

Урок 1. Введение в алгоритмы и структуры данных на Python

# ОГЛАВЛЕНИЕ

[ОГЛАВЛЕНИЕ 2](#_Toc88752869)

[ВВЕДЕНИЕ В АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ НА PYTHON 3](#_Toc88752870)

[Почему вам стоит пройти этот курс? 3](#_Toc88752871)

[Не забудьте о важности стиля 5](#_Toc88752872)

[Важно научиться анализировать эффективность алгоритма 5](#_Toc88752873)

[Начнем 6](#_Toc88752874)

[O(1) — постоянное время (константная сложность) 12](#_Toc88752875)

[O(log n) — логарифмическое время 12](#_Toc88752876)

[O(n) — линейное время 14](#_Toc88752877)

[O(n log n) — линейно-логарифмическое время 14](#_Toc88752878)

[O(n^2) — квадратичное время 15](#_Toc88752879)

[O(2^n) — экспоненциальное время 15](#_Toc88752880)

[O(n!) — факториальное время 16](#_Toc88752881)

[ФУНДАМЕНТАЛЬНЫЕ АЛГОРИТМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ 17](#_Toc88752882)

[Стек данных 17](#_Toc88752883)

[Применение стека 20](#_Toc88752884)

[Очередь 21](#_Toc88752885)

[Дек 23](#_Toc88752886)



# ВВЕДЕНИЕ В АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ НА PYTHON

## Почему вам стоит пройти этот курс?

Вы сможете:

**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, отличаются по объему кода. Т.е. при запуске эти различных вариантов решений у вас не возникают синтаксические или семантические ошибки (когда задача дает результат, который и должна давать, например, 2+2=4), но решения эти отличаются именно по своим качественным характеристикам.

При этом, однозначно сказать какое решение эффективнее нельзя, поскольку в одним условиях нам важна в первую очередь скорость работы скрипта, а в других объем выделяемой памяти, в-третьих – объем кода. Наша задача – научиться профилировать (замерять) время и память для разных вариантов решения задачи, чтобы потом выбрать наиболее подходящий, с учетом требований в ТЗ. При выборе решения конкретной задачи часто приходится идти на компромисс, что пытаться улучшить – время или память, учитывать ли стильность того или иного подхода и т.д.

**Развить алгоритмическое мышление, 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 будем практиковаться для отработки знаний, полученных по каждой теме.

**Научиться перекладывать сформулированный алгоритм на язык реализации.**

Любой алгоритм проходит несколько стадий, от его вербального 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.** Существуют 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 возможности 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 и приемы мы могли бы применить при решении каких реальных задач.

**ВАЖНО:** *часто слушатели курса спрашивают, зачем нам алгоритмы, зачем нам это курс, где мы сможем это применить на практике? Друзья, многим не хватает именно навыков «кодинга», когда есть задача и вроде все понятно, но садишься писать код и ничего не получается написать. Часто я получаю именно такие сообщения. Т.е. курс основ пройден, все основные конструкции языка слушатель знает, но как дело доходит до «написать какой-то код», тут же ступор. Потому что не хватает навыков реального программирования, т.е. не создания каких-то глобальных проектов – парсеров, веб-приложений, и т.д., а именно простых элементарных навыков кодинга. Значит наша задача – поработать над этими навыками, развить их. Это мы и будем делать на курсе. Плюс узнаем, как оценить решение задачи и выбирать лучшее. Это тоже немаловажно.*

## Не забудьте о важности стиля

При решении задач важна 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, повышает вероятность успешного прохождения собеседований. Наконец, правильный стиль кода упрощает его последующую модификацию и поддержание в рабочем состоянии.

**П.С.** студенты часто недовольны, что преподаватель излишне «придирается» к стилю и снижает за это оценки. Но хочу отметить, что делается это с одной благой целью – показать важность этого момента. Мы с вами уже не на основах Python и делать стилевые ошибки уже недопустимо.

## Важно научиться анализировать эффективность алгоритма

На тему анализа эффективности 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, уже есть заранее известный вариант функции «О-большое».

## Начнем

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 набор действий для решения задачи. Программа – это вариант того, 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.

Возьмем две простые функции, т.е. два алгоритма. Как определить какой из них эффективнее?

**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, что в одной из них есть лишние операции и код нужно немного оптимизировать. Здесь мы нашли решение быстро, мы обошлись без замеров, просто выполнили визуальный анализ кода. Но визуально оценивать эффективность кода мы можем только после большого опыта в программировании.

А что делать, если мы говорим о сложных, объемных 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 сравнивать алгоритмы?

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

Как же измерить время? Воспользуемся самым простым способом – модулем time. Таким образом, мы сможем получить абсолютные цифры.

**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* |

Если мы запустим код листинга 2 на различных машинах, мы получим разное итоговое время. Даже, если запустить код несколько раз на одной машине, цифры все равно будут разные.

Почему так?

Потому что цифры зависят от ресурсов конкретной машины (объема оперативной памяти), текущей загрузки процессора, разрядности ОС и даже версии интерпретатора.

Теперь понятно, что получать оценку времени в цифровом выражении не так уж эффективно. Хорошо было бы для оценки алгоритмов использовать характеристику, не привязанную к параметрам машины, параметрам ОС и интерпретатора. Но чтобы эта характеристика позволяла дать однозначный ответ, какой из алгоритмов более эффективный.

Самое время перейти к О-нотации (нотация О-большое). Это подход непростой в понимании, но очень важный. Если взять научное определение, то это математическое обозначение для сравнения асимптотического поведения функций.

Или более простыми словами эта нотация описывает как увеличивается потенциальное время выполнения алгоритма с ростом размера решаемой задачи.

Что такое размер задачи? По сути это количество операций, выполняемых в алгоритме. Определять время выполнения этих операций в абсолютных цифрах, как уже стало понятно, дело необъективное. Цифры будут разными.

Выполнить подсчет операций в алгоритме это уже более реально и объективно, т.к. количество операций не привязывается к параметрам вычислительной машины.

Т.е. мы можем взять несколько алгоритмов и определить, как в них будет меняться количество операций в зависимости от изменения входящего объема данных, т.е. от изменения размера решаемой задачи.

Конечно мы не будем подсчитывать количество операций в цифрах, потому что входящий объем данных будет разным. И значит количество операций будет разным.

Нам нужно именно оценить алгоритмы с помощью некоторых формул, где не было бы привязки к цифрам, но совершенно точно было показано, как меняется количество операций алгоритма (а значит и его время) с ростом входящего объема данных.

Вот такое непростое на первый взгляд описание. Теперь пример:

**Листинг 3. task\_3.py**

|  |
| --- |
| *"""Проверка сложности"""  # первые три присваивания - это константа (3) 3+300+20+1* VAL\_A = 5 VAL\_B = 6 VAL\_C = 10 *# три присваивания выполняются n^2 раз внутри вложенной итерации (3n^2)* **for** i **in** range(n):  **for** j **in** range(n):  x = i \* i  y = j \* j  z = i \* j  *# n=3 for I in range(n): print(i)  # 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.

Как мы это определили?

Сложили сложности отдельных выражений:

*T(n)=3+3n^2+2n+1=3n\*\*2+2n+4*

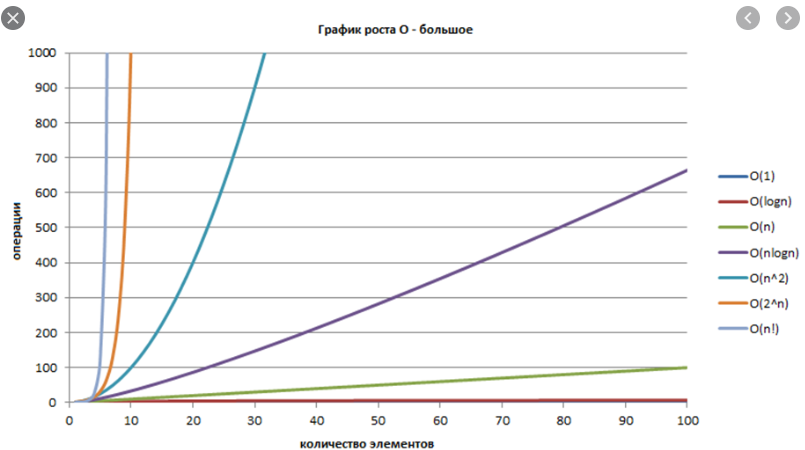
И выбрали то слагаемое выражения, которое оказывает наибольшее влияние на время выполнения всего скрипта (доминантную составляющую).

По сути под размером задачи здесь имеется в виду число операций алгоритма в зависимости от **n**. В других задачах зависимость может быть от размера массива или какого-либо другого объекта

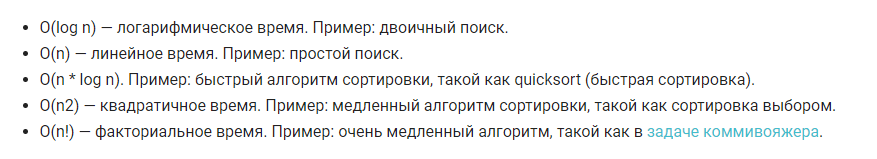
В таблице 1 приведено сравнение функций для нотации О-большое. Это и есть те самые обозначения, которыми мы должны описывать алгоритмы. Т.е. каждому алгоритму присуще какое-либо обозначение из этого перечня. Самое сложное конечно определить какое обозначение будет у нашего алгоритма.



На рис. 1 показаны графики функций из таблицы.



Основные варианты функций О-нотации и примеры задач под эти функции:

В листингах **task\_4-task\_9** представлен код примеров с алгоритмами различной сложности:

## O(1) — постоянное время (константная сложность)

Главная особенность – каким бы ни был размер передаваемых в алгоритм данных, время его работы будет одним и тем же. Например, мы можем передать массив, в котором 5 или 1000 элементов – это никак не повлияет на производительность алгоритма.

**Листинг 4. task\_4.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) *# O(1)* **return** length *# O(1)* print(get\_lst\_func(first\_lst)) print(get\_lst\_func(second\_lst)) |

Главная особенность – каким бы ни был размер передаваемых в алгоритм данных, время его работы будет одним и тем же. Например, мы можем передать массив, в котором 5 или 1000 элементов – это никак не повлияет на производительность алгоритма.

В данном случае, например, **len()** – это встроенная функция. Как мы узнали ее сложность?

Все стандартные операции и функции имеют заранее известную сложность. Где ее найти? Например, в таблицах, которые приложены в материалы урока.

## O(log n) — логарифмическое время

Главная особенность – время выполнения алгоритма определяется по формуле: логарифм от размера входных данных. Яркий пример – бинарный поиск, когда массив данных последовательно делится пополам до достижения результата – нахождения нужного элемента.

**Листинг 5. task\_5.py**

|  |
| --- |
| *""" Сложность: O(log n) (основанием здесь можно пренебречь) Название: логарифмическое время  Яркий пример - двоичный (бинарный) поиск  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)) |

## O(n) — линейное время

Главная особенность – время выполнения алгоритма определяется по формуле: логарифм от размера входных данных. Яркий пример – бинарный поиск, когда массив данных последовательно делится пополам до достижения результата – нахождения нужного элемента.

**Листинг 6. task\_6.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) |

## O(n log n) — линейно-логарифмическое время

Главная особенность – объединяет черты линейной и логарифмической сложностей. В данном алгоритме также фигурирует подход «Разделяй и властвуй», как при логарифмической сложности, но массив сперва разбивается на подмассивы до тех пор, пока в каждом не будет по два элемента. Каждая пара сортируется, а далее отсортированные пары объединяются и продолжается сортировка полученного результата. Ярки пример алгоритма с такой сложностью – сортировка слиянием. Ее мы подробнее рассмотрим на уроке 7.

## O(n^2) — квадратичное время

Главная особенность – время выполнения алгоритма зависит от квадрата размера входных данных.

**Листинг 7. task\_7.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\_1(lst):  **for** el\_lst **in** range(n):  **for** el **in** el\_lst:  **for** el **in** el\_lst:  print(el)   **def** get\_lst\_func\_2(lst):  **for** el\_lst **in** lst:  print(el)   get\_lst\_func\_1(obj\_lst) get\_lst\_func\_2(obj\_lst) |

## O(2^n) — экспоненциальное время

Главная особенность – удваивание вычислений при добавлении очередного нового элемента в массив. Такую сложность можно наблюдать в рекурсивных алгоритмах – в тех, когда один вызов функции порождает еще два вызова этой же функции. Яркий пример – вычисление чисел Фибоначчи.

**Листинг 8. task\_8.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* |

## O(n!) — факториальное время

Главная особенность – время выполнения алгоритма растет факториально от размера входных данных. Яркий пример – обычная рекурсия. Здесь даже при небольшом росте входных данных, время выполнения алгоритма растет очень существенно.

**Листинг 9. task\_9.py**

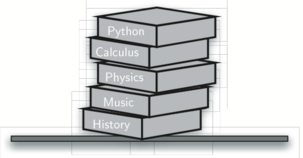
|  |
| --- |
| *""" Сложность: O(n!) (основанием здесь можно пренебречь) Название: факториальное время  Яркий пример: Задача коммивояжера  Это факториальная временная сложность. Каждое действие требует вычисления всех перестановок в коллекции и итоговое время выполнения действия вычисляется как факториал 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)) |

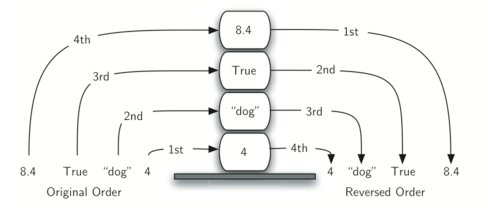
# ФУНДАМЕНТАЛЬНЫЕ АЛГОРИТМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Это именно фундаментальные структуры данных, некоторые сложные системы, а не встроенные функции или классы. Структуры данных – это не списки, кортежи, словари и т.д, но они могут создаваться на основе списков, словарей и т.д.

## Стек данных

Под стеком понимается упорядоченная структура элементов. Вставка и удаление элементов строится по принципу **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.





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 (рекурсия) или даже создаем их сами. Правда в случае с рекурсией мы говорим не о стеке данных. А о стеке вызовов.

Мы не создаем его сами, вручную, он уже заложен в библиотеку Python. Подробнее о стеке вызовов мы поговорим на уроке 2.

Сейчас же мы говорим о стеке данных, но это не встроенный класс. Как же тогда его создать, сымитировать? А как мы описываем собственные сущности? С помощью ООП.

Реализуем стек в виде класса. Стековые операции будут воплощены в его методах. Для реализации хранения элементов стека 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.

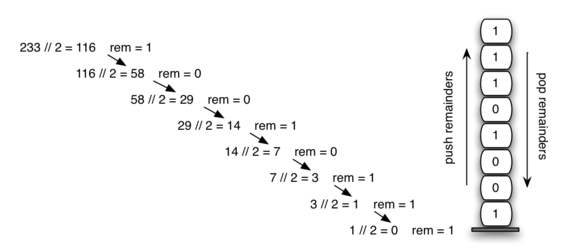
**Листинг 10. task\_10.py**

|  |
| --- |
| *"""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)   **if** \_\_name\_\_ == **'\_\_main\_\_'**:   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

Стек данных (именно о нем мы говорим в этом уроке) может применяться при реализации алгоритмов в различных предметных областях, там, где необходимы возможности стека, например, при реализации алгоритмов парсинга, обхода сложных структур, деревьев, графов и т.д.

Рассмотрим один учебный 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.

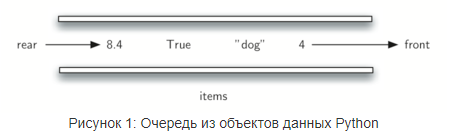
Мы простыми итерациями шаг за шагом разбиваем число в десятичном формате на два, далее получаем и записываем остаток в стек. У четного числа будет ноль в остатке и соответствующая цифра на первом месте. У нечетного числа в остатке наоборот будет единица и эта же цифра на первом месте.

**Листинг 11. task\_11.py**

|  |
| --- |
| **from** task\_10 **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 коллекция элементов, напоминающая конвейерную линию. Элементы добавляются в начало коллекции, а удаляются с конца. Это так называемый принцип **FIFO**, (first-in first-out).  Ещё он известен, как «первым пришёл - первым вышел».



В информатике тоже есть распространённые примеры очередей. В 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.

В Python очередь создадим аналогично стеку – через использование ООП и списков. Будем использовать функцию **insert()** для вставки элементов в очередь (в ее начало), а **pop()** для удаления переднего элемента. Так будет вполне удобно.

**Листинг 12. task\_12.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 – алгоритм игры «Горячая картошка». По условиям данной игры участники выстраиваются в круг и перекидывают предмет от соседа к соседу. В определенный момент игры весь процесс останавливается и участник, 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 участника, оставшегося после повторяющегося отсчета **num**.

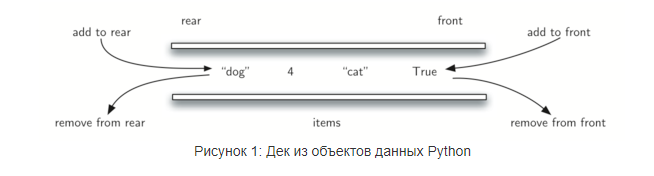
Для имитации круга участников используем очередь. Представим, что игрок, который держит картошку, занимает первую позицию в очереди. Он перебрасывает картошку, мы извлекаем его из очереди и ставим обратно, но уже в хвост очереди. Далее игрок будет ждать, пока все, кто перед ним, побудут первыми, а затем 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 участник в очереди.

**Листинг 13. task\_13.py**

|  |
| --- |
| **from** task\_12 **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.

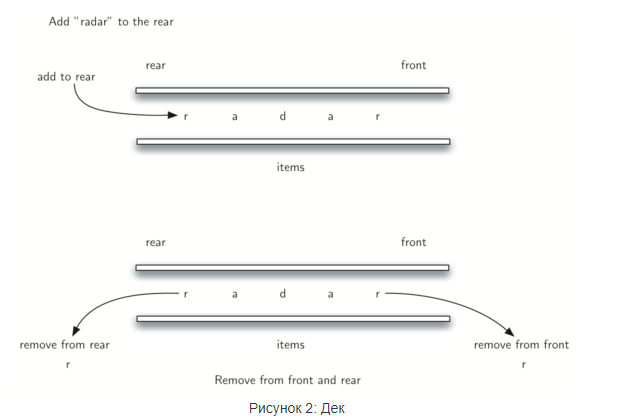


Реализуем дек на Python, как ранее, с использованием ООП.

**Листинг 14. task\_14.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)   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  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 15. task\_15.py**

|  |
| --- |
| *# палиндром* **from** task\_15 **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(**"молоко делили ледоколом"**)) |