1. **Введение в курс, его цели.**

Научиться сравнивать различные варианты решения задач по ключевым критериям и выбирать наиболее эффективный в текущей ситуации вариант.

У большинства задач можно выделить несколько вариантов решения, которые дают один и тот же результат, но при этом занимают разное время выполнения, по -разному потребляют память. При этом, сказать какое решение эффективнее однозначно нельзя, поскольку в одним условиях нам важна в первую очередь скорость работы скрипта, а в других объем выделяемой памяти. Наша задача научиться замерять характеристики времени и памяти для разных вариантов решения задачи, чтобы потом выбрать наиболее подходящий в текущей ситуации. При этом нам важно обратить внимание на аналитические техники, которые в том числе должны давать результаты, не зависящие от характеристик, например, вычислительной машины. При выборе решения конкретной задачи часто приходится идти на компромисс.

Развитие алгоритмического мышления, т.е. научиться представлять весь ход решения задачи от ее постановки до получения итогового результата.

Умение мысленно «разложить» задачу на составляющие является важным умением программиста-алгоритмизатора, для развития которого необходима постоянная практика. На курсе мы в каждом уроке обязательно практикуемся для отработки знаний, полученных по определенной теме.

Научиться перекладывать сформулированный алгоритм на язык реализации.

Любой алгоритм проходит несколько стадий, от его словесного описания и псевдокода до реализации на языке программирования. Задача программиста-алгоритмизатора – понять, какие необходимо применить средства языка программирования для успешного воплощения алгоритма.

Освоить фундаментальные алгоритмы, не зависящие от языка реализации, например, решето Эратосфена, алгоритмы сортировки и т.д.

Существую алгоритмы, имеющие, например, математическое доказательство и практическую ценность при решении реальных задач. Эти фундаментальные алгоритмы можно «переложить» на различные языки программирования. Но прежде необходимо понять суть самих таких алгоритмов.

Изучить возможностей применения приемов алгоритмизации для реальных задач.

Мы проходим курс с единственной глобальной целью – получить знания, которые мы сможем применить на практике. Нужно понять какие, например, фундаментальные алгоритмы мы могли бы применить при решении каких реальных задач.

1. **Еще раз о важности стиля кода.**

При решении задач важная не только семантическая правильность кода (программа должна давать ожидаемый результат), но и его стилевая корректность, т.е. соответствие стандарту PEP-8. Это способствует формированию положительной репутации для разработчика, его успешному прохождению собеседования. Да и просто правильный стиль кода упрощает его последующую модификацию и поддержание в рабочем состоянии.

1. **Об анализе алгоритмов**

**Введение**

На тему анализа алгоритмов мы подробнее поговорим на уроках 4 и 6. Пока же нужно понять, что это действительно важно. При этом для оценки времени выполнения мы будем вычислять не только его абсолютные значения, но и описывать время выполнения с помощью нотации «О большое».

Также мы обязательно узнаем, что, например, у многих стандартных операций, выполняемых над списками и словарями в Python, можно оценить «О большое».

Также мы узнаем о влиянии реализации данных в Python на анализ алгоритмов.

И, наконец, мы разберемся, как определять критерии оценки простые Python-программ.

**Поехали!**

Представим, у нас есть две программы, которые решают одну и ту же задачу. Как понять, что одна программа лучше другой?

Давайте вспомним, что алгоритм – это универсальная пошаговая инструкция для решения задачи. Это метод, который на некоторых исходных данных выдает нужный результат. Программа – это подход, как алгоритм переложен на язык программирования. Для одного и того же алгоритма мы можем написать множество версий программ, которые зависят от языка и разработчика, его индивидуальных особенностей написания кода.

**Листинг 1. task\_1.py**

|  |
| --- |
| *""" Вычисление суммы первых n целых чисел """* **def** get\_sum\_1(n):  *"""  В основе идеи алгоритма - переменная-счетчик, инициализируемая нулем  и к которой в процессе решения задачи прибавляются числа, перебираемые в цикле* **:param** *n:* **:return***:  """* res = 0  **for** i **in** range(1, n + 1):  res = res + i  **return** res   print(get\_sum\_1(10))   **def** get\_sum\_2(obj):  *"""  Текущее решение является неудачным из-за избыточного присваивания,  а также неудачного выбора имен переменных* **:param** *obj:* **:return***:  """* var = 0  **for** part **in** range(1, obj + 1):  dec = part  var = var + dec  **return** var   print(get\_sum\_2(10)) |

Но эти две функции простые и без труда можно определить наиболее удачную из них. А что делать, если мы говорим о сложных скриптах?

Тогда нужно сказать, что лучшим будет тот, который наиболее эффективно использует ресурсы или ему их требуется меньше.

Соответственно, нужно найти такой алгоритм, но по каким критериям их сравнивать?

Это в прежде всего время и память, требуемые алгоритмы для решения задачи.

Как же измерить время? Воспользуемся самым простым способом, который у нас есть благодаря модулю time.

**Листинг 2. task\_2.py**

|  |
| --- |
| *""" Проведение сравнительных замеров """* **import** time   **def** check\_1(n):  *"""  Фиксируем отсечки времени до и после выполнения основной логики* **:param** *n:* **:return***: кортеж из результата ф-ции и затраченного времени  """* start\_val = time.time()  res = 0   **for** i **in** range(1, n + 1):  res = res + i   end\_val = time.time()   **return** res, end\_val - start\_val   *# результат хорошо повторяем # для n = 10000 # затрачивается примерно 0.000979 сек* **for** i **in** range(5):  print(**f'Операция заняла {**check\_1(10000)[1]**} сек'**)  print() *# для n = 100000 # затрачивается примерно 0.005877 сек* **for** i **in** range(5):  print(**f'Операция заняла {**check\_1(100000)[1]**} сек'**)  print() *# для n = 100000 # затрачивается примерно 0.065629 сек* **for** i **in** range(5):  print(**f'Операция заняла {**check\_1(1000000)[1]**} сек'**)  **""" На различных итерациях цикла даже на одной и той же машине мы получаем различные результаты. Почему?  Влияют многие факторы: 1) версия интерпретатора 2) разрядность ОС 3) текущая нагрузка процессора """** *# Получается абсолютные цифры различаются и на что же ориентироваться? # Можно выполнить подсчет средней величины времени # Хотя есть еще один вариант оценки, не привязанный к абсолютным цифрам # Но об этом позже* |

 Наконец, рассмотрим третий способ решения все той же задачи.

**Листинг 3. task\_3.py**

|  |
| --- |
| *""" Теперь воспользуемся формулой (n\*(n+1))/2 """* **import** time   **def** check\_3(n):   start\_val = time.time()   end\_val = time.time()   **return** (n\*(n+1))/2, end\_val - start\_val   print(check\_3(10))   **for** i **in** range(5):  print(**f'Операция заняла {**check\_3(10000)[1]**} сек'**)  **""" Стоит обратить внимание, что какое бы значение n мы не передали в функцию, итоговое время окажется экстремальном малым  Намного эффективнее других вариантов, не правда ли?  Давайте выявим некоторые особенности первых двух решений. 1) Они итеративные, а такие решения из-за повторения некоторого набора шагов потребуют больше времени. 2) Также мы заметили, что время итеративного решения увеличивается с ростом входного значения n  Вроде все понятно. Нужно использовать третий вариант. Но и здесь оказывается не все так просто. Если запустить третью ф-цию на разных машинах или реализовать алгоритм этой ф-ции на других языках программирования, то результаты скорее всего не будут идентичными. Чем старше будет машина, тем больше потребуется времени.   Теперь понятно, что получать оценку времени в цифровом выражении не так уж хорошо Ведь это время зависит от конкретной машины, программы, времени дня, компилятора и языка программирования. Было бы хорошо подобрать характеристику, не зависящую от этих параметров, но дающую понятные результаты.  Чтобы мы могли бы применить эту характеристику для оценки алгоритма самого по себе и сравнения одного алгоритма с другими. """** |

Самое время перейти к О-нотации.

1. **Нотация О-большое**

Т.к. мы уже поняли, что получать абсолютное время нет смысла и нужна какая-то другая характеристика, то самое время изучить нотацию О-большое, ведь именно в ее основе заложена эта характеристика.

Давайте подведем небольшие итоги. Мы хотим охарактеризовать эффективность алгоритма независимо от компьютера, ОС и других параметров. Тогда нам нужно оценить число операций и шагов, необходимых алгоритму. Если каждый такой шаг принять за базовую единицу вычисления, то общее время алгоритма будет оцениваться в виде набора операций, требуемых для решения задачи. Цель, алгоритмизатора – понять, размер задачи влияет на время работы алгоритма.

Но специалисты в области алгоритмизации предпочитают оценивать не точное число операций, а определять доминирующую часть, т.е. то выражение или блок, который дает наибольшее время выполнения. Т.е. с ростом задачи некая часть (доминирующая), определяющая время выполнения алгоритма, перекрывает все остальные выражения. Специалисту требуется определить ту часть, которая и дает наибольший рост. В этом в общем выражается смысл нотации «О-большое».

Представим, что для некоторого алгоритма точное число операций выражается формулой:



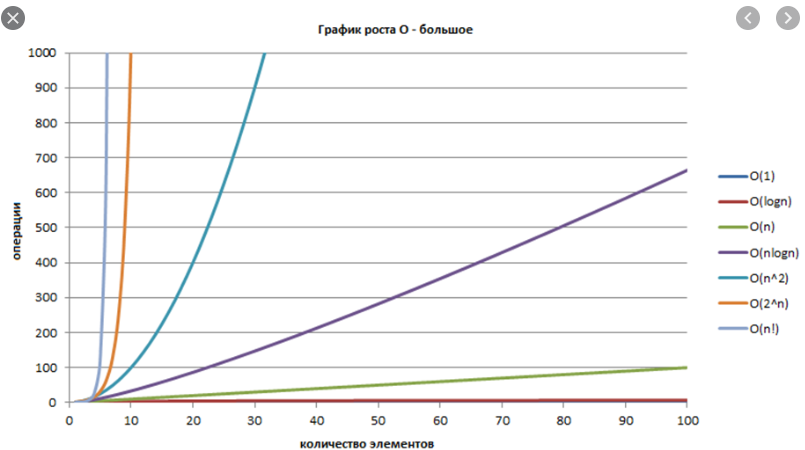
На небольших значениях n (например, 1 или 2) доминирующей частью является константа 1005. Но с ростом n ее роль начинает играть первое слагаемое. На очень больших n два другие слагаемые перестают иметь значимость при определении итогового результата. Подведем итоги! С ростом n другие слагаемые мы можем проигнорировать и ориентироваться только на первое. В нотации О-большое такая формула соответствует функции О(n^2). Это квадратичная сложность.

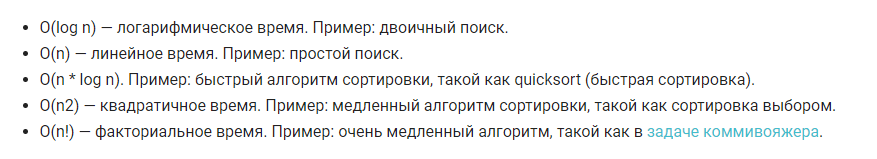
НО! Иногда производительность алгоритма определяется в большей степени точностью значений, нежели размером задачи. Для подобных алгоритмов нужно определять их эффективность для наилучшего, наихудшего, усредненного случая. В худшем случае мы говорим о таком наборе данных, на которым алгоритм «отрабатывает» наиболее плохо. Хотя различные коллекции данных для одного и того же алгоритма отличаются хорошей производительностью, в основном алгоритм обладает усредненной производительностью. Но в рамках данной нотации мы ориентируемся на худший случай.

В таблице 1 приведено сравнение функций для нотации О-большое.



На рис. 1 показаны графики функций из таблицы.



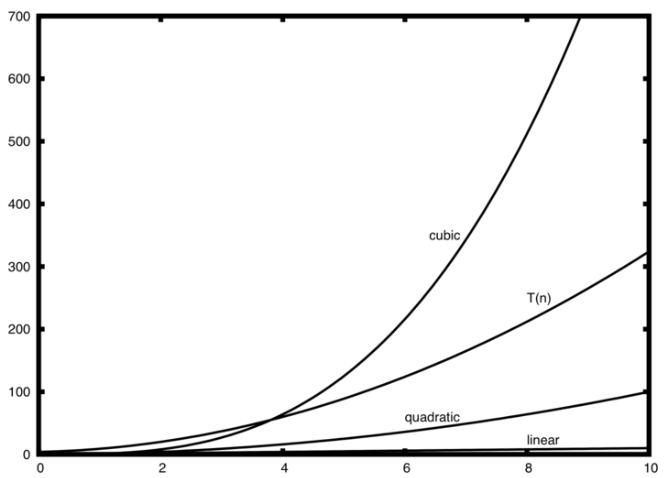


Нужно обратить внимание, что при небольших n эти ф-ции примерно совпадают и сложно сказать, кто доминирует, но с ростом уже начинают выявляться тенденции и соотношение ф-ций друг с другом.

**Листинг 4. task\_4.py**

|  |
| --- |
| *"""Проверка сложности"""  # первые три присваивания - это константа (3)* VAL\_A = 5 VAL\_B = 6 VAL\_C = 10 *# три присваивания выполняются n2 раз внутри вложенной итерации (3n2)* **for** i **in** range(10):  **for** j **in** range(10):  x = i \* i  y = j \* j  z = i \* j *# два присваивания, повторяющиеся n раз (2n)* **for** k **in** range(10):  w = VAL\_A\*k + 45  v = VAL\_B\*VAL\_B *# последний оператор присваивания (1)* VAL\_D = 33  *# T(n)=3+3n\*\*2+2n+1=3n\*\*2+2n+4 # O(n\*2)* |

Приведенный алгоритм характеризуется квадратичной сложностью.



Сравним несколько распространенных ф-ций нотации большое-О по сравнению с рассматриваемой функцией. Изначально наша ф-ция больше кубической, но с ростом n кубическая начинает доминировать. Можно увидеть и сделать вывод, что с ростом n наша ф-ция соответствует квадратичной ф-ции.

В листингах task\_5-task\_10 представлен код примеров с алгоритмами различной сложности:

**Листинг 5. task\_5.py**

|  |
| --- |
| *""" Сложность: O(2^n) Название: экспоненциальная  Время работы такого алгоритма удваивается с каждым очередным дополнением к набору данных Кривая роста этой функции экспоненциальная: сначала она очень пологая, а затем стремительно поднимается вверх.  Примером алгоритма с экспоненциальной сложностью может послужить рекурсивный расчет чисел Фибоначчи """* **def** fibo\_recur(number):  **if** number <= 1:  **return** number  **return** fibo\_recur(number - 2) + fibo\_recur(number - 1)   print(fibo\_recur(6))  *# для number = 6 результат 8 - 0, 1, 1, 2, 3, 5, 8* |

**Листинг 6. task\_6.py**

|  |
| --- |
| *""" Сложность: O(1) Название: постоянное время (сложность - константа)  Время работы алгоритма не зависит от объема входящих данных """* first\_lst = [**"one"**, **"two"**, **"three"**, **"four"**] second\_lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]   **def** get\_lst\_func(lst):  length = len(lst)  **return** length   print(get\_lst\_func(first\_lst)) print(get\_lst\_func(second\_lst)) |

**Листинг 7. task\_7.py**

|  |
| --- |
| *""" Сложность: O(n) Название: линейная сложность  Время возрастает линейно и прямо пропорционально количеству передаваемых данных """* first\_lst = [**"one"**, **"two"**, **"three"**, **"four"**] second\_lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]   **def** get\_lst\_func(lst):  **for** el **in** lst:  print(el)   get\_lst\_func(first\_lst) print() get\_lst\_func(second\_lst) |

**Листинг 8. task\_8.py**

|  |
| --- |
| *""" Сложность: O(n^2) Название: квадратичная сложность  Квадратичное время представляет алгоритм, производительность которого прямо пропорциональна квадрату размера входящих данных  Распространенный пример такого алгоритма — цикл, вложенный в другой цикл. По мере увеличения вложенности растет и временная сложность (O(n^3), O(n^4) и т. д.). """* obj\_lst = [[1, **'one'**], [2, **'two'**], [3, **'three'**], [4, **'four'**]]   **def** get\_lst\_func(lst):  **for** el\_lst **in** lst:  **for** el **in** el\_lst:  print(el)   get\_lst\_func(obj\_lst) |

**Листинг 9. task\_9.py**

|  |
| --- |
| *""" Сложность: O(log n) (основанием здесь можно пренебречь) Название: логарифмическое время  Яркий пример - обработка n-массивов двоинчым поиском  В отсортированном наборе данных выбирается серединный элемент и сравнивается с искомым значением. Если значение совпадает, поиск завершен.  Если искомое значение больше, чем значение серединного элемента, нижняя половина набора данных (все элементы с меньшими значениями, чем у нашего серединного элемента) отбрасывается и дальнейший поиск ведется тем же способом в верхней половине.  Если искомое значение меньше, чем значение серединного элемента, дальнейший поиск ведется в нижней половине набора данных.  Эти шаги повторяются, при каждой итерации отбрасывая половину элементов, пока не будет найдено искомое значение или пока оставшийся набор данных станет невозможно разделить напополам: """* **def** binary\_search(lst, number):  start = 0  end = len(lst) - 1   **while** start <= end:  mid = int((start + end) / 2)  **if** lst[mid] == number:  **return True  elif** lst[mid] < number:  start = mid + 1  **else**:  end = mid - 1  **return False   def** find\_number(lst, number):  flag = 0  **for** el **in** lst:  **if** binary\_search(el, number):  flag += 1  **if** flag == 0:  **return "Искомое число не найдено"  else**:  **return f"Искомое число встречается в {**flag**} коллекциях"** number\_lst = [[1, 3, 4, 2, 5, 7, 6, 8, 9], [1, 3, 4, 2, 5, 7, 6, 8, 9], [  1, 3, 4, 2, 5, 7, 6, 8, 9], [1, 3, 4, 2, 7, 6, 8, 9]] searched\_number = 5  print(find\_number(number\_lst, searched\_number)) |

**Листинг 10. task\_10.py**

|  |
| --- |
| *""" Сложность: O(n!) (основанием здесь можно пренебречь) Название: факториальное время  Яркий пример: Задача коммивояжера  Это факториальная временная сложность. Каждое действие требует вычисления всех перестановок в коллекции и итоговое время выполнения действия вычисляется как факториал исходной коллекции Это очень большое расчетное время и неоптимальный алгоритм  Рассмотрим пример, в котором выводятся все возможные комбинации элементов списка В основе алгоритма выбор на каждом этапе пары элементов,  чтобы выполнить каждую замену этих элементов только один раз  Итоговое время будет расти факториальным образом, в зависимости от размера входных данных, поэтому мы можем сказать, что алгоритм имеет факторную сложность времени O(n!). """* **def** func\_calc(lst, n):  **if** n == 1:  print(lst)  **return   for** i **in** range(n):  func\_calc(lst, n - 1)  **if** n % 2 == 0:  lst[i], lst[n - 1] = lst[n - 1], lst[i]  **else**:  lst[0], lst[n - 1] = lst[n - 1], lst[0]   lst = [1, 2, 3] func\_calc(lst, len(lst)) |

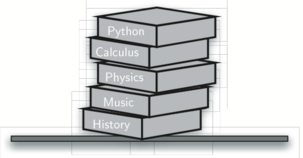
1. **Производительность основных структур данных в Python.**

Теперь мы понимаем смысл нотации О-большое. Теперь нужно понять о производительности операций над основными строительными блоками в Python – списками и словарями. Но этой интересной задачей мы займемся на 4-м уроке, когда освоим специализированные средства профилирования времени выполнения задачи.

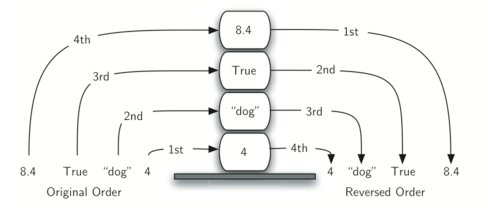
1. **Введение в абстрактные типы данных: стек, очередь, дек, список.**

**Стек**

Под стеком понимается упорядоченная коллекция элементов. Причем добавление нового или удаление существующего элемента всегда происходит только на одном из концов. В основании стека находятся элементы, находящиеся в стеке дольше всего. Элемент, добавленный последним, может быть удален в первую очередь (принцип LIFO - «последним пришёл — первым вышел»). Таким образом, более новые элементы находятся ближе к вершине стек, старые – к основанию. Примеры стеков из жизни – стопка книг, тарелок и т.д.



Фундаментальная важность стеков заключается в том, что их можно применять для изменения порядка элементов. Последовательность вставок обратная последовательности удалений.



Со стеками мы сталкиваемся ежедневно. Например, в каждом веб-браузере существует кнопка «Назад». Когда мы перемещаемся от одной страницы к другой, URL-ы этих страниц помещаются в стек. Таким образом, текущая страница, занимает вершину, а самая первая из просмотренных – занимает основание. При нажатии кнопки «Назад» мы будем перемещаться по страницам в обратном направлении.

В программировании мы также очень часто имеем дело со стеками или даже создаем их сами. В этом случае мы говорим о стеках не в физическом выражении, а в абстрактном их понимании. Соответственно в языке реализации должны быть возможности создать стек, добавить элемент на вершину, удалить верхний элемент со стека, возвратить верхний элемент, но не удалить его, проверить стек на пустоту, получить количество элементов в стеке.

Для реализации абстрактных сущностей идеально подходят классы. Реализуем стек в виде класса. Стековые операции будут воплощены в его методах. Для реализации самого стека, который по сути является коллекцией, воспользуемся мощью примитивных коллекций, реализованных в Python. Будем использовать список. Но нужно определиться, какой из его концов будет являться вершиной стека, а какой – базой. После принятия решения можно приступать к реализации, опираясь на всем знакомые методы списков, в частности, append и pop.

**Листинг 11. task\_11.py**

|  |
| --- |
| *"""Пример создания стека через ООП"""* **class** StackClass:  **def** \_\_init\_\_(self):  self.elems = []   **def** is\_empty(self):  **return** self.elems == []   **def** push\_in(self, el):  *"""Предполагаем, что верхний элемент стека находится в конце списка"""* self.elems.append(el)   **def** pop\_out(self):  **return** self.elems.pop()   **def** get\_val(self):  **return** self.elems[len(self.elems) - 1]   **def** stack\_size(self):  **return** len(self.elems)   SC\_OBJ = StackClass()  print(SC\_OBJ.is\_empty()) *# -> стек пустой  # наполняем стек* SC\_OBJ.push\_in(10) SC\_OBJ.push\_in(**'code'**) SC\_OBJ.push\_in(**False**) SC\_OBJ.push\_in(5.5)  *# получаем значение первого элемента с вершины стека, но не удаляем сам элемент из стека* print(SC\_OBJ.get\_val()) *# -> 5.5  # узнаем размер стека* print(SC\_OBJ.stack\_size()) *# -> 4* print(SC\_OBJ.is\_empty()) *# -> стек уже непустой  # кладем еще один элемент в стек* SC\_OBJ.push\_in(4.4)  *# убираем элемент с вершины стека и возвращаем его значение* print(SC\_OBJ.pop\_out()) *# -> 4.4  # снова убираем элемент с вершины стека и возвращаем его значение* print(SC\_OBJ.pop\_out()) *# -> 5.5  # вновь узнаем размер стека* print(SC\_OBJ.stack\_size()) *# -> 3* |

Стек также может быть реализован через список, где вершиной будет первый, а не последний элемент. В этом случае методы append и pop не сработают. Нам необходимо в этом случае применить методы insert (для первого элемента – позиции с индексом 0) и pop.

**Листинг 12. task\_12.py**

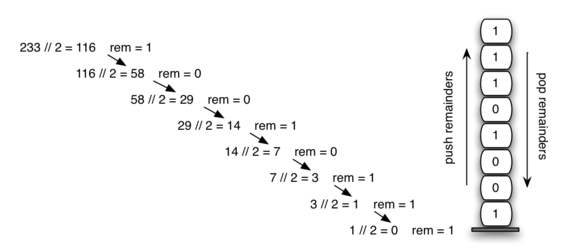
|  |
| --- |
| *"""Пример создания стека через ООП"""* **class** StackClass:  **def** \_\_init\_\_(self):  self.elems = []   **def** is\_empty(self):  **return** self.elems == []   **def** push\_in(self, el):  *"""Предполагаем, что верхний элемент стека находится в начале списка"""* self.elems.insert(0, el)   **def** pop\_out(self):  **return** self.elems.pop(0)   **def** get\_val(self):  **return** self.elems[0]   **def** stack\_size(self):  **return** len(self.elems)   SC\_OBJ = StackClass()  print(SC\_OBJ.is\_empty()) *# -> стек пустой  # наполняем стек* SC\_OBJ.push\_in(10) SC\_OBJ.push\_in(**'code'**) SC\_OBJ.push\_in(**False**) SC\_OBJ.push\_in(5.5)  *# получаем значение первого элемента с вершины стека, но не удаляем сам элемент из стека* print(SC\_OBJ.get\_val()) *# -> 5.5  # узнаем размер стека* print(SC\_OBJ.stack\_size()) *# -> 4* print(SC\_OBJ.is\_empty()) *# -> стек уже непустой  # кладем еще один элемент в стек* SC\_OBJ.push\_in(4.4)  *# убираем элемент с вершины стека и возвращаем его значение* print(SC\_OBJ.pop\_out()) *# -> 4.4  # снова убираем элемент с вершины стека и возвращаем его значение* print(SC\_OBJ.pop\_out()) *# -> 5.5  # вновь узнаем размер стека* print(SC\_OBJ.stack\_size()) *# -> 3* |

ПРИМЕНЕНИЕ СТЕКА:

1. Программный вид стека может применяться при реализации алгоритма Хаффмана (эффективная методика кодирования строк), алгоритмов обхода структур данных, деревьев, графов.
2. Аппаратный стек имеет место быть при реализации рекурсий.
3. Программный вид стека может применяться при реализации алгоритмов в различных предметных областях, там, где необходимы возможности стека.

Рассмотрим еще один пример использования стека, в котором мы реализуем конвертацию числа из десятичного формата в двоичный. Применим алгоритм «Разделяй на два», в котором стек позволяет запомнить цифры двоичного результата.

Мы простыми итерациями шаг за шагом разбиваем число в десятичном формате на два, далее получаем и записываем остаток. У четного числа будет ноль в остатке и соотв. цифра на первом месте. У нечетного числа в остатке наоборот будет единица и эта же цифра на первом месте.



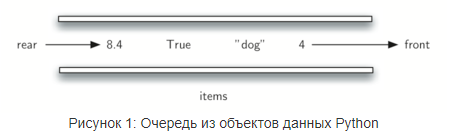
**Листинг 13. task\_13.py**

|  |
| --- |
| **from** task\_11 **import** StackClass   **def** divide\_by\_two(dec\_number):  sc\_obj = StackClass()   **while** dec\_number > 0:  res = dec\_number % 2  sc\_obj.push\_in(res)  dec\_number = dec\_number // 2   bin\_string = **""  while not** sc\_obj.is\_empty():  bin\_string = bin\_string + str(sc\_obj.pop\_out())   **return** bin\_string   print(divide\_by\_two(233)) |

**Очередь**

Под очередью понимается набор элементов, в котором добавление осуществляется с одного конца («хвоста»), а удаление существующих – с другого («головы»). При добавлении элемента в конец очереди, он перемещается к ее началу, ожидая удаления предыдущих.

Самые последние добавленные элементы «ждут» в конце коллекции. Элемент, пробывший в очереди дольше всех - находится в ее начале. **FIFO, first-in first-out** (англ. “первым пришёл - первым вышел”). Ещё он известен, как “первым пришёл - первым обслужен”.



В информатике тоже есть распространённые примеры очередей. В нашей компьютерной лаборатории стоит 30 компьютеров, подключённых по сети к одному принтеру. Когда студенты хотят что-то распечатать, они набирают задание “встать в очередь” вместе со всеми другими ожидающими печати заданиями. Первое задание - следующее, которое будет выполнено. Если вы последний в очереди, то должны ждать, пока напечатаются все стоящие перед вами документы. Позднее мы исследуем этот интересный пример более подробно.

В Python очередь создадим аналогично стеку – через использование ООП и списков. Будем использовать ф-цию insert для вставки элементов в конец очереди, а pop для удаления переднего элемента. Так будет вполне удобно.

**Листинг 14. task\_14.py**

|  |
| --- |
| **class** QueueClass:  **def** \_\_init\_\_(self):  self.elems = []   **def** is\_empty(self):  **return** self.elems == []   **def** to\_queue(self, item):  self.elems.insert(0, item)   **def** from\_queue(self):  **return** self.elems.pop()   **def** size(self):  **return** len(self.elems)   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  qc\_obj = QueueClass()  print(qc\_obj.is\_empty()) *# -> True. Очередь пустая   # помещаем объекты в очередь* qc\_obj.to\_queue(**'my\_obj'**)  qc\_obj.to\_queue(4)  qc\_obj.to\_queue(**True**)   print(qc\_obj.is\_empty()) *# -> False. Очередь пустая* print(qc\_obj.size()) *# -> 3* print(qc\_obj.from\_queue()) *# -> my\_obj* print(qc\_obj.size()) *# -> 2* |

Рассмотрим интересный пример использования очередей – алогритм игры «Горячая картошка». По условиям данной игры дети выстраиваются в круг и перекидывают предмет от соседа к соседу. В определенный момент игры весь процесс останавливается и ребенок, у которого в руках остался предмет (картошка), покидает игру. Игра продолжается, пока не останется единственный победитель.

В нашем случае программа получает на вход набор имен и параметр num, применяемый для подсчета. Программа будет возвращать имя последнего ребенка, оставшегося после повторяющегося отсчета num.

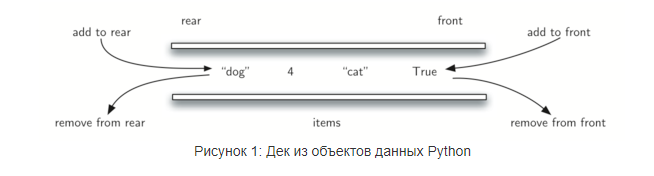
Для имитации круга детей используем очередь. Представим, что игрок, который держит картошку, занимает первую позицию в очереди. Он перебрасывает картошку, мы извлекаем его из очереди и ставим обратно, но уже в хвост очереди. Далее игрок будет ждать, пока все, кто перед ним, побудут первыми, а затем вернется на указанное место снова. После выполнения num операций извлечений/постановок участник, находящийся впереди, удаляется окончательно и цикл повторяется сначала. Процесс повторяется до тех пор, пока размер очереди не станет равным 1, т.е. пока не останется один ребенок в очереди.

**Листинг 15. task\_15.py**

|  |
| --- |
| **from** task\_14 **import** QueueClass   **def** hot\_potato(names\_lst, num):  queue\_obj = QueueClass()  **for** name **in** names\_lst:  queue\_obj.to\_queue(name)   **while** queue\_obj.size() > 1:  **for** i **in** range(num):  queue\_obj.to\_queue(queue\_obj.from\_queue())   queue\_obj.from\_queue()   **return** queue\_obj.from\_queue()   print(hot\_potato([**"Вася"**, **"Петя"**, **"Света"**, **"Жанна"**, **"Катя"**, **"Лена"**], 7)) |

**Дек**

Под деком (или двусторонней очередью) понимается упорядоченный набор элементов. Его отличительная особенность заключается в том, что новые элементы можно добавить, и в «голову», и в «хвост». Аналогично, существующие элементы могут быть удалены из обоих концов. Это «гибрид» стека и очереди. Хотя дек обладает характеристиками и очередей, и стеков, он не поддерживает LIFO или FIFO упорядочение.

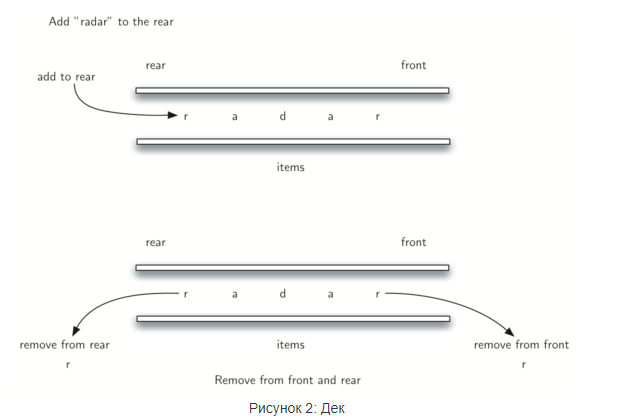


Реализуем дек на Python, как ранее.

**Листинг 16. task\_16.py**

|  |
| --- |
| **class** DequeClass:  **def** \_\_init\_\_(self):  self.elems = []   **def** is\_empty(self):  **return** self.elems == []   **def** add\_to\_front(self, elem):  self.elems.append(elem)   **def** add\_to\_rear(self, elem):  self.elems.insert(0, elem)   **def** remove\_from\_front(self):  **return** self.elems.pop()   **def** remove\_from\_rear(self):  **return** self.elems.pop(0)   **def** size(self):  **return** len(self.elems)   dc\_obj = DequeClass() print(dc\_obj.is\_empty()) *# -> True  # добавить элементы в хвост* dc\_obj.add\_to\_rear(10) dc\_obj.add\_to\_rear(**'my\_str'**)  *# добавить элементы в голову* dc\_obj.add\_to\_front(**None**) dc\_obj.add\_to\_front(**True**)  *# размер дека* print(dc\_obj.size()) *# -> 4* print(dc\_obj.is\_empty()) *# -> False  # добавить элемент в хвост* dc\_obj.add\_to\_rear(3.3)  print(dc\_obj.remove\_from\_rear()) *# -> 3.3* print(dc\_obj.remove\_from\_front()) *# -> True* |

Рассмотрим применение дека на примере решения задачи «палиндром». Это строка, которая одинаково читается с обеих сторон. Например, «молоко делили ледоколом». Мы обработаем строку слева направо и добавим каждый ее элемент по очереди в хвост дека.



Т.к. мы можем удалить сразу два элемента, то можно выполнить их сравнение и продолжить только если символы совпадают. При соответствии первого и последнего в процессе всего решения, то мы в конце придем или к отсутствию символов или останемся с деком размером 1. В обоих случаях исходный набор символов (строка) будет палиндромом.

**Листинг 17. task\_17.py**

|  |
| --- |
| **from** task\_16 **import** DequeClass   **def** pal\_checker(string):  dc\_obj = DequeClass()   **for** el **in** string:  dc\_obj.add\_to\_rear(el)   still\_equal = **True   while** dc\_obj.size() > 1 **and** still\_equal:  first = dc\_obj.remove\_from\_front()  last = dc\_obj.remove\_from\_rear()  **if** first != last:  still\_equal = **False   return** still\_equal   print(pal\_checker(**"топот"**)) |