**Материалы для самостоятельного изучения. Парадигмы программирования и функциональное программирование**

Языки программирования предлагают различные средства для декомпозиции задачи. Существует несколько парадигм программирования:

* Императивное (структурное, процедурное) программирование: программы являются последовательностью инструкций, которые могут читать и записывать данные из памяти. При изучении предыдущих тем мы пользовались, в основном, императивной парадигмой. Ряд языков, такие как Паскаль (не Object Pascal) или C являются яркими предтавителями императивных языков.
* Декларативное программирование: описывается задача и ожидаемый результат, но не описываются пути её решения. Ярким представителем является язык запросов к базам данных SQL: большая часть внутреннего устройства скрыта в СУБД, программист описывает только структуру базы данных и ожидаемый результат запросов.
* Объектно-ориентированное программирование: программы манипулируют наборами объектов, при этом объекты обладают сохраняющимся во времени состоянием и методами для изменения этого состояния (или создания новых объектов). Мы познакомимся с ООП подробнее на одной из следующих лекций. Примером языка с объектно-ориентированной парадигмой является Java, ООП также поддерживается в Питоне и C++.
* Функциональное программирование: задача разбивается на набор функций. В идеале, функции только принимают параметры и возвращают значения, не изменяя состояния объектов или программы. Представителем функциональных языков является Haskell.

Иногда, также, выделяют и другие парадигмы программирования.

Некоторые языки предназначены, в основном, для написания программ в рамках одной парадигмы, другие же поддерживают несколько парадигм. Например, Питон и C++ поддерживают различные парадигмы программирования. Разные части программы можно писать в разных парадигмах, например, использовать функциональный стиль для обработки больших данных, объектно-ориентированный подход для реализации интерфейса и императивное программирование для промежуточной логики.

**Функциональное программирование**

Несмотря на то, что в функциональном стиле писать достаточно сложно и непривычно, функциональное программирование имеет массу плюсов:

* Достаточно легко доказать формальную корректность алгоритмов. Хотя доказательство, даже для функциональных программ, часто намного длиннее, чем сама программа, но для фундаментальных алгоритмов, которые широко используются, лучше иметь формальное доказательство, чтобы не попадать в глупые ситуации. Строить формальные доказательства императивных программ намного сложнее.
* Для программ, написанных в функциональном стиле, легко проводить декомпозицию, отладку и тестирование. Когда вся программа разбита на функции, выполняющие элементарные действия, то их разработка и проверка занимает намного меньше времени.
* Для функциональных программ легко автоматически проводить распараллеливание, векторизацию и конвейеризацию.

Задача распараллеливания выполнения программ крайне актуальна сейчас, когда скорости ядер процессоров практически перестали расти. Единственным способом ускорить выполнение программ является их параллельное вычисление на нескольких устройствах.

Последние успехи в решении задач машинного обучения связаны с использованием большого количества простых вычислительных устройств, например, рассчетов на видеокарте.

В функциональных программах не принято вносить изменения в объекты (и, вообще говоря, желательно, чтобы все объекты были неизменямыми). Поэтому, если нам нужно посчитать результат вычисления двух функций, то во многих случаях можно делать это параллельно на разных ядрах процессора. Такое распараллеливание легко сделать автоматически.

Векторизация - это когда одни и те же действия выполняются над большим набором данных. Тогда данные можно нарезать на куски и раскидать по разным вычислительным устройствам, а затем собрать результат вычислений в один объект.

Конвейеризация - это разбиение вычислений на несколько этапов, причём данные поступают на следующий этап обработки по времени готовности. Например, если нам нужно попарно перемножить значения в векторах A и B, а затем сложить со значениями в векторе C, то мы можем посчитать несколько первых результатов подсчета произведения и уже выполнять с ними сложение, не дожидаясь подсчета остальных значений. Это позволяет быстрее получить первые результаты и занять еще большое количество вычислительных устройств параллельно.

## Встроенные функции для работы с последовательностями

В языке Питон есть много функций, которые принимают в качестве параметра iterable и могут сделать что-то полезное. С некоторыми из них, такими как sorted или map мы уже немного знакомы. Рассмотрим еще некоторые из них:

sum - находит сумму всех элементов iterable.

min, max - находит минимум и максимум в последовательности iterable.

map - умеет принимать более двух параметров. Например, такая запись map(f, iterA, iterB) вернет iterable со значениями f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), ...

filter(predicate, iterable) - применяет функцию predicate ко всем элементам iterable и возвращает iterable, который содержит только те элементы, которые удовлетворяли предикаты (т.е. функция predicate вернула True). Например, так может выглядеть решение задачи о поиске минимального положительного элемента в списке:

|  |  |
| --- | --- |
| 1 | print(min(filter(lambda x: x > 0, map(int, input().split())))) |

Здесь в качестве предиката использована лямбда-функция, которая возвращает True при значении параметра больше 0, а в качестве входного iterable - результат вызова map для функции int и нарезанного на слова ввода. Функция min применена к тому iterable, который был возвращен функцией filter.

enumerate - возвращает кортежи из номера элемента (при нумерации с нуля) и значения очередного элемента. С помощью enumerate, например, удобно перебирать элементы iterable (доступ по индексу в которых невозможен) и выводить номера элементов, которые обладают некоторым свойством:

|  |  |
| --- | --- |
| 1 | f = open('data.txt', 'r', encoding='utf8') |
| 2 | for i, line in enumerate(f): |
| 3 | if line.strip() == '': |
| 4 | print('Blank line at line', i) |

any, all - возвращают истину, если хотя бы один или все элементы iterable истинны соответственно. Например, так можно проверить, не превышают ли все члены последовательности 100 по модулю:

|  |  |
| --- | --- |
| 1 | print(all(map(lambda x: abs(int(x)) <= 100, input().split()))) |

zip(iterA, iterB, ...) - конструирует кортежи из элементов (iterA[0], iterB[0], ...), (iterA[1], iterB[1], ...), ...

## Пример решения задачи в функциональном стиле

С помощью этих функций можно решить достаточно сложные задачи, без использования циклов и условных операторов. Например, задачу из домашнего задания по сортировкам про такси: в первой строке задано количество людей и автомобилей такси, в следующих двух строках расстояние в километрах для каждого человека и цена за километр для каждого такси. Необходимо сопоставить каждому человеку номер такси, чтобы суммарная цена поездок была минимальна. Идея решения заключается в том, чтобы люди, которым ехать дальше, ехали на более дешевых такси:

|  |  |
| --- | --- |
| 1 | input()  # Пропускаем количества, обойдемся без них |
| 2 | people = map(int, input().split()) |
| 3 | sortedPeople = sorted(enumerate(people), key=lambda x: x[1]) |
| 4 | taxi = map(int, input().split()) |
| 5 | sortedTaxi = sorted(enumerate(taxi), key=lambda x: x[1], reverse=True) |
| 6 | ans = zip(sortedPeople, sortedTaxi) |
| 7 | sortedAns = sorted(ans, key=lambda x: x[0][0]) |
| 8 | print(\*map(lambda x: x[1][0], sortedAns)) |

А теперь избавимся от переменных, подставляя на их места исходные значения. Сделаем переносы для удобного восприятия:

|  |  |
| --- | --- |
| 1 | input()  # Пропускаем количества, обойдемся без них |
| 2 | print( |
| 3 | \*map( |
| 4 | lambda x: x[1][0], |
| 5 | sorted( |
| 6 | zip( |
| 7 | sorted( |
| 8 | enumerate(map(int, input().split())), |
| 9 | key=lambda x: x[1] |
| 10 | ), |
| 11 | sorted( |
| 12 | enumerate(map(int, input().split())), |
| 13 | key=lambda x: x[1], |
| 14 | reverse=True |
| 15 | ) |
| 16 | ), |
| 17 | key=lambda x: x[0][0] |
| 18 | ) |
| 19 | ) |
| 20 | ) |

Это та же программа!

## itertools, functools

## Генерация комбинаторных объектов itertools

В Питоне есть библиотека itertools, которая содержит много функций для работы с итерируемыми объектами. С этими функциями можно ознакомиться в официальной документации к языку.

Нам наиболее интересны функции, генерирующие комбинаторные объекты.

itertools.combinations(iterable, size) - генерирует все подмножества множества iterable размером size в виде кортежей. Это может быть использовано вместо вложенных циклов при организации перебора. Например, мы можем неэффективно решить задачу о поиске трех чисел в последовательности, дающих наибольшее произведение:

|  |  |
| --- | --- |
| 1 | from itertools import combinations |
| 2 |  |
| 3 | nums = list(map(int, input().split())) |
| 4 | combs = combinations(range(len(nums)), 3) |
| 5 | print(max(map(lambda x: nums[x[0]] \* nums[x[1]] \* nums[x[2]], combs))) |

itertools.permutations(iterable) - генерирует все перестановки iterable. Существует вариант функции с двумя параметрами, второй параметр является размером подмножества. Тогда генерируются все перестановки всех подмножеств заданного размера.

itertools.combinations\_with\_replacement(iterable, size) - генерирует все подмножества iterable размером size с повторениями, т.е. одно и то же число может входить в подмножество несколько раз.

## partial, reduce, accumulate

Модуль functools содержит некоторые функции, которые могут полезны для обработки последовательностей и не только.

Функция functools.partial предназначена для оборачивания существующих функций с подстановкой некоторых параметров. Например, мы можем создать функцию для печати в файл, чтобы каждый раз не указывать какие-то параметры. Например, существует вариант функции int с двумя параметрами: первый - это переменная, которую необходимо преобразовать в число, а второй - система счисления в которой записано число. С помощью partial мы можем создать функцию-обёртку, преобразующую строки из 0 и 1 в числа:

|  |  |
| --- | --- |
| 1 | from functools import partial |
| 2 |  |
| 3 | binStrToInt = partial(int, base=2) |
| 4 | print(binStrToInt('10010')) |

В модуле functools также содержатся функции для обработки последовательностей.

functools.reduce(func, iterable) позволяет применить функцию ко всем элементам последовательности, используя в качестве первого аргумента накопленный результат. Например, если в последовательности были элементы myList = [A, B, C], то результатом применения reduce(f, myList) будет f(f(A, B), C). С помощью reduce, например, можно найти НОД всех чисел в iterable:

|  |  |
| --- | --- |
| 1 | from functools import reduce |
| 2 |  |
| 3 | def gcd(a, b): |
| 4 | while b != 0: |
| 5 | a, b = b, a % b |
| 6 | return a |
| 7 |  |
| 8 | print(reduce(gcd, map(int, input().split()))) |

itertools.accumulate(iterable, func), которая возвращает iterable со всеми промежуточными значениями, т.е. для списка [A, B, C] accumulate вернет значения A, f(A, B), f(f(A, B), C). Например, можно узнать максимальный элемент для каждого префикса (некоторого количества первых элементов) заданной последовательности:

|  |  |
| --- | --- |
| 1 | from itertools import accumulate |
| 2 |  |
| 3 | print(\*accumulate(map(int, input().split()), max)) |

## Итераторы и генераторы

В Питоне итераторы - это объекты, которые имеют внутреннее состояние и метод \_\_next\_\_ для перехода к следующему состоянию. Например, можно сконструировать итератор от списка и перебрать все значения:

|  |  |
| --- | --- |
| 1 | myList = [1, 2, 3] |
| 2 | for i in iter(myList): |
| 3 | print(i) |
| 4 | for i in myList: |
| 5 | print(i) |

В этой программе два цикла эквивалентны.

В Питоне можно создавать и свои итераторы. Их разумно использовать в том случае, если нужно перебрать большое количество значений и существует правило, по которому можно получить следующее значение, однако хранений всех этих значений не имеет смысла, т.к. они пригодятся только один раз.

Для создания итераторов в Питоне используется специальный вид функций, называемых генераторами. В обычной функции return прекращает работу функции. В генераторе вместо return используется оператор yield, который также возвращает значение, но не прекращает выполнение функции, а приостанавливает его до тех пор, пока не потребуется следующее значение итератора. При этом работа функции продолжится с того места и в том состоянии, в котором она находилась на момент вызова yield. Посмотрим, как может быть реализован генератор, аналогичный стандартному range с одним параметром:

|  |  |
| --- | --- |
| 1 | def myRange(n): |
| 2 | i = 0 |
| 3 | while i < n: |
| 4 | yield i |
| 5 | i += 1 |
| 6 | for i in myRange(10): |
| 7 | print(i) |

Генераторы могут иметь и сложную рекурсивную структуру. Например, мы можем написать генератор, который будет выдавать все числа заданной длины, цифры в которых не убывают и старшая цифра не превосходит заданного параметра:

|  |  |
| --- | --- |
| 1 | def genDecDigs(cntDigits, maxDigit): |
| 2 | if cntDigits > 0: |
| 3 | for nowDigit in range(maxDigit + 1): |
| 4 | for tail in genDecDigs(cntDigits - 1, nowDigit): |
| 5 | yield nowDigit \* 10\*\*(cntDigits - 1) + tail |
| 6 | else: |
| 7 | yield 0 |
| 8 |  |
| 9 | print(\*genDecDigs(2, 3)) |

Вывод этой программы будет выглядеть так: 0 10 11 20 21 22 30 31 32 33

В этой программе рекурсивный генератор перебирал все допустимые цифры в качестве той, которая должна стоять на заданной позиции и генерировал все возможные последующие цифры.

Также в этой программе мы использовали одну особенность функции print: если переди именем iterable (а результат, возвращаемый генератором является iterable) поставить \*, то будут напечатны все значения через пробел.

Результат работы генератора можно сохранить, например, в список с помощью функции list, как мы уже делали это с результатом работы range.