.
z1 = 123812048374719357913758145 79 z2 = 123812048374719357913758145 79 z1 is z2 -> False # две переменные ссылаются на разные объекты
Интерпретатор Python постоянно отслеживает сколько переменных ссылается на данный объект:
import sys # Подключаем модуль sys
sys.getrefcount(obj) # определяем количество ссылок на объект
Когда число ссылок ·на объект становится равно нулю, объект автоматически удаляется из оперативной памяти.

Так работает механизм автоматической сборки мусора в Python.
Исключением являются объекты, которые подлежат кэшированию.
Удалить переменную (т.е. удалить ссылку на объект) можно с помощью инструкции del: del z3

2. Циклы в Python, работа и устройство цикла for, типичное применение range и enumerate в цикле for.

Синтаксис оператора цикла while:
while *boolean_expression*:
```
    suite
```

else:
```
    else_suite
```

блок else не обязательный

Операторы: break - преждевременный выход из цикла, continue - преждевременное окончание итерации цикла и переход к очередной проверке условия while

Синтаксис оператора цикла for:
for expression in iterable:
```
    for_suite
```

else:
```
    else_suite
```

блок else не обязательный; внутри цикла могут использоваться операторы break и continue

В качестве iterable может использоваться любой тип данных, допускающий выполнение итераций по нему, включая строки (где итерации выполняются по символам строки), списки, кортежи и другие типы коллекций языка Python.

Итерируемый тип данных - это такой тип, который может возвращать свои элементы по одному. Любой объект, имеющий метод *_iter_*(), или любая последовательность (то есть объект, имеющий метод *_getitem_*(), принимающий целочисленный аргумент со значением от 0 и выше), является итерируемым и может предоставлять итератор. Итератор - это объект, имеющий метод *_next_*(), который при каждом вызове возвращает очередной элемент и возбуждает исключение StopIteration после исчерпания всех элементов.
Порядок, в котором возвращаются элементы, зависит от итерируемого объекта. В случае списков и кортежей элементы обычно возвращаются в предопределенном порядке, начиная с первого элемента (находящегося в позиции с индексом 0), но другие итераторы возвращают элементы в произвольном порядке - например, итераторы словарей и множеств.
Переменные, созданные в выражении expression цикла for ... in, **продолжают существовать после завершения цикла**. Как и любые локальные переменные, они прекращают свое существование после выхода из области видимости, включающей их.

В качестве выражения expression обычно используется либо единственная переменная, либо последовательность переменных, как правило, в форме кортежа. Если в качестве выражения expression используется кортеж или список, каждый элемент итерируемого объекта iterable распаковывается в элементы expression.

Функция **range** возвращает целочисленный итератор.
Способы обращения к функции range:
range (stop)
С одним аргументом (stop) итератор представляет последовательность целых чисел от 0 до stop - 1.
range (start, stop)
С двумя аргументами (start, stop) - последовательность целых чисел от start до stop - 1.
range (start, stop, step)
С тремя аргументами - последовательность целых чисел от start до stop - 1 с шагом step.
Пример:

```
for i in range(-3, 5, 2):
  print(i)
-3
-1
1
3
```

Синтаксис функции ***enumerte*** **:**

enumerate(i)

enumerate (i, start)

Обычно используется в циклах for ... in, чтобы получить последовательность кортежей (index, item), где значения индексов начинают отсчитывать от 0 или от значения start;

Пример:

```
st = 'Hello world'
for ind, symb in enumerate(st):
  print("Symbol '{}' has index {}".format(symb, ind))
Symbol 'H' has index 0
```

```
Symbol 'e' has index 1
Symbol 'l' has index 2
Symbol 'l' has index 3
…
```

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

Списки – это нумерованные наборы объектов. Каждый элемент набора содержит лишь ссылку на объект. Поэтому списки могут содержать объекты произвольного типа данных и иметь неограниченную степень вложенности. Позиция элемента в наборе задается индексом. Нумерация начинается с 0.
Список – изменяемый тип данных.

Создание списков:
Создать список можно несколькими способами:
➢ явно указав все элементы списка в тексте. Элементы списка перечисляются в квадратных скобках через запятую. Они могут быть разных типов:
Пример:
L1=[]
L2=[4, ‘cd‘, [5,4,3]]

➢ с помощью функции list(). Функция позволяет преобразовать в список данные других типов:

Пример:
L1=list()
L2=list(“Пример”) #[‘П’,’р’,’и’,’м’,’е’,’р’]

➢ заполнив список поэлементно с помощью метода append(), который добавляет в конец списка указанное значение(метод изменяет текущий список):

L=[]
L.append(2)
Оператор + объединяет два списка в новый список. В результате получается новый список.
L=[“a”]
L=L+[“b”] #получим L=[“a”,”b”]

➢ с помощью генератора списка:

L=[i**2 for i in range(1,4)] # [1,4,9]

Выражение L[i] позволяет получить значение с номером i из списка L. Элементы списка нумеруются с нуля. Можно использовать отрицательные индексы: последний элемент имеет индекс -1, предпоследний – индекс -2 и так далее. Списки – изменяемый тип данных. Поэтому присвоить значение элементу списка L с индексом i можно с помощью оператора L[i] = x.

Получение среза
Операция позволяет выделить из списка фрагмент.
В общем случае срез задается тремя целыми числами:
«начало» : «конец» : «шаг»
Параметр «шаг» по умолчанию равен 1.
Если параметр «начало» не указан, то используется индекс первого элемента для положительного шага, и индекс последнего элемента – для отрицательного.
Если параметр «конец» не указан, то возвращается фрагмент до конца списка.

Пример:
L=[4,7,6,8,4]
A=L[1:3] #A=[7,6]
B=L[: :-1] #B=[4,8,6,7,4]

Перебор элементов списка

Для обработки всех элементов списка обычно используется цикл for.

Если вам не нужно изменять элементы списка, например, при вычислении суммы элементов, то самым удобным будет вариант цикла по значениям списка. Переменная x в данном случае по очереди будет равна каждому элементу списка.

Для поиска элемента можно использовать следующий метод: **index( )** – возвращает индекс элемента, имеющего указанное значение. Второй и третий параметры определяют часть списка, в которой выполняется поиск. Если значение не найдено, возникнет ошибка.

Метод **count( )** – возвращает количество элементов с указанным значением.

Операции над списками

➢ конкатенация (+)
Операция позволяет объединить два списка в один. Результатом будет новый список:

➢ повторение (*)
Повторяет список указанное количество раз:

```
L = [1, 2]
L1 = L * 3  # L1=[1,2,1,2,1,2]
n = 5
L2 = [0] * n # L2=[0,0,0,0,0]
```

➢ проверка на вхождение (in, not in)
Оператор in проверяет наличие значения в списке, оператор not in – отсутствие значения. Возвращают значения True или False:

➢ доступ по индексу ([])
Выражение L[i] позволяет получить значение с номером i из списка L. Элементы списка нумеруются с нуля. Можно использовать отрицательные индексы: последний элемент имеет индекс -1, предпоследний – индекс -2 и так далее. Списки – изменяемый тип данных. Поэтому присвоить значение элементу списка L с индексом i можно с помощью оператора L[i] = x.

```
L = [1, 2, 3]
if 5 in L: #для этого L выражение имеет значение False
```

➢ удаление (del)
Оператор удалит из списка элемент с заданным индексом или все элементы, попавшие в срез:

```
L = [10, 20, 30, 40, 50]
del L[1]    #L будет равно [10, 30, 40, 50]
del L[::2]  #L будет равно [30, 50]
```

Копирование списков:
Создание поверхностной копии списка.

```
lst_from = [1, 2, 3, 4, 5] # создание списка
lst_to = list(lst_from) # создание нового списка на основе объекта, который возвращает последовательность значений
lst_to
Out:
```

```
[1, 2, 3, 4, 5]
lst_from is lst_to # проверка на совпадение ссылок на списки
False
lst_from == lst_to # проверка на равенство значений списков
Out[43]:
True
# изменение исходного списка не влияет на его копию:
lst_from[0] = 13
print('lst_from:', lst_from)
print(
```

поверхностное копирование вложенных списков не приводит к копированию вложенных списков:
Ссылки на списки не совпадают, а вложенные списки являются одним объектом, на который имеется две ссылки (при изменении одного внутреннего списка будет меняться и другой)

Создание глубокой копии списков

```
import copy # подключаем модуль copy
lst_form2 = [1, 2, [13, 4, 5] ]
lst_to2_deep  =  copy.deepcopy(lst_from2) #  делаем полную копию списка
lst_from2[2] is lst_to2_deep[2] # вложенные списки НЕ совпадают
Out:  False
```

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

Словарь – набор объектов, каждый из которых является парой вида ***ключ : значение***. Ключи используются для доступа к значениям элементов словаря так же, как индексы для доступа к элементам списка. Значения могут быть любого типа. Все ключи в словаре уникальные. В терминологии Python словари являются отображениями.

Перебор элементов словаря

Перебор всех элементов словаря выполняется с помощью цикла for.

Если в заголовке цикла указать просто имя словаря, то переменная цикла на каждой итерации будет равна ключу очередного элемента словаря. Доступ к значению при необходимости выполняется стандартным образом, ключи могут выдаваться совсем не в том порядке, в котором элементы добавлялись в словарь:

```
d={'a':1, 'b':2, 'c':3}
for k in d:
    print( (k, d[k]), end='  ')
print()
#выведет ('a', 1)  ('b', 2)  ('c', 3)
```

Если использовать в заголовке цикла метод keys(), то результат будет точно такой же:

Если использовать в заголовке цикла метод values(), то переменная цикла на каждой итерации будет равна значению очередного элемента словаря. Ключи в таком варианте цикла будут недоступны. Пример:

```
for k in d.values():
    print(k, end='  ')
print()
#выведет 1  2  3
```

Если использовать в заголовке цикла метод items(), то значением переменной цикла k будет кортеж из двух элементов. Тогда k[0] – это ключ очередного элемента словаря, а k[1] – значение этого элемента:

```
for x, v in d.items():
    print( x, str(v), sep=', ', end=' ... ')
print()
#выведет a, 1 ... b, 2 ... c, 3 ...
```

Функция zip() возвращает последовательность, которую можно **преобразовать в список или словарь**:

```
id = ["a", "b", "c"] #список ключей (строк)
val = [2, 8, 16]     #список значений (чисел)

#получаем список пар с помощью zip()
L = list(zip(id, val)) #L = [('a', 2), ('b', 8), ('c', 16)]

#получаем словарь с помощью zip()
d5 = dict(zip(id, val))#d5 = {'a': 2, 'b': 8, 'c': 16}
```

Операции над словарями

➢ доступ по ключу ([])
Для получения значения элемента словаря с заданным ключом нужно указать имя словаря и значение ключа в квадратных скобках. Если в словаре нет элемента с таким ключом, возникнет ошибка:

Чтобы подобные ошибки не возникали, можно либо вначале проверять наличие элемента в словаре, либо использовать метод get(), либо

перехватывать эти ошибки (обрабатывать исключение KeyError

➢ проверка наличия элемента с заданным ключом (in)
Оператор in проверяет наличие элемента с указанным ключом в словаре. Возвращает значения True или False:

➢ удаление (del)
Удаляет из словаря элемент с заданным ключом. Если элемент с таким ключом отсутствует, то возникает ошибка (возбуждается исключение KeyError). Пример:

Словари не являются последовательностями, поэтому операции получения среза, конкатенации, повторения для них недопустимы.

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

Множество – это неупорядоченная коллекция уникальных элементов, с которой можно сравнивать другие элементы, чтобы определить, принадлежат ли они этому множеству. Множество может содержать только элементы неизменяемых типов, например числа, строки, кортежи. Объявить множество можно с помощью функции set().

Создание множества

Создать множество можно несколькими способами:
➢ явно указав все элементы в программе. Элементы множества перечисляются в фигурных скобках через запятую.

```
A = {7, 4, 3, 5, 7, 4}     #A = {3, 4, 5, 7}
B = {'a', 1, 'b', 2}       #B = {1, 'a', 2, 'b'}
C = {}   #Это не множество! Это пустой словарь
x = 11
D = {10, x, x+1}           #D = {10, 11, 12}
```

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

```
M1 = set()                 #M1 - пустое множество
M2 = set('abcdab')         #M2 = {'a', 'b', 'c', 'd'}
M3 = set( [1, 4, 6, 2, 4, 5, 1] ) #M3 = {1, 2, 4, 5, 6}
M4 = set( ('a', 145) )             #M4 = {145, 'a'}
M5 = set( {'a':2, 'b':3})          #M5 = {'a', 'b'}
M6 = set( {3, 4, 4} )              #M6 = {3, 4}
```

➢ с помощью генератора множества.
Синтаксис генераторов множеств похож на синтаксис генераторов списков, но выражение заключается в фигурные скобки.

```
A = {t for t in range(1, 6)} #A = {1, 2, 3, 4, 5}
B = {t for t in range(1, 10) if t%3==0} #B = {3, 6, 9}
```

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

Операции над множествами

➢ объединение множеств (|)

```
A = {1, 2 , 3}
B = {3, 4, 5}
C = A | B  #C = {1, 2, 3, 4, 5}
```

➢ пересечение множеств (&)
➢ разность множеств (-) Результатом A - B является множество, содержащее все элементы A, которые не входят в B:

➢ симметричная разность множеств (^) Результатом A ^ B является множество, содержащее те элементы A и B, которые не входят в пересечение этих множеств

Приоритет операций в порядке убывания: -, &, ^, |

Методы, для получения, изменения:

add( ) – добавляет объект, указанный в качестве параметра, в множество.

Метод изменяет текущее множество и ничего не возвращает.

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).

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.

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:

```
A.clear()
```

copy( ) – создает копию множества. Пример:

Перебор элементов множества

Для обработки всех элементов множества используется цикл for:

```
A = {2, 3, 4, 5}
for x in A:
    print(x, end = " ")
print()
```

6. Кортежи в Python. Отличия кортежей от списков. Распаковка и частичная распаковка кортежей.

Кортежи, так же как и списки, являются упорядоченными последовательностями элементов. Кортежи во многом аналогичны спискам, но имеют одно очень важное отличие - изменить кортеж нельзя. Можно сказать, что кортеж- это список, доступный "только для чтения".

Когда справа от оператора присваивания указывается последовательность (в данном случае - это кортежи), а слева указан кортеж, мы говорим, что последовательность справа распаковывается.

```
a, b, c = (10, 11, 12)
print('a = {}, b = {}, c = {}'.format(a, b, c))
```

```
out: a = 10, b = 11, c = 12
In[]:
x = 10
y = 1
y, x = x, y
print('x = {}, y = {}'.format(x, y))
x = 1, y = 10
```

Существует возможность частичной распаковки:

```
a, b, *c = range (10, 16)
print('a = {}, b = {}, c = {}'.format(a, b, c))
out: a = 10, b = 11, c = [12, 13, 14, 15]
# символ звездочка может находиться в любом месте, но не бол
ее одного раза:
a, *b, c = range (10, 16)
print('a = {}, b = {}, c = {}'.format(a, b, c))
a = 10, b = [11, 12, 13, 14], c = 15
In [43]:
a, *b, c = range (10, 12)
print('a = {}, b = {}, c = {}'.format(a, b, c))
a = 10, b = [], c = 11
```

7. Выражения генераторы и генераторы списков в Python. Использование условий в генераторах.

Генераторы списков:

Инструкция, выполняемая внутри цикла, находится пер ед ним, выражение внутри скобок не содержит оператор а присваивания.

```
n = 3
A = [ i ** 2 for i in range(1, n + 1)]  #[1, 4, 9]
```

Генераторы списков могут иметь сложную структуру, например, состоять из нескольких вложенных циклов и содержать оператор ветвления.

```
D = [ i*10+j for i in range(1,n) for j in range(1,n) if i!
                                                #[12, 21]
```

Вычисление списка D с помощью генератора эквивалентно следующему коду:

```
D = []
for i in range(1,n):
    for j in range(1,n):
        if i!=j:
            D.append(i*10+j)
```

Использование генератора позволяет не только написать более компактную программу, но и работать она будет быстрее, чем эквивалентные циклы for. При необходимости изменить элементы списка будет происходить возвращение нового списка, а не изменение исходного. На каждой итерации цикла будет генерироваться новый элемент, которому неявным образом присваивается результат выполнения выражения внутри цикла.

Если выражение разместить не внутри квадратных скобок, а внутри круглых, то буде возвращаться не список, а итератор. Такие конструкции называются **выражениями-генераторами.**

```
arr=[1,4,12,45,10]
sum(i for i in arr if  i%2 == 0))
```

output:26

8. 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.

Объявление и вызов (+ теория о том, что это)

Функция описывается с помощью ключевого слова def по следующей схеме:

```
def <Имя функции> ([<Параметры>]):
    ["""Строка документирования""
"]
    <Тело функции>
    [return <Значение>]
```

Имя функции должно быть уникальным идентификатором, состоящим из латинских букв, цифр и знаков подчеркивания, причем имя функции не может начинаться с цифры. В качестве имени нельзя использовать ключевые слова, кроме того, следует избегать совпадений с названиями встроенных идентификаторов. Регистр символов в названии функции имеет значение.

После имени функции в круглых скобках можно указать один или несколько параметров через запятую. После круглых скобок ставится двоеточие.

Необязательная инструкция return позволяет вернуть значение из функции. После исполнения этой инструкции выполнение функции будет остановлено. Если необходимо передать несколько значений, то их нужно указать через запятую, будет возвращен кортеж из них.

Инструкция def создает объект, имеющий тип function, и сохраняет ссылку на него в идентификаторе, указанном после инструкции def. Таким образом, мы можем сохранить ссылку на функцию в другой переменной.

Имя переменной в вызове функции может не совпадать с именем переменной в определении функции. Кроме того, глобальные переменные x и y не конфликтуют с одноименными переменными в определении функции, т.к. они расположены в разных областях видимости. Переменные, указанные в определении функции, являются **локальными** и доступны только внутри функции.

Определение функции должно быть расположено ПЕРЕД ВЫЗОВОМ ФУНКЦИИ.

Получение информации о функции:
Объекты функций поддерживают множество атрибутов. обратиться к которым можно указав атрибут после названия функции через точку. Например, через атрибут *name* можно получить название функции в виде строки, через атрибут *doc* - строку документирования и т. д. (примеры: summa.__name__ , summa.__doc__)

Необязательные параметры и сопоставление по ключам

Чтобы сделать некоторые параметры необязательными, следует в определении функции присвоить этому параметру начальное значение.

```
In [16]:
def summa(x, y=2):
  return x + y
In [17]:
summa(5)
Out[17]:
7
In [18]:
summa(5, 4)
Out[18]:
9
```

Синтаксис определения параметров функции не позволяет указывать параметры без значений по умолчанию, после параметров со значениями по умолчанию, поэтому такое определение: def function(a, b=1, c):, будет вызывать синтаксическую ошибку.

Язык Python позволяет также передать значения в функцию, используя сопоставление по ключам. Для этого при вызове функции параметрам присваиваются значения. Последовательность указания параметров может быть произвольной.

```
In [20]:
summa(y=10, x=20)
```

Переменное число параметров в функции:

Если значения параметров, которые планируется передать в функцию, содержатся в кортеже или списке, то перед объектом следует указать символ *.

```
def multi_summa(x=1, y=1, z=1):
    return x + 10* y + 100* z
t1 = (5, 10 ,15)
multi_summa(*t1)
```

output: 30

Если перед параметром в определении функции указать символ *, то функции можно будет передать произвольное количество параметров. Все переданные параметры сохраняются в кортеже.

Пример:

```
def all_summa(*t):
    """Функция принимает произвольное количество параметров"""
    res = 0
    for i in t:
        res += i
    return  res
all_summa(10, 20, 30, 40, 50)
```

output: 150

Если перед параметром в определении функции указать **, то все именованные параметры будут сохранены в словаре.

```
def d_summa(**d):
    for k, v in d.items():  # Перебираем словарь  с  переданными параметрами
        print("{} => {}".format(k, v), end=" ")
d_summa(a=1, x=10, z=-2)
```

output: `a => 1 x => 10 z => -2`

При комбинировании параметров параметр с двумя звездочками указывается самым последним.

Комментирование функции:

Такие примечания необходимо помещать сразу же после определения класса, метода, функции или модуля, заключая текст в тройные кавычки.

```
def greeting():
    """Функция приветствия.
    Создает и выводит строку на экран.
    """
    print("Hello World!") # выводим строку на
экран
greeting() # вызываем функцию greeting()
print(greeting.__doc__) # выводим docstring
```

9. Передача переменного количества параметров (именованных и не именованных) в функции Python. Вызов функции с позиционными параметрами, находящимися в списке, и именованными параметрами, находящимися в словаре.

Необязательные параметры и сопоставление по ключам

Чтобы сделать некоторые параметры необязательными, следует в определении функции присвоить этому параметру начальное значение.

```
In [16]:
def summa(x, y=2):
  return x + y
In [17]:
summa(5)
Out[17]:
7
In [18]:
summa(5, 4)
Out[18]:
9
```

Синтаксис определения параметров функции не позволяет указывать параметры без значений по умолчанию, после параметров со значениями по умолчанию, поэтому такое определение: def function(a, b=1, c):, будет вызывать синтаксическую ошибку.

Язык Python позволяет также передать значения в функцию, используя сопоставление по ключам. Для этого при вызове функции параметрам присваиваются значения. Последовательность указания параметров может быть произвольной.

```
In [20]:
summa(y=10, x=20)
```

Переменное число параметров в функции:

Если значения параметров, которые планируется передать в функцию, содержатся в кортеже или списке, то перед объектом следует указать символ *.

```
def multi_summa(x=1, y=1, z=1):
   return x + 10* y + 100* z
t1 = (5, 10 ,15)
multi_summa(*t1)
output: 30
```

Если перед параметром в определении функции указать символ *, то функции можно будет передать произвольное количество параметров. Все переданные параметры сохраняются в кортеже.

Пример:

```
def all_summa(*t):
   """Функция принимает произвольное количество параметров
"""
   res = 0
   for i in t:
      res += i
   return  res
all_summa(10, 20, 30, 40, 50)
output: 150
```

Если перед параметром в определении функции указать **, то в се именованные параметры будут сохранены в словаре.

```
def d_summa(**d):
   for k, v in d.items():  #  Перебираем словарь  с  переданными па раметрами
      print("{} => {}".format(k, v), end=" ")
d_summa(a=1, x=10, z=-2)
```

output: `a => 1 x => 10 z => -2`

При комбинировании параметров параметр с двумя звездочками указывается самым последним.

10. Синтаксис и семантика обработки исключительных ситуаций в Python. Создание пользовательских исключений и инструкция assert.

Исключения – это извещения интерпретатора, возбуждаемые в случае возникновения ошибки в программном коде или при наступлении какого-либо события. Если в коде не предусмотрена обработка исключения, то программа прерывается и выводится сообщение об ошибке.

Существуют три типа ошибок в программе:

- синтаксические - это ошибки в имени оператора или функции, отсутствие закрывающей или открывающей кавычек и т. д., т. е. ошибки в синтаксисе языка. Как правило, интерпретатор предупредит о наличии ошибки, а программа не будет выполняться совсем.
- логические - это ошибки в логике работы программы, которые можно выявить только по результатам работы скрипта. Как правило, интерпретатор не предупреждает о наличии ошибки. А программа будет

выполняться, т. к. не содержит синтаксических ошибок. Такие ошибки достаточно трудно выявить и исправить;

- ошибки времени выполнепия - это ошибки, которые возникают во время работы скрипта. Причиной являются события, не предусмотренные программистом. Классическим примером служит деление на ноль

в языке Python исключения возбуждаются не только при ошибке, но и как уведомление о наступлении каких-либо событий. Например, метод index () возбуждает исключение ValueError, если искомый фрагмент не входит в строку
Для обработки исключений предназначена инструкция try. Формат инструкции:

```
try: <Блок, в котором перехватьшаются исключения>

[ except [ <Исключение1> [ as <Объект исключения>] ] :

<Блок, вьмолняемьм при возникновении исключения>

[ ... except [<ИсключениеN>[ as <Объект исключения>]]

: <Блок, вьмолняемьм при возникновении исключения>]]

[e1se: <Блок, вьмолняемьм, если исключение не возникло>]

 [finally: <Блок, вьмолняемьм в любом случае>]
```

Инструкции, в которых перехватываются исключения, должны быть расположены внутри блока try. В блоке except в

параметре <Исключение1> указывается класс обрабатываемого исключения. В инструкции except можно указать сразу несколько исключений, перечислив их через запятую внутри круглых скобок

Пример: обработать исключение, возникающее при делении на ноль, можно так:

```
In [4]:
x = 0
try: # Перехватьшаем исключения
  x = 1/0 # Ошибка: деление на 0
except ZeroDivisionError: # Указьшаем класс исключения
  print("Обработали деление на 0")
  # x = 0
print(x)
```

```
Обработали деление на 0
```

Пользовательские исключения

Инструкция raise возбуждает указанное исключение. Она имеет несколько форматов:

- raise <Экземпляр класса>
- raise <Название класса>
- raise <Экземпляр или название класса> from <Объект исключения>
- raise

В первом формате инструкции raise указывается экземпляр класса возбуждаемого исключения. *При создании экземпляра можно передать данные конструктору класса. Эти данные будут доступны через второй параметр в инструкции except.*

```
In [81]:
try:
  raise ValueError("Описание исключения")
```

```
except ValueError as msg:
  print(msg) # Выведет: Описание исключения
Описание исключения
```

Инструкция assert

Инструкция assert возбуждает исключение AssertionError, если логическое выражение возвращает значение False. Инструкция имеет следующий формат:

assert <Логическое выражение> [, <Сообщение>]

Инструкция assert эквивалентна следующему коду:

```
if __debug__: if not <Логическое выражение>:
raise AssertionError(<Сообщение>)
```

пример:

```
def factorial(n):
    assert n >= 0, 'Аргумент n должен быть больш
e 0!'
    assert n % 1 == 0, 'Аргумент n должен быть ц
елым!'
    f = 1
    for i in range(2, n+1):
        f *= i
    return f
factorial (-1)
output: AssertionError: Аргумент n должен быть
больше 0!
```

11. Концепция класса и объекта. Принципы и механизмы ООП.

ООП – это способ организации программы, позволяющий использовать один и тот же код

многократно. В отличие от функций и модулей, ООП позволяет не только разделить программу на фрагменты, но и описать предметы реального вида в виде объектов, а также организовать связи между этими объектами.
Основным «кирпичиком» ООП является класс. Класс – это объект, включающий набор переменных и функций для управления этими переменными. Переменные называются атрибутами, а функции – методами. Класс позволяет создать неограниченное количество экземпляров, основанных на этом классе.

Принципы:
Пожалуй, самым главным понятием ООП является принцип наследования. При помощи наследования мы можем создать новый класс, в котором будет доступ ко всем атрибутам и методам базового класса. Если имя метода в обоих классах совпадает, то будет использоваться метод из подкласса. Чтобы вызвать одноименный метод из базового класса, можно воспользоваться функцией super(). Например: super().__init__ ().
Второц принцип: Абстракция означает выделение главных, наиболее значимых характеристик предмета и наоборот — отбрасывание второстепенных, незначительных.
Третий принцип: Инкапсуляция - ограничение доступа к данным и возможностям их изменения. Очевидные примеры инкапсуляции, с которыми ты уже работал, — геттеры-сеттеры. Если поле age у класса Cat не инкапсулировать, кто угодно сможет написать:
Cat.age = -1000
А механизм инкапсуляции позволяет нам защитить поле age при помощи метода-сеттера, в который мы можем

поместить проверку того, что возраст не может быть отрицательным числом.
Четвертый принцип: Полиморфизм — это возможность работать с несколькими типами так, будто это один и тот же тип. При этом поведение объектов будет разным в зависимости от типа, к которому они принадлежат.

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

Класс описывается с помощью ключевого слова class по следующей схеме:

Class <Название класса> (<класс1>, <класс2>, … <классN>):
<Описание атрибутов и методов>

Инструкция создает новый объект и присваивает ссылку на него идентификатору, указанному после ключевого слова class. После названия класса в круглых скобках можно указать 1 или несколько базовых классов через запятую. Все выражения внутри инструкции class выполняются при создании объекта, а не при создании экземпляра класса.
Создание переменной (атрибута) внутри класса аналогично созданию обычной переменной. *Метод внутри класса создается так же, как и обычная функция, в первом параметре автоматически передается ссылка на экземпляр класса. Общепринято этот параметр называть self.*

При создании экземпляра класса интерпретатор автоматически вызывает метод инициализации __init__(self, <значение1>, …, <значниеN>)
С помощью метода __init__() можно присвоить начальные значения атрибутам класса. При создании экземпляра класса начальные значения указываются в круглых скобках.
Доступ к атрибутам и методам класса производится через переменную self с помощью точечной нотации. Например к атрибуту x из метода класса можно обратиться self.x.
Чтобы использовать атрибуты и методы класса необходимо создать экземпляр класса:
<Экземпляр класса> = <Название класса> (<Параметры>)
При обращении к методам класса используется следующий формат:
<Экземпляр класса>.<Имя метода> (<Параметры>)
Обращение к атрибутам класса осуществляется следующим образом:
<Экземпляр класса>.<Имя атрибута>

Также для доступа к атрибутам и методам можно использовать следующие функции:

- getattr () – возвращает значение атрибута по его названию, заданному в виде строки. С помощью этого атрибута модно сформировать название атрибута динамически во время выполнения программы. Если атрибут не найден, возбуждается исключение AttributeError.
- setattr () – задает значение атрибута. Название атрибута указывается в виде строки.

- delattr () – удаляет указанный атрибут. Название атрибута указывается в виде строки.
- hasattr () – проверяет наличие указанного атрибута. Если атрибут существует, функция возвращает True.

Все атрибуты в классе являются открытыми, т.е. доступны для непосредственного изменения. Можно создавать динамически после создания класса как атрибут объекта класса, так и атрибут экземпляра класса.
Атрибут объекта класса доступен всем экземплярам класса, и после его изменения значения изменится во всех экземплярах калсса. Атрибут экземпляра класса модет хранить уникальное значение для каждого экзмепляра.

Пожалуй, самым главным понятием ООП является принцип наследования. При помощи наследования мы можем создать новый класс, в котором будет доступ ко всем атрибутам и методам базового класса. Если имя метода в обоих классах совпадает, то будет использоваться метод из подкласса. Чтобы вызвать одноименный метод из базового класса, можно воспользоваться функцией super(). Например: super().__init__ ().

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

Полиморфизм — это возможность работать с несколькими типами так, будто это один и тот же тип.

При этом поведение объектов будет разным в зависимости от типа, к которому они принадлежат. Простым примером полиморфизма может служить функция `count()`, выполняющая одинаковое действие для различных типов обьектов: `'abc'.count('a')` и `[1, 2, 'a'].count('a')`. Полиморфизм – важная функция в определении классов Python. Она применяется тогда, когда у классов или подклассов есть общие методы. Таким образом, функции могут использовать объекты любого из этих полиморфных классов, не зная различий между классами.

Полиморфизм в Python основан на **утиной типизации**. Утиная типизация подразумевает определение пригодности объекта для конкретной цели. При использовании обычной типизации эта пригодность определяется типом объекта в отдельности, но в утиной типизации для этого используются методы и свойства рассматриваемого объекта.

Если несколько классов содержат методы с одинаковыми именами, но реализуют их по-разному, эти классы являются полиморфными. Функция сможет оценить эти полиморфные методы, не зная, какой класс она вызывает.

Проверка принадлежности к классу:

type(c_1) == Car

или так

c_1.__class__ == Car

Проверка принадлежности базовому классу:

isinstance (object, class)

Можно проверить, является ли один класс потомком другого:

issubclass (CargoCar, Car)

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

Метод внутри класса создается так же, как и обычная функция, в первом параметре автоматически передается ссылка на экземпляр класса. Общепринято этот параметр называть self.

Пример:

```
Class Myclass:
        def __init__(self):
                self.x=10
        def printx (self):
                print(self.x)
c=Myclass()
c.print_x()
```

Методы класса создаются с помощью декоратора @classmethod. В качестве первого параметра передается ссылка на класс, а не на экземпляр класса. Вызов метода класса осуществляется следующим образом:

<Название класса>.<Название метода> (<Параметры>)
Кроме того, можно вызвать метод класса через экземпляр класса:
<Экземпляр класса>.<Название метода> (<Параметры>)

Пример:

```
class Myclass:
        @classmethod
        def func(cls, x):
                print (cls,x)
Myclass.func(10)
c=Myclass()
c.func(10)
```

Внутри класса можно создать метод, который будет доступен без создания экземпляра класса. Для этого перед определением метода внутри класса следует указать декоратор
@staticmethod. Вызов статического метода без создания экземпляра класса осуществляется следующим образом:
<Название класса>.<Название метода> (<Параметры>)
Кроме того, можно вызвать метод класса через экземпляр класса:
<Экземпляр класса>.<Название метода> (<Параметры>)

Пример

```
class Myclass:
        @staticmethod
        def summa(x,y):
                return x+y
print(Myclass.summa(10,20))
c=Myclass()
print(c.summa(10,20))
```

В определении статического метода нет параметра self, это означает, что внутри него нет доступа к атрибутам и методам экземпляра класса.

Статическая переменная - это переменная, которая была распределена статически, что означает, что ее время жизни - это весь запуск программы. Статические переменные существуют только в одном экземпляре для каждого класса и не создаются.

В Python переменные, объявленные внутри определения класса, но не внутри метода, являются переменными класса или статики.

```
class MyClass:
        static_var = 2
```

Специальные методы:
Это методы, с помощью которых можно добавить в классы «магию». Они всегда обрамлены двумя нижними подчеркиваниями (например, __init__ или __lt__). Перезагрузка операторов позволяет экземплярам классов участвовать в обычных операциях. Для перезагрузки математических методов используются такие методы, как:
x+y – сложение – x.__add__(y)
x+=y – сложение и присваивание – а x.__iadd__(y) и др.

15. Глобальные и локальные переменные в функциях на примере Python. Побочные эффекты вызова функций и их последствия.

Глобальные переменные – это переменные, объявленные в программе вне функции. Они видны в любой части модуля, включая функции. Все изменения одноименной переменной внутри функции не затронут значение одноименной глобальной переменной.
Локальные переменные - это переменные, которым внутри функции присваивается значение. Если имя локальной переменной совпадает с именем глобальной перемснной, то все операции внутри функции осуществляются с локальной переменной, а значение глобальной переменной не изменяется. Локальные персменные видны только внутри тела функции.

Для того чтобы значение глобальной переменной можно было изменить внутри функции, необходимо объявить переменную глобальной с помощью ключевого слова global.
Пример:

```
In [28]:
def func2():
            print('func begin')
             local = 77 # локальная переменная
              global glob # объявление глобальной переменной вну
три func2
             glob = 25 # ИЗМЕНЕНИЕ ГЛОБАЛЬНОЙ переменной вн
утри функции
              print(f'\tЗначение локальной переменной local:{local
}')
```

```
        print(f'\tЗначение ГЛОБАЛЬНОЙ переменной glob:{glob}')
        print('func end')
glob = 10 # глобальная переменная
print(f'Значение глобальной переменной glob:{glob}')
func2() # вызов функции
print(f'Значение глобальной переменной glob:{glob}')
```

```
Значение глобальной переменной glob:10
func begin
        Значение локальной переменной local:77
        Значение ГЛОБАЛЬНОЙ переменной glob:25
func end
Значение глобальной переменной glob:25
```

Поиск идентификатора, используемого внутри функции, будет производиться в следующем порядке:

1. Поиск объявления идентификатора внутри функции (в локальной области видимости).
2. Поиск объявления идентификатора в глобальной области.
3. Поиск во встроенной области видимости (встроенные функции, классы и т. д.).

Побочные эффекты

Изменение внешних переменных при вызове функции определяется как побочный эффект.

```
In [30]:
```

```
# Источником побочных эффектов являются использование глобальных переменных:
factor = 0
def multiples(n):
```

```
  global factor
  factor = factor + 1
  return factor * n
In [31]: factor
Out[31]: 0
In [32]: multiples(2)
Out[32]: 2
In [33]: multiples(2) # Резальтаты вызова функции с одним и тем же параметром различаются!
Out[33]: 4
# Источником побочных переменных так же является использо вание состояния объекта (или функции):

class Multiplier:
  def __init__(self):
    self.factor = 0

  def multiples(self, n):
    self.factor = self.factor + 1
    return self.factor * n
In [36]: m = Multiplier()
In [37]: m.multiples(2)
Out[37]: 2
In [38]: m.multiples(2)
Out[38]: 4
In [39]:
# Источником побочных эффектов может быть ввод/вывод:

def read_byte(f):
  return f.read(1)
In [40]:
# источником побочных эффектов может быть использование датчика случайных чисел:
```

```
import random
def get_random(n):
  return random.randrange(1, n + 1)
```

Отрицательные моменты побочных эффектов:

- тяжелее тестировать программу
- тяжелее делать программы для параллельных вычислений (параллельной работы на нескольких ядрах / процессорах / компьютерах)
- тяжелее производить формальную верификацию (проверку) программы

16. Функции высшего порядка и декораторы в Python.

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

```
# функции обратного вызова (callback functions)
def add(a, b):
  return a + b
def sub(a, b):
  return a - b
def apply_operation(operation, arg1, arg2):
  return operation(arg1, arg2)
In [46]:apply_operation(add, 2, 3)
Out[46]:5
In [47]:  apply_operation(sub, 2, 3)
Out[47]:-1
```

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

```
def positive_result(function): # декоратор
  def wrapper(*args, **kwargs): # обертка
    result = function(*args, **kwargs) #
    assert result >= 0, function.__name__ + "() result isn't >= 0"
    return result
  wrapper.__name__ = function.__name__
  wrapper.__doc__ = function.__doc__
  return wrapper
In [55]:
@positive_result
def discriminant(a, b, c):
  return (b ** 2) - (4 * a * c)
In [56]: discriminant(1, 1, -1)
Out[56]: 5
Input: discriminant(1, 1, 1)
output: AssertionError: discriminant() result
isn't >= 0


def positive_result2(function):
# декоратор, который обеспечивает, что wrapper будет носить
имя исходной функции и ее строку документирования.
  @functools.wraps(function)
  def wrapper(*args, **kwargs):
    result = function(*args, **kwargs)
```

```
    assert result >= 0, function.__name__ + "() result isn't >= 0 !"
    return result
  return wrapper
```

Кроме того, можно создавать декораторы с параметрами

17. Концепция map/filter/reduce. Реализация map/filter/reduce в Python и пример их использования.

Map

Встроенная функция map() позволяет применить функцию к каждому элементу последовательности. Функция имеет следующий формат:

map(<Функция>, <Последовательность1>[, ... , <ПоследовательностьN>])

Функция возвращает объект, поддерживающий итерации, а не список.

Примеры:

```
m1 = map(squared, range(10))
list(m1)
Out[8]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
In [9]: ls0 = list(zip(range(10), range(0, 100, 10)))
#список кортежей
list(map(sum, ls0))
Out[11]: [0, 11, 22, 33, 44, 55, 66, 77, 88, 99]
```

Передача нескольких последовательностей:

```
ls1 = list(range(10))
ls2 = list(range(0, 100, 10))
```

```
list(map(lambda x, y: x+y, ls1, ls2))
Out[15]: [0, 11, 22, 33, 44, 55, 66, 77, 88, 99]
```

Filter

Функция filter() позволяет выполнить проверку элементов последовательности. Формат функции:

filter(<Функция>, <Последовательность>)

Если в первом параметре вместо названия функции указать значение None, то каждый элемент последовательности будет проверен на соответствие значению True. Если элемент в логическом контексте возвращает значение False, то он не будет добавлен в возвращаемый результат. Функция возвращает объект, поддерживающий итерации, а не список.

Пример:

l=[4,6,-2,4,-4,2,8]

lp=list(filter(lambda x: x >=0, l)

print(lt)

output: [4,6,4,2,8]

Reduce

В Python 3 встроенной функции reduce() нет, но её можно найти в модуле functools.

```
import functools
```

или

```
from functools import reduce
```

Вычисляет функцию от двух элементов последовательно, для элементов последовательности с права на лево таким образом, что результатом вычисления становится единственное число.

Левый аргумент функции это аккумулированное значение, правый аргумент - очередное значение из списка.

Если передан необязательный аргумент initializer, то он используется в качестве левого аргумента при первом применении функции (исходного аккумулированного значения).

Если initializer не передан, а последовательность имеет только одно значение, то возвращается это значенние.

```
# Пример:
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) # вычисляется как ((((1+2)+3)+4)+5)
Out[43]: 15
```

18. 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.

Итерируемый тип данных – это такой тип, который может возвращать свои элементы по одному. Любой объект, имеющий метод __iter__(), или любая последовательность (то есть объект, имеющий метод __getitem__(), принимающий целочисленный аргумент со значением от 0 и выше), является итерируемым и может предоставлять итератор.

Итератор - это объект, имеющий метод __next__(), который при каждом вызове возвращает очередной элемент и

возбуждает исключение StopIteration после исчерпания всех элементов.

Получить итератор из любых итерируемых объектов можно, вызвав встроенную функцию iter для них.
Основное место использования итераторов – это цикл *for*.

Примеры создания и обхода итераторв:

```
ls5 = [1, 2, 5, 3, 4, 8]
product = 1
it = iter(ls5) # iter() вызывает l5.__iter__() (ls5 - значение итериру
емого типа, i - итератор )
while True:
  try:
    i = next(it) # next вызывает i.__next__()
    product *= i
  except StopIteration:
    break
product
Out[19]: 960
In [20]:
product = 1
for i in ls5:
  product *= i
print(product)
output: 960
In [61]:
# другой способ:
reduce(op.mul, ls5)
Out[61]: 960
```

К итерируемым типам могут применяться: all(), any(), len(), min(), max(), sum(), **enumerate(iterable, start=0)** - *Возвращает итератор, при каждом проходе предоставляющем кортеж из номера и*

соответствующего члена последовательности.

```
len(ls5), min(ls5), max(ls5), sum(ls5)
Out[62]:
(6, 1, 8, 23)
```

Модуль itertools

```
import itertools as itl
```

itertools.islice(iterable, [start,]stop[, step])

Создает итератор, воспроизводящий элементы, которые вернула бы операция извлечения среза iterable[start:stop:step]. Первые start элементов пропускаются и итерации прекращаются по достижении позиции, указанной в аргументе stop. В необязательном аргументе step передается шаг выборки элементов. В отличие от срезов, в аргументах start, stop и step не допускается использовать отрицательные значения. Если аргумент start опущен, итерации начинаются с 0. Если аргумент step опущен, по умолчанию используется шаг 1.

```
list(itl.islice('ABCDEFG', 2,5))
Out[69]: ['C', 'D', 'E']
```

Бесконечные итераторы

itertools.cycle(iterable)

Создает итератор, который в цикле многократно выполняет обход элементов в объекте iterable. За кулисами создает копию элементов в объекте iterable. Эта копия затем используется для многократного обхода элементов в цикле.

```
list(itl.islice(itl.cycle('ABCD'), 10))
Out[71]: ['A', 'B', 'C', 'D', 'A', 'B', 'C', 'D', 'A', 'B']
```

itertools.count(start=0, step=1)

Создает итератор, который воспроизводит упорядоченную и непрерывную последовательность целых чисел, начиная с n. Если аргумент n опущен, в качестве первого значения возвращается число 0

```
list(itl.islice(itl.count(7), 10))
Out[73]: [7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

itertools.repeat(object[, times])

Создает итератор, который многократно воспроизводит объект object. В необязательном аргументе times передается количество повторений. Если этот аргумент не задан, количество повторений будет бесконечным.

```
list(itl.repeat('a', 10))
Out[75]: ['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']
```

Существует еще множество других итераторов в данном модуле.

19. Функции генераторы и выражения генераторы: создание и применение в Python.

Функции генераторы

Функция-генератор, или метод-генератор – это функция, или метод, содержащая выражение yield. В результате обращения к функции-генератору возвращается итератор. Значения из

итератора извлекаются по одному, с помощью его метода __next__(). При каждом вызове метода __next__() он возвращает результат вычисления выражения yield. (Если выражение отсутствует, возвращается значение None.) Когда функция-генератор завершается или выполняет инструкцию return, возбуждается исключение StopIteration.

На практике очень редко приходится вызывать метод __next__() или обрабатывать исключение StopIteration. Обычно функция-генератор используется в качестве итерируемого объекта.

```
#Возвращает каждое значение по требованию:
def letter_range_g(a, z):
  while ord(a) < ord(z):
    yield a
    a = chr(ord(a) + 1)
In [89]:
for l in letter_range_g('a', 'o'):
  print(l, end=', ')
a, b, c, d, e, f, g, h, i, j, k, l, m, n,
```

Кроме функций-генераторов существует возможность создавать еще и выражения-генераторы. Синтаксически они очень похожи на генераторы списков, единственное отличие состоит в том, что они заключаются не в квадратные скобки, а в круглые.

```
# функция-генератор, возвращающая из словаря пары ключ-зна
чение в порядке убывания ключа
def items_in_key_order(d):
  for key in sorted(d, reverse=True):
    yield key, d[key]
d1 = dict(zip(['ac','vg','ddxf','c','ff', 'bfafakl'], range(6)))
list(items_in_key_order(d1))
```

```
Out[97]: [('vg', 1), ('ff', 4), ('ddxf', 2), ('c', 3), ('bfafakl', 5), ('ac', 0)]
In [99]:
[(key, d1[key]) for key in sorted(d1, reverse=True)]
Out[99]: [('vg', 1), ('ff', 4), ('ddxf', 2), ('c', 3), ('bfafakl', 5), ('ac', 0)]
```

Благодаря отложенным вычислениям в генераторах (функциях и выражения) можно экономить ресурсы и создавать генераторы бесконечных последовательностей.

```
# бесконечный генератор четвертей:
def quarters(next_quarter=0.0):
  while True:
    yield next_quarter
    next_quarter += 0.25
result = []
for x in quarters():
  result.append(x)
  if x >= 5.0:
    break
result
```

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

Стек— абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

Стэк можно представить в виде стопки тарелок, книг и т.д.

Базовые операции для работы со стеком:

S.push(e): добавление элемента e на вершину стека S.

S.pop(): удаляет и возвращает верхний элемент стека S. Если стек пуст, то возникает ошибка.

Абстрактная структура данных Стек:

Базовые операции для работы со стеком:

S.push(e): добавление элемента e на вершину стека S.

S.pop(): удаляет и возвращает верхний элемент стека S. Если стек пуст, то возникает ошибка.

```
from collections import deque
class Stack(deque):
   def push(self, a):
      self.append(a)
s1 = Stack()
s1
Out[32]: Stack([])
In [33]:
s1.push(5)
s1.push (3)
s1
Out[34]: Stack([5, 3])
In [35]:
s1.pop ()
Out[35]: 3
s1
Out[36]: Stack([5])
```

Дополнительные операции для работы со стеком:

S.top(): возвращает верхний элемент стека S не удаляя его. Если стек пуст, то возникает ошибка.

S.is_empty( ): возвращает True если стек S не содержит ни одного элемента.

len(S): возвращает количество элементов в стеке S.

Производительность реализации стека на базе динамического массива:

Operation	Running Time
S.push(e)	$O(1)^*$
S.pop( )	$O(1)^*$
S.top( )	$O(1)$
S.is_empty( )	$O(1)$
len(S)	$O(1)$

*amortized

Абстрактная структура данных Очередь:

О́чередь — абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» (FIFO, англ. first in, first out).

Базовые операции для работы с очередью:

Q.enqueue(e): добавление элемента e в конец очереди Q.

Q.dequeue(): удаляет и возвращает первый элемент очереди Q. Если очередь пуста, то возникает ошибка.

```
class Queue(deque):
  def enqueue(self, a):
    self.append(a)
  def dequeue(self):
```

```
        return self.popleft()
q1 = Queue()
q1.enqueue(5)
q1.enqueue(3)
q1
Out[62]: Queue([5, 3])
In [64]:
q1.dequeue()
Out[64]: 5
q1
output: Queue([3])
```

Дополнительные операции для работы с очередью:

Q.first(): возвращает первый элемент очереди Q не удаляя его. Если очередь пуста, то возникает ошибка.

Q.is empty(): возвращает True если очередь Q не содержит ни одного элемента.

len(Q): возвращает количество элементов в очереди Q.

Производительность реализации очереди на базе динамического массива:

Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue( )	$O(1)^*$
Q.first( )	$O(1)$
Q.is_empty( )	$O(1)$
len(Q)	$O(1)$

*amortized

Реализация deque в модуле collections позволяет эффективно добавлять и удалять значения из начала/конца очереди.

Быстрые операции с концом (правым концом) очереди в deque (аналогичны операциям для списка):

append(x) Add x to the right side of the deque.

pop() Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.

Быстрые операции с началом (левым концом) очереди в deque (в списке нет аналогов этих операций):

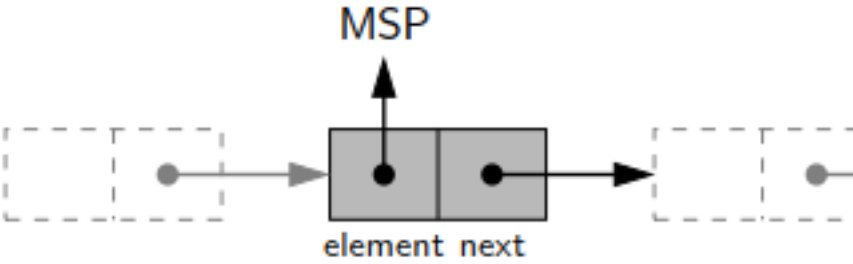
appendleft(x) Add x to the left side of the deque.

popleft() Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.

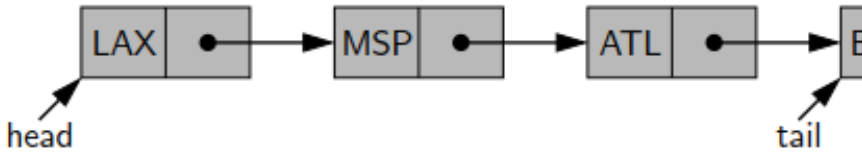
Реализация deque основана на двунаправленных связанных списках массивов фиксированной длины. deque поддерживает итерацию, операции **len(d)**, **reversed(d)**, проверку вхождения с помощью оператора **in** . Операции получения элемента по индексу (такие как `d[0]`, `d[-1]`) быстрые (имеют сложность **O(1)** для двух концов очереди, но длительные (сложность **O(n)**) для элментов в середине списка. Т.е. для быстрого произвольного доступа к элементам списка нужно использовать **list** вместо **deque**.

21. Связные списки: однонаправленные и двунаправленные – принцип реализации. Сравнение скорости выполнения основных операций в связных списках и в динамическом массиве.

Однонаправленные связанные списки.
Каждый узел ссылается на объект, который является элементом последовательности и на следующий узел списка (или хранит значение None, если в списке больше нет узлов)

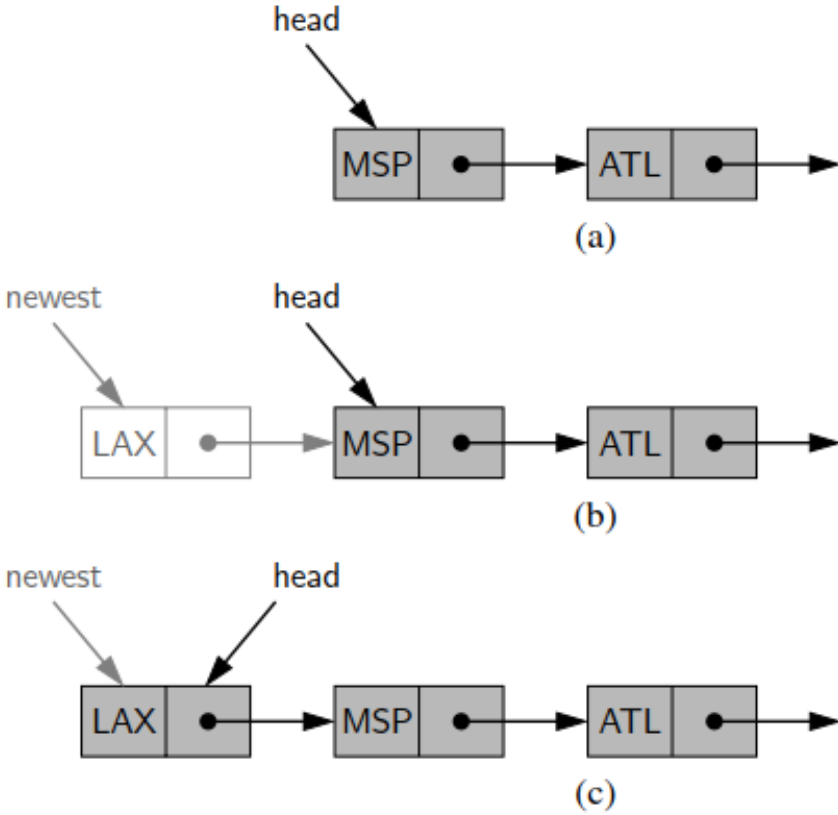


Упрощенная иллюстрация однонаправленного связанного списка (для упрощения, объекты, являющиеся элементами последовательности, представлены встроенными в узлы, а не внешними объектами, на которые ссылаются
узлы)



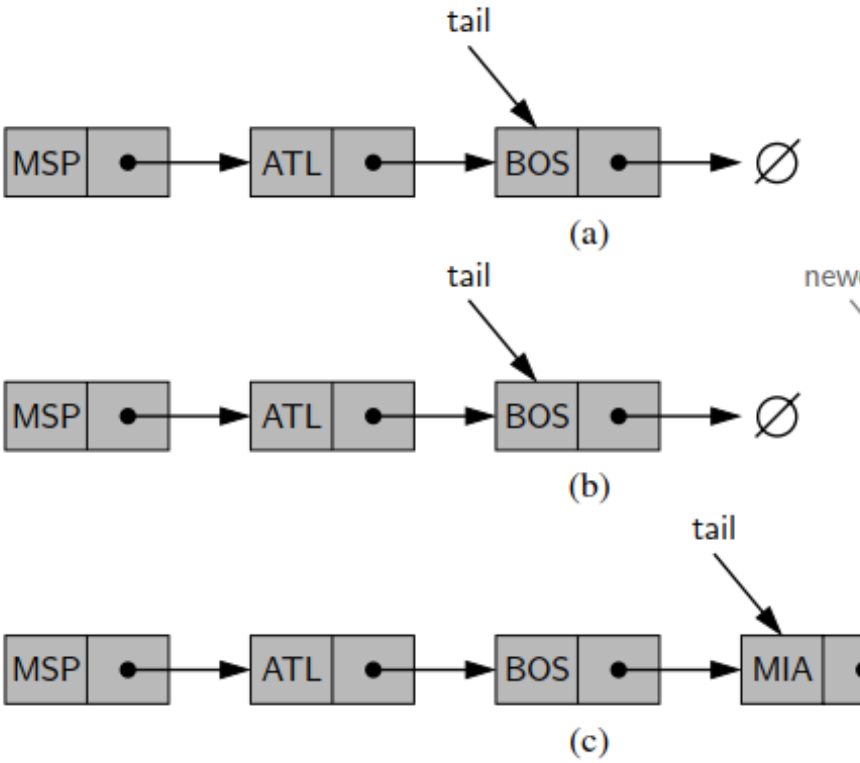
Процедура вставки узла в начало (head) однонаправленного связного списка (a - до вставки, b -

после создания нового элемента, c - после переприсвоения значения ссылки на первый узел).



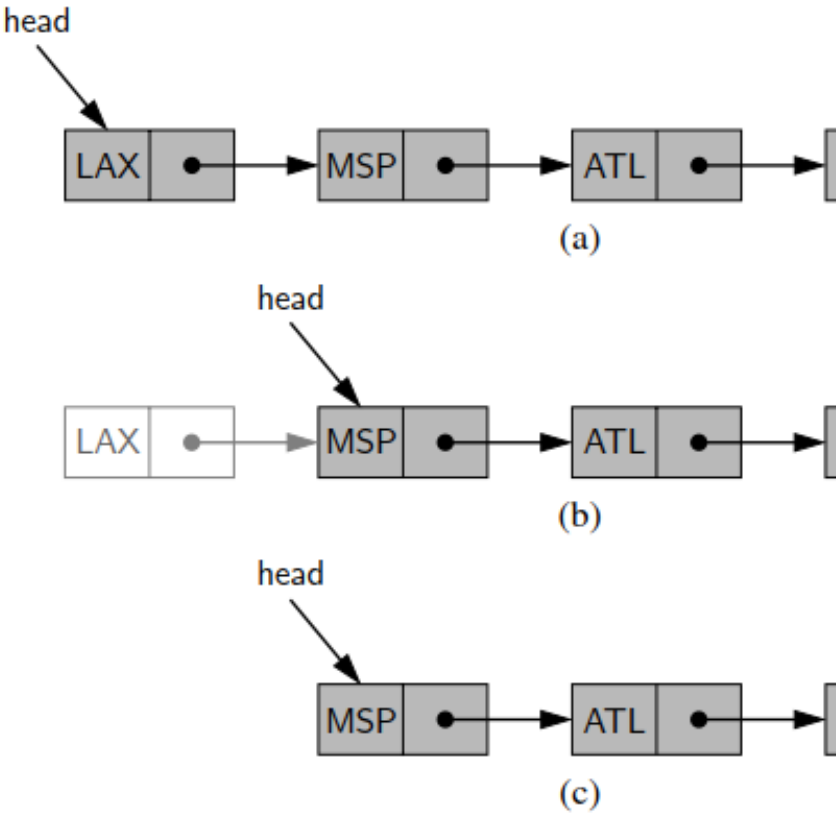
Процедура вставки узла в конец (tail) однонаправленного связного списка (a - до вставки, b - после создания нового элемента, c - после переприсвоения значения ссылки на последний

узел).



Процедура удаления узла из начала (head) однонаправленного связного списка (a - до удаления, b - после изменения ссылки на первый узел списка, c - итоговая

конфигурация).



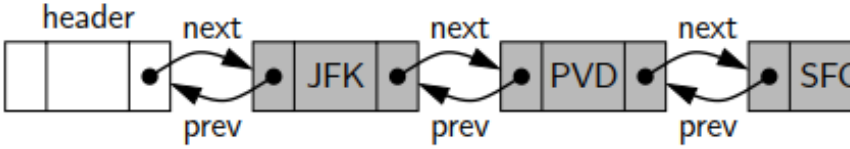
Производительность реализации стэка на основе связанного списка. Все значения определяют

консервативные оценки сложности.

Operation	Running Time
S.push(e)	$O(1)$
S.pop( )	$O(1)$
S.top( )	$O(1)$
len(S)	$O(1)$
S.is_empty( )	$O(1)$

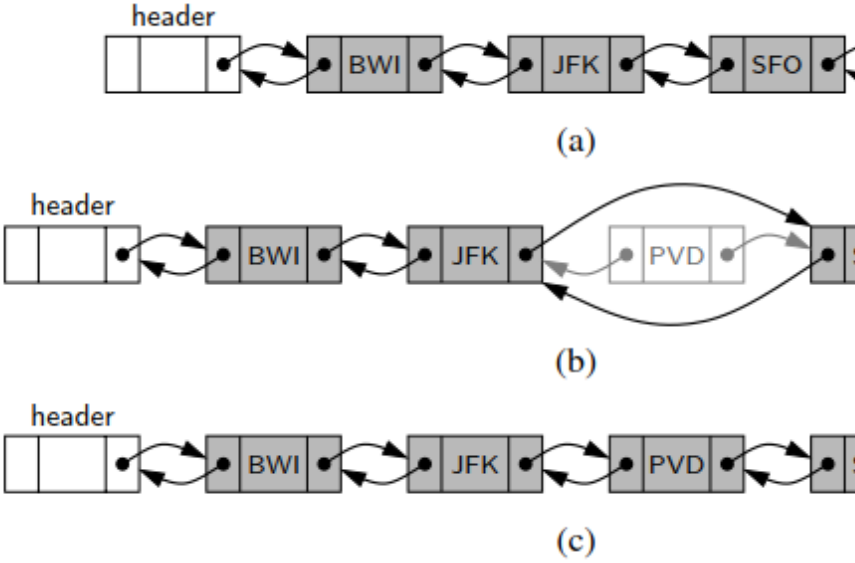
Двунаправленные связные списки

Пример двунаправленного связного списка использующего специальные узлы-ограничители (header,
trailer).

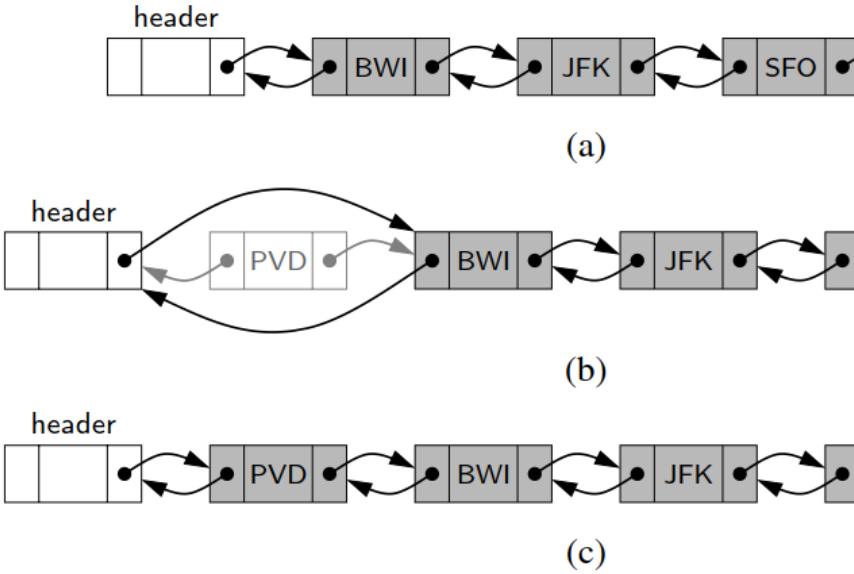


Добавление узла в двунаправленный связаный список с ограничетлями: a - до операции, b - после создания узла, c - после

операции.

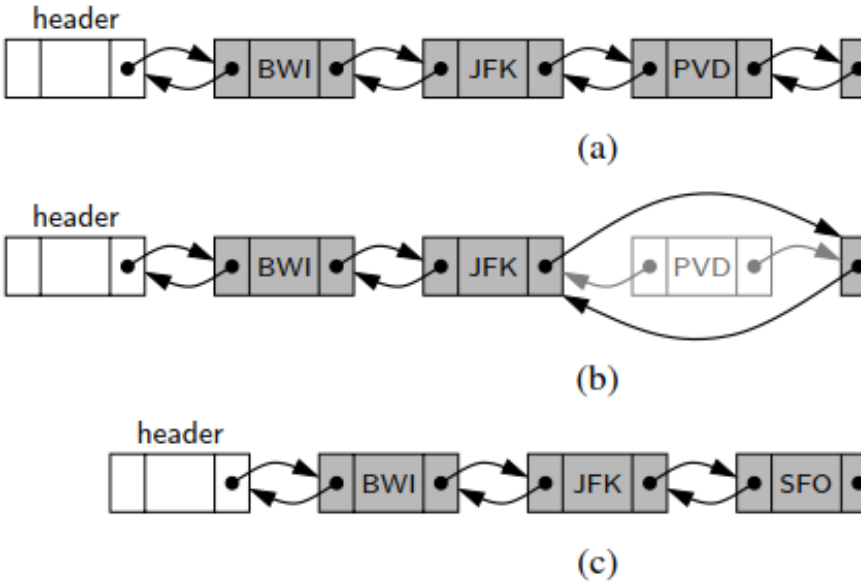


Добавление узла в начало двунаправленного связаного списка с ограничетлями: a - до операции, b - после создания узла, c - после операции.



Удаление элемента из двунаправленного связаного списка с ограничетлями: a - до операции, b - после удаления связей с узлом, c - после

операции.



Сравнение:

- Массивы обеспечивают за O(1) время доступа к элементу на основе целочисленного индекса. Напротив, для нахождения k − го элемента в связанном списке требуется O(k).

- Представления на основе массива обычно используют пропорционально меньше памяти, чем связанные структуры. Как списки на основе массивов, так и связанные списки являются ссылочными структурами, поэтому основная память для хранения фактических объектов, являющихся элементами, одинакова для обеих структур. Что отличается, так это вспомогательные объемы памяти, которые используются двумя структурами. Для

контейнера на основе массива из n элементов типичным наихудшим случаем может быть то, что недавно измененный динамический массив выделил память для 2n ссылок на объекты. При использовании связанных списков память должна быть выделена не только для хранения ссылки на каждый содержащийся объект, но и для явных ссылок, связывающих узлы. Таким образом, односвязный список длины n уже требует 2n ссылок (ссылка на элемент и следующая ссылка для каждого узла). С двусвязным списком – 3n ссылок.

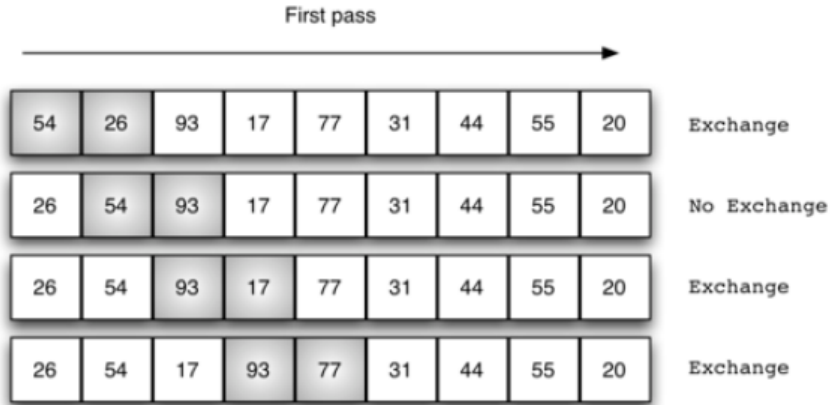
Преимущества последовательностей на основе ссылок

- Структуры на основе ссылок поддерживают O (1)-временные вставки и удаления в произвольных положениях.

22. Алгоритм обменной сортировки, сложность сортировки и возможности по ее улучшению.

Обменные сортировки (Bubble Sort)

Алгоритм прямого обмена основывается на сравнении и смене позиций пары соседних элементов. Процесс продолжается до тех пор, пока не будут упорядочены все элементы.
Пример первого прохода алгоритма сортировки пузырьком:




```
# Сортировка пузырьком:
def bubble_sort(l):
    for i in range(len(l) - 1, 0, -1):
        for j in range(i):
            if l[j] > l[j + 1]:
                l[j],j[j+1]=l[j+1],l[j]
    return l
test_list=[54, 26, 93, 17, 77, 31, 44, 55, 20]
bubble_sort(test_list)
```

Out[14]: [17, 20, 26, 31, 44, 54, 55, 65, 77, 93]

Введение переменной flag для сокращения времени работы, flag показывает отсортирован ли массив.

```
# Сортировка пузырьком:
def bubble_sort_2(l):
    for iin range(len(l) - 1, 0, -1):
        flag = False
        for j in range(i):
            if l[j] > l[j + 1]:
                l[j],l[j+1]=l[j+1],l[j]
                flag = True
        if not flag:
```

```
        return l
bubble_sort_2(test_list)
Out[16]: [17, 20, 26, 31, 44, 54, 55, 65, 77, 93]
```

Алгоритм имеет среднюю и максимальную временные сложности $O(n2)$ (два вложенных цикла, зависящих от n линейно). Введение переменной Flag и прерывание работы в случае отсортированного массива позволяет свести минимальную временную сложность к $O(\mathrm{n})$. Отметим одну особенность приведенного алгоритма: легкий пузырек снизу поднимется наверх за один проход, тяжелые пузырьки опускаются с минимальной скоростью: один шаг за итерацию.

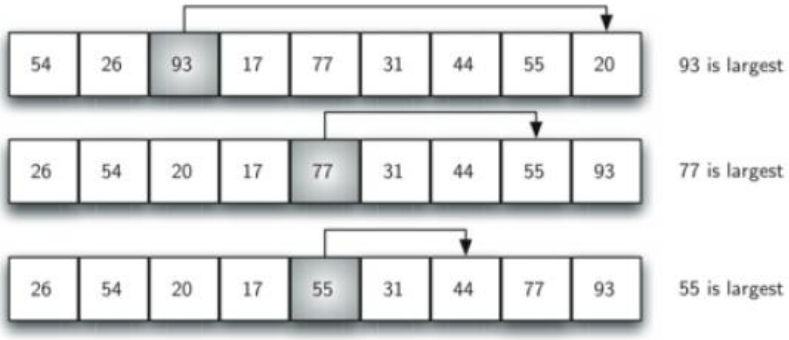
23. Алгоритм сортировки выбором, сложность сортировки и возможности по ее улучшению.

Сортировка выбором (извлечением) (Selection Sort)

Массив делится на уже отсортированную часть:
A_{i+1},A_{i+2}...A_nAi+1,Ai+2…An

и неотсортированную:
A_1,A_2,...,A_iA1,A2,…,Ai.

На каждом шаге извлекается максимальный элемент из неотсортированной части и ставится в начало отсортированной части. Оптимизация по сравнению с сортировкой пузырьком происходит за счет выполнения только одного обмена за каждый проход через список. В результате каждого прохода находится наибольшее значение в неотсортированной части и устанавливается в корректное место отсортированного фрагмента массива.



Пример работы алгоритма сортировки выбором:

```
def selection_sort(l):
   for fill_slot in range(len(a_list) - 1, 0, -1):
      pos_of_max = 0
      for location in range(1, fill_slot + 1):
         if  l[location] > l[pos_of_max]:
            pos_of_max = location
     l[fill_slot], l[pos_of_max]= l[pos_of_max], l[fill_slot]
         return a_list
test_list
Out[18]:  [54, 26, 93, 17, 77, 31, 44, 55, 20, 65]
In  [19]:
selection_sort(test_list)
Out[19]:  [17, 20, 26, 31, 44, 54, 55, 65, 77, 93]
```

С учетом того, что количество рассматриваемых на очередном шаге элементов уменьшается на единицу, общее количество операций:

$(n-1)+(n-2)+(n-3)+...+1=1/2(n2-n)=O(n^2)$.

По сравнению с обменной сортировкой:

- (+) сущенственно меньше перестановок элементов $O(n)$ по сравнению $O(n\text{^}2)$
- (-) нет возможности быстро отсортировать почти отсортированный массив

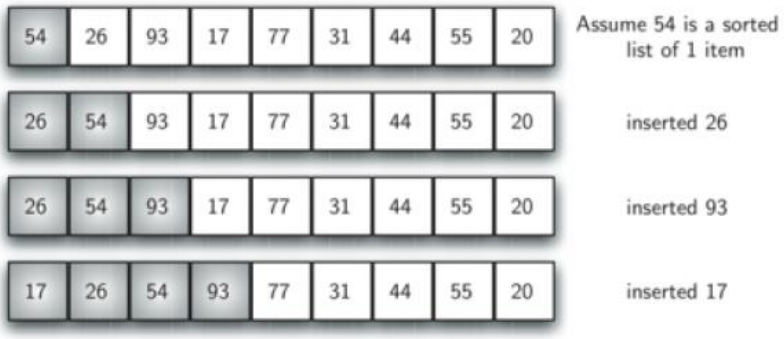
Естественной идеей улучшения алгоритма выбором является идея использования информации, полученной при сравнении элементов при поиске максимального (минимального) элемента на предыдущих шагах.
В общем случае, если n – точный квадрат, можно разделить массив $\sqrt{}$n групп по $\sqrt{}$n элементов и находить максимальный элемент в каждой подгруппе. Любой выбор, кроме первого, требует не более чем $\sqrt{}$n–2 сравнений внутри группы ранее выбранного элемента плюс $\sqrt{}$n–1сравнений среди "лидеров групп“. Этот метод получил название квадратичный выбор общее время его работы составляет порядка $O(n\sqrt{}\text{n})$ что существенно лучше, чем $O(n\text{^}2)$.

24. Алгоритм сортировки вставками, его сложность. Алгоритм быстрого поиска в отсортированном массиве. Сложность поиска в отсортированном и не отсортированном массиве.

Сортировка вставками (Insertion Sort)

Массив делится на 2 части: отсортированную и неотсортированную. На каждом шаге берется очередной элемент из неотсортированной части и включается в отсортированную. Простое включение предполагает, что отсортировано начало массива $A1,A2,\ldots,Ai$–1A1,A2,…,Ai–1, остаток массива $Ai,\ldots,An$Ai,…,An – неотсортирован. На

очередном шаге Ai включается в отсортированную часть на соответствующее место.
Пример работы алгоритма сортировки включением:




```
def insertion_sort(l):
    for index in range(1, len(l)):
        current_value = l[index]
        position = index
        while position > 0 and l[position - 1] > current_value:
            l[position] = l[position - 1]
            position -= 1
        l[position] = current_value
    return l
test_list
Out[21]: [54, 26, 93, 17, 77, 31, 44, 55, 20, 65
]
In [22]:
insertion_sort(list(test_list))
Out[22]: [17, 20, 26, 31, 44, 54, 55, 65, 77,
93]
```

Алгоритм имеет сложность $O(n\text{^}2)$, но в случае исходно отсортированного массива внутренний цикл не будет выполняться ни разу, поэтому метод имеет в этом случае временную сложность $O(n)$.

- (+) является эффективным алгоритмом для маленьких наборов данных
- (+) на практике более эффективен чем остальные простые квадратичные сортировки

Поиск:

Сравнение количества сравнений, используемых при **последовательном** поиске в **несортированном** списке:

Case	Best Case	Worst Case
item is present	1	n
item is not present	n	n

Сравнение количества сравнений, используемых при **последовательном** поиске в **отсортированном** списке:

Case	Best Case	Worst Case
item is present	1	n
item is not present	1	n

Бинарный поиск является быстрым алгоритмом поиска в отсортированном массиве
Пример проведения двоичного поиска в отсортированном списке целых чисел:

```
def binary_search(a_list, item):
    first = 0
    last = len(a_list) - 1
    while first <= last:
```

```
    midpoint = (first + last) // 2
    if a_list[midpoint] == item:
      return True
    else:
      if item < a_list[midpoint]:
        last = midpoint - 1
      else:
        first = midpoint + 1
  return False
```

Сравнение количества оставшихся для рассмотрения элементов в зависимости от количества выполненных операций сравнения:

Comparisons	Approximate Number Of Items Left
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	...
i	$\frac{n}{2^i}$

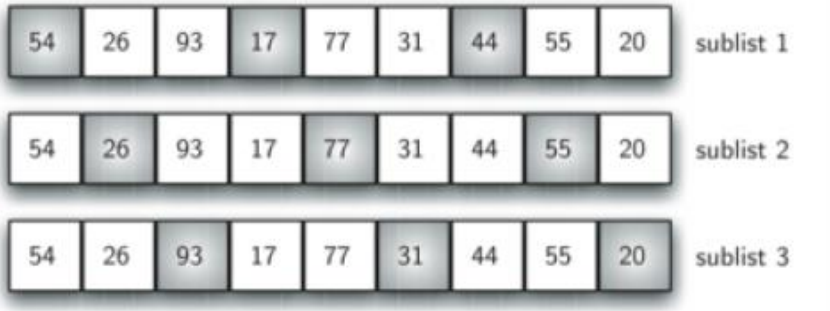
Бинарный поиск в отсортированном списке имеет сложность O(ln n).

25. Алгоритм сортировки Шелла, сложность сортировки и возможности по ее улучшению.

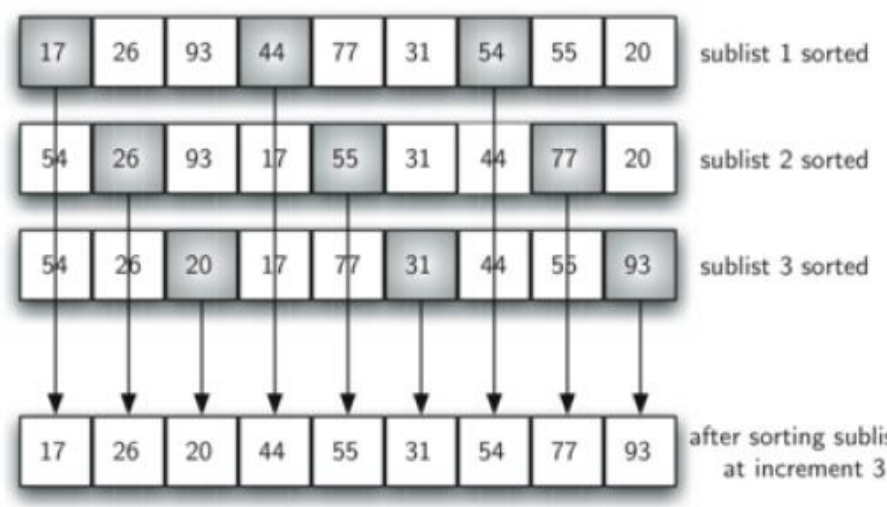
Сортировка Шелла (Shell Sort)

Сортировка Шелла является модификацией алгоритма сортировки включением, которая состоит в следующем: вместо включения A[i] в подмассив предшествующих

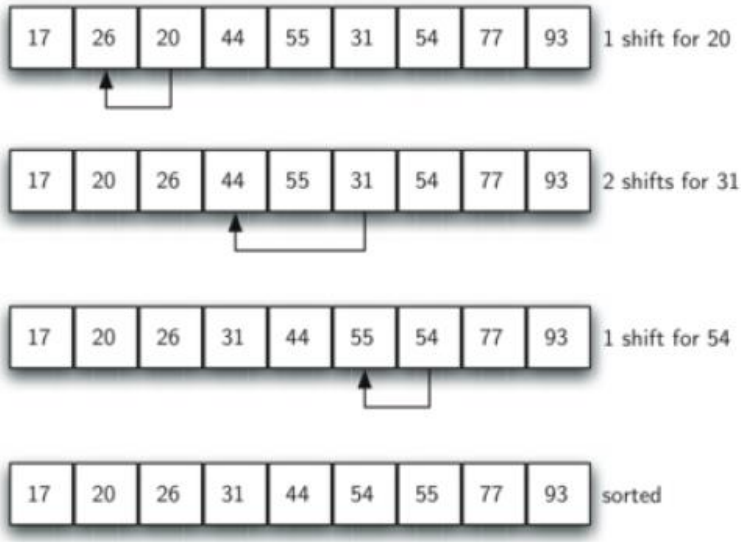
ему элементов, его включают в подсписок, содержащий элементы A[i-h], A[i-2h], A[i-3h] и тд, где h – положительная константа. Таким образом, формируется массив, в котором «h-серии» элементов, отстоящих друг от друга на h, сортируются отдельно. Процесс возобновляется с новым значением h, меньшим предыдущего. И так до тех пор, пока не будет достигнуто значение h=1.
Пример работы сортировки Шелла (разбиение исходго массива с шагом 3):



Пример работы сортировки Шелла с шагом 3 (после сортировки каждого подсписка):



Пример работы последней стадии сортировки Шелла с шагом 1 (сортировка вставкой):



 # Модификация сортировки вставкой для подмассива с шагом gap и смещением start:

```
def gap_insertion_sort(a_list, start, gap):
    for i in range(start + gap, len(a_list), gap):
        current_value = a_list[i]
        position = i
        while position >= gap and a_list[position - gap] > current_value
:
            a_list[position] = a_list[position - gap]
            position = position - gap
        a_list[position] = current_value
def shell_sort(a_list):
    sublist_count = len(a_list) // 2
    while sublist_count > 0:
        for start_position in range(sublist_count):
            gap_insertion_sort(a_list, start_position, sublist_count)
        print("After inc. of size", sublist_count, "Lst:", a_list)
        sublist_count = sublist_count // 2
shell_sort(list(test_list))
```

```
After inc. of size 5 Lst: [31, 26, 55, 17, 65, 5
4, 44, 93, 20, 77]
After inc. of size 2 Lst: [20, 17, 31, 26, 44, 5
4, 55, 77, 65, 93]
After inc. of size 1 Lst: [17, 20, 26, 31, 44, 5
4, 55, 65, 77, 93]
```

В слегка модифицированных версиях данного метода сортировки применяются специальные элементы массива, называемые сигнальными метками. Они не принадлежат к собственно сортируемому массиву, а содержат специальные значения, соответствующие наименьшему возможному и наибольшему возможному элементам. Это устраняет необходимость проверки выхода за границы массива. Однако применение сигнальных меток элементов требует конкретной

информации о сортируемых данных, что уменьшает универсальность функции сортировки.

Временная сложность для алгоритма Шелла – $O(n\text{^}4/3)$ и $\Theta(n\text{^}7/6)$, среднее число перемещений ~ $1{,}66n\text{^}1{,}25$.

Количество перестановок элементов по результатам экспериментов со случайным массивом иллюстрируется следующей таблицей.

Размерность	n = 25	n = 1000	n = 100000
Сортировка Шелла	50	7700	2 100 000
Сортировка простыми вставками	150	240 000	2.5 млрд.

26. Алгоритм быстрой сортировки, сложность сортировки и возможности по ее улучшению.

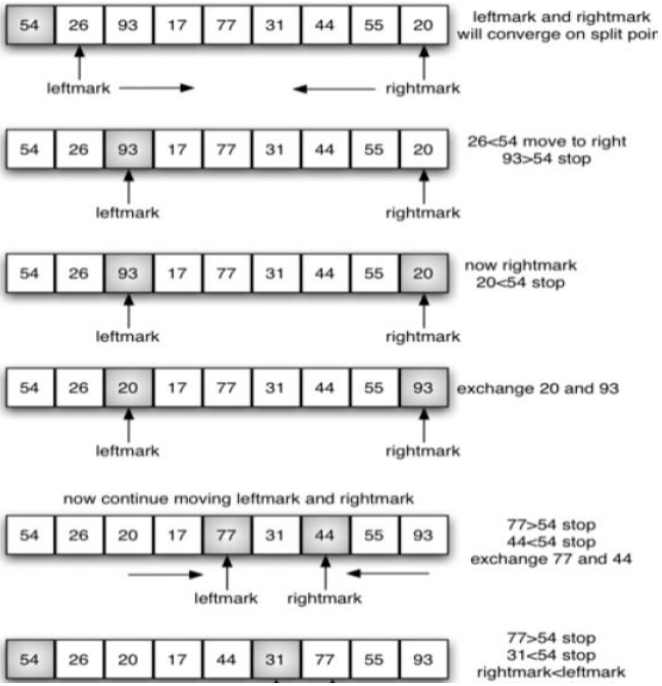
Быстрая сортировка (Quick Sort)
Быстрая сортировка – это алгоритм сортировки, время работы которого для входного массива из n чисел в наихудшем случае равно $O(n2)$. Несмотря на такую медленную работу в наихудшем случае, этот алгоритм на практике зачастую оказывается оптимальным благодаря тому, что в среднем время его работы намного лучше: $O(n\ln n)$. Алгоритм обладает также тем преимуществом, что сортировка в нем выполняется без использования дополнительной памяти, поэтому он хорошо работает даже в средах с виртуальной памятью.

Алгоритм быстрой сортировки является реализацией парадигмы «разделяй и властвуй». Разделение исходного массива осуществляется по следующему принципу:

1. Выбрать наугад какой-либо элемент массива – x
2. Просмотреть массив слева направо, пока не обнаружим элемент Ai > x
3. Просмотреть массив справа налево, пока не встретим Ai < x
4. Поменять местами эти два элемента
5. Процесс просмотра и обмена продолжается, пока указатели обоих просмотров не встретятся

Пример работы быстрой сортировки (выбор элемента для разделения массива):

Пример работы быстрой сортировки (разделения массива на две части: со значениями меньшими и большими чем у разделяющего элемента):




```
def quick_sort(a_list):
    quick_sort_helper(a_list, 0, len(a_list) - 1)
    return a_list
def quick_sort_helper(a_list, first, last):
    if first < last:
        split_point = partition(a_list, first, last)
        quick_sort_helper(a_list, first, split_point - 1)
        quick_sort_helper(a_list, split_point + 1, last)

def partition(a_list, first, last):
    pivot_value = a_list[first]
    left_mark = first + 1
    right_mark = last
    done = False
    while not done:
```

```
    while left_mark <= right_mark and a_list[left_mark] <= pivot_va
lue:
        left_mark = left_mark + 1
    while a_list[right_mark] >= pivot_value and right_mark >= left_
mark:
        right_mark = right_mark - 1
    if right_mark < left_mark:
        done = True
    else:
        a_list[left_mark], a_list[right_mark] = a_list[right_mark], a_list[
left_mark]
  a_list[first], a_list[right_mark] = a_list[right_mark], a_list[first]
  return right_mark
```

Ожидаемое число обменов в быстром алгоритме – $(n-1)/6$, общее число сравнений $n\ln n$. Наихудший случай – в качестве элемента для разбиения x выбирается наибольшее из всех значений в указанной области, т.е. левая часть состоит из $n-1$ элементов, а правая из 1, тогда временная сложность становится пропорциональна $n\text{^}2$.

Все улучшения можно разделить на три группы: придание алгоритму устойчивости, устранение деградации производительности специальным выбором опорного элемента, и защита от переполнения стека вызовов из-за большой глубины рекурсии при неудачных входных данных.

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

Двоичное (бинарное) дерево (binary tree) — 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.

 # Фрагмент реализации бинарного дерева с помощью представления в виде узлов и ссылок:

```
class BinaryTree:
  def __init__(self, root):
    self.key = root
    self.left_child = None
    self.right_child = None

  def insert_left(self,new_node):
    if self.left_child == None:
      self.left_child = BinaryTree(new_node)
    else:
      t = BinaryTree(new_node)
      t.left_child = self.left_child
      self.left_child = t

  def insert_right(self,new_node):
    if self.right_child == None:
      self.right_child = BinaryTree(new_node)
    else:
      t = BinaryTree(new_node)
      t.right_child = self.right_child
      self.right_child = t
```

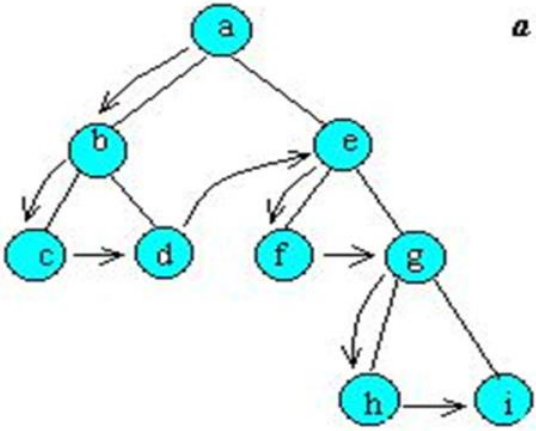
```
    def get_right_child(self):
        return self.right_child

    def get_left_child(self):
        return self.left_child

    def set_root_val(self,obj):
        self.key = obj

    def get_root_val(self):
        return self.key

    def __str__(self):
        return '{} ({}, {})'.format(self.get_root_val(), str(self.get_left_chil
d()), str(self.get_right_child()))
r = BinaryTree('a')
print(r)
a (None, None)
In [13]:
r.insert_left('b')
print(r)
print(r.get_left_child())
print(r.get_left_child().get_root_val())
a (b (None, None), None)
b (None, None)
b
```

Обход бинарных деревьев

Прямой (preorder) порядок обхода дерева:

1. Первым просматривается корневой узел

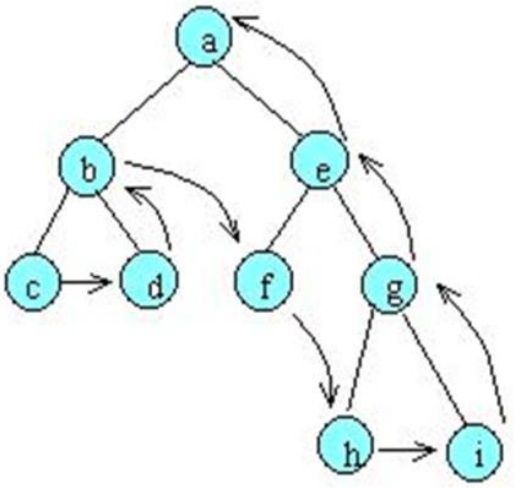
2. Затем производится рекурсивный прямой обход левого поддерева.
3. Затем производится рекурсивный прямой обход правого поддерева.



a b c d e f g h i

Обратный (postorder) порядок обхода дерева:

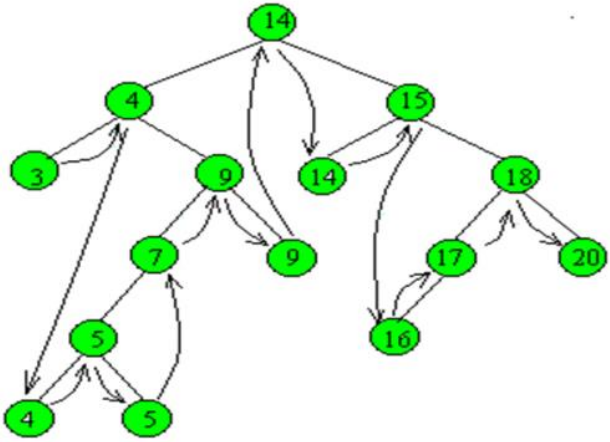
1. Первым производится рекурсивный обратный обход левого поддерева.
2. Затем производится рекурсивный обратный обход правого поддерева.
3. Затем просматривается корневой узел



c d b f h i g e a

Симметричный (inorder) порядок обхода дерева:

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



Прохождение в симметричном порядке:
3, 4, 4, 5, 5, 7, 9, 9, 14, 14, 15, 16, 17, 18, 20

28. Абстрактный тип данных - ассоциативный массив и принцип его реализации на основе хэш-таблиц и хэш-функций.

Абстрактный тип данных - ассоциативный массив

Словарь (ассоциативного массива (map, dictionary, associative array)) - абстрактная структура данных позволяющая хранить пары вида "ключ - значение" и поддерживающая операции добавления пары, а также поиска и удаления пары по ключу. Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами. Ассоциативный массив с точки зрения интерфейса удобно рассматривать как обычный

массив, в котором в качестве индексов можно использовать не только целые числа из определенного диапазона, но и значения других типов — например, строки. В Python словарь реализуется при помощи dict(). Далее мы рассмотрим различные решения задачи реализации словаря.

Хеш-таблица

Хеш-таблица представляет собой эффективную структуру данных для **реализации словарей**. Хотя на поиск элемента в хеш-таблице может в наихудшем случае потребоваться столько же времени, что и в связанном списке, а именно $O(n)$, на практике хеширование исключительно эффективно. При вполне обоснованных допущениях **математическое ожидание времени поиска** элемента в хеш-таблице составляет $O(1)$.

Хеш-таблица (hash table) представляет собой обобщение обычного массива. Если количество реально хранящихся в массиве ключей мало по сравнению с количеством возможных значений ключей, эффективной альтернативой массива с прямой индексацией становится **хеш-таблица**, которая обычно использует массив с размером, пропорциональным количеству реально хранящихся в нем ключей. Вместо непосредственного использования ключа в качестве индекса массива, **индекс вычисляется по значению ключа**. Идея хеширования состоит в использовании некоторой частичной информации, полученной из ключа, т.е. вычисляется хеш-адрес $h(key)$, который используется для индексации в хеш-таблице.

Когда множество К хранящихся в словаре ключей гораздо меньше пространства возможных ключей U, хеш-таблица требует существенно меньше места, чем

таблица с прямой адресацией. Точнее говоря, требования к памяти могут быть снижены до $\Theta(|K|)$, при этом время поиска элемента в хеш-таблице остается равным $O(1)$. *Надо только заметить, что это граница среднего времени поиска, в то время как в случае таблицы с прямой адресацией эта граница справедлива для наихудшего случая.*

Хеш-таблица это коллекция, хранящая проиндексированные элементы (значения). Каждая позиция в хеш-таблице - слот таблицы (slot, bucket) может содержать элемент адресованный целочисленным не отрицательным индексом внутри таблицы. Т.е. в таблице есть слот с индексом 0, 1 и т.д. При создании все слоты в хеш-таблице пустые. В Python хеш-таблицу можно реализовать в виде списка заполненного значениями None или их заменителями (для случаев, если пользователю нужно хранить в хеш-таблице значения None).

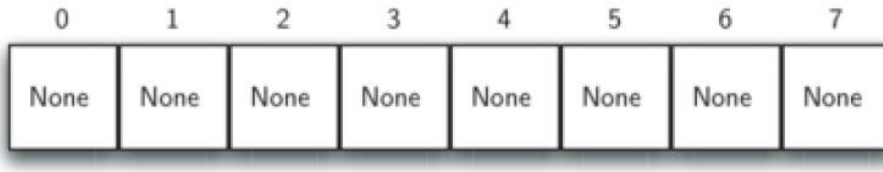
В случае прямой адресации элемент с ключом k хранится в ячейке k. При хешировании этот элемент хранится в ячейке $h(k)$, т.е. мы используем хеш-функцию h для вычисления ячейки для данного ключа k. Функция h отображает пространство ключей U на ячейки хеш-таблицы $\mathrm{T}[0..m-1]$

Величина $h(k)$ называется хеш-значением ключа k.

Пример

$T[0,\ldots,10]$; $h(k)=k \bmod 11$

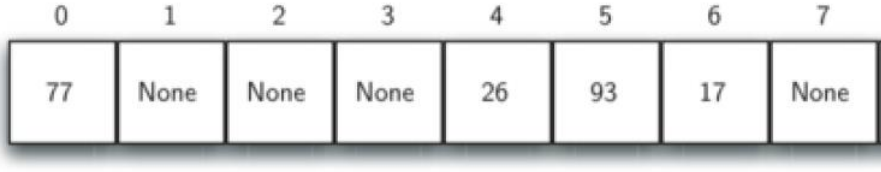
Пустая хеш-таблица:




```
In [40]:
def h(k):
    return k % 11
keys1 = [54, 26, 93, 17, 77, 31]
 [h(k) for k in keys1]
```

Out[42]: [10, 4, 5, 6, 0, 9]

Хеш-таблица с 6ю элементами:



При построении хеш-таблиц есть одна проблема: два ключа могут быть хешированы одну и ту же ячейку. Такая ситуация называется **коллизией.**

 # пример коллизии:

h(12), h(23)

Т.к. h является детерминистической и для одного и того же значения k всегда дает одно и то же хеш-значение $h(k)$, то поскольку $|U|>m$, должно существовать как минимум два ключа, которые имеют одинаковое хеш-значение. Таким образом, полностью **избежать коллизий невозможно** в принципе.

Используются два подхода для борьбы с этой проблемой:

- выбор хеш-функции снижающей вероятность коллизии;
- использование эффективных алгоритмов разрешения коллизий.

Хеш-функция

Хеш-функция выполняет преобразование массива входных данных произвольной длины (ключа, сообщения) в (выходную) битовую строку установленной длины (хеш, хеш-код, хеш-сумму). Хеш-функции применяются в следующих задачах:

- построение ассоциативных массивов;
- поиске дубликатов в сериях наборов данных;
- построение уникальных идентификаторов для наборов данных;
- вычислении контрольных сумм от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;
- сохранении паролей в системах защиты в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся обратной по отношению к использованной хеш-функции);
- выработке электронной подписи (на практике часто подписывается не само сообщение, а его «хеш-образ»); и многих других.

Для решения различных задач требования к хеш-функциям могут очень существенно отличаться.

"Хорошая" хеш-функция должна удовлетворять двум свойствам:

- быстрое вычисление;
- минимальное количество коллизий.

Для обеспечения минимального количества коллизий хеш-функция удовлетворяет (приближенно) предположению простого **равномерного хеширования**: для каждого ключа равновероятно помещение в любую из m ячеек, независимо от хеширования остальных ключей. К сожалению, это условие обычно невозможно проверить, поскольку, как правило, распределение вероятностей, в соответствии с которым поступают вносимые в таблицу ключи, неизвестно; кроме того, вставляемые ключи могут не быть независимыми.
При построении хеш-функции хорошим подходом является подбор функции таким образом, чтобы она никак **не коррелировала с закономерностями,** которым могут подчиняться существующие данные. Например, мы можем потребовать, чтобы **"близкие" в некотором смысле ключи давали далекие хеш-значения** (например, хеш функция для подряд идущих целых чисел давала далекие хеш-значения). В некоторых приложениях хеш-функций требуется противоположное свойство - непрерывность (близкие ключи длолжны порождать близкие хеш-значения).
Зачастую расчет хеш-функции $h(k)$ можно представить в виде двух операций: рассчета хеш-кода который превращает ключ k в целое число и функции компрессии, которая преобразует хеш-код в целое число в заданном диапазоне $[0,m-1]$.

Метод деления

Построение хеш-функции методом деления состоит в отображении ключа k в одну из ячеек путем получения остатка от деления k на m, т.е. хеш-функция имеет вид $h(k)=k \mathrm{mod} m$. Зачастую хорошие результаты можно получить, выбирая в качестве значения mm простое число, достаточно далекое от степени двойки.

Метод MAD

Хеш-функция multiply-add-and-divide (часто именуемая как MAD) преобразует целое число kk по следующему алгоритму. У хеш-функци имеются следующие параметры: pp - большое простое число, $a \in \{1,2,...,p-1\}$a∈{1,2,…,p–1} и $b \in \{0,1,...,p-1\}$b∈{0,1, …,p–1}, mm - количество значений в диапозоне значений хеш-функции.

$$h_{a,b}(k)=((ak+b) \mathrm{mod} p) \mathrm{mod} m$$

Данную хеш-функцию (семейство хеш-функций) можно использовать для **универсального хеширования**.

Полиномиальная хеш-функция

Приведенная ранее функция построения хеш-кода, основанная на суммировании (или опреации xor) хеш-кодов, плохо подходит для работы с символьными строками и другими объектами различной длины, которые могут быть представлены в виде кортежа $(x_0,x_1,...,x_{n-1})$, в котором позиция элемента x_i имеет значение, т.к. такой хеш-код создает коллизии для строк (последовательностей) с одинаковым составом элементов.

Такого рода коллизии не будут возникать в хеш-функции, которая учитыват положение элементов в массиве входных данных. Примером такой хеш-функции является функция, использующая константу a ($a \neq 0$) при построении хеш-функции вида:

$$x_0 a^{n-1} + x_1 a^{n-2} + \ldots + x_{n-2} a + x_{n-1}.$$

Т.е. это полином, использующий элементы массива входных данных $(x_0, x_1, \ldots, x_{n-1})$ в качестве коэффициентов.

Стандартным способом для получения хеш-кода в Python является встроенная функция **hash(x)**. Она возвращает целочисленное значение для объекта x. Однако, в Python **только неизменяемые типы** данных могут возвращать значение хеш-кода. Это ограчиние гарантирует, что хеш-код для объека не изменится во время его жизни. Это свойство очень важно для корректной работы при использовании хеш-кодов объектов в хеш-таблицах, например в dict().