САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ – ПРОЦЕССОВ УПРАВЛЕНИЯ

Мишенин А.Н., Стученков А.Б.

КЛАССИФИКАЦИЯ ТЕКСТОВ

Часть 1:

Программирование на Python. NumPy

Учебно-методическое пособие

Санкт-Петербург
2020

УДК 025.4.03 : 004.738.5
ББК 32.971.353 : 32.973

А.Н. Мишенин, А.Б. Стученков
Классификация текстов. Часть 1: Программирование на Python. NumPy.
Учебно-методическое пособие – СПб., 2020. – 69с.

Учебно-методическое пособие разработано на основе опыта чтения курсов лекций и проведения практических занятий по учебным дисциплинам «Классификация документов» и «Информационный поиск» на факультете Прикладной Математики – Процессов Управления Санкт-Петербургского государственного университета. В первой части внимание уделено изучению основ языка Python и библиотеки NumPy.

Содержание

Введение

`Python`[2][1] - скриптовый язык программирования общего назначения.

Основные особенности:

- строгая динамическая типизация
- автоматическое управление памятью

Преимущества:

- интерпретируемость, нет необходимости компилировать код (с другой стороны, это можно воспринимать как недостаток)
- развитая стандартная библиотека
- большое количество сторонних библиотек
- простота, отлично подходит для быстрого прототипирования

Недостатки:

- проблемы с производительностью в определенных сценариях (`CPython`)
- проблемы с статическим анализом кода, невозможно создать вменяему IDE
- исторические проблемы, связанные с дизайном языка

Исторически существует две версии `Python 2` и `Python 3`. Они не совместимы и фактически являются разными языками программирования, такой переход был связан с попыткой избавиться от исторических проблем в дизайне языка (поддержка юникода, неконсистентность некоторых синтаксических конструкций). Переход между версиями был сопряжен с определенными проблемами, так

как не все библиотеки были портированы на третью версию. Но в настоящий момент проблема, в целом, решена. Мы будем использовать `Python 3`.

```
# Статическая типизация
int a = 5;
a = "hello";

# Динамическая типизация
a = 5
a = "hello" # Ok
```

```
# Строгая типизация
a = "10"
b = a + 5 #ошибка

# Слабая типизация
a = "10"
a = a + 5 # ???? 15? "105"?
```

Python для научных вычислений и машинного обучения

Язык общего назначения, но с помощью сторонних библиотек (Scientific Python) можно использовать качестве альтернативы для MATLAB, Mathematica, R и в качестве языка для построения моделей глубокого обучения. Приведем некоторый список популярных библиотек:

- **NumPy**[5] - линейная алгебра, случайные числа, базовые операции
- **scipy**[11] - статистика, численные методы и многое другое
- **tensorflow**[14]- машинное обучение, автоматическое дифференцирование, нейронные сети

- **pytorch**[9] - машинное обучение, автоматическое дифференцирование, нейронные сети
- **matplotlib**[3] - рисование графиков, визуализация
- **plotly**[7]- рисование графиков, визуализация
- **pandas**[6] - работа с табличными данными в стиле R
- **sympy**[13] - символьные вычисления (Maple)
- **scikit-learn**[10] - машинное обучение
- **statsmodels**[12] - статистика, машинное обучение
- **PyMC3**[8] - байесовское статистическое моделирование
- **networkx**[4] - работа с графами, визуализация, раскладка

Преимущества:

- свободное программное обеспечение
- огромный выбор библиотек
- можно создавать production-ready системы
- низкоуровневые библиотеки написаны на `C` (производительность)
- можно легко писать C-расширения (`cython`)
- параллельные вычисления

Недостатки:

- по функциональности может уступать специализированным языкам программирования (например, **R**)
- врожденные ограничения интерпретатора `CPython` (отсутствие `JIT`, `GIL` и т.д.)

Проверка версии:

```
$ python ——version
```

Реализации Python

CPython

- эталонная реализация
- интерпретатор, с проблемами в дизайне (GIL)

PyPy

- JIT
- в общем случае быстрее
- не совместим со многими библиотеками, использующими нативные библиотеки. Практически бесполезно для научных вычислений.

Jython

- Реализация под `JVM`. Исходный код компилируется в Java Byte Code

IronPython

- Реализация под `CLR`. Исходный код компилируется в `CIL`

GraalPython

- Реализация для `GraalVM`. На начальном этапе разработки (2020)

Установка Python

Linux: Можно использовать системный пакетный менеджер (`apt`, `yum`). Для Debian-based дестрибутиво установка всего необходимого может выглядеть так

```
$ sudo apt-get install python3-dev python3-pip
    cython3 python3-numpy python3-scipy
```

Windows, Linux, Mac OS: Удобно использовать специальный дистрибутив с собственным менеджером пакетов Anaconda

Docker: Если установлен `Docker`, то можно официальный репозиторий https://hub.docker.com/_/python/ с самыми последними версиями.

Скрипты Python

Скрипты хранятся в файлах с расширением *.py

В начале каждого файла рекомендуется добавлять шапку:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# тут код
```

- на первой строчке находится т.н. shebang, в Unix-системах указание оболочке, какую программу нужно использовать для запуска скрипта
- на второй строчке указание интерпретатору Python на кодировку исходного текста скрипта. Разумеется всегда нужно использовать `utf-8`

Запуск скрипта:

```
$ python python_script.py
```

Jupyter notebook

Проект `Jupyter` - это интегрированная среда для разработки, которая эволюционировала из обычного `REPL` (Read Eval Print Loop). Её можно установить с помощью пакетного менеджера `conda`:

```
$ conda install —c conda—forge jupyterlab
```

или `pip`:

```
$ pip install jupyterlab
```

Запуск из командной строки:

```
$ jupyterlab
```

Зачем это нужно?

- своего рода web-based IDE
- “альтернативный” подход, как в Mathematica
- удобно для экспериментов
- богатая инфраструктура, программы, взаимодействие с `R`, `Julia` и так далее
- блокноты хранятся в специальном формате *.ipynb

Концепция ячеек:

- в каждой ячейке может быть текст (документация) или код
- ячейку можно запустить и сразу получить результат
- в каждой ячейке нужно писать какую-то малую логическую часть кода
- состояние можно сохранить

Горячие клавиши:

- Ctrl+Enter - запустить ячейку
- Ctrl+Shift - запустить ячейку и переместиться вниз
- Esc+m - текстовая ячейка
- Esc+y - ячейка с кодом
- Ctrl+m+b - вставить новую ячейку за текущей ячейкой
- Ctrl+m+a - вставить новую ячейку перед текущей ячейкой
- Ctrl+m+x - удалить текущую ячейку

<tab> - автодополнение кода

Основы программирования на Python

Система модулей

`Python` имеет систему пакетов и модулей, модуль подключается с помощью директивы `import`. Модуль (module) - базовое понятие языка. Содержит код и глобальные переменные. После подключения модуля в текущем пространстве имен становятся доступны все определенные в модуле функции, классы и переменные.

```
1 import random
2 random.randint(1,5)
```

```
5
```

Дополнительный синтаксис, если мы хотим импортировать только определенные функции:

```
# импортирует из модуля функции randint и gauss
from random import randint, gauss

gauss(0, 4)
```

```
-1.5369677758312392
```

Можно так:

```
from random import *
import math as m

m.log(2)
```

```
0.6931471805599453
```

В среде `Jupyter` можно бысло получить справку по функциям:

```
randint?
```

или с помощью встроенной функции **`help`**

```
help(m.log)
```

В результате получится

```
Help on built-in function log in module math:

log(...)
    log(x[, base])

    Return the logarithm of x to the given base.
    If the base not specified, returns the natural
        logarithm (base e) of x.
```

Переменные

Имена переменных в Python могут содержать буквы, цифры и _, начинаться должны с буквы (_ в начале может имеет семантическую нагрузку). По соглашению, имена переменных начинаются с маленькой буквы, имена классов - с большой.

Ключевые слова:

```
and, as, assert, break, class, continue, def, del, elif,
else, except,  exec, finally, for, from, global, if,
import, in, is, lambda, not, or, pass, print, raise,
return, try, while, with, yield
```

Простые базовые типы данных

Целые числа в Python поддерживают длинную арифметику.

```
1  a = 2 # int
2  type(a)
```

```
int
```

Числа с плавающей точкой

```
1  a = 2.5
2  type(a)
```

```
float
```

Комплексный числа

```
a = 2 + 5j
b = 3j
type(a)
```

```
complex
```

Логический тип

```
a = True
b = False
type(a)
```

```
bool
```

Тип `NoneType`, который имеет единственное значение - `None`.

```
a = None
type(a)
```

```
NoneType
```

Преобразование типов

Преобразовать один тип к другому можно с помошью конструкторов

```
int(True), int(False)
```

```
(1, 0)
```

```
float(2)
```

```
2.0
```

Математические операции

Нужно иметь ввиду, что в результате операции деления целый / может получиться число с плавающей точкой. Для целочисленного деления используется оператор //.

```
2 / 3, 2 // 3, 10 ** 106
```

```
(0.6666666666666666,
 0,
 100000000000000000....)
```

В Python 3 тип целых чисел с длинной арифметикой и обычных целых совпадает

```
type(2 ** 160)
```

```
int
```

```
a = False
not a
```

```
True
```

```
a = 4
a < 5, 1 < a < 7, a <= 4, 2 == 4
```

```
(True, True, True, False)
```

Строки

Строки в Python 3 поддерживают unicode. Могут обозначаться двойными кавычками, одинарными, тремя одинарными или тремя двойными.

```
1  s = "first string"
2  s, type(s)
```

```
('first string', str)
```

Основные методы

```
1  v: int  = s.find('<str>') # поиск первого вхождения
2  v: list = s.split('<str>') # разделить на строки
3  v: str  = s.islower() # s.isupper(), s.isdigit(), ..
4  v: str  = s.replace('<str>', '<str>')
5
6  # удалить пробельные символы по краям
7  v: str  = s.strip()
8  v: int  = s.count('<str>') # число вхождний подстроки
9  v: bool = s.startswith('<str>') # s.endswith('<str>')
10 v: str  = s.join(['<str>', '<str>', '<str>'])
```

slicing - получение подстроки

```
1  s = 'Hello world'
2  s[:-1]
```

```
'Hello worl'
```

Индексирование

```
s = 'Hello World!'
len(s), s[0], s[-1]
```

```
(12, 'H', '!')
```

Конкатенация подстрок

```
b = s[0:3] + s[-2:-1]
b
```

```
'Held'
```

Более сложные примеры, реверс строки, получение подстроки через несколько символов

```
s, s[::-1], s[1:-1:2]
```

```
('Hello World!', '!dlroW olleH', 'el ol')
```

Если между двоеточиями подразумевается ноль, то этот ноль, как в примере выше, можно опустить

```
s[:0:-1]
```

```
'!dlroW olle'
```

Строку можно получить из любого типа с помощью встроенной функции `str`

```
str(5)
```

```
'5'
```

Форматирование строк

Традиционный способ с помощью оператора `%`

```
'%05d %05.3f %s' % (5, 3.123456, 'hello')
'%04d %.2f' % (5, 2.578)
'%05d' % 120
```

```
'00005 3.123 hello'
'0005 2.58'
'00120'
```

Можно использовать метод `format`

```
'{0:.2f} {1:05d} {2}'.format(0.5, 5, 'hello')
```

```
'0.50 00005 hello'
```

или более удобный способ - специальный тип строки с префиксом `f`

```
k = 5
v = 10
f'k={k}, v={v}'
```

```
'k=5, v=10'
```

Списки

Реализация структуры данных “список” - упорядоченных набор элементов произвольной природы. Python имеет специальный синтаксис для работы со списками, операции индексирования как со строками, но списки можно изменять

Создадим список

```
[1, 'aaa', 2]
```

```
[1, 'aaa', 2]
```

Длина, первый и последний элементы списка

```
lst = [1, 2, 3]
len(lst), lst[0], lst[-1]
```

```
(3, 1, 3)
```

Основные методы

```
el = 3
col = [1, 2, 3]

lst.append(el)    # добавление элемента в конец
lst.extend(col)   # добавления коллекции в конец
lst.remove(el)    # удалить первое вхождение
lst.pop()         # удалить последний элемент
lst.count(el)      # число элементов в списке
lst.index(el)      # индекс первого элемент
lst.insert(1, el) # вставить *el* на позицию *1*
lst.clear()

lst += lst
lst += [el]
```

Метод append добавляет элемент к концу списка

```
lst = [1, 2, 3]

lst.append(4)
lst
```

```
[1, 2, 3, 4]
```

списки можно конкатенировать с помощью операции +

```
lst2 = [4, 5, 6]
lst3 = lst + lst2
lst3
```

```
[1, 2, 3, 4, 4, 5, 6]
```

вытащить последний элемент

```
lst3.pop()
```

```
6
```

вставить элемент на конкретную позицию

```
lst3.insert(1, 'a')
lst3
```

```
[1, 'a', 2, 3, 4, 4, 5]
```

удаление элемента по значению

```
lst = [1, 3, 2, 2, 4]
lst.remove(2)
lst
```

```
[1, 3, 2, 4]
```

удаление по индексу

```
del lst[2]
lst
```

```
[1, 3, 4]
```

Список из упорядоченных чисел можно получить используя встроенную функцию `range`

```
list(range(0, 5))
```

```
[0, 1, 2, 3, 4]
```

Кортежи

Кортежи похожи на списки, но:

- их нельзя изменять
- они предназначены для хранения структурированных данных

Кортежи создаются похожим способом, только вместо квадратных скобок используются круглые

```
t = (1, 2, 3)
t
```

```
(1, 2, 3)
```

основные методы

```
el = 2

t.count(el)     # число элементов в списке
t.index(el)     # индекс первого элемент
```

```
1
```

кортежи нельзя модифицировать, соответственно они поддерживают вычисление хеш-значения

```
t[0] = 5
```

```
TypeError   Traceback (most recent call last)

<ipython-input-47-6dd06f73cec4> in <module>
----> 1 t[0] = 5


TypeError: 'tuple' object does not support item
    assignment
```

```
a, b, c = t
a
```

```
1
```

подходят для хранения структурированных данных

```
t = ('Ivanov', 'Ivan', 'Ivanovich', 25, 'M')
t
```

```
('Ivanov', 'Ivan', 'Ivanovich', 25, 'M')
```

Множества

Структура данных “множество” - неупорядоченного набора уникальных элементов. В реализации используется хеш-таблица.

Создание множества

```
s = {1, 2, 3}
s = set([1, 2, 3])
s
```

```
{1, 2, 3}
```

“Множества” поддерживают базовые теоретико-множественные операции. Например объединение

```
l = {1, 2, 3}
l.union([5]) # l + set([5])

l
```

```
{1, 2, 3, 5}
```

И другие

```
el = 5

s = {1, 2, 3}
s = set()
s = set([1, 2, 3])

s.add(el)
s.discard(el)  # удалить элемент
s.pop()        # удалить произвольный элемент
s.remove(2)   # удалить заданный элемент
s.update({1, 2, 3})
s.clear()
r: set  = s.difference({1, 2, 3})
r: set  = s.union({1, 2, 3})
r: set  = s.intersection({1, 2, 3})
r: set  = s.issubset({1, 2, 3})
r: bool = s.issuperset({1, 2, 3})

s |= {1, 2, 3}
s &= {1, 2, 3}
```

Проверить, входит ли элемент в множество можно с помощью оператора `in`

```
2 in s
```

```
True
```

его можно использовать в виде `not in`

```
5 not in s
```

```
True
```

для добавления одиночного элемента используется метод `add`

```
s.add(5)
s
```

```
{1, 2, 3, 5}
```

для теоретико-множественных операций можно использовать –, +, '|', '&'

```
{ 1 } | {2, 3, 5} – {5}
{1, 2} & {2, 5}
```

```
{1, 2, 3}
{2}
```

удаление элемента из множества осуществляется с помощью метода `remove`

```
s = {1, 2, 3}
s.remove(3)
```

Словари

Словарь, `dict`, это реализация ассоциативного массива, то есть словарь позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу. Используется хеш-таблица, как и в случае с **`set`**.

Основные методы

```
d = dict(zip([1, 2, 3], ['1', '2', '3']))
d = dict.fromkeys([1, 2, 3])
d = {1: 'a', 2: '3', 3: '4' }

el = 3

d.items()       # итератор пар (ключ, значение)
d.keys()        # итератор ключей
d.values()      # итератор ключей
d.items()       # итератор для пар (ключ, значение)
d.pop(el)       # удалить ключ, возвратить значение
d.get(el)       # значение по ключу

# значение по ключу, по умолчанию - 5
d.get(el, 5)
d.values()      # список значений

# добавить все пары ключ/значение из *d*
d.update({5: '7', 9: '10'})
d.clear()
```

положить пару ключ-значение в словарь можно с помощью операции индексации `[...]`

```
d = dict() # a =
d['hello'] = 2
d[5] = 'a'
d
```

```
{'hello': 2, 5: 'a'}
```

словарь можно создать из списка

```
d = dict([(1, 5), ('t', 7)])
d = {1: 5, 't': 7}
d
```

```
{1: 5, 't': 7}
```

если ключа нет в словаре, то будет ошибка - выбросится исключение

```
d[8]
```

```
---------------------------------------------------------------------------
KeyError
Traceback (most recent call last)

<ipython-input-62-4ef738dbabd0> in <module>
----> 1 d[8]


KeyError: 8
```

Для того, чтобы подобное избежать, нужно использовать метод `get`

```
d.get(8, -1)
```

```
-1
```

Можно получить набор всех ключей и значений

```
d.keys(), d.values()
```

```
(dict_keys([1, 't']), dict_values([5, 7]))
```

Управляющие конструкции

Базовый цикл в Python - **for**, он применяется для обхода коллекций

```
s = 0
for i in [1, 2, 3]:
    s += i
print(s)
```

```
6
```

Если нужны индексы, можно использовать встроенную функцию enumerate

```
list(enumerate(['a', 'b', 'c']))
```

```
[(0, 'a'), (1, 'b'), (2, 'c')]
```

в цикле

```
lst = ['a', 'b', 'c']
result = []
for (i, x) in enumerate(lst):
    result.append((x, i))

result
```

```
[('a', 0), ('b', 1), ('c', 2)]
```

Цикл `while`

```
i = 0
lst = []
while len(lst) < 6:
    lst.append(i)
    i += 1
lst
```

```
[0, 1, 2, 3, 4, 5]
```

Оператор `if`

```
a = 2
if a < 5:
    a = 1
elif a == 5:
    a = 2
else:
    a = 3
```

List comprehension (списковое включение)

Синтаксический сахар, для быстрых манипуляций со списками

```
l = [1, 2, 3]
l = [x * x for x in l]
l
```

```
[1, 4, 9]
```

Например, можно создать новый список, который содержит квадраты только четных элементов

```
[x * x for x in range(1, 10) if x % 2 == 0]
```

```
[4, 16, 36, 64]
```

или список состоящий из списков

```
[list(range(x)) for x in range(1, 10) if x % 2 == 1]
```

```
[[0],
 [0, 1, 2],
 [0, 1, 2, 3, 4],
 [0, 1, 2, 3, 4, 5, 6],
 [0, 1, 2, 3, 4, 5, 6, 7, 8]]
```

Можно комбинировать итерировании по нескольким коллекциям

```
[(x, y) for x in range(1, 20) for y in range(1, 20)
    if x * y == 16]
```

```
[(1, 16), (2, 8), (4, 4), (8, 2), (16, 1)]
```

Аналогично устроена работа с множествами

```
{ x * x : x for x in range(5) }
```

и словарями

```
{0: 0, 1: 1, 4: 2, 9: 3, 16: 4}
```

Пример применения для обработки слов в предложении:

```
text = ['Hello', 'world', '!']
text_filtered = [t.lower() for t in text if t.isalpha
    ()]
```

Функции

Функции в Python можно определить с помощью ключевого слова def или lambda.

```
def add(a, b):
    return a + b

div = lambda a, b : a / b

add(2, 3), add('2', '3'), div(4, 2), div(b=2, a=2)
```

```
(5, '23', 2.0, 1.0)
```

Python поддерживает функции и неопределенным числом аргументов, при этом аргументы могут быть позиционными и именованными.

```
def foo(a, *args, **kwargs):
    print(a)
    print(args, type(args))
    print(kwargs, type(kwargs))

foo(1, 7, 10, c=10)
```

```
1
(7, 10) <class 'tuple'>
{c: 10} <class 'dict'>
```

Функции в Python - функции высшего порядка, можно использовать в качестве аргументов других функций и возвращать из других функций.

```
def foo(f, a, b):
    return f(a, b)

foo(add, 5, 7)
```

```
12
```

Здесь функция возвращает функцию

```
def appl(f, a):
    return lambda x: f(a, x)

add_2 = appl(add, 2)
add_2(10)
```

```
12
```

В Python есть большой набор встроенных функций, в частности для сортировки:

```
lst = [3, 2, 5, 1, 7]
sorted(lst)
```

```
[1, 2, 3, 5, 7]
```

Альтернативно можно использовать метод sort у списка.

```
lst.sort()
lst
```

```
[1, 2, 3, 5, 7]
```

Можно передать функцию, которая определяет отношения порядка между элементами списка

```
lst = [(5, 'the'), (2, 'sun'), (2, 'shines'),
       (7, 'brightly')]
lst.sort(key=lambda x: -x[0])
lst
```

```
[(7, 'brightly'), (5, 'the'), (2, 'sun'), (2, 'shines')]
```

Итераторы и генераторы

Итератор - объект, предоставляющий доступ к элементам коллекции и навигацию по ним. В Python есть концепции итерируемого объекта , итератора и генератора. В общих слова:

- итератор - специальный объект, который можно получить из коллекции с помощью вызова встроенной функции `iter`. В дальнейшем следующие элементы выдаются после вызова `next()`.
- итерируемый объект - объект, по которому с помощью вызова `iter()` можно создать итератор
- генератор - частный случай итератора, который создается с помощью функций специального вида

```
it = iter([1, 2, 3])
next(it), next(it), next(it)
```

```
(1, 2, 3)
```

В цикле `for` итератор создается неявным образом, главное чтобы в качестве аргумента был итерируемый объект. Следующий код

```
def action(x):
    print(x)

lst = [1, 2, 3]

for x in lst:
    action(x)
```

```
1
2
3
```

приблизительно эквивалентен

```
it = iter(lst)
while True:
    try:
        value = next(it)
    except StopIteration:
        break
    action(value)
```

```
1
2
3
```

“Функция-генератор” - это специальная функция, которая содержит хотя бы один оператор `yield` . Вызов функции интерпретируется как создание итератора, в момент вызова `yield` выполнение приостанавливается.

```
def gen(val):
    for i in range(3):
        yield '%s #%d' % (val, i)
        print('After yield %d' % i)

for val in gen('From yield'):
    print(val)

it = gen('From yield once again')
next(it), next(it)
```

```
From yield #0
After yield 0
From yield #1
After yield 1
From yield #2
After yield 2
After yield 0
```

```
('From yield once again #0', 'From yield once again #1')
```

В качестве альтернативы можно использовать list comprehension, только с круглыми скобками, в этом случае так же создается генератор:

```
gen = (x for x in range(1, 100) if x % 2 == 0)
next(gen), next(gen)
```

```
(2, 4)
```

Классы

Для создания пользовательских классов можно использовать простой синтаксис, можно заметить:

- явно передается ссылка на объект
- специализированные методы начинаются и заканчиваются двумя подчеркиваниями. Это одна из черт общей философии языка Python. В данном примере __str__ применяется для создания строкового представления, а __repr__ используется для отладки и результат должен отображать внутреннее состояние экземпляра класса.

```
class Person:
    def __init__(self, name, age, children=[]):
        self.name = name
        self.age = age
        self.children = children

    def add_child(self, child):
        self.children.append(child)

    def __str__(self):
        return self.name

    def __repr__(self):
        return f'Person("{self.name}", {self.age}, {
            repr(self.children)})'
```

Создадим экземпляр класса (оператора new в Python нет)

```
person = Person('Ivan Ivanov', 35)
print('{}\n{}\n{}'.format(repr(person), str(person),
    person))
```

```
Person("Ivan Ivanov", 35, [])
Ivan Ivanov
Ivan Ivanov
```

Экземпляр класса в Python очень упрощенно представляет собой сло-

варь - ассоциативный массив из полей и методов

```
vars(person)
```

```
{'name': 'Ivan Ivanov', 'age': 35, 'children': []}
```

```
dir(person)
```

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'add_child',
 'age',
 'children',
 'name']
```

для классов, предназначенных только для хранения данных, можно использовать `namedtuple` или `@dataclass`

```
from collections import namedtuple
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int

Person = namedtuple('Person', ['name', 'age'])
person = Person(name='Ivanov', age=20)
person, person.name, person.age
```

```
(Person(name='Ivanov', age=20), 'Ivanov', 20)
```

```
class FibSequence:
    def __init__(self, start0, start1):
        self.start0 = start0
        self.start1 = start1

    def __iter__(self):
        yield self.start0
        yield self.start1
        while True:
            self.start0, self.start1  = self.start1,
                self.start0 + self.start1
            yield self.start1
```

```
fib_seq = FibSequence(1, 2)

for i, num in enumerate(fib_seq):
    print(i, num)
    if i >= 5:
        break
```

```
0 1
1 2
2 3
3 5
4 8
5 13
```

Задачи

Задача 1. Свертка списка - это обобщенная операция над списком, с помощью которой можно преобразовать список в единое значение. Например, рассмотрим реализации свертки слева и свертки справа (левоассоциативную свертку и правоассоциативную свертку):

```
1  def foldl(f, x0, lst):
2      if not lst:
3          return x0
4      return foldl(f, f(x0, lst[0]), lst[1:])
5
6  def foldr(f, x0, lst):
7      if not lst:
8          return x0
9      return f(lst[0], foldr(f, x0, lst[1:]))
```

Задача: реализовать foldl через foldr и наоборот. Вместо многоточий нужно вставить выражения, которые бы привели к нужному результату.

```
1  def foldl2(f, x0, lst):
2      return foldr(..., ..., lst)(...)
3
4  def foldr2(f, x0, lst):
5      return foldl(..., ..., lst)(...)
```

Задача 2. Нужно написать функцию, которая принимает две строки

и проверяет, входит ли хотя бы одна перестановка второй строки в первую. Например:

> a = ‘abcrotm’
>
> b = ‘tro’

функция def check_inv(a, b) вернет True, так как ‘rot’ содержится в ‘abcrotm’. Нужно подумать как можно более оптимальный алгоритм и оценить его сложность.

```
def check_inv(a, b):
    pass
```

Избранные конструкции и библиотеки

Декораторы

Функции в Python - функции высшего порядка. Декораторы - это специальные функции, которые служат для временного изменения поведения других функций. Они принимают в качестве параметра старую функцию и возвращают новую. Для них определен синтаксический сахар:

При создании декораторов лучше использовать стандартный декоратор из стандартной библиотеки

```
@functools.wraps
```

это позволяет сохранять мета информацию для декорируемой функции

Напишем декоратор, который изменяет поведение функций, возвращающий целое число - прибавляет к результату единицу

```
import functools

def plus_one(old_func):
    @functools.wraps(old_func)
    def wrapper(*args, **kwargs):
        return old_func(*args, **kwargs) + 1
    return wrapper

@plus_one
def my_func():
    """Docstring"""
    return 41

my_func()
```

```
42
```

```
print(my_func.__name__, my_func.__doc__)
```

```
my_func Docstring
```

Контекстные менеджеры

Контекстные менеджеры позволяют контролировать выделение и освобождение ресурсов с помощью специального оператора `with`. Например для закрытие файлового дескриптора, освобождения блокировки, закрытия соединения с базой данных и т.д.

Пример с файлом:

```
with open('file.txt', 'w', encoding='utf-8') as f:
    f.write('Hello\n')
```

в заголовке оператора `with` создается объект файлового дескриптора, который автоматически закрывается при выходе из тела оператора.

Контекстный менеджер можно реализовать самостоятельно. Следующий код

```
class MyContextMng(object):
    def __init__(self, s):
        self.s = s
        pass

    def __enter__(self):
        print('Enter')
        return self.s

    def __exit__(self, exception_type, exception_val,
         trace):
        print('Exit')
        return True


with MyContextMng('Hello') as f:
    #actions
    pass
```

```
Enter
Exit
```

приблизительное преобразуется в следующее

```
tmp = MyContextMng('Hello')
f = tmp.__enter__()
try:
    actions
    pass
except:
    pass
finally:
    tmp.__exit__(None, None, None)
```

```
Enter
Exit
```

Написание контекстных менеджеров можно упростить с помощью декоратора `@contextmanager` и генераторов

```
from contextlib import contextmanager

@contextmanager
def my_context(s):
    print('Enter')
    yield s
    print('Exit')

with my_context('Hello') as f:
    #actions
    pass
```

```
Enter
Exit
```

Операции с файлами

чтение файла

```
with open('data/real_estate.csv', 'r', encoding='utf
    -8') as f:
    print(f.readline().strip())
    for c, line in enumerate(f):
        pass
```

```
КомнатРайон
#,, городаАдресЭтажОбщЖилКухМТелПримечанияЦена,,,,,,,,,
252
```

список файлов в директории

```
import os

os.listdir('data/')
```

```
['faces.npy',
 'man.png',
 'real_estate.csv',
 'svm_example_circle.npz',
 'svm_example_moons.npz',
 'texts.zip',
 'transport_log.zip',
 'weather.csv',
 'wiki.xml']
```

временные файлы

```
import tempfile

with tempfile.NamedTemporaryFile() as fn:
    with open(fn.name, 'w', encoding='utf-8') as f:
        f.write('Hello world!\n')

    with open(fn.name, 'r', encoding='utf-8') as f:
        print(f.readlines())
```

```
['Hello world!\n']
```

архивы

```
import zipfile
import io

with zipfile.ZipFile('data/texts.zip', 'r') as zf:
    with zf.open('texts.txt', 'r') as f:
        f_unicode = io.TextIOWrapper(f, 'utf-8')
        print(f_unicode.readline()[:50])
        print(f.readline().decode('utf-8')[:50])
```

```
0    «Школазлословия »учитприкуситьязыкСохранитсют
: приключенческаяканваопираласьнаотличноезн
```

Коллекции и итераторы

модуль itertools

итерирование одновременно по нескольким коллекциям

```
list(zip(['а', 'б', 'в'], ['a', 'b', 'c']))
list(enumerate(['a', 'b', 'c']))
list(zip(['a', 'b', 'c'], [1, 2, 3]))
```

```
a
[('', 'a'), б('', 'b'), в('', 'c')]
[(0, 'a'), (1, 'b'), (2, 'c')]
[('a', 1), ('b', 2), ('c', 3)]
```

конкатенация нескольких коллекций

```
import itertools

ch = itertools.chain(g(), [1, 2, 3], {3, 5, 6}, iter
    ([1, 2]))
list(ch)
```

```
[1, 2, 1, 2, 3, 3, 5, 6, 1, 2]
```

Созданием словаря из спика пар:

```
lst = ['a', 'b', 'c']
dict(enumerate(lst))
```

```
{0: 'a', 1: 'b', 2: 'c'}
```

модуль functools позволяет применять некоторые концепции из функционального программирования, кэшировать результат выполнения функции и так далее

```
from functools import partial

add = lambda x, y: x + y
foo = partial(add, y=5)
foo(10)
```

```
15
```

Кэшировать результат функции (LRU-кэш) можно с помощью специального декоратора

```
from functools import lru_cache
from time import sleep

@lru_cache(maxsize=5)
def heavy_stateless_computations(param):
    sleep(5)
    return param ** 2

%time heavy_stateless_computations(20)
%time heavy_stateless_computations(20)
```

```
CPU times: user 1.98 ms, sys: 0 ns, total: 1.98 ms
Wall time: 5 s
CPU times: user 16 µs, sys: 0 ns, total: 16 µs
Wall time: 26 µs
```

```
400
```

модуль collections содержит утилиты для работы с коллекциями.

`defaultdict` позволяет создать словарь, в котором для каждого отсутствующего ключа при первом обращении создается значение по умолчанию.

```
from collections import defaultdict

d = defaultdict(lambda: []) # можно просто
    defaultdict(list)
d['word1'].append(1)
d['word1'].append(2)
d['word2'].append(3)
d
```

```
defaultdict(<function __main__.<lambda>>, {'word1': [1,
    2], 'word2': [3]})
```

`Counter` позволяет быстро организовать подсчет элементов

```
from collections import Counter

c = Counter()
c['word1'] += 1
c['word2'] += 2
c.update({'word1': 5, 'word3': 4})
c
```

```
Counter({'word1': 6, 'word2': 2, 'word3': 4})
```

В дальнейшем можно получить список наиболее популярных (самых частотных) элементов в коллекции.

```
c.most_common(2)
```

```
[('word1', 6), ('word3', 4)]
```

Сериализация

Сериализация - представление структуры данных в бинарном виде, для дальнейшего сохранения этой структуры или передачи (например по сети). В `Python` существует стандартный механизм сериализации - модуль `pickle`. Так же возможно сохранение базовых объектов в формате `json`.

В данном примере сначала создаются временные файлы, куда записывается представление объектов в бинарном виде и в `json`. Затем это представление восстанавливается.

```python
import tempfile
import pickle
import json

obj = {'hello' : [1, 2, 3], 'world': [4, 5, 6]}

with tempfile.NamedTemporaryFile() as fn:
    with open(fn.name, 'wb') as f:
        pickle.dump(obj, f)
    print(pickle.load(open(fn.name, 'rb')))

with tempfile.NamedTemporaryFile() as fn:
    with open(fn.name, 'w') as f:
        json.dump(obj, f)
    print(json.load(open(fn.name, 'r')))
```

```
{'hello': [1, 2, 3], 'world': [4, 5, 6]}
{'hello': [1, 2, 3], 'world': [4, 5, 6]}
```

Задачи

Задача 1. Реализовать бинарное дерево (класс Tree), в нём методы **repr**, **str**, **iter** (итерация только по листьям).

```
class Tree:
    def __init__(self, value=None, left=None, right=
        None):
        self.left = left
        self.right = right
        self.value = value

    def __iter__(self):
        pass

    def __str__(self):
        pass

    def __repr__(self):
        pass

tree = Tree(0, Tree(1, Tree(3), Tree(4)),
               Tree(2))

list(tree) == [3, 4, 2]
```

Задача 2. Реализовать простейший калькулятор математических выражений: - только целые числа - умножение, сложеник, вычитание, деление - скобки

```
def calc(expr):
    pass

calc('2 * (15 — 3 * 4) — 2') == 4
```

NumPy

NumPy используется как основа почти для всех математических библиотек в Python. В пакете реализоавны и хорошо оптимизированы базовые операции над векторами, матрицами и т.п. (BLAS). Основные функции NumPy реализованы на языке C.

Стандартная процедура импортирования модуля NumPy. Встречается почти в каждом скрипте.

```
import numpy as np
```

Создание массивов

Основная концепция в NumPy - типизированный многомерный массив, весь остальной функцинал завязан вокруг этого. Существует много способов создания массива.

Например из обычного списка

```
a  = np.array([[1.5, 4.5], [1, 2]])
```

```
a.astype('int64')
```

```
array([[1, 4],
       [1, 2]])
```

```
ar = np.array([1, 2.6])
```

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

```
array([[1, 2],
       [3, 4]])
```

Размерность массива можно узнать с помощью функции > np.shape()

или свойства .shape > ar.shape

```
print(np.shape(ar), ar.shape)
```

```
(2, 2) (2, 2)
```

Массив можно создать с помощью специальных функций.

Одномерный массив из нулей

```
np.zeros(5)
```

```
array([0., 0., 0., 0., 0.])
```

Двумерный массив (матрица) из нулей

```
np.zeros((3, 3), dtype='int64')
```

```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Двумерный массив из единиц

```
np.ones((3, 3))
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Массив из чисел от 1 включительно до 10 через 2

```
np.arange(1, 10, 2)
```

```
array([1, 3, 5, 7, 9])
```

Массив из чисел на отрезке от 1 до 10 состоящий и 5 элементов на равных промежутках

```
np.linspace(1, 10, 5)
```

```
array([ 1.  ,  3.25,  5.5 ,  7.75, 10.  ])
```

Диагональная матрица с указанными элементами

```
np.diag([1, 2, 3])
```

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Единичная матрица

```
np.identity(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Массивы `NumPy` типизированы, тип массива выводится во время создания, но его можно указать явно:

```
ar = np.array([1, 2, 3], dtype=np.double)
ar
```

```
array([1., 2., 3.])
```

Для получения типа массива можно использовать свойство `dtype`

```
ar.dtype
```

Задать тип массива можно с помощью строки

```
dtype('float64')
```

Физически массивы NumPy - это массив байт, размерность хранится отдельно и её можно поменять с помощью метода (общий размер не должен меняться)

> .reshape

Изменим размер матрицы 2×3 на 3×2. При этом данные буфера общие для двух массивов.

```
a = np.array([[1, 2, 3], [3, 4, 5]])
b = a.reshape((3, 2))
a[1, 1] = 7
b
```

```
array([[1, 2],
       [3, 3],
       [7, 5]])
```

```
a[1, 1]
```

```
7
```

Изменим размерность массива из чисел от 0 до 15

```
1  a = np.arange(0, 16)
2  b = a.reshape((2, 8))
3  print('a =', a)
4  print('b =', b)
```

```
a = [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
b = [[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]]
```

При этом видим, что у них общий буфер. Если мы изменим один массив, то изменится и другой

```
1  b[1, 0] = 18
2  print('a =', a)
3  print('b =', b)
```

```
a = [ 0  1  2  3  4  5  6  7 18  9 10 11 12 13 14 15]
b = [[ 0  1  2  3  4  5  6  7]
 [18  9 10 11 12 13 14 15]]
```

Для изменения общего размера массива применяется метод `.resize` . При этом может производиться копирование данных, эту операцию следует использовать с осторожностью

```
1  a = np.arange(0, 16)
2  a.resize((3, 8))
3
4  print(a)
```

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [ 0  0  0  0  0  0  0  0]]
```

Индексация массивов

К элементам массива можно обращаться так же, как мы это делаем для списков или строк Python

```
1  a = np.arange(0, 16).reshape((4, 4))
2  print(a)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

```
1  print(a[0, 2], a[0][2])
```

```
2 2
```

Slicing

Slicing - это получение “среза” массива или под массива. В случае матрицы операцию можно интерпретировать как получение под матрицы.

В данном случае берем первые две строки и 1,2-й столбец.

```
1  a = np.arange(0, 16).reshape((4, 4))
2  print(a[0:2,1:3])
```

```
[[1 2]
 [5 6]]
```

Можно оставить только третью строку, в этом случае получим одномерный массив

```
print(a[2, :])
```

```
[ 8  9 10 11]
```

можно взять строки и столбцы в обратном порядке

```
print(a[0:2, ::-1])
```

```
[[3 2 1 0]
 [7 6 5 4]]
```

Slicing можно использовать для изменения массивов, например изменяем все элементы второго столбцы на 1.

```
a[:, 1] = 1
print(a)
```

```
[[ 0  1  2  3]
 [ 4  1  6  7]
 [ 8  1 10 11]
 [12  1 14 15]]
```

Или первые два элемента первой строки

```
a[0, :2] = np.ones(2)
print(a)
```

```
[[ 1  1  2  3]
 [ 4  1  6  7]
 [ 8  1 10 11]
 [12  1 14 15]]
```

Advanced indexing

Если в качестве индексатора использовать массивы одинакового размера, то создастся подмасив из соответствующих элементов

```
1  a = np.arange(16).reshape((4, 4))
2  print(a)
3
4  rows = [1, 3]
5  columns = [2, 2]
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

```
1  # получаем массив [ a[row[0], column[0]], a[row[1],
       column[1]] ]
2  print(a[rows, columns])
```

```
[ 6 14]
```

Или даже так, в этом случае получаем матрица из некоторых элементов исходного массива

```
1  print(a[ [[0, 0], [1, 1]], [[2, 2], [3, 3]] ])
```

```
[[2 2]
 [7 7]]
```

Можно смешивать способы индексации

```
print(a[[0, 1], :2])

print(a[2:, [0, 1]])
```

```
[[0 1]
 [4 5]]
[[ 8  9]
 [12 13]]
```

Можно использовать маски из логических значений:

Тут на получаем новый массив из булевских значений, всем элементам строго больше 3 соответствует `True`.

```
print(a > 3)
```

```
[[False False False False]
 [ True  True  True  True]
 [ True  True  True  True]
 [ True  True  True  True]]
```

Можно получить одномерный массив из элементов удовлетворяющих условию

```
print(a[a > 3])
```

```
[ 4 5 6 7 8 9 10 11 12 13 14 15]
```

Получение подматрицы, где остаются только те строки, сумма элементов которых строго больше 8

```
a[a.sum(axis=1) > 8]
```

```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Базовые операции

Массивы имеют обширный набор методов для совершения тех или иных операций над своими элементами. Например для суммирования, нахождение максимума или минимума, нахождения среднего значения и мное другое. Характерная особенность таких операций состоит в том, что они могут применяться к элементам по определенным осям (“вдоль оси”).

Например, в данном примере происходит суммирование всех элементов матрицы, суммировании строк (вдоль оси “0”) и столбцов (вдоль оси “1”).

```
print(ar.sum(), ar.sum(axis=0), ar.sum(axis=1), sep='
    ;')
```

```
40.0;[ 4.  8. 12. 16.];[10. 10. 10. 10.]
```

Матрицы можно развернуть в одномерный массив

```
ar.ravel()
```

```
array([1., 2., 3., 4., 1., 2., 3., 4., 1., 2., 3., 4.,
    1., 2., 3., 4.])
```

Два массива можно соединить вдоль определенной оси. При этом обязательно, чтобы вдоль этой оси размерность была одинаковой. В примере одна матрица присоединяется к другой “сверху” или “вдоль оси 0”

```
a = np.arange(0, 5).reshape((1, 5))
b = np.arange(5, 10).reshape((1, 5))
np.concatenate((a, b), axis=0)
```

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

Аналогично “вдоль оси 1”

```
np.concatenate((a, b), axis=1)
```

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

Для наглядности, если нужно соединить сверху или вдоль, то применяются методы `vstack` и `hstack`

```
np.vstack((a, b))
```

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

```
np.hstack((a, b))
```

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

Линейная алгебра

Для массивов NumPy переопределены все базовые математические операции, при этом они выполняются поэлементно, если размерности совпадают и с помощью специальных правил (“broadcasting”) в противном случае

```
ar = np.ones((4, 4)) * np.array([[1, 2, 3, 4]])
ar
```

```
array([[1., 2., 3., 4.],
       [1., 2., 3., 4.],
       [1., 2., 3., 4.],
       [1., 2., 3., 4.]])
```

```
mat = np.ones((5, 5))
mat + mat
```

```
array([[2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.]])
```

Если размерности не совпадают, то операции совершаются рекурсивно над подмассивом соответствующей размерности вдоль возможной старшей оси.

Ко всем элементам матрицы можно прибавить число

```
mat + 2
```

```
array([[3., 3., 3., 3., 3.],
       [3., 3., 3., 3., 3.],
       [3., 3., 3., 3., 3.],
       [3., 3., 3., 3., 3.],
       [3., 3., 3., 3., 3.]])
```

Ко всем строкам можно прибавить строку

```
mat + np.array([1, 2, 3, 4, 5])
```

```
array([[2., 3., 4., 5., 6.],
       [2., 3., 4., 5., 6.],
       [2., 3., 4., 5., 6.],
       [2., 3., 4., 5., 6.],
       [2., 3., 4., 5., 6.]])
```

Ко всем столбцам - столбец

```
mat + np.array([[1], [2], [3], [4], [5]])
```

```
array([[2., 2., 2., 2., 2.],
       [3., 3., 3., 3., 3.],
       [4., 4., 4., 4., 4.],
       [5., 5., 5., 5., 5.],
       [6., 6., 6., 6., 6.]])
```

Ну или просто сложить две матрицы

```
mat + np.ones(5)
```

```
array([[2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.]])
```

Аналогично с операцией умножения (важно понимать, что это не матричное умножение), деления и так далее

```
5 * mat
```

```
array([[5., 5., 5., 5., 5.],
       [5., 5., 5., 5., 5.],
       [5., 5., 5., 5., 5.],
       [5., 5., 5., 5., 5.],
       [5., 5., 5., 5., 5.]])
```

Синус над всеми элементами матрицы

```
np.sin(mat)
```

```
array([[0.841, 0.841, 0.841, 0.841, 0.841],
       [0.841, 0.841, 0.841, 0.841, 0.841],
       [0.841, 0.841, 0.841, 0.841, 0.841],
       [0.841, 0.841, 0.841, 0.841, 0.841],
       [0.841, 0.841, 0.841, 0.841, 0.841]])
```

Транспонирование матрицы

```
mat = np.arange(16).reshape(4, 4)
#транспонирование
mat.T, mat.transpose()
```

Матричное умножение

```
np.dot(mat, mat)
mat @ mat
mat.dot(mat)
```

Определитель матрицы

```
np.linalg.det(mat)
```

Обратная матрица

```
np.linalg.inv(np.array([[1, 0], [0, 1]]))
```

Псевдообратная матрица

```
np.linalg.pinv(mat)
```

Собственные числа

```
np.linalg.eigvals(mat)
```

Собственные вектора

```
np.linalg.eig(mat)
```

SVD разложение

```
np.linalg.svd(mat)
```

Для матричного умножения используется оператор `@` или метод `dot`

```
mat = np.arange(16).reshape(4, 4)
mat @ mat
mat.dot(mat)
np.dot(mat, mat)
```

```
array([[ 56,  62,  68,  74],
       [152, 174, 196, 218],
       [248, 286, 324, 362],
       [344, 398, 452, 506]])
```

Функции

Важно понимать, что для массивов, помимо умножения, деления и пр., нужно использовать лишь специализированные версии операций из пакета NumPy. Например, в данном примере при попытке использовать операцию cos из стандартной библиотеки Python произойдет ошибка

```
1  def foo(x):
2      return x * np.cos(x) - np.sin(x)
3
4  foo(np.array([1, 2, 7]))
```

```
array([-0.30116868, -1.7415911 ,  4.62032918])
```

С помощью декоратора vectorize можно получить специализированную версию функции, которая будет применяться ко всем элементам массива NumPy поэлементно. Важно понимать, что часто подобные операции неэффективны и их следует применять лишь в исключительных случаях

```
1  @np.vectorize
2  def foo(x):
3      if x > 5:
4          return 1
5      return 0
6
7  foo(np.array([1, 2, 7]))
```

```
array([0, 0, 1])
```

Сохранение состояния

Массивы NumPy могут быть сохранены в бинарном виде

```
import tempfile

with tempfile.NamedTemporaryFile(suffix='.npy') as fn
    :
    np.save(fn.name, np.array([1, 2, 3]))
    a = np.load(fn.name)
    print(a)
```

```
[1 2 3]
```

Случайные числа

NumPy содержит большое количество различных математических функций. Остановимся на модуле numpy.random в котором содержится большое количество методов для получения массивов из псевдослучайных чисел.

Получение случайной матрицы 5×5 из целых чисел на полуинтервале от 0 до 5.

```
import numpy.random as rnd

rnd.randint(0, 5, (5, 5))
```

```
array([[4, 4, 1, 0, 3],
       [1, 0, 0, 4, 0],
       [4, 2, 4, 2, 1],
       [1, 2, 4, 4, 1],
       [1, 1, 4, 1, 2]])
```

Сэмплинг матрицы размерностью 2×2 из нормального распределения, с математическим ожиданием 5 и среднеквадратичным отклонением 1.

```
1  rnd.normal(5., 1, size=(2, 2))
```

```
array([[6.21217647, 4.17759224],
       [5.40441408, 5.44658326]])
```

Семплинг из распределения Дирихле.

```
1  rnd.dirichlet(alpha=[2, 3])
```

```
array([0.47996603, 0.52003397])
```

Задачи

NB. Все упражнения ниже нужно делать без использования циклов Python

Задача 1. Посчитать

$$(2, \frac{2^2}{2}, \dots, \frac{2^{20}}{20})$$

Задача 2. Посчитать:

$$\sum_{i=0}^{5} 0.1^{3i} 0.2^{4i}$$

Задача 3.

Создать нулевую матрицу 8×8, и заполнить её единицами в шахматном порядке.

Список литературы

1. Pilgrim M. , Willison S. . Dive Into Python 3 : т. Т. 2. Springer, 2009. с.

2. Van Rossum G. , Drake F. L. . Python 3 Reference Manual : т. Scotts Valley, CA: CreateSpace, 2009. с.

3. matplotlib [Электронный ресурс] / (01.03.2020). Режим доступа: https://pypi.python.org/pypi/matplotlib/.

4. networkx [Электронный ресурс] / (01.03.2020). Режим доступа: http://pymc-devs.github.io/pymc.

5. NumPy [Электронный ресурс] / (01.03.2020). Режим доступа: https://numpy.org/.

6. Pandas [Электронный ресурс] / (01.03.2020). Режим доступа: https://pypi.python.org/pypi/pandas.

7. Plotly [Электронный ресурс] / (01.03.2020). Режим доступа: https://plot.ly.

8. PyMC3 [Электронный ресурс] / (01.03.2020). Режим доступа: http://pymc-devs.github.io/pymc.

9. PyTorch [Электронный ресурс] / (01.03.2020). Режим доступа: https://pytorch.org/.

10. scikit-learn [Электронный ресурс] / (01.03.2020). Режим доступа: http://scikit-learn.org.

11. SciPy [Электронный ресурс] / (01.03.2020). Режим доступа: https://scipy.org/.

12. statsmodels [Электронный ресурс] / (01.03.2020). Режим доступа: http://statsmodels.org.

13. SymPy [Электронный ресурс] / (01.03.2020). Режим доступа: http://www.sympy.org/en/index.html.

14. Tensorflow [Электронный ресурс] / (01.03.2020). Режим доступа: https://tensorflow.org.