0.1. Введение в CV и Python

Примеры задач. Синтаксис языка, стандартная библиотека, работа с пакетами, NumPy

Никита Ковалев

19 февраля 2022

Примерный план

1. О задачах компьютерного зрения
2. Python: начало работы
3. Python: основы
4. Python: особенности и трюки
5. Python: массивы NumPy

Обо мне

- 2014 — 2020: Мехмат ЮФУ, Unversity of Twente. BSc & MSc по прикладной математике и информатике
- 2017 — 2021: Zuzex, ведущий разработчик и тимлид, главный по компьютерному зрению
- 2022 — н.в.: AndersenLab, ведущий разработчик по Python back-end

Для связи: Telegram

Почта: nick.tommybee37@gmail.com

Задачи компьютерного зрения

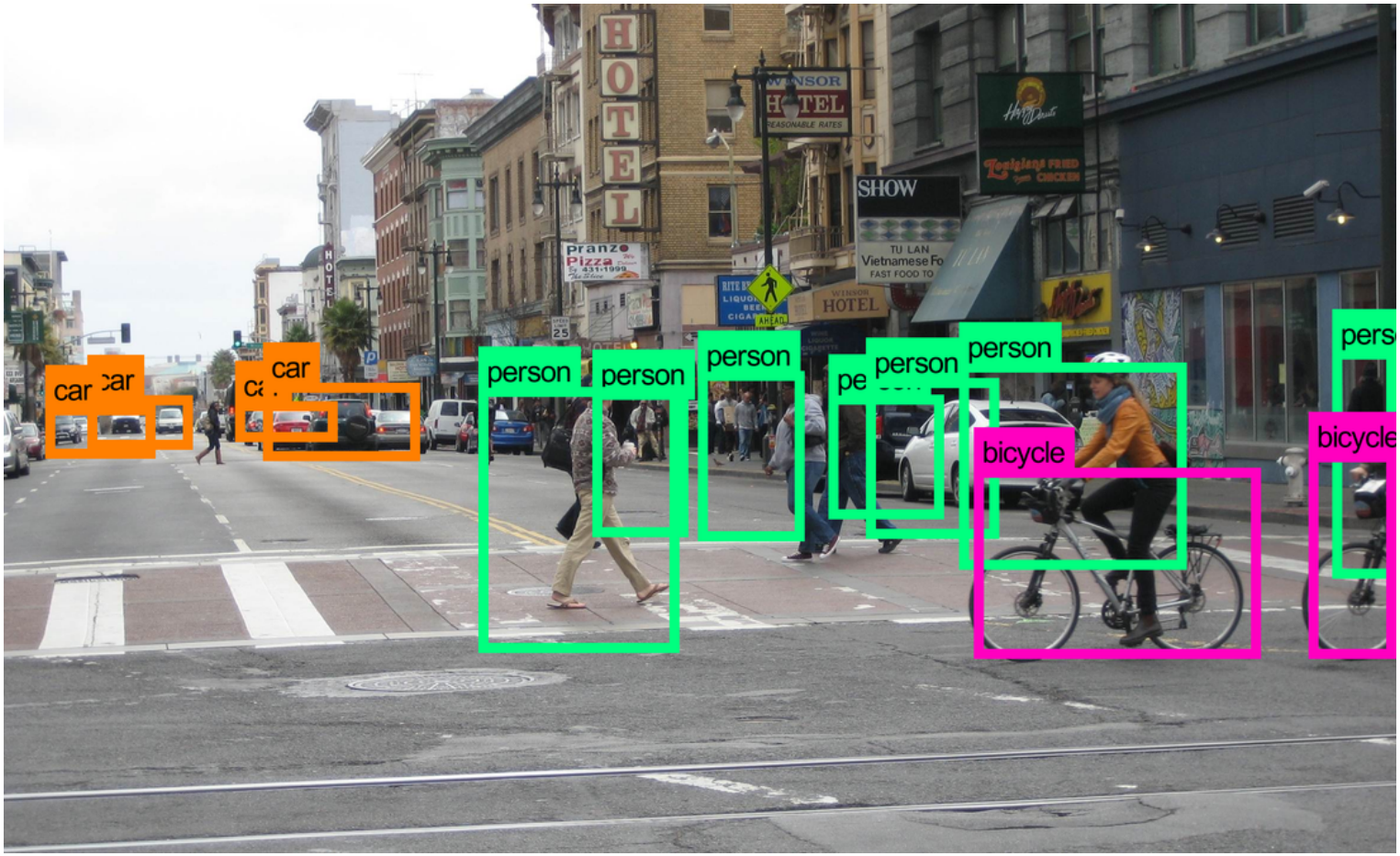


Рис. 1: Detection

Задачи компьютерного зрения

Рис. 2: Segmentation

Задачи компьютерного зрения

Рис. 3: Tracking

Задачи компьютерного зрения

Рис. 4: Recognition

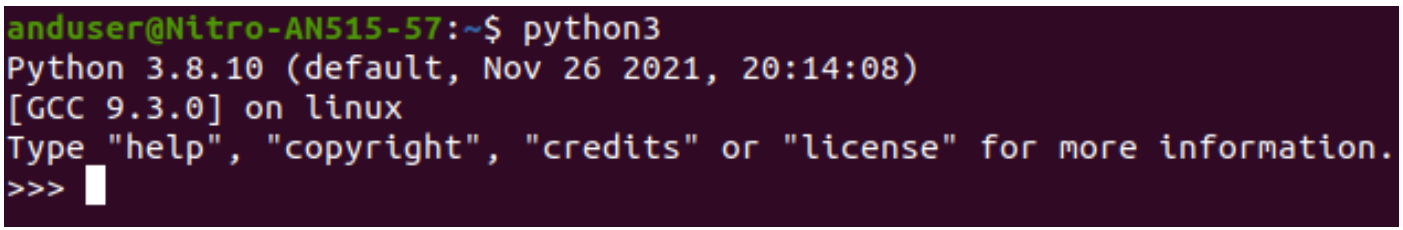
Python: установка, запуск

Для установки акутальной версии Python на системах семейства Debian (Ubuntu, Linux Mint, etc.) достаточно выполнить команду

```
$ sudo apt-get install python3.8
```

Для систем семейства RPM (Fedora, RedHat, etc.) apt-get меняется на pacman, а к названию библиотеки в начало добавляется lib (т.е. получится libpython3.8)

Python — интерпретируемый язык, поэтому в нём доступен интерактивный режим выполнения инструкций (кода). Достаточно просто запустить команду python3 без аргументов:



Установка пакетов и их учёт

У языка есть свой пакетный менеджер с похожим интерфейсом: pip. Его можно установить как пакет (python3.8-pip), а можно воспользоваться предлагаемым самими разработчиками способом.

Установка пакетов осуществялется при помощи команды

```
pip install <library-name>[==version]
```

Для реальных проектов считается хорошим тоном иметь файл requirements.txt, содержащий спецификации по всем используемым, не входящим в стандартный набор библиотекам (и их версиям!). Например:

```
numpy==1.14.1
opencv-python>=4.2
```

Помимо этого, можно использовать ~=, это позволит обозначить нижнюю границу версии, но допустить более новые в рамках ничего не ломающей мажорной версии. Более подробно почитать можно здесь.

Подключение пакетов

Для подключения пакетов используются ключевые слова `import`, `from` и `as`. Существует несколько возможных конструкций, основные принципы построения которых представлены ниже:

```
from typing import List, Optional
from time import *
import numpy as np
```

- Импорт отдельных имен
- Нежелательный импорт всех имен пространства
- Импорт всех имен, но с использованием имени модуля (с алиасом)

При импортировании имен из модулей разного характера следует соблюдать порядок: первыми идут стандартные библиотеки Python, далее установленные сверх стандартных пакеты, а в последнюю очередь — написанные в рамках проекта пользовательские модули.

Пользовательские пакеты

```
- module1
  - __init__.py
  - feature1.py
- module2
  - __init__.py
  - feature2.py
- main.py
```

Для создания пользовательского модуля достаточно внутри проекта создать папку с имеющимся внутри файлом `__init__.py`, который может быть как пустым, так и содержать импорты и определения классов и методов. Имея описанную выше схему проекта, допустим, что реализация метода в `feature2.py` зависит от метода `func1` из `feature1.py`, тогда импорт будет выглядеть следующим образом:

```
from ..module1.feature1 import func1
```

Если же метод `func1` содержался бы в `module1/__init__.py`, первая часть импорта превратилась бы просто в `from ..module1`.

Запуск программ

Для того, чтобы код на Python был исполнен при подстановке файла в команду python3, достаточно просто осуществить операции и вызовы в самой внешней области видимости - вне любых классов и функций. Однако, выполнение инструкций в таком неоформленном формате считается плохой практикой (хотя есть исключения). Вместо этого код, который следует исполнить при запуске файла, помещается в специальную область видимости в самом конце файла, аналогичную `int main() {}` в C/C++

```
def do_stuff():
    pass
# Плохо
do_stuff()
if __name__ == '__main__':
    # Корректно
    do_stuff()
```

Правила синтаксиса

- Области видимости определяются отсупами (обычно — 4 пробела) и двоеточием в конце выражений, определеяющих область
- Функции и классы разделяются **двумя** пустыми строками, методы внутри класса — **одной**
- Наименования классов — CamelCase, наименования методов и переменных — snake_case
- Статические поля класса определяются на одном уровне с методами, к статическим методам необходимо применять декоратор @staticmethod
-
-

Циклы и условия

Любое выражение, определяющее вложенный scope, офомляется без обязательных скобок и завершается двоеточием:

```
for i in (1, 2, 3, 4, 5, 6, 7, 8, 9, 10):
    if i % 2:
        print(i)
```

В Python нет традиционных циклов с инкрементами, но есть возможность легко определить аналоги с помощью стандартного средства `range`. Помимо этого, любой итерируемый контейнер (список, кортеж, множество, массив и т.д.) можно использовать после оператора `in`. Также есть очень удобный метод `enumerate`, позволяющий одновременно получать и текущий элемент, и его индекс.

Альтернатива циклам

Помимо обычных циклов, можно осуществлять операции, зависящие от индекса и/или значения элементов итерируемых сущностей, с помощью list comprehensions. Например, заполнение списка элементами (основное применение):

```
new_list = [i ** 2 for i in range(1, 11)]
```

Если заменить квадратные скобки на круглые, можно создавать генераторы: объекты, по которым можно итерировать, причём каждый очередной объект создается и подсчитывается только по требованию, что всегда эффективнее, например, по памяти, чем списки, а иногда (в случае early stopping) еще и по времени, потраченном на создание. Известный уже нам `range` является генератором.

```
gen_elems = (i ** 2 for i in range(1, 11))
for elem in gen_elems:
    if condition(elem):
        print(elem)
```

Структуры данных (порядковые коллекции)

В стандартной библиотеке Python есть несколько различных широко используемых типов коллекций. Рассмотрим их с точки зрения функциональности и временной сложности операций (см. документацию).

- `list` — аналог массива в Python — обсепечивает порядок элементов, в котором они были добавлены в список, поддерживает соответствующую индексацию со следованием заданному порядку, а также слайсы — срез цельной части списка. Почти все операции по изменению списка выполняются за линейное время, то же самое касается и удаления и поиска элементов.
- `tuple` — неизменяемая коллекция (кортеж), служащая для хранения и переноса объектов фиксированных количества и качества. Заменять объекты внутри кортежа на другие нельзя, но если сам объект может менять своё внутренне состояние, это не запрещено.

Структуры данных (коллекции на основе hashmap)

- `set` — реализация математического множества на основе хэш-таблицы. Благодаря этому доступ к любым элементам и проверка на вхождение по значению осуществляются в среднем случае за константное время (а как оно обозначается в Big O нотации?).
- `dict` — словарь/map для обеспечения работы с парами ключ-значение, основан также на хэш-таблице. Позволяет задавать однозначное соответствие не только из ряда 0, 1, 2... во множество элементов, как в списке, но и используя любые хэшабельные объекты в качестве ключей (чаще всего это строки).

Принцип мутабельности

Все объекты в памяти, вообще говоря, имеют свои уникальные идентификаторы. Но что произойдет, если мы попытаемся поменять его? Это и описывает принцип мутабельности — если объект mutable, то его адрес в памяти и, соотвтетственно, идентификатор не изменятся, и наоборот. При попытке изменить иммутабельный объект будет создан новый со своим уникальным идентификатором.
Важным моментом является нерекурсивность свойства мутабельности. Объект, являющийся частью иммутабельной коллекции или класса, не обязан быть иммутабельным: так, например, список может быть одним из элементов кортежа.

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Хэширование

Хэширование — процесс кодирования различных объектов в строку фиксированной длины, необходимый, преимущественно, для быстрого сравнения объектов на равенство. Хэширование производится с помощью отведённых под это хэш-функций, целью которых является обеспечение быстрого преобразования данных в хэш, который практически гарантированно уникален (хорошая хэш-функция будет иметь минимальное число "коллизий").
Хэширование и мутабельность связаны: только immutable объекты, все вложенные объекты которых также являются immutable (рекурсивно), могут быть хэшированы, поскольку иначе может случиться ситуация, когда хэш-сумму объекта уже использовали, затем изменили что-либо внутри него, а после снова пытаются посчитать хэш и произвести некую операцию, как будто этот тот же объект. Реализации хэш-функций таковы, что с большой вероятностью будет получено совершенно другое значение хэш-суммы, и не получится понять, что объект тот же, что и в первом случае хэширования.

Вместо статической типизации

Python — интерпретируемый и динамически типизируемый язык. В связи с этим, типы данных явным образом могут нигде не указываться. Однако, хорошим тоном и полезной практикой является аннотация входных и выходных типов функций:

```
from typing import List, ClassVar
def to_array(lst: List[int], data_type: ClassVar) -> np.ndarray:
    return np.array(lst, dtype=data_type)
```

С помощью таких явных указаний типов мы не запретим функции принимать данные других типов, поскольку строгой статической типизации нет, но сделаем код более прозрачным и позволим среде разработки дополнять код подсказками гораздо эффективнее.

Система классов

Каждый класс в Python унаследован от общего предка — object, а явное наследование кастомных классов осуществляется простым указанием класса-предка в скобках. Также возможно множественное наследование. За выбор реализации каждого метода при вызовах отвечает система MRO — method resolution order — которая основана на алгоритме линеаризации C3.

Классы в Python, помимо обычных методов, могут реализовывать так называемые magic методы. Практически всегда необходимо описывать инициализацию объектов класса — за это отвечает метод `__init__`. Каждый не статический метод класса должен содержать аргумент, который принято называть `self` — он будет указывать на объект класса, с которым ведется работа.

Менеджер контекста и обработка исключений

В Python очень много фич, сделанных для удобства и автоматического контроля за исполнением кода. Одним из примеров является конструкция `with ... as:` — с её помощью можно создавать объект, который нужен только в определённой области выполнения программы, а затем автоматически осуществлять все необходимые операции для его успешного завершения/закрытия/очистки. Работает это на основе magic методов `__enter__` и `__exit__`, которые, соответственно, вызываются при входе и при выходе из блока with.

Для обработки исключений используется блок `try: ... catch: ... finally: ...`. Примечательно, что код, написанный под finally, будет выполняться в любом случае — произошла ошибка или нет, в том числе в случае, когда внутри try или catch был осуществлен возврат значения функции.

Работа с файлами

Классическим примером использования менеджера контекста для обеспечения безошибочной корректной работы является открытие файлов. По завершении исполнения кода внутри with или по выкинутому исключению будет произведено закрытие файла, что избавляет нас от необходимости обрабатывать различные случаи и прописывать закрытие файла вручную. Помимо этого, такой механизм позволяет чётко визуально отображать область видимости открытого файла

```
with open('input.txt', 'r') as f:
    print(f.readlines())
```

Режимы чтения файлов все стандартые: текстовые чтение, запись и добавление ('r', 'w', 'a'), а также бинарные варианты этих режимов ('rb', 'wb', 'ab').

NumPy

NumPy — крайне широко используемая библиотека для обработки массивов данных любоых типа и размерности. Отличается большим количеством методов манипуляции, преобразования и представления структурированных данных.

Формат хранения

Прежде всего, введём общепринятый псевдоним для библиотеки — np. Обычно она импортируется целиком: `import numpy as np`.

Массивы numpy — объекты типа `np.ndarray`. Они имеют множество полей и методов, и с точки зрения структуры самое удобное и важное среди них — shape. Это кортеж, описывающий размерность массива и длину каждой из размерностей.

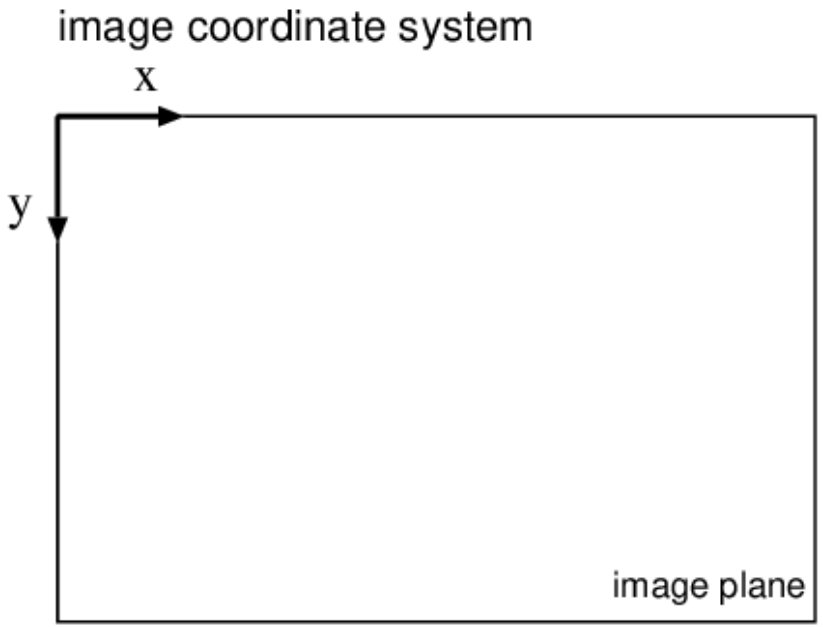
Под капотом numpy хранение данных реализованы через одномерный массив, в рамках которого, в зависимости от shape, происходит n-мерная адресация. Формат хранения C-подобный — наименее быстро изменяемая размерность индексируется в первую очередь. Для практического понимания такой тип хранения в случае двумерных массивов — матриц — называют **построчным**, то есть при первом индексировании матрицы возвращается **строка**, а затем по второму индексу — элемент строки.

Поскольку при мультииндексировании (`array[i][j]`) каждый раз возвращаются копии масива на 1 размерность меньше, гораздо эффективнее адресовать один раз с точностью до элемента необходимой размерности (`array[i, j]`). То же самое и со слайсами: например, чтобы получить столбец, а не строку, достаточно сделать `array[:, j]`.

NumPy: работа с картинками

При любой обработке данных, в том числе изображений, важно знать их формат и возможности манипуляции. В случае представления изображений с помощью массивов numpy, из-за построчного хранения, индексирование первой размерности отвечает за выделение строк, а второй — за выделение столбцов. Для многоканальных изображений еще есть третья размерность, которая отвечает за выбор канала (например, в наиболее частом случае, каналы R, G и B).

У координатной сетки изображения есть важная особенность — за начало координат принят левый верхний угол. Это значит, что ось Y инвертирована и направлена сверху вниз.
Обычно двумерные координаты указываются в виде упорядоченных пар (x, y) — необходимо это учитывать при адресации пикселей в массивах numpy и первым индексом указывать координату Y.



Типы данных

Для массивов numpy обычно используют типы данных из этой же библиотеки. Основной тип для хранения и визуализации изображений — np.uint8 — 8-битное беззнаковое целое, то есть целые числа от 0 до 255 включительно. При обработке различными функциями бывает полезно (или, даже, необходимо) приводить такие матрицы к вещественному типу (`np.float32` , `np.float64`) или более обширному целому (`np.int16` , `np.int32`). Преобразование делается с помощью переприсваивания результата метода `arr.astype(data_type)` .

Помимо этого, numpy позволяет легко создавать некоторые шаблонные массивы любого типа и размера. Например, есть возможность создать забитую нулями (`np.zeros`) или единицами (`np.ones`) матрицу, единичную (`np.eye`) и так далее.