**1. Многопроцессорные архитектуры с общей и разделяемой памятью – специфика и сравнение**

В архитектуре с общей памятью несколько процессоров синхронно используют общую память, причем все изменения одним процессором в памяти доступны всем другим процессорам. Это привычная модель программирования за счет единого адресного пространства. Архитектура имеет высокую скорость обмена данными между процессорами. С другой стороны, ее сложно масштабировать, так как использование большого количества процессоров сильно нагрузят шину. Также необходима синхронизация при доступе к общим данным. В архитектуре с распределённой памятью у каждого процессора своя память, недоступная другим процессорами. Обмен данными происходит через коммуникационную систему или сеть. Такая архитектура хорошо масштабируется, однако возрастает сложность программирования из-за необходимости в передаче сообщений и сложности в реализации некоторых структур данных и алгоритмов. Также понижается скорость обмена данными.

**2. Подходы к декомпозиции крупных вычислительных задач на подзадачи для параллельного исполнения**

Существует три подхода при декомпозиции крупных вычислительных задач на подзадачи для параллельного исполнения: декомпозиция на основе данных, на основе задач и на основе потока обработки. При декомпозиции на основе данных исходный набор данных разбивается на блоки, которые параллельно обрабатываются несколькими обработчиками. Частный случай - рекурсивный параллелизм, в котором данные рекурсивно разбиваются на блоки до определенно уровня, далее происходит параллельная обработка и ее результаты сливаются по мере разрешения рекурсии. При декомпозиции на основе задач набор функциональных задач разбивается на функциональные подзадачи, которые параллельно выполняются несколькими обработчиками. Частный случай - рекурсивный параллелизм, но только не для данных, а для задач. При декомпозиции на основе потока обработки чаще всего реализуется конвейерная обработка, в которой распараллеливаются последовательные стадии обработки данных.

**3. Модели параллельного программирования и их сочетаемость с архитектурами параллельных вычислительных систем**

В модели параллельного программирования на основе передачи сообщений каждая подзадача работает только со своей памятью, которая недоступна другим подзадачам. Общение между подзадачами происходит за счет обмена сообщениями. Пример реализации - процессы. Модель сочетается с архитектурой с распределенной памятью. Модель параллельного программирования с общей памятью подразумевает использование общего адресного пространства. Пример реализации - потоки. Модель параллельного программирования на основе параллельной обработки данных подразумевает реализацию глобальных операций над данными, обычно над массивами. Массивы делятся на фрагменты, которые обрабатываются параллельно. Модель сочетается как с архитектурой с распределенной памятью, так и с архитектурой с общей памятью. Последнее время все чаще на практике стали применять модель параллельного программирования на основе параллельной обработки данных. Например, Dask реализует эту модель.

**4. Профилирование реализации алгоритмов на Python, принципы решения задачи оптимизации производительности алгоритма**

Профилирование - сбор характеристик работы алгоритма, в частности времени выполнения отдельных фрагментов, объема используемой оперативной памяти и т. д. Для профилирования в IPython можно использовать следующие встроенные магические функции:

* %time - время выполнения
* %timeit - усредненное время выполнения по нескольким выполнениям, количество которых определяется автоматически
* %memit - объем используемой оперативной памяти

Основные принципы решения задачи оптимизации производительности алгоритма заключаются в том, чтобы оценить алгоритм с точки зрения времени и памяти, найти его узкие места и оптимизировать их. Для этого полезно оценивать работу алгоритма по строкам. Для этого можно использовать пакеты line\_profiler и memory\_profiler, которые предоставляют магические команды %lprun для оценки времени выполнения по строкам и %mprun для оценки памяти по строкам.

**5. Проблема Global Interpreter Lock в Python и способы обхода ее ограничений**

Данная проблема проявляется при построении многопоточных программ на Python и заключается в том, что, когда один из потоков программы выполняется, происходит глобальная блокировка интерпретатора, то есть все другие потоки блокируются. В действительности это просто означает, что в Python на разных ядрах процессора не могут работать одновременно несколько потоков одной программы, в отличие от других ЯП, где это возможно. GIL — это одна из причин, по которой решать CPU-bound задачи с помощью потоков неэффективно. Хорошим решением для обхода GIL может быть использование многопроцессности. Также GIL работает только в CPython, поэтому другим обходом может стать использование другого интерпретатора Python.

**6. Технологический стек Python для обработки и анализа данных, Python как glue language, специфика библиотеки NumPy и ее роль в экосистеме Python**

Базовыми инструментами для обработки и анализа данных на Python являются интерпретаторы CPython и IPython, среда Jupyter, NumPy, Dask, Numba. Более высокоуровневыми инструментами являются matplotlib, pandas, SciPy и другие. Еще более высокоуровневыми — инструменты для машинного обучения и нейронных сетей, например scikit-learn. Python считают glue language, то есть языком, который может напрямую взаимодействовать с готовыми решениями, являясь связующем звеном. Яркий пример — это NumPy, решение, которое написано на Си и Fortran. Мы можем взаимодействовать с ним через Python как через связующее звено. В основе NumPy — массивы, быстрые, эффективные, потребляющие мало памяти и являющиеся базой для быстрых расчетов и обработки данных. Более того, на NumPy основаны многие другие библиотеки технологического стека Python для обработки и анализа данных.

**7. Организация массивов в NumPy: хранение данных, создание массивов, принципы реализации операций с едиными исходными данными**

В действительности массивы ndarray в NumPy хранят в себе ссылку на одномерный массив элементов, которая дополняется различными свойствами. Массив типизированный, поэтому все элементы имеют одинаковый тип (ndarray.dtype). Затем, на основании этого буфера элементов и с помощью ndarray.shape (показывает количество измерений и количество элементов в каждом измерении) и ndarray.strides (показывает сколько байт нужно отступить, чтобы перейти к следующему столбцу или строке) задаются массивы произвольной размерности. При работе с ndarray многие функции реализованы по принципу операций с едиными исходными данными. Это значит, что чаще всего при преобразовании ndarray возвращается объект, ссылающийся на копию данных, а объект, который ссылается на прежние данные в памяти, но имеет другие атрибуты. Создание массива может быть на основании списка Python np.array([1, 2, 3]) или с помощью функций для генерации. Например

np.arrange(1, 4, 1) > array([1, 2, 3]) или

np.zeros((2, 2)) > array([[0, 0], [0, 0]])

и множество других функций генерации.

**8. Универсальные функции и применение функций по осям в NumPy**

Универсальные функции - функции, которые выполняют поэлементные операции над массивами. Причем такие функции векторизованы на базе простых операций, работающих со скалярами и возвращающих скаляр. Среди таких операций - операции сравнения, арифметические операции, различные математические функции и т. д. Например:

np.array([1, 2, 3]) + np.array([4, 5, 6]) > array([5, 7, 9])

Существуют также агрегирующие функции, у которых с помощью параметра axis можно задать ось, по которой нужно выполнить агрегирующую функцию. В случае, если axis=None, происходит обход всех элементов по порядку. Например:

a = np.array([[1, 2],

[3, 4]])

a.sum(axis=None) > 10 - сумма чисел по порядку 1, 2, 3 и 4.

a.sum(asix=0) > array([4, 6]) - сумма по столбцам (вдоль нулевой оси)

a.sum(asix=1) > array([3, 7]) - сумма по строкам (вдоль первой оси)

**9. Принцип распространения значений при выполнении операций в NumPy: общий алгоритм и примеры**

Распространение (broadcasting) описывает, как NumPy обрабатывает массивы с различными формами во время арифметических операций. Пример использования бродскастинга: совершение операций между вектором и скаляром. Для этого нужно копировать число столько раз, сколько у нас компонентов в векторе, получив вектор той же размерности, а потом умножить эти векторы. NumPy не заставляет нас вручную превращать скаляр в вектор, он сам добавляет размерность и клонирует содержимое нужное число раз, а потом уже производит поэлементное умножение. Бродкастинг работает и в более сложных ситуациях по следующим правилам: соответствующие измерения двух массивов должны либо совпадать, либо одно из них должно быть равно единице. Если в одном из массивов не хватает измерений, причем измерения имеют наименьшие номера, то им приписывается размерность 1.

Пример:

a = np.arange(6).reshape(3, 2)

b = np.arange(3).reshape(3,1)

a + b > array([[0, 1],

[3, 4],

[6, 7]])

**10. Маскирование и прихотливое индексирование в NumPy**

Маскирование - это использование масок для индексирования. Маски обычно создаются с помощью применения к массивам NumPy логических операций, результатом которых является массив такой же формы, но состоящий из логических значений True или False. Далее маска передается в качестве индекса. Тогда будет возвращен массив таких элементов, у которых в той же позиции в маске стоит True. Например, маска содержит True для четных элементов и False для нечетных, а при ее применении мы получаем все четные элементы массива:

a = np.array([1, 2, 3, 4])

a[a % 2 == 0] > array([2, 4])

Прихотливое индексирование - это использование в качестве индекса списка. В таком случае это будет интерпретировано как необходимость взять сразу несколько индексов. Например, можно взять все строки, и указать в качестве индекса по первой оси список индексов столбцов: сначала второй, а потом нулевой:

a = np.array([[1, 2, 3],

[4, 5, 6]])

a[:, [2, 0]] > array([[3, 1],

[6, 4]])

**11. Векторизация в numpy: ключевые параметры функции, примеры применения, использование обобщенной сигнатуры функции**

В NumPy векторизация применяется тогда, когда векторизованные универсальные функции не могут обеспечить решение задачи. Векторизованные функции создаются с помощью класса np.vectorize, конструктор которого принимает функцию, которая принимает один или несколько скаляров и возвращает один или несколько скаляров. Например:

def add(x, y):

return x + y

vadd = np.vectorizer(add)

vadd(np.array[1, 2, 3], np.array[3, 2, 1]) > array([4, 4, 4])

Обобщенная сигнатура функции определяет, как каждый из входных массивов будет разбиваться и становиться параметром векторизуемой функции и каким будет возвращаемый результат, собранный из того, что возвращает векторизуемая функция. Это позволяет принимать и возвращать не только скаляры, но и массивы произвольной размерности, а значит внутри векторизуемой функции использовать другие векторизованные функции. Сигнатура указывается в параметре signature конструктора np.vectorize. В сигнатуре используются списки размерностей - (i\_1, …, n). Например, запись «(i),(i)->()» означает, что векторизуемая функция принимает два вектора и возвращает скаляр. Таким образом, в такую векторизованную функцию можно передать два двумерных массива и получить, например, массив скалярных произведений соответствующих массивов.

**12. Numba: принципы работы, базовые примеры использования**

Numba - библиотека, позволяющая код, написанный на Python, на лету компилировать в машинный код, что полезно для ускорения. При этом при использовании Numba можно достичь скорости, сопоставимой с C. Numba полезен, когда код ориентирован на математические вычисления, использует NumPy и множество циклов. Numba предоставляет базовые декораторы @jit и @njit, которые нужно применить к функции, которую нужно ускорить. При использовании @jit будет использован режим object, а при использовании @njit - режим nopython, который в отличие от object mode не допускает использование операций Python, которые нельзя скомпилировать. Также в аргументы декоратора можно указать типы сигнатуры функции, чтобы использование скомпилированного кода происходило при первом же запуске, а не при последующих. Пример:

@njit(int32(int32, int32))

def add(x, y):

return x + y

**13. Организация Pandas DataFrame и организация индексации для DataFrame и Series**

DataFrame - аналог двухмерного массива с индексами строк и именами столбцов. DataFrame можно также воспринимать как таблицу. Каждая Series содержит данные одного столбца, а каждая его строка проиндексирована, причем индекс совпадает с единым для всего DataFrame индексом. Series поддерживают интерфейс словарей Python и механизмы индексации, аналогичные NumPy: срезы, маскирование, прихотливое индексирование. Когда мы используем в качестве индекса неявный индекс, то есть если индекс не является последовательностью чисел 0, 1, …, N, при индексировании могут возникать неожиданные результаты. Существуют loc и iloc, которые используются для явного и неявного индексирования соответственно. DataFrame можно рассматривать как упорядоченный словарь серий или как серию серий, поэтому обращение к DataFrame по ключу с названием столбца вернет соответствующую серию. Срезы и маскирование применяются к строкам. DataFrame поддерживает двумерные варианты loc и iloc.

Например:

df = pd.DataFrame({‘A’: [0, 1], ‘B’: [2, 3]})

df[‘A’] вернет серию-столбец с названием «A»

Например, создадим серию с неявным целочисленным индексом:

s = pd.Series([‘a’, ‘b’, ‘c’], index=[1, 3, 5])

Тогда лучше индексироваться через loc и iloc:

s.loc[3] > ‘b’ - использование явного индекса

s.iloc[3] > ‘с’ - использование неявного индекса

**14. Применение универсальных функций и работа с пустыми значениями в Pandas**

Универсальные функции NumPy - функции, которые выполняют поэлементные операции над массивами. Все универсальные функции библиотеки NumPy работают с объектами Series и DataFrame библиотеки Pandas. При выполнении бинарных операциях, итоговый объект будет содержать индексы первого и второго, а пропущенные значения будет обозначены как NaN. Например, если серия series состоит из чисел, можно сделать np.sin(series). Пустые значения также могут обозначаться как NA. Для работы с пустыми значениями предусмотрены следующие функции:

* DataFrame.isna() - возвращает датафрейм, заполненный True в тех позициях, где стоит пустое значением, и False в остальных позициях;
* DataFrame.dropna() - удаляет по умолчанию строки, содержащие пропущенные значения (можно параметром axis=1 указать удаление по столбцам);
* DataFrame.fillna() - заполняет значения пустые значения некоторым значением.

Аналогичные методы существуют для серий.

Например, если датафрейме df имеет столбец «age», в котором пропущены значения, их можно заполнить средний значением среднего возраста:

df[‘age’] = df[‘age’].fillna(df[‘age’].mean())

**15. Объединение данных из нескольких Pandas DataFrame: общая логика и примеры**

Функция pd.merge() по умолчанию выполняет поиск в двух входных объектах соответствующих названий столбцов и использует найденные столбцы в качестве ключа объединения. Если имена столбцов не совпадают, то имеются специальные параметры:

* on для явного указания одно или нескольких столбцов в обеих датафреймах;
* left\_on и right\_on для указания столбцов, если у первого и второго датафрейма они не совпадают;
* left\_index и right\_index для указания, что для объединения нужно использовать индекс.

Можно также указать тип соединения: внутреннее (inner) и внешнее (outer) и другие. Также существует аналогичный метод датафрейма DataFrame.merge(). Пример использования:

positions = pd.DataFrame({'employee': ['Bob', 'Jake'],

'group': ['Engineering', 'HR']})

hiring = pd.DataFrame({'employee': ['Bob', 'Jake'], 'hire\_date': [2004, 2008,]})

pd.merge(positions, hiring)

Функция pd.merge() будет объединять датафреймы по столбу employee, так как он имеется в обоих датафреймах.

Если необходимо объединить датафреймы не горизонтально, а вертикально, можно использовать универсальную функцию для объединения pd.concat(), или метод датафрейма DataFrame.append(). Например: pd.concat([df1, df2]).

**16. Операция GroupBy в Pandas DataFrame и реализация в ней подхода «разбиение, применение и объединение»**

Для группировки используется метод DataFrame.groupby(). Операцию GroupBy удобно представить в виде последовательного применения операций:

* split (шаг разбиения): включает разделение на части и группировку объекта DataFrame на основе значений заданного столбца или нескольких столбцов;
* apply (шаг применения): применение к группам какой-либо функции, обычно агрегирующей;
* combine (шаг объединения): выполнение слияния результатов предыдущих операций в датафрейм.

После вызова groupby() вернется объект, готовый к группировке, но в действительности группировка произойдет только на этапе применения. На этапе применения у объектов GroupBy кроме обычных агрегирующих методов, таких как sum(), median() и т. п., имеются методы aggregate(), filter(), transform() и apply(), позволяющие решать более универсальные задачи по обработке групп.

Пример получения среднего возраста по полу:

df.groupby(‘sex’)[[‘age’]].mean()

Пример нахождения доли мальчиков и девочек в разных школах:

df.groupby(‘school’)[[‘sex’]].apply(lambda x: x / x.count())

**17. Специфика текстовых и бинарных файлов, форматы файлов CSV и Pickle, представление данных в этих форматах и взаимодействие с ними в Python**

Текстовые файлы просто интерпретировать, читать и редактировать. Однако, текстовые файлы неэффективны как с точки зрения объема, так и с точки зрения скорости записи и чтения. В бинарных же файлах информация хранится в том же формате, что и в памяти, поэтому они более эффективные. С другой стороны, их нельзя просто прочитать или отредактировать. Формат CSV - это пример текстового файла, который предназначен для хранения табличных данных. Значения в строках файла разделяются запятой или другим разделителем. С помощью модуля csv можно с помощью csv.writer() писать в CSV-файл или с помощью csv.reader() читать из него. Pickle — пример бинарного файла для хранения объектов Python. С помощью модуля pickle можно с помощью pickle.dump() записать Python-объект в файл, а с помощью pickle.load() получить Python-объект из файла. Примеры использования модулей:

csv\_file = open(‘data.csv’)

reader = csv.reader(csv\_file, delimiter=’,’)

for line in reader:

print(line)

pickle\_file = open(‘data.csv’, ‘wb’)

pickle.dump([1, 2, 3], pickle\_file)

**18. Задача сериализации и десериализации, описание формата файла JSON и пример описания данных в этом формате и взаимодействия с ним в Python**

Задача сериализации - зафиксировать объект в определенном формате, например в формате JSON. Задача десериализации - по такому формату восстановить этот объект. Формат JSON (JavaScript Object Notation) основан на JavaScript и может содержать объекты (набор ключей и значений), массив, число (целое или дробное), true/false, null и строки. Для работы с JSON в Python используется модуль json и следующие его функции:

* json.load() десериализует JSON-строку из файла в объект
* json.dump() сериализует объект в JSON-строку в файл
* json.loads() десериализует JSON-строку в объект
* json.dumps() сериализует объект в JSON-строку

Правила сериализации/десериализации следующие:

* JSON-объекты – словари
* массивы – списки
* true/false – True/False
* null – None

Конечно, сериализовать можно только примитивы (словари, списки, строки, числа и т. д.). Например, нельзя сериализовать функцию.

**19. Формат XML и модель DOM: общая характеристика, пример описания данных в XML и DOM, работа с ними с помощью библиотеки BeautifulSoup**

Формат XML позволяет хранить данные в виде иерархического дерева элементов. Элементами являются теги, которые могут быть вложены друг в друга. У тега могут быть указаны атрибуты, чтобы описать его свойства. DOM — объектная модель документа, способ представления XML-подобных форматов, с помощью которого можно программно взаимодействовать с объектами DOM-дерева, как с объектами того ЯП, на котором мы работаем. В Python для этого используется популярная библиотека BeautifulSoup.

Фрагмент XML:

<persons>

<person male=”M”>Никита</person>

<person male=”M”>Петя</person>

<person male=”F”>Юля</person>

</persons>

Фрагмент Python-кода, который представит XML в виде объектов, найдет все теги person и выведет на экран их содержимое и атрибут male:

bs = BeautifulSoup(...) # Указываем весь XML в виде строки

for person in bs.find\_all('person'):

print(person.text, person[‘male’])

**20. Форматы файлов NPY и HDF общая характеристика, пример взаимодействие с данными этих форматов в Python**

Формат NPY - формат бинарного файла, который используется для хранения массива NumPy. Пример взаимодействия с файлом формата NPY:

a = np.array([1, 2, 3])

np.save(‘my\_array.npy’, a) - сохранение массива в файл

np.load(‘my\_array.npy’) - чтение массива из файла

Формат HDF - формат, который представляет собой контейнер для объектов объектов group, которые в свою очередь являются контейнерами других group и объектов dataset. Проще говоря, в файле формата HDF хранение данных организовано подобно файловой системе. В Python для работы с этим форматом можно использовать библиотеку h5py. Пример взаимодействия с файлом формата HDF:

file = h5py.File(‘datasets.hdf5’, ‘w’) # открываем файл

group = f.create\_group(‘group’) # создаем группу с названием «group»

group.create\_dataset(‘dataset’, data=np.array([1, 2, 3])) # создаем датасет в новой группе с названием «dataset»

file[‘/group/dataset’] # получаем доступ к данным датасета

**21. Взаимодействие с Excel из Python с помощью XLWings: принципы работы и примеры использования**

В первую очередь для работы c Excel с помощью XLWings необходимо открыть необходимую Excel-книгу с помощью класса Book. Book.sheets позволяет работать с листами книги. Способов обращения к ячейкам множество, но самый простой способ работы с ячейками - обращение к листу как к словарю c ключом в виде названия ячейки или диапазона. У диапазона ячеек есть несколько ключевых атрибутов: value (работа со значениями), formula (работа с Excel-формулой), options (позволяет сделать промежуточные преобразования диапазона) и другие. XLWings поддерживает NumPy и Pandas. Также XLWings позволяет писать макросы, интегрировать графики и многое, работать с форматом ячеек и многое другое. Пример записи значений, использования формулы и чтения в виде датафрейма:

wb = xw.Book(‘Книга1.xlsx’)

sht = wb.sheets[‘Лист1’]

sht[‘A1’].value = [[‘Значение1’, Значение2’], [1, 2]]

sht[‘C3’].formula = ‘=SUM(B1:B2)’

df = sht[‘A1’].options(pd.DataFrame, expand=’table’).value

**22. Основы работы с регулярными выражениями: базовый синтаксис, примеры использования модуля re в Python**

Для экранирования спецсимвола используется обратный слеш. В квадратных скобках можно перечислить символы, которые могут встречаться на этом месте. При перечислении можно указать диапазон символов через дефис. Например: «[абв]», «[а-я]». Символ ^ после первой скобки указывает, что все указанные в скобках символы должны отсутствовать на этой позиции. Можно использовать стандартные сокращения, например, «\d» (любая цифра). С помощью квантификаторов можно задать количество вхождений символа. Стандартных квантификаторы: «\*» - ноль и более, «+» - одно и более, «?» - ноль или одно. Модуль re предоставляет следующие функции:

* re.match() возвращает первое вхождение по шаблону с начала строки;
* re.search() возвращает первое вхождение по шаблону по всей строке;
* re.findall() возвращает список всех вхождений по шаблону;
* re.split() разбивает строку по шаблону;
* re.sub() заменяет в строке подстроку по шаблону на другую.

Например, re.findall(r’ [\w.-]+@[\w.-]+\.\w+’, s) возвращает список всех email, найденных в строке s.

**23. Сегментация и токенезация текста на естественном языке, стеммминг и лемматизация, примеры на Python**

Сегментация — разделение текста на сегменты, обычно предложения или токены. Сегментация в Python реализуется пакетом nltk. Чтобы разбить текст на предложения, используется функция nltk.tokenize.sent\_tokenize(). Токенизация — разбиение текста на токены, обычно на слова, знаки препинания и т. д. Чтобы разбить текст на токены, используется функция nltk.tokenize.word\_tokenize(). Стемминг — приведение слова к неизменяемой форме. Для этого используется nltk.stem.SnowballStemmer. Нужно создать объект с нужным языком: stemmer = SnowballStemmer(‘russian’) и для получения неизменяемой формы слова вызвать метод stemmer.stem(word). Лемматизация — приведение слова к словарной форме. Для лемматизации используется пакет pymorph2. Нужно сделать analyzer = pymorphy2.MorphAnalyzer() и затем analyzer.parse(word). Результатом будет список специальных объектов-разборов, предоставляющих всю информацию о найденной словарной форме. Они упорядочены по вероятности того, что данный разбор правильный.

**24. Расстояние Левеншнтейна: определение, алгоритм эффективного поиска** **оптимального редакционного предписания пример поиска на Python**

Расстояние Левенштейна - минимальное количество операций вставки символа, удаления символа и замены одного символа другим, необходимых для превращения одной строки в другую. Для поиска Расстояние Левенштейна в Python используется функция edit\_distance() из библиотеки nltk с параметрами по умолчанию. Например, edit\_distance(‘rain’, ‘shine’) вернет три (необходимо три действия: добавить букву «е», заменить «r» на «h» и добавить букву «s»). Для эффективного поиска оптимального редакционного предписания и расстояния Левенштейна используют алгоритм Вагнера-Фишера, который использует рекуррентную формулу и принцип динамического программирования снизу-вверх.

Изображение выглядит как текст

Автоматически созданное описание

означает расстояние между первыми символами первой строки и первыми символами второй строки. Базовые случаи: когда строки пустые, расстояние между ними равно нулю; когда одна из строк пустая, нужно просто сделать или вставок. В случае одинаковых символов на конце строк новые операции не требуются. Иначе выбрать минимальное из попытки удалить или добавить символ или сделать его замену, причем каждое действие увеличивает количество операция от предыдущего случая на единицу. Таким образом, начиная с можно дойти до решения.

**25. Векторное представление текста на естественном языке: общий алгоритм подходов TF; TF-IDF**

Тексты на естественном языке удобнее сравнивать друг с другом, если они представлены с помощью векторов. Для этого все термы (значимые элементы текста) всех текстов, входящих в корпус, обычно упорядочиваются в лексикографическом порядке. Если для некоторого текста для каждого терма выписать вес этого терма, то получится вектор, представляющий этот текст. Существует несколько алгоритмов определения этих весов. В TF вес определяется как частота вхождения терма в документ. В TF-IDF вес определяется как число, пропорциональное частоте вхождения в документе этого терма и обратно пропорциональное частоте вхождения этого терма во всех текстах корпуса. TF-IDF в отличие от TF хорош тем, что учитывает не только то, как часто слово входит в текст, но и то, насколько это слово специфичное в контексте других текстов.

**26. Модуль multiprocessing – назначение и основные возможности, API multiprocessing.Pool**

Модуль multiprocessing используется для того, чтобы реализовывать программы, работающие в несколько процессов. Основными классами, которые содержатся в multiprocessing, являются Process и Pool. С помощью Process можно выполнить функцию в отельном процессе. С помощью Pool можно организовать множественное выполнение одной и той же функции, но с разными аргументами с помощью пула процессов. Также в multiprocessing содержатся вспомогательные классы, которые позволяют организовать обмен данными между процессами, например Queue (очередь). API у класса Pool следующей. В конструктор Pool можно опционально передать количество процессов. Метод Pool.map() — аналог встроенной функции map. Pool.starmap() позволяет вызывать функцию, которая может принимать несколько аргументов. Существуют неблокирующие аналоги, например метод Pool.map\_async(), который возвращает объекты AsyncResult, и, таким образом, к возвращаемому значению можно обратиться позже. Pool.close() позволяет отменить все задачи в очереди и завершить все процессы. Pool.terminate() позволяет немедленно прервать все процессы.

**27. Различия между потоками и процессами, различие между различными планировщиками в Dask**

Поток - самостоятельная программа, которой операционная система выделяет необходимые ей системные ресурсы. Процессы имеют изолированное для других процессов адресное пространство. Поток - часть процесса. Внутри одного процесса работает один и более потоков. Потоки разделают общую память процесса. Процессы более тяжеловесные абстракции операционной системы. В Dask существует три основных планировщика: синхронный, многопоточный и многопроцессный. Синхронный выполняет код без распараллеливания и используется для отладки. В многопоточном используется пул потоков, а в многопроцессном - пул процессов. Многопроцессный подходит в том случае, если выполняется множество вычислений, а многопоточный подходит в том случае, если выполняется множество задач по вводу-выводу. Для вычислений многопоточность лучше не использовать во многом из-за GIL, однако если вычисления делаются на другом языке, например на Си как в случае использования NumPy и других инструментов на его основе, лучше использовать многопоточный планировщик и соответственно более легковесный потоки. Коллекции Dask имеют потоки по умолчанию. Например, для Dask.Array это многопоточный планировщик.

**28. Граф зависимостей задач – суть структуры данных, ее построение и использование в Dask**

Граф зависимостей задач - это ориентированный граф, вершинами которого являются вычислительные блоки, причем блоки связаны дугами в том случае, если работа одного вычислительного блока зависит от результатов работы другого. Данный граф строится для планировщика задач, который будет основываться на имеющихся связях в задачах и будет по возможности распараллеливать независимые задачи. Для стандартных коллекций Dask, например для Dask.Array, графы зависимостей задач строится автоматически. Граф зависимостей можно построить вручную. Пример:

def add(x, y):

return x + y

def mul(x, y):

return x \* y

operations = {

‘a’: 1,

‘b’: 2,

‘c’: (add, ‘a’, 1),

‘d’: (add, ‘b’, 2),

‘e’: (mul, ‘c’, ‘d’)

}

get(operations, ‘e’)

При такой указанной вручную последовательностью задач мы получим такой граф зависимостей задач:

Изображение выглядит как часы

Автоматически созданное описание