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

**Архитектура с общей памятью:**

Принципы многопроцессорной архитектуры с общей памятью (shared memory):

* несколько процессоров работают независимо, но совместно используют общую память
* изменения в памяти, осуществляемые одним процессором видны всем другим процессорам

Типы реализации архитектуры с общей памятью:

* **однородный доступ к памяти (Uniform Memory Access, UMA)** – равный (в т.ч. по времени) доступ всех процессоров ко всем областям основной памяти (процессоры могут иметь свой (когерентный) кэш). Типично для архитектуры с симметричной мультипроцессорностью (Symmetric Multiprocessing, SMP).
* **неоднородный доступ к памяти (Non-Uniform Memory Access, NUMA)** – время доступа к памяти определяется её расположением по отношению к процессору. Обычно реализуется как соединение нескольких SMP узлов.

Преимущества и недостатки:

* Привычная модель программирования за счет единого адресного пространства
* Высокая скорость и низкая латентность обмена данными между параллельными задачами
* Низкая масштабируемость (обычно до 16 процессоров) из-за геометрического роста нагрузки на шину CPU-RAM
* Проблема поддержания когерентности кэшей
* Трудоемкая организация эффективного использование памяти в NUMA-системах
* Необходимость синхронизации при доступе к общим данным (критические секции)

**Архитектура с распределенной памятью и гибридная архитектура:**

Принципы многопроцессорной архитектуры с распределенной памятью (distributed memory):

* несколько процессоров работают с собственной памятью, недоступной напрямую для других процессоров (отсутствует общая адресация памяти)
* обмен данными между процессорами производится через коммуникационную сеть и явно определяется исполняемой программой

Реализации архитектуры с распределенной памятью:

* возможно большое количество вариантов организации коммуникационной сети между узлами архитектуры с распределенной памятью
* на практике часто узлами систем с распределенной памятью являются многопроцессорные узлы с общей памятью (гибридная архитектура)

Преимущества и недостатки:

* Высокая масштабируемость
* Объем памяти растет пропорционально количеству ядер
* Возможность использовать недорогие массовые компоненты
* Специальные подходы к программированию: необходимость использования передачи сообщений (message passing)
* Сложность реализации некоторых структур данных и алгоритмов
* Высокая латентность и низкая скорость обмена данными между узлами
* Неоднородность, отказы узлов

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

С появлением параллельных систем возникли новые проблемы:

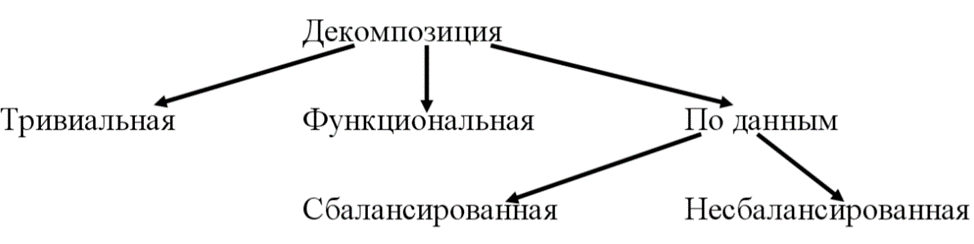
* как обеспечить эффективное решение задач на той или иной параллельной системе
* какими критериями эффективности следует пользоваться.

Проблемы описания классов:

* задач, которые естественно решать на данной параллельной системе,
* задач, не поддающихся эффективному распараллеливанию.

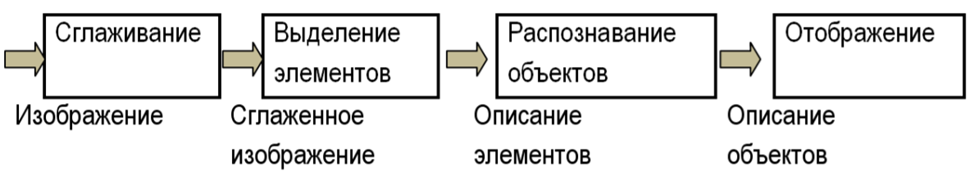
На этапе декомпозиции, общая задача вычислений и обработки данных делится на подзадачи меньшего размера. Игнорируются проблемы практической реализации, например число процессоров используемого в будущем компьютера. Все внимание сосредотачивается на возможном параллелизме исходной задачи.

Методы декомпозиции:



**Тривиальная декомпозиция** предполагает, что имеется последовательная программа, которая может исполняться независимо от большого числа различных исходных данных. Можно ввести параллелизм путем запуска некоторого числа этих программ параллельно. Так как нет зависимости между разными запусками программы, то число процессоров, которые могут быть использованы, ограничено только числом исполняемых запусков. В этом случае время исполнения набора запусков будет временем исполнения самого длинного по времени запуска в наборе.

**Функциональная декомпозиция.** Настоящий метод декомпозиции, разбивающий программу на подпрограммы. Простая форма функциональной декомпозиции называется "конвейер", которая очень схожа с конвейерной обработкой команд в процессоре. Пример с изображением:



Другой вид функциональной декомпозиции – это «фермы задач». Термин "farm" обозначает способа организации многомашинной системы, при котором одна из машин действует как планировщик, а другие – как рабочие. Отдельная задача в функциональной декомпозиции может быть распределена между некоторым числом процессоров. Эта технология обычно так и называется – ферма задач (task farming). Для обработки выбираются небольшие порции данных и один процессор, источник работы, или планировщик, который управляет набором необработанных порций данных. Некоторое число рабочих процессоров периодически запрашивают порцию данных из источника, обрабатывают ее и затем отправляют результаты к сборщику результатов, который может совпадать или нет с источником работы.

**Преимущество технологии фермы задач:**

Не требует предварительных предположений о наборе данных. Если порции данных достаточно малы, равномерная загрузка будет обеспечена за счет того, что рабочие процессоры, которые получают порции данных, не требующих большого объема работы над ними, будут просто запрашивать больше порций данных. Работа фермы задач требует постоянного потока запросов и ответов для частиц между рабочими машинами и управляющей. Стоимости, связанные с поддержанием динамического баланса загрузки, являются существенными в случае, когда выполняется много итераций над набором данных.

**Декомпозиция по данным:**

Многие задачи связаны с обработкой очень больших наборов данных, распределенных в регулярной сеточной структуре, при этом применяются некоторые операции трансформации над элементами данных. Когда данные могут быть разделены на регулярные подсетки и распределены между несколькими процессами, тогда преобразования могут быть применены параллельно, что позволяет решить задачу в меньшие сроки, чем это было бы возможно в обычных условиях. В методе декомпозиции регулярной области берется большая сетка элементов данных, разбивается на регулярные подсетки (блоки данных), и эти блоки распределяются по отдельным процессам, где они обрабатываются.

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

**Потоковая безопасность (thread safety)** – специфика кода (например функций или библиотек), позволяющая использовать его из нескольких потоков одновременно

* Источником нарушения потоковой безопасности может быть:
  + доступ к глобальным переменным или динамической памяти
  + выделение/освобождение глобальных ресурсов (например файлов)
  + неявный доступ через указатели
  + побочный эффект функций
* Эффективным подходом является изменение только локальных переменных потока.

GIL не является потокобезопасным.

**Проблема Global Interpreter Lock (GIL):**

**Global Interpreter Lock** – **способ синхронизации потоков** используемый в рефернсной реализации Python (CPython) и в реализациях некоторых других интерпретируемых языков программирования.

* Интерпретатор CPython **НЕ является потоково-безопасным** т.к. некоторые ключевые структуры данных могут быть одновременно доступны только одному потоку.
* GIL является самым **простым и быстрым при исполнении однопоточных приложений** способом обеспечения потоковой безопасности при одновременном обращении разных потоков к одним и тем же участкам памяти.
* Наличие **GIL не является требованием языка** программирования Python, а только спецификой реализации самого популярного интерпретатора CPython, существуют другие интерпретаторы Python не имеющие GIL.

Для **обхода проблемы GIL** для реализации параллельных вычислений в Python вместо многопоточного подхода с разделяемой памятью используется более тяжеловесная конструкция:

* + множество процессов, в каждом из которых работает собственный интерпретатор с собственным GIL и имеется собственная копия данных и кода
  + обмен данными между процессами обычно производится через передачу данных и кода с помощью сериализации
  + по сути, это вариация на тему модели параллельного программирования на основе передачи сообщений (реализуемой в т.ч. при вычислении на компьютере с разделяемой памятью)

Важно отметить, что контроль со стороны GIL распространяется только на объекты Python, т.е. для тех же операций чтения/записи (системных вызовов) можно использовать распараллеливание на уровне потоков, а не процессов. Основное отличие потоков от процессов - общая память.

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

**Разделяемая память** (shared memory):

* Аналогия - **доска объявлений**
* Подзадачи используют общее адресное пространство (оперативной памяти)
* Подзадачи **взаимодействуют, асинхронно** читая и записывая информацию в общем пространстве
* Реализация: многопоточные приложения, OpenMP

**Передача сообщений** (message passing):

* Аналогия – **отправка писем** с явным указанием отправителя и получателя
* Каждая подзадача работает с собственными локальными данными
* Подзадачи взаимодействуют за счет обмена сообщениями
* Реализация: MPI (message passing interface)

**Параллельная обработка данных** (data parallelization):

* Строго описанные глобальные операции над данными
* (Может обозначаться как чрезвычайная параллельность (embarrassingly parallel) – очень хорошо распараллеливаемые вычисления)
* Обычно данные равномерно разделяются по подзадачам
* Подзадачи выполняются как набор независимых операций
* Реализация может быть сделана как с помощью разделяемой памяти, так и с помощью передачи сообщений

***Модель параллельного программирования на основе передачи сообщений:***

Основные характеристики модели на основе передачи сообщений:

* Набор задач, имеющих свою собственную локальную память во время вычислений
* Задачи могут находиться как на одной машине (в т.ч. с разделяемой памятью), так и на разных машинах
* Задачи обмениваются данными с помощью отсылки и приема сообщений явно описываемых в программном коде
* Зачастую передача данных подразумевает их сериализацию/десериализацию, что требует соответствующих накладных расходов
* Как правило передача данных требует совместной работы, выполняемой как задачей-отправителем, так и задачей-получателем

Программирование для модели на основе передачи сообщений:

* С точки зрения программирования модель на основе передачи сообщений выглядит как внедрение вызовов специализированной библиотеки в программный код.
* За реализацию параллелизма полностью отвечает программист, а не компилятор
* Общепринятым стандартом для модели параллельного программирования на основе передачи сообщений является библиотека MPI (Message Passing Interface).
* Возможны реализации не двухточечной коммуникации с дополнительными структурами данных, например, очередями сообщений

***Модель параллельного программирования на основе параллельной обработки данных:***

Параллельная обработка данных (data parallelization):

* Строго описанные глобальные операции над данными (может обозначаться как чрезвычайная параллельность (embarrassingly parallel) – очень хорошо распараллеливаемые вычисления)
* Обычно данные равномерно разделяются по подзадачам
* Подзадачи выполняются как последовательность независимых операций
* Реализация может быть сделана как с помощью разделяемой памяти, так и с помощью передачи сообщений

Основные характеристики модели на основе параллельной обработки данных:

* Основные параллельные задачи сфокусированы на выполнении операций над неким массивом данных
* **Массив данных** обычно организован в виде **однородной структуры**, например массива или гиперкуба
* Задачи обычно параллельно выполняют аналогичные операции над выделенными им фрагментами одного массива данных
* В реализации на архитектурах без разделяемой памяти массив данных делится на фрагменты, которые находятся в распоряжении отдельных задач
* Программирование для данной модели обычно представляет собой написания программы, оперирующей с конструкциями для параллельной обработки данных, например в виде вызовов специализированной библиотеки

Наиболее распространенной моделью параллельного программирования является модель с общей памятью (другие названия: shared memory, параллелизм по управлению). В модели с общей памятью используются параллельные процессы, которые имеют доступ к одним и тем же переменным памяти.

Естественно, что модель с общей памятью стала интенсивно использоваться для параллельных вычислений. Многие параллельные алгоритмы изначально разрабатывались для реализации с общей памятью, а затем и для остальных моделей. Системы с небольшим числом процессоров представляют собой системы с общей памятью (SMPсистемы). Процессоры соединены с модулями памяти либо через общую шину, либо через коммутатор. Это позволяет обеспечить общее адресное пространство нескольким процессорам.

Одним из механизмов выполнения параллельных процессов стало многонитевое программирование «легковесных процессов», для которых не создаются полноценные процессы с отдельным адресным пространством, но которые все же могут распределяться по процессорам. Появился стандарт создания нитей POSIX Pthread, поддерживаемый всеми ведущими производителями. В настоящее время широко используется OpenMP – расширение языков С/С ++ и Фортран, представляющее директивы компилятору. OpenMP представляет концепцию нитей и часто ими же реализуется. Программа на языке OpenMP является последовательно-параллельной. Последовательные участки выполняются корневым процессом. Для выполнения параллельных участков образуются нити.

Другой моделью параллельного программирования является модель с распределённой памятью. Эта модель подразумевает наличие различных непересекающихся наборов данных для параллельных процессов. Взаимодействие и обмен данными происходит с помощью передач сообщений. Главным достоинством данной модели является отсутствие необходимости синхронизации по общим переменным. Это позволяет достаточно просто наращивать вычислительные мощности простым масштабированием систем. Главным недостатком данной модели является необходимость передач данных. Это усложняет программные модели, снижает производительность. Для уменьшения транспортных расходов на передачу данных алгоритмы для данной модели обычно конструируются таким образом, чтобы использовать крупноблочный параллелизм. Каждый процессор имеет свою локальную память. Обращение к данным другого процессора требует передачи данных через коммуникационную сеть. Системы с распределенной памятью можно условно разделить на системы с фиксированной и переменной топологией. Системы с фиксированной топологией, обычно строятся под определенные схемы обменов сообщениями программами и хорошо отражают связь между прикладными программами и аппаратурой.

При программировании таких систем чаще всего используется коммуникационные библиотеки PVM и MPI, обеспечивающие передачи сообщений между процессорами. MPI фактически стал международным стандартом программирования модели с передачей сообщений. Реализации MPI существуют практически для всех многопроцессорных вычислительных систем. MPI включает большой набор средств, ключевыми являются подпрограммы передачи данных от одного процесса к другому (другим). MPI реализован для языков С и Фортран и поддерживает все типы данных, имеющиеся в этих языках

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

**Профилирование** — сбор характеристик работы программы, таких как:

* время выполнения отдельных фрагментов (например, функций)
* число верно предсказанных условных переходов
* объем используемой оперативной памяти

Инструмент, используемый для анализа работы, называют профайлером (profiler). Обычно профилирование выполняется в процессе оптимизации программы.

**Магические функции IPython для профилирования в Jupyter Notebook:**

* %time - длительность выполнения отдельного оператора;
* %timeit - длительность выполнения отдельного оператора при неоднократном повторе (может использоваться для обеспечения большей точности оценки);
* %prun - выполнение кода с использованием профилировщика;
* %lprun - пошаговое выполнение кода с применением профилировщика;
* %memit - оценка использования оперативной памяти для отдельного оператора;
* %mprun - пошаговое выполнение кода с применением профилировщика памяти.

Команда %timeit выполняет оценку времени многократного выполнения фрагментов кода и автоматически подстраивает кол-во повторов выполнения под длительность работы функции.

Утилита **gprof2dot** генерирует картинку с деревом вызовов функций и информацией о времени их выполнения. В большинстве случаев этого достаточно для поиска узких мест в программе.

**pycallgraph** позволяет строить дерево вызовов программы Python.

**line\_profiler**, как следует из его названия, позволяет построчно отпрофилировать нужные участки. Для этого необходимо задекорировать необходимую функцию с помощью @profile, например:

@profile

**def** **is\_prime**(num):

pass

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

**Векторизация** позволяет записывать применение функции для преобразования множества значений (вектора) за одну операцию.

Векторизация позволяет:

* писать более компактный и выразительный код
* оптимизировать выполнение векторных операций по сравнению с применением циклов за счет специальных оптимизаций, в т.ч. за счет использования специальных возможностей процессоров, многие из которых поддерживают векторные операции на аппаратном уровне.

Метод **numpy.vectorize** принимает метод Python возвращает векторизованную версию функции. Векторизованная версия функции принимает последовательность объектов или массивов NumPy в качестве входных данных и вычисляет функцию Python над каждым элементом входной последовательности.

**Вид функции:**

**numpy.vectorize(pyfunc, otypes=None, doc=None, excluded=None, cache=False, signature=None)**

**Необходимые параметры:**

**pyfunc** : Функция, которую мы хотим применить к последовательности объектов

**Дополнительные параметры:**

**otypes** (str или список dtypes) : тип выходных данных.Он должен быть указан либо в виде строки символов шрифта, либо в виде списка спецификаторов типа данных. Для каждого вывода должен быть один спецификатор типа данных.

**Пример применения:**

def foo(a, b):

""" Если a > b возвращает a + b, Иначе возвращает a - b."""

if a >= b:

return a + b

else:

return a - b

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

b = 2

vecfoo = np.vectorize(foo, otypes=[float]) # указываем какой тип данных хотим получить

res = vecfoo(a, b)

print(type(res[0]))

OUT: <class 'numpy.float64'>

**doc** : Для указания строки документа созданного. Если не указано, будет использоваться исходная строка документа функции (если таковая имеется).

**excluded** : Набор строк или целых чисел, представляющих позиционные или ключевые аргументы, для которых функция не будет векторизована.

**Пример применения:**

import numpy as np

def g(x,p):

return p[0]+x\*p[1]+x\*x\*p[2]

print(g(5,[0,0,1]))

vg = np.vectorize(g, excluded=['p'])

print(vg(x=[0,1,2,3,4,5],p=[0,0,1])) # p не будет итерироваться

vg\_1 = np.vectorize(g)

print(vg\_1([0,1,2,3,4,5],[0,0,1])) # выпадет ошибка IndexError

OUT: 25

[ 0 1 4 9 16 25]

IndexError: invalid index to scalar variable.

**cache** : Если True , то кэшируйте первый вызов функции, определяющий количество выходов, если o-типов не предусмотрено.

**signature** : Обобщенная универсальная сигнатура функции, например, (m,n),(n)->(m) для векторизованного умножения матрицы на вектор. Позволяет векторизировать функции, которые действуют на нескалярных массивы фиксированной длины.

**Использование обобщенной сигнатуры функции:**

Имеется потребность проводить векторизацию не только скалярных функций, но и "векторных" функций.

* В результате векторизации векторные функции могут эффективно применяться для массивов больших размерностей.
* Для реализации этого механизма конструктору numpy.vectorize необходимо передать информацию о том какая векторная структура у входных параметров и выходных значений. Это делается с помощью передачи обобщенной сигнатуры функции через параметр signature.

**Обобщенная сигнатура функции** (generalized ufunc signature) определяет как размерности каждого из входных/выходных массивов разбиваются на размерности относящиеся к ядру (т.е. становятся параметрами единичного вызова векторизуемой функции pyfunc) и на размерности, использующиеся для векторизации.

Основные правила:

* каждое измерение в сигнатуре соотносится с измерениями соответствующих передаваемых массивов (соответствие строится начиная с конца кортежа, определяющего форму (shape) передаваемого массива).
* Измерения ядра, которым присвоены одинаковые имена, должны точно совпадать по размерам, в этом случае распространение (broadcasting) не производится.
* При применении векторизации измерения ядра убираются из всех входов, а для остающиеся измерений выполняется бродкастинг для выполнения итераций по ним в рамках работы векторизации.

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

Главный объект NumPy - это однородный многомерный массив (в numpy называется numpy.ndarray). NumPy обеспечивает эффективное хранение и лучшие способы обработки данных для математических операций с использованием простых API, что является преимуществом.

Создать массив numpy можно тремя способами:

* из списков или кортежей Python
* с помощью функций, которые предназначены для генерации массивов numpy (например: arange, linspace и т.д.)
* из данных, хранящихся в файле

В отличие от списков в Python массивы в NumPy строго "прямоугольные".т.е. количество элементов по каждой из размерностей во всех частях массива должно строго совпадать. Также все элементы массива должны быть строго одного типа.

**Примеры**

Создание ndarray на базе списка Python (не эффективный способ!):

*a = np.array([1,2,3,4,5,6,7,8])*

*A = np.array([[1, 2, 3], [4, 5, 6]])*

С помощью функции np.arange:

*import numpy as np*

*np\_array = np.arange(12)*

Построение обычного двухмерного массива ndarray:

b = np.array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14]])

Построение np.array из данных, хранящихся в файле

a = np.load('example\_1.npy')

**Чем отличаются массивы numpy от списков (и вложенных списков) Python**

Массивы numpy:

* **статически типизированы**: тип объектов массива определяется во время объявления массива и не может меняться
* **однородны**: все элементы массива имеют одинаковый тип
* **статичны**: размер массива неизменен, массивы должны быть "прямоугольными", но проекции массива по осям могут меняться

За счет этих свойств массивы numpy:

* **эффективно хранятся в памяти**
* операции над массивами numpy могут быть реализованы на компилируемых языках (C, Fortran). Это **на порядок повышает скорость выполнения операций**. Для массивов numpy в виде высокоэффективных функций реализованы основные математические операции.
* **не обладают гибкостью списков Python**
* прежде всего ориентированы на работу с числовой информацией (т.е. **имеют ограничения по типам используемой информации**)

Функция reshape не копирует массив, а создает новый заголовок, работающий с теми же данными. Идеология NumPy предполагает, что массив не копируется везде, где это явно не определено. Эта логика продиктована тем, что библиотека предназначена для обработки больших объемов данных. Неявное копирование этих данных при выполнении операций приведет к снижению производительности и возникновению проблем с доступной оперативной памятью.

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

**Базовый технологический стек Python** представляет из себя следующие компоненты:

1. Python - собственно, сам интерпретатор
2. SQL - язык запросов к SQL-подобным СУБД (Oracle/Postgres/MySQL)
3. SQLAlchemy - ORM для работы с SQL-подобными СУБД
4. Matplotlib - библиотека для визуализации и построения графиков
5. Pandas - библиотека для представления данных в табличном виде
6. numpy - библиотека для работы с матрицами
7. Seaborn - библиотека для создания статистических графиков на Python, основывается на Matplotlib
8. Dask - библиотека для параллельных вычислений при работе с большими данными на Python

**Python как glue language**

Glue language, связывающий язык - это язык, который используется в качестве промежуточного. Например, внутри NumPy работает код, написанный на C и Fortran, при этом ввод/вывод/препроцессинг данных осуществляется интерпретатором Python.

**Специфика библиотеки NumPy и ее роль в экосистеме Python**

Библиотека NumPy написана на языках C и Fortran. Это компилируемые языки, на которых вычисления производятся гораздо быстрее и эффективнее, чем на интерпретируемых языках. Используется в следующих сферах:

1. Научные вычисления. NumPy пользуются ученые для решения многомерных задач в математике и физике, биоинформатике, вычислительной химии и даже когнитивной психологии.
2. Создание новых массивных библиотек. На основе NumPy появляются новые типы массивов, возможности которых выходят за рамки того, что предлагает библиотека. Например, библиотеки Dask, CuPy или XND.
3. Data Science. В основе экосистемы для анализа данных лежит NumPy. Библиотека используется на всех этапах работы с данными: извлечение и преобразование, анализ, моделирование и оценка, репрезентация.
4. Machine Learning. Библиотеки для машинного обучения scikit-learn и SciPy тоже работают благодаря вычислительным мощностям NumPy.
5. Визуализация данных. По сравнению с Python возможности NumPy позволяют исследователям визуализировать наборы данных, которые гораздо больше по размеру. Например, библиотека лежит в основе системы PyViz, которая включает в себя десятки программ для визуализации.
6. Универсальные функции и применение функций по осям в NumPy

Универсальные функции NumPy на самом деле являются математическими функциями. Математические функции NumPy в NumPy сформулированы как универсальные функции. Эти универсальные работают с массивом NumPy и выполняют поэлементные операции со значениями данных. Универсальные функции NumPy принадлежат классу numpy.ufunc в Python. Некоторые из основных математических операций вызываются внутренне, когда мы вызываем определенные операторы. Например, когда мы кадрируем x + y, он внутренне вызывает универсальную функцию numpy.add ().

Мы даже можем создавать свои собственные универсальные функции, используя метод frompyfunc ().

Пример:

В этом примере мы преобразовали функцию **product** в универсальную функцию с помощью метода **frompyfunc ()**

**import numpy as np**

**def product(a, b):**

**return a\*b**

**product = np.frompyfunc(product, 2, 1)**

**res = product([1, 2, 3, 4], [1, 1, 1, 1])**

**print(res)**

**Вывод:**

**[1 2 3 4]**

Таким образом, теперь метод product() ведет себя как универсальная математическая функция и выполняет поэлементное умножение, когда массивы передаются ему в качестве параметров.

***Универсальные функции*** *(ufuncs)* - функции, выполняющие поэлементные операции над данными, хранящимися в массиве. Это векторные операции на базе простых функций, работающих с одним или несколькими скалярными значениями и возвращающими скаляр.

Основные универсальные функции:

* операции сравнения: <, <=, ==, !=, >=, >
* арифметические операции: +, -, \*, /, %, reciprocal, square
* экспоненциальные функции: exp, expm1, exp2, log, log10, log1p, log2, power, sqrt
* тригонометрические функции: sin, cos, tan, acsin, arccos, atctan
* гиперболические функции: sinh, cosh, tanh, acsinh, arccosh, atctanh
* побитовые операции: &, |, ~, ^, left\_shift, right\_shift
* логические операции: and, logical\_xor, not, or
* предикаты: isfinite, isinf, isnan, signbit
* другие функции: abs, ceil, floor, mod, modf, round, sinc, sign, trunc

**Основные функции, которым может передаваться ось:**

* Агрегирующие функциии: sum(), mean(), argmin(), argmax(), cumsum(), cumprod()
* Предикаты a.any(), a.all()
* Манипуляция векторными данными: argsort(), a.transpose(), trace(), reshape(...), ravel(), fill(...), clip(...)

1. Маскирование и прихотливое индексирование в NumPy

**Маскированные массивы** — это массивы, в которых могут быть отсутствующие или недопустимые записи. Модуль numpy.ma обеспечивает почти аналогичную замену numpy, которая поддерживает массивы данных с масками. Во многих случаях наборы данных могут быть неполными или испорченными из-за наличия неверных данных. Модуль numpy.ma предоставляет удобный способ решения этой проблемы путем введения маскированных массивов. Маскированный массив - это комбинация стандартного numpy.ndarray и маски.

Когда элемент маски имеет значение False, соответствующий элемент связанного массива действителен и считается немаскированным. Когда элемент маски имеет значение True, соответствующий элемент связанного массива считается замаскированным (недействительным).

Пакет гарантирует, что маскированные записи не будут использоваться в вычислениях. В качестве иллюстрации рассмотрим следующий набор данных:

import numpy as np

import numpy.ma as ma

x = np.array([1, 2, 3, -1, 5])  
Мы хотим отметить четвертую запись как недействительную. Самое простое — это создать массив по маске:

mx = ma.masked\_array(x, mask=[0, 0, 0, 1, 0])

Маска замаскированного массива доступна через его атрибут mask.  
Другая возможность - использовать функции getmask и getmaskarray . getmask(x) выводит маску x , если x является замаскированным массивом, и специальное значение nomask в противном случае. getmaskarray(x) выводит маску x , если x является замаскированным массивом. Если x не имеет недопустимой записи или не является замаскированным массивом, функция выводит логический массив False с таким количеством элементов, как x.

Для индексирования мы можем использовать **маски** (маскирование): если массив NumPy содержит элементы типа bool, то элемент выбирается в зависимости от булевского значения.

import numpy as np

arr = np.arange(10)

arr

Out: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

mask = arr%2 == 0

mask

Out: array([True, False, True, False, True, False, True, False, True, False])

arr[mask]

Out: array([0, 2, 4, 6, 8])

***Прихотливым индексированием*** *(fancy indexing)* называется использование массива или списка в качестве индекса. Это дает возможность быстрого доступа к различным подмножествам.

В случае "прихотливой" индексации форма результата отражает форму массивов индексов (index arrays), а не форму индексированного массива.

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

В качестве аргументов универсальных функций могут быть массивы с различной, но сравнимой формой. В этом случае применяется механизм **распространения (broadcasting)**. Правила выполнения распространения: соответствующие измерения двух массивов должны либо совпадать, либо одно из них должно быть равно единице.

Если в одном из массивов не хватает измерений, то считается, что недостающее количество измерений — это младшие измерения (измерения с наименьшими номерами), которым приписывается размерность 1.

**ПРИМЕРЫ:**

**1.** **В примере скаляр распространяется до массива размерности (5,):**

import numpy as np

np.arange(5)

Out: array([0, 1, 2, 3, 4])

np.arange(5) + 10

Out: array([10, 11, 12, 13, 14])

**2.** **Пример распространения для протяженных массивов разной размерности**

import numpy as np

a2 = np.arange(6).reshape(3, 2)

a2

Out: array([[0, 1], [2, 3], [4, 5]])

b2 = np.arange(10, 40, 10).reshape(3, 1)

b2

Out: array([[10], [20], [30]])

a2+b2

Out: array([[10, 11], [22, 23], [34, 35]])

1. Numba: принципы работы, базовые примеры использования

**Numba** - JIT компилятор с открытым исходным кодом, который компилирует подмножество кода Python и NumPy в быстрый машинный код.

**JIT-компиляция** - технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы.

**Преимущество:** достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом (сравнимая с компилируемыми языками)

**Недостаток:** увеличение потребления памяти (для хранения результатов компиляции) и дополнительные затраты времени на компиляцию на лету.

**Что еще может Numba:** SSE, AVX (в т.ч AVX-512) инструкции, использование CUDA ядер на GPU, эффективное распараллеливание, векторизация циклов

**Когда даёт хорошие результаты:**

* Код ориентирован на численные операции
* Код активно использует NumPy
* В коде много циклов (большое количество итераций)

**Как работает:**

1. Читает байт код Python для декорированной функции.
2. Собирает информацию о типах входных аргументов функций.
3. Анализирует и оптимизирует код.
4. Использует библиотеку для компиляции LLVM для генерации машинного кода функции для конкретного CPU.
5. Данный машинный код используется каждый раз при вызове данной функции (с аргументами того же типа).

**Пример использования с помощью декоратора @njit.**

Декоратор @njit имеет два режим работы:

1. Режим nopython. Устанавливается параметром nopython=True или использованием декоратора @njit. Это рекомендуемый для использования и наиболее быстрый режим. Приводит к компиляции кода функции, практически не использующего интерпретатор Python.
2. Режим object

# наивная реализация суммы квадратов элементов матрицы:

@njit

def sum\_sq\_2d\_jit(arr):

m, n = arr.shape

result = 0.0

for i in range(n):

for j in range(n):

result += ar[i,j] \*\* 2

return result

%%time

# во время первого запуска с данным типом параметров производства компиляции функции:

sum\_sq\_2d\_jit(arr)

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

Операцию GroupBy удобно представить в виде последовательного применения операций: разбиение, применение и объединение (**split, apply, combine**):

* **split** (шаг разбиения): включает разделение на части и группировку объекта DataFrame на основе значений заданного ключа.
* **apply** (шаг применения): включает вычисление какой-либо функции, обычно агрегирующей, преобразование или фильтрацию в пределах отдельных групп.
* **combine** (шаг объединения): во время шага выполняется слияние результатов предыдущих операций в выходной массив.

Для DataFrame операцию "разбить, применить, объединить" можно реализовать с помощью метода groupby(), передав в него имя желаемого ключевого столбца. Функция groupby() возвращает не набор объектов DataFrame, а объект DataFrameGroupBy, который можно рассматривать как специальное представление объекта DataFrame, готовое к группировке, но не выполняющее никаких фактических вычислений до этапа применения агрегирования (используется принцип отложенных вычислений).

**Пример:**

# подсчитываем количество не NaN значений в каждой группе:

planets.groupby('year').count()

# группировка экзопланет по методу их идентификации:  
planets.groupby('method').count()

Метод aggregate() может принимать на входе строку, функцию или список и вычислять все сводные показатели сразу.

**Пример:**

planets.groupby('method')['orbital\_period'].aggregate(['min', np.median, max])

**Пример** split (разбили один df на два с определенным условием):

df1, df2 = [x for \_, x in df.groupby(df['Sales'] < 30)]

In [1048]: df1

Out[1048]:

A Sales

2 7 30

3 6 40

4 1 50

0

In [1049]: df2

Out[1049]:

A Sales

0 3 10

1 4 20

**Метод apply()** позволяет применять произвольную функцию к результатам группировки. В качестве параметра эта функция должна получать объект DataFrame, а возвращать или объект библиотеки Pandas (например, DataFrame, Series), или скалярное значение, в зависимости от возвращаемого значения будет вызвана соответствующая операция объединения.

**Пример:**

def norm\_by\_min\_in\_year(x):

# x – объект DataFrame сгруппированных значений

x[‘orbital\_period\_normalized’] = x[‘orbital\_period’]/x[‘orbital\_period’].min()

return x

planets.groupby(‘year’).apply(norm\_by\_min\_in\_year) # planets – набор данных

1. Применение универсальных функций и работа с пустыми значениями в Pandas

**Работа с пустыми значениями:**

Отсутствующие данные объектов можно заменить на конкретные числовые значения, для этого можно использовать **метод *fillna()***.

Для того, чтобы удалить все объекты, которые содержат значения *NaN* воспользуйтесь методом ***dropna(****)* без аргументов.

Можно использовать df.isnull или df.notnull, которые покажут нам датафрейм с булевыми значениями, есть ли там данные или нет

**Применение универсальных функций:**

Универсальные функции NumPy на самом деле являются математическими функциями. Математические функции NumPy в NumPy сформулированы как универсальные функции. Эти универсальные (математические функции NumPy) работают с массивом NumPy и выполняют поэлементные операции со значениями данных.

Универсальные функции NumPy принадлежат классу numpy.ufunc в Python. Некоторые из основных математических операций вызываются внутренне, когда мы вызываем определенные операторы. Например, когда мы кадрируем x + y, он внутренне вызывает универсальную функцию numpy.add ().

Мы даже можем создавать свои собственные универсальные функции, используя метод frompyfunc ().

\*\*\*

numpy.frompyfunc(function-name, input, output)

\*\*\*

function-name : имя функции, которая будет оформлена как универсальная функция

input : Количество входных массивов

output : Количество выходных массивов

Пример:

В этом примере мы преобразовали функцию **product** в универсальную функцию с помощью метода **frompyfunc ()**

Таким образом, теперь метод product() ведет себя как универсальная математическая функция и выполняет поэлементное умножение, когда массивы передаются ему в качестве параметров.

**import numpy as np**

**def product(a, b):**

**return a\*b**

**product = np.frompyfunc(product, 2, 1)**

**res = product([1, 2, 3, 4], [1, 1, 1, 1])**

**print(res)**

**Вывод:**

**[1 2 3 4]**

*Универсальные тригонометрические функции в NumPy*

1. numpy. deg2rad() : Эта функция помогает нам преобразовать значение градуса в радианы.
2. функция numpy.sinh () : Вычисляет значение гиперболического синуса.
3. функция numpy.sin () : Вычисляет обратную гиперболическую величину синуса.
4. функция numpy.hypot () : Вычисляет гипотенузу для прямоугольной структуры треугольника.

*Универсальные статистические функции*

1. функция numpy.amin () : Представляет минимальное значение из массива.
2. функция numpy.amax () : Представляет максимальное значение из массива.
3. функция numpy.ptp () : Представляет диапазон значений массива по оси, который вычисляется путем вычитания минимального значения из максимального значения.
4. функция numpy.average () : вычисляет среднее значение элементов массива.
5. Объединение данных из нескольких Pandas DataFrame: общая логика и примеры

Данные из объектов pandas можно собрать несколькими путями:

1. Объединение pandas.merge()
2. Конкатенация pandas.concat()
3. Добавление данных в текущий DF - pandas.dataframe.append()
4. Комбинирование функция pandas.DataFrame.combine\_first()

**Объединение** — функция pandas.merge() соединяет строки в Dataframe на основе одного или нескольких ключей, может принимать несколько DataFrame подряд. Данная операция соответствует JOIN из SQL, состоит из объединения данных за счет соединения строк на основе одного или нескольких ключей.

Для использования merge библиотеке pandas нужны ключевые колонки, на основе которых будет проходить объединение. Иногда pandas не сможет распознать их автоматически, и тогда нужно указать названия колонок (left\_on и right\_on)

По умолчанию функция merge() выполняет inner join. Другие возможные варианты: left join, right join и outer join (параметр how)

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание**Конкатенация** — функция pandas.concat() конкатенирует объекты по оси.

По умолчанию функция concat() работает на axis=0 и возвращает объект Series. Если axis=1, то результатом будет объект Dataframe. Индекс результирующего дублируется; каждый индекс повторяется. Если результирующий объект должен следовать своей собственной индексации - ignore\_index=True

**Функция pandas.dataframe.append()** не изменяет DataFrame, но возвращает новый с добавленной строкой.

**Комбинирование** — функция pandas.DataFrame.combine\_first() является методом, который позволяет соединять пересекающиеся данные для заполнения недостающих значений в структуре, используя данные другой структуры.

Import pandas as pd

Изображение выглядит как текст

Автоматически созданное описаниеdf1 = pd.DataFrame({‘X’: [None, 0], ‘Y’: [None, 5]})

df2 = pd.DataFrame({‘X’: [3, 3], ‘Y’: [4, 4]})

df1.combine\_first(df2)

1. Организация Pandas DataFrame и организация индексации для DataFrame и Series

Dataframe — это табличная структура данных, напоминающая таблицы из Microsoft Excel. Ее главная задача — позволить использовать многомерные Series. Dataframe состоит из упорядоченной коллекции колонок, каждая из которых содержит значение разных типов (числовое, строковое, булевое и так далее).

Простейший способ создания Dataframe — передать объект dict в конструктор DataFrame(). Объект dict содержит ключ для каждой колонки, которую требуется определить, а также массив значений для них.

**Индексация в DataFrame и Series**

Индексирование в pandas означает простой выбор определенных строк и столбцов данных из DataFrame. Индексирование может означать выбор всех строк и некоторых столбцов, некоторых строк и всех столбцов или некоторых из всех строк и столбцов. Индексирование также может быть известно как выбор подмножества.

* **Фрейм данных. [];** Эта функция также известна как оператор индексации
* [**Dataframe.loc []**](http://espressocode.top/python-pandas-extracting-rows-using-loc/) **:** эта функция используется для меток.
* [**Dataframe.iloc []**](http://espressocode.top/python-extracting-rows-using-pandas-iloc/) **:** эта функция используется для позиций или целых чисел
* [**Dataframe.ix []**](http://espressocode.top/python-pandas-dataframe-ix/) **:** эта функция используется как для меток, так и для целых чисел.

Series — это объект библиотеки pandas, спроектированный для представления одномерных структур данных, похожих на массивы, но с дополнительными возможностями. Его структура проста, ведь он состоит из двух связанных между собой массивов. Основной содержит данные (данные любого типа NumPy), а в дополнительном, index, хранятся метки. Если не определить индекс при объявлении объекта, метки будут соответствовать индексам (положению в массиве) элементов объекта Series. Серии поддерживают интерфейс, близкий к словарям Python (извлечение элемента серии по аналогии с использованием словаря, поддерживается проверка вхождения элемента в индекс серии). Серии поддерживают механизмы индексации, аналогичные массивам NumPy: срезы, маскирование и прихотливое индексирование.

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

**Текстовые** - могут содержать только текстовые данные. Однако, в отличие от двоичных файлов, они с меньшей вероятностью будут повреждены. (пример CSV)

**Бинарные** - обычно содержат последовательность байтов или упорядоченные группы из восьми битов (пример Pickle)

*Форматы файлов:*

**CSV** (comma-separated value) - это формат представления табличных данных (например, данные из таблицы или БД), где каждая строка файла - это строка таблицы.

Пример файла в формате CSV:

id,name,city

102,Apple,California

155,Water,Moscow

999,Oil,London

Чтение формата CSV в Python:

import csv

with open(‘sw\_data.csv’) as f:

reader = csv.reader(f)

headers = next(reader)

print(‘Headers: ’, headers)

for row in reader:

print(row)

*csv.reader возвращает итератор, это часто используется для получения заголовков таблицы*

**Pickle** используется для сериализации и десериализации объектов Python, данный формат позволяет хранить сложные объекты Python.

Формат данных, используемый [модулем pickle](https://docs-python.ru/standart-library/modul-pickle-python/) зависит от Python. Это имеет то преимущество, что нет никаких ограничений, налагаемых внешними стандартами, такими как JSON или XDR, которые не могут представлять совместное использование указателей. Это означает, что программы, не являющиеся Python, могут не иметь возможности реконструировать выбранные объекты Python.

Запись данных в формат Pickle:

**Изображение выглядит как текст

Автоматически созданное описание**

Чтение данных формата Pickle:

import pickle

file = open(‘data.pkl’, ‘rb’)

data2 = pickle.load(file)

file.close()

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

**Сериализации** - это процесс конвертации состояния объекта в поток байтов. Данный поток байтов может быть дополнительно сохранен в любом файлоподобном объекте, таком как файл на диске или поток памяти. Он также может передаваться через сокеты и т.д.

**Десериализация** - это процесс восстановления объекта из потока байтов.

**JSON** - формат обмена данными, в котором вы можете передавать данные с клиента на сервер и с сервера на клиент, использует пары ключ-значение.

Пример файла JSON:

{

"size": "small",

"price": 12.6,

"recipe": ["apple", "water", "raspberry"],

"delivery": true,

"client": {

"name": "Jone",

"email": "jone@email.com"

}

}

Чтение данных формата JSON:

import json

with open(‘recipes.json’) as f:

recipes = json.load(f)

print(recipes)

Запись данных формата JSON:

**import json**

**to\_json ={**

**‘apple cake’: [“apple, “etc..”],**

**‘lemon cake’: [“lemon”, “etc..”],**

**‘chocolate cake’: [“chocolate”, “etc..”]**

**}**

**with open(‘recipes.json’, ‘w’) as f:**

**f.write(json.dumps(to\_json))**

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

**XML** — это сокращение от eXtensible Markup Language, переводится как «Расширяемый язык разметки». Смысл XML в том, чтобы выстроить внутри документа логическую структуру — чтобы было видно, что к чему относится и как всё связано между собой, в каком формате представлены данные.

**DOM** (от англ. Document Object Model — «объектная модель документа») — это независящий от платформы и языка программный интерфейс (API), позволяющий программам и скриптам получить доступ к содержимому HTML и XML-документов, а также изменять содержимое, структуру и оформление таких документов.

Пример XML файла:

*<games>*

*<game>*

*<title>Little Nightmares</title>*

*<author>Dave Mervik</author>*

*<price>9.99</price>*

*</game>*

*<game>*

*<title>Minecraft</title>*

*<author>Mojang Studios</author>*

*<price>29.99</price>*

*</game>*

*</games>*

Работа с XML при помощи **BeautifulSoup**:

from bs4 import BeautifulSoup

file = open(“games.xml”, “r”)

content = file.read()  
# загружаем содержимое xml файла в объект BeautifulSoup

soup = BeautifulSoup(content, ‘xml’)

# методом find\_all находим всех авторов

authors = soup.find\_all(‘author’)

for author in authors:

# методом get\_text получаем текст тега

print(author.get\_text())

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

**Формат файла NPY** - это файл данных, связанный с языком программирования Python. В основном используются для хранения объемных данных

**Основное использование:** NumPy - один из важных пакетов, необходимых для выполнения вычислений и анализа данных. Файлы NPY используются в NumPy для хранения информации о файлах массивов, необходимой для проведения требуемого анализа данных. Файлы NPY хранят данные массива в формате двоичного файла. Эти файлы NPY состоят из всей информации, необходимой для регенерации массива в его нынешнем виде, вместе с данными dtype и shape, на другом компьютере с различной архитектурой.

**Пример:**

import numpy as np

a1 = np.aaray([[1,2,3],[4,5,6]])

with open(‘a1.npy’, ‘wb’) as f:

np.save(f, a1)

with open(‘a1.npy’, ‘rb’) as f:

a1\_l = np.load(f)

**HDF5** позволяет эффективно хранить большие объемы данных. Формат HDF5 поддерживает файлы любого размера, и каждый файл имеет внутреннюю структуру, которая позволяет вам искать определенный набор данных. Это можно представить как отдельный файл со своей собственной иерархической структурой. По умолчанию данные хранятся в двоичном формате, и библиотека совместима с разными типами данных. Одним из наиболее важных вариантов формата HDF5 является то, что он позволяет прикреплять метаданные к каждому элементу структуры, что делает его идеальным для создания автономных файлов.

В Python интерфейс с форматом HDF5 можно построить с помощью пакета h5py. Одной из наиболее интересных особенностей этого пакета является то, что данные считываются из файла только тогда, когда это необходимо.

**Пример:**

import h5py

import numpy as np

arr = np.random.randn(1000)

with h5py.File(‘random.hdf5’, ‘w’) as f:

dset = f.create\_dataset(“default”, data=arr)

with h5py.File(‘random.hdf5’, ‘r’) as f:

new\_arr = f[‘default’]

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

**xlwings** - это библиотека с открытым исходным кодом для автоматизации Excel с помощью Python вместо VBA, которая работает на Windows и macOS. Поставляется c Anaconda, но можно установить в иных случаях c помощью пакетного менеджера pip или conda

**Работа с базовыми конструкциями:**

* Книги:
  + создаем новую книгу

import xlwings as xw

wb = xw.Book()

* подключаемся к существующему файлу  
  import xlwings as xw  
  wb = xw.Book('тест1.xlsx')
* Листы

Обращение к листу:

* + По индексу  
    wb.sheets[1]  
    xw.Range((2,3), (3,5)) *(индекс строки; индекс столбца)*
  + По имени  
    wb.sheets[‘Лист1’]
  + Как к активному листу  
    (функция active() позволяет изменить текущий активный объект)  
    wb.sheets.active
  + Добавление нового листа:

wb.sheets.add(name=”массивы”, after=wb.sheets[0].name)

* Range:
  + Чтение значения из области:

sht.range(‘A1’).value

* + Изменение значения в области  
    sht = wb.sheets['Лист1']  
    sht.range(‘A1’).value = ‘Значение 1’
  + Явно определенный диапазон:

sht.range(‘C2:D2’).value

* + Именованный диапазон:  
    sht.range(‘чпс’).value

**Options:**

Функция options() позволяет задать преобразование для диапазона. Преобразования определяют как диапазон ячеек Excel будет преобразован к значениям во время операций чтения и записи.

Сигнатура: options(convert=None, \*\*options)

· *expand(str, default None)* [выше]

· *transpose(Boolean, default False)* [для работы со столбцом вместо строки]

· *ndim(int, default None)* [количество измерений]

· *numbers (type, default None)*  
sht.range(‘A1’).options(numbers=int).value

· *dates(type, default None)*import datetime as dt  
sht.range(‘A1’).options(dates=st.date).value

· empty (преобразование пустых ячеек)  
sht.range(‘A1’).options(empty=’NA’).value

представление диапазона excel в виде массива numpy:

sht\_arr.range(‘A1’).options(np.array, expand=’table’).value

представление диапазона excel в виде Dataframe:

sht\_arr.range(‘A10’).options(pd.DataFrame, expand=’table’).value

**Манипулирование дополнительными атрибутами диапазонов (ячеек):**

* цветом диапазонов (ячеек)
  + добавление цвета (формат rgb)  
    xw.Range(‘A1’).color = (255, 0, 0)
  + удаление цвета  
    xw.Range(‘A1’).color = None
* размер и форматирование диапазонов (ячеек)
  + ширина столбца:

rng = xw.Range((2,3), (3,5))

# ширина столбца = количеством символов в заголовке

rng.columns[0].column\_width = len(rng.value[0][0])

# автоматическое приведение ширины столбца:

rng.columns[1].autofit()

* объединение

xw.Range(‘C1:D1’).api.merge()

* добавление и удаление диапазонов (ячеек)
  + копи-паст  
    sht.range((1,1),(5,2)).copy()  
    sht.range((6,1),(10,2)).paste()
  + прямое копирование  
    sht.range((1,1),(5,2)).copy(destination= sht.range((6,1),(10,2)))
  + удаление  
    sht.range((2,1)).delete() # удаление ячейки  
    sht.range(‘2:2’).api.Delete() # удаление столбца

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

Регулярные выражения предназначены для выполнения сложного поиска или замены в строке. В языке Python использовать регулярные выражения позволяет модуль re. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

import re

Создать откомпилированный шаблон регулярного выражения позволяет функция compile(). **Функция имеет следующий формат:**

<Шаблон> = rе.соmрilе(<Регулярное выражение>[, <Модификатор>])

**Пример регулярного выражения**

import re

# шаблон, соответствующий строке, начинающейся на 0-6 русских букв

p = re.compile(r”[а-яё]{0,6}”, re.I)

r = p.match(‘тикtik’).group(0)

print(r)

#получим ‘тик’

**Составление регулярных выражений**

Внутри регулярного выражения (внутри кавычек, его определяющих) символы

. ^ $ \* + ? { } [ ] \ | ( ) -

имеют специальное значение. Если эти символы требуется выводить как есть, то их следует экранировать с помощью слэша. Некоторые специальные символы теряют свое особое значение, если их разместить внутри квадратных скобок. В этом случае экранировать их не нужно.

В квадратных скобках [] можно указать символы, которые могут встречаться на этом месте в строке. **Можно перечислять символы подряд или указать их диапазон через тире. Примеры:**

* [09] - соответствует цифре 0 или 9
* [0-9] - соответствует одной цифре от 0 до 9
* [абв] - соответствует букве "а", "б" или "в"
* [а-г] - соответствует букве "а", "б", "в" или "г"
* [а-я] - соответствует любой букве от "а" до "я", кроме буквы "ё"
* [а-яё] - соответствует любой букве от "а" до "я"
* [АБВ] - соответствует букве "А", "Б" или "В"
* [А-ЯЁ] - соответствует любой букве от "А" до "Я"
* [а-яА-ЯёЁ] - соответствует любой русской букве в любом регистре
* [0-9а-яА-ЯёЁа-zА-Z] - любая цифра и любая буква независимо от регистра и языка

**Примеры**

import re

# шаблон корректной даты

p = re.compile(r’ [0-3] [0-9] . [01] [0-9] . [12] [0-9] [0-9] ‘)

date\_ = “20.01.2023”

res = p.match(date\_)

print(res)

Вместо перечисления символов можно использовать **стандартные классы**:

. - любой символ, кроме перевода строки (если точка не экранирована и не заключена в квадратные скобки)  
\d - соответствует любой цифре (эквивалентно [0-9])   
\w - соответствует любой букве, цифре или символу подчеркивания ([a-zA-Zа-яЁА-ЯЁ0-9\_])  
\s - любой пробельный символ (пробел, перевод строки, табуляция и т.д.)  
\D - не цифра (эквивалентно [^0-9])   
\W - не буква, не цифра и не символ подчеркивания (эквивалентно [^a-zA-Zа-яЁА-ЯЁ0-9\_])  
\S - не пробельный символ  
\b - обозначение левой или правой границы слова (где слово трактуется как последовательность букв или цифр)

**ПРИМЕР**

import re

p = re.compile(r’ [\d] [\D] [\s] [\S] ’)

**Квантификаторы**

С помощью квантификаторов задается количество вхождений символа в строку. Указывается после символа, к которому относится разрешенное количество повторений: <br>

{n} - n вхождений символа в строку. Например. шаблон `r"[0-9]{2}"` соответствует двум вхождениям любой цифры  
{n,} - n или более вхождений символа в строку. Например. шаблон `r"[0-9]{2,}"` соответствует двум и более вхождениям любой цифры  
{n,m} - не менее n и не более m вхождений символа в строку. Числа указываются через запятую без пробела. Например, шаблон `r"[0-9]{2,4}"` соответствует от двух до четырех вхождениям любой цифры

\* - ноль или большее число вхождений символа в строку. Эквивалентно комбинации {0,}  
+ - одно или большее число вхождений символа в строку. Эквивалентно комбинации {1,}  
? - ни одного или одно вхождение символа в строку. Эквивалентно комбинации {0,1}.

Если необходимо получить только содержимое тегов, то нужный фрагмент шаблона следует разместить внутри круглых скобок.

Круглые скобки также часто используются для группировки фрагментов внутри шаблона. По умолчанию все фрагменты в скобках выводятся в результат.

**Часто используемые методы:**

**re.match()** - Этот метод ищет по заданному шаблону в начале строки. Возвращает первое вхождение подстроки в виде объекта SRE\_Match object, из которого:

можно получить результирующую подстроку с помощью функции group

индексы начальной и конечной позиции с помощью функций start() и end(), соответственно.

**re.search()** - Этот метод ищет по заданному шаблону во всей строке. Возвращает первое вхождение подстроки в виде объекта SRE\_Match object, из которого:

можно получить результирующую подстроку с помощью функции group

индексы начальной и конечной позиции с помощью функций start() и end(), соответственно.

**re.findall()** - Этот метод возвращает список всех найденных совпадений (подстрок).  
**re.split()** - Этот метод разделяет строку по заданному шаблону. Первый аргумент функции - регулярное выражение, обозначающее разделитель, второй аргумент - исходная строка.

**re.sub()** - Этот метод ищет шаблон в строке и заменяет его на указанную подстроку.  
**re.compile()** - собирает регулярное выражение в отдельный объект, который может быть использован для поиска. Это также избавляет от переписывания одного и того же выражения.

1. Модуль multiprocessing – назначение и основные возможности, API multiprocessing.Pool

Multiprocessing – модуль для реализации параллельных вычислений на основе процессов. Позволяет избегать ограничения GIL для параллельных вычислений. Позволяет избегать использования примитивов для синхронизации (модель передачи сообщений). Включает абстракции с интерфейсом, похожим на threading.Thread

**Класс Pool**

Модуль multiprocessing также вводит API, которые не имеют аналогов в модуле threading. Ярким примером этого является объект Pool, который предлагает удобные средства распараллеливания выполнения функции по нескольким входным значениям, распределения входных данных между процессами (параллелизм данных).

Класс Pool() модуля multiprocessing создает объект, управляющий пулом рабочих процессов, в который могут быть отправлены задания. Пул рабочих процессов поддерживает асинхронное выполнение задач с тайм-аутами и обратными вызовами и имеет параллельную реализацию.

**Синтаксис:**

from multiprocessing import Pool

pool = Pool([processes[, initializer

[, initargs[, maxtasksperchild

[, context]]]]])

Параметры:

* processes - количество используемых рабочих процессов,
* initializer - вызываемый объект (функция),
* initargs - аргументами для initializer
* maxtasksperchild - количество задач рабочего процесса до обновления,
* context - контекста для запуска рабочих процессов.

Методы:

* Pool.apply
* Pool.map
* Pool.apply\_async
* Pool.map\_async

Методы Pool.apply и Pool.map эквивалентны встроенным методам apply и map.

- apply(), apply\_async() возвращают результат в произвольном порядке

- map() в порядке следования данных в исходной коллекции

Pool.map и Pool.apply блокируют основную программу до тех пор, пока все процессы не будут завершены. Используют, чтобы получить результаты в определенном порядке для определенных приложений.

Напротив, варианты async стартуют все процессы сразу и получат результаты, как только они будут готовы. Еще одно отличие - нужно использовать метод get после вызова apply\_async(), чтобы получить возвращаемые значения завершенных процессов.

pool = mp.Pool(processes=4)

results = pool.map(cube\_.cube, range(1,7))

print(type(results[0]))

print(results)

Out: <class ‘int’>

Out: [1, 8, 27, 64, 125, 216]

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

**Токенизация** - это разделение текста на слова и знаки пунктуации, то есть токены. Как и в случае сегментации предложения, знаки препинания могут быть сложными.Например, Соединенное Королевство следует рассматривать как один токен, а «не делай» следует разбивать на два токена: «делай» и «не делай».

Стемминг и лемматизация являются важными частями процесса нормализации. Во время процесса определения основы слово определяется путем удаления суффиксов, таких как -ed и -ing. Результирующий стебель не обязательно слово. Аналогично, лемматизация включает в себя удаление префиксов и суффиксов с тем важным отличием, что результат принадлежит языку. Этот результат называется леммой.

Оба метода уменьшают шум в тексте, преобразовывая слова в их базовую форму. Для большинства приложений, таких как классификация текста или кластеризация документов, где важно сохранить значение слова, лучше использовать лемматизацию, а не основание. Например, встреча (*имя существительное*) и встреча (*глагол*) будет встречаться, так что теряет свое первоначальное значение, в то время как соответствующие леммы будут встречаться и встречаться.

Другие методы нормализации включают: расширение аббревиатур, удаление цифр и знаков препинания, исправление типичных грамматических ошибок и т. д. Большинство из этих операций могут быть выполнены с помощью регулярных выражений.

**Пример токенизации:**

from nltk.tokenize import word\_tokenize

tokens = nltk.word\_tokenize(cleaned\_review)  
print(cleaned\_review)

print(tokens)

**Пример стеммизации:**

from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

stemmed\_review = [stemmer.stem(word) for word in filtered\_review]

print(stemmed\_review)

**Пример лемматизации:**

from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

lemm\_review = [lemmatizer.lemmatize(word) for word in filtered\_review]

print(lemm\_review)

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

**Расстояние Левенштейна** (редакционное расстояние, дистанция редактирования) - минимальное количество операций необходимых для превращения одной строки в другую. Рассматриваются следующие операции:

* вставка одного символа
* удаление одного символа
* замена одного символа на другим.

В общем случае стоимость различных операций может быть различной. Обычно цена отражает разную вероятность событий и может зависеть от вида операции (вставка, удаление, замена) и/или от участвующих в ней символов.

Если к списку разрешённых операций добавить транспозицию (два соседних символа меняются местами), получается расстояние Дамерау - Левенштейна.

* Дамерау показал, что 80 % ошибок при наборе текста человеком являются транспозициями.
* Кроме того, это расстояние используется и в биоинформатике.

**Алгоритм эффективного поиска оптимального редакционного предписания:**

Редакционным предписанием называется последовательность действий, необходимых для получения второй строки из первой кратчайшим образом. Обычно действия обозначаются так: D (англ. delete) — удалить, I (англ. insert) — вставить, R (replace) — заменить, M (match) — совпадение.

По сути, редакционное предписание это кратчайшие пути на графе с весами, в котором существует 3 вида ориентированных ребер (D, I, M), а вершинами являются строки (слова). В общем случае для конкретной пары слов может существовать несколько редакционных предписаний (кратчайших путей на графе).

**Пример поиска (расстояния Левенштейна) на Python:**

from nltk.metrics.distance import edit\_distance

r = edit\_distance(‘intention’, ‘execution’)

**Недостатки подхода**:

* При перестановке местами слов или частей слов получаются сравнительно большие расстояния.
* Расстояния между совершенно разными короткими словами оказываются небольшими, в то время как расстояния между очень похожими длинными словами оказываются значительными.

Алгоритм Вагнера-Фишера решается использованием динамического программирования:

**Динамическое программирование** - способ решения сложных задач путём разбиения их на более простые подзадачи. Он применим к *задачам с оптимальной подструктурой*, выглядящим как *набор перекрывающихся подзадач*, сложность которых чуть меньше исходной. В этом случае время вычислений, по сравнению с «наивными» методами, можно значительно сократить.

*Оптимальная подструктура* в динамическом программировании означает, что оптимальное решение подзадач меньшего размера может быть использовано для решения исходной задачи.

В общем случае мы можем решить задачу, в которой присутствует оптимальная подструктура, проделывая следующие три шага.

1. Разбиение задачи на подзадачи меньшего размера.
2. Нахождение оптимального решения подзадач рекурсивно, проделывая такой же трехшаговый алгоритм.
3. Использование полученного решения подзадач для конструирования решения исходной задачи.
4. Различия между потоками и процессами, различие между различными планировщиками в Dask

У потоков общая память поэтому синхронизация между ними быстрее, у каждого процесса память и интерпретатор питона свой, поэтому нужно некоторое время для выделения отдельных процессов и синхронизации между ними результатов

**Процесс (process)**

Процесс – полноценная программа, для создания используется системный вызов, что занимает времени больше, чем создание потока

Разные процессы имеют изолированное адресное пространство и память.

Процессы взаимодействуют через системные механизмы межпроцессной коммуникации. Т.е. вся синхронизация между ними происходит на уровне системы

**Поток (thread)**

Поток является частью процесса и при его создании не используются системные вызовы.

Потоки имеют общую память и другие ресурсы, что благоприятно влияет на скорость синхронизации между ними.

**Когда использовать процесс, а когда использовать поток**

Если мы рассматриваем Python и Dask, то для оптимального использования времени выполнения программы необходимо использовать разные планировщики:

* Используем **потоки** только в том случае, когда из под Python делаем какой-либо системный вызов (scheduler="**threads**"). Например, операцию чтения/записи данных т.к. эти операции не относятся к объектам Python и, как следствие, GIL в данном случае не контроллирует ссылки на объект для сбора мусора.
* Используем **процессы** во всех объектах Python (scheduler="**processes**"). Если мы используем любой класс, наследуемый от object, то GIL хранит на него ссылки для сбора мусора и один из самых распространенных способов как-то с этим жить - запускать отдельные процессы со своими интерпретаторами Python, а потом уже синхронизировать между ними результаты.

Планировщик можно передавать в качестве строкового аргумента в dask, например, в dask.compute:

dask.compute(list\_с\_delayed, scheduler="processes")

dask.compute(list\_с\_delayed, scheduler="threads")

*Чтоб понять че это за GIL такой и почему он все контроллирует см вопрос 8.*

Если мы рассматриваем вопрос масштабирования вычислений, то у Dask также есть два семейства планировщиков задач:

1. **Планировщик для одной машины**: Этот планировщик предоставляет основные функции для локального процесса или пула потоков. Этот планировщик был создан первым и используется по умолчанию. Он прост и дешев в использовании, хотя может использоваться только на одной машине и не масштабируется

2. **Распределенный планировщик**: Этот планировщик более сложный, предлагает больше функций, но также требует немного больше усилий для настройки. Он может работать локально или распределяться по кластеру

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

***TF*** *— term frequency,* ***IDF*** *— inverse document frequency*

TF-IDF статистическая мера, используемая для оценки важности слова в контексте документа, являющегося частью коллекции документов или корпуса. Вес некоторого слова пропорционален частоте употребления этого слова в документе и обратно пропорционален частоте употребления слова во всех документах коллекции.

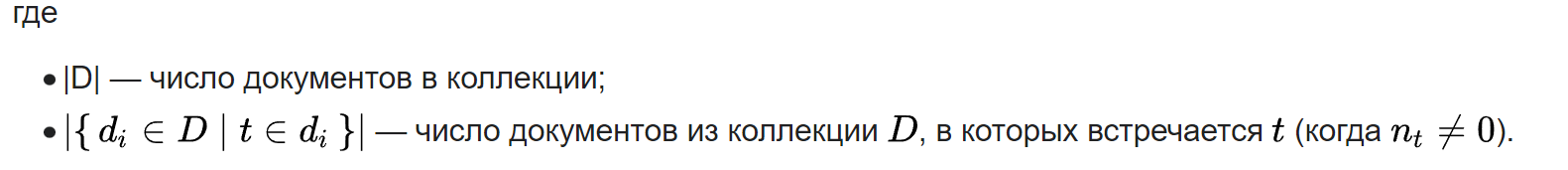
**TF** (*term frequency* — частота слова) — отношение числа вхождений некоторого слова к общему числу слов документа. Таким образом, оценивается важность слова



где nt есть число вхождений слова t в документ, а в знаменателе — общее число слов в данном документе.

**IDF** (*inverse document frequency* — обратная частота документа) — инверсия частоты, с которой некоторое слово встречается в документах коллекции. Учёт IDF уменьшает вес широкоупотребительных слов. Для каждого уникального слова в пределах конкретной коллекции документов существует только одно значение IDF.





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

Внутренне Dask кодирует алгоритмы в простом формате, включающем словари, кортежи и функции. Этот формат графа может использоваться отдельно от коллекций Dask.

**Пример:**

**def inc(i):**

**return i + 1**

**def add(a, b):**

**return a + b**

**x = 1**

**y = inc(x)**

**z = add(y, 10)**

**И теперь можно закодировать это как словарь таким образом:**

d = {‘x’: 1, ‘y’: (inc, ‘x’), ‘z’: (add, ‘y’, 10)}

Библиотека Dask в настоящее время содержит несколько планировщиков для выполнения этих графиков. Каждый планировщик работает по-разному, гарантируя разную производительность в разного характера задачах и работая в разных предметных областях.

В Dask существует два основных семейства планировщиков задач. (подробнее см. вопрос №31)

1. Локальные планировщики - для локального пк
2. Распределенные планировщики - для кластерных вычислений

Локальные планировщики по принципу разбиения задач на:

* процессы – processes

import dask

dask.config.set(scheduler=’processes’)

* потоки - local threads

import dask

dask.config.set(scheduler=’threads’)

* одиночный поток (без параллелизации) - single thread

import dask

dask.config.set(scheduler=’synchronouns’)

* Распределение Dask - Dask Distributed (local**)**

from dask.distributed import Client

client = Client()

# или

client = Client(processes=False0

Кластерный планировщик:

* Распределение Dask - Dask Distributed (cluster**)**

from dask.distributed import Client

client = Client(…)

df.x.sum().compute()

Кроме того, существует возможность писать свои планировщики под свои задачи.

При создании пользователем задачи, Dask анализирует её для того, чтобы вычислить её оптимальным способом.

**Определения**

Граф задач — это словарь, отображающий ключи к вычислениям.

Ключ — это хешируемое значение, которое не является таском

Task — это кортеж с вызываемым первым элементом. Задачи представляют собой атомарные единицы работы, предназначенные для выполнения одиночным исполнителем задач.

Мы представляем задачу(task) как кортеж, первый элемент которого - функция. А последующие элементы этого кортежа — это аргументы, передаваемые этой функции. Аргумент также может являться любым валидным вычислением.

**Правильные примеры использования:**

np.array([…])

(add, 1, 2)

(add, ‘x’, 2)

(add, (inc, ‘x’), 2)

(sum, [1, 2])

(sum, [‘x’, (inc, ‘x’)])

(np.dot, np.array([…]), np.array([…]))

[(sum, [‘x’, ‘y’]), ‘z’]

**Точка входа - функция Get**

Функция get служит точкой входа для всех планировщиков. Эта функция получает значение по переданному ей ключу. Этот ключ может ссылаться на хранимую информацию, как например в случае с “x” или так же может ссылаться на задачу, как в случае с “z”. Ну и в последнем случае, с “w”, get должен выполнить все необходимые вычисления для извлечения вычисленного значения.

from dask.threaded import get

from operator import add

dsk = {‘x’: 1, ‘y’: 2, ‘z’: (add, ‘x’, ‘y’), ‘w’: (sum, [‘x’. ‘y’, ‘z’])}

get(dsk, ‘x’)

Out: 1

get(dsk, ‘z’)

Out: 3

get(dsk, ‘w’)

Out: 6

1. Dask.Array – структура данных, специфика реализации и применения, процедура создания

Dask Array реализует подмножество интерфейса NumPy ndarray, используя алгоритмы в блочной форме. Большой массив разбивается на относительно небольшие блоки, которые обрабатываются независимо. Этот подход позволяет оперировать массивами, большими чем оперативная память и использовать все доступные ядра.

Координация задач, возникающих при исполнении блочной формы алгоритмов, осуществляется при помощи реализованного в Dask графа зависимостей задач.

Dask Array представляет собой сетку из массивов NumPy, обработку которых он организует, порождая для каждой операции со всем массивом множество операций с массивами NumPy.

Массивы NumPy могут:

* находится в оперативной в памяти
* находится в распределенной оперативной в памяти кластера (т.е. хранится на узлах кластера)
* находится на диске (по крайней мере часть времени вычислений).

Принцип организации данных в Dask Array: Массивы Dask координируют множество массивов Numpy, расположенных в виде блоков внутри сетки. Они поддерживают большое подмножество API Numpy.

Пример создания случайного массива:

import dask.array as da

x = da.random.random((10000), chunks=(1000))

Dask Array может читать из любого массива, например, структуры, поскольку он поддерживает numpy, как срезы.shape собственность с помощью dask.array.from\_array метод. Он также может читать из .npy а также .zarr файлы.

**Пример**

import dask.array as da

import numpy as np

arr = np.random.randit(1, 1000, (10000, 10000))

darr = da.from\_array(arr, chunks=(1000, 1000))

darr.npartitions

В настоящее время, Dask делает ленивую оценку каждого метода. Итак, чтобы на самом деле вычислить значение функции, вы должны использовать .compute() метод. Он будет вычислять результат параллельно в блоках, распараллеливая каждую независимую задачу в то время.

1. Dask.Array – поддерживаемые операции и отличия от NumPy ndarray

Dask Array реализует подмножество интерфейса NumPy ndarray, используя алгоритмы в блочной форме. Большой массив разбивается на относительно небольшие блоки, которые обрабатываются независимо. Этот подход позволяет оперировать массивами, большими чем оперативная память и использовать все доступные ядра.

Dask Array представляет собой сетку из массивов NumPy, обработку которых он организует порождая для каждой операции со всем массивом множество операций с массивами NumPy.

Dask Array поддерживает большинство интерфейсов NumPy, в частности:

* Арифметические операции и скалярные функции: +, \*, exp, log, …
* Агрегирующие функции (в т.ч. вдоль осей): sum(), mean(), std(), sum(axis=0), …
* Умножение матриц, свёртка тензоров: tensordot
* Получение срезов: x[:100, 500:100:-2]
* Прихотливое индексирование вдоль одной оси: x[:, [10, 1, 5]]
* Работу с протоколами массивов \_\_array\_\_ и \_\_array\_ufunc\_\_
* Некоторые операции линейной алгебры: svd, qr, solve, solve\_triangular, lstsq

**Что НЕ поддерживает от NumPy**

* Не реализована большая часть пакета np.linalg
* Не поддерживаются операции с массивами неизвестного размера
* Операции наподобие sort , которые по своей сути сложно выполнять параллельно не поддерживаются. Зачастую, вместо таких операций предлагается альтернативная функция, дружественная к параллельному вычислению
* Не поддерживаются операции типа tolist, т.к. это очень неэффективно для больших наборов данных, тем более что, обход этих данных в циклах очень неэффективен.

1. Dask.Bag - структура данных, специфика реализации и применения, процедура создания DaskBag

**Мультимножество (bag, multiset) в математике** - обобщение понятия множества, допускающее включение одного и того же элемента по нескольку раз.

* list: упорядоченная коллекция, допускающая повторы элементов. Пример: [1, 2, 3, 2]
* bag: неупорядоченная коллекция, допускающая повторы элементов. Пример: 1, 2, 2, 3

Таким образом, bag можно рассматривать как **список, не гарантирующий порядка элементов.**

**Реализация Dask.Bag** основана на координации множества списков или итераторов, каждый из которых представляет собой сегмент большой коллекции. Данная реализация обеспечивает:

* параллельное выполнение операций
* потребность в небольшом объеме памяти за счет использования итераторов Python и ленивых вычислений. Это обеспечивает возможность обработки данных больших чем объем оперативной памяти, даже при использовании всего одного сегмента.

Dask.Bag хорошо подходит для распараллеливания простой обработки неструктурированных или полу-структурированных данных, таких как: текстовые данные, файлы логирования, записи в формате JSON, специальных объектов Python и т.д.

Если выполнение задачи возможно при помощи Dask.DataFrame или Dask.Array , то стоит выбрать эти варианты, так как основной объем вычислений будет выполняться за счет быстрых библиотек написанных на компилируемых языках, тогда как Dask.Bag использует только код на Python. При этом преимуществом Dask.Bag является возможность использовать любые пользовательские функции написанные на Python и существенно меньшие требования к наличию строгой структуры у обрабатываемых данных.

**Специфика реализации и использования**

По умолчанию, Dask.Bag использует для исполнения планировщик dask.multiprocessing. Это позволяет обойти проблему GIL и полноценно использовать несколько процессорных ядер для объектов реализованных на чистом Python. Минусом этого подхода является наличие больших накладных расходов при обмене данных между исполнителями, что важно для производительности вычислений, требующих интенсивного обмена данными. Это редко бывает проблемой, так как типичный поток задач для Dask.Bag подразумевает:

* или чрезвычайно параллельные вычисления
* или обмен небольшим объемом данных в процессе свертки (англ. folding, также известна как reduce, accumulate).

**Чрезвычайная параллельность (embarrassingly parallel)** - тип задач в системах параллельных вычислений, для которых не требуется прилагать больших усилий при разделении на несколько отдельных параллельных задач (распараллеливании).

**Ограничения Dask.Bag**

Dask.Bag позволяет выполнять любую функцию на Python, эта универсальность имеет свою цену. Dask.Bag имеет следующие ограничения:

* по умолчанию обработка Dask.Bag выполняется планировщиком, на базе multiprocessing, что создает ряд ограничений
* Bag является неизменяемой структурой данных, таким образом нет возможности изменить единичный элемент Dask.Bag не выполнив операцию преобразования для всей структуры данных
* операции над Bag медленнее, чем операции над Array/DataFrame по тем же причинам, по которым операции на Python медленнее чем аналогичные операции в NumPy и Pandas
* операция Bag.groupby выполняется медленно, по возможности нужно использовать вместо нее Bag.foldby или преобразовывать данные в структуру DataFrame.

**Создание Dask Bag**

Bag можно создать из последовательности (итерируемого объекта) Python, из файла (файлов) и т.п. Данные в Bag разбиваются на сегменты (блоки), каждый из которых содержит множество элементов исходного набора данных.

# набор данных из целых чисел

b = db.from\_sequence([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], npartitions=2) #разбиваем последовательность

b.take(3) # выводит 3 элемента из Bag

b = db.read\_text(‘myfile.txt’)

b = db.read\_text([‘myfile.1.txt’, ‘myfile.2.txt’, …])

b = db.read\_text(‘myfile.\*.txt’)

# данные загружаются из набора заархивированных файлов формата JSON

# элементами Bag будут строки файлов, сегментация будет проведена по файлам

data\_path = ‘./data’

import os

b = db.read\_text(os.path.jaoin(data\_path, ‘accounts.\*.json.gz’))

b.take(1)

1. API Dask.Bag – функции группировки и свертки

**SHUFFLE:**

**Примеры:**

**groupby()**

b = db.from\_sequence(range(10))

# группирует значения из bag по результатам функции key

list(b.groupby(lambda x: x%2 == 0))

Out: [(False, [7, 9, 1, 3, 5]), (True, [8, 0, 4, 2, 6])]

**distinct()**

b = db.from\_sequence([‘Alice’, ‘Bob’, ‘Alice’])

# возвращает уникальные значения из bag

list(b.distinct())

Out: [‘Alice’, ‘Bob’]

**frequencies()**

# подсчитывает частоты для элементов bag

b = db.from\_sequence([‘Alice’, ‘Bob’, ‘Alice’])

list(b.frequencies())

Out: [(‘Alice’, 2), (‘Bob’, 1)]

**Функции светки:**

**reduce() -** вычисляет функцию от двух элементов последовательно для элементов последовательности слева направо таким образом, что результатом вычисления становится единственное значение, которое становится первым аргументом для следующей итерации применения funct.

**Примеры:**

**fold()**

b = db.from\_sequence(range(5))

# fold – параллельная версия reduce, binop – бинарный оператор используемый для свертки

b.fold(op.add).compute()

Out: 10

**foldby() -** комбинирует свертку и группировку и выполняет эту операцию намного эффективнее последовательного применения groupby и reduce.

b = db.from\_sequence(range(10))

iseven = lambda x: x%2 == 0

# параметр key – определение ключа для группировки, binop – функция для проведения свертки

# сумма четных и нечетных чисел из bag:

list(b.foldby(iseven, op.add))

Out: [(True, 20), (False, 25)]

1. Организация вычислений с помощью Map / Filter / Reduce : общий принцип и специфика параллельной реализации обработки данных в Dask.Bag

Dask.Bag реализует такие операции, как map, filter, fold (аналог reduce) и groupby над коллекциями объектов Python.

Реализация Dask.Bag основана на координации множества списков или итераторов, каждый из которых представляет собой сегмент большой коллекции. Данная реализация обеспечивает:

* параллельное выполнение операций
* потребность в небольшом объеме памяти за счет использования итераторов Python и **ленивых вычислений**. Это обеспечивает возможность обработки данных больших чем объем оперативной памяти, даже при использовании всего одного сегмента.

**Ленивые вычисления** (lazy evaluation, или отложенные вычисления) — стратегия вычислений, согласно которой вычисления откладываются до тех пор, пока не понадобится их результат.

По умолчанию, Dask.Bag использует для исполнения **планировщик** dask.multiprocessing.

* Это позволяет **обойти проблему GIL** и полноценно использовать несколько процессорных ядер для объектов реализованных на чистом Python.
* Минусом этого подхода является наличие больших накладных расходов при обмене данных между исполнителями, что важно для производительности вычислений, требующих интенсивного обмена данными. Это редко бывает проблемой, так как типичный поток задач для Dask.Bag подразумевает:
  + или чрезвычайно параллельные вычисления
  + или обмен небольшим объемом данных в процессе **свертки** (англ. folding, также известна как reduce, accumulate).

**Чрезвычайная параллельность** (embarrassingly parallel) - тип задач в системах параллельных вычислений, для которых не требуется прилагать больших усилий при разделении на несколько отдельных параллельных задач (распараллеливании).

* Чаще всего **не существует зависимости (или связи) между параллельными задачами**, то есть их результаты не влияют друг на друга.
* Чрезвычайно параллельные задачи **практически не требуют согласования** между результатами выполнения отдельных этапов, что отличает их от задач распределенных вычислений, которые требуют связи промежуточных результатов.
* Такие задачи **легки для исполнения массово параллельных системах** (кластерах с очень большим количеством вычислительных узлов).

Map

*b.map(lambda x: x \*\* 2)*

Filter

*b.filter(lambda x: x % 2 == 0)*

Reduce

*Bag.fold(binop[, combine, initial, split\_every])*

**Fold**

import dask.bag as db

text = db.from.from\_sequence(‘Fold is like the builtin function reduce’.split())

text = text.map(list)

def add\_to\_set(acc, x):

return acc | set(x)

text.fold(add\_to\_set, set.union, initial=set()).compute()

**Map**

Встроенная функция map() позволяет применить функцию к каждому элементу последовательности

* Функция имеет следующий формат: mар(<Функция>, <Последовательность1>[, ... , <ПоследовательностьN>])
* Функция возвращает объект, поддерживающий итерацию, а не список.

*list(map(lambda x: x\*\*2, range(10)))*

Функции map() можно передать несколько последовательностей. В этом случае в функцию обратного вызова будут передаваться сразу несколько элементов, расположенных в последовательностях на одинаковом смещении.

**Filter**

Функция filter() позволяет выполнить проверку элементов последовательности.

* Формат функции: filtеr(<Функция>, <Последовательность>)
* Если в первом параметре вместо названия функции указать значение None, то каждый элемент последовательности будет проверен на соответствие булевому значению True.
* Если элемент в логическом контексте возвращает значение False, то он не будет добавлен в возвращаемый результат.
* Функция возвращает объект, поддерживающий итерацию, а не список.

*list(filter(lambda x: x % 2 == 0, range(10)))*

**Reduce**

functools.reduce(funct, iterable[, initializer])

Вычисляет функцию от двух элементов последовательно для элементов последовательности слева направо таким образом, что результатом вычисления становится единственное значение, которое становится первым аргументом для следующей итерации применения funct.

*from functools import reduce*

*reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])*

Левый аргумент функции funct (аргумента reduce) — это аккумулированное значение, правый аргумент - очередное значение из списка.

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

1. API Dask.Bag – функции мэппинга, фильтрации и преобразования

Объекты Bag поддерживают стандартное API, аналогичное имеющемуся в стандартной библиотеке Python и библиотеках toolz или pyspark. В частности, имеются функции, отвечающие за маппинг (map и т.п.), фильтрацию и группировку (filter, groupby и т.п.) и свертку (reduce и т.п.).

**MAP**

Встроенная функция map() позволяет применить функцию к каждому элементу последовательности. Функция возвращает объект, поддерживающий итерацию, а не список.

**Примеры:**

**map()**

import dask.bag as db

b1 = db.from\_sequence(range(6), npartitions=2)

b1.map(lambda x: x + 1).compute()

Out: [1, 2, 3, 4, 5, 6]

b2 = db.from\_sequence(range(6, 12), npartitions=2)

b1.map(op.add, b2).compute()

Out: [6, 8, 10, 12, 14, 16]

**starmap()**

b = db.from\_sequence([(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)], npartitions=2)

b.starmap(op.add).compute()

Out: [3, 7, 11, 15, 19]

**pluck()**

b = db.from\_sequence([{‘name’: ‘Alice’, ‘credits’: [1, 2, 3]},

{‘name’: ‘Bob’, ‘credits’: [10, 20]},

{‘name’: ‘Rob’}])

b.pluck(‘name’).compute()

Out: [‘Alice’, ‘Bob’, ‘Rob’]

**FILTER**

Функция filter() позволяет выполнить проверку элементов последовательности.

**Примеры:**

b = db.from\_sequence(range(5))

list(b.filter(lambda x: x%2 == 0))

Out: [0, 2, 4]

**remove()**

b = db.from\_sequence(range(5))

# удаляет все элементы, для которых выполняется предикат

list(b.remove(lambda x: x%2 == 0))

Out: [1, 3]

**BAG**

**Примеры:**

**concat()**

a = db.from\_sequence([1, 2, 3])

b = db.from\_sequence([4, 5, 6])

print(a.npartitions)

c = db.concat([a, b])

print(c. npartitions)

list(c)

Out: 3

Out: 6

Out: [1, 2, 3, 4, 5, 6]

**zip()**

evens = db.from\_sequence(range(0, 20, 2), partition\_size=4)

odds = db.from\_sequence(range(1, 20, 2), partition\_size=4)

pairs = db.zip(evens, odds)

list(pairs)

Out: [(0,1), (2, 3), (4, 5), (6, 7), (8, 9), (10, 11), (12, 13), (14, 15), (16, 17), (18, 19)]

**join()**

list(people.join(fruit\_l, lambda x: x[0]))

Out: [(‘Apple’, ‘Alice’), (‘Apricot’, ‘Alice’), (‘Banana’, ‘Bob’)]

**flatten()**

people.map(list).comute()

Out: [[‘A’, ‘l’, ‘i’, ‘c’, ‘e’], [‘B’, ‘o’, ‘b’], [‘C’, ‘h’, ‘a’, ‘r’, ‘l’, ‘i’, ‘e’], [‘R’, ‘o’, ‘b’, ‘e’, ‘r’, ‘t’]]

people\_flat = people.map(list).flatten()

print(people\_flat.npartitions)

print(list(people\_flat))

Out: [‘A’, ‘l’, ‘i’, ‘c’, ‘e’, ‘B’, ‘o’, ‘b’, ‘C’, ‘h’, ‘a’, ‘r’, ‘l’, ‘i’, ‘e’, ‘R’, ‘o’, ‘b’, ‘e’, ‘r’, ‘t’]